5. 最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

```
输入："babad"
输出："bab"
注意："aba" 也是一个有效答案。
```

示例 2：

```
输入："cbbd"
输出："bb"
```

思路： 状态表达式：right-left<2,则i-j为回文子串的条件是char[i]=char[j],也就是两个字符相同 right-left>=2,则i-j为回文子串的条件是char[i]=char[j]且char[i+1]=char[j-1]是回文子串.

```
看到那个i+1没有,需要dp数组下一行的数据,一般按行求dp数组是不行的了,所以这个情况是按列来求
```

7. 整数反转

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

```
输入：123
输出：321
```

示例 2:

```
输入：-123
输出：-321
```

示例 3:

```
输入：120
输出：21
```

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 $[-2^{31}, 2^{31} - 1]$。请根据这个假设，如果反转后整数溢出那么就返回 0。

思路:别看那些瞎逼逼的题解.只需要从原数末尾取得一位数,再不断放到结果的最后一位中去.最后窄化一下值变了那就是溢出了

9. 回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

```
输入：121
输出：true
```

示例 2:

```
输入：-121
输出：false
解释：从左向右读，为 -121 。 从右向左读，为 121- 。因此它不是一个回文数。
```

示例 3:

```
输入：10
输出：false
解释：从右向左读，为 01 。因此它不是一个回文数。
```

进阶:

你能不将整数转为字符串来解决这个问题吗?

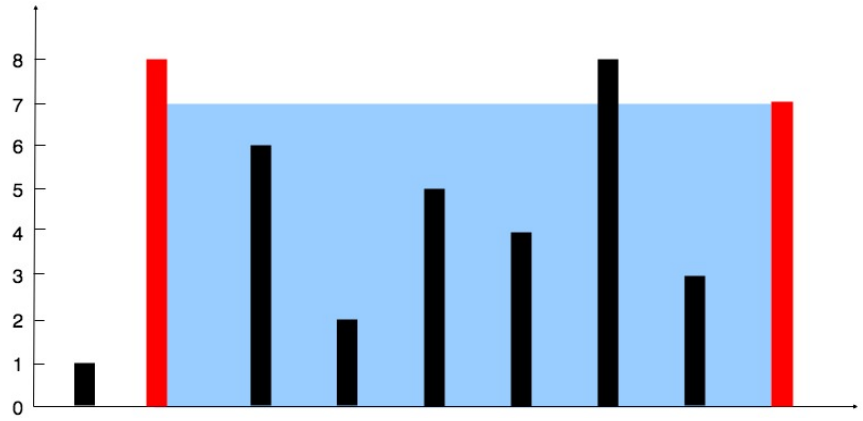
思路:和整数反转一样的思路

---

11. 盛最多水的容器

给你 n 个非负整数 a_1，a_2，...，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，且 n 的值至少为 2。



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:

```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
```

```
class Solution {
    public int maxArea(int[] height) {
        int i=0,j=height.length-1;
        int result=0;
        while(i<j)
        {
            int tempResult=(j-i)*Math.min(height[i],height[j]);
            result=result>tempResult?result:tempResult;
            if(height[i]<=height[j])
            {
                i++;
            }
            else
            {
                j--;
            }
        }
        return result;
    }
}
```

思路: 一开始两个指针一个指向开头一个指向结尾，此时容器的底是最大的，接下来随着指针向内移动，会造成容器的底变小，在这种情况下想要让容器盛水变多，就只有在容器的高上下功夫。 那我们该如何决策哪个指针移动呢？我们能够发现不管是左指针向右移动一位，还是右指针向左移动一位，容器的底都是一样的，都比原来减少了 1。这种情况下我们想要让指针移动后的容器面积增大，就要使移动后的容器的高尽量大，所以我们选择指针所指的高较小的那个指针进行移动，这样我们就保留了容器较高的那条边，放弃了较小的那条边，以获得有更高的边的机会。

---

14. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 `""`。

示例 1:

```
输入: ["flower","flow","flight"]
输出: "fl"
```

示例 2:

```
输入: ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。
```

说明:

所有输入只包含小写字母 `a-z` 。

就注意{"aa","a"}这种情况,所以那里要加一个判定条件

---

15. 三数之和

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 ？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

思路:排序,然后固定最左边一个指针,右边两个指针移动,如果三个指针值相加大于0,则右指针左移;小于0则左指针右移。 注意答案中不可以包含重复的三元组这个条件，所以要处理重复的情况。 首先对于外层固定指针i，如果在向右遍历时出现nums[i]=nums[i-1]则要跳过i,因为只会得到重复的三个数;而对于左右指针,当遍历得到等于0的情况后,也要左指针向右右指针向左找不相同的数的下标.

---

16. 最接近的三数之和

给定一个包括 *n* 个整数的数组 `nums` 和 一个目标值 `target`。找出 `nums` 中的三个整数，使得它们的和与 `target` 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

示例：

```
输入：nums = [-1,2,1,-4], target = 1
输出：2
解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2) 。
```

提示：

- `3 <= nums.length <= 10^3`
- `-10^3 <= nums[i] <= 10^3`
- `-10^4 <= target <= 10^4`

思路:和三数之和大同小异,并且比他简单

---

19. 删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

示例：

```
给定一个链表：1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 *n* 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if(n==0)
        {
            return head;
        }
        ListNode dummy=new ListNode(-1);
        dummy.next=head;
        ListNode quick=dummy,slow=dummy;
        for(int i=0;i<n;i++)
        {
            quick=quick.next;
        }
        while(quick.next!=null)
        {
            slow=slow.next;
            quick=quick.next;
        }
        slow.next=slow.next.next;
        return dummy.next;
    }
}
```

思路： 使用快慢指针，但由于删除倒数第N个节点需要倒数第N+1个节点的引用，所以需要创建一个dummy头结点。主要要处理的是([1,2] 2)这种情况。

---

22. 括号生成

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

示例：

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

```
class Solution {
    List<String>result=new ArrayList<>();
    public List<String> generateParenthesis(int n) {
        backtrack(0,0,new StringBuilder(),n);
        return result;
    }
    public void backtrack(int countL,int countR,StringBuilder cur,int limit)
    {
        if(countL>limit||countR>limit||countR>countL)
        {
            return;
        }
        if(countL==limit&&countR==limit)
        {
            result.add(cur.toString());
            return;
        }
        backtrack(countL+1,countR,cur.append('('),limit);
        cur.deleteCharAt(cur.length()-1);
        backtrack(countL,countR+1,cur.append(')'),limit);
        cur.deleteCharAt(cur.length()-1);
    }
}
```

思路：用两个int记录左括号和右括号数量，只要左括号和右括号数量最终都等于n，并且在方括号的时候右括号的数量永远不大于左括号数量，那么怎么放都是对的

24. 两两交换链表中的节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例:

```
给定 1->2->3->4, 你应该返回 2->1->4->3.
```

思路：注意一下退出条件就好

26. 删除排序数组中的重复项

给定一个排序数组，你需要在 **原地 (http://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)** 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地 (https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 *nums* = **[1,1,2]**,

函数应该返回新的长度 **2**, 并且原数组 *nums* 的前两个元素被修改为 **1**, **2**。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 *nums* = **[0,0,1,1,1,2,2,3,3,4]**,

函数应该返回新的长度 **5**, 并且原数组 *nums* 的前五个元素被修改为 **0**, **1**, **2**, **3**, **4**。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

双指针

31. 下一个排列

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。

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 (https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)**修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1

```
import java.util.*;
class Solution {
    public void nextPermutation(int[] nums) {
        if(nums.length<2)
        {
            return ;
        }
        for(int i=nums.length-1;i>0;i--)
        {
            //从后往前找升序序列
            if(nums[i-1]<nums[i])
            {
                //找到升序序列后,向右找最小的大数
                int min=i;
                for(int j=i;j<nums.length;j++)
                {
                    if(nums[j]>nums[i-1]&&nums[j]<nums[i])
                    {
                        min=j;
                    }
                }
                int temp=nums[i-1];
                nums[i-1]=nums[min];
                nums[min]=temp;
                //最后对其右边的数字排序
                Arrays.sort(nums,i,nums.length);
                return;
            }

        }
        //来到这里说明没有升序序列,则全体倒置
        Arrays.sort(nums);
    }
}
```

思路:

1. 我们希望下一个数比当前数大，这样才满足“下一个排列”的定义。因此只需要将后面的「大数」与前面的「小数」交换，就能得到一个更大的数。比如 123456，将 5 和 6 交换就能得到一个更大的数 123465。
2. 我们还希望下一个数增加的幅度尽可能的小，这样才满足“下一个排列与当前排列紧邻“的要求。为了满足这个要求，我们需要：
    1. 在尽可能靠右的低位进行交换，需要从后向前查找 将一个 尽可能小的且比小数要大的「大数」 与前面的「小数」交换。比如 123465，下一个排列应该把 5 和 4 交换而不是把 6 和 4 交换
    2. 将「大数」换到前面后，需要将「大数」后面的所有数重置为升序，升序排列就是最小的排列。以 123465 为例：首先按照上一步，交换 5 和 4，得到 123564；然后需要将 5 之后的数重置为升序，得到 123546。显然 123546 比 123564 更小，123546 就是 123465 的下一个排列

作者：imageslr 链接：https://leetcode-cn.com/problems/next-permutation/solution/xia-yi-ge-pai-lie-suan-fa-xiang-jie-si-lu-tui-dao-/ (https://leetcode-cn.com/problems/next-permutation/solution/xia-yi-ge-pai-lie-suan-fa-xiang-jie-si-lu-tui-dao-/) 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。

---

34. 在排序数组中查找元素的第一个和最后一个位置

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。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

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]
```

```
class Solution {
    public int[] searchRange(int[] nums, int target) {


        int left=0,right=nums.length;
        int targetIndex=-999;
        while(left<right)
        {
            int mid=(right+left)/2;
            if(nums[mid]>target)
            {
                right=mid;
            }
            else if(nums[mid]<target)
            {
                left=mid+1;
            }
            else{
                targetIndex=mid;
                break;
            }
        }
        int leftBoard=-1,rightBoard=-1;
        if(targetIndex!=-999)
        {
            for(int i=targetIndex,j=targetIndex;;)
            {
                if(i-1>=0&&nums[i-1]==target)
                {
                    i--;
                }
                else{
                    leftBoard=i;
                }
                if(j+1<nums.length&&nums[j+1]==target)
                {
                    j++;
                }
                else{
                    rightBoard=j;
                }
                if(leftBoard!=-1&&rightBoard!=-1)
                {
                    break;
                }
            }
        }
        return new int[]{leftBoard,rightBoard};
    }
}
```

思路:难得不用看题解的题

---

38. 外观数列

给定一个正整数 *n*（1 ≤ *n* ≤ 30），输出外观数列的第 *n* 项。

注意：整数序列中的每一项将表示为一个字符串。

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。前五项如下：

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

第一项是数字 1

描述前一项，这个数是 1 即 “一个 1 ”，记作 11

描述前一项，这个数是 11 即 “两个 1 ” ，记作 21

描述前一项，这个数是 21 即 “一个 2 一个 1 ” ，记作 1211

描述前一项，这个数是 1211 即 “一个 1 一个 2 两个 1 ” ，记作 111221

示例 1:

```
输入：1
输出："1"
解释：这是一个基本样例。
```

示例 2:

```
输入：4
输出："1211"
解释：当 n = 3 时，序列是 "21"，其中我们有 "2" 和 "1" 两组，"2" 可以读作 "12"，也就是出现频次 = 1 而 值 = 2；类似 "1" 可以读作
```

```
class Solution {
   public  String countAndSay(int n) {
        String s=new String("1");
        String[]arr=new String[n];
        arr[0]=s;
        for(int i=1;i<n;i++)
        {
            StringBuilder sb=new StringBuilder();
            char[] chars=arr[i-1].toCharArray();
            int j=chars.length-1,k=j-1;
            Integer count=1;
            while(k>=0)
            {
                if(chars[k]==chars[j])
                {
                    count++;
                }
                else
                {
                    sb.insert(0,chars[j]);
                    sb.insert(0,count.toString().charAt(0));
                    j=k;

                    count=1;
                }
                k--;
            }

                sb.insert(0,chars[j]);
                sb.insert(0,count.toString().charAt(0));

            arr[i]=sb.toString();
        }
        return arr[n-1];
    }
}
```

思路:双指针计算个数

39. 组合总和

给定一个**无重复元素**的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 target ）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,
所求解集为：
[
  [7],
  [2,2,3]
]
```

示例 2：

```
输入：candidates = [2,3,5], target = 8,
所求解集为：
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

提示：

- 1 <= candidates.length <= 30
- 1 <= candidates[i] <= 200
- candidate 中的每个元素都是独一无二的。
- 1 <= target <= 500

暴力回溯

42. 接雨水

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。

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）。 **感谢 Marcos** 贡献此图。

示例:

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

