LeetCode数据结构定义

```
普通链表：
class Node {
    int val;
    Node next;
    public Node(int val) {
        this.val = val;
        this.next = null;
    }
}
其他链表：（每个节点加入random指针）
class Node {
    int val;
    Node next, random;
    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
    }
}
栈：
    Stack<Character> stack = new Stack<Character>();
```

常用集合类型API

```
LinkedList
    size()
    get(index)  检索
    getFirst/Last()
    indexOf(object) 返回第一次出现元素的位置
    LastIndexOf()
    set(index,element)  替换
    offer() 在列表尾部添加元素E，返回布尔型
    add()   添加元素
 栈：
    addLast()    入栈
    removeLast()     出栈
    pop()   弹出一个元素，栈顶
    push()  添加一个元素

 列表：
    offer(e)    在列表尾部添加元素
    offerFirst/Last(e)
    peek(e)     检索列表头，但不删除
    peekFirst/Last(e)
    pop()   检索栈顶的一个元素，并删除
    pollLast/First()    检索尾/头第一元素


HashMap
    containsKey()   查找元素是否存在
    put(k,v)    添加元素
    getOrDefault(num,0)+1   没有数就返回指定默认值，可以计数
        map.put(num,map.getOrDefault(num,0)+1);
    keySet      返回key的集合。可以在foreach循环中遍历
        for(Map.Entry<Integer,Integer> entry : map.entrySet())
int[]
    length
 创建方式：
    int[] nums = new int[2]{};
    int[] nums1 = {1,2,3,4};
    int[] nums2 = new int[]{1,3,4,5};


ArrayList
    add()
    get(index)
    size()
    contains()


HashSet
    add(e)  添加唯一的
    remove(o)   存在则删除    返回true
    contains(o)     判断存在
    size()
```

```
    clone() 实例副本，复制引用
```

常用API

```
array：数组     基于索引的数据结构，检索，读取快，删除慢
list：列表      list是个链表，检索较慢

list不需要指定初始大小，array一般要指定大小
list不是一段连续的存储结构，array存储结构连续

Array转换为List    Array.asList
List转换为Array    List.toArray

List转arr
arr = list.stream.mapToInt(Integer::valueOf).toArray(); ArrayList<Integer> 转 int[]
arr = list.toArray(new Integer[0]);

Object[] array = arraylist.toArray();
String[] array = arraylist.toArray(new String[arraylist.size()]);
/
int[] res = new int[list2.size()];
        for (int i = 0; i < list2.size(); i++) {
            res[i] = list2.get(i);
        }

int[] res = new int[list2.size()];
        int i = 0;
        for (int num : list2) {
            res[i++] = num;
        }

Math
    min()   返回两个比较小的那个
    max()

Array
<String/int/char>[]
    length

String
    length()
    charAt(index)   返回检索的值   类型为char
    compareTo(String)       比较两个字符串 返回int
    comcat(String)      在末尾添加字符串String
    contains(String)    判断是否存在       boolean
    subString(begin,end)    拆分字符串，从给定起始位置到终止位置，截取下来
    split(",")      按照设定的来划分，
    toCharArray     转换为字符数组
    startsWith(prefix)      判断前缀是否相等
    equals("")
    indexof()   返回字符串或字符第一次出现的索引
        for(char c : word1.toCharArray()){
            if(word2.indexOf(c) != -1) return true;
        }//判断是否有相同的字符
    valueOf(数组) 以字符串的形式

StirngBuilder
    append()    追加到字符串上，拼接


Arrays
    sort    排序
    parallelSort    数字顺序排序
    toString    返回字符串
    copyof      复制



Integer
    parseInt(String)    将数字字符串转换为数字
    MAX_VALUE       2 的 31 次方 - 1 = 2147483648 - 1 = 2147483647
    MIN_VALUE       Integer.MAX_VALUE + 1 = Integer.MIN_VALUE = -2147483648


Character
    isLowerCase(char)   判断是否为小写
    isUpperCase(char)   判断


System.arrayc
    public static native void arraycopy(Object src,  int  srcPos,
                                        Object dest, int destPos,
                                        int length);
   //src表示源数组，srcPos表示源数组要复制的起始位置，desc表示目标数组，length表示要复制的长度。
```

笔记

1　两数之和
- 抛出非法参数异常```throws IllegalArgumentException```
- hash查找效率远大于两次遍历
- 注意结尾return 空值
- map.containsKey方法可以查找相等的key

2　有序数组下的两数之和
- 二分查找，容易写错，划分边界细节太复杂，在一个while中，不断分两份
- 双指针：通过前后两个指针来求和比较，大于目标值，就右指针左移，小于，就左指针右移

3　只出现一次的数
- 可以使用一个哈希映射统计数组中每一个元素出现的次数。value = map.getOrDefault(num,0)+1
- 异或运算可以筛选出出现次数为单次的数，通过一个数将数组分为两组，然后分别异或就可以得到出现次数为单数的的两个数

4　用两个栈实现队列
- 创建一个栈对象可以用：LinkedList<Integer> stack1;
- linkedList　add添加元素到栈，pop弹出栈顶元素

5　包含min函数的栈
- 手写一个node链表，会很方便
- 创建一个栈对象 Stack，peek返回栈顶对象

6　从头到尾打印链表
- push入栈，pop出栈　可用数据类型：LinkedList（addLast入栈）（removeLast出栈）
- 辅助栈法：创建一个栈然后，入栈添加元素，再for转换为数组返回
- 递归法：创建一个方法，在里面不停调用自身，用全局ArrayList去接受值，再for转为数组
- 递归法先深入，再赋值　　digui(head.next);　　tmp.add(head.val);　注意判断链表是否为空

