剑指Offer

001:三个线程循环打印数字

```
public class Test05 {
    public static void main(String[] args) throws InterruptedException{
        Thread t1 = new Thread(new MyThread1(0));
        Thread t2 = new Thread(new MyThread1(1));
        Thread t3 = new Thread(new MyThread1(2));
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();

    }
}

class MyThread1 implements Runnable{
    private static Object lock = new Object();
    private static int count = 0;
    int no;

    MyThread1(int no){
        this.no = no;
    }

    @Override
    public void run() {
        while (true){
            synchronized (lock){
                if(count>100){
                    break;
                }
                if(count%3==this.no){
                    System.out.println(this.no+"----->"+count);
                    count++;
                }else{
                    try{
                        lock.wait();
                    }catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
                lock.notifyAll();
            }
        }
    }
}
```

002: 死锁Demo

```
package ByteDance;

/**
 * 线程死锁
 */
public class Lock {
    private static Object resource1 = new Object();
    private static Object resource2 = new Object();

    public static void main(String[] args) {
        new Thread(()->{
            synchronized(resource1){
                System.out.println(Thread.currentThread()+"get resource1");
                try{
                    Thread.sleep(1000);

                }catch (InterruptedException e){
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread()+"waitting get
resource2");
                synchronized (resource2){
                    System.out.println(Thread.currentThread()+"get resource2");
                }
            }
        },"线程1").start();

        new Thread(()->{
            synchronized (resource2){
                System.out.println(Thread.currentThread()+"get resource2");
                try{
                    Thread.sleep(1000);

                }catch (InterruptedException e){
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread()+"waitting get
resource1");
                synchronized(resource1){
                    System.out.println(Thread.currentThread()+"get resource1");
                }
            }
        },"线程2").start();
    }

    //解决死锁方法---------TODO--------
            new Thread(() -> {
            synchronized (resource1) {   //和线程1获取资源顺序同步
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource2");
                synchronized (resource2) {  //和线程1获取资源顺序同步
```

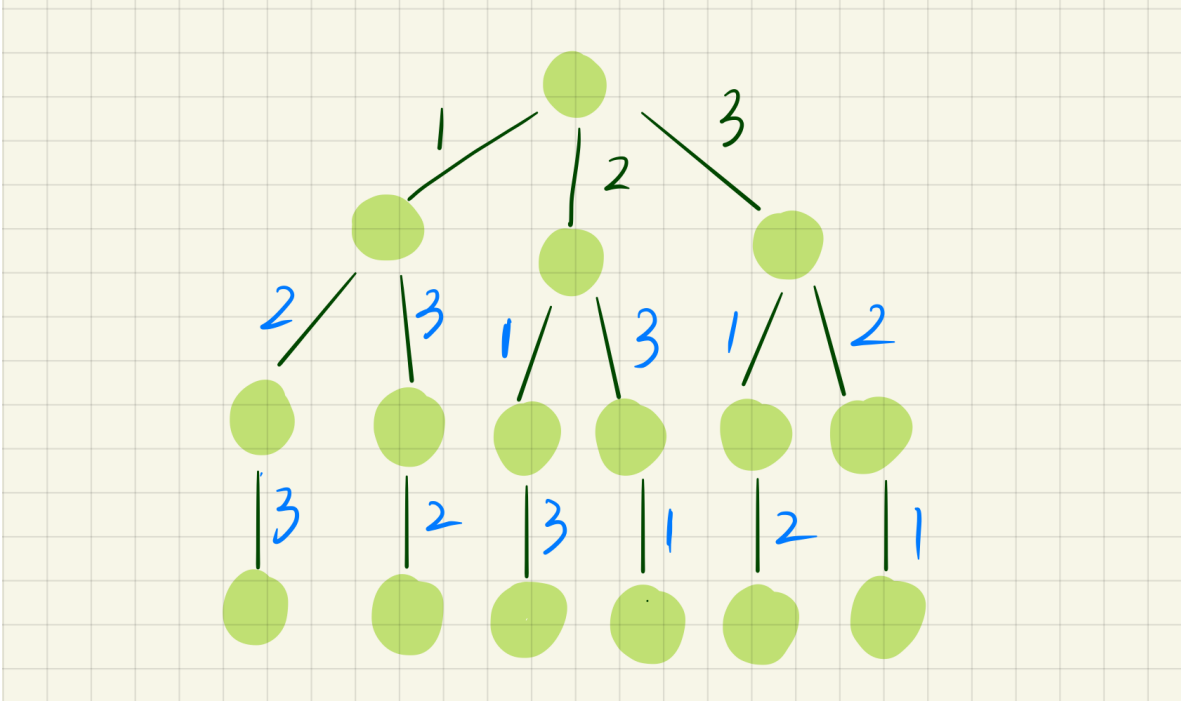
```
                    System.out.println(Thread.currentThread() + "get
resource2");
                }
            }
        }, "线程 2").start();
}
```

01：数组全排列(☆)

给定一个 **没有重复** 数字的序列，返回其所有可能的全排列。

示例:

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




```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> ans = new LinkedList<>();
        Queue<List<Integer>> queue = new LinkedList<>();
        queue.add(new LinkedList<Integer>());

        while(!queue.isEmpty()){
            List<Integer> list = queue.poll();
            int size = list.size();
            if(size == nums.length){
```

```
                ans.add(list);
                continue;
            }
            for(int i = 0; i<=nums.length-1;i++){
                if(!list.contains(nums[i])){
                    List<Integer> temp = new LinkedList<>(list);
                    temp.add(nums[i]);
                    queue.add(temp);
                }

            }
        }
        return ans;
    }
}
```

02：数组中重复的数字

题目描述

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

复制代码

```
Input:
{2, 3, 1, 0, 2, 5}

Output:
2
```

解题思路

要求时间复杂度 O(N)，空间复杂度 O(1)。因此不能使用排序的方法，也不能使用额外的标记数组。

对于这种数组元素在 [0, n-1] 范围内的问题，可以将值为 i 的元素调整到第 i 个位置上进行求解。

以 (2, 3, 1, 0, 2, 5) 为例，遍历到位置 4 时，该位置上的数为 2，但是第 2 个位置上已经有一个 2 的值了，因此可以知道 2 重复：

```
class Solution {
    public int findRepeatNumber(int[] nums) {
        for(int i=0;i<nums.length;i++){
            if(nums[i]!=i){
                if(nums[i]==nums[nums[i]]){
                    return nums[i];
                }
                int temp = nums[i];
                nums[i] = nums[temp];  //数组下标一定要用temp
                nums[temp] = temp;
            }
        }
        return -1;
    }
}
```

03：二维数组中的查找(☆)

题目描述

给定一个二维数组，其每一行从左到右递增排序，从上到下也是递增排序。给定一个数，判断这个数是否在该二维数组中。

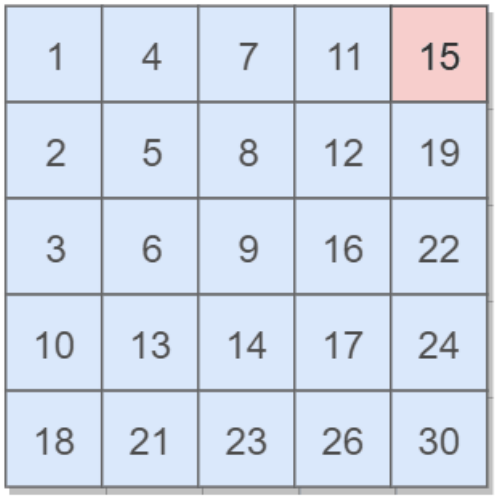
```
Consider the following matrix:
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]

Given target = 5, return true.
Given target = 20, return false.
```

复制代码

解题思路

要求时间复杂度 O(M + N)，空间复杂度 O(1)。其中 M 为行数，N 为 列数。

该二维数组中的一个数，小于它的数一定在其左边，大于它的数一定在其下边。因此，从右上角开始查找，就可以根据 target 和当前元素的大小关系来缩小查找区间，当前元素的查找区间为左下角的所有元素。



① target(16) > num(15)

CyC2018

```java
public boolean Find(int target, int[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
        return false;
    int rows = matrix.length, cols = matrix[0].length;
    int r = 0, c = cols - 1; // 从右上角开始
    while (r <= rows - 1 && c >= 0) {
        if (target == matrix[r][c])
            return true;
        else if (target > matrix[r][c])
```

```
            r++;
        else
            c--;
    }
    return false;
}
```

04：替换空格

题目描述

将一个字符串中的空格替换成 "%20"。

```
Input:
"A B"

Output:
"A%20B"
```

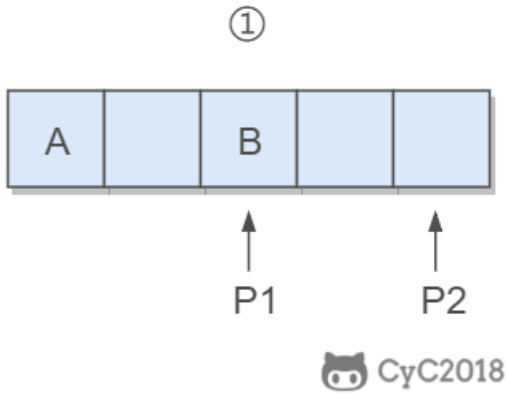
复制代码

解题思路

在字符串尾部填充任意字符，使得字符串的长度等于替换之后的长度。因为一个空格要替换成三个字符（%20），因此当遍历到一个空格时，需要在尾部填充两个任意字符。

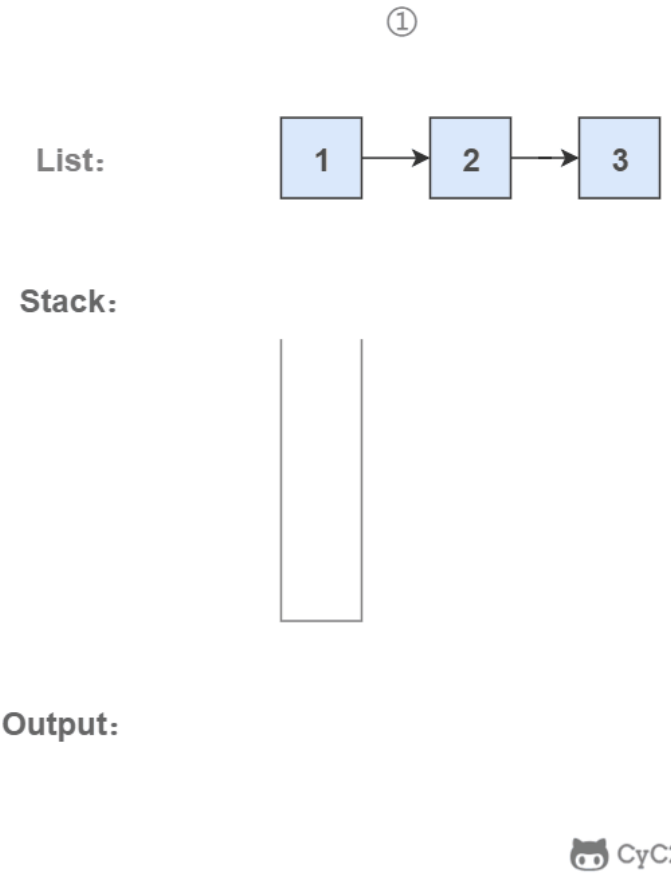
令 P1 指向字符串原来的末尾位置，P2 指向字符串现在的末尾位置。P1 和 P2 从后向前遍历，当 P1 遍历到一个空格时，就需要令 P2 指向的位置依次填充 02%（注意是逆序的），否则就填充上 P1 指向字符的值。

