目 录

1.给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。如果数组中不存在目标值 `target`，返回 $[-1,-1]$。OJ 链接-在排序数组中查找元素的第一个和最后一个位置。

Tips

使用二分查找。

关键在于等于时的情况处理：

寻找下界:如果 `mid` 处的值等于目标值,判断 `mid` 的左侧值是否等于目标值,如果左侧没有值或者左侧不等于目标值,说明该 `mid` 为下界,否则需要从 `[lo,mid - 1]` 区间来查找下界，即 `hi = mid - 1`

寻找上界：与寻找下界逻辑相反，如果 `mid` 处的值等于目标值，判断 `mid` 的右侧值是否等于目标值，如果右侧没有值或者右侧不等于目标值，说明该 `mid` 为上界，否则需要从 `[mid + 1, hi]` 区间来查找上界，即 `lo = mid + 1`

在排序数组中查找元素的第一个和最后一个位置

```java
class Solution {
    private int findFirst(int[] nums, int target) {
        int lo = 0;
        int hi = nums.length - 1;
        int mid = 0;
        while(lo <= hi){
            mid = lo + ((hi - lo) >> 1);
            if(nums[mid] < target){
                lo = mid + 1;
            }else if(target < nums[mid]){
                hi = mid - 1;
            }else{
                if(mid == 0 || nums[mid - 1] != target){
                    return mid;
                }else{
                    hi = mid - 1;
                }
            }
        }
        return -1;
    }

    private int findLast(int[] nums, int target) {
        int lo = 0;
        int hi = nums.length - 1;
        int mid = 0;
        while(lo <= hi){
            mid = lo + ((hi - lo) >> 1);
            if(nums[mid] < target){
                lo = mid + 1;
            }else if(target < nums[mid]){
                hi = mid - 1;
            }else{
                if(mid == nums.length - 1 || nums[mid + 1] != target){
                    return mid;
                }else{
                    lo = mid + 1;
                }
```

```
            }
        }
        return -1;
    }
    public int[] searchRange(int[] nums, int target) {
        return new int[]{findFirst(nums,target),findLast(nums,target)};

    }
}
```

2.根据**逆波兰表示法**，求表达式的值。有效的运算符包括 `+` , `-` , `*` , `/` 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。OJ 链接-逆波兰表达式求值

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

Tips

使用题目给的提示：适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

逆波兰表达式求值

```
class Solution {
    private boolean isNumber(String str){
        return !"+".equals(str) && !"-".equals(str) && !"*".equals(str) && !"/".
            equals(str);
    }

    public int evalRPN(String[] tokens) {
        Deque<Integer> stk = new ArrayDeque<>();
        for(String str : tokens){
            if(isNumber(str)){
                stk.push(Integer.valueOf(str));
                continue;
            }
            int n1 = stk.pop();
            int n2 = stk.pop();
            switch(str){
                case "+":
                    stk.push(n2 + n1);
                    break;
                case "-":
                    stk.push(n2 - n1);
                    break;
                case "*":
                    stk.push(n2 * n1);
                    break;
                case "/":
                    stk.push(n2 / n1);
                    break;
            }
```

```
29
30          }
31          return stk.pop();
32      }
33  }
```

3.根据**逆波兰表示法**，求表达式的值。有效的运算符包括 `+` , `-` , `*` , `/` 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。OJ 链接-逆波兰表达式求值

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

Tips

使用题目给的提示：适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

逆波兰表达式求值

```
1   class Solution {
2       private boolean isNumber(String str){
3           return !"+".equals(str) && !"-".equals(str) && !"*".equals(str) && !"/".
                equals(str);
4       }
5
6       public int evalRPN(String[] tokens) {
7           Deque<Integer> stk = new ArrayDeque<>();
8           for(String str : tokens){
9               if(isNumber(str)){
10                  stk.push(Integer.valueOf(str));
11                  continue;
12              }
13              int n1 = stk.pop();
14              int n2 = stk.pop();
15              switch(str){
16                  case "+":
17                      stk.push(n2 + n1);
18                      break;
19                  case "-":
20                      stk.push(n2 - n1);
21                      break;
22                  case "*":
23                      stk.push(n2 * n1);
24                      break;
25                  case "/":
26                      stk.push(n2 / n1);
27                      break;
28              }
29
30          }
31          return stk.pop();
32      }
```

```
}
```

4.给你一个整数数组 `nums` ，你需要找出一个**连续子数组**，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。请你找出符合题意的 最短 子数组，并输出它的长度。OJ 链接-最短无序连续子数组

Tips

1. 如果数组长度为 0 或长度为 1，则数组一定有序。
2. 对于长度大于 1 的数组：

符合题意的任意一个应该为：有序 1 + 无序 + 有序 2。

升序排序，所以无序中的最大值不会超过有序 2 中的最小值，无序中的最小值不会小于有序 1 中的最大值。

使用 `lo` 表示无序的左边界，使用 `ri` 表示无序的右边界。

从左至右找无序的右边界:将已经扫描过的最大元素记为 `max` ，显然有序 2 的所有元素都应该大于等于 `max` ,如果某个元素 `nums[i]<max` ,说明 `nums[i]` 不在有序 2 中，应该在无序中，把无序的右边界更新为 `i` 。