```
import java.util.*;
class Solution {
    public int trap(int[] height) {
        Stack<Integer> stack=new Stack();
        stack.push(0);
        int count=0;
        for(int i=1;i<height.length;i++)
        {
                //如果迭代到的矩形比栈顶矩形高度低，则入栈
            if(height[stack.peek()]>=height[i])
            {
                stack.push(i);
                continue;
            }
                        //如果迭代到的矩形比栈顶矩形高度高，则准备出栈统计雨水数量
           while(!stack.isEmpty()&&height[stack.peek()]<height[i])
            {
                                //将比当前矩形高度小的栈顶pop()出,先只pop()一个,因为本质上一样是一层层的求雨水
                            int j=height[stack.pop()];
                //将相同元素出栈
                while(!stack.isEmpty() &&height[stack.peek()]==j)
                {
                    stack.pop();
                }
                if(!stack.isEmpty())
                {int width=i-stack.peek()-1;
                                //栈顶矩形的高度和当前矩形高度的较小值减去前矩形高度小的栈顶高度
                int height2=Math.min(height[stack.peek()],height[i])-j;
                count+=(height2*width);
                }
            }
                        //最后将当前矩形加入栈
            stack.push(i);
        }
        return count;
    }
}
```

思路： 单调栈思路，看老阿姨的解题思路,本质上还是一层层的qiu https://leetcode-cn.com/problems/trapping-rain-water/solution/dan-diao-zhan-jie-jue-jie-yu-shui-wen-ti-by-sweeti/ (https://leetcode-cn.com/problems/trapping-rain-water/solution/dan-diao-zhan-jie-jue-jie-yu-shui-wen-ti-by-sweeti/)

关联题目: 84. 柱状图中最大的矩形

---

55. 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。
```

示例 2:

```
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

解题思路： 如果某一个作为 起跳点 的格子可以跳跃的距离是 3，那么表示后面 3 个格子都可以作为 起跳点。 可以对每一个能作为 起跳点 的格子都尝试跳一次，把 能跳到最远的距离 不断更新。 如果可以一直跳到最后，就成功了。

作者：ikaruga 链接：https://leetcode-cn.com/problems/jump-game/solution/55-by-ikaruga/ (https://leetcode-cn.com/problems/jump-game/solution/55-by-ikaruga/) 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。

---

56. 合并区间

给出一个区间的集合，请合并所有重叠的区间。

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。
```

注意：输入类型已于2019年4月15日更改。 请重置默认代码定义以获取新方法签名。

提示：

- `intervals[i][0] <= intervals[i][1]`

```
class Solution {
    public int[][] merge(int[][] arr) {
        if(arr == null || arr.length<=1)
            return arr;
        List<int[]> list = new ArrayList<>();
        //Arrays.sort(arr,(a,b)->a[0]-b[0]);
        Arrays.sort(arr,new Comparator<int[]>(){
            @Override
            public int compare(int[] a,int[] b){
                return a[0]-b[0];
            }
        });
        int i=0;
        int n = arr.length;
        while(i<n){
            int left = arr[i][0];
            int right = arr[i][1];
            while(i<n-1 && right>=arr[i+1][0]){
                  //此时intervals[i+1][0]是一定比intervals[i][0]大的，因为已经排好了序
                //只需要处理剩下的两种情况也就是intervals[i+1][1]>intervals[i][1]和
                //intervals[i+1][1]<intervals[i][1]
                right = Math.max(right,arr[i+1][1]);
                i++;
            }
            list.add(new int[] {left,right});
            i++;
        }
        return list.toArray(new int[list.size()][2]);
    }
}
```

思路:按每个数组的第[0]为排序,之后两个循环,内循环不断合并覆盖区间,外循环负责在没有覆盖情况时将指针往右挪.

---

61. 旋转链表

给定一个链表，旋转链表，将链表每个节点向右移动 *k* 个位置，其中 *k* 是非负数。

示例 1:

```
输入: 1->2->3->4->5->NULL, k = 2
输出: 4->5->1->2->3->NULL
解释:
向右旋转 1 步: 5->1->2->3->4->NULL
向右旋转 2 步: 4->5->1->2->3->NULL
```

示例 2:

```
输入: 0->1->2->NULL, k = 4
输出: 2->0->1->NULL
解释:
向右旋转 1 步: 2->0->1->NULL
向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL
```

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head==null||k==0)
        {
            return head;
        }
        ListNode p=head;
        int length=1;
        while(p.next!=null)
        {
            p=p.next;
            length++;
        }
        p.next=head;
        int move=k%length;
        move=length-move;
        for(int i=0;i<move-1;i++)
        {
            head=head.next;

        }
       p=head;
        head=p.next;
        p.next=null;
        return head;
    }
}
```

思路: 没啥好说的,将链表做成环,发现其中规律。

62. 不同路径

一个机器人位于一个 *m x 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?

例如，上图是一个7 x 3 的网格。有多少可能的路径?

示例 1:

```
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 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右
```

示例 2:

```
输入: m = 7, n = 3
输出: 28
```

提示:

- `1 <= m, n <= 100`
- 题目数据保证答案小于等于 `2 * 10 ^ 9`

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][]dp=new int[n][m];
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
                if(i==0||j==0)
                    dp[i][j]=1;
                else
                {
                    dp[i][j]=dp[i-1][j]+dp[i][j-1];
                }
            }
        }
        return dp[n-1][m-1];
    }
}
```

dp[i][j]的值就是从起始点（也就是(0,0)）走到(i, j)的路径数，那么如何求出这个值，我们就需要确定状态转移方程，我们思考一下，假设我们全都知道dp[i][j]的值，题目中说到，小机器人只能往右或者往下，那么dp[i][j]的值就是第 i 行第 j 列这个格子的上面那个格子的值加上左边那个格子的值，也就是dp[i][j] = dp[i-1][j] + dp[i][j-1]，因为这两个格子都可以走到dp[i][j]这个格子，那么他们的路径数之和就是dp[i][j]的值。

上面说到状态转移方程是dp[i][j] = dp[i-1][j] + dp[i][j-1]，那么当 i == 0 或者 j == 0 的时候会越界，而我们想一下，当 i == 0 或者 j == 0 的时候无外乎就是最上一行或者最左一列，我们在最上一行的路径数只能是一条（因为只能一直往左走），所以 dp[0][j]的值全为 1，同理最左一列的值也是 1（因为只能一直往下走），其余的值按照状态转移方程就可以填满了，最后返回最右下角的值（dp[n-1][m-1]）就可以了。

---

64. 最小路径和

给定一个包含非负整数的 *m* x *n* 网格，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:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 7
解释: 因为路径 1→3→1→1→1 的总和最小。
```

递归将会重复走结尾的一段路径，也就是说可能他们开头走的路不同，但是如果不记录，最终后面所走的路径就会发生相同，这就多扫描了很多次。 具体思路就是动态规划，每次都选较小的那一方相加就ok

---

69. x 的平方根

实现 `int sqrt(int x)` 函数。

计算并返回 *x* 的平方根，其中 *x* 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

```
输入: 4
输出: 2
```

示例 2:

```
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,
     由于返回类型是整数，小数部分将被舍去。
```

从 0~ k 之间取中间值m，m*m 与 k 做比较 若 mm > k ; 则下次取 0 ~ mm 中间值比较 若mm < k ; 则下次取 mm ~ k 中间值比较 逐渐逼近...

```
class Solution {
    public int mySqrt(int x) {
        if(x == 1)
            return 1;
        int max=x,min=0;
        while(max-min>1)
        {
            int t=(max+min)/2;
            //用x/m<m而不是m*m>x防止溢出
            if(x/t<t)
            {
                max=t;
            }
            else
            {
                min=t;
            }
        }
        return min;
    }
}
```

---

75. 颜色分类

给定一个包含红色、白色和蓝色，一共 *n* 个元素的数组，**原地(https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95)**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。

注意:
不能使用代码库中的排序函数来解决这道题。

示例:

```
输入: [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]
```

进阶：

- 一个直观的解决方案是使用计数排序的两趟扫描算法。
  首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。
- 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 i=0;
       while(i<=right)
       {
           if(nums[i]==0)
           {
               int temp=nums[left];
               nums[left]=nums[i];
               nums[i]=temp;
               left++;i++;
           }
           else if(nums[i]==2)
           {
               int temp=nums[right];
               nums[right]=nums[i];
               nums[i]=temp;
               right--;
           }
           else
                i++;

       }

    }
}
```

注意事项： 你是从左边开始遍历的，所以在i的左边是不会出现2这个数字的懂吗？所以if(nums[i]==0)之后可以直接i++，因为不可能需要再次调换；而当nums[i]=2的时候向右调换就有可能会换到一个0,那么此时就需要处理nums[i]=0的情况,所以此时不能i++ 思路:双指针

82. 删除排序链表中的重复元素 II

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 *没有重复出现* 的数字。

示例 1:

```
输入: 1->2->3->3->4->4->5
输出: 1->2->5
```

示例 2:

```
输入: 1->1->1->2->3
输出: 2->3
```

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy=new ListNode(-1);
        dummy.next=head;
        ListNode quick=head,slow=dummy;
        while(slow.next!=null&&quick.next!=null)
        {
            boolean flag=false;
            while(quick.next!=null&&quick.next.val==slow.next.val)
            {
                flag=true;
                quick=quick.next;
            }
            if(flag)
            {
                slow.next=quick.next;
                quick=slow.next;
            }
            else
            {
                slow=slow.next;
                quick=quick.next;
            }
        }
        return dummy.next;
    }
}
```

思路： 遇到这种要触及第一个节点的操作，就新建一个dummy头结点就好了，之后就是两个指针的移动和边界限制了。

---

83. 删除排序链表中的重复元素

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入: 1->1->2
输出: 1->2
```

示例 2:

```
输入: 1->1->2->3->3
输出: 1->2->3
```

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head==null)
        {
            return null;
        }
        if(head.next==null)
        {
            return head;
        }
        //因为要留下一个元素，所以就不用头结点了
        ListNode p=head,q=head.next;
        while(true)
        {
            if(q!=null&&p.val==q.val)
            {
                q=q.next;
            }
            else
            {
                p.next=q;
                p=p.next;
                if(q==null)
                {
                    break;
                }
                q=q.next;
            }
        }
        return head;

    }
}
```

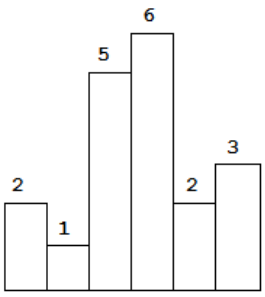
思路: 就是要注意控制[1,2,3,3]这种情况。

---

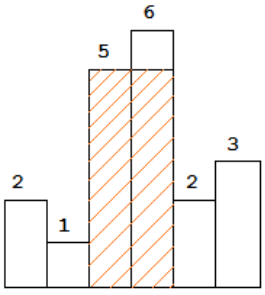
84. 柱状图中最大的矩形

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。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

示例:

```
输入: [2,1,5,6,2,3]
输出: 10
```

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        if(heights==null||heights.length==0)
        {
            return 0;
        }
               // 这里为了代码简便，在柱体数组的头和尾加了两个高度为 0 的柱体。
        int[] tmp = new int[heights.length + 2];
        System.arraycopy(heights, 0, tmp, 1, heights.length);

        Stack<Integer>stack=new Stack<>();
        int sum=0;
        for(int i=0;i<tmp.length;i++)
        {
            if(stack.size()==0||tmp[stack.peek()]<=tmp[i])
            {
                stack.push(i);
            }
            else
            {
               //每pop出一个就计算一次面积
                while(stack.size()!=0&&tmp[stack.peek()]>tmp[i])
                {
                    int h=tmp[stack.pop()];
                    //这里为什么用stack.peek()，自己想想左边界的定义和【2，1，2】的情况
                    int tempSum=(i-stack.peek()-1)*h;
                    sum=tempSum>sum?tempSum:sum;
                }
                stack.push(i);
            }
        }
        return sum;

    }
}
```

思路： 其主要思路其实就是遍历数组中的每个元素，找每个高度的左右边界。而左右边界则是从左数从右数第一个比当前高度小的高度。这样**以左右边界为宽（自己想想[2，1，2]这种情况，重点关注这个左边界的含义）**，以当前下标的高度为高就可以求出一个面积。而这种思路除了用暴力法之外，还可以用单调递增栈实现。每遇到一个比栈顶高度小的下标，就以这个下标为右边界，栈顶的前的栈的下标为左边界，栈顶元素的高度为高去计算面积

88. 合并两个有序数组

给你两个有序整数数组 *nums1* 和 *nums2*，请你将 *nums2* 合并到 *nums1* 中，使 *nums1* 成为一个有序数组。

说明:

- 初始化 *nums1* 和 *nums2* 的元素数量分别为 *m* 和 *n* 。
- 你可以假设 *nums1* 有足够的空间（空间大小大于或等于 *m + n*）来保存 *nums2* 中的元素。

示例:

```
输入:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出: [1,2,2,3,5,6]
```

思路:用双指针比较两个数组中的数据,然后从nums1后面开始放数据,那么就能避免数据移动

90. 子集 II

给定一个可能包含重复元素的整数数组 ***nums***，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

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

只需要判断当前数字和上一个数字是否相同，相同的话跳过即可。当然，要把数字首先进行排序。

91. 解码方法

一条包含字母 `A-Z` 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的**非空**字符串，请计算解码方法的总数。

示例 1:

```
输入: "12"
输出: 2
解释: 它可以解码为 "AB"（1 2）或者 "L"（12）。
```

示例 2:

```
输入: "226"
输出: 3
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。
```