从后向前遍是为了在改变 P2 所指向的内容时，不会影响到 P1 遍历原来字符串的内容。




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

05：逆序打印链表（栈）



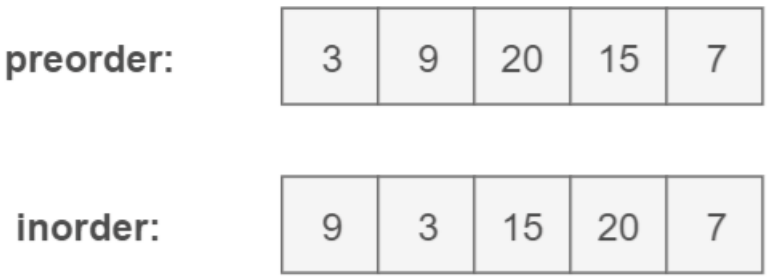

```
class Solution {
    public int[] reversePrint(ListNode head) {
        List<Integer> list=new ArrayList<>();
        while(head!=null){
            list.add(head.val);
            head=head.next;
        }
        int[] res=new int[list.size()];
        for(int i=0;i<res.length;i++)
            res[i]=list.get(list.size()-i-1);
        return res;
    }

}
```

06：重建二叉树（*TODO*😫）

题目描述

根据二叉树的前序遍历和中序遍历的结果，重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

①



CyC2018

```
class Solution {

    //key是中序遍历的值，value是中序遍历的结果
    HashMap<Integer,Integer> indexMap=new HashMap<>();

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        //保存中序遍历的信息
        for(int i=0;i<inorder.length;i++){
            indexMap.put(inorder[i],i);
        }
        return createTree(preorder,0,inorder,0,inorder.length-1);
    }

    //preIndex是前序遍历的索引，inStart和inEnd是中序遍历的索引范围
    private TreeNode createTree(int[] preorder,int preIndex,int[] inorder,int
inStart,int inEnd){
        if(inStart>inEnd)
            return null;
        //获取前序遍历的值
        int val=preorder[preIndex];
        //获取前序遍历值在中序遍历的位置
        int inIndex=indexMap.get(val);
        //以该值作为根节点的值创建根节点
        TreeNode root=new TreeNode(val);
        //根节点的左子树节点数目
        int leftNum=inIndex-inStart;
        //根节点以左创建左子树，根节点以右创建右子树
        root.left=createTree(preorder,preIndex+1,inorder,inStart,inIndex-1);

 root.right=createTree(preorder,preIndex+1+leftNum,inorder,inIndex+1,inEnd);
```

```
            return root;
        }
    }
```

07：二叉树的下一个节点

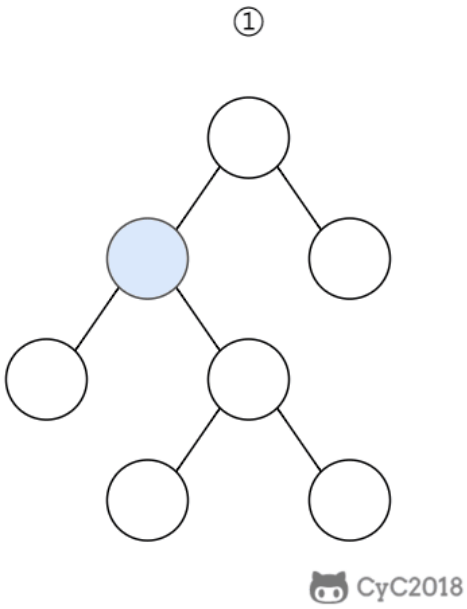
题目描述

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

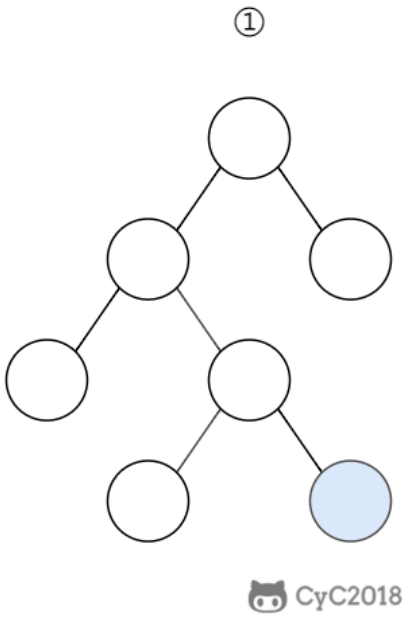
```
1  public class TreeLinkNode {
2
3      int val;
4      TreeLinkNode left = null;
5      TreeLinkNode right = null;
6      TreeLinkNode next = null;
7
8      TreeLinkNode(int val) {
9          this.val = val;
10     }
11 }
```

复制代码

1：如果一个节点的右子树不为空，那么下一个节点为右子树的最左节点



2：否则，向上找第一个祖先节点




```
public TreeLinkNode GetNext(TreeLinkNode pNode) {
    if (pNode.right != null) {
        TreeLinkNode node = pNode.right;
        while (node.left != null)
            node = node.left;
        return node;
    } else {
        while (pNode.next != null) {
            TreeLinkNode parent = pNode.next;
            if (parent.left == pNode)
                return parent;
            pNode = pNode.next;
        }
    }
    return null;
}
```

08：两个栈实现队列(☆)

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 `appendTail` 和 `deleteHead` ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，`deleteHead` 操作返回 -1)

```
class CQueue {
    Stack<Integer> stack1;
    Stack<Integer> stack2;
    public CQueue() {
        stack1 = new Stack<Integer>();
        stack2 = new Stack<Integer>();

    }

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

```
    }

    public int deleteHead() {
        if(stack2.isEmpty()){
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }

        if(stack2.isEmpty()){
            return -1;
        }

        int item = stack2.pop();
        return item;
    }
}

/**
 * Your CQueue object will be instantiated and called as such:
 * CQueue obj = new CQueue();
 * obj.appendTail(value);
 * int param_2 = obj.deleteHead();
 */
```

09：旋转数组的最小数字

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 `[3,4,5,1,2]` 为 `[1,2,3,4,5]` 的一个旋转，该数组的最小值为1。

示例 1：

```
输入：[3,4,5,1,2]
输出：1
```

复制代码

示例 2：

```
输入：[2,2,2,0,1]
输出：0
```

复制代码

思路和代码：

比较简单，我们从头遍历，如果一个数字大于它的下一个，就返回它的下一个。

例如示例1中，5>1，返回1，示例2中，2>0，返回0。

```
    public int minNumberInRotateArray(int [] array) {
        for(int i=0;i<array.length-1;i++){
            if(array[i]>array[i+1]){
                return array[i+1];
            }
        }
        return 0;
    }
```

10：青蛙跳台阶

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 `n` 级的台阶总共有多少种跳法。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

```
输入：n = 2
输出：2
```

复制代码

示例 2：

```
输入：n = 7
输出：21
```

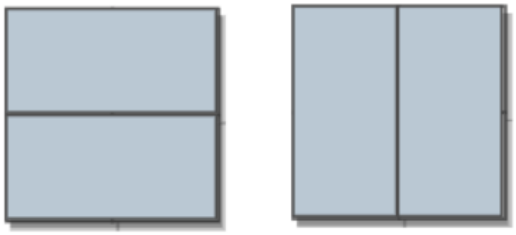
复制代码

提示：

```
    public int JumpFloor(int target) {
        int[] dp = new int[target+1];
        dp[0] = dp[1] = 1;
        for(int i=2;i<=target;i++){
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[target];
    }


//一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。
    public int JumpFloorII(int target) {
        int[] dp = new int[target];
        Arrays.fill(dp,1);
        for(int i=0;i<target;i++){
            for(int j=0;j<i;j++){
                dp[i] += dp[j];
            }
        }

        return dp[target-1];
    }
```

11：矩形覆盖

当 n 为 2 时，有两种覆盖方法：



n=2

CyC2018

要覆盖 2*n 的大矩形，可以先覆盖 2*1 的矩形，再覆盖 2*(n-1) 的矩形；或者先覆盖 2*2 的矩形，再覆盖 2*(n-2) 的矩形。而覆盖 2*(n-1) 和 2*(n-2) 的矩形可以看成子问题。该问题的递推公式如下：

$$f(n) = \begin{cases} 1 & n = 1 \\ 2 & n = 2 \\ f(n-1) + f(n-2) & n > 2 \end{cases}$$

```
    public int RectCover(int target) {
        if(target<=2){
            return target;
        }
        return RectCover(target-1)+RectCover(target-2);
    }
```

12：二进制中1的个数

题目描述

输入一个整数，输出该数二进制表示中 1 的个数。

n&(n-1)

该位运算去除 n 的位级表示中最低的那一位。

```
n       : 10110100
n-1     : 10110011
n&(n-1) : 10110000
```

时间复杂度：O(M)，其中 M 表示 1 的个数。

```
public int NumberOf1(int n) {
    int count = 0;
    while(n!=0){
        count ++;
        n = n&(n-1);
    }
    return count;
}
```

13：数值的整数次方(☆)

题目描述

给定一个 double 类型的浮点数 base 和 int 类型的整数 exponent，求 base 的 exponent 次方。

解题思路

下面的讨论中 x 代表 base，n 代表 exponent。

$$x^n = \begin{cases} (x*x)^{n/2} & n\%2 = 0 \\ x*(x*x)^{n/2} & n\%2 = 1 \end{cases}$$

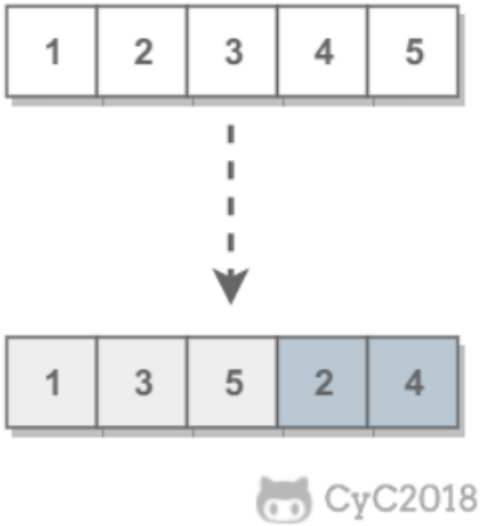
$$x^n = \begin{cases} (x*x)^{n/2} & n\%2 = 0 \\ x*(x*x)^{n/2} & n\%2 = 1 \end{cases}$$