从右至左找左边界：与找右边界逻辑一致，将已经扫描过的最小元素记为 `min` ，有序 1 的所有元素都应该小于等于 `min` ，如果某个元素 `nums[j]>min` ，说明 `nums[j]` 不在有序 1 中，应该在无序中，把无序的左边界更新为 `j` 。

最短无序连续子数组

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        if(nums.length == 0 || nums.length == 1 ){
            return 0;
        }
        int lo = nums.length - 1;
        int hi = 0;
        int max = nums[0];
        int min = nums[nums.length - 1];
        for(int i = 0; i < nums.length; i++){
            if(nums[i] < max){
                hi = i;
            }else{
                max = nums[i];
            }

            if(nums[nums.length - 1 - i] > min){
                lo = nums.length - 1 - i;
            }else{
                min = nums[nums.length - 1 - i];
            }
        }
        return hi - lo < 0 ? 0 : hi - lo + 1;
    }
}
```

5.给定一组字符，使用原地算法将其压缩。压缩后的长度必须始终小于或等于原数组长度。数组的每个元素应该是长度为 1 的字符（不是 `int` 整数类型）。在完成原地修改输入数组后，返回数组的新长度。OJ 链接-压缩字符串

Tips

使用三个指针。

1. 指针 `i` 指向开头。第三个指针 `index` 从头开始。
2. `j` 指向 `i` 的下一个位置。
3. 如果指针 `j` 指向的字符等于指针 `i` 指向的字符，指针 `j` 向后移动，直到与指针 `i` 指向的字符不相等（或者到字符串的末尾）停下来。

4. 两种情况：

a. 若 j 在 i 下一位停下来，表示 i 指向的字符只有一个，不需要压缩，此时 j - i 为 1。

b. 若 j 在 i 的后边位置停下来（超过 1 位），假设 i 为 0， j 为 4，说明位置 0、1、2、3 的字符都相等，即有连续 j - i 个字符相等。

5. 在 index 处记录 i 处的字符，然后向右移动一位，如果 j - i 为 1，不需要压缩，如果 j - i 大于 1，题目给的字符数组的长度最大为 1000，可能出现连续字符个数最大为 1000，即 j - i 超过一位数，把该数转换为字符数组，然后把数的每一位依次填入 index 处（填一位 index 向右移动一位）（代码的 12-15 行）。

6. 上一步执行完后，将指针 i 移到指针 j 处，执行 2-5 步，直到全部字符串扫描结束。

压缩字符串

```
class Solution {
    public int compress(char[] chars) {
        int index = 0;
        int i = 0;
        int j = 1;
        while(i < chars.length && index < chars.length){
            while(j < chars.length && chars[j] == chars[i]){
                j++;
            }
            chars[index++] = chars[i];
            int sum = j - i;
            if(sum > 1 && index < chars.length){
                for(char ch : ("" + sum).toCharArray()){
                    chars[index++] = ch;
                }
            }
            i = j;
            j = i + 1;
        }
        return index;
    }
}
```

6.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。OJ 链接-验证回文串

Tips

1. 处理空字符串：是回文串

2. 只关注字母和数字，所以写一个私有方法来判断当前字符是否为字母/数字。

3. 使用左右指针指向字符串的开头和结尾。

4. 如果不是字母或数字，左指针右移，右指针左移。

5. 经过上一后左右指针指向的字符则为字母或数字。题目中不区分大小写，所以如果字符是大写字母，将其转换为小写字母 (也可以把小写的字符转为大写的)。

6. 因为字符串是不可变对象，所以使用字符来接收左右指针指向的字符，大写字符转小写：ch - 'A'+ 'a'，字符相加减本质上是其对应的 ASCII 码相加减，因此还需将其强制转换为字符。

7. 大写转为小写后，比较两个字符是否相等，如果不相等，返回 false 。如果相等，左指针左移，右指针右移。继续第 5 到第 7 步。

8. 若左指针和右指针重合，说明是回文串。

验证回文串

```
class Solution {
    private boolean isAlphaAndNum(char ch){
        if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' &&
            ch <= '9')){
            return true;
```