7　键盘行
- 通过String的contains方法判断单个word里的单个单词是否存在，已经设定的三个键盘行
- 然后通过计数，存在第一行就加一，若单行存在的次数等于word的字符长度，就确定这个单词满足条件
- foreach将String[]，拆分为单个word，再for(i)循环检索每个单词

8　复杂链表的复制
- 方法一：复制+拼接+拆分，该题的难点就在：　在复制链表的过程中构建新链表各节点的 random 引用指向。
- 方法一原理很简单，先复制节点在每个节点后面插入一个克隆节点，在构建新节点的random指针，再将新链表拆分为二
- 复制链表最好有head节点，在while循环中，更改指针来插入克隆节点。
- 构建新节点的random指针：　cur.next.random = cur.random.next;记得在while循环体最后一句移动当前节点
- 拆分链表：cur.next = cur.next.next;处理好源节点的尾结点，返回复制出来的链表
- 方法二：哈希表。先用hashmap装节点，<Node,CloneNode>，再构建新链表的指针（next，random）。
- 两个操作都在while循环中执行，map.put(cur, new Node(cur.val));
- map.get(cur).next = map.get(cur.next);map.get(cur).random = map.get(cur.random);

9　三数之和
- 三数之和，方法：排序+双指针。建立二维数组的方式　List<List<Integer>> ans = new ArrayList<>();
- 方法一需要排除一些额外情况，比如数组值为空或个数小于3，还有数值相等，数值太大的情况
- 在for循环中，用while来移动双指针，判断是否等于num[i]的负数。
- 判断确定，还要再深入while循环来移动指针寻找其他满足条件的情况。来避免指针遇到连续数值相同的情况
- 二维数组添加的方式：ans.add(new ArrayList<>(Arrays.asList(nums[i], nums[left], nums[right])));
- 若两个指针大于num[i]的负数，右指针左移，反之右移

10　四数之和
- 跟上面的三数和相同，就是需要在for循环中再添加一个for循环，while循环体基本不变，也是排序+双指针

11　分糖果
- 要求唯一，使用HashSet结构，一堆糖果分出有多少类型

12　斐波那契数
- 除了暴力递归，还有很多方法，比如，动态规划，矩阵快速幂，通项公式

13　爬楼梯
- 和12题一样，递归，动态规划，矩阵快速幂，通项公式。各种方法。最简单的动态规划问题

14　替换空格
- Java 等语言中，字符串都被设计成「不可变」的类型，即无法直接修改字符串的某一位字符
- 需要新建一个字符串实现。
- 方法一：遍历拼接
- 初始化一个 list (Python) / StringBuilder (Java) ，记为 res ；
- 遍历列表 s 中的每个字符 c ：
- 当 c 为空格时：向 res 后添加字符串 "%20" ；
- 当 c 不为空格时：向 res 后添加字符 c ；
- 将列表 res 转化为字符串并返回。

15　删除俩表中的节点
- 如何让自己在世界上消失，但又不死？ —— 将自己完全变成另一个人，再杀了那个人就行了。

算法

动态规划

题：17，18，19，23

找到具有最优值的解，用于求解具有某种最优性质的问题
动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。

1. 简单递归
2. 备忘录算法：`map或者字典表`，以空间换时间，记忆化搜索，递归树的剪枝，带备忘录的递归
3. 迭代：在爬楼梯和斐波那契数都是。记录前两次计算的状态，来得到下一次计算的结果
    1. 即：F(n) = F(n - 1) + F(n - 2)（这就是状态转移方程式）　　条件：F(1)和F(2)已知（这就是边界条件）

三个核心元素：最优子结构，边界，状态转移方程式

记忆化搜索

题：19

DFS深度优先算法

题：31

深度优先搜索在搜索过程中访问某个顶点后，需要递归地访问此顶点的所有未访问过的相邻顶点。

它沿着树的深度遍历树的节点，尽可能深的搜索树的分 支。当节点v的所有边都已被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发 现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。DFS属于盲目搜索。

深度优先遍历图算法步骤：

1. 访问顶点v；
2. 依次从v的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和v有路径相通的顶点都被访问；
3. 若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

BFS广度优先算法

题：30

贪心算法

NP完全问题

缩写为NPC

NP完全问题(NP-C问题)，是世界七大数学难题之一。 NP的英文全称是Non-deterministic Polynomial的问题，即多项式复杂程度的非确定性问题。简单的写法是NP=P？，问题就在这个问号上，到底是NP等于P，还是NP不等于P。

P类问题：所有可以在多项式时间内求解的判定问题构成P类问题。

NP类问题：所有的非确定性多项式时间可解的判定问题构成NP类问题。

NPC问题：**NP中的某些问题的复杂性与整个类的复杂性相关联.这些问题中任何一个如果存在多项式时间的算法,那么所有NP问题都是多项式时间可解的.这些问题被称为NP-完全问题(NPC问题)。

二分法

```
//经典二分  旋转数组最小的数
class Solution {
    public int minArray(int[] numbers) {
        int low = 0;
        int high = numbers.length - 1;
        while (low < high) {
            int pivot = low + (high - low) / 2;
            if (numbers[pivot] < numbers[high]) {
                high = pivot;
            } else if (numbers[pivot] > numbers[high]) {
                low = pivot + 1;
            } else {
                high -= 1;
            }
        }
        return numbers[low];
    }
}
```

分类

字符串

字典表 28 29

```
//创建'a'-'z'的字典
        int[] target = new int[26];
//遍历，将字符统计到字典数组
        for (int i = 0; i < s.length(); i++) {
            target[s.charAt(i) - 'a']++;
        }
```

