二分搜索代码：

```
public static int binarySearch(int s, int[] a) {
int l=0;
int h=a.length-1;
while(l<h) {
int middle = l + (h-l)/2;
if(a[middle] == s) {
return middle;
}else if(a[middle] < s) {
l = middle+1;
}else {
h = middle-1;
}
}
return -1;
}
```

小根堆：PriorityQueue<Integer> s = new PriorityQueue<>();默认小根堆

大根堆：PriorityQueue<Integer> s = new PriorityQueue<>((V1,V2)->V2-V1)

9.回文数

只需要拿到后面一半和前面一半比较即可，复杂度降低一半。奇数要去掉中间数字

```
class Solution {
public boolean isPalindrome(int x) {
if(x<0 || (x%10==0&&x!=0)){
return false;
        }
int reverse = 0;
while(x>reverse){
            reverse=reverse*10+x%10;
            x/=10;
        }
return x==reverse || x==reverse/10;
    }
}
```

11.盛水最多的容器

```
class Solution {
public int maxArea(int[] height) {
int left=0;
int right=height.length-1;
int max=0;
while(left<right){
int area=Math.min(height[left],height[right]) * (right-left);
            max=Math.max(max, area);
if(height[left] <= height[right]){
                ++left;
            }else{
                --right;
            }
        }
return max;
    }
}
```

13. 根据罗马符号算出罗马数字

```
class Solution {
public int romanToInt(String s) {
        Map<Character, Integer> lm = new HashMap<>();
        lm.put('I', 1);
        lm.put('V', 5);
        lm.put('X', 10);
        lm.put('L', 50);
        lm.put('C', 100);
        lm.put('D', 500);
        lm.put('M', 1000);
int sum = 0;
int pre = lm.get(s.charAt(0));
for(int i=1;i<s.length();i++){
int now=lm.get(s.charAt(i));
if(pre<now){
                sum -= pre;
            }else{
                sum += pre;
            }
            pre = now;
        }
        sum += pre;
return sum;
    }
}
```

14.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：strs = ["flower","flow","flight"]

输出："fl"

示例 2：

输入：strs = ["dog","racecar","car"]

输出：""

解释：输入不存在公共前缀。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/longest-common-prefix


```
class Solution {
public String longestCommonPrefix(String[] strs) {
if(strs.length==0){
return "";
      }
      String res = strs[0];
for(String s : strs){
if(res == "") return "";
while(!s.startsWith(res)){
            res = res.substring(0, res.length()-1);
        }
      }
return res;
   }
}
```

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：strs = ["flower","flow","flight"]

输出："fl"

15. 三数之和

```
class Solution {
public List<List<Integer>> threeSum(int[] nums) {
int n=nums.length;
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        Arrays.sort(nums);
for(int i=0;i<n;i++){
if(i>0&&nums[i] == nums[i-1]){
continue;
            }
int target = -nums[i];
int k=n-1;
for(int j=i+1;j<n;j++){
if(j>i+1 && nums[j] == nums[j-1]){
continue;
                }
while(j<k&&nums[j]+nums[k]>target){
                    k--;
                }
if(j==k){
```

```
break;
                }
if(nums[j]+nums[k]==target){
                    List<Integer> temp = new ArrayList<Integer>();
                    temp.add(nums[i]);
                    temp.add(nums[j]);
                    temp.add(nums[k]);
                    res.add(temp);
                }


            }
        }
return res;
    }
}
```

20.有效的括号

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

和 20 题类似的题，求括号的最大嵌套深度；

```
class Solution {
public int maxDepth(String s) {
int max = 0;
int num = 0;
for (int i = 0; i < s.length(); i++) {
char c = s.charAt(i);
if(c == '(') {
                num++;
            }else if(c == ')') {
                num--;
            }
            max = Math.max(max, num);
        }
return max;
    }
}
```

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

```
class Solution {
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
if(l1 == null){
return l2;
```

```
        }
if(l2 == null){
return l1;
        }
if(l1.val<l2.val){
            mergeTwoLists(l1.next, l2);
return l1;
        }else{
            mergeTwoLists(l1,l2.next);
return l2;
        }
       }
}
```

26. 去掉数组中重复数字
例如：1，1，2，2，3，3
用双指针 L 和 R，L 指向索引 0，R 指向索引 1;
当 L 和 R 指向的数字相同时，R 向前进一位找到下一位数字，还是重复的话继续上一步，找到不重复的数字时，L+1 指向的数字变为 R 指向的数字，然后 R++;

```
class Solution {
public int removeDuplicates(int[] nums) {
if(nums.length==0){
return 0;
        }
int l=0;
int r=1;
while(r<nums.length){
if(nums[l]==nums[r]){
                r++;
            }else{
                l++;
                nums[l]=nums[r];
                r++;
            }
        }
return ++l;

    }
}
```