待做

92. 反转链表 II

反转从位置 *m* 到 *n* 的链表。请使用一趟扫描完成反转。

说明:
1 ≤ *m* ≤ *n* ≤ 链表长度。

示例:

```
输入: 1->2->3->4->5->NULL, m = 2, n = 4
输出: 1->4->3->2->5->NULL
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if(m==n)
        {
            return head;
        }
        int index=1;
        ListNode left=head;
        ListNode pre=null;
        if(m==1)
        {
            left=null;
            pre=head;
        }
      else{
          pre=head.next;
          index++;
        while(index<m)
        {
            pre=pre.next;
            left=left.next;
            index++;
        }
      }

        ListNode current=pre.next;
        ListNode next=current.next;
        while(index!=n)
        {
            current.next=pre;
            pre=current;
            current=next;
            index++;
            if(current==null)
            {
                break;
            }
            next=current.next;
        }
        //判断反转的是否是整条链表
        if(left==null&&current==null)
        {
            head.next=null;
            return pre;
        }
        //反转的是以head为开头的子链表
          else if(left==null)
         {
            head.next=current;
            return pre;
         }
         //反转的范围覆盖到源链表的尾部节点情况
         else if(current==null)
         {
             left.next.next=null;
           left.next=pre;
           return head;
         }
         //反转的范围是一条子链表
         else{
            left.next.next=current;
            left.next=pre;
            return head;
```

```
        }
    }
}
```

思路:

1. 注意反转的边界的判断
2. 在反转结束后重新将反转后的链表接上源链表，这个时候就会出现四种情况，需要进行判断。

```
class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if(m==n||head==null)
        {return head;}
         ListNode dummy = new ListNode(0);
         dummy.next=head;
        ListNode pre=dummy;
        int index=1;
        //取得目标的前驱
        while(m!=index)
        {
          pre=pre.next;
          index++;
        }
      ListNode curLeft=pre.next;
      ListNode curRight=curLeft;
      ListNode next=pre.next.next;
      while(index!=n)
      {
          pre.next=next;
          curRight.next=next.next;
          next.next=curLeft;
          index++;
          curLeft=next;
          next=curRight.next;

      }
      return dummy.next;
    }
}
```

另一种思路是: 由1->2->3->4->5到1->3->2->4->5,之后调换4和3到1->4->3->2->5.这种是最正常的做法. 注意:

1. 新增一个头结点,以解决m=1的问题,设立了头结点之后就不用再单独解决m=1的问题了
2. 设立4个指针,其中curLeft和curRight用来标识已经反转好了的链表

---

96. 不同的二叉搜索树

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种?

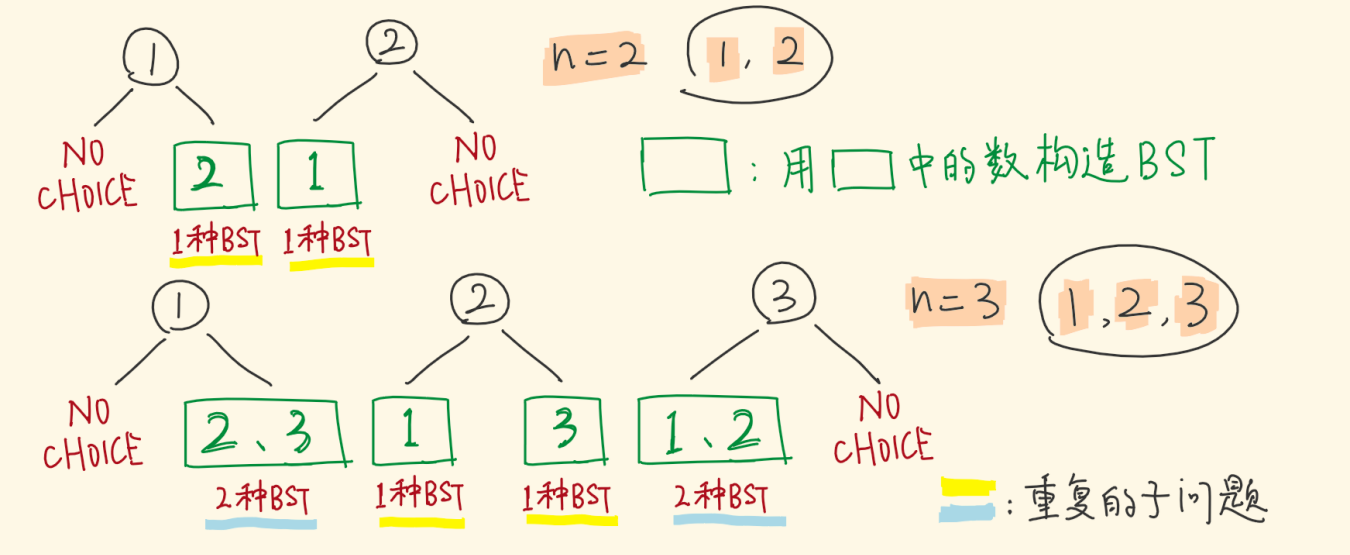
示例:

```
输入: 3
输出: 5
解释:
给定 n = 3, 一共有 5 种不同结构的二叉搜索树:

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

- 按照 BST 的定义，如果整数 1 到 n 中的整数 k 作为根节点值，则 1 ~ k-1 会去构建左子树，k+1 ~ n 会去构建右子树
- 以 k 为根节点的 BST 种类数 = 左子树 BST 种类数 * 右子树 BST 种类数
- 问题变成：计算不同的 k 之下，等式右边的种类数，的累加结果

那么对于3,它的左子树分别有0,1,2个节点三种情况



左子树 BST 用j个节点，右子树 BST 用i-j-1个节点，能构建出dp[j] * dp[i-j-1]种不同的BST

---

98. 验证二叉搜索树

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 。
```

中序遍历时，判断当前节点是否大于中序遍历的前一个节点，如果大于，说明满足 BST，继续遍历；否则直接返回 false。记住是前驱,而不是父结点. 记住涉及到二叉搜索树的遍历一般都要用中序遍历

---

108. 将有序数组转换为二叉搜索树

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树*每个节点* 的左右两个子树的高度差的绝对值不超过 1。

示例:

给定有序数组：[-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：

```
      0
     / \
   -3   9
   /   /
 -10  5
```

每次都取中间的数作为子结点

109. 有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树*每个节点* 的左右两个子树的高度差的绝对值不超过 1。

示例:

给定的有序链表： [-10, -3, 0, 5, 9],

一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树：

```
      0
     / \
   -3   9
   /   /
 -10  5
```

快慢指针找中点

113. 路径总和 II

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 sum = 22，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

返回:

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

深搜就可以,注意看是要从根节点到叶子节点,不需要枚举的

114. 二叉树展开为链表

给定一个二叉树，原地 (https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95/8010757)将它展开为一个单链表。

例如，给定二叉树

```
    1
   / \
  2   5
 / \   \
3   4   6
```

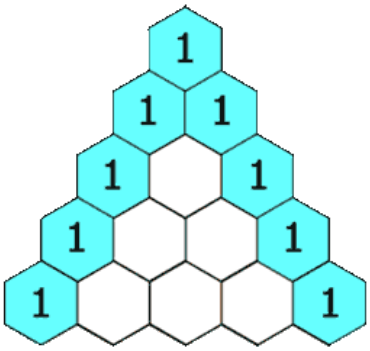
将其展开为：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

采用后续遍历,将左子树的最右边结点.right=右子树这样接上,就是要考虑好nullptr,如[1,null,2,3]这个情况

118. 杨辉三角

给定一个非负整数 *numRows*，生成杨辉三角的前 *numRows* 行。



在杨辉三角中，每个数是它左上方和右上方的数的和。

示例:

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

写死1、2行，然后循环中只专注中间部分的相加，最后再将1加入链表头尾

121. 买卖股票的最佳时机

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

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: [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。
```

待做

122. 买卖股票的最佳时机 II

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：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）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。
```

示例 2:

```
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [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 <= 3 * 10 ^ 4`
- `0 <= prices[i] <= 10 ^ 4`

待做

125. 验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

示例 1:

```
输入: "A man, a plan, a canal: Panama"
输出: true
```

示例 2:

```
输入: "race a car"
输出: false
```

双指针。 注意事项：

1. 看题。只考虑字母和数字字符
2. 考虑aa情况

---

136. 只出现一次的数字

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
```

异或

---

139. 单词拆分

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，判定 *s* 是否可以被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。
- 你可以假设字典中没有重复的单词。

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
```

记住dp[i] 表示 s 中以 i - 1 结尾的字符串是否可被 wordDict 拆分 而检测是否能拆分的方法就是遍历i前面的dp数组查看是否有等于true（左边字符串）且分隔开后的右边字符串

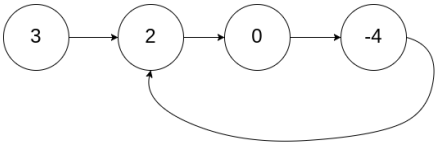
141. 环形链表

给定一个链表，判断链表中是否有环。

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。

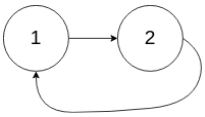
示例 1:

```
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。
```

进阶:

你能用 *O(1)*（即，常量）内存解决此问题吗?

快慢指针判断相遇

142. 环形链表 II

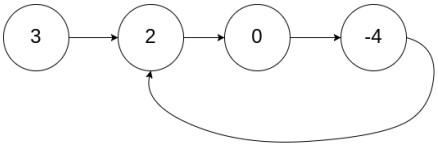
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 `null`。

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。

说明：不允许修改给定的链表。

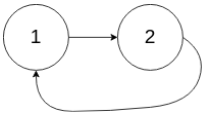
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
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：tail connects to node index 0
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：no cycle
解释：链表中没有环。
```

进阶：
你是否可以不用额外空间解决此题？

快慢指针判定，然后将其中一个指针移到头部，同时走

---

143. 重排链表

给定一个单链表 L：$L_0→L_1→…→L_{n-1}→L_n$，
将其重新排列后变为：$L_0→L_n→L_1→L_{n-1}→L_2→L_{n-2}→…$

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1:

```
给定链表 1->2->3->4, 重新排列为 1->4->2->3.
```

示例 2:

```
给定链表 1->2->3->4->5, 重新排列为 1->5->2->4->3.
```

你的反转链表写完之后可以验证一个3个节点以上的链表吗废物？有一个指针是不变的，写了多少次了？ 思路： 快慢指针找中点，反转链表，逐个插入

144. 二叉树的前序遍历

给定一个二叉树，返回它的 *前序* 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,2,3]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

栈：先放右节点再放左节点

---

146. LRU缓存机制

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 (https://baike.baidu.com/item/LRU)。它应该支持以下操作： 获取数据 `get` 和 写入数据 `put` 。

获取数据 `get(key)` - 如果关键字 (key) 存在于缓存中，则获取关键字的值（总是正数），否则返回 -1。
写入数据 `put(key, 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:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得关键字 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得关键字 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

```
class LRUCache {
 private HashMap<Integer, Node> map = new HashMap<>();
    private int size = 0;
    private int capacity;
    private Node tail ;
    private Node head ;


    public LRUCache(int capacity)
    {
        this.capacity = capacity;
        this.tail=new Node(-1,null,-1,null);
        this.head=new Node(-1,tail,-1,null);
        tail.pre=head;
    }

    public int get(int key)
    {
        Node cur=map.get(key);
        if(cur==null)
        {
            return -1;
        }
        top(cur);
        return cur.value;

    }
    private void top(Node cur)
    {
        cur.pre.next=cur.next;
        cur.next.pre=cur.pre;
        cur.next=tail;
        Node p=tail.pre;
        tail.pre=cur;
        cur.pre=p;
        p.next=cur;
    }
    //tail作为最多使用的结点地址
    public void put(int key, int value)
    {
        if(map.containsKey(key))
        {
            Node t=map.get(key);
            t.value=value;
            map.put(key,t);
            top(t);
            return;
        }
        if (size >= capacity)
        {
            map.remove(head.next.key);
            head.next = head.next.next;
            head.next.pre=head;
            size--;
        }
        Node p=tail.pre;
        tail.pre= new Node(key, tail, value,p);
        p.next=tail.pre;
        size++;
        map.put(key, tail.pre);
    }

    private class Node
    {

        private Node next;
        private int key;
        private int value;
        private Node pre;
        private Node(int key, Node next, int value,Node pre)
        {
```

```
            this.pre=pre;
            this.value = value;
            this.next = next;
            this.key = key;
        }
    }
}
```

思路: 构造一个有头尾结点的双端链表,用HashMap记录结点方便O(1)寻找. 要重点注意的是,无论是put(放入数据或者是更新数据)还是get,都要交换该结点在链表的位置

---

148. 排序链表

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。

示例 1:

```
输入: 4->2->1->3
输出: 1->2->3->4
```

示例 2:

```
输入: -1->5->3->4->0
输出: -1->0->3->4->5
```

思路: 1.归并思想 2.用快慢指针确定链表中点,快指针走两步,慢指针走一步 3.合并时创建一个临时链表存放排好序的链表.要使用三个指针同时在三个链表中移动.千万注意细节.