```
    public double Power(double base, int exponent) {
        if(exponent == 0){
            return 1;
        }
        if(exponent == 1){
            return base;
        }

        boolean isNeg = false;

        if(exponent<0){
            isNeg = true;
            exponent = -exponent;
        }

        double ans = Power(base,exponent/2);
        if(exponent %2 == 0){
            ans = ans*ans;
        }else{
            ans = ans*ans*base;
        }

        return isNeg ? 1/ans:ans;
    }
```

14：调整数组顺序使奇数在前偶数在后

题目描述

需要保证奇数和奇数，偶数和偶数之间的相对位置不变，这和书本不太一样。

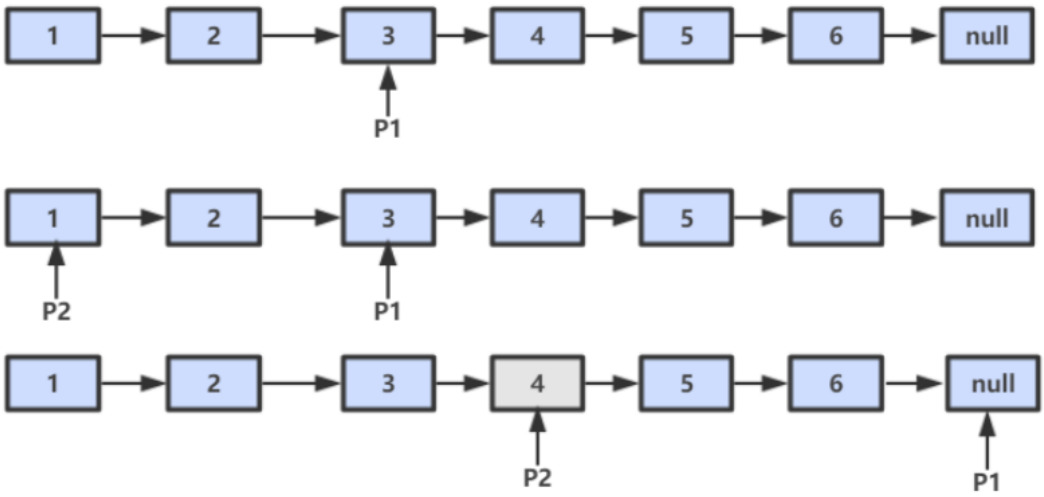



```
public void reOrderArray(int [] array) {
        if(array == null || array.length == 0){
            return;
        }
        int count = 0;  //奇数个数-------->谁在前统计谁的个数 否则会处理不了【1】数组长度为1的情况
        for(int num:array){
            if(num %2 == 1){
                count ++;
            }
        }

        int[] ans = array.clone();
        int i = 0;
        int j = count;
        for(int num:ans){
            if(num %2 == 1){
                array[i++] = num;
            }else{
                array[j++] = num;
            }
        }
    }
```

15：链表倒数第K个节点(☆)

解题思路

设链表的长度为 N。设置两个指针 P1 和 P2，先让 P1 移动 K 个节点，则还有 N - K 个节点可以移动。此时让 P1 和 P2 同时移动，可以知道当 P1 移动到链表结尾时，P2 移动到第 N - K 个节点处，该位置就是倒数第 K 个节点。




```
public ListNode FindKthToTail(ListNode head,int k) {
        if(head == null){
            return null;
        }

        ListNode p1 = head;
        while(p1!=null && k>0){
            k--;
            p1 = p1.next;
        }
        //k>列表长度
        if(k>0){
            return null;
        }

        ListNode p2 = head;
        while(p1!=null){
            p1 = p1.next;
            p2 = p2.next;
        }
        return p2;
    }
```

16：反转单链表(☆)

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:

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

复制代码

```
public ListNode ReverseList(ListNode head) {
    if(head == null || head.next==null){
```

```
        return head;
    }

    ListNode pre = head;
    ListNode cur = pre.next;
    pre.next = null;
    ListNode next = null;

    while(cur!=null){
        next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }

    return pre;
}
```

17：合并两个排序链表(☆)

输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。

示例1：

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

```
//方法1---->优先队列
public ListNode Merge(ListNode list1,ListNode list2) {
        PriorityQueue<ListNode> queue = new PriorityQueue<>(new
Comparator<ListNode>(){
            public int compare(ListNode o1,ListNode o2){
                return o1.val-o2.val;
            }
        });

        while(list1!=null){
            queue.add(list1);
            list1 = list1.next;
        }

        while(list2!=null){
            queue.add(list2);
            list2 = list2.next;
        }
        ListNode pre = new ListNode(-1);
        ListNode temp = pre;
        while(!queue.isEmpty()){
            temp.next = queue.poll();
            temp = temp.next;
        }
        return pre.next;
    }

//方法2----->双指针
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
```

```
    ListNode pre=new ListNode(-1);
    ListNode cur=pre;
    while(l1!=null&&l2!=null){
        if(l1.val<l2.val){
            cur.next=l1;
            l1=l1.next;
        }else{
            cur.next=l2;
            l2=l2.next;
        }
        cur=cur.next;
    }
    cur.next=l1==null?l2:l1;
    return pre.next;
}
```

18：合并K个有序链表

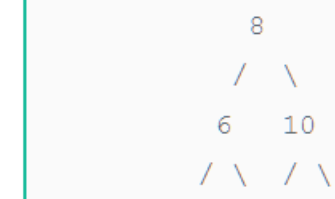
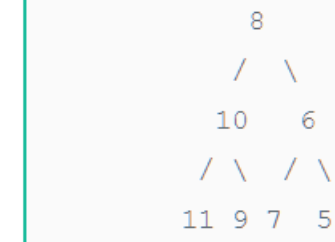
```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists==null || lists.length==0){
        return null;
    }
    int k = lists.length;
    PriorityQueue<ListNode> queue = new PriorityQueue<>(new Comparator<ListNode>
(){
        public int compare(ListNode o1,ListNode o2){
            //升序排序
            return (o1.val-o2.val);
        }
    });

    //把每个链表的数据放入queue
    for(int i=0;i<k;i++){
        ListNode head  = lists[i];
        while(head!=null){
            queue.add(head);
            head = head.next;
        }
    }

    ListNode temp = new ListNode(-1);
    ListNode head = temp;
    while(!queue.isEmpty()){
        temp.next = queue.poll();
        temp = temp.next;
    }
    temp.next = null;
    return head.next;
}
```

19：镜像二叉树(☆)

题目描述

操作给定的二叉树，将其变换为源二叉树的镜像。

输入描述:

> 二叉树的镜像定义：源二叉树
>
> 
>
> 镜像二叉树
>
> 

```
public void Mirror(TreeNode root) {
    if(root == null){
        return;
    }

    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);

    while(!queue.isEmpty()){
        TreeNode curNode = queue.poll();
        TreeNode temp = curNode.left;
        curNode.left = curNode.right;
        curNode.right = temp;

        if(curNode.left != null){
            queue.add(curNode.left);
        }
        if(curNode.right != null){
            queue.add(curNode.right);
        }
    }
}
```

20：顺时针打印矩阵

```
public ArrayList<Integer> printMatrix(int [][] matrix) {
    ArrayList<Integer> ans = new ArrayList<>();
    int RL = 0;
    int RH = matrix.length-1;

    int CL = 0;
```

```
    int CH = matrix[0].length-1;

    while(RL<=RH && CL<=CH){
        for(int i=CL;i<=CH;i++){
            ans.add(matrix[RL][i]);
        }
        for(int i=RL+1;i<=RH;i++){
            ans.add(matrix[i][CH]);
        }

        if(RL!=RH){
            for(int i=CH-1;i>=CL;i--){
                ans.add(matrix[RH][i]);
            }
        }

        if(CL!=CH){
            for(int i=RH-1;i>RL;i--){
                ans.add(matrix[i][CL]);
            }
        }
        RL++;
        RH--;

        CL++;
        CH--;

    }
    return ans;
}
```

21：最小栈(☆)

题目描述

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

```
public class Solution {
    Stack<Integer> dataStack = new Stack<>();
    Stack<Integer> minStack = new Stack<>();

    public void push(int node) {
        dataStack.push(node);
        if(minStack.isEmpty()){
            minStack.push(node);
        }else{
            minStack.push(Math.min(node,minStack.peek()));
        }
    }

    public void pop() {
        dataStack.pop();
        minStack.pop();
    }
```

```
    public int top() {
        return dataStack.peek();
    }

    public int min() {
        return minStack.peek();
    }
}
```

22：栈的压入、弹出(☆)

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

利用一个辅助栈来模拟，停止条件是压栈序列全部入栈，然后入栈时如果栈顶和出栈序列的当前元素相同就出栈一个，直到栈空或栈顶不等于出栈序列当前元素。

最后判断栈是否空，如果空代表入栈出栈有效。

```
public boolean IsPopOrder(int [] pushA,int [] popA) {
    Stack<Integer> stack = new Stack<>();
    int pushIndex = 0;
    int popIndex = 0;

    while(pushIndex!=pushA.length){
        stack.push(pushA[pushIndex]);
        while(!stack.isEmpty() && stack.peek()==popA[popIndex]){
            stack.pop();
            popIndex ++;
        }
        pushIndex ++;
    }

    return stack.isEmpty();
}
```

23：层次遍历二叉树(☆)

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:

给定二叉树: `[3,9,20,null,null,15,7]`,

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

返回:

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

```
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
    ArrayList<Integer> ans = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();

    if(root!=null){
        queue.add(root);
    }
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
        ans.add(node.val);
        if(node.left!=null){
            queue.add(node.left);
        }
        if(node.right!=null){
            queue.add(node.right);
        }
    }
    return ans;
}
```

24：树的子结构(😫)

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构， 即 A中有出现和B相同的结构和节点值。

示例 1:

```
输入: A = [1,2,3], B = [3,1]
输出: false
```

示例 2:

```
输入: A = [3,4,5,1,2], B = [4,1]
输出: true
```

```
public class Solution {
    public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        if(root1==null || root2==null){
            return false;
        }
        return isSubTree(root1,root2) || HasSubtree(root1.left,root2) ||
HasSubtree(root1.right,root2);
    }

    public boolean isSubTree(TreeNode root1,TreeNode root2){
        if(root1==null){
            return false;
        }

        if(root2==null){
            return true;
        }

        if(root1.val != root2.val){
```

```
            return false;
        }

        return isSubTree(root1.left,root2.left) &&
isSubTree(root1.right,root2.right);
    }
}
```

25：二叉树的后续遍历序列（😭）

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 `true`，否则返回 `false`。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

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

复制代码

示例 1：

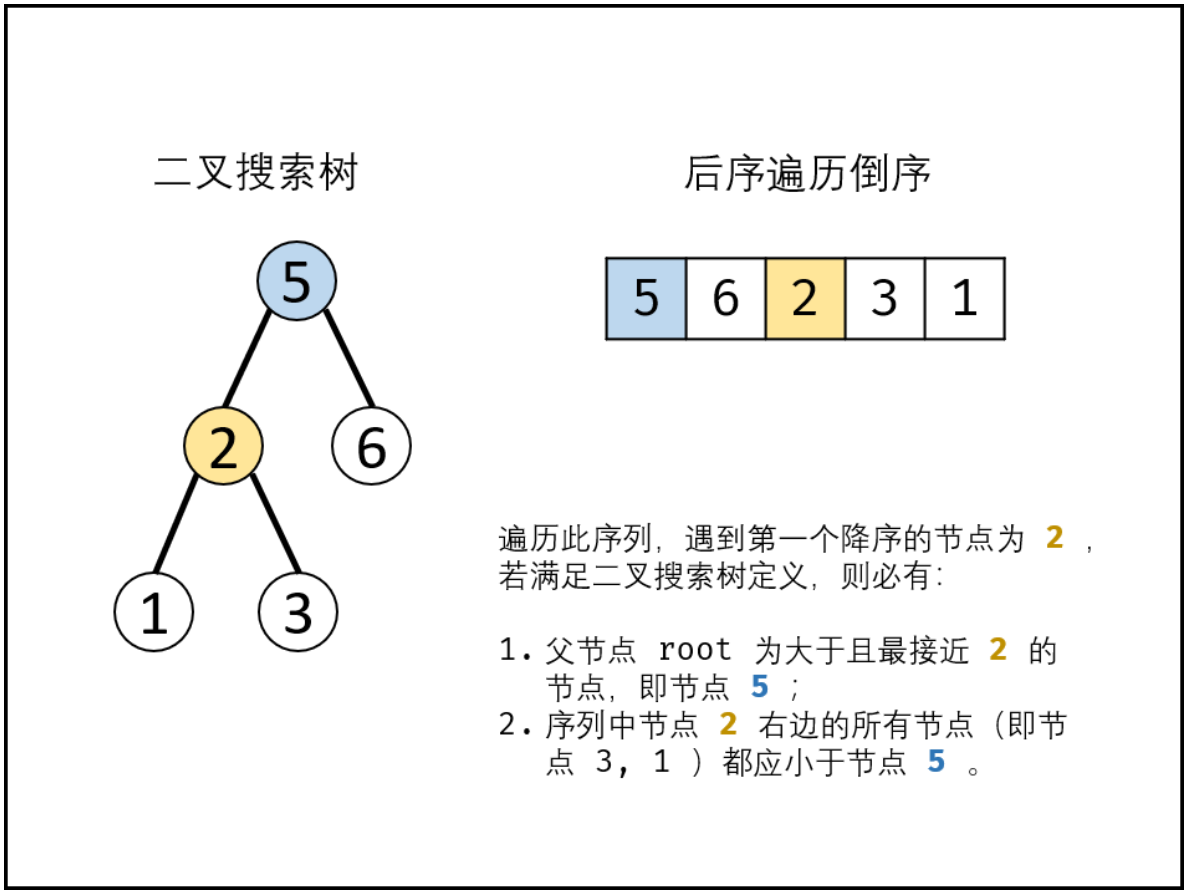
```
输入: [1,6,3,2,5]
输出: false
```

复制代码

示例 2：

```
输入: [1,3,2,6,5]
输出: true
```

复制代码

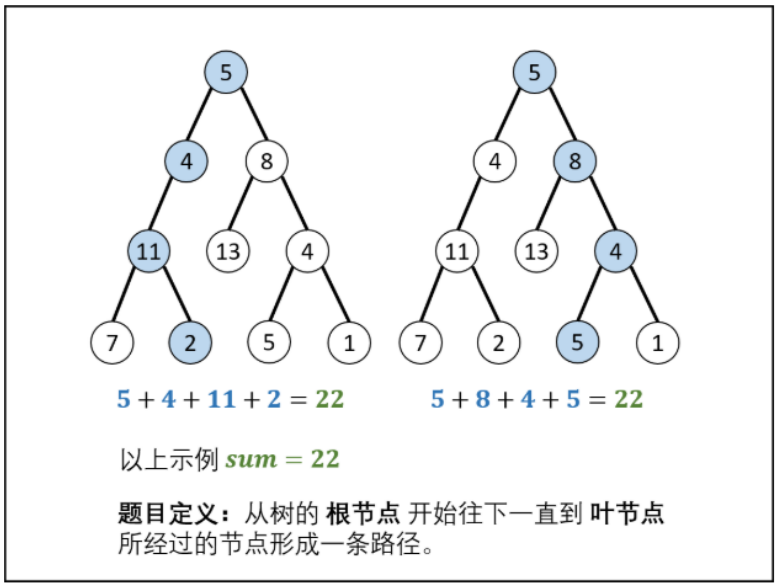



```
public boolean VerifySquenceOfBST(int [] sequence) {
    if(sequence==null || sequence.length==0){
```

```
            return false;
        }
        Stack<Integer> stack = new Stack<>();
        int root = Integer.MAX_VALUE;
        int length = sequence.length;

        for(int i=length-1;i>=0;i--){
            if(sequence[i]>root){
                return false;
            }
            while(!stack.isEmpty() && stack.peek()>sequence[i]){
                root = stack.pop();
            }
            stack.push(sequence[i]);
        }
        return true;
    }
```

26：二叉树中和为某一值的路径(😩)

- **先序遍历**： 按照 “根、左、右” 的顺序，遍历树的所有节点。
- **路径记录**： 在先序遍历中，记录从根节点到当前节点的路径。当路径为 ① 根节点到叶节点形成的路径 **且** ② 各节点值的和等于目标值 `sum` 时，将此路径加入结果列表。




```
public class Solution {
    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
    LinkedList<Integer> temp = new LinkedList<>();
    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {
        help(root,target);
        return ans;
    }

    public void help(TreeNode root,int sum){
        if(root==null){
            return;
        }
        temp.add(root.val);
        if(sum-root.val == 0 && root.left == null && root.right == null){
            ans.add(new ArrayList(temp));
        }
```

```
            help(root.left,sum-root.val);
            help(root.right,sum-root.val);
            temp.removeLast();  //不符合条件的当前节点回退
        }
    }
```

27：复杂链表的复制(😫)

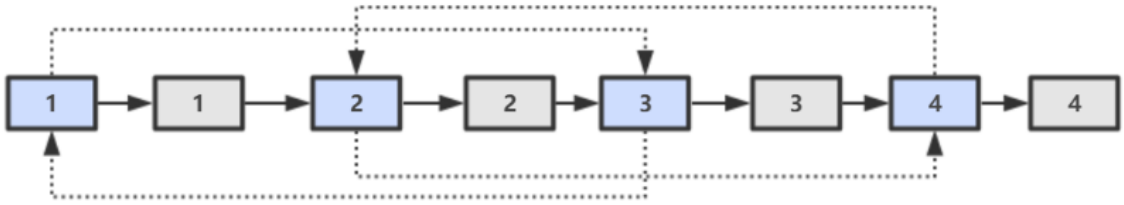
题目描述

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的 head。

复制代码

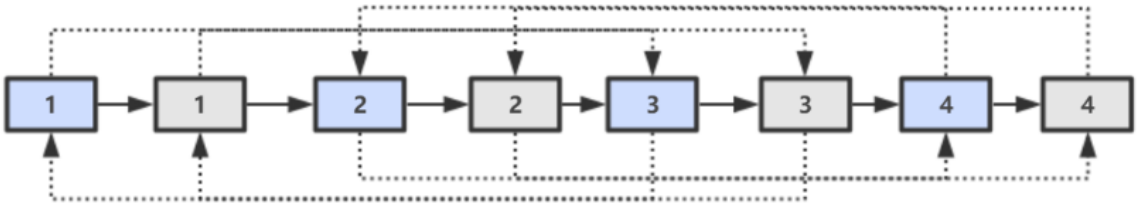
```
public class RandomListNode {
    int label;
    RandomListNode next = null;
    RandomListNode random = null;

    RandomListNode(int label) {
        this.label = label;
    }
}
```

第一步，在每个节点的后面插入复制的节点。



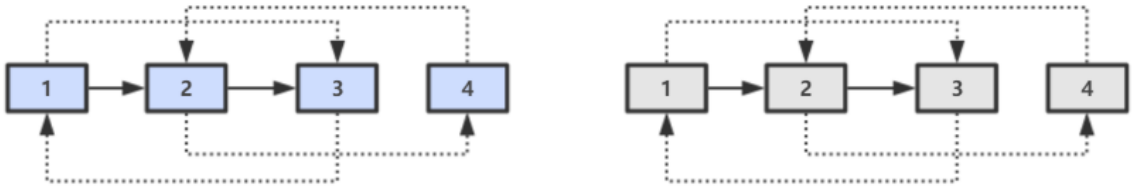
CyC2018

第二步，对复制节点的 random 链接进行赋值。



CyC2018

第三步，拆分。



CyC2018

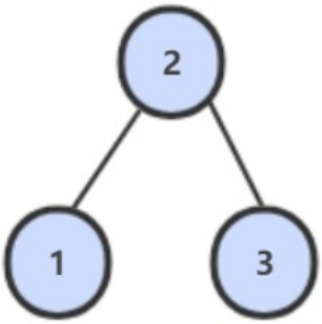
```
/*
public class RandomListNode {
    int label;
    RandomListNode next = null;
```

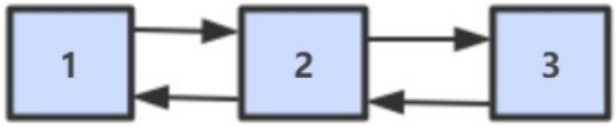
```
    RandomListNode random = null;

    RandomListNode(int label) {
        this.label = label;
    }
}
*/
public class Solution {
    public RandomListNode Clone(RandomListNode pHead)
    {
        if(pHead == null){
            return null;
        }