27.删除数组中和目标值一样的元素。

```
class Solution {
public int removeElement(int[] nums, int val) {
if(nums.length==0 || nums==null){
return 0;
        }
int l=0;
for(int i=0; i<nums.length; i++){
```

```
if(nums[i]!=val){
                nums[l]=nums[i];
                l++;
            }
        }
return l;
    }
}
```

另外一种写法，效率也很高

```
class Solution {
public int removeElement(int[] nums, int val) {
if(nums.length==0 || nums==null){
return 0;
    }
int l=0;
int r=0;
while(r<nums.length){
if(nums[r] == val) {
          r++;
       }else{
          nums[l] = nums[r];
          l++;
          r++;
       }
    }
return l;
  }
}
```

28. 给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从 0 开始)。如果不存在，则返回 -1。

```
class Solution {
public int strStr(String haystack, String needle) {
int n = needle.length();
int h = haystack.length();
if(n == 0){return 0;}
for(int i=0;i<=h-n;i++){
if(haystack.charAt(i)==needle.charAt(0)){
int temp = i+1;
int j = 1;
while(j < n){
if(haystack.charAt(temp) == needle.charAt(j)){
                        temp++;
                        j++;
                    }else{
break;
                    }
                }
if(j==n){return i;}
            }
        }
```

```
return -1;
    }
}
```

35. 搜索插入位置，给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。你可以假设数组中无重复元素。

解法 1：暴力解法，如果 nums[i]等于 target,那返回 i 即可；

如果 nums[i]<target，将 target 往后移；直到 nums[i]>target, 将 nums[i]往后移，target 占 i 的位置。

如果一个循环下来没找到 i, 那就是 target 是最大值，放到整个数组的最后面。

```
class Solution {
public int searchInsert(int[] nums, int target) {
for(int i = 0; i < nums.length; i++){
if(nums[i]>=target){
return i;
        }
     }
return nums.length;
   }
}
```

解法 2：二分查找，遇到排序数组要很敏感地想到二分搜索的算法，如果用暴力解法，面对一个很长的数组，刚好 target 比所有数都大，时间复杂度会很大，产生很多不需要的遍历。

53.最大子列和

```
class Solution {
public int maxSubArray(int[] nums) {
int max=nums[0];
int sum=0;
for(int x : nums){
            sum=Math.max(sum+x, x);
            max=Math.max(max, sum);
        }
return max;
    }
}
```

58.给定一个仅包含大小写字母和空格 ' ' 的字符串 s，返回其最后一个单词的长度。如果字符串从左向右滚动显示，那么最后一个单词就是最后出现的单词。如果不存在最后一个单词，请

返回 0 。说明：一个单词是指仅由字母组成、不包含任何空格字符的 最大子字符串。

```
class Solution {
public int lengthOfLastWord(String s) {
int end = s.length()-1;
while(end>=0 && s.charAt(end)==' '){
        end--;
    }
if(end < 0) return 0;
int start = end;
while(start >= 0 && s.charAt(start) != ' ') start --;
return end - start;
}
}
```

62.一个机器人位于一个 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？

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

再优化一下：每一步只跟前两种可能有关，所以可以优化成二维数组。

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。

二分法

```
class Solution {
public int mySqrt(int x) {
int l = 0;
int r = x;
int res = -1;
if(x == 1 || x == 0) return x;
while(l<=r){
int mid = (l+r)/2;
if(mid <= x/mid){
            l = mid + 1;
            res = mid;
          }else {
            r = mid - 1;
          }
      }
return res;
   }
}
```

牛顿迭代

```
class Solution {

    public int mySqrt(int x) {

        if (x == 0) {

            return 0;

        }

        double C = x, x0 = x;

        while (true) {

            double xi = 0.5 * (x0 + C / x0);

            if (Math.abs(x0 - xi) < 1e-7) {

                break;

            }

            x0 = xi;

        }

        return (int) x0;
```

```
    }
}
```

83.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。

```
class Solution {
public ListNode deleteDuplicates(ListNode head) {
    ListNode temp = head;
while(temp!=null&&temp.next!=null){
if(temp.next.val == temp.val){
            ListNode node = temp.next;
            temp.next = node.next;
        }else {
            temp = temp.next;
        }
    }
return head;
  }
}
```