---

152. 乘积最大子数组

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) {
        //dp[i][0]代表以nums[i]为终点的前面一段的最小值
        int[][]dp=new int[nums.length][2];
        //由于dp计算的是以nums某个元素为重点的最大值,而不是整体最大值,所以要有一个整体最大值的变量
        int max=nums[0];
        //初始化
        dp[0][0]=nums[0];
        dp[0][1]=nums[0];
        for(int i=1;i<nums.length;i++)
        {
            //在思路中讲这个设计的原理
            if(nums[i]>=0)
            {
                dp[i][0]=Math.min(dp[i-1][0]*nums[i],nums[i]);
                dp[i][1]=Math.max(dp[i-1][1]*nums[i],nums[i]);
                max=Math.max(dp[i][1],max);
            }
            else
            {
                dp[i][0]=Math.min(dp[i-1][1]*nums[i],nums[i]);
                dp[i][1]=Math.max(dp[i-1][0]*nums[i],nums[i]);
                max=Math.max(dp[i][1],max);
            }
        }
        return max;

    }
}
```

思路: 首先因为这是乘法,所以一个最小值在乘以一个负数后就会变成最大值,而一个最大值乘以一个负数后就会变成最小值。由于这两个状态的转换所以我们呢两个值都要维护并记录。 其次就是状态方程。

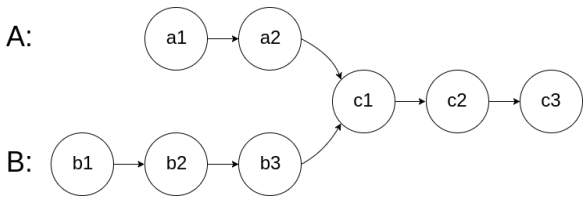
- 由于状态的设计 nums[i] 必须被选取（请大家体会这一点，这一点恰恰好也是使得子数组、子序列问题更加简单的原因：当情况复杂、分类讨论比较多的时候，需要固定一些量，以简化计算）；
- nums[i] 的正负和之前的状态值（正负）就产生了联系，由此关系写出状态转移方程：
    - 当 nums[i] > 0 时，由于是乘积关系：
        - 最大值乘以正数依然是最大值；
        - 最小值乘以同一个正数依然是最小值；
    - 当 nums[i] < 0 时，依然是由于乘积关系：
        - 最大值乘以负数变成了最小值；
        - 最小值乘以同一个负数变成最大值；
    - 当 nums[i] = 0 的时候，由于 nums[i] 必须被选取，最大值和最小值都变成 00 ，合并到上面任意一种情况均成立。
- 但是，还要注意一点，之前状态值的正负也要考虑：例如，在考虑最大值的时候，当 nums[i] > 0 是，如果 dp[i - 1][1] < 0 （之前的状态最大值），此时 nums[i] 可以另起炉灶，所以 `Math.max(dp[i-1][1]*nums[i],nums[i])` 这里要比较 `nums[i]`

---

160. 相交链表

编写一个程序，找到两个单链表相交的起始节点。

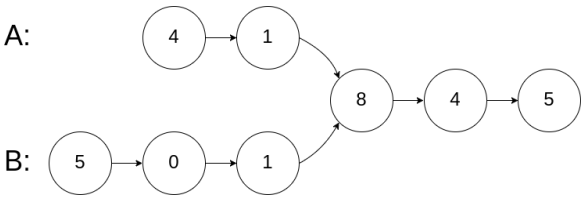
如下面的两个链表：



(https://assets.leetcode-cn.com/aliyun-lc-upload/uploads/2018/12/14/160_statement.png)

在节点 c1 开始相交。

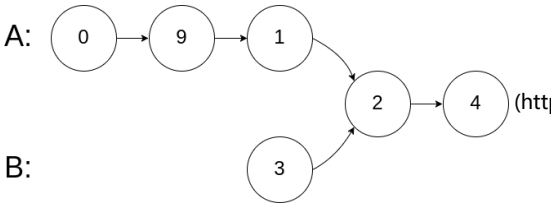
示例 1：



(https://assets.leetcode.com/uploads/2018/12/13/160_example_1.png)

```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：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
```

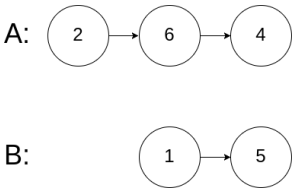
示例 2：



(https://assets.leetcode.com/uploads/2018/12/13/160_example_2.png)

```
输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：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]。
```

示例 3：



(https://assets.leetcode.com/uploads/2018/12/13/160_example_3.png)

```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：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
解释：这两个链表不相交，因此返回 null。
```

注意：

- 如果两个链表没有交点，返回 `null` .
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(*n*) 时间复杂度，且仅用 O(*1*) 内存。

其中一种方法： 创建两个指针 pApA 和 pBpB，分别初始化为链表 A 和 B 的头结点。然后让它们向后逐结点遍历。 当 pApA 到达链表的尾部时，将它重定位到链表 B 的头结点 (你没看错，就是链表 B); 类似的，当 pBpB 到达链表的尾部时，将它重定位到链表 A 的头结点。 若在某一时刻 pApA 和 pBpB 相遇，则 pApA/pBpB 为相交结点。

或者计算出两条链表的长度，然后让长的先走，然后一起走调用==比较

---

162. 寻找峰值

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 `nums` ，其中 `nums[i] ≠ nums[i+1]` ，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1:

```
输入: nums = [1,2,3,1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

示例 2:

```
输入: nums = [1,2,1,3,5,6,4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2;
     或者返回索引 5， 其峰值元素为 6。
```

说明:

你的解法应该是 *O*(*logN*) 时间复杂度的。

二分

---

171. Excel表列序号

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
    A -> 1
    B -> 2
    C -> 3
    ...
    Z -> 26
    AA -> 27
    AB -> 28
    ...
```

示例 1:

```
输入: "A"
输出: 1
```

示例 2:

```
输入: "AB"
输出: 28
```

示例 3:

```
输入: "ZY"
输出: 701
```

致谢:
特别感谢 @ts (http://leetcode.com/discuss/user/ts) 添加此问题并创建所有测试用例。

26进制转十进制

---

172. 阶乘后的零

给定一个整数 *n*，返回 *n*! 结果尾数中零的数量。

示例 1:

```
输入: 3
输出: 0
解释: 3! = 6, 尾数中没有零。
```

示例 2:

```
输入: 5
输出: 1
解释: 5! = 120, 尾数中有 1 个零.
```

说明: 你算法的时间复杂度应为 *O*(log *n*) 。

首先题目的意思是末尾有几个0 比如6! = 【1* 2* 3* 4* 5* 6】 其中只有2*5末尾才有0，所以就可以抛去其他数据 专门看2 5 以及其倍数 毕竟 4 * 25末尾也是0 比如10！ = 【2*4*5*6*8*10】 其中 4能拆成2*2 10能拆成2*5 所以10！ = 【2 *（2*2）* 5 *（2*3）*（2*2*2）*（2*5）】 一个2和一个5配对 就产生一个0 所以10！末尾2个0

```
转头一想 2肯定比5多 所以只数5的个数就行了
```

---

178. 分数排名

编写一个 SQL 查询来实现分数排名。

如果两个分数相同，则两个分数排名（Rank）相同。请注意，平分后的下一个名次应该是下一个连续的整数值。换句话说，名次之间不应该有“间隔”。

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

例如，根据上述给定的 Scores 表，你的查询应该返回（按分数从高到低排列）：

```
+-------+------+
| Score | Rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
```

重要提示：对于 MySQL 解决方案，如果要转义用作列名的保留字，可以在关键字之前和之后使用撇号。例如 **`Rank`**

思路:看这玩意,select的字段可以引用from表的值.正在join的两个表之间不可引用,引用了就不是从哪个join都可以了.

---

185. 部门工资前三高的所有员工

Employee 表包含所有员工信息，每个员工有其对应的工号 Id ，姓名 Name ，工资 Salary 和部门编号 DepartmentId 。

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 85000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
| 7  | Will  | 70000  | 1            |
+----+-------+--------+--------------+
```

`Department` 表包含公司所有部门的信息。

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

编写一个 SQL 查询，找出每个部门获得前三高工资的所有员工。例如，根据上述给定的表，查询结果应返回：

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Randy    | 85000  |
| IT         | Joe      | 85000  |
| IT         | Will     | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
```

解释：

IT 部门中，Max 获得了最高的工资，Randy 和 Joe 都拿到了第二高的工资，Will 的工资排第三。销售部门（Sales）只有两名员工，Henry 的工资最高，Sam 的工资排第二。

```
```

select d.Name as Department,e.Name as Employee,e.Salary from Department as d join Employee as e on d.Id=e.DepartmentId where

#对Join后的每行数据进行统计,查看该行的Salary在Employee同部门中是否有三个比他的工资高的,若是则不是排名前三 (select count(distinct Salary) from Employee where e.salary<salary and e.DepartmentId=DepartmentId)<3 order by d.Name,e.Salary,e.Name ;

---

187. 重复的DNA序列

所有 DNA 都由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如：“ACGAATTCCG”。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找目标子串，目标子串的长度为 10，且在 DNA 字符串 `s` 中出现次数超过一次。

示例：

```
输入：s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
输出：["AAAAACCCCC", "CCCCCAAAAA"]
```

就hashset去重就好了

---

189. 旋转数组

给定一个数组，将数组中的元素向右移动 *k* 个位置，其中 *k* 是非负数。

示例 1:

```
输入: [1,2,3,4,5,6,7] 和 k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右旋转 1 步: [7,1,2,3,4,5,6]
向右旋转 2 步: [6,7,1,2,3,4,5]
向右旋转 3 步: [5,6,7,1,2,3,4]
```

示例 2:

```
输入: [-1,-100,3,99] 和 k = 2
输出: [3,99,-1,-100]
解释:
向右旋转 1 步: [99,-1,-100,3]
向右旋转 2 步: [3,99,-1,-100]
```

说明:

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 要求使用空间复杂度为 O(1) 的 **原地** 算法。

```
class Solution {
    public void rotate(int[] nums, int k) {
        if(k==0||nums.length<=1)
        {
            return ;
        }
         k=k%(nums.length);

        reverse(nums,0,nums.length-1);
        reverse(nums,0,k-1);
        reverse(nums,k,nums.length-1);

    }
    public void reverse(int[]nums,int start,int end)
    {
        for(int i=start,j=end;i<j;i++,j--)
        {
            int temp=nums[i];
            nums[i]=nums[j];
            nums[j]=temp;

        }
    }
}
```

思路: 三次反转

---

197. 上升的温度

给定一个 `Weather` 表，编写一个 SQL 查询，来查找与之前（昨天的）日期相比温度更高的所有日期的 Id。

```
+---------+------------------+------------------+
| Id(INT) | RecordDate(DATE) | Temperature(INT) |
+---------+------------------+------------------+
|       1 |       2015-01-01 |               10 |
|       2 |       2015-01-02 |               25 |
|       3 |       2015-01-03 |               20 |
|       4 |       2015-01-04 |               30 |
+---------+------------------+------------------+
```

例如，根据上述给定的 `Weather` 表格，返回如下 Id:

```
+----+
| Id |
+----+
|  2 |
|  4 |
+----+
```

dateDiff(a.RecordDate,b.RecordDate)函数是两个日期的天数差

199. 二叉树的右视图

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

```
输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

取null左边的结点，还是那几个用队列写要注意的点

200. 岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向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:
[
['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:
[
['1','1','0','0','0'],
['1','1','0','0','0'],
['0','0','1','0','0'],
['0','0','0','1','1']
]
输出: 3
解释: 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。
```

标记深搜，并查集就不要想了麻烦的一批

202. 快乐数

编写一个算法来判断一个数 `n` 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是 **无限循环** 但始终变不到 1。如果 **可以变为** 1，那么这个数就是快乐数。

如果 `n` 是快乐数就返回 `True` ；不是，则返回 `False` 。

示例：

```
输入：19
输出：true
解释：
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1
```

如果出现无限循环则说明数已经成环了，成环那就要用快慢指针解决，当求快慢快乐数相等时那么就可以返回false了

215. 数组中的第K个最大元素

在未排序的数组中找到第 **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
```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

大顶堆

221. 最大正方形

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:

```
输入:

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

输出: 4
```

dp[i][j]表示以第i行第j列为右下角所能构成的最大正方形边长, 则递推式为: dp[i][j] = 1 + min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]);

```
class Solution {
    public int maximalSquare(char[][] matrix) {
        if(matrix.length==0)
            return 0;
        int[][]dp=new int[matrix.length][matrix[0].length];
        int res=0;
        for(int i=0;i<dp.length;i++)
            for(int j=0;j<dp[0].length;j++)
            {
            if(matrix[i][j]=='1')
             {   if(j-1>=0&&i-1>=0)
                {
                    dp[i][j]=Math.min(dp[i][j-1],Math.min(dp[i-1][j-1],dp[i-1][j]))+1;
                }
                else
                    dp[i][j]=1;
              }
              res=Math.max(res,dp[i][j]);
            }
            return res*res;
    }
}
```

227. 基本计算器 II

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式仅包含非负整数，`+`， `-`，`*`，`/` 四种运算符和空格 。 整数除法仅保留整数部分。

示例 1:

```
输入: "3+2*2"
输出: 7
```

示例 2:

```
输入: " 3/2 "
输出: 1
```

示例 3:

```
输入: " 3+5 / 2 "
输出: 5
```

说明:

- 你可以假设所给定的表达式都是有效的。
- 请**不要**使用内置的库函数 `eval`。

我做你妈呢sb题面向用例

234. 回文链表

请判断一个链表是否为回文链表。

示例 1:

```
输入: 1->2
输出: false
```

示例 2:

```
输入: 1->2->2->1
输出: true
```

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?