        //插入新节点
        RandomListNode cur = pHead;
        while(cur!=null){
            RandomListNode clone = new RandomListNode(cur.label);
            clone.next = cur.next;
            cur.next = clone;
            cur = clone.next;
        }
        //建立random连接
        cur = pHead;
        while(cur != null){
            RandomListNode clone = cur.next;
            if(cur.random !=null){
                clone.random = cur.random.next;  //这里的next是被random指向的clone
的相同节点
            }
            cur = clone.next;
        }

        //拆分
        cur = pHead;
        RandomListNode pCloneHead = cur.next;
        while(cur.next!=null){
            RandomListNode next = cur.next;
            cur.next = next.next;
            cur = next;
        }
        return pCloneHead;
    }
}
```

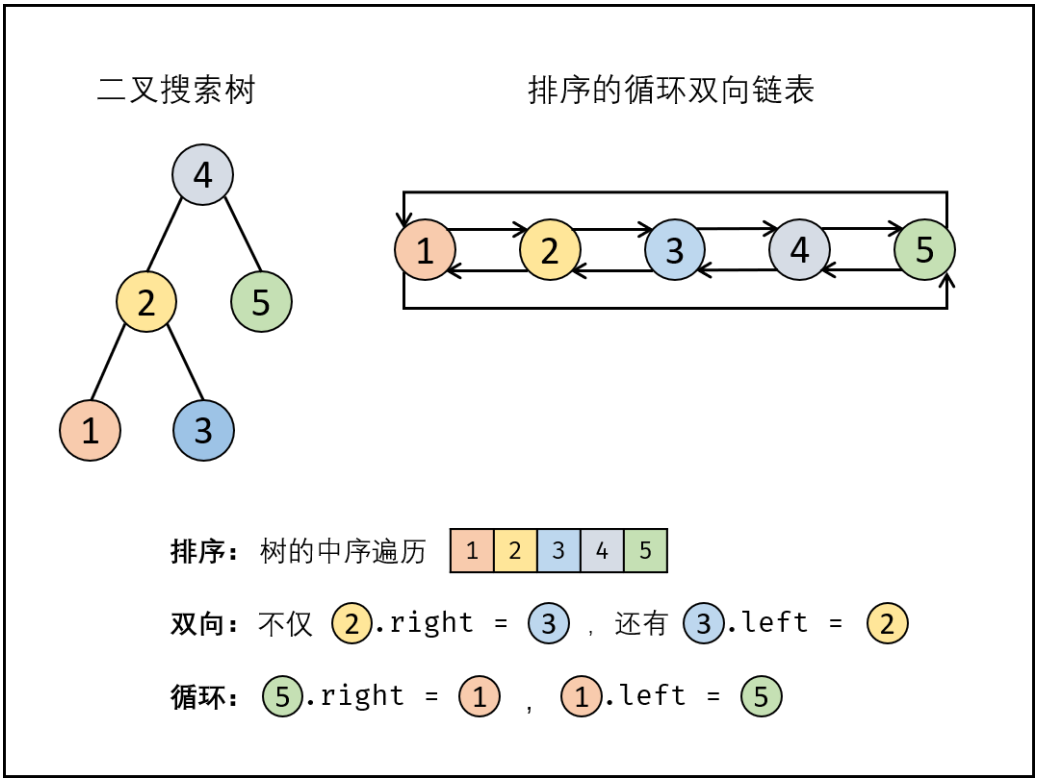
28：二叉搜索树与双向链表(😫)

题目描述

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。








```
class Solution {
    Node head,pre;
    public Node treeToDoublyList(Node root) {
        if(root == null){
            return null;
        }
        dfs(root);
        pre.right = head;
        head.left = pre;
        return head;
    }
    public void dfs(Node cur){
        if(cur == null){
            return;
        }
        dfs(cur.left);
        if(pre == null){
            head = cur;
        }else{
```

```
            pre.right = cur;
        }
        cur.left = pre;
        pre = cur;
        dfs(cur.right);
    }
}
```

29：字符串排列(😫)

输入一个字符串，打印出该字符串中字符的所有排列。

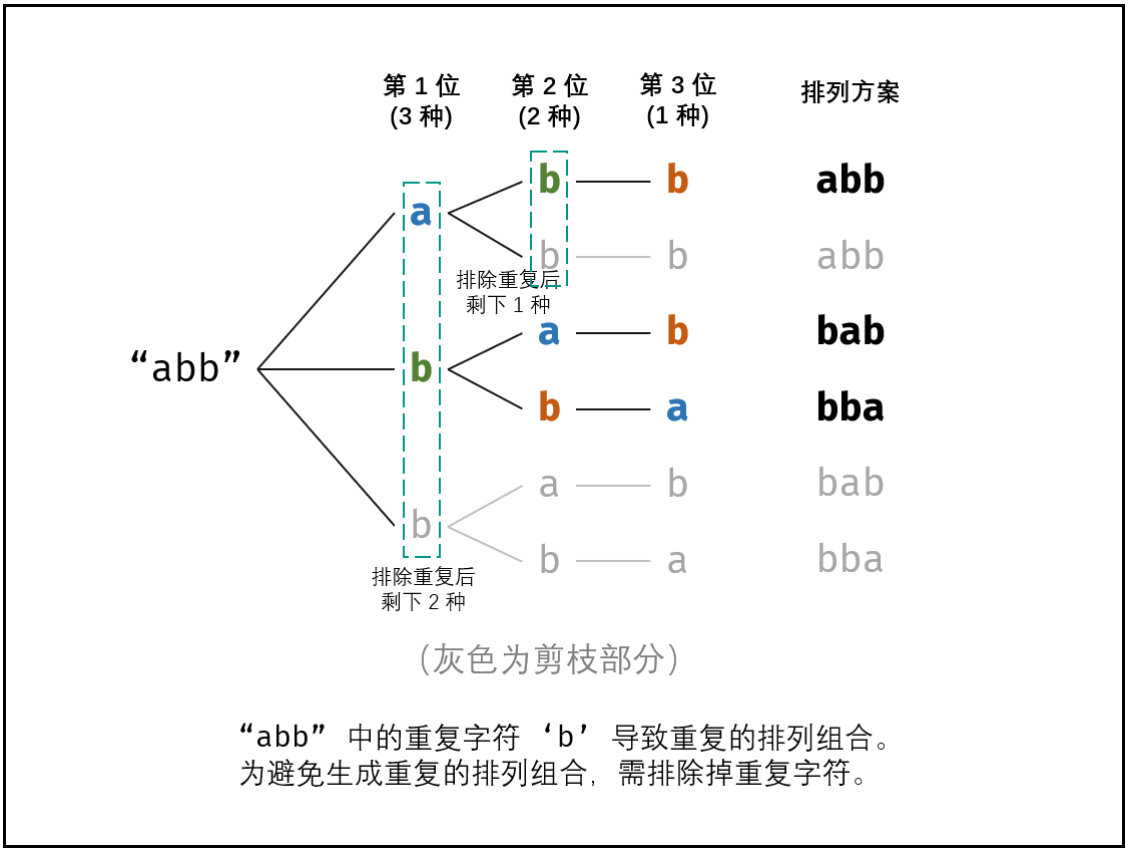
你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

```
输入：s = "abc"
输出：["abc","acb","bac","bca","cab","cba"]
```

递归解析：

1. **终止条件：** 当 $x = len(c) - 1$ 时，代表所有位已固定（最后一位只有 1 种情况），则将当前组合 `c` 转化为字符串并加入 `res` ，并返回；
2. **递推参数：** 当前固定位 x ；
3. **递推工作：** 初始化一个 Set ，用于排除重复的字符；将第 x 位字符与 $i \in [x, len(c)]$ 字符分别交换，并进入下层递归；
    1. **剪枝：** 若 $c[i]$ 在 Set 中，代表其是重复字符，因此“剪枝”；
    2. 将 $c[i]$ 加入 Set ，以便之后遇到重复字符时剪枝；
    3. **固定字符：** 将字符 $c[i]$ 和 $c[x]$ 交换，即固定 $c[i]$ 为当前位字符；
    4. **开启下层递归：** 调用 $dfs(x+1)$ ，即开始固定第 $x+1$ 个字符；
    5. **还原交换：** 将字符 $c[i]$ 和 $c[x]$ 交换（还原之前的交换）；



（灰色为剪枝部分）

“abb” 中的重复字符 ‘b’ 导致重复的排列组合。
为避免生成重复的排列组合，需排除掉重复字符。

```java
public class Solution {
    ArrayList<String> ans = new ArrayList<>();
    char[] c;
    public ArrayList<String> Permutation(String str) {
       c = str.toCharArray();
       dfs(0);
       Collections.sort(ans);
       return ans;
    }

    public void dfs(int x){
        if(x == c.length-1){
            ans.add(String.valueOf(c));
            return;
        }

        HashSet<Character> set = new HashSet<>();
        for(int i=x;i<c.length;i++){
            if(set.contains(c[i])){
                continue;
            }
            set.add(c[i]);
            swap(i,x);
            dfs(x+1);
            swap(i,x);
        }
    }