88.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。初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。你可以假设 nums1 的空间大小等于 m + n，这样它就有足够的空间保存来自 nums2 的元素。

运用双指针，从最后面开始循环，把 nums1 作为最终数组。比较两个数组不为零的最后一位，谁大就放到 nums1 的最后面取代最后一个数；然后移动指针

```
class Solution {
public void merge(int[] nums1, int m, int[] nums2, int n) {
int p1 = m-1;
int p2 = n-1;
int total = m+n-1;
while(p2>=0){
if(p1>=0 && nums1[p1]>nums2[p2]){
            nums1[total--] = nums1[p1--];
        }else{
            nums1[total--] = nums2[p2--];
        }
    }
  }
}
```

98.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。

递归 1：

```
long pre = Long.MIN_VALUE;
public boolean isValidBST(TreeNode root) {
if(root == null){
return true;
    }
if(!isValidBST(root.left)){
return false;
    }
if(root.val <= pre) return false;


    pre = root.val;
return isValidBST(root.right);
```

递归 2：

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


    pre = root;
return isValidBST(root.right);
```

100(树结构).给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
public boolean isSameTree(TreeNode p, TreeNode q) {
if(p==null && q==null) return true;
else if(p==null || q==null) return false;
else if(p.val != q.val) return false;
else{return isSameTree(p.left, q.left)&&isSameTree(p.right, q.right);
        }
    }
}
```

101.给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树

[1,2,2,3,4,4,3]
是对称的。

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

public boolean check(TreeNode node1, TreeNode node2){
if(node1 == null && node2 == null) return true;
if(node1 == null || node2 == null) return false;
if(node1.val != node2.val) return false;
return check(node1.left, node2.right) && check(node1.right, node2.left);
    }
}
```

102.给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

示例：

二叉树：[3,9,20,null,null,15,7],

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

返回其层序遍历结果：

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

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        Queue<TreeNode> quene = new LinkedList<TreeNode>();
if(root == null)
return res;
        quene.offer(root);
```

```
while(!quene.isEmpty()){
int count = quene.size();
        List<Integer> list = new ArrayList<Integer>();
for(int i = 1;i <= count; ++i){
            TreeNode temp = quene.poll();
            list.add(temp.val);
if(temp.left != null){
                quene.offer(temp.left);
            }
if(temp.right != null){
                quene.offer(temp.right);
            }
        }
        res.add(list);
    }
return res;
  }
}
```

104.给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

给定二叉树 [3,9,20,null,null,15,7]

递归版本，也可以用 DFS 和 BFS，但是效率不如递归

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

108.给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

class Solution {

```
    public TreeNode sortedArrayToBST(int[] nums) {
        return dfs(nums, 0, nums.length - 1);
    }

    private TreeNode dfs(int[] nums, int lo, int hi) {
        if (lo > hi) {
            return null;
        }
        // 以升序数组的中间元素作为根节点 root。
        int mid = lo + (hi - lo) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        // 递归的构建 root 的左子树与右子树。
        root.left = dfs(nums, lo, mid - 1);
        root.right = dfs(nums, mid + 1, hi);
        return root;
    }
}
```

110.

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

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 。

```
class Solution {
public boolean isBalanced(TreeNode root) {
return height(root) >= 0;
    }
private int height(TreeNode root){
```

```
if(root == null){
return 0;
    }
int left = height(root.left);
int right = height(root.right);
if(left >= 0 && right >= 0 && Math.abs(left - right ) <= 1){
return Math.max(left+1, right+1);
   }else{
return -1;
    }
    }
}
```

112.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）。

```
class Solution {
public boolean hasPathSum(TreeNode root, int targetSum) {
if(root == null) return false;
if(root.left == null && root.right == null){
return root.val == targetSum;
       }
return hasPathSum(root.left, targetSum - root.val) || hasPathSum(root.right, targetSum - root.val);
    }
}
```

118.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>> res = new ArrayList<List<Integer>>();
for(int i = 0;i < numRows;i++){
           List<Integer> temp = new ArrayList<Integer>();
for(int j = 0;j <= i; j++){
if(j == 0 || j == i){
                   temp.add(1);
               }else {temp.add(res.get(i-1).get(j)+res.get(i-1).get(j-1));
               }
           }
           res.add(temp);
       }
return res;
    }
}
```

119.给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。