1、两数之和

①

找出两数的和为目标数

暴力法 时间复杂度：O(n^2)　空间复杂度：O(1)

```
class Solution {
    public int[] twoSum(int[] nums, int target) throws IllegalArgumentException {
        for(int i =0;i < nums.length;i++){
            for(int j = i+1;j <nums.length;j++ ){
                if(nums[j] == target - nums[i]){
                    return new int[]{i,j};
                }
            }
        }
        return new int[]{};
    }
}
```

hash查找法 时间复杂度：O(n)　空间复杂度：O(n)

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int [] indexs = new int[2];
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        for(int i =0;i<nums.length;i++){
            if(map.containsKey(nums[i])){
                indexs[0] = i;
                indexs[1] = map.get(nums[i]);
                return indexs;
            }
            map.put(target-nums[i],i);
        }
        return null;
    }
}
```

2、有序数组下的两数之和

①⑥⑦

方法一：**二分查找** 时间复杂度：O(n \log n)] 空间复杂度O(1)

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        for (int i = 0; i < numbers.length; ++i) {
            int low = i + 1, high = numbers.length - 1;
            while (low <= high) {
                int mid = (high - low) / 2 + low;
                if (numbers[mid] == target - numbers[i]) {
                    return new int[]{i + 1, mid + 1};
                } else if (numbers[mid] > target - numbers[i]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            }
        }
        return new int[]{-1, -1};
    }
}
```

方法二：**双指针**时间复杂度：O(n) 空间复杂度：O(1)

初始时两个指针分别指向第一个元素位置和最后一个元素的位置。每次计算两个指针指向的两个元素之和，并和目标值比较。如果两个元素之和等于目标值，则发现了唯一解。如果两个元素之和小于目标值，则将左侧指针右移一位。如果两个元素之和大于目标值，则将右侧指针左移一位。移动指针之后，重复上述操作，直到找到答案。

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        for(int i = 0,j = numbers.length-1;i < j;){
            int sum = numbers[i]+numbers[j];
            if(target == sum)
            return new int[]{i+1,j+1};
            else if(sum > target) j--;
            else i++;
        }
        return null;
    }
}
```

3、只出现一次的数

②⑥

hash法

我们可以使用一个哈希映射统计数组中每一个元素出现的次数。

在统计完成后，我们对哈希映射进行遍历，将所有只出现了一次的数放入答案中。

时间复杂度：O(n)

空间复杂度：O(n)

```
class Solution {
    public int[] singleNumber(int[] nums) {
        Map<Integer,Integer> map = new HashMap<Integer,Integer>();
        for(int num : nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        int [] arrs =new int[2];
        int index = 0;
        for(Map.Entry<Integer,Integer> entry : map.entrySet()){
            if(entry.getValue() == 1)
            arrs[index++] = entry.getKey();
        }
        return arrs;
    }
}
```

位运算法 时间复杂度：O(n)，空间复杂度：O(1)

```
class Solution {
    public int[] singleNumber(int[] nums) {
        int xorsum = 0;
        for (int num : nums) {
            xorsum ^= num;
        }
        // 防止溢出
        int lsb = (xorsum == Integer.MIN_VALUE ? xorsum : xorsum & (-xorsum));
        int type1 = 0, type2 = 0;
        for (int num : nums) {
            if ((num & lsb) != 0) {
                type1 ^= num;
            } else {
                type2 ^= num;
            }
        }
        return new int[]{type1, type2};
    }
}
//上面是官方写法，下面自写法
//首先将数组异或出来
class Solution {
    public int[] singleNumber(int[] nums) {
        //首先得到异或的值xor
        int xor = 0;
        for(int num : nums){
            xor ^= num;
        }
        //得到xor中二进制（从右到左的第一个为1的值），将他们分为两组
        int div = 1;
        while ((xor & div) == 0){
```

```
            div =div << 1;
        }
        //分为两组后，对数组与运算，1为一组，0,为一组。然后再异或得到的就是我们想要的两个值
        int a = 0;
        int b = 0;
        for (int num1 : nums){
            if((num1 & div) ==0 ){
                a ^= num1;
            }else {
                b ^= num1;
            }
        }
        return new int[]{a,b};
    }
}
```

3.1数组中重复的数字

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

```
//自己写的
class Solution {
    public int findRepeatNumber(int[] nums) {
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        for(int num : nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        for(Map.Entry<Integer,Integer> entry :map.entrySet()){
            if(entry.getValue() > 1){
                return entry.getKey();
            }
        }
        return 0;
    }
}
//官方题解HashSet
class Solution {
    public int findRepeatNumber(int[] nums) {
        Set<Integer> set = new HashSet<Integer>();
        int repeat = -1;
        for (int num : nums) {
            if (!set.add(num)) {
                repeat = num;
                break;
            }
        }
        return repeat;
    }
}
class Solution {
    public int findRepeatNumber(int[] nums) {
        Set<Integer> dic = new HashSet<>();
        for(int num : nums) {
            if(dic.contains(num)) return num;
            dic.add(num);
        }
        return -1;
    }
}
//排序
//原地置换（效率最高）
class Solution {
    public int findRepeatNumber(int[] nums) {
        int i = 0;
        while(i < nums.length) {
            if(nums[i] == i) {
                i++;
                continue;
            }
            if(nums[nums[i]] == nums[i]) return nums[i];
            int tmp = nums[i];
            nums[i] = nums[tmp];
            nums[tmp] = tmp;
        }
        return -1;
    }
}
```

3.2统计一个数字在排序数组中出现的次数