```
        }
        return false;
    }
    public boolean isPalindrome(String s) {
        if(s == null || " ".equals(s)){
            return true;
        }
        int lo = 0;
        int hi = s.length() - 1;
        char a;
        char b;
        while(lo < hi){
            if(!isAlphaAndNum(s.charAt(lo))){
                lo++;
            }
            if(!isAlphaAndNum(s.charAt(hi))){
                hi--;
            }
            if(lo < hi && isAlphaAndNum(s.charAt(lo)) && isAlphaAndNum(s.charAt(hi
                ))){
                a = s.charAt(lo);
                b = s.charAt(hi);
                if(a >= 'A' && a <= 'Z'){
                    a = (char)(a + 'a' - 'A');
                }
                if(b >= 'A' && b <= 'Z'){
                    b = (char)(b + 'a' - 'A');
                }
                if(a != b){
                    return false;
                }
                lo++;
                hi--;
            }
        }
        return true;
    }
}
```

7.字符串转换函数OJ 链接-字符串转换整数 atoi。

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数（类似 `C/C++` 中的 `atoi` 函数）。

函数 `myAtoi(string s)` 的算法如下：

- 读入字符串并丢弃无用的前导空格。
- 检查第一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正。
- 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
- 将前面步骤读入的这些数字转换为整数（即，”123” → 123，”0032” →32）。如果没有读入数字，则整数为 0 。必要时更改符号（从步骤 2 开始）。

- 如果整数数超过 32 位有符号整数范围 $[-2^{31},\ 2^{31}\ -1]$，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被固定为 -2^{31}，大于 $2^{31}\ -1$ 的整数应该被固定为 $2^{31}\ -1$。
- 返回整数作为最终结果。

注意：

- 本题中的空白字符只包括空格字符 `' '` 。
- 除前导空格或数字后的其余字符串外，请勿忽略 任何其他字符。

Tips

1. 对于空字符串和 `null`，直接返回 0;
2. 一般字符串:

依题意，首先扫描空格，当指针没有到达末尾且指向的字符为空格时，指针向右移动。

空格扫描完后，判断指针是否到达末尾，如果到达末尾，返回 0.

没有到达末尾,扫描下一个字符是否是 `+` 或 `-`,使用 `flag` 作为正负的标记(默认为 1),若扫描到 `+`，`flag` 不变,指针向后移动一位。若扫描到 `1`，`flag` 置为 `-1`，指针向后移动一位。

再次判断是否到达末尾，如果到达末尾，返回 0.

没有到达末尾，继续扫描:(使用 `ret` 存储扫描的数字，因为数字可能超过 `int` 的最大范围，所以 `ret` 为 `long` 类型)

如果是数字,将其放到 `ret` 的后边,即 `ret = ret * 10`,然后判断 `ret` 的大小,如果 `flag` 为 1,且 `ret` 超过整型的最大值,则返回 `Integer.MAX_VALUE`，如果 `flag` 为-1，且 `-ret` 小于整型的最小值，则返回 `Integer.MIN_VALUE`。

如果不是数字或者指针已经扫描到了末尾，退出循环。

如果 `ret` 没有超过整型的范围，返回 `flag * ret`。要注意类型的转换。

字符串转换整数 atoi

```java
class Solution {
    public int myAtoi(String s) {
        if(s == null || s.length() == 0){
            return 0;
        }
        int i = 0;
        long ret = 0;
        int flag = 1;
        while(i < s.length() && s.charAt(i) == ' '){
            ++i;
        }
        if(i >= s.length()){
            return 0;
        }
        if(s.charAt(i) == '-'){
            flag = -1;
            i++;
        }else if(s.charAt(i) == '+'){
            i++;
        }

        if(i >= s.length()){
            return 0;
        }

        while(i < s.length() && (s.charAt(i) >= '0' && s.charAt(i) <= '9')){
            int x = s.charAt(i) - '0';
            ret = ret * 10 + x;
```

```
            if(flag > 0 && ret > Integer.MAX_VALUE){
                return Integer.MAX_VALUE;
            }
            if(flag < 0 && -ret < Integer.MIN_VALUE){
                return Integer.MIN_VALUE;
            }
            ++i;
        }
        return flag > 0 ? (int)ret : -(int)ret;
    }
}
```

8.给你两个二进制字符串，返回它们的和（用二进制表示）。输入为 **非空** 字符串且只包含数字 `1` 和 `0`。OJ 链接-二进制求和

Tips

1. 首先将短的字符串用 0 将高位补齐。因为要对字符频繁的操作，所以使用可变字符序列对象 `StringBuilder`。可以先将题中的两个字符串翻转，在最短的字符串后补 0（使用 `append` 方法），最后计算结束再将字符串翻转。
2. 字符串补齐后，使用列竖式的方法计算每一位，用 `carry` 来表示进位，刚开始进位为 0。
3. 获取每一位的字符，因为只有 0 和 1，所以减去字符 0 即为数字 0 和 1。每一位的和为 `carry + a对应位置的值 + b对应位置的值`，记为 `count`，因为是 2 进制，所以 `count` 大于等于 2 时，要进行进位，即 `carry` 置为 1，同时该位置的 `count%2`。
4. 计算结束后，要判断 `carry` 的值，如果这个值是 1，说明还需要进一位。
5. 别忘了翻转和返回值的类型。

二进制求和

```
class Solution {
    public String addBinary(String a, String b) {
        //求两个字符串的长度
        int a1 = a.length();
        int b1 = b.length();
        //求字符串的最大长度
        int max = a1 >= b1 ? a1 : b1;
        //将字符串存入可变字符序列并进行翻转
        StringBuilder stringBuilder1 = new StringBuilder(a).reverse();
        StringBuilder stringBuilder2 = new StringBuilder(b).reverse();
        //对短字符串进行补0
        while(stringBuilder1.length() < max){
            stringBuilder1.append("0");
        }
        while(stringBuilder2.length() < max){
            stringBuilder2.append("0");
        }
        //进位，初始为0
        int carry = 0;

        int x = 0;
        int y = 0;
        int count = 0;
        //存放相加的结果
        StringBuilder ret = new StringBuilder();
        for(int i = 0; i < max; i++){
```