```
class Solution {
public List<Integer> getRow(int rowIndex) {
        Integer dp[] = new Integer[rowIndex+1];
        Arrays.fill(dp,1);
for(int i = 2;i < dp.length;i++){
for(int j = i-1; j>0; j--){
                dp[j] = dp[j-1] + dp[j];
            }
        }
        List<Integer> res = Arrays.asList(dp);
return res;
    }
}
```

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

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

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

```
public class Solution {
public int maxProfit(int prices[]) {
int minPrice = Integer.MAX_VALUE;
int maxProfit = 0;
for(int i = 0;i < prices.length; i++){
if(prices[i] < minPrice){
                minPrice = prices[i];
            }else if(maxProfit < prices[i] - minPrice){
                maxProfit = prices[i] - minPrice;
            }


        }
return maxProfit;
    }
}
```

122 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）

贪心：

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

动态规划：

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

优化一下：

```
class Solution {
public int maxProfit(int[] prices) {
int n = prices.length;
int dp0 = 0, dp1 = -prices[0];
for (int i = 1; i < n; ++i) {
int newDp0 = Math.max(dp0, dp1 + prices[i]);
int newDp1 = Math.max(dp1, dp0 - prices[i]);
            dp0 = newDp0;
            dp1 = newDp1;
        }
return dp0;
    }
}
```

125.难度简单 335 收藏分享切换为英文接收动态反馈

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。

```
class Solution {
public boolean isPalindrome(String s) {
int left = 0;
int right = s.length()-1;
      s = s.toLowerCase();
while(left < right){
if(!Character.isLetterOrDigit(s.charAt(left))){
            left++;
continue;
          }
if(!Character.isLetterOrDigit(s.charAt(right))){
            right--;
continue;
          }
if(s.charAt(left)!=s.charAt(right)){
return false;
          }
          left++;
          right--;
      }
return true;
   }
}
```

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

进阶：你可以设计并实现时间复杂度为 O(n) 的解决方案吗？

```
class Solution {
public int longestConsecutive(int[] nums) {
      HashSet<Integer> set = new HashSet<>();
for(int x : nums){
          set.add(x);
      }
int res = 0;
for(int x:nums){
if(set.contains(x-1)){
continue;
          }else{
int len = 0;
while(set.contains(x++)){
                len++;
                res = Math.max(res, len);
```

```
            }
        }
    }
return res;
  }
}
```

136.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？

位运算：

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

141.给定一个链表，判断链表中是否有环。

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

如果链表中存在环，则返回 true 。 否则，返回 false 。

首先考虑哈希表，把每个节点加入进去，循环过后如果遇到一个节点加入不进去，那就是遇到重复的了。但是效率有点低

```
public class Solution {
public boolean hasCycle(ListNode head) {
    Set<ListNode> set = new HashSet<>();
while(head != null){
if(!set.add(head)){
return true;
        }
        head = head.next;
    }
```

```
return false;
    }
}
```

快慢指针，慢的走一步，快的走两布，如果有环，那快的会超慢的一圈，就会有一个时刻两个指针会相等。如果没有相等，那就没环

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

144 二叉树的前序遍历：.给你二叉树的根节点

root

，返回它节点值的 前序 遍历。

递归版本：

```
class Solution {
public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        preorder(root, res);
return res;
    }


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

迭代版本：

```
class Solution {
public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
if(root == null) return res;
        Stack<TreeNode> temp = new Stack<>();
```

```
        TreeNode node = root;
while(node != null || !temp.isEmpty()){
while(node != null){
            res.add(node.val);
            temp.push(node);
            node = node.left;
        }
        node = temp.pop();
        node = node.right;
        }
return res;
}
}
```

99.二叉树的中序遍历

递归：

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

迭代：

```
class Solution {
public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        Deque<TreeNode> stk = new LinkedList<TreeNode>();
while (root != null || !stk.isEmpty()) {
while (root != null) {
                stk.push(root);
                root = root.left;
            }
            root = stk.pop();
            res.add(root.val);
            root = root.right;
        }
return res;
    }
}
```