```
class Solution {
    public int search(int[] nums, int target) {
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        for(int num : nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        if(map.containsKey(target)){
            return map.get(target);
        }
        return 0;
    }
}
//二分查找
class Solution {
    public int search(int[] nums, int target) {
        int leftIdx = binarySearch(nums, target, true);
        int rightIdx = binarySearch(nums, target, false) - 1;
        if (leftIdx <= rightIdx && rightIdx < nums.length && nums[leftIdx] == target && nums[rightIdx] == target) {
            return rightIdx - leftIdx + 1;
        }
        return 0;
    }

    public int binarySearch(int[] nums, int target, boolean lower) {
        int left = 0, right = nums.length - 1, ans = nums.length;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] > target || (lower && nums[mid] >= target)) {
                right = mid - 1;
                ans = mid;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}
作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/solution/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-wl6kr/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

4、用两个栈实现队列

通过两个栈，实现队列功能，从尾部插入，从头部删除

```
class CQueue {

    LinkedList<Integer> stack1;
    LinkedList<Integer> stack2;

    public CQueue() {
            stack1 = new LinkedList<>();
            stack2 = new LinkedList<>();
    }

    public void appendTail(int value) {
        stack1.add(value);
    }

    public int deleteHead() {
        if(stack2.isEmpty()){
            if(stack1.isEmpty()) return -1;
            while(!stack1.isEmpty()){
                stack2.add(stack1.pop());
            }
            return stack2.pop();
        }else return stack2.pop();
    }
}
```

5、包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

```
class MinStack {
    Stack<Integer> A, B;
    public MinStack() {
        A = new Stack<>();
        B = new Stack<>();
    }
    public void push(int x) {
        A.add(x);
        if(B.empty() || B.peek() >= x)
            B.add(x);
    }
    public void pop() {
        if(A.pop().equals(B.peek()))
            B.pop();
    }
    public int top() {
        return A.peek();
    }
    public int min() {
        return B.peek();
    }
}
```

6、从尾到头打印链表

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

辅助栈法:

push入栈，pop出栈 可用数据类型：LinkedList（addLast入栈）（removeLast出栈）

```
class Solution {
    public int[] reversePrint(ListNode head) {
        LinkedList<Integer> stack = new LinkedList<Integer>();
        while(head != null){
            stack.addLast(head.val);
            head = head.next;
        }
        int[] arrs = new int[stack.size()];
        for(int i = 0;i<arrs.length;i++){
            arrs[i] = stack.removeLast();
        }
        return arrs;
    }
}
```

7、键盘行

给你一个字符串数组 words ，只返回可以使用在 美式键盘 同一行的字母打印出来的单词。键盘如下图所示。

美式键盘 中：

第一行由字符 "qwertyuiop" 组成。
第二行由字符 "asdfghjkl" 组成。
第三行由字符 "zxcvbnm" 组成。

示例 1：

```
输入：words = ["Hello","Alaska","Dad","Peace"]
输出：["Alaska","Dad"]
```

```
class Solution {
    public String[] findWords(String[] words) {
        String s1 = "qwertyuiopQWERTYUIOP";
        String s2 = "asdfghjklASDFGHJKL";
        String s3 = "zxcvbnmZXCVBNM";
        List<String> list = new ArrayList<>();
        for(String word :words){
            int n1 = 0,n2 = 0,n3 = 0,leng = word.length();
            for(int i = 0;i < leng;i++){
                if(s1.contains(word.charAt(i)+"")) n1++;
                else if(s2.contains(word.charAt(i)+"")) n2++;
                else n3++;
            }
            if(n1==leng || n2==leng || n3==leng) list.add(word);
        }
```

```
        return list.toArray(new String[list.size()]);
    }
}
```

8、复杂链表的复制

我题都看不懂，【剑指offer35题】

拼接+拆分法

```
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null){
            return head;
        }
        //完成链表节点复制
        Node cur = head;
        while(cur != null){
            Node copyNode = new Node(cur.val);
            copyNode.next = cur.next;
            cur.next = copyNode;
            cur = cur.next.next;
        }

        //完成链表复制节点的随机指针复制
        cur = head;
        while(cur != null){
            if(cur.random != null){
                cur.next.random = cur.random.next;
            }
            cur = cur.next.next;
        }
        // 将链表一分为二
        Node copyHead = head.next;
        cur = head;
        Node curCopy = head.next;
        while (cur != null) {
            cur.next = cur.next.next;
            cur = cur.next;
            if (curCopy.next != null) {
                curCopy.next = curCopy.next.next;
                curCopy = curCopy.next;
            }
        }
        return copyHead;
    }
}
```