    public void swap(int a,int b){
        char temp = c[a];
        c[a] = c[b];
        c[b] = temp;
    }
}
```

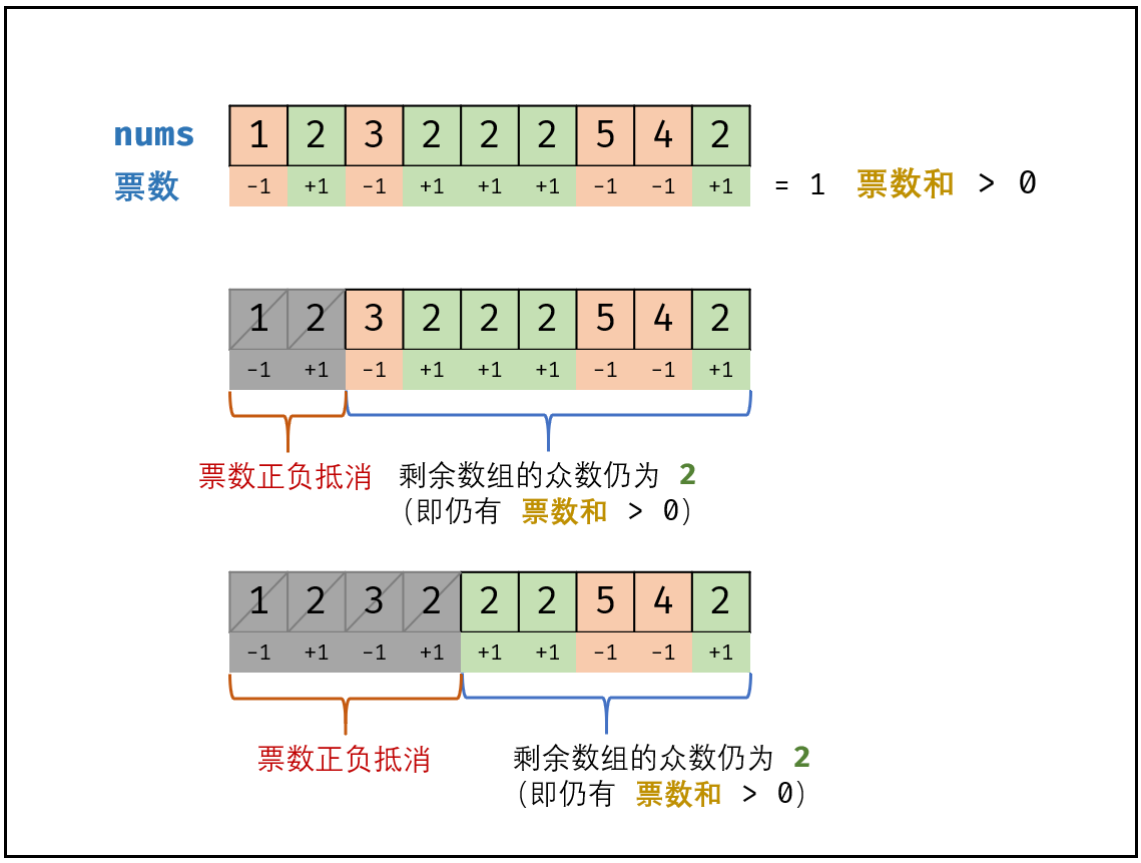
30：数组中超过一半的元素（😁）

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

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




```java
public int MoreThanHalfNum_Solution(int [] array) {
    //HashMap放方法
    HashMap<Integer,Integer> map = new HashMap<>();
    for(int i=0;i<array.length;i++){
        if(map.containsKey(array[i])){
            map.put(array[i],map.get(array[i])+1);
            continue;
        }
        map.put(array[i],1);
    }
    for(Map.Entry<Integer,Integer> entry:map.entrySet()){
        if(entry.getValue()>array.length/2){
            return entry.getKey();
        }
    }
    return 0;

    //投票法  选择数组中的众数
    int x = 0,votes =0,count =0;
    for(int num:array){
        if(votes==0){
            x = num;  //新的众数待选
        }
        votes += num==x ? 1:-1;
    }
    for(int num : array){
        if(num == x){
            count ++;
        }
    }
    return count>array.length/2? x:0;
}
```

31：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。

示例 1：

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

示例 2：

```
输入：arr = [0,1,2,1], k = 1
输出：[0]
```

大根堆解决前最下K个数

小根堆解决前最大K个数

```
//// 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。

public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
    if(input.length == 0 || k==0 || k>input.length){
        return new ArrayList<Integer>();
    }

    ArrayList<Integer> ans = new ArrayList<>();
    PriorityQueue<Integer> queue = new PriorityQueue<>(new Comparator<Integer>()
{
        public int compare(Integer o1,Integer o2){
            return o2 - o1;
        }
    });

    for(int num:input){
        if(queue.size()<k){
            queue.add(num);
        }else if(num<queue.peek()){
            queue.poll();
            queue.add(num);
        }
    }

    while(queue.size()>0){
        ans.add(queue.poll());
    }
    return ans;
}
```

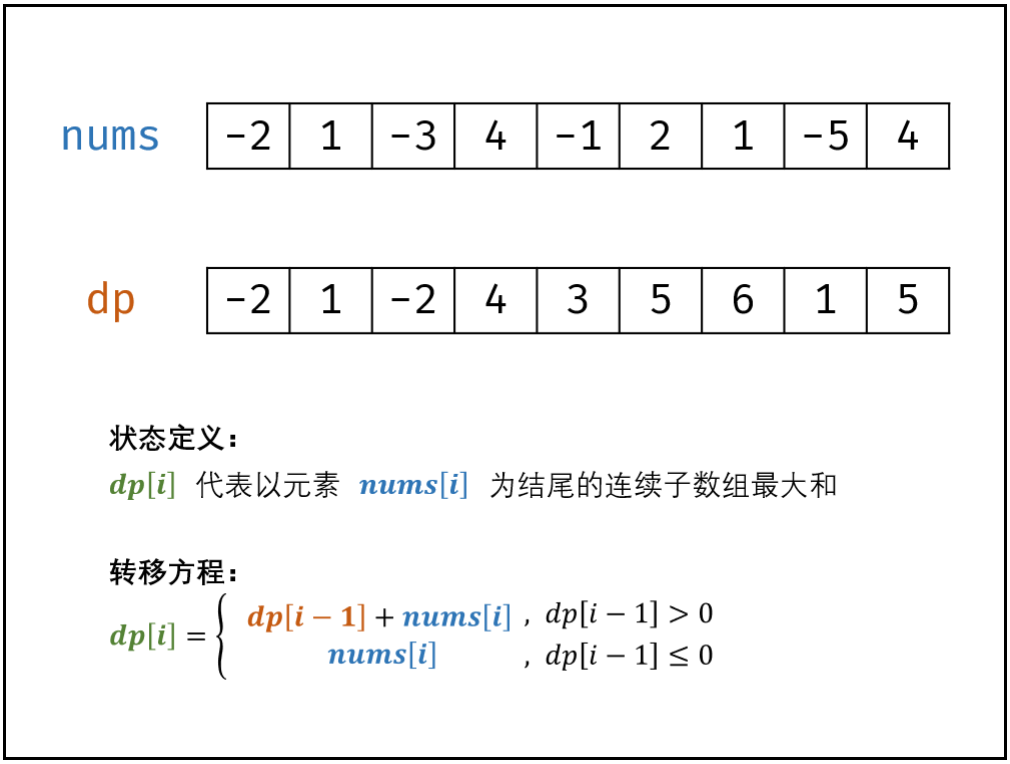
32：连续子数组的最大和(dp--->😄)

输入一个整型数组，数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值。

要求时间复杂度为O(n)。

示例1:

```
输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6。
```




```java
public int FindGreatestSumOfSubArray(int[] array) {
    int[] dp = new int[array.length];
    dp[0] = array[0];  //截止当前索引的最大和
    int max = Integer.MIN_VALUE;
    for(int i=1;i<array.length;i++){
        if(dp[i-1]<=0){
            dp[i] = array[i];
            max = Math.max(max,dp[i]);
        }else{
            dp[i] = dp[i-1]+array[i];
            max = Math.max(max,dp[i]);
        }
    }
    return max;
}
```

33：把数组排成最小的数(☆)

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

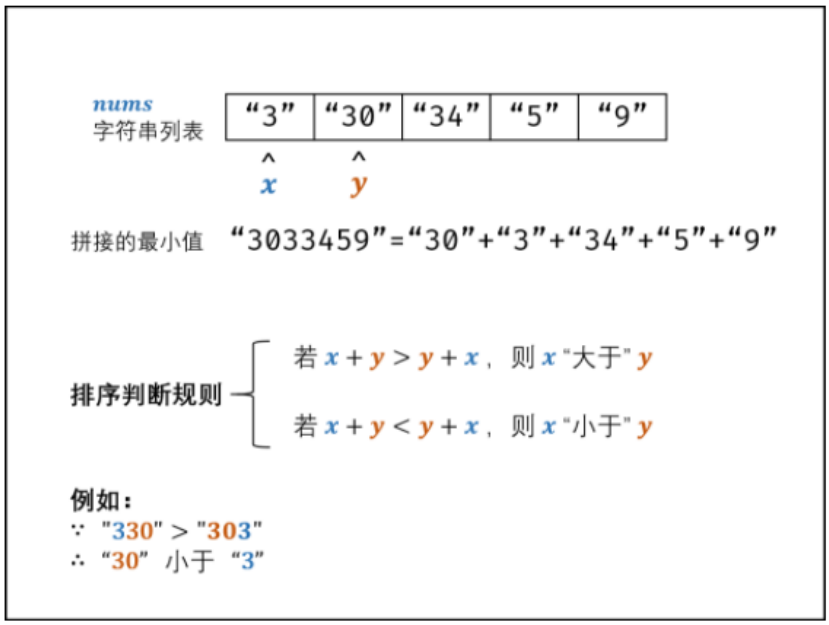
```
输入：[10,2]
输出："102"
```

示例 2:

```
输入：[3,30,34,5,9]
输出："3033459"
```

解题思路：

- 此题求拼接起来的“最小数字”，本质上是一个排序问题。
- **排序判断规则：** 设 $nums$ 任意两数字的字符串格式 x 和 y，则
  - 若拼接字符串 $x+y>y+x$，则 $m>n$；
  - 反之，若 $x+y<y+x$，则 $n<m$；
- 根据以上规则，套用任何排序方法对 $nums$ 执行排序即可。




```
public String PrintMinNumber(int [] numbers) {
    if(numbers == null || numbers.length == 0){
        return "";
    }

    String[] ans = new String[numbers.length];
    for(int i=0;i<numbers.length;i++){
        ans[i] = String.valueOf(numbers[i]);
    }
    Arrays.sort(ans,new Comparator<String>(){
        public int compare(String s1,String s2){
            return (s1+s2).compareTo(s2+s1);
        }
    });
```

```
    StringBuilder sb = new StringBuilder();
    for(String s:ans){
        sb.append(s);
    }
    return sb.toString();
}
```

34：丑数(☆)

> 我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。
>
> **示例:**
>
> ```
> 输入：n = 10
> 输出：12
> 解释：1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
> ```

思路:

```
一个十分巧妙的动态规划问题
1.我们将前面求得的丑数记录下来，后面的丑数就是前面的丑数*2，*3，*5
2.但是问题来了，我怎么确定已知前面k-1个丑数，我怎么确定第k个丑数呢
3.采取用三个指针的方法，p2,p3,p5
4.p2指向的数字下一次永远*2，p3指向的数字下一次永远*3，p5指向的数字永远*5
5.我们从2*p2 3*p3 5*p5选取最小的一个数字，作为第k个丑数
6.如果第K个丑数==2*p2，也就是说前面0-p2个丑数*2不可能产生比第K个丑数更大的丑数了，所以p2++
7.p3,p5同理
8.返回第n个丑数
```

```
public int GetUglyNumber_Solution(int index) {
    if(index == 0){
        return 0;
    }
    int p2=0,p3=0,p5=0;
    int[] dp = new int[index];
    dp[0] = 1;
    for(int i=1;i<index;i++){
        dp[i] = Math.min(dp[p2]*2,Math.min(dp[p3]*3,dp[p5]*5));
        if(dp[p2]*2 == dp[i]) p2++;
        if(dp[p3]*3 == dp[i]) p3++;
        if(dp[p5]*5 == dp[i]) p5++;
    }

    return dp[index-1];
}
```

35：第一个只出现一次的字符

题目描述

在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）.（从0开始计数）