145 二叉树的后序遍历

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

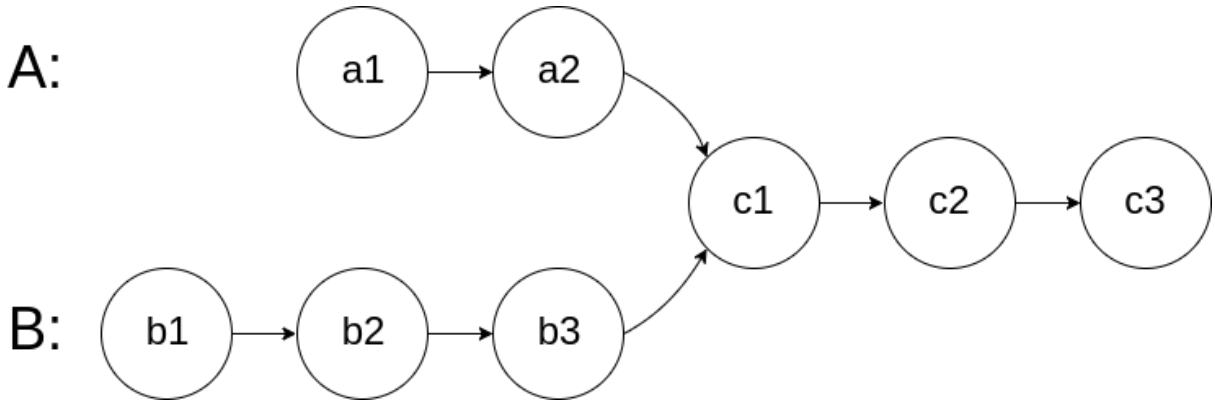
迭代：

```
class Solution {
public List<Integer> postorderTraversal(TreeNode root) {
     LinkedList<Integer> res = new LinkedList<>();
     Stack<TreeNode> temp = new Stack<>();
while(root != null || !temp.isEmpty()){
while(root != null){
            temp.push(root);
            res.addFirst(root.val);
            root = root.right;
         }
          root = temp.pop();
          root = root.left;
     }
return res;
   }
   }
```

160.

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：



在节点 c1 开始相交。

解法也是双指针，如果有环，两个指针肯定有相交的时候

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

167. 给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

返回的下标值（index1 和 index2）不是从零开始的。

你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

```
class Solution {
public int[] twoSum(int[] nums, int target) {
int l = 0;
int r = nums.length - 1;
while(l<r){
if(nums[l] + nums[r] == target){
return new int[]{l+1,r+1};
            }else if(nums[l] + nums[r] > target){
                r--;
            }else {
                l++;
            }
        }
return new int[0];
    }
}
```

172.

给定一个整数 n，返回 n! 结果尾数中零的数量。

综上，规律就是每隔 5 个数，出现一个 5，每隔 25 个数，出现 2 个 5，每隔 125 个数，出现 3 个 5...

以此类推。

最终 5 的个数就是 n / 5 + n / 25 + n / 125 ...

写程序的话，如果直接按照上边的式子计算，分母可能会造成溢出。所以算 n / 25 的时候，我们先把 n 更新，n = n / 5，然后再计算 n / 5 即可。后边的同理。

```
class Solution {
public int trailingZeroes(int n) {
int res = 0;
while(n > 0){
        res += n/5;
        n = n/5;
    }
return res;
  }
}
```

换一种方式：

```
class Solution {
public int trailingZeroes(int n) {
int res = 0;
int divisor = 5;
while(n/divisor >= 1){
        res += n/divisor;
        divisor*=5;
    }
return res;
  }
}
```

191.编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

```
public class Solution {
// you need to treat n as an unsigned value
public int hammingWeight(int n) {
int sum = 0;
while (n != 0) {
    sum++;
    n &= (n - 1);
  }
return sum;
}
}
```

202.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 。

```
class Solution {
public int getnext(int n){
int sum = 0;
while(n>0){
int d = n%10;
        n/=10;
        sum += d*d;
    }
return sum;
  }
public boolean isHappy(int n) {
int slow = n;
int fast = getnext(n);
while(slow!=fast){
        slow = getnext(slow);
        fast = getnext(fast);
        fast = getnext(fast);
    }
return slow==1;
  }
}
```

204. 统计所有小于非负整数 n 的质数的数量。

```
class Solution {
public int countPrimes(int n) {
int count=0;
boolean prime[] = new boolean[n];
        Arrays.fill(prime, true);


//从 2 开始
for(int i=2;i*i<n;i++){
if(prime[i]){
for(int j=i+i;j<n;j=j+i){
                    prime[j]=false;
                }
            }
        }
for(int z=2;z<prime.length;z++){
if(prime[z]==true){
                count++;
            }
```

```
        }
return count;
    }
}
```

198.打家劫舍（动态规划）

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

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

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

优化一下内存：

```
class Solution {
public int rob(int[] nums) {
if(nums == null || nums.length == 0){
return 0;
      }
int pre = 0;
int cur = 0;
for(int i : nums){
int temp = Math.max(cur, pre + i);
          pre = cur;
          cur = temp;
      }
return cur;
   }
}
```