hash表法

```
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null) return null;
        Node cur = head;
        Map<Node, Node> map = new HashMap<>();
        // 3. 复制各节点，并建立 “原节点 -> 新节点” 的 Map 映射
        while(cur != null) {
            map.put(cur, new Node(cur.val));
            cur = cur.next;
        }
        cur = head;
        // 4. 构建新链表的 next 和 random 指向
        while(cur != null) {
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
            cur = cur.next;
        }
        // 5. 返回新链表的头节点
        return map.get(head);
    }
}

作者：jyd
链接：https://leetcode-cn.com/problems/fu-za-lian-biao-de-fu-zhi-lcof/solution/jian-zhi-offer-35-fu-za-lian-biao-de-fu-zhi-ha-xi-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

9、三数之和为0

排序+双指针

找到三个数和为0，但数不相同的值

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {// 总时间复杂度：O(n^2)
        List<List<Integer>> ans = new ArrayList<>();
        if (nums == null || nums.length <= 2) return ans;

        Arrays.sort(nums); // O(nlogn)

        for (int i = 0; i < nums.length - 2; i++) { // O(n^2)
            if (nums[i] > 0) break; // 第一个数大于 0，后面的数都比它大，肯定不成立了
            if (i > 0 && nums[i] == nums[i - 1]) continue; // 去掉重复情况
            int target = -nums[i];
            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                if (nums[left] + nums[right] == target) {
                    ans.add(new ArrayList<>(Arrays.asList(nums[i], nums[left], nums[right])));

                    // 现在要增加 left，减小 right，但是不能重复，比如: [-2, -1, -1, -1, 3, 3, 3], i = 0, left = 1, right = 6, [-2,
-1, 3] 的答案加入后，需要排除重复的 -1 和 3
                    left++; right--; // 首先无论如何先要进行加减操作
                    while (left < right && nums[left] == nums[left - 1]) left++;
                    while (left < right && nums[right] == nums[right + 1]) right--;
                } else if (nums[left] + nums[right] < target) {
                    left++;
                } else {  // nums[left] + nums[right] > target
                    right--;
                }
            }
        }
        return ans;
    }
}
```

10、四数之和

四个不同，和为目标值

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);

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

            if (i > 0 && nums[i - 1] == nums[i]) {
                continue;
            }

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

                if (j > i + 1 && nums[j - 1] == nums[j]) {
                    continue;
                }

                int left = j + 1;
                int right = nums.length - 1;
                while (right > left) {
                    int sum = nums[i] + nums[j] + nums[left] + nums[right];
                    if (sum > target) {
                        right--;
                    } else if (sum < target) {
                        left++;
                    } else {
                        result.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));

                        while (right > left && nums[right] == nums[right - 1]) right--;
                        while (right > left && nums[left] == nums[left + 1]) left++;

                        left++;
                        right--;
                    }
                }
            }
        }
        return result;
    }
}
```

11、分糖果

575

Alice 有 n 枚糖，其中第 i 枚糖的类型为 candyType[i] 。Alice 注意到她的体重正在增长，所以前去拜访了一位医生。

医生建议 Alice 要少摄入糖分，只吃掉她所有糖的 n / 2 即可（n 是一个偶数）。Alice 非常喜欢这些糖，她想要在遵循医生建议的情况下，尽可能吃到最多不同种类的糖。

给你一个长度为 n 的整数数组 candyType ，返回： Alice 在仅吃掉 n / 2 枚糖的情况下，可以吃到糖的最多种类数。

示例 1：

输入：candyType = [1,1,2,2,3,3]
输出：3
解释：Alice 只能吃 6 / 2 = 3 枚糖，由于只有 3 种糖，她可以每种吃一枚。

```
class Solution {
    public int distributeCandies(int[] candyType) {
        Set<Integer> set = new HashSet<Integer>();
        for(int candy : candyType) set.add(candy);
        return Math.min(set.size(),candyType.length>>1);
    }
}
```

12、斐波那契数

斐波那契数，通常用 F(n) 表示，形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0，F(1) = 1
F(n) = F(n - 1) + F(n - 2)，其中 n > 1
给你 n ，请计算 F(n) 。

```
//递归
class Solution {
    public int fib(int n) {
        if(n == 0 || n == 1){
            return n;
        }
        return fib(n - 1) + fib(n - 2);
    }
}
//动态规划
class Solution {
    public int fib(int n) {
        if (n < 2) {
            return n;
        }
        int p = 0, q = 0, r = 1;
        for (int i = 2; i <= n; ++i) {
            p = q;
            q = r;
            r = p + q;
        }
        return r;
    }
}
```

13、爬楼梯

```
class Solution {
    public int climbStairs(int n) {
        int p = 0, q = 0, r = 1;
        for (int i = 1; i <= n; ++i) {
            p = q;
            q = r;
            r = p + q;
        }
        return r;
    }
}
```

14、替换空格

```
class Solution {
    public String replaceSpace(String s) {
        StringBuilder sb = new StringBuilder();
        for(int i = 0 ; i < s.length(); i++){
            char c = s.charAt(i);
            if(c == ' ') sb.append("%20");
            else sb.append(c);
        }
        return sb.toString();
    }
}
```

15、删除链表中的节点

请编写一个函数，用于 删除单链表中某个特定节点 。在设计函数时需要注意，你无法访问链表的头节点 head ，只能直接访问 要被删除的节点 。

题目数据保证需要删除的节点 不是末尾节点 。

输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
解释：指定链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9

```
class Solution {
    public void deleteNode(ListNode node) {
        node.val = node.next.val;   //将自己变成下一个（倒霉蛋）节点
        node.next = node.next.next;     //然后将倒霉蛋干掉
    }
}
```

16、提莫攻击

输入：timeSeries = [1,4], duration = 2
输出：4
解释：提莫攻击对艾希的影响如下：

- 第 1 秒，提莫攻击艾希并使其立即中毒。中毒状态会维持 2 秒，即第 1 秒和第 2 秒。
- 第 4 秒，提莫再次攻击艾希，艾希中毒状态又持续 2 秒，即第 4 秒和第 5 秒。
  艾希在第 1、2、4、5 秒处于中毒状态，所以总中毒秒数是 4 。

```
class Solution {
    public int findPoisonedDuration(int[] timeSeries, int duration) {
        int ans = 0;
        int prev = timeSeries[0];
        for(int i=1;i<timeSeries.length;i++){
            int cur = timeSeries[i];
            if(prev + duration-1 < cur){
                ans += duration;
            }else{
                ans += (cur - prev);
            }
            prev = cur;
        }
        ans += duration;
        return ans;
    }
}
```