```
            x = stringBuilder1.charAt(i) - '0';
            y = stringBuilder2.charAt(i) - '0';
            //每一位结果等于进位加各数
            count = x + y + carry;
            //二进制，结果超过2，就需要进位
            if(count >= 2){
                carry = 1;
                ret.append(count % 2);
            }else{
                carry = 0;
                ret.append(count);
            }
        }
        //最高位的进位判断
        if(carry == 1){
            ret.append("1");
        }
        //别忘了翻转，注意方法的返回类型
        return ret.reverse().toString();
    }
}
```

9.给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 的那 两个整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。你可以按任意顺序返回答案。要求时间复杂度 $O(n)$, 当然也可以选择使用暴力的 $O(n^2)$ 的解法。OJ 链接-两数之和

Tips

1. 从左至右遍历数组。
2. 一个指针指向当前元素，另一个指针从当前元素的下一位至数组最后遍历，寻找与当前元素的和等于目标值的元素。找到后返回这两个指针对应的下标值。

两数之和

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

10.给你一个非空数组，返回此数组中 第三大的数。如果不存在，则返回数组中最大的数。OJ 链接-第三大的数

Tips

1. 数组长度为 1，直接返回该元素。
2. 数组长度为 2，返回两个中最大的那个。

3. 数组长度大于等于 3：
题目中数的范围为 int 的范围，所以数组中第三大的元素可能是 int 的最小值（ Integer.MIN_VALUE ）。

使用三个数 max1 、max2 、max3 分别表示数组中第一大，第二大，第三大的元素，将其初始化为 int 的最小值减一（即 Integer.MIN_VALUE - 1 ，把这个值记为 min ），因为超出了 int 的范围，所以：类型为 long 。

遍历数组，对于任意元素：
如果该元素大于 max1 ，就把 max2 的值赋给 max3 ，把 max1 的值赋给 max2 ，把该元素的值赋给 max1 ；
如果该元素等于 max1 ，检查下一个元素，即 continue ；
如果该元素小于 max1 ，则将该元素与 max2 比较：
如果该元素大于 max2 ，就把 max2 的值赋给 max3 ，把该元素的值赋给 max2 ；
如果该元素等于 max2 ，检查下一个元素，即 continue ；
如果该元素小于 max2 ，则将该元素与 max3 比较：
只有该元素大于 max3 时，才将该元素的值赋给 max3 ，否则不操作。
遍历完数组后，看 max3 的值是否为 min ，如果是，说明没有第 3 大的数，返回 max1 ，否则返回 max3 。因为其类型为 long ，还需要进行类型转换。

第三大的数

```java
class Solution {
    public int thirdMax(int[] nums) {
        if(nums.length == 1){
            return nums[0];
        }
        if(nums.length == 2){
            return nums[0] > nums[1] ? nums[0] : nums[1];
        }
        long min = (long)Integer.MIN_VALUE - 1;
        long max1 = min;
        long max2 = max1;
        long max3 = max2;

        for(int i = 0; i < nums.length; i++){
            if(nums[i] > max1){
                max3 = max2;
                max2 = max1;
                max1 = nums[i];
            }else if(nums[i] == max1) {
                continue;
            }else{
                if(nums[i] > max2){
                    max3 = max2;
                    max2 = nums[i];
                }else if(nums[i] == max2){
                    continue;
                }else{
                    if(nums[i] > max3){
                        max3 = nums[i];
                    }
                }
            }
        }
        return max3 == min ? (int)max1 : (int)max3;
    }
}
```

11.给定一个由**整数**组成的 **非空** 数组所表示的非负整数，在该数的基础上加一。最高位数字存放在数组的首位，数组中每个元素只存储单个数字。你可以假设除了整数 0 之外，这个整数不会以零开头。(第一个数字是 9 或者最后一个是 9 要考虑进位)OJ 链接-加一

Tips

1. 将数组的最后一个元素加一；
2. 加一后只有两种情况，等于 10 或小于 10.
3. 等于 10，将其除 10 取余即为该位的值，同时将指针前移一位，将前一位加一，然后重复 3、4 步；
4. 小于 10，不需要进位，直接返回。
5. 对于 9,99,999…这种加一后，长度增加的，直接创建一个比原数组长度长 1 的数组，将新数组首位初始化为 1，其余保持不变（默认初始化为 0）。

加一

```
class Solution {
    public int[] plusOne(int[] digits) {
        for(int i = digits.length - 1; i >= 0; i--){
            digits[i]++;
            digits[i] = digits[i] % 10;
            if(digits[i] != 0){
                return digits;
            }
        }
        int[] arr = new int[digits.length + 1];
        arr[0] = 1;
        return arr;
    }
}
```

12.给定一个整数类型的数组 `nums`，请编写一个能够返回数组“中心索引”的方法。

我们是这样定义数组 中心索引 的：数组中心索引的左侧所有元素相加的和等于右侧所有元素相加的和。

如果数组不存在中心索引，那么我们应该返回 -1。如果数组有多个中心索引，那么我们应该返回最靠近左边的那一个。OJ 链接-寻找数组的中心索引

Tips

1. 计算出数组的和；
2. 从左至右遍历数组，计算出第 `i` 个元素左侧元素的和（包含该元素 `A[i]`）`sum1` 和右侧元素的和（包含该元素 `A[i]`）`sum2`，如果 `sum1` 和 `sum2` 相等，说明该位置为中心索引。
3. 遍历完依旧没有符合题意的返回 `-1`。

寻找数组的中心索引