209. 给定一个含有 n 个正整数的数组和一个正整数 s ， 找出该数组中满足其和 ⩾ s 的长度最小的连续 子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

```
class Solution {
public int minSubArrayLen(int s, int[] nums) {
if(nums.length == 0){
return 0;
        }
int sum = 0;
int l = 0;
int r = 0;
int min = Integer.MAX_VALUE;
while(r < nums.length){
            sum += nums[r];
while(sum >= s){
                min = Math.min(min, r-l+1);
                sum -= nums[l];
                l++;
            }
            r++;
        }
return min==Integer.MAX_VALUE? 0 : min;
    }
}
```

215.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。

```
class Solution {
public int findKthLargest(int[] nums, int k) {
      PriorityQueue<Integer> que = new PriorityQueue<>();
for(int x : nums){
         que.offer(x);
if(que.size() > k){
              que.poll();
              }


        }
return que.peek();
    }
}
```

或者用快速选择来做：快速选择是快速排序的另一种形式

```
class Solution {
public int findKthLargest(int[] nums, int k) {
int lo = 0, hi = nums.length - 1;
// 索引转化
   k = nums.length - k;
while (lo <= hi) {
// 在 nums[lo..hi] 中选一个分界点
int p = partition(nums, lo, hi);
if (p < k) {
```

```
// 第 k 大的元素在 nums[p+1..hi] 中
            lo = p + 1;
        } else if (p > k) {
// 第 k 大的元素在 nums[lo..p-1] 中
            hi = p - 1;
        } else {
// 找到第 k 大元素
return nums[p];
        }
    }
return -1;
        }
public static int partition(int a[], int left, int right) {
int pivot = a[left];
int i = left + 1;
int j = right;
while(true) {
while(i <= j && a[i] <= pivot) i++;
while(i <= j && a[j] >= pivot) j--;
if(i >= j) break;
int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
        a[left] = a[j];
        a[j] = pivot;
return j;
    }
}
```

263.编写一个程序判断给定的数是否为丑数。

丑数就是只包含质因数

2, 3, 5
的正整数。

```
class Solution {
public boolean isUgly(int x) {
if(x<1) return false;
while(x % 2 == 0){x/=2;}
while(x % 3 == 0){x/=3;}
while(x % 5 == 0){x/=5;}
return x==1;
    }
}
```

217.给定一个整数数组，判断是否存在重复元素。

如果存在一值在数组中出现至少两次，函数返回

true

。如果数组中每个元素都不相同，则返回

false

。

```
class Solution {
public boolean containsDuplicate(int[] nums) {
        HashSet<Integer> res = new HashSet<>();
for(int x : nums){
if(!res.add(x)) return true;
        }
return false;
    }
}
```

219.给定一个整数数组和一个整数 k，判断数组中是否存在两个不同的索引 i 和 j，使得 nums [i] = nums [j]，并且 i 和 j 的差的 绝对值 至多为 k。

```
class Solution {
public boolean containsNearbyDuplicate(int[] nums, int k) {
        Set<Integer> set = new HashSet<>();
for(int i = 0;i < nums.length; i++){
if(set.contains(nums[i])){
return true;
            }
            set.add(nums[i]);
if(set.size() > k){
                set.remove(nums[i-k]);
            }
        }
return false;
    }
}
```

264.编写一个程序，找出第 n 个丑数。用三指针，模拟手算的结果

```
class Solution {
public int nthUglyNumber(int n) {
int i2=0;int i3=0;int i5=0;
int [] dp = new int[n];
```

```
        dp[0] = 1;
for(int i = 1; i < n; i++){
            dp[i] = Math.min(Math.min(2*dp[i2], 3*dp[i3]), 5*dp[i5]);
if(dp[i] == 2*dp[i2]) i2++;
if(dp[i] == 3*dp[i3]) i3++;
if(dp[i] == 5*dp[i5]) i5++;
    }
return dp[n-1];
    }
}
```

小根堆，但是效率有点低

```
class Solution {
public int nthUglyNumber(int n) {
long res = 1;
    Queue<Long> queue = new PriorityQueue<>();
while (n-- > 1) {
        queue.offer(res * 2);
        queue.offer(res * 3);
        queue.offer(res * 5);
        res = queue.poll();
while (!queue.isEmpty() && res == queue.peek()) queue.poll();
    }
return (int) res;
}
}
```

283.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]

```
class Solution {
public void moveZeroes(int[] nums) {
int l=0;
int r=0;
while(r<nums.length){
if(nums[r] == 0) {
            r++;
        }else{
            nums[l] = nums[r];
            l++;
            r++;
        }
    }
for(;l<nums.length;l++){
```

```
            nums[l]=0;
        }
    }
}
```