```
class Solution {
    public int findPoisonedDuration(int[] timeSeries, int duration) {
        int miao = 0;
        int length = timeSeries.length;
        for(int i = 1;i < length;i++){
            if(timeSeries[i] >= timeSeries[i-1] + duration){
                miao = miao + duration;
            }else {
                miao = miao + (timeSeries[i] - timeSeries[i-1]);
            }
        }
        return miao + duration;
    }
}
```

17、最大子序和

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

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 。

暴力法（超出时间限制哈哈哈）时间复杂度O（n^2）

```
class Solution {
    public int maxSubArray(int[] nums) {
        int max = Integer.MIN_VALUE;
        int numsSize = nums.length;
        for(int i = 0 ;i<numsSize;i++){
            int sum = 0;
            for(int j = i;j<numsSize;j++){
                sum += nums[j];
                if(sum > max){
                    max = sum;
                }
            }
        }
        return max;
    }
}
```

动态规划

```
public class Solution {

    public int maxSubArray(int[] nums) {
        int len = nums.length;
        // dp[i] 表示：以 nums[i] 结尾的连续子数组的最大和
        int[] dp = new int[len];
        dp[0] = nums[0];

        for (int i = 1; i < len; i++) {
            if (dp[i - 1] > 0) {
                dp[i] = dp[i - 1] + nums[i];
            } else {
                dp[i] = nums[i];
            }
        }

        // 也可以在上面遍历的同时求出 res 的最大值，这里我们为了语义清晰分开写，大家可以自行选择
        int res = dp[0];
        for (int i = 1; i < len; i++) {
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}

//上面的代码简写为下面

public class Solution {

    public int maxSubArray(int[] nums) {
        int pre = 0;
        int res = nums[0];
        for (int num : nums) {
            pre = Math.max(pre + num, num);
            res = Math.max(res, pre);
        }
        return res;
    }
}
```

18、分割等和子集

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] 。

二维动态规划

```
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
        for (int num : nums) {
            sum += num;
            maxNum = Math.max(maxNum, num);
        }
        if (sum % 2 != 0) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        boolean[][] dp = new boolean[n][target + 1];
        for (int i = 0; i < n; i++) {
            dp[i][0] = true;
        }
        dp[0][nums[0]] = true;
        for (int i = 1; i < n; i++) {
            int num = nums[i];
            for (int j = 1; j <= target; j++) {
                if (j >= num) {
                    dp[i][j] = dp[i - 1][j] | dp[i - 1][j - num];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n - 1][target];
    }
}
```


优化为一维动态规划。不会啊！！

```
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
        for (int num : nums) {
            sum += num;
            maxNum = Math.max(maxNum, num);
        }
        if (sum % 2 != 0) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        boolean[] dp = new boolean[target + 1];
        dp[0] = true;
        for (int i = 0; i < n; i++) {
            int num = nums[i];
            for (int j = target; j >= num; --j) {
                dp[j] |= dp[j - num];
            }
        }
        return dp[target];
    }
```

```
}
```

19、零钱兑换

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

```
//记忆化搜索
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if (amount < 1) {
            return 0;
        }
        return coinChange(coins, amount, new int[amount]);
    }

    private int coinChange(int[] coins, int rem, int[] count) {
        if (rem < 0) {
            return -1;
        }
        if (rem == 0) {
            return 0;
        }
        if (count[rem - 1] != 0) {
            return count[rem - 1];
        }
        int min = Integer.MAX_VALUE;
        for (int coin : coins) {
            int res = coinChange(coins, rem - coin, count);
            if (res >= 0 && res < min) {
                min = 1 + res;
            }
        }
        count[rem - 1] = (min == Integer.MAX_VALUE) ? -1 : min;
        return count[rem - 1];
    }
}
//动态规划
public class Solution {
    public int coinChange(int[] coins, int amount) {
        int max = amount + 1;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

20、下一个更大元素

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：

左括号必须用相同类型的右括号闭合。
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

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

20、下一个更大的元素

给你两个 没有重复元素 的数组 nums1 和 nums2 ，其中nums1 是 nums2 的子集。

请你找出 nums1 中每个元素在 nums2 中的下一个比其大的值。

nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置的右边的第一个比 x 大的元素。如果不存在，对应位置输出 -1 。

示例 1:

输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释:
对于 num1 中的数字 4 ，你无法在第二个数组中找到下一个更大的数字，因此输出 -1 。
对于 num1 中的数字 1 ，第二个数组中数字1右边的下一个较大数字是 3 。
对于 num1 中的数字 2 ，第二个数组中没有下一个更大的数字，因此输出 -1

通过Stack、HashMap解决

先遍历大数组nums2，首先将第一个元素入栈；
继续遍历，当当前元素小于栈顶元素时，继续将它入栈；当当前元素大于栈顶元素时，栈顶元素出栈，此时应将该出栈的元素与当前元素形成key-value键值对，存入HashMap中；
当遍历完nums2后，得到nums2中元素所对应的下一个更大元素的hash表；
遍历nums1的元素在hashMap中去查找‘下一个更大元素’，当找不到时则为-1。

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Stack<Integer> stack = new Stack<Integer>();
        HashMap<Integer, Integer> hasMap = new HashMap<Integer, Integer>();

        int[] result = new int[nums1.length];

        for(int num : nums2) {
            while(!stack.isEmpty() && stack.peek()<num){
                hasMap.put(stack.pop(), num);
            }
            stack.push(num);
        }

        for(int i = 0; i < nums1.length; i++) result[i] = hasMap.getOrDefault(nums1[i], -1);

        return result;
    }
}
```

21、132模式

给你一个整数数组 nums ，数组中共有 n 个整数。132 模式的子序列 由三个整数 nums[i]、nums[j] 和 nums[k] 组成，并同时满足：i < j < k 和 nums[i] < nums[k] < nums[j] 。

如果 nums 中存在 132 模式的子序列 ，返回 true ；否则，返回 false 。

输入：nums = [-1,3,2,0]
输出：true
解释：序列中有 3 个 132 模式的的子序列：[-1, 3, 2]、[-1, 3, 0] 和 [-1, 2, 0] 。