思路: 1.一开始想到递归

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    ListNode head;
    ListNode p;
    boolean flag=true;
    public boolean isPalindrome(ListNode head) {
        this.head =head;
        this.p=head;
        recursion(head);
        return flag;
    }
    public void recursion(ListNode head)
    {
        if(head==null)
        {
            return;
        }
        recursion(head.next);
        if(head.val!=p.val)
        {
            flag=false;
        }
        p=p.next;
    }
}
```

2.然后看题解原来是用快慢指针得出中点,然后反转后半的链表.最后循环bi'jiao

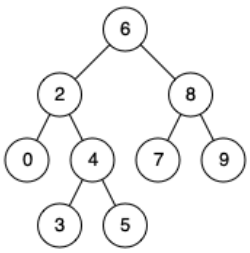
---

235. 二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科 (https://baike.baidu.com/item/%E6%9C%80%E8%BF%91%E5%85%AC%E5%85%B1%E7%A5%96%E5%85%88/8918834?fr=aladdin)中最近公共祖先的定义为: “对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 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**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

注意事项：

1. 利用二叉搜索树的特点向下寻找，找到第一个比较后两结点要分开走的结点，那这个节点就是他们的最近gonbg'gong祖先

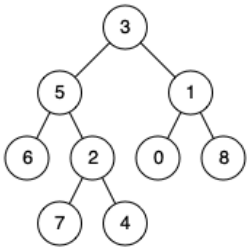
---

236. 二叉树的最近公共祖先

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 (https://baike.baidu.com/item/%E6%9C%80%E8%BF%91%E5%85%AC%E5%85%B1%E7%A5%96%E5%85%88/8918834?fr=aladdin)中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 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**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



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。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    TreeNode result=null;
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        dfs(root,p,q);
        return result;
    }
    public boolean dfs(TreeNode root,TreeNode p,TreeNode q)
    {
        boolean currentFlag=false;
        //控制结点为空的情况
        if(root==null)
        {
            return false;
        }
        //剪枝，找到结果就直接返回
        if(result!=null)
        {
            return false;
        }
        //我所说的第三种情况的判断
        if(root.val==p.val&&root.val==q.val)
        {
            result=root;
            return true;
        }
        //处理第二种情况
        else if(root.val==p.val||root.val==q.val)
        {
            currentFlag=true;
        }
        boolean flagLeft=dfs(root.left,p,q);
        boolean flagRight=dfs(root.right,p,q);
        //处理第二种和第一种情况
        if((flagLeft&&flagRight)||(flagLeft&&currentFlag)||(flagRight&&currentFlag))
        {
            result=root;
            return true;
        }
        else if(flagLeft||flagRight||currentFlag)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

思路： 考虑三种情况：1.两个结点在当前结点左右子树中 2.一个结点是本结点，另一个结点在左子树或右子树中 3.两个结点值相同 要对这三种情况分别处理。之后就是递归判断的逻辑了，如果左子树中有目标结点且右子树也有目标结点 或 当前结点是目标结点，且左子树或右子树也有目标结点，那么第一个检测出这个情况的调用栈的结点则是最终结果

---

238. 除自身以外数组的乘积

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。）

思路： 分开两个数组。left数组负责统计索引左边的乘积，right数组负责统计索引右边的乘积。这样分别从左向右遍历相乘得到left数组，从右向左遍历相乘得到right数组，最后将各个下标元素相乘即可

---

260. 只出现一次的数字 III

给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

示例 :

```
输入：[1,2,1,3,2,5]
输出：[3,5]
```

注意：

1. 结果输出的顺序并不重要，对于上面的例子， `[5, 3]` 也是正确答案。
2. 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现?

同

---

268. 缺失数字

给定一个包含 `0, 1, 2, ..., n` 中 *n* 个数的序列，找出 0 .. *n* 中没有出现在序列中的那个数。

示例 1:

```
输入：[3,0,1]
输出：2
```

示例 2:

```
输入：[9,6,4,2,3,5,7,0,1]
输出：8
```

说明:
你的算法应具有线性时间复杂度。你能否仅使用额外常数空间来实现?

因此我们可以先得到 [0..n][0..n] 的异或值，再将结果对数组中的每一个数进行一次异或运算。未缺失的数在 [0..n][0..n] 和数组中各出现一次，因此异或后得到 0。而缺失的数字只在 [0..n][0..n] 中出现了一次，在数组中没有出现，因此最终的异或结果即为这个缺失的数字。

作者：LeetCode 链接：https://leetcode-cn.com/problems/missing-number/solution/que-shi-shu-zi-by-leetcode/ (https://leetcode-cn.com/problems/missing-number/solution/que-shi-shu-zi-by-leetcode/) 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。

---

279. 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。

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.
```

递推表达式：numSquares(n)=min(numSquares(n-k) + 1） 其中k代表某个小于n的完全平方数。 所以首先要构造一个完全平方数数组，其界限是sqrt(n)+1.然后dp[i]表示的就是n=i时的最大完全平方组合个数.然后在内部循环中遍历小于n的完全平方数进行计算既可以了.

```
class Solution {
    public int numSquares(int n) {
        int numSquare[]=new int[(int)Math.sqrt(n)+1];
        for(int i=0;i<numSquare.length;i++)
        {
            numSquare[i]=i*i;
        }
        int[]dp=new int[n+1];
        dp[1]=1;
        for(int i=2;i<dp.length;i++)
        {
            int min=Integer.MAX_VALUE;
            for(int j=1;j<numSquare.length;j++)
            {
                if(numSquare[j]<=i)
                {
                    min=Math.min(min,dp[i-numSquare[j]]+1);
                }
                else
                    break;
            }
            dp[i]=min;
        }
        return dp[dp.length-1];
    }
}
```

---

283. 移动零

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。

第一次想到的思路是找非0数,和0对换.为什么要这么麻烦呢?直接将所有非0数全都堆到前面去不就好了,0又没有顺序的为什么要管他?

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

287. 寻找重复数

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。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明:

1. **不能**更改原数组（假设数组是只读的）。
2. 只能使用额外的 *O*(1) 的空间。
3. 时间复杂度小于 $O(n^2)$ 。
4. 数组中只有一个重复的数字，但它可能不止重复出现一次。

快慢指针思想, fast 和 slow 是指针, nums[slow] 表示取指针对应的元素 注意 nums 数组中的数字都是在 1 到 n 之间的(在数组中进行游走不会越界), 因为有重复数字的出现, 所以这个游走必然是成环的, 环的入口就是重复的元素, 即按照寻找链表环入口的思路来做

```
class Solution {
    public int findDuplicate(int[] nums) {
        int fast=0,slow=0;
        while(true)
        {
            fast=nums[nums[fast]];
            slow=nums[slow];
            if(fast==slow)
            {
                        //此时只是找到了环内的元素而没有找到环的入口
                fast = 0;
                while(nums[slow] != nums[fast]) {
                    fast = nums[fast];
                    slow = nums[slow];
                }
                return nums[slow];
            }
        }
    }
}
```

292. Nim 游戏

你和你的朋友，两个人一起玩 Nim 游戏 (https://baike.baidu.com/item/Nim游戏/6737105)：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。拿掉最后一块石头的人就是获胜者。你作为先手。

你们是聪明人，每一步都是最优解。 编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

示例:

```
输入：4
输出：false
解释：如果堆中有 4 块石头，那么你永远不会赢得比赛；
     因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。
```

规律题做来有意思? 一个找规律的题，挺有意思的：当我拿完还剩1、2、3个时，必败，故我拿前有4个时必败，所以只要在我拿前有5、6、7个时，就可以必胜（5个时拿走一个，6拿2，7拿3，使对手转入拿前4个的必败状态），所以我拿前还有8个时必败（使对手转入必胜的拿前5、6、7状态）... ...

---

300. 最长上升子序列

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入：[10,9,2,5,3,7,101,18]
输出：4
解释：最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 O(n^2) 。

进阶: 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) {
        if(nums.length==0)
            return 0;
        int[]dp=new int[nums.length];
        int maxDP=0;
        for(int i=0;i<dp.length;i++)
        {
            int max=0;

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

                if(nums[j]<nums[i])
                    max=Math.max(dp[j],max);
            }
            dp[i]=max+1;
            maxDP=Math.max(dp[i],maxDP);
        }
        return maxDP;
    }
}
```

比较简单,就是要注意返回的时候是需要遍历dp数组的,dp[i]代表的是nums[i~0]的符合nums[i]的递增序列,而不是整体的最大递增序列

---

313. 超级丑数

编写一段程序来查找第 `n` 个超级丑数。

超级丑数是指其所有质因数都是长度为 `k` 的质数列表 `primes` 中的正整数。

示例:

```
输入：n = 12, primes = [2,7,13,19]
输出：32
解释：给定长度为 4 的质数列表 primes = [2,7,13,19]，前 12 个超级丑数序列为：[1,2,4,7,8,13,14,16,19,26,28,32] 。
```

说明:

- 1 是任何给定 primes 的超级丑数。
- 给定 primes 中的数字以升序排列。
- 0 < k ≤ 100, 0 < n ≤ 10^6, 0 < primes[i] < 1000 。
- 第 n 个超级丑数确保在 32 位有符整数范围内。

既然求第n小的丑数，可以采用最小堆来解决。每次弹出堆中最小的丑数，然后检查它分别乘以primes后的数是否生成过，如果是第一次生成，那么就放入堆中。第n个弹出的数即为第n小的丑数。 丑数乘以质数列表的数还是等于丑数

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
 if(n==1)
            return 1;
        HashSet<Integer>set=new HashSet<>();
        PriorityQueue<Integer>q=new PriorityQueue<>();
        q.offer(1);
        int count=n-1;
        int arr[]=new int[n];
        for(int i=0;i<arr.length;i++)
        {
            int t=q.poll();
            arr[i]=t;
            for(int j=0;j<primes.length;j++)
            {
                int temp=t*primes[j];
                if(!set.contains(temp)&&count>=0)
                {q.offer(temp);
                    set.add(temp);

                }

            }
            //注意是每poll一次才减一次而不是每放进堆里面一个数就减一次，堆里面top的数是目前序列最小的，但是除top之外的数那就是无法确定的
            count--;

        }

        return arr[arr.length-1];
    }
}
```

322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算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 。

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
```

说明:
你可以认为每种硬币的数量是无限的。

主要就是建立amount长度的数组,对每个数都求一遍个数就好了。就是要注意coins数在dp循环中要跳开

---

326. 3的幂

给定一个整数，写一个函数来判断它是否是 3 的幂次方。

示例 1:

```
输入: 27
输出: true
```

示例 2:

```
输入: 0
输出: false
```

示例 3:

```
输入: 9
输出: true
```

示例 4:

```
输入: 45
输出: false
```

进阶:
你能不使用循环或者递归来完成本题吗?

// 3的n次幂对应的3进制数---> 1 10 100 1000 ... 用正则解决

---

334. 递增的三元子序列

给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

数学表达式如下:

> 如果存在这样的 *i, j, k,* 且满足 0 ≤ *i* < *j* < *k* ≤ *n*-1 ,
> 使得 *arr[i]* < *arr[j]* < *arr[k]* ， 返回 true ; 否则返回 false 。

说明: 要求算法的时间复杂度为 O(*n*)，空间复杂度为 O(*1*) 。

示例 1:

```
输入: [1,2,3,4,5]
输出: true
```

示例 2:

```
输入: [5,4,3,2,1]
输出: false
```

---

m1, m2保存两个较小数，找出一个同时大于m1和m2的数即返回。

```
class Solution {
    public boolean increasingTriplet(int[] nums) {
       int a=Integer.MAX_VALUE ,b=Integer.MAX_VALUE;
       for(int i=0;i<nums.length;i++)
       {
           if(nums[i]<=a)
           {
               a=nums[i];
           }
           else if(nums[i]<=b)
           {
               b=nums[i];
           }
           else
           {
               return true;
           }
       }
       return false;
    }
}
```

337. 打家劫舍 III

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]

     3
    / \
   2   3
    \   \
     3   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]

     3
    / \
   4   5
  / \   \
 1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

思路：做这道题的目标就是，对于每一次打劫都要记录好偷当前结点时的最多钱数和不投当前结点时的最多钱数这两个东西，来让偷下一个结点的时候能根据这两个状态继续构造除这两个状态。 然而还要注意一个点是这个问题没有办法化成 题意那种局部问题，意思就是无法从根节点开始进行抢劫而是要后序遍历从叶子节点开始抢劫，然后遍历完整棵树后才能得知状态。**所以对于每一个节点，如果偷，那么其左右子树偷的钱就不能算进去了；如果不偷，则要将左右子树算上。每一个节点都代表以其作为根节点的子树的抢钱多少 这个状态**

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int rob(TreeNode root) {
        int []t=dfs(root);
      return Math.max(t[0],t[1]);

    }

    private int[] dfs(TreeNode root)
    {
        if(root==null)
            return new int[2];
        int[]a=dfs(root.left);
        int[]b=dfs(root.right);
        int[]t=new int[2];
        t[0]=Math.max(a[0],a[1]) + Math.max(b[0],b[1]);
        t[1]=a[0]+b[0]+root.val;
        return t;

    }
}
```

347. 前 K 个高频元素

给定一个非空的整数数组，返回其中出现频率前 ***k*** 高的元素。

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:

- 你可以假设给定的 *k* 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
- 你的算法的时间复杂度**必须**优于 O(*n* log *n*) , *n* 是数组的大小。
- 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。
- 你可以按任意顺序返回答案。

第一个思路就是HashMap+堆. 第二个思路就是HashMap+桶，也就是说将数字按频率存放入List桶数组中，最后在从后往前遍历桶数组以达到取出频率较大的数的目的

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer,Integer> map=new HashMap<>();
        for(int i=0;i<nums.length;i++)
        {
            if(map.containsKey(nums[i]))
                map.put(nums[i],map.get(nums[i])+1);
            else
                map.put(nums[i],1);
        }
        List<Integer>[]bucket=new List[nums.length+1];
        for(Map.Entry<Integer,Integer> e:map.entrySet())
        {
            int key=e.getKey();
            int value=e.getValue();
            if(bucket[value]==null)
            {
                bucket[value]=new ArrayList<Integer>();
                bucket[value].add(key);
            }
            else
                bucket[value].add(key);
        }
        int result[]=new int[k];
        kp:for(int count=0,i=bucket.length-1;i>=0;i--)
        {
            if(bucket[i]!=null)
            {
                for(Integer in:bucket[i])
                {
                    if(count==k)
                        break kp;
                    result[count]=in;
                    count++;
                }
            }
        }
        return result;
    }
}
```

349. 两个数组的交集

给定两个数组，编写一个函数来计算它们的交集。

示例 1：

```
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]
```

示例 2：

```
输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]
```

说明：

- 输出结果中的每个元素一定是唯一的。
- 我们可以不考虑输出结果的顺序。

？？直接Set

378. 有序矩阵中第K小的元素

给定一个 *n* x *n* 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。
请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

提示:
你可以假设 k 的值永远是有效的， $1 \le k \le n^2$ 。

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        PriorityQueue<Point>q=new PriorityQueue<>((p1,p2)->Integer.compare(matrix[p1.x][p1.y],matrix[p2.x][p2.y]));
        for(int i=0;i<matrix.length;i++)
        {
            Point t=new Point(i,0);
            q.offer(t);
        }
        for(int i=0;i<k-1;i++)
        {
            Point p=q.poll();
            if(p.y+1<matrix[0].length)
            {
                q.offer(new Point(p.x,p.y+1));
            }
        }
        Point result=q.poll();
        return matrix[result.x][result.y];
    }
   static class Point
    {
        int x,y;
        public Point(int x,int y)
        {
            this.x=x;
            this.y=y;
        }
    }
}
```

其实解决这个问题的关键，在于维护一组“最小值候选人”： 你需要保证最小值必然从这组候选人中产生，于是每次只要从候选人中弹出最小的一个即可。 我们来选择第一组候选人，在这里可以选择第一列，因为每一个数字都是其对应行的最小值，全局最小值也必然在其中。 随后在弹出时肯定就弹出左上角的那个数,因为他肯定是最小,那么弹出之后怎么办呢?就将他在数组中右边位置的元素加进堆就好了,因为没有了左上角的元素就是他有可能最小了

394. 字符串解码

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："abcabccdcdcdef"
```

示例 4：

```
输入：s = "abc3[cd]xyz"
输出："abccdcdcdxyz"
```

使用两个栈,栈chars负责存放'['和字母,而nums栈负责存放数字,每当遇到 ']'时就 持续将栈chars中的字母pop出,并根据nums栈顶的重复个数进行拷贝,然后再将所有的字母放进chars栈中

```
class Solution {
    public String decodeString(String s) {
        Stack<Character>chars=new Stack<>();
        Stack<Integer>nums=new Stack<>();
        char[]arr=s.toCharArray();
        StringBuilder res=new StringBuilder();
        for(int i=0;i<arr.length;)
        {
          if(arr[i]>='0'&&arr[i]<='9')
           {
                 int j=i+1;
                 StringBuilder sb=new StringBuilder(String.valueOf(arr[i]));
                 for(;arr[j]!='[';j++)
                    sb.append(arr[j]);
                 nums.push(Integer.parseInt(sb.toString()));
                 i=j;
                 continue;
           }
           else if(arr[i]==']')
           {

               char t;
               StringBuilder temp=new StringBuilder();
               while((t=chars.pop())!='[')
               {
                    temp.insert(0,t);
               }
               int k=nums.pop();
               StringBuilder temp2=new StringBuilder();
               for(int j=0;j<k;j++)
               {
                   temp2.append(temp.toString().toCharArray());
               }
               for(char c:temp2.toString().toCharArray())
               {
                   chars.push(c);
               }

           }
           else
                chars.push(arr[i]);

           i++;
        }
        while(!chars.isEmpty())
            res.insert(0,chars.pop());
        return res.toString();


    }

}
```

402. 移掉K位数字

给定一个以字符串表示的非负整数 *num*，移除这个数中的 *k* 位数字，使得剩下的数字最小。

注意:

- *num* 的长度小于 10002 且 ≥ *k*。
- *num* 不会包含任何前导零。

示例 1 :

```
输入: num = "1432219", k = 3
输出: "1219"
解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。
```

示例 2 :

```
输入: num = "10200", k = 1
输出: "200"
解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。
```

示例 **3 :**

```
输入: num = "10", k = 2
输出: "0"
解释: 从原数字移除所有的数字，剩余为空就是0。
```

对于两个相同长度的数字序列，最左边不同的数字决定了这两个数字的大小，例如，对于 A = 1axxx，B = 1bxxx，如果 a > b 则 A > B。 知道了这个以后，我们可以想到，在删除数字时应该从左向右迭代。 对于每个数字，如果该数字小于栈顶部，即该数字的左邻居，则弹出堆栈，即删除左邻居。否则，我们把数字推到栈上。 我们重复上述步骤（1），直到任何条件不再适用，例如堆栈为空（不再保留数字）。或者我们已经删除了 k 位数字。 注意:

1. 还要注意的是112这种情况,因为最大的数字在最后所以没有数字被删除,所以要在最后新增一个循环删除最后的数字
2. 注意前面都是0的情况,还是要一个循环将其全部删除

```
class Solution {
 public String removeKdigits(String num, int k) {
     if(num.length()<=k)
     {
         return "0";
     }
     char[]arr=num.toCharArray();
     Stack<Character>s=new Stack<>();
     s.push(arr[0]);
     for(int i=1;i<arr.length;i++)
     {

         if(!s.isEmpty()&&s.peek()>arr[i]&&k>0)
         {
             while(!s.isEmpty()&&s.peek()>arr[i]&&k>0)
             {
                 s.pop();
                 k--;
             }
             s.push(arr[i]);

         }
         else
             s.push(arr[i]);
     }
     StringBuilder sb=new StringBuilder();
     while(k!=0)
     {
         s.pop();
         k--;
     }
     while(!s.isEmpty())
     {
         sb.insert(0,s.pop());
     }

     while(sb.length()>1&&sb.charAt(0)=='0')
     {
         sb.deleteCharAt(0)        ;
     }
     return sb.toString().equals("")?"0":sb.toString();
 }
}
```

---

406. 根据身高重建队列

假设有打乱顺序的一群人站成一个队列。 每个人由一个整数对 (h, k) 表示，其中 h 是这个人的身高， k 是排在这个人前面且身高大于或等于 h 的人数。 编写一个算法来重建这个队列。

注意:
总人数少于1100人。

示例

```
输入:
[[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]]
```

完全看不懂题

---

409. 最长回文串

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

注意:
假设字符串的长度不会超过 1010。

示例 1:

```
输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。
```

注意事项: 如果某字母有偶数个，因为偶数有对称性，可以把它全部用来构造回文串；但如果是奇数个的话，并不是完全不可以用来构建，也不是只能选最长的那个，而是只要砍掉1个，剩下的变成偶数就可以全部计入了 但奇数字母里，可以保留1个不砍，把它作为回文串的中心，所以最后还要再加回一个1 但是！如果压根没有奇数的情况，这个1也不能随便加，所以还要分情况讨论

总之就是将偶数次数的字符直接加入结果,而对于奇数字符有两种情况,第一种情况是该字符的个数最大,那么将其当作重点那一段,也就是可以直接加入result,而其它的就-1再加入 我的代码有错误的地方是因为在找奇数最大值的时候覆盖掉的前面的奇数最大值没有再一次将它算进result中

```
class Solution {
    public int longestPalindrome(String s) {
        HashMap<Character,Integer> map=new HashMap<>();
        char[]arr=s.toCharArray();
        for(char c:arr)
        {
            if(!map.containsKey(c))
            {
                map.put(c,1);
            }
            else
            {
                map.put(c,map.get(c)+1);
            }
        }
        int result=0;
        int countOdd=0;
        for(Map.Entry<Character,Integer> e:map.entrySet() )
        {
            int t=e.getValue();
            if((t&1)==0)
            {
                result+=t;
            }
            else if(countOdd<t)
            {
                if(countOdd!=0)
                {
                  result+=countOdd-1;
                }

                countOdd=t;
            }
            else
                result+=t-1;
        }
        return result+countOdd;
    }
}
```

415. 字符串相加

给定两个字符串形式的非负整数 `num1` 和 `num2` ，计算它们的和。

提示:

1. `num1` 和 `num2` 的长度都小于 5100
2. `num1` 和 `num2` 都只包含数字 `0-9`
3. `num1` 和 `num2` 都不包含任何前导零
4. **你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式**

---

tm的不能转Integer就利用-'0'算的具体的值呗

416. 分割等和子集

给定一个**只包含正整数**的**非空**数组。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。

注意:

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

示例 1:

```
输入: [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: [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.
```

一、01背包

- 问题描述:01背包问题是指有n件商品，每件商品价值v[i]重量w[i]，并且有可以承受w重量的背包,问怎么拿价值最高.(物品只有一件只能拿走一次)

01背包是以背包容量作为dp数组列进行解决的，dp[i][j]代表着j容量下背包最多的价值,而行数i的增加表示的则是商品可选择种类的增加。

状态方程: `dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i])` (以递归调用看这个表达式,不选当前物品,那么容量不减,商品种类减去当前物品;如果选,同样减去当前物品并且减去相应容量,增加价值)

若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为j的背包中”，价值为f[i-1][j]；如果放第i件物品，那么问题就转化为“后面的递归调用要将前i-1件物品放入剩下的容量为v-w[i]的背包中”，此时能获得的最大价值就是f [i-1][j-w[i]]再加上通过放入第i件物品获得的价值v[i]。

二、完全背包

- 完全背包唯一的不同点就是商品可以拿无数件. 由于可以拿无数件,那么也就表示我们的解决方案要从01背包的只判断拿和不拿到完全背包的拿多少件,比如0件,1件,2件,3件,4件. 所以此时的状态方程变成: `dp(i,j) = max{dp(i-1, j - w[i] * k) + v[i] * k};` （`0 <= k * w[i] <= t`） 其中k代表拿多少件,j代表背包可承受重量,w代表该物品重量,v代表物品价值

三、本题

本题作为01背包的变体，实际上两者其实差不多。对于题目来说:

- 给定一个只包含正整数的非空数组。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。 其实可以化成一下理解:
  - 是否可以从输入数组中挑选出一些正整数，使得这些数的和 等于 整个数组元素的和的一半。 所以此时dp[i][j]表示的是:对于数组下标[0~i],是否存在一个数组元素集合使得总和等于j,其中j表示的是整个数组元素总和sum /2.在有了这层理解之后,那么也就能够理解为什么在构造dp数组的时候不需要考虑另外一个集合了,因为根据上面所说的那个理解,对于可供选择的数的总和是sum,而sum/2就是j。那也就是说只要在可供选择的数中找到了和为sum一半的组合，那么另外一个和为sum一半的组合也肯定存在，因为sum就是他们所有元素的和啊。

所以也能够得出状态转移方程: `dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i]]` 前半部分表示不选择当前元素；后半部分表示选择当前元素，所以减去元素的值递归查看 `dp[i - 1][j - nums[i]]` 是否是true。这里仍然用递归的含义去看比较好理解。 还有要注意dp[0][0]这种情况,必定是true的,因为不选就是0了.

---

448. 找到所有数组中消失的数字

给定一个范围在 1 ≤ a[i] ≤ *n* (*n* = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 [1, *n*] 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为*O(n)*的情况下完成这个任务吗? 你可以假定返回的数组不算在额外空间内。

示例:

```
输入:
[4,3,2,7,8,2,3,1]

输出:
[5,6]
```

不就是tm交换吗??评论都在秀啥

451. 根据字符出现频率排序

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

示例 1:

```
输入:
"tree"

输出:
"eert"

解释:
'e'出现两次，'r'和't'都只出现一次。
因此'e'必须出现在'r'和't'之前。此外，"eetr"也是一个有效的答案。
```

示例 2:

```
输入:
"cccaaa"

输出:
"cccaaa"

解释:
'c'和'a'都出现三次。此外，"aaaccc"也是有效的答案。
注意"cacaca"是不正确的，因为相同的字母必须放在一起。
```

示例 3:

```
输入:
"Aabb"

输出:
"bbAa"

解释:
此外，"bbaA"也是一个有效的答案，但"Aabb"是不正确的。
注意'A'和'a'被认为是两种不同的字符。
```

不能用排序解决,比如出现lolo这种情况,则会产生乱序. 所以一定要用HashMap的Entry作为存储对象,然后根据Entry的value大小去构造新字符串

454. 四数相加 II

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 `(i, j, k, l)` ，使得 `A[i] + B[j] + C[k] + D[l] = 0`。

为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 0 ≤ N ≤ 500 。所有整数的范围在 -2^{28} 到 $2^{28} - 1$ 之间，最终结果不会超过 $2^{31} - 1$ 。

例如:

```
输入：
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

输出：
2

解释：
两个元组如下：
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

```
class Solution {
    public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
        HashMap<Integer,Integer>map=new HashMap<>();
        int result=0;
        for(int i=0;i<A.length;i++)
        {
            for(int j=0;j<B.length;j++)
            {
                if(map.containsKey(A[i]+B[j]))
                    map.put(A[i]+B[j],map.get(A[i]+B[j])+1);
                else
                    map.put(A[i]+B[j],1);
            }
        }
          for(int i=0;i<C.length;i++)
        {
            for(int j=0;j<D.length;j++)
            {
                if(map.containsKey(-C[i]-D[j]))
                {
                    result+=map.get(-C[i]-D[j]);
                }
            }

        }
        return result;
    }
}
```

思路:将所有A+B的和放进HashMap中,然后遍历C+D求得相反数

461. 汉明距离

两个整数之间的汉明距离 (https://baike.baidu.com/item/%E6%B1%89%E6%98%8E%E8%B7%9D%E7%A6%BB)指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 `x` 和 `y`，计算它们之间的汉明距离。

注意：
$0 \leq x, y < 2^{31}$.