292. 你和你的朋友，两个人一起玩 Nim 游戏：

桌子上有一堆石头。

你们轮流进行自己的回合，你作为先手。

每一回合，轮到的人拿掉 1 - 3 块石头。

拿掉最后一块石头的人就是获胜者。

假设你们每一步都是最优解。请编写一个函数，来判断你是否可以在给定石头数量为 n 的情况下赢得游戏。如果可以赢，返回 true；否则，返回 false 。

```
publicbooleancanWinNim(int n){
        return (n % 4 != 0);
 }
```

322.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。

你可以认为每种硬币的数量是无限的。

```
class Solution {
public int coinChange(int[] coins, int amount) {
if(coins.length == 0)
return -1;
int dp[] = new int[amount + 1];
        dp[0] = 0;
for(int i = 1; i <= amount; i++){
int min = Integer.MAX_VALUE;
for(int j = 0; j < coins.length; j++){
if(i - coins[j] >= 0 && dp[i-coins[j]] < min){
                    min = dp[i-coins[j]]+1;
                }
            }
            dp[i] = min;
        }
return dp[amount] == Integer.MAX_VALUE? -1 : dp[amount];
    }
```

```
}
```

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

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

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

```
class Solution {
public int fib(int n) {
if(n<=1){
return n;
        }
int s1 = 0;
int s2 = 1;
for(int i=2;i<=n;i++){
int temp = s1+s2;
            s1 = s2;
            s2 = temp;
        }
return s2;
    }
}
```

递归法：

```
class Solution {
public int fib(int n) {
while(n<=1){
return n;
        }
return fib(n-1)+fib(n-2);
    }
}
```

724.给你一个整数数组 nums，请编写一个能够返回数组 “中心下标” 的方法。

数组 中心下标 是数组的一个下标，其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下标，返回 -1 。如果数组有多个中心下标，应该返回最靠近左边的那一个。

方法 1：相当于双指针

```
class Solution {
public int pivotIndex(int[] nums) {
int sum = 0;
for(int x : nums){
            sum += x;
        }
int half = 0;
for(int i = 0; i < nums.length; i++){
            half += nums[i];
if(half == sum) return i;
            sum = sum - nums[i];
        }
return -1;
    }
}
class Solution {
public int pivotIndex(int[] nums) {
int sum = 0;
for(int x : nums){
            sum += x;
        }
int half = 0;
for(int i = 0; i < nums.length; i++){
if(half * 2 + nums[i] == sum) return i;
            half += nums[i];
        }
return -1;
    }
}
```

求出大于或等于 N 的最小回文素数。

回顾一下，如果一个数大于 1，且其因数只有 1 和它自身，那么这个数是素数。

例如，2，3，5，7，11 以及 13 是素数。

回顾一下，如果一个数从左往右读与从右往左读是一样的，那么这个数是回文数。

例如，12321 是回文数。

```
class Solution {
public int primePalindrome(int N) {
while(true){
            String digit= String.valueOf(N);
if(digit.length()%2 == 0 && N>11){
                N=(int)Math.pow(10,digit.length())+1;
            }

if(isPalindrome(N)&&isPrime(N)){
return N;
            }
            N++;
        }
    }

public boolean isPalindrome(int x){
int rev=0;
while(x>rev){
            rev=rev*10+x%10;
            x/=10;
        }
return x==rev || x==rev/10;
    }

public boolean isPrime(int N) {
if (N < 2) return false;
int R = (int) Math.sqrt(N);
for (int d = 2; d <= R; ++d)
if (N % d == 0) return false;
return true;
    }

}
```

```
class Solution {
public int findRepeatNumber(int[] nums) {
int repeat = -1;
        Set<Integer> s = new HashSet<Integer>();
for(int n : nums){
```

```
if(s.contains(n)){
                repeat = n;
break;
            }
            s.add(n);
        }
return repeat;
    }
}
```

```
class Solution {
public int findRepeatNumber(int[] nums) {
int temp = 0;
for(int i = 0; i<nums.length;i++){
while(nums[i]!=i){
if(nums[i]==nums[nums[i]]){
return nums[i];
                }
                temp=nums[i];
                nums[i]=nums[temp];
                nums[temp]=temp;
            }
        }
return -1;
    }
}
```