```
class Solution {
    public int pivotIndex(int[] nums) {
        int sum1 = 0;
        for(int x : nums){
            sum1 += x;
        }
        int sum2 = 0;
        for(int i = 0; i < nums.length; i++){
            sum2 += nums[i];
            if(sum1 == sum2){
                return i;
```

```
            }
            sum1 = sum1 - nums[i];
        }
        return -1;
    }
}
```

13.给定一个非负的整数数组 `A`，返回一个数组，在该数组中， `A` 的所有偶数元素之后跟着所有奇数元素。OJ 链接-按奇偶排序数组

Tips

1. 使用左指针指向数组的左侧，右指针指向数组的右侧
2. 如果左指针指的是偶数，左指针向右移动，右指针指的是奇数，右指针向左移动
3. 经过上一步，左指针指向为奇数，右指针指向的为偶数，交换这两个指针指的元素的值。
4. 左指针在右指针的左侧时，重复第二步、第三部。

按奇偶排序数组

```
class Solution {
    public int[] sortArrayByParity(int[] A) {
        int lo = 0;
        int hi = A.length - 1;
        while(lo < hi){
            if(A[lo] % 2 == 0){
                lo++;
            }
            if(A[hi] % 2 == 1){
                hi--;
            }
            if(lo < hi && A[lo] % 2 == 1 && A[hi] % 2 == 0){
                int tmp = A[lo];
                A[lo] = A[hi];
                A[hi] = tmp;
            }
        }
        return A;
    }
}
```

14.给定一个字符串 S，返回“反转后的”字符串，其中不是字母的字符都保留在原地，而所有字母的位置发生反转。OJ 链接-仅仅反转字母

Tips

1. 使用两个指针分别指向字符串的前后。
2. 判断左右指针指的元素是否为字母，如果左指针指的不是字母，左指针向右移动，右指针指的不是字母，右指针向左移动。
3. 经过第二步后，两个指针指的为字母，如果两个指针指的字母相等，不用交换，如果不相等，交换，交换完毕，左指针向右移动一下，右指针向左移动一下。第二步第三部。
4. 如果左指针没有越过右指针，或左指针和右指针重合之前，重复

仅仅反转字母

```
class Solution {
    private boolean isAlpha(char ch){
        return ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'));
```

```
    }
    public String reverseOnlyLetters(String S) {
        int lo = 0;
        int hi = S.length() - 1;
        char[] ch = S.toCharArray();
        while(lo < hi){
            if(!isAlpha(S.charAt(lo))){
                lo++;
            }
            if(!isAlpha(S.charAt(hi))){
                hi--;
            }
            if(lo < hi && isAlpha(S.charAt(lo)) && isAlpha(S.charAt(hi))){
                if(S.charAt(lo) != S.charAt(hi)){
                    char tmp = ch[lo];
                    ch[lo] = ch[hi];
                    ch[hi] = tmp;
                }
                lo++;
                hi--;
            }
        }
        return new String(ch);
    }
}
```

15.给定一个按非递减顺序排序的整数数组 `A`，返回每个数字的平方组成的新数组，要求新数组也按非递减顺序排序。（注意：非递减顺序即递增，要注意原数组里的负数）OJ 链接-有序数组的平方

Tips

1. 类似于归并排序，使用两个指针指向数组的左边和右边。
2. 判断两个指针指向的元素的平方的大小，将大的那个的平方填入新数组的后边，即如果左指针指向的元素的平方大，就把该元素平方填到新数组的后边，然后左指针向右移动一位，新数组中的指针向前移动一位，以此类推。
3. 退出条件：左指针越过右指针，或者新数组的指针小于 0. 两者是等价的

有序数组的平方

```
class Solution {
    public int[] sortedSquares(int[] nums) {
        int[] arr = new int[nums.length];
        int lo = 0;
        int hi = nums.length - 1;
        int i = nums.length - 1;
        while(lo <= hi){//或者写成while(i >= 0)
            if(nums[lo] * nums[lo] < nums[hi] * nums[hi]){
                arr[i--] = nums[hi] * nums[hi];
                hi--;
            }else{
                arr[i--] = nums[lo] * nums[lo];
                lo++;
            }
        }
```

```
        return arr;
    }
}
```

16.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。OJ 链接-合并两个有序数组

Tips

方法 1：将两个数组合并，然后重新排序。

方法 2：依据题目描述，此题为归并排序算法的归并部分。

因为最终的结果要存放在 `nums1` 中，为了避免 `nums1` 中的数据被覆盖，使用一个数组 `arr` 来存放 `nums1` 的数据。

步骤：

1. 使用两个指针分别指向数组 `arr` 和 `nums2` 的开头，比较其对应元素的大小，将较小的那个元素放入到 `nums1` 中，同时将 `nums1` 和拿出元素的那个数组的指针向后移动一位。

2. 其中一个数组遍历完后，将没有遍历完的那个数组的剩余部分放在 `nums1` 后边即可。

合并两个有序数组

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int[] arr = new int[m];
        for(int i = 0; i < m; i++){
            arr[i] = nums1[i];
        }
        int a1 = 0;
        int a2 = 0;
        int i = 0;
        while(a1 < m && a2 < n){
            if(arr[a1] <= nums2[a2]){
                nums1[i++] = arr[a1++];
            }else{
                nums1[i++] = nums2[a2++];
            }
        }
        for(;i < m + n; i++){
            if(a1 < m){
                nums1[i] = arr[a1++];
            }
            if(a2 < n){
               nums1[i] = nums2[a2++];
            }
        }
    }
}
```