示例:

```
输入: x = 1, y = 4

输出: 2

解释:
1   (0 0 0 1)
4   (0 1 0 0)
       ↑   ↑

上面的箭头指出了对应二进制位不同的位置。
```

异或+求1

---

494. 目标和

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 - 。对于数组中的任意一个整数，你都可以从 + 或 - 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例:

```
输入：nums: [1, 1, 1, 1, 1], S: 3
输出：5
解释：

-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

一共有5种方法让最终目标和为3。
```

提示:

- 数组非空，且长度不会超过 20 。
- 初始的数组的和不会超过 1000 。
- 保证返回的最终结果能被 32 位整数存下。

代入背包问题一下就想出来了状态方程: `dp(i,j)=dp(i-1,j-nums[i])+dp(i-1,j+nums[i])` 然而有 `+nums[i]` 这一项,所以就意味着得知dp(j=s)不可能只求s,而是要求整个可能的和的范围也就是1000.又由于S可能为负数,所以就以1000表示目标数=0的情况.

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        if(Math.abs(S)>1000)
            return 0;
        int[][]dp=new int[nums.length][2001];
        //以1000为界,1000以下表示负数,1000以上表示正数
        dp[0][nums[0]+1000]=1;
        dp[0][-nums[0]+1000]+=1;
        for(int i=1;i<nums.length;i++)
        {
            for(int j=-1000;j<1001;j++)
            {
                if(j+1000-nums[i]>=0)
                    dp[i][j+1000]+=dp[i-1][j+1000-nums[i]];
                if(j+1000+nums[i]<2001)
                     dp[i][j+1000]+=dp[i-1][j+1000+nums[i]];

            }
        }
        return dp[nums.length-1][S+1000];
    }
}
```

560. 和为K的子数组

给定一个整数数组和一个整数 **k**，你需要找到该数组中和为 **k** 的连续的子数组的个数。

示例 1 :

```
输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。
```

说明 :

1. 数组的长度为 [1, 20,000]。
2. 数组中元素的范围是 [-1000, 1000] ，且整数 **k** 的范围是 [-1e7, 1e7]。

首先计算前缀和数组preixSum， 因为 sum(nums[i]...nums[j])=preixSum[j]-preixSum[i-1]..... (j>=i) 而且 sum(nums[i]...nums[j])==K..... (j>=i)
所以，preixSum[j]-preixSum[i-1]==K..... (j>=i) 即，preixSum[j]-K=preixSum[i-1]..... (j>=i)

所以对每个前缀和元素preixSum[j]，想办法判断它的值减去K的值是否存在于前缀和数组中即可。 又因为j>=i,所以可以用哈希表记录j之前的所有前缀和元素，查看该哈希表里是否存在preixSum[j]-K，若存在，它的个数就是满足等式sum(nums[i]...nums[j])==K的个数，直接加到返回值上

作者：jin-ai-yi 链接：https://leetcode-cn.com/problems/subarray-sum-equals-k/solution/qian-zhui-he-ha-xi-biao-on-by-jin-ai-yi/ (https://leetcode-cn.com/problems/subarray-sum-equals-k/solution/qian-zhui-he-ha-xi-biao-on-by-jin-ai-yi/) 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。

581. 最短无序连续子数组

给定一个整数数组，你需要寻找一个**连续的子数组**，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。

你找到的子数组应是**最短**的，请输出它的长度。

示例 1:

```
输入: [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。
```

说明 :

1. 输入的数组长度范围在 [1, 10,000]。
2. 输入的数组可能包含**重复**元素 ，所以**升序**的意思是**<=。**

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        if(nums.length<=1)
        {
            return 0;
        }
       int max=nums[0];
       int min=nums[nums.length-1];
       int left=-1,right=-1;
       for(int i=1;i<nums.length;i++)
       {
           if(max<=nums[i])
           {
               max=nums[i];
           }
           else
           {
               right=i;
           }
       }
       for(int i=nums.length-2;i>=0;i--)
       {
           if(min>=nums[i])
           {
               min=nums[i];
           }
           else
           {
               left=i;
           }
       }
       if(left!=-1&&right!=-1)
       {
           return right-left+1;
       }
       return 0;
    }
}
```

思路: 如果最右端的一部分已经排好序，这部分的每个数都比它左边的最大值要大，同理，如果最左端的一部分排好序，这每个数都比它右边的最小值小。所以我们从左往右遍历，如果i位置上的数比它左边部分最大值小，则这个数肯定要排序， 就这样找到右端不用排序的部分，同理找到左端不用排序的部分，它们之间就是需要排序的部分

---

713. 乘积小于K的子数组

给定一个正整数数组 `nums` 。

找出该数组内乘积小于 `k` 的连续的子数组的个数。

示例 1:

```
输入: nums = [10,5,2,6], k = 100
输出: 8
解释: 8个乘积小于100的子数组分别为: [10], [5], [2], [6], [10,5], [5,2], [2,6], [5,2,6]。
需要注意的是 [10,5,2] 并不是乘积小于100的子数组。
```

说明:

- `0 < nums.length <= 50000`
- `0 < nums[i] < 1000`
- `0 <= k < 10^6`

有空再做。 给点提示：用滑动窗口是不可能覆盖所有连续组合的，要怎么计算zi'ji'xaing

---

725. 分隔链表

给定一个头结点为 `root` 的链表, 编写一个函数以将链表分隔为 `k` 个连续的部分。

每部分的长度应该尽可能的相等: 任意两部分的长度差距不能超过 1，也就是说可能有些部分为 null。

这k个部分应该按照在链表中出现的顺序进行输出，并且排在前面的部分的长度应该大于或等于后面的长度。

返回一个符合上述规则的链表的列表。

举例： 1->2->3->4, k = 5 // 5 结果 [[1], [2], [3], [4], null]

示例 1：

```
输入：
root = [1, 2, 3], k = 5
输出：[[1],[2],[3],[],[]]
解释：
输入输出各部分都应该是链表，而不是数组。
例如, 输入的结点 root 的 val= 1, root.next.val = 2, \root.next.next.val = 3, 且 root.next.next.next = null。
第一个输出 output[0] 是 output[0].val = 1, output[0].next = null。
最后一个元素 output[4] 为 null, 它代表了最后一个部分为空链表。
```

示例 2：

```
输入：
root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3
输出：[[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]
解释：
输入被分成了几个连续的部分，并且每部分的长度相差不超过1.前面部分的长度大于等于后面部分的长度。
```

提示:

- `root` 的长度范围： `[0, 1000]`.
- 输入的每个节点的大小范围：`[0, 999]`.
- `k` 的取值范围： `[1, 50]`.

---

1. 获取链表的长度len；
2. size = len / k; extra = len % k;
3. 有extra个子链表的长度是size+1，剩余子链表的长度为size；
4. 弄明白第三点之后，接下来就是分隔链表的事情了。

---

768. 最多能完成排序的块 II

这个问题和“最多能完成排序的块”相似，但给定数组中的元素可以重复，输入数组最大长度为 `2000` ，其中的元素最大为 `10**8` 。

`arr` 是一个可能包含**重复元素**的整数数组，我们将这个数组分割成几个“块”，并将这些块分别进行排序。之后再连接起来，使得连接的结果和按升序排序后的原数组相同。

我们最多能将数组分成多少块？

示例 1:

```
输入: arr = [5,4,3,2,1]
输出: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [5, 4], [3, 2, 1] 的结果是 [4, 5, 1, 2, 3]，这不是有序的数组。
```

示例 2:

```
输入: arr = [2,1,3,4,4]
输出: 4
解释:
我们可以把它分成两块，例如 [2, 1], [3, 4, 4]。
然而，分成 [2, 1], [3], [4], [4] 可以得到最多的块数。
```

注意:

- `arr` 的长度在 `[1, 2000]` 之间。
- `arr[i]` 的大小在 `[0, 10**8]` 之间。

``import java.util.ArrayList; class Solution { public int maxChunksToSorted(int[] arr) { List leftMax=new ArrayList(); ListrightMin=new ArrayList(); int max=-1; int count=1; for(int i=0;i<arr.length-1;i++) { max=max>arr[i]?max:arr[i]; leftMax.add(max);

} int min=max; for(int i=arr.length-1;i>0;i--) { min=min<arr[i]?min:arr[i]; rightMin.add(min);

} for(int i=0;i<leftMax.size();i++) { if(leftMax.get(i)<=rightMin.get(leftMax.size()-1-i)) { count++; } } return count; } }`

``` 思路： 这道题和上一题 LeetCode 最多完成排序的块 基本一样，蛋式由于含重复元素并且arr[i]是任意值， 所以排序后不是[0, 1, 2, 3, … , n]这种，然而我们在判断index处是否能够切开还是需要判断比arr[index]小的 元素是否都出现在index的前面。然而这个问题可以转化一下，我们只要寻找[index + 1, arrSize - 1]这一段 是否都比[0, index]这一段的值大，即[0, index]的最大值 < [index + 1, arrSize - 1]

---

# 769. 最多能完成排序的块

数组 `arr` 是 `[0, 1, ..., arr.length - 1]` 的一种排列，我们将这个数组分割成几个“块”，并将这些块分别进行排序。之后再连接起来，使得连接的结果和按升序排序后的原数组相同。

我们最多能将数组分成多少块?

**示例 1:**

```
输入: arr = [4,3,2,1,0]
输出: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [4, 3], [2, 1, 0] 的结果是 [3, 4, 0, 1, 2]，这不是有序的数组。
```

**示例 2:**

```
输入: arr = [1,0,2,3,4]
输出: 4
解释:
我们可以把它分成两块，例如 [1, 0], [2, 3, 4]。
然而，分成 [1, 0], [2], [3], [4] 可以得到最多的块数。
```

注意:

- `arr` 的长度在 `[1, 10]` 之间。
- `arr[i]` 是 `[0, 1, ..., arr.length - 1]` 的一种排列。

```
class Solution {
    public int maxChunksToSorted(int[] arr) {
  //当遍历到第i个位置时，如果可以切分为块，那前i个位置的最大值一定等于i。
        //否则，一定有比i小的数划分到后面的块，那块排序后，一定不满足升序。
        int res = 0, max = 0;
        for (int i = 0; i < arr.length; i++){
            max = Math.max(max, arr[i]);//统计前i个位置的最大元素
            if (max == i) res++;
        }
        return res;
    }
}
```

思路：在判断index处是否能够切开是需要判断比arr[index]小的元素是否都出现在index的前面。因此我们从左向右遍历，如果已经观测到的最大值小于等于这个区间的index，则可以划分区间。
```

945. 使数组唯一的最小增量

给定整数数组 A，每次 *move* 操作将会选择任意 `A[i]`，并将其递增 `1`。

返回使 `A` 中的每个值都是唯一的最少操作次数。

示例 1:

```
输入：[1,2,2]
输出：1
解释：经过一次 move 操作，数组将变为 [1, 2, 3]。
```

示例 2:

```
输入：[3,2,1,2,1,7]
输出：6
解释：经过 6 次 move 操作，数组将变为 [3, 4, 1, 2, 5, 7]。
可以看出 5 次或 5 次以下的 move 操作是不能让数组的每个值唯一的。
```

提示:

1. `0 <= A.length <= 40000`
2. `0 <= A[i] < 40000`

```
import java.util.*;
class Solution {
    //处理四万个数全都是40000的情况
    int hash[]=new int[80000];
    public int minIncrementForUnique(int[] A) {

        Arrays.fill(hash, -1);
        int count=0;
        for(int i=0;i<A.length;i++)
        {
            int b=findPos(A[i]);
            count+=b-A[i];
        }
        return count;
    }
    public int findPos(int k)
    {
        //如果位置为空,则直接放入
        if(hash[k]==-1)
        {
            hash[k]=k;
            return k;
        }
        int b=0;
        //位置不为空,但hash[k]==k则说明产生hash碰撞
        if(hash[k]==k)
        {
            b=findPos(k+1);
        }
        //位置不为空,且hash[k]!=k则说明不仅产生了hash碰撞且遇到了路径压缩
        else
        {
            b=findPos(hash[k]+1);
        }
        //路径压缩的原理,将线性探测所经过的路径全部设为最终放入元素的数组下标
        hash[k]=b;
        return b;

    }
}
```