1004.给定一个由若干 0 和 1 组成的数组

A

，我们最多可以将 k 个值从 0 变成 1 。

返回仅包含 1 的最长（连续）子数组的长度。

```
class Solution {
public int longestOnes(int[] A, int K) {
int n = A.length;
int left = 0, right = 0;
int zeros = 0;
int res = 0;
while(right < n){
if(A[right] == 0){
            zeros++;
        }
while(zeros > K){
if(A[left++] == 0){
                zeros--;
            }
        }
        res = Math.max(res, right - left + 1);
        right++;
    }
return res;
```

```
    }
}
```

剑指 OFFER40：最小的 K 个数

输入整数数组

arr

，找出其中最小的

k

个数。例如，输入 4、5、1、6、2、7、3、8 这 8 个数字，则最小的 4 个数字是 1、2、3、4。

```
// 保持堆的大小为 K，然后遍历数组中的数字，遍历的时候做如下判断：
// 1. 若目前堆的大小小于 K，将当前数字放入堆中。
// 2. 否则判断当前数字与大根堆堆顶元素的大小关系，如果当前数字比大根堆堆顶还大，这个数就直接跳过；
//反之如果当前数字比大根堆堆顶小，先 poll 掉堆顶，再将该数字放入堆中。
class Solution {
public int[] getLeastNumbers(int[] arr, int k) {
if (k == 0 || arr.length == 0) {
return new int[0];
}
// 默认是小根堆，实现大根堆需要重写一下比较器。
Queue<Integer> pq = new PriorityQueue<>((v1, v2) -> v2 - v1);
for (int num: arr) {
if (pq.size() < k) {
pq.offer(num);
} else if (num < pq.peek()) {
pq.poll();
pq.offer(num);
}
}
// 返回堆中的元素
int[] res = new int[pq.size()];
int idx = 0;
for(int num: pq) {
res[idx++] = num;
}
return res;
}
}

//快排变形：快速选择
public int[] getLeastNumbers(int[] arr, int k) {
if (k == 0) {
```

```
return new int[0];
} else if (arr.length <= k) {
return arr;
}
// 原地不断划分数组
partitionArray(arr, 0, arr.length - 1, k);
// 数组的前 k 个数此时就是最小的 k 个数，将其存入结果
int[] res = new int[k];
for (int i = 0; i < k; i++) {
res[i] = arr[i];
}
return res;
}

void partitionArray(int[] arr, int lo, int hi, int k) {
// 做一次 partition 操作
int m = partition(arr, lo, hi);
// 此时数组前 m 个数，就是最小的 m 个数
if (k == m) {
// 正好找到最小的 k(m) 个数
return;
} else if (k < m) {
// 最小的 k 个数一定在前 m 个数中，递归划分
partitionArray(arr, lo, m-1, k);
} else {
// 在右侧数组中寻找最小的 k-m 个数
partitionArray(arr, m+1, hi, k);
}
}

// partition 函数和快速排序中相同，具体可参考快速排序相关的资料
// 代码参考 Sedgewick 的《算法 4》
int partition(int[] a, int lo, int hi) {
int i = lo;
int j = hi + 1;
int v = a[lo];
while (true) {
while (a[++i] < v) {
if (i == hi) {
break;
}
}
while (a[--j] > v) {
if (j == lo) {
break;
}
}
```

```
if (i >= j) {
break;
}
swap(a, i, j);
}
swap(a, lo, j);

// a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
return j;
}

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

类似的题：找到一个数组中第 K 大的元素。

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。

```
class Solution {
public int findKthLargest(int[] nums, int k) {

int len = nums.length;
// 使用一个含有 len 个元素的最小堆，默认是最小堆，可以不写 lambda 表达式：
(a, b) -> a - b
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(len, (a, b) -> a - b)
;
for(int num:nums){
if(minHeap.size()<k){
                minHeap.offer(num);
            }else if(num>minHeap.peek()){
                minHeap.poll();
                minHeap.offer(num);
            }
        }
return minHeap.peek();


    }
}
```

剑指 OFFER57：输入一个正整数 target ， 输出所有和为 target 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

```
class Solution {
public int[][] findContinuousSequence(int target) {
        ArrayList<int[]> res = new ArrayList<>();
for(int i=2;i<=target;i++){
int temp = target-i*(i-1)/2;
if(temp<=0){
break;
            }
if(temp%i==0){
int a1=temp/i;
int a[]=new int[i];
for(int j=1;j<=i;j++){
                    a[j-1]=a1+j-1;
                }
                res.add(a);
            }
        }
        Collections.reverse(res);
return res.toArray(new int[res.size()][]);
    }
}
```