```
class Solution {
    public boolean find132pattern(int[] nums) {
        int n = nums.length;
        int last = Integer.MIN_VALUE; // 132中的2
        Stack<Integer> sta = new Stack<>();// 用来存储132中的3
        if(nums.length < 3)
            return false;
        for(int i=n-1; i>=0; i--){
```

```
            if(nums[i] < last) // 若出现132中的1则返回正确值
                return true;

            // 若当前值大于或等于2则更新2（2为栈中小于当前值的最大元素）
            while(!sta.isEmpty() && sta.peek() < nums[i]){
                last = sta.pop();
            }

            // 将当前值压入栈中
            sta.push(nums[i]);
        }
        return false;
    }
}
```

22、杨辉三角II

给定一个非负索引 `rowIndex`，返回「杨辉三角」的第 `rowIndex` 行。

在「杨辉三角」中，每个数是它左上方和右上方的数的和。

```
输入: rowIndex = 3
输出: [1,3,3,1]
```

```
class Solution {
    public List<Integer> getRow(int rowIndex) {
        List<List<Integer>> C = new ArrayList<List<Integer>>();
        for (int i = 0; i <= rowIndex; ++i) {
            List<Integer> row = new ArrayList<Integer>();
            for (int j = 0; j <= i; ++j) {
                if (j == 0 || j == i) {
                    row.add(1);
                } else {
                    row.add(C.get(i - 1).get(j - 1) + C.get(i - 1).get(j));
                }
            }
            C.add(row);
        }
        return C.get(rowIndex);
    }
}
优化

class Solution {
    public List<Integer> getRow(int rowIndex) {
        List<Integer> pre = new ArrayList<Integer>();
        for (int i = 0; i <= rowIndex; ++i) {
            List<Integer> cur = new ArrayList<Integer>();
            for (int j = 0; j <= i; ++j) {
                if (j == 0 || j == i) {
                    cur.add(1);
                } else {
                    cur.add(pre.get(j - 1) + pre.get(j));
                }
            }
            pre = cur;
        }
        return pre;
    }
}
再优化
class Solution {
    public List<Integer> getRow(int rowIndex) {
        List<Integer> row = new ArrayList<Integer>();
        row.add(1);
        for (int i = 1; i <= rowIndex; ++i) {
            row.add(0);
            for (int j = i; j > 0; --j) {
                row.set(j, row.get(j) + row.get(j - 1));
            }
        }
        return row;
    }
}
```

23、完全平方数

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

```
//动态规划
class Solution {
    public int numSquares(int n) {
        int[] f = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            int minn = Integer.MAX_VALUE;
            for (int j = 1; j * j <= i; j++) {
                minn = Math.min(minn, f[i - j * j]);
            }
            f[i] = minn + 1;
        }
        return f[n];
    }
}
```

24、路径总和

```
//题：112 广度优先算法
class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if (root == null) {
            return false;
        }
        Queue<TreeNode> queNode = new LinkedList<TreeNode>();
        Queue<Integer> queVal = new LinkedList<Integer>();
        queNode.offer(root);
        queVal.offer(root.val);
        while (!queNode.isEmpty()) {
            TreeNode now = queNode.poll();
            int temp = queVal.poll();
            if (now.left == null && now.right == null) {
                if (temp == sum) {
                    return true;
                }
                continue;
            }
            if (now.left != null) {
                queNode.offer(now.left);
                queVal.offer(now.left.val + temp);
            }
            if (now.right != null) {
                queNode.offer(now.right);
                queVal.offer(now.right.val + temp);
            }
        }
        return false;
    }
}
```

25、二叉树的中序遍历

```
94
递归法
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        inorder(root, res);
        return res;
    }

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

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/binary-tree-inorder-traversal/solution/er-cha-shu-de-zhong-xu-bian-li-by-leetcode-solutio/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

26、检测大写字母

我们定义，在以下情况时，单词的大写用法是正确的：

全部字母都是大写，比如 "USA" 。
单词中所有字母都不是大写，比如 "leetcode" 。
如果单词不只含有一个字母，只有首字母大写， 比如 "Google" 。
给你一个字符串 word 。如果大写用法正确，返回 true ；否则，返回 false 。

```
class Solution {
    public boolean detectCapitalUse(String word) {
        char[] cs = word.toCharArray();
        int upper = 0,lower = 0;
        for(int i = 0;i<cs.length;i++){
            if(cs[i]>='a') lower++;
            else upper++;
        }
        if(upper == cs.length) return true;
        if(lower == cs.length) return true;
        if(upper == 1 && cs[0]<'a') return true;
        return false;
    }
}
class Solution {
    public boolean detectCapitalUse(String word) {
        // 若第 1 个字母为小写，则需额外判断第 2 个字母是否为小写
        if (word.length() >= 2 && Character.isLowerCase(word.charAt(0)) && Character.isUpperCase(word.charAt(1))) {
            return false;
        }

        // 无论第 1 个字母是否大写，其他字母必须与第 2 个字母的大小写相同
        for (int i = 2; i < word.length(); ++i) {
            if (Character.isLowerCase(word.charAt(i)) ^ Character.isLowerCase(word.charAt(1))) {
                return false;
            }
        }
        return true;
    }
}
```