17.给定一个仅包含大小写字母和空格 `' '` 的字符串，返回其最后一个单词的长度。如果不存在最后一个单词，请返回 0。最后一个单词的长度

Tips

1. 字符串长度为 0 或者字符串引用指向为空，那最后一个单词长度为 0.

2. 一般情况：

用右指针 `hi` 从字符串的最后一位开始遍历，如果是空格，指针就向前移动，如果遇到了字母就停下来，记录该位置。当 `hi` 小于 0 时，说明字符串遍历完毕也没遇到字母，即不存在最后一个单词，返回 0.

如果 `hi` 不小于 0，记录 `hi` 当前的位置，使用一个新的指针 `lo` 从该位置向前遍历，直到遇到空格（或遍历字符串结束即 `lo` 小于 0）停下来，`lo` 和 `hi` 之间的字符个数就是最后一个单词的长度。

最后一个单词的长度

```java
class Solution {
    public int lengthOfLastWord(String s) {
        //char[] ch = s.toCharArray();
        int hi = s.length() - 1;
        if(s.length() == 0 || s == null){
            return 0;
        }
        while(hi >= 0 && s.charAt(hi) == ' '){
            hi--;
        }
        if(hi < 0){
            return 0;
        }
        int lo = hi;
        while(lo >= 0 && s.charAt(lo) != ' '){
            lo--;
        }
        return hi - lo;
    }
}
```

18.赎金信

给定一个赎金信（`ransom`）字符串和一个杂志（`magazine`）字符串，判断第一个字符串`ransom`能不能由第二个字符串`magazines`里面的字符构成。如果可以构成，返回`true`；否则返回`false`。

(题目说明：为了不暴露赎金信字迹，要从杂志上搜索各个需要的字母，组成单词来表达意思。杂志字符串中的每个字符只能在赎金信字符串中使用一次。)

假设两个字符串均只含有小写字母。

Tips

1. 杂志字符串长度小于赎金信字符串长度，一定不能构成赎金信。
2. 因为只有小写字母，所以分别统计各字符的个数，将两字符串各字符的个数放入数组中。
3. 分别比较两字符串对应字符的个数，赎金信相应字符的个数超过杂志该字符的个数，则一定不能表示，否则就能表示。

赎金信

```java
class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        if(ransomNote.length()>magazine.length()){
            return false;
        }
        char[] ch1 = ransomNote.toCharArray();
        char[] ch2 = magazine.toCharArray();
        int[] rans = new int[26];
        int[] maga = new int[26];
        for(char c1 : ch1){
            rans[c1 - 'a']++;
        }
        for(char c2 : ch2){
            maga[c2 - 'a']++;
        }
```

```
16          for(int i = 0; i < 26; i++){
17              if(rans[i] > maga[i]){
18                  return false;
19              }
20          }
21          return true;
22      }
23  }
```

19.回文数

Tips

1. 根据题意，所有负数都不是回文数。
2. 小于 10 的自然数都是回文数。
3. 能被 10 整除的都不是回文数。(能被 10 整除，最后一位是 0，而第一位不会是 0，所以不是回文数)
4. 其他的数字 `x`，每次从后边取一位 `x%10`，放入新的数 `num` 中：`num = num * 10 + x%10`，
5. 每取完一位，就将最后一位拿掉：`x = x / 10`。

判断 `x` 是不是小于 `num`，如果小于，说明已经过半了，即：

6. 如果 `x` 是奇数位，例如 5 位，取了 2 位后，`x` 还有 3 位，此时 `num` 是 2 位，`num < x`。再取一位，`x` 变成 2 位，`num` 变成 3 位，两位数小于三位数，此时说明取的位数超过一半，就不用取了，如果 `x` 和 `num` 的前两位 `num / 10` 相等，说明是回文数，否则不是。
7. 如果 `x` 是偶数位，例如 4 位，取了 2 位后，`num` 是 2 位，`x` 剩余 2 位，如果 `num` 等于 `x`，说明是回文数。

取一半数字进行判断

```
1   class Solution {
2       public boolean isPalindrome(int x) {
3           if(x < 0){
4               return false;
5           }
6           if(0 <= x && x < 10){
7               return true;
8           }
9           if(x % 10 == 0){
10              return false;
11          }
12          int num = 0;
13          while(x > num){
14              num = num * 10 + x % 10;
15              x /= 10;
16          }
17          return x == num || x == num / 10;
18      }
19  }
```