```
public int FirstNotRepeatingChar(String str) {
    char[] chars = str.toCharArray();
    HashMap<Character,Integer> map = new LinkedHashMap<>();
    for(int i=0;i<chars.length;i++){
        if(!map.containsKey(chars[i])){
            map.put(chars[i],1);
            continue;
        }
        map.put(chars[i],map.get(chars[i])+1);
    }
    for(int i=0;i<chars.length;i++){
        if(map.get(chars[i])==1){
            return i;
        }
    }
    return -1;
}
```

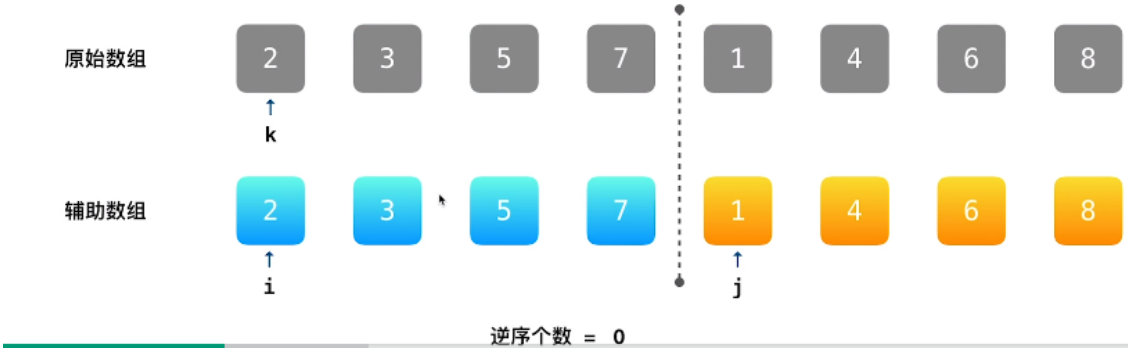
36：逆序对(😭 归并排序并计数)

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

```
输入: [7,5,6,4]
输出: 5
```

- 合并两个有序数组



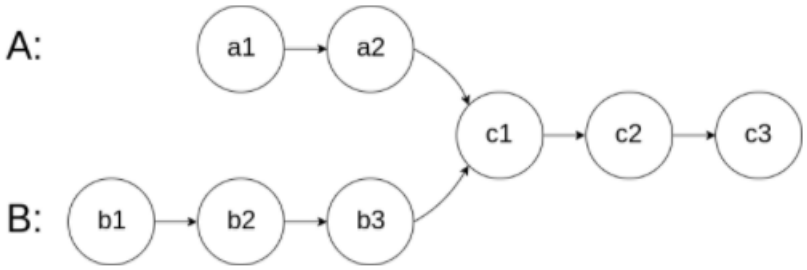

```
public class Solution {
    public int InversePairs(int [] array) {
        int len = array.length;
        if(len<2){
            return 0;
        }
        int[] copy = new int[len];
```

```
        for(int i=0;i<len;i++){
            copy[i] = array[i];
        }

        int[] temp = new int[len];
        return (int)sort(copy,0,len-1,temp)%1000000007;
    }

    public int sort(int[] array,int left,int right,int[] temp){
        if(left == right){
            return 0;
        }

        int mid = left + (right-left)/2;
        int leftPairs = sort(array,left,mid,temp);
        int rightPairs = sort(array,mid+1,right,temp);
        if(array[mid]<=array[mid+1]){
            return leftPairs+rightPairs;
        }
        int crossPairs = mergeCount(array,left,mid,right,temp);

        return leftPairs+rightPairs+crossPairs;
    }

    public int mergeCount(int[] array,int left,int mid,int right,int[] temp){
        for(int i=left;i<=right;i++){
            temp[i] = array[i];
        }

        int i = left;
        int j = mid + 1;

        int count = 0;
        for(int k=left;k<=right;k++){
            if(i==mid+1){  //左边遍历完  将所有右边数组加入temp
                array[k] = temp[j];
                j++;
            }else if(j==right+1){
                array[k] = temp[i];
                i++;
            }else if(temp[i]<=temp[j]){
                array[k] = temp[i];
                i++;
            }else{
                array[k] = temp[j];
                j++;
                count += (mid-i+1);
            }
        }
        return count;
    }
}
```

37：链表第一个公共节点(☆)

输入两个链表，找出它们的第一个公共节点。

如下面的两个链表：

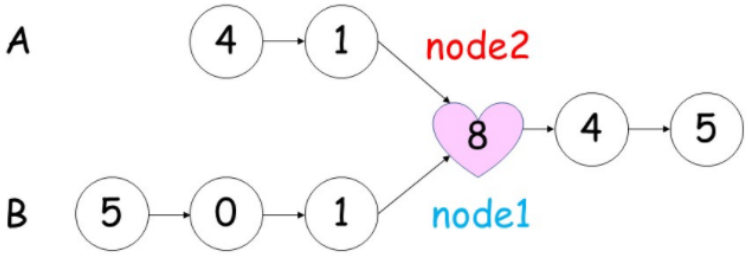


在节点 c1 开始相交。

解题思路：

我们使用两个指针 `node1`， `node2` 分别指向两个链表 `headA`， `headB` 的头结点，然后同时分别逐结点遍历，当 `node1` 到达链表 `headA` 的末尾时，重新定位到链表 `headB` 的头结点；当 `node2` 到达链表 `headB` 的末尾时，重新定位到链表 `headA` 的头结点。

这样，当它们相遇时，所指向的结点就是第一个公共结点。




```
public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
    ListNode p1 = pHead1;
    ListNode p2 = pHead2;
    while(p1!=p2){
        if(p1!=null){
            p1 = p1.next;
        }else{
            p1 = pHead2;
        }

        if(p2!=null){
            p2 = p2.next;
        }else{
            p2 = pHead1;
        }
    }
    return p1;
}
```

38：排序数组中数字出现的次数(☆)

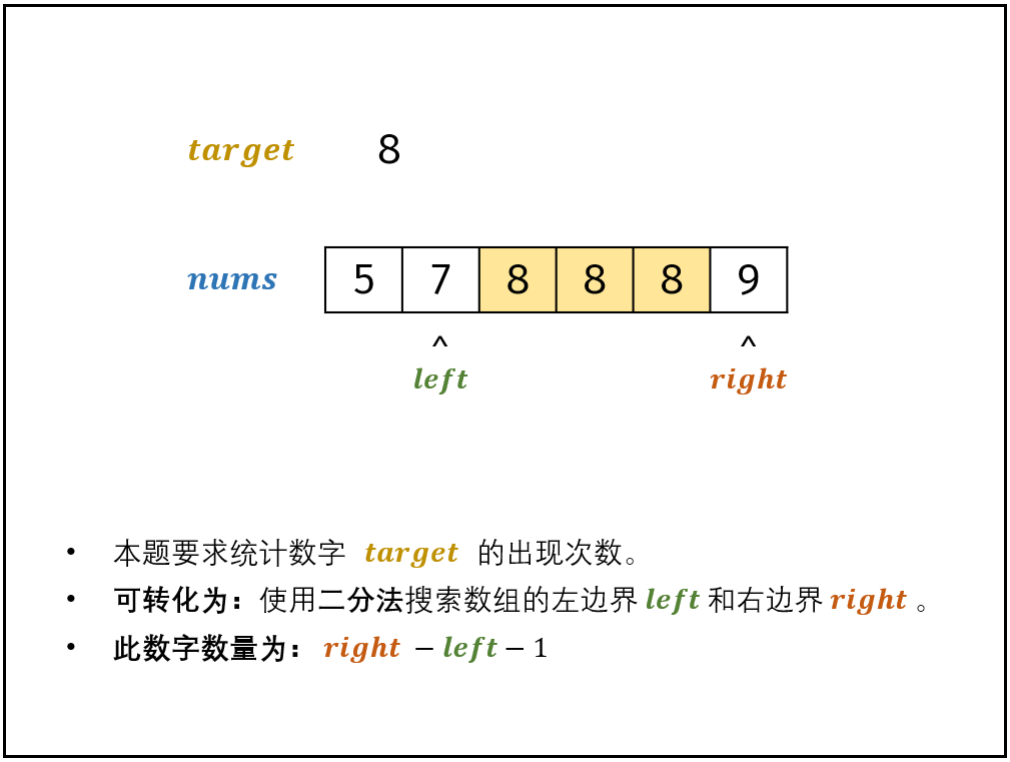
统计一个数字在排序数组中出现的次数。

示例 1:

```
输入：nums = [5,7,7,8,8,10], target = 8
输出：2
```

示例 2:

```
输入：nums = [5,7,7,8,8,10], target = 6
输出：0
```

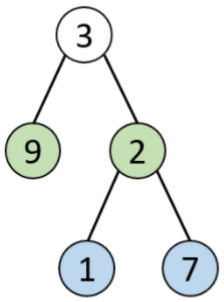


- 本题要求统计数字 *target* 的出现次数。
- **可转化为：**使用**二分法**搜索数组的左边界 *left* 和右边界 *right* 。
- **此数字数量为：** $right - left - 1$

```
public int GetNumberOfK(int [] array , int k) {
    //搜索右边界
    int i=0;
    int j=array.length-1;
    while(i<=j){
        int m = i+(j-i)/2;
        if(array[m]<=k){  //**************等于号*********
            i = m+1;
        }else{
            j = m-1;
        }
    }
    int right = i;
    //搜索左边界
    i = 0;
    j = array.length-1;
    while(i<=j){
        int m = i + (j-i)/2;
        if(array[m]<k){ //**************无等于号*************
            i = m+1;
```

```
            }else{
                j = m-1;
            }
        }
        int left = j;

        return right-left-1;