27、二位数组中的查找

在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例:

现有矩阵 matrix 如下：

[
 [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]
]
给定 target = 5，返回 true。

给定 target = 20，返回 false。

```
//暴力查找
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) {
            return false;
        }
        int m = matrix.length, n = matrix[0].length;
        int row = 0, col = n - 1;
        while(row < m && col >= 0) {
            if(matrix[row][col] > target) {
                col--;
            }else if(matrix[row][col] < target) {
                row++;
            }else {
                return true;
            }
        }
        return false;
    }
}
```

28、第一个出现一次的字符

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

```
//字典表
class Solution {
    public char firstUniqChar(String s) {
        int[] target = new int[26];
        for(int i = 0;i<s.length();i++) {
            target[s.charAt(i) - 'a']++;
        }

        for(int i =0;i<s.length();i++){
            if(target[s.charAt(i)-'a'] == 1) return s.charAt(i);
        }
        return ' ';
    }
}
```

29、最大单词长度乘积

给定一个字符串数组 words，找到 length(word[i]) * length(word[j]) 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

示例 1:

输入: ["abcw","baz","foo","bar","xtfn","abcdef"]
输出: 16
解释: 这两个单词为 "abcw", "xtfn"。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/maximum-product-of-word-lengths


```
//暴力法
// 暴力解法
// m 表示单词的平均长度，n 表示单词的个数
// 时间复杂度：O(n^2 * m)
// 空间复杂度：O(1)
public int maxProduct1(String[] words) {
    int ans = 0;
    for (int i = 0; i < words.length; i++) {
        String word1 = words[i];
        for (int j = i + 1; j < words.length; j++) {
            String word2 = words[j];
            // 每个单词的 bitMask 会重复计算很多次
            if (!hasSameChar(word1, word2)) {
                ans = Math.max(ans, word1.length() * word2.length());
            }
        }
    }
    return ans;
}
```

```
// O(m^2)
private boolean hasSameChar1(String word1, String word2) {
    for (char c : word1.toCharArray()) {
        if (word2.indexOf(c) != -1) return true;
    }
    return false;
}
```
作者：tangweiqun
链接：https://leetcode-cn.com/problems/maximum-product-of-word-lengths/solution/jian-dan-yi-dong-javac-pythonjsgo-zui-da-ertr/
来源：力扣（LeetCode）


```
//位运算法
class Solution {
    public int maxProduct(String[] words) {
        int[] bits = new int[words.length];
        int res = 0;
        // 将字符串转换为bit数组
        for (int i = 0; i < words.length; i++) {
            char[] chars = words[i].toCharArray();
            for (int j = 0; j < chars.length; j++) {
                bits[i] |= 1 << (chars[j] - 'a');
            }
        }
        //双重for循环找到最大
        for (int i = 0; i < bits.length; i++) {
            for (int j = i + 1; j < bits.length; j++) {
                if((bits[i] & bits[j]) == 0)
                    res = Math.max(res, words[i].length() * words[j].length());
            }
        }
        return res;
    }

}
```
作者：LittleSongFly
链接：https://leetcode-cn.com/problems/maximum-product-of-word-lengths/solution/xiao-song-man-bu-wei-yun-suan-ti-huan-zi-i95k/
来源：力扣（LeetCode）

30、从上到下打印二叉树

```
class Solution {
    public int[] levelOrder(TreeNode root) {
        if(root == null) return new int[0];
        Queue<TreeNode> queue = new LinkedList<>(){{ add(root); }};
        ArrayList<Integer> ans = new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            ans.add(node.val);
            if(node.left != null) queue.add(node.left);
            if(node.right != null) queue.add(node.right);
        }
        int[] res = new int[ans.size()];
        for(int i = 0; i < ans.size(); i++)
            res[i] = ans.get(i);
        return res;
    }
}
```

作者：jyd
链接：https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/solution/mian-shi-ti-32-i-cong-shang-dao-xia-da-yin-er-ch-4/
来源：力扣（LeetCode）


```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            for(int i = queue.size(); i > 0; i--) {
                TreeNode node = queue.poll();
                tmp.add(node.val);
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            res.add(tmp);
        }
        return res;
```

```
    }
}
```

作者：jyd
链接：https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/solution/mian-shi-ti-32-ii-cong-shang-dao-xia-da-yin-er-c-5/
来源：力扣（LeetCode）


```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            LinkedList<Integer> tmp = new LinkedList<>();
            for(int i = queue.size(); i > 0; i--) {
                TreeNode node = queue.poll();
                if(res.size() % 2 == 0) tmp.addLast(node.val); // 偶数层 -> 队列头部
                else tmp.addFirst(node.val); // 奇数层 -> 队列尾部
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

31、二叉树的坡度

给定一个二叉树，计算 整个树 的坡度 。

一个树的 节点的坡度 定义即为，该节点左子树的节点之和和右子树节点之和的 差的绝对值 。如果没有左子树的话，左子树的节点之和为 0 ；没有右子树的话也是一样。空结点的坡度是 0 。

整个树 的坡度就是其所有节点的坡度之和。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/binary-tree-tilt


```
class Solution {
    int ans = 0;

    public int findTilt(TreeNode root) {
        dfs(root);
        return ans;
    }

    public int dfs(TreeNode node) {
        if (node == null) {
            return 0;
        }
        int sumLeft = dfs(node.left);
        int sumRight = dfs(node.right);
        ans += Math.abs(sumLeft - sumRight);
        return sumLeft + sumRight + node.val;
    }
}
```