线性探测法+路径压缩 老阿姨思路:https://leetcode-cn.com/problems/minimum-increment-to-make-array-unique/solution/ji-shu-onxian-xing-tan-ce-fa-onpai-xu-onlogn-yi-ya/ (https://leetcode-cn.com/problems/minimum-increment-to-make-array-unique/solution/ji-shu-onxian-xing-tan-ce-fa-onpai-xu-onlogn-yi-ya/)

1171. 从链表中删去总和值为零的连续节点

给你一个链表的头节点 `head`，请你编写代码，反复删去链表中由 **总和** 值为 `0` 的连续节点组成的序列，直到不存在这样的序列为止。

删除完毕后，请你返回最终结果链表的头节点。

你可以返回任何满足题目要求的答案。

（注意，下面示例中的所有序列，都是对 `ListNode` 对象序列化的表示。）

示例 1：

```
输入：head = [1,2,-3,3,1]
输出：[3,1]
提示：答案 [1,2,1] 也是正确的。
```

示例 2：

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

示例 3：

```
输入：head = [1,2,3,-3,-2]
输出：[1]
```

提示：

- 给你的链表中可能有 `1` 到 `1000` 个节点。
- 对于链表中的每个节点，节点的值：`-1000 <= node.val <= 1000`.

```
class Solution {
    public ListNode removeZeroSumSublists(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        Map<Integer, ListNode> map = new HashMap<>();

        // 首次遍历建立 节点处链表和<->节点 哈希表
        // 若同一和出现多次会覆盖，即记录该sum出现的最后一次节点
        int sum = 0;
        for (ListNode d = dummy; d != null; d = d.next) {
            sum += d.val;
            map.put(sum, d);
        }

        // 第二遍遍历 若当前节点处sum在下一处出现了则表明两结点之间所有节点和为0 直接删除区间所有节点
        sum = 0;
        for (ListNode d = dummy; d != null; d = d.next) {
            sum += d.val;
            d.next = map.get(sum).next;
        }

        return dummy.next;
    }
}
```

思路: 难得有点过分。虽说是利用前缀和加HashMap的思想，但是真正删除起来却异常恶心。 看完题解迷惑的点如下：

1. 如果出现…→0→0→…（0代表某连续节点的值为0）怎么办？仔细看代码，map中存在的是该sum最后一次出现的结点（前面有出现都不用管，因为只要记住最后一个结点的sum，其中间就算有相同sum的结点也好，在其后也被抵消掉了所以才能出现最后一个结点的sum，如a[1,2,4,-6,2,4]，根本不用急记a[1],因为他组合后面的[2,4,-6]的时候就已经被抵消掉了），所以在第二次遍历的时候会将这个两个0覆盖的所有节点都删掉
2. 为什么要遍历第二遍？遍历一遍顺便删除不是就能完事了吗？不是的。要知道删除后的节点是不能继续计算在sum中的，所以在第二遍遍历中并没有加被删除节点的sum，况且map中存在的是该sum最后一次出现的结点，没法记录相同sum的两个结点，所以要重新求sum。（不知道自己在说什么实在是很难，记细节吧）
3. dummy为0
4. ☆**最重要要记住，map记录的是sum最后一次出现的结点，所以一旦出现相同，无论中间的数怎么排列其和也都是等于0，也即都可以删除，如[1,2,4,-6,2,4,5,-3,-2],当时我在担心把[2,4,-6]删掉后,如果[1,2,4,5,-3,-2]出现sum=0的话不就错误了?不会的,因为总是记录sum最后一次出现的结点,如果[1,2,4,5,-3,-2]有组合,那么map中存放的就不是[-6]这个结点了,而是后面的结点了**

难度爆表,所以老老实实暴力O(n平方)挺不错的。

```
class Solution {
    public ListNode removeZeroSumSublists(ListNode head) {
        ListNode dummyHead = new ListNode(-1);
        dummyHead.next = head;

        ListNode cur = dummyHead;
        while (cur.next != null) {
            ListNode lastNode = zeroSum(cur.next);
            if (lastNode != null) {
                cur.next = lastNode.next;
            } else {
                cur = cur.next;
            }
        }

        return dummyHead.next;
    }

    /**
     * 找到从head计算起，总和为0的连续节点
     * @param head 头结点不能为null
     * @return 返回最后一位，如果未找到，返回null
     */
    private ListNode zeroSum(ListNode head) {
        int res = 0;
        ListNode cur = head;
        while (cur != null) {
            res += cur.val;
            if (res == 0) {
                return cur;
            }
            cur = cur.next;
        }

        return null;
    }
}
```

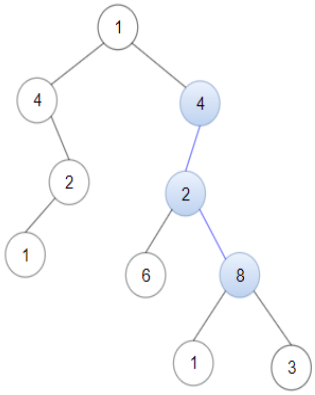
1367. 二叉树中的列表

给你一棵以 `root` 为根的二叉树和一个 `head` 为第一个节点的链表。

如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 `head` 为首的链表中每个节点的值，那么请你返回 `True` ，否则返回 `False` 。

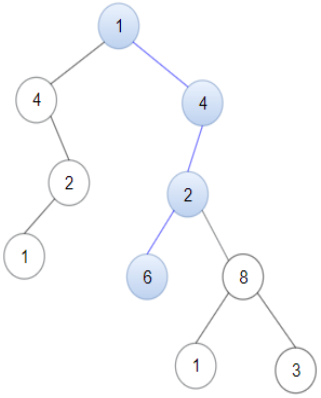
一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。

示例 1：




```
输入：head = [4,2,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
输出：true
解释：树中蓝色的节点构成了与链表对应的子路径。
```

示例 2：




```
输入：head = [1,4,2,6], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
输出：true
```

示例 3：

```
输入：head = [1,4,2,6,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
输出：false
解释：二叉树中不存在一一对应链表的路径。
```

提示：

- 二叉树和链表中的每个节点的值都满足 `1 <= node.val <= 100` 。
- 链表包含的节点数目在 `1` 到 `100` 之间。
- 二叉树包含的节点数目在 `1` 到 `2500` 之间。

```
class Solution {

    public boolean isSubPath(ListNode head, TreeNode root) {

        if(root==null)
        {
            return false;
        }
            return dfs(head,root)||isSubPath(head,root.left)||isSubPath(head,root.right);
    }
    public boolean dfs(ListNode head, TreeNode root)
    {
        if(head==null)
        {
            return true;
        }
        if(root==null)
        {
            return false;
        }
        if(head.val==root.val)
        {
            return dfs(head.next,root.left)||dfs(head.next,root.right);
        }
        return false;

    }
}
```

思路： 枚举法。之前的思路是：如果在某个时刻，遍历到链表和树结点的值不同，那么从当前树节点继续往下找，但是是从链表的头结点开始。问题是，之前遍历过的树节点可能是正确路径的一部分，这里从链表头结点开始就跳过了这部分。比如链表是 1 2 1 2 4，树节点遍历是 1 2 1 2 1 2 4，此时错误的思路就会从第五位数 1 开始设置链表头，所以就会判断成错误。所以是必定要枚举的，即将树中的每个节点都当成是链表头深搜一遍。

面试题 02.06. 回文链表

编写一个函数，检查输入的链表是否是回文的。

示例 1：

```
输入： 1->2
输出： false
```

示例 2：

```
输入： 1->2->2->1
输出： true
```

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？

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isPalindrome(ListNode head) {
        if((head==null)||(head.next==null))
        {
            return true;
        }

        ListNode slow=head,quick=head;
        ListNode rightHalf=null;
        //分隔链表
        while(quick.next!=null&&quick.next.next!=null)
        {
            quick=quick.next.next;
            slow=slow.next;
        }
        //处理奇数个链表节点和偶数个链表结点情况
        if(quick.next==null)
        {
            ListNode temp=new ListNode(slow.val);
            temp.next=slow.next;
            rightHalf=temp;
            slow.next=null;
        }
        else
        {
            rightHalf=slow.next;
            slow.next=null;
        }

        //链表反转
        ListNode dummy=new ListNode(-1);
        dummy.next=rightHalf;

        ListNode right=dummy.next;
        if(right.next!=null)
        {
            ListNode curr=right.next;
            ListNode next=curr.next;
            while(true)
            {
                curr.next=dummy.next;
                right.next=next;
                dummy.next=curr;
                curr=next;
                if(curr==null)
                {
                    break;
                }
                next=next.next;
            }
        }
        //开始遍历比较
       rightHalf=dummy.next;
       while((head!=null&&rightHalf!=null)&&head.val==rightHalf.val)
       {

           head=head.next;
           rightHalf=rightHalf.next;

       }
       if(head==null&&rightHalf==null){
       return true;}
       else{
```

```
            return false;
        }
    }

}
```

思路：

1. 用快慢指针分割链表，要注意奇数个节点和偶数个节点的中点是不一样的。奇数个节点需要将中点复制一份给后半部分链表以做反转（我的思路就是复制，不复制也行）
2. 创建一个dummy头结点作为后半部分头结点，然后反转链表
3. 之后逐个进行比较，需要注意的是比较中出现 （null 非null）比较情况时的处理

注意事项： []和[101]都属于回文链表

面试题 02.05. 链表求和

给定两个用链表表示的整数，每个节点包含一个数位。

这些数位是反向存放的，也就是个位排在链表首部。

编写函数对这两个整数求和，并用链表形式返回结果。

示例：

```
输入：(7 -> 1 -> 6) + (5 -> 9 -> 2)，即617 + 295
输出：2 -> 1 -> 9，即912
```

进阶：假设这些数位是正向存放的，请再做一遍。

示例：

```
输入：(6 -> 1 -> 7) + (2 -> 9 -> 5)，即617 + 295
输出：9 -> 1 -> 2，即912
```

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry=0;
        ListNode p=l1,q=l2;
        ListNode res=new ListNode(-1);
        ListNode k=res;
        while(p!=null||q!=null)
        {
            int num1=(p==null?0:p.val);
            int num2=(q==null?0:q.val);
            int resNum=num1+num2+carry;
            carry=resNum/10;
            resNum=(carry!=0?(resNum-10):resNum);
            k.next=new ListNode(resNum);
            k=k.next;
            p=(p==null?null:p.next);
            q=(q==null?null:q.next);
        }
        if(carry!=0)
        {
            k.next=new ListNode(carry);
        }
        return res.next;

    }
}
```

注意事项：

1. 首先，想到的思路是将其转换为int来计算，但是链表中的值完全有可能要超过任何能表示的基本类型大小，所以这一步不可行并且效率不高 思路： 将链表中各位的值自己进行相加，然后记进位，如果有进位，则在计算下一个节点时将进位加上，期间不断地构造链表节点，这样就能避免大数的问题，而且只需要一次遍历

剑指 Offer 48. 最长不含重复字符的子字符串

请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

示例 1:

```
输入: "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: "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: "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。
```

提示：

- `s.length <= 40000`

注意：本题与主站 3 题相同：https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/ (https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/)

思路：set+滑动窗口，不过要注意的是当发现用重复的字符时，左指针应该是要删除set的字符直到没有重复就停止，而不是要让left=right才停止，如例子dvdf

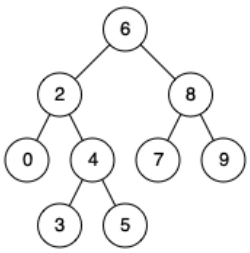
---

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科 (https://baike.baidu.com/item/%E6%9C%80%E8%BF%91%E5%85%AC%E5%85%B1%E7%A5%96%E5%85%88/8918834?fr=aladdin)中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 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**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出：6
解释：节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出：2
解释：节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

注意：本题与主站 235 题相同：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/ (https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/)

利用二叉搜索树的性质 左子树<根<右子树，从而判断p, q在哪个位置 如果都在左子树，那么根节点 root = root->left; 如果都在右子树，那么根节点 root = root->right; 否则就是一个在左子树，一个在右子树，那么最近的公共祖先就是当前根节点，函数返回

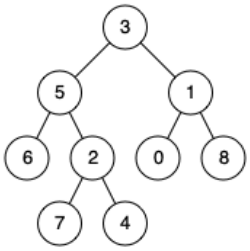
作者：xiao123 链接：https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian-lcof/solution/li-yong-er-cha-shu-xing-zhi-di-gui-by-xiao123/ (https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian-lcof/solution/li-yong-er-cha-shu-xing-zhi-di-gui-by-xiao123/) 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。

剑指 Offer 68 - II. 二叉树的最近公共祖先

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 (https://baike.baidu.com/item/%E6%9C%80%E8%BF%91%E5%85%AC%E5%85%B1%E7%A5%96%E5%85%88/8918834?fr=aladdin)中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 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**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



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。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

注意：本题与主站 236 题相同：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/ (https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/)

和二叉搜索树几乎一致，只不过要遍历整棵树了