20.搜索插入位置

Tips

有序数组，采用二分查找。

1. 找到元素：返回元素的下标。
2. 如果数组中没有该元素。退出循环的前一次左指针、右指针、中间指针指向同一元素，若该元素小于目标值，即 `nums[mid] < target`，左指针指向中间指针的下一个元素，此时左指针在右指针的右边，退出循环。同时左指针指向的位置也是目标元素要插入的位置。
3. 若该元素大于目标值，即 `target < nums[mid]`，右指针指向中间指针的前一个元素，右指针在左指针的左边，退出循环。左指针的左侧元素必定小于目标值，而左指针指向的元素大于目标值，目标值应该插入到左指针指向的位置处。
4. 综上，找到目标值就返回该目标值对应的下标，没找到该目标值就返回左指针指向的位置。

```java
class Solution {
    public int searchInsert(int[] nums, int target) {
        if(nums.length == 0){
            return 0;
        }
        int lo = 0;
        int hi = nums.length - 1;
        int mid = (lo + hi)/2;
        while(lo <= hi){
            mid = (lo + hi)/2;
            if(nums[mid] < target){
                lo = mid + 1;
            }else if(target < nums[mid]){
                hi = mid - 1;
            }else{
                return mid;
            }
        }
        return lo;
    }
}
```

21.移除元素

Tips

1. 边界情况：数组为空时，要返回 0。
分析：不考虑顺序，返回的为不含 `val` 的前边一段，思路：把 `val` 换到后边去。
2. 左指针 `lo` 指向数组的左边，找 `val`，如果没找到 `val`，就把这个指针向右移动；
3. 右指针 `hi` 指向数组的右边，`val` 就应该位于数组后边，所以如果值是 `val`，指针就左移。
4. 经过 2、3 步后，左指针指的是 `val`，右指针指的是非 `val`，将这俩指针对应元素交换。
5. 左右指针重合时，数组就遍历完了。
6. 左指针的左边一定是不含 `val` 的，其左边有 `lo` 个元素，如果此时的左指针对应元素的值时 `val`，那就返回 `lo`，如果左指针对应元素的值不是 `lo`，就返回 `lo + 1`。

```java
class Solution {
    public int removeElement(int[] nums, int val) {
        if(nums.length == 0){
            return 0;
        }
        int lo = 0;
        int hi = nums.length - 1;
        int len = 0;
        while(lo < hi){
            if(nums[lo] != val && lo < hi){
                lo++;
            }
            if(nums[hi] == val && lo < hi){
                hi--;
            }
            if(lo < hi){
                int tmp = nums[lo];
                nums[lo] = nums[hi];
```

```
                nums[hi] = tmp;
            }
        }
        if(nums[lo] == val){
            return lo;
        }
        return lo + 1;
    }
}
```

22.旋转数组

Tips

在 C 语言做过这个题。

1. 要先把旋转的位置除数组的长度取余，因为向右移动数组长度个位置（或其倍数）等于没有移动。
2. 把整个数组旋转得到新的数组。
3. 依次旋转数组的前 `k` 个元素（ `0`到`k-1` ），和后边剩余的元素。

```
class Solution {
    public void rotate(int[] nums, int 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 lo,int hi){
        while(lo < hi){
            int tmp = nums[lo];
            nums[lo] = nums[hi];
            nums[hi] = tmp;
            lo++;
            hi--;
        }
    }
}
```

23.转换成小写字母

Tips

转大写：`str.toUpperCase()`

转小写：`str.toLowerCase()`

转换成小写字母

```
class Solution {
    public String toLowerCase(String str) {
        return str.toLowerCase();
    }
}
```

24.反转一个单链表

示例

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

Tips

1. 链表为空（`head == null`）或者链表只有一个元素（`head.next == null`）。反转后还是自身：`return head;`
2. 链表反转：首先用一个引用储存当前元素的下一个元素。`nextNode = curNode;` 然后把当前元素 `curNode` 的 `next` 指向它的前一个元素 `prevNode`：`curNode.next = prevNode`
3. 第 2 步执行完后，处理下一个元素：`prevNode = curNode;curNode = curNode.next`。
4. `curNode == null` 时，说明到了链表的末端（循环退出条件）。此时 `prevNode` 指向的是原链表的最后一个元素，即新链表的首节点：`return prevNode;`
5. 初始化：第一个元素没有前驱结点，反转后，原链表第一个节点为新链表的最后一个节点，新链表的最后一个节点的 `next` 应指向 `null`。所以 `prevNode` 初始化为 `null`：`Node prevNode = null;`，`curNode` 最开始应指向链表的首节点：`Node curNode = head;`。

迭代版本

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode prevNode = null;
        ListNode curNode = head;

        while(curNode != null){
            ListNode nextNode = curNode.next;
            curNode.next = prevNode;
            prevNode = curNode;
            curNode = nextNode;
        }
        return prevNode;
    }
}
```

Tips

1. 可以将问题分为一个头结点和链表的剩余部分。
2. 头结点为问题的平凡解：链表只有一个节点，直接返回。
3. 子问题的解：头节点（`head`）的下一个节点 `head.next` 的 `next` 指向头结点。
4. 头结点指向 `null`

递归版本

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

25.移除链表元素

删除链表中等于给定值 val 的所有节点。

示例

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

Tips

要考虑链表为空的情况。链表为空，直接返回 `null`

使用傀儡节点 `dummy` 来进行辅助删除。 `dummy.next = head`

两个指针，一个指向待删除元素（ `cur` ）。一个指向待删除元素的前一个位置（ `prev` ）。

如果找到了待删除元素:将该元素前一个位置(`pre`)的 `next` 指向删除元素的 `next` : `pre.next = cur.next` 。然后将 `cur` 复位,即指向 `pre.next` : `cur = pre.next`

如果没有找到，两个指针同时向后移动: `pre = pre.next;cur = cur.next;`

循环结束的条件: `cur` 指向空，表示已经找到了链表的最后。

最后返回 `dummy.next`

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if (head == null){
            return null;
        }
        ListNode dummy = new ListNode(0,head);
        ListNode prev = dummy;
        ListNode cur = dummy.next;
        while(cur != null){
            if(cur.val == val){
```

```
                prev.next = cur.next;
                cur = prev.next;
            }else{
                prev = prev.next;
                cur = cur.next;
            }
        }
        return dummy.next;
    }
}
```

26.杨辉三角

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

```
class Solution {
    public List<List<Integer>> generate(int numRows) {

        List<List<Integer>> pasTri = new ArrayList<>();
        if(numRows == 0){
            return pasTri;
        }
        List<Integer> first = new ArrayList<>();
        first.add(1);
        pasTri.add(first);
        if(numRows == 1){
            return pasTri;
        }
        List<Integer> second = new ArrayList<>();
        second.add(1);
        second.add(1);
        pasTri.add(second);
        if(numRows == 2){
            return pasTri;
        }
        for(int i = 3; i <= numRows; i++){
            List<Integer> list1 = new ArrayList<>();
            list1.add(1);
            for(int j = 2; j <= i - 1; j++){
                int a = pasTri.get(i - 1 - 1).get(j - 1);
                int b = pasTri.get(i - 1 - 1).get(j - 1 - 1);
                list1.add(a + b);
            }
            list1.add(1);
            pasTri.add(list1);
        }
        return pasTri;
    }
}
```

27.存在连续三个奇数的数组

给你一个整数数组 `arr`，请你判断数组中是否存在连续三个元素都是奇数的情况：如果存在，请返

回 `true`；否则，返回 `false`。

示例 1

```
输入：arr = [2,6,4,1]
输出：false
解释：不存在连续三个元素都是奇数的情况。
```

示例 2

```
输入：arr = [1,2,34,3,4,5,7,23,12]
输出：true
解释：存在连续三个元素都是奇数的情况，即 [5,7,23] 。
```

版本 1：LeetCode 提供的

```
class Solution {
    public boolean threeConsecutiveOdds(int[] arr) {
        for(int i = 0; i < arr.length - 2; i++){
            if((arr[i] % 2 == 1) && (arr[i+1] % 2 == 1) && (arr[i + 2] % 2 == 1)){
                return true;
            }
        }
        return false;
    }
}
```

版本 2：自己使用顺序表实现

```
public class MyArrayList1 {
    public boolean threeConsecutiveOdds(List<Integer> arr) {
        for (int i = 0; i < arr.size() - 2; i++) {
            if ((arr.get(i) % 2 == 1) && (arr.get(i + 1) % 2 == 1) && (arr.get(i +
                2) % 2 == 1)){
                return true;
            }
        }
        return false;
    }
}
```

28.员工的重要性

给定一个保存员工信息的数据结构，它包含了员工唯一的 id，重要度和直系下属的 id。比如，员工 1 是员工 2 的领导，员工 2 是员工 3 的领导。他们相应的重要度为 15, 10, 5。那么员工 1 的数据结构是 [1, 15, [2]]，员工 2 的数据结构是 [2, 10, [3]]，员工 3 的数据结构是 [3, 5, []]。注意虽然员工 3 也是员工 1 的一个下属，但是由于并不是直系下属，因此没有体现在员工 1 的数据结构中。

现在输入一个公司的所有员工信息，以及单个员工 id，返回这个员工和他所有下属的重要度之和。

```
示例 1:

输入: [[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1
```

```
输出: 11
解释:
员工1自身的重要度是5，他有两个直系下属2和3，而且2和3的重要度均为3。因此员工1的总重要度是 5 + 3 + 3 = 11。
注意:
```

一个员工最多有一个直系领导，但是可以有多个直系下属, 员工数量不超过 2000。

Tips

1. 遍历整个员工列表 employees，找到符合 id 的员工 employee
2. 这个员工 employee 如果没有下属 employee.subordinates.size()==0，重要度就是自己的重要度 employee.importance。
3. 如果这个员工有下属，算出每个下属及下属的重要度。

```
/*
// Definition for Employee.
class Employee {
    public int id;
    public int importance;
    public List<Integer> subordinates;
};
*/

class Solution {
    public int getImportance(List<Employee> employees, int id) {

        for(int i = 0;i < employees.size();i++){
            Employee employee = employees.get(i);
            if(employee.id == id){
                if(employee.subordinates.size() == 0){//没有下属
                    return employee.importance;
                }
                for(int j = 0; j < employee.subordinates.size(); j++){
                    employee.importance += getImportance(employees,employee.
                        subordinates.get(j));
                }
                return employee.importance;
            }
        }
        return 0;
    }
}
```