    }
```

39：二叉树的深度(☆---->层次遍历+计数)



$queue = [9, 2]$

$tmp = [1, 7]$

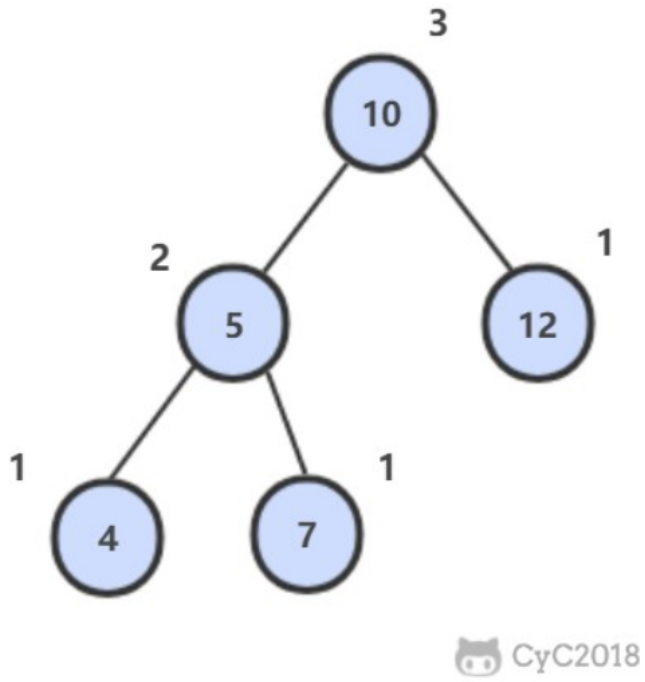
$res = 1$

遍历**第二层**，将**第三层**节点加入 tmp

```
public int TreeDepth(TreeNode root) {
    if(root == null){
        return 0;
    }
    //层次遍历统计树的深度
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int ans = 0;
    while(!queue.isEmpty()){
        LinkedList<TreeNode> temp = new LinkedList<>();
        for(TreeNode node:queue){
            if(node.left!=null) temp.add(node.left);
            if(node.right!=null) temp.add(node.right);
        }
        queue = temp;
        ans ++;
    }
    return ans;
}
```

40：平衡二叉树(☆)

平衡二叉树左右子树高度差不超过 1。




```
public class Solution {
    private boolean isBanlanced = true;
    public boolean IsBalanced_Solution(TreeNode root) {
        height(root);
        return isBanlanced;
    }

    public int height(TreeNode root){
        if(root == null){
            return 0;
        }

        int left = height(root.left);
        int right = height(root.right);

        if(Math.abs(left-right)>1){
            isBanlanced = false;
        }

        return 1+Math.max(left,right);
    }
}
```

41：数组中数字出现的次数(☆)

一个整型数组 nums 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

示例 1：

```
输入：nums = [4,1,4,6]
输出：[1,6] 或 [6,1]
```

示例 2：

```
输入：nums = [1,2,10,4,1,4,3,3]
输出：[2,10] 或 [10,2]
```

1：全员异或，得到两个不同数字的异或值（相同数值异或为0）

2：找到异或结果中第一个为1的bit位即mask

3：遍历数组，根据数字的该位是否为1划分为两个子数组，且每个子数组只包含一个出现一次的数字

```
class Solution {
    public int[] singleNumbers(int[] nums) {
        //*******************Step1*********************
        int sum = 0;
        for(int n:nums){
            sum ^= n;
        }
        //*******************Step2*********************
        int mask = 1;
        while((mask & sum) == 0){
            mask <<=1;
        }
        //S*******************Step3*********************
        int a = 0;
        int b = 0;
        for(int n:nums){
            if((mask & n) == 0){
                a ^= n;
            }else{
                b ^= n;
            }
        }
        return new int[]{a,b};
    }
}
```

42：和为s的连续正数序列(☆---->双指针)

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。

示例 1：

```
输入：target = 9
输出：[[2,3,4],[4,5]]
```

示例 2：

```
输入：target = 15
输出：[[1,2,3,4,5],[4,5,6],[7,8]]
```

```java
public class Solution {
    public ArrayList<ArrayList<Integer> > FindContinuousSequence(int target) {
       ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
        int i = 1;
        int j = 1;
        int sum = 0;
        while(i<=target/2){
            if(sum<target){
                target += j;
                j++;
            }else if(sum >target){
                target -= i;
                i++;
            }else{
                ArrayList<Integer> temp = new ArrayList<>();
                for(int k = i;k<j;k++){
                    temp.add(k);
                }
                ans.add(temp);
                sum -= i;
                i++;
            }
        }

        return ans;
    }
}
```

43：和为S的两个数字(😄)

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

示例 1：

```
输入：nums = [2,7,11,15], target = 9
输出：[2,7] 或者 [7,2]
```

示例 2：

```
输入：nums = [10,26,30,31,47,60], target = 40
输出：[10,30] 或者 [30,10]
```

```
public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
    //因为排序  所以双指针即可
    ArrayList<Integer> ans = new ArrayList<>();
    int i = 0;
    int j = array.length-1;

    while(i<j){
        if((array[i]+array[j])>sum){
            j--;
        }else if((array[i]+array[j])<sum){
            i++;
        }else{
            ans.add(array[i]);
            ans.add(array[j]);
            break;
        }
    }
    return ans;
}
```

44：左旋转字符串(😄)

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

示例 1：

```
输入：s = "abcdefg", k = 2
输出："cdefgab"
```

```
public String LeftRotateString(String str,int n) {
    if(n>str.length()){
        return "";
    }
    StringBuilder ans = new StringBuilder();
```

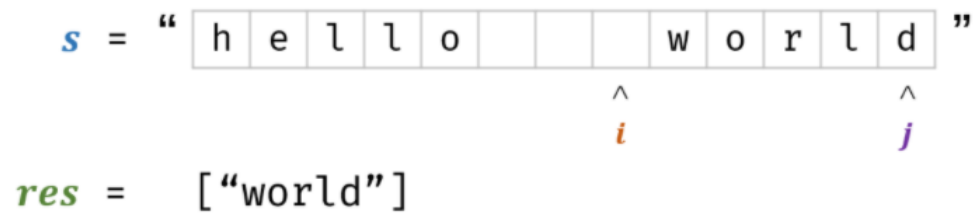
```
    for(int i=n;i<str.length();i++){
        ans.append(str.charAt(i));
    }
    for(int i=0;i<n;i++){
        ans.append(str.charAt(i));
    }

    return ans.toString();
}
```

45：翻转单词顺序列(☆)

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. "，则输出"student. a am I"。

示例 1：

```
输入："the sky is blue"
输出："blue is sky the"
```



循环执行：

1. 索引 i 从右向左搜索首个空格；
2. 添加单词 $s[i+1:j+1]$ 至 res ；
3. 索引 i 跳过两单词间的所有空格；
4. 执行 $j=i$，此时 j 指向下个单词的尾字符；

```
public class Solution {
    public String ReverseSentence(String str) {
        if(str==null || str.trim().length()==0){
            return str;
        }
        str.trim();
        int i = str.length()-1;
        int j = str.length()-1;
        StringBuilder ans = new StringBuilder();
        while(i>=0){
            //**************遍历单词加入ans************
            while(i>=0 && str.charAt(i)!=' '){
                i--;
            }
            ans.append(str.substring(i+1,j+1)+" ");
            //**************遍历单词间的空格************
```

```
            while(i>=0 && str.charAt(i)==' '){
                i--;
            }
            j = i;
        }
        return ans.toString().trim();
    }
}
```

46：扑克牌中的顺子

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A不能视为 14。

示例 1:

```
输入: [1,2,3,4,5]
输出: True
```

```
public class Solution {
    public boolean isContinuous(int [] numbers) {
        if(numbers == null || numbers.length == 0){
            return false;
        }
        int joker = 0;
        Arrays.sort(numbers);
        for(int i=0;i<4;i++){
            if(numbers[i]==0){
                joker ++;
            }else{
                if(numbers[i]==numbers[i+1]){
                    return false;
                }
            }
        }
        return numbers[4]-numbers[joker]<5;
    }
}
```

47：圆圈中剩下的数字(😄)

0,1,,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

```
输入：n = 5, m = 3
输出：3
```

示例 2：

```
输入：n = 10, m = 17
输出：2
```

```java
public class Solution {
    public int LastRemaining_Solution(int n, int m) {
        if(n == 0 || m == 0){
            return -1;
        }
        List<Integer> list = new ArrayList<>();
        for(int i=0;i<n;i++){
            list.add(i);
        }
        int startIndex = 0;
        while(list.size()>1){
            int deleteIndex = (startIndex+m-1)%list.size();
            list.remove(deleteIndex);
            startIndex = deleteIndex;
        }
        return list.get(0);
    }
}
```

48：不用加减乘数求1+2+....n(☆)

求 `1+2+...+n` ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

示例 1：

```
输入：n = 3
输出：6
```

示例 2：

```
输入：n = 9
输出：45
```

`if ( A && B )`

- 若 **A** 为 **false** ，则不会执行判断 **B** （即 && 短路）

`if ( A || B )`

- 若 **A** 为 **true** ，则不会执行判断 **B** （即 || 短路）

`n > 1 && sumNums(n - 1)`

- 若 **n** > 1 成立 ，则开启下层递归 sumNums(**n** - 1)
- 若 **n** > 1 不成立 ，则终止递归

```
public class Solution {
    int ans = 0;
    public int Sum_Solution(int n) {
        boolean x = n>1 && Sum_Solution(n-1)>0;
        ans += n;
        return ans;
    }
}
```

49：位运算做加法(☆)

写一个函数，求两个整数之和，要求在函数体内不得使用 “+”、“-”、“*”、“/” 四则运算符号。

示例:

```
输入：a = 1, b = 1
输出：2
```

本题考察对位运算的灵活使用，即使用位运算实现加法。
设两数字的二进制形式 a, b ，其求和 $s = a + b$ ， $a(i)$ 代表 a 的二进制第 i 位，则分为以下四种情况：

$a(i)$	$b(i)$	无进位和 $n(i)$	进位 $c(i+1)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

观察发现，**无进位和** 与 **异或运算** 规律相同，**进位** 和 **与运算** 规律相同（并需左移一位）。因此，无进位和 n 与进位 c 的计算公式如下：

$$\begin{cases} n = a \oplus b & \text{非进位和：异或运算} \\ c = a \& b << 1 & \text{进位：与运算 + 左移一位} \end{cases}$$

（和 s ）= （非进位和 n ）+ （进位 c ）。即可将 $s = a + b$ 转化为：

$$s = a + b \Rightarrow s = n + c$$

循环求 n 和 c ，直至进位 $c = 0$ ；此时 $s = n$ ，返回 n 即可。

```
class Solution {
    public int add(int a, int b) {
        while(b != 0) { // 当进位为 0 时跳出
            int c = (a & b) << 1;  // c = 进位
            a ^= b; // a = 非进位和
            b = c; // b = 进位
        }
        return a;
    }
}
```

50：字符串转数字(☆)

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0

输入描述:

> 输入一个字符串,包括数字字母符号,可以为空

输出描述:

> 如果是合法的数值表达则返回该数字，否则返回0

示例1

输入 复制

```
+2147483647
1a33
```

```java
public class Solution {
    public int StrToInt(String str) {
        int cur = 0;
        int length = str.length();
        char[] chars = str.toCharArray();

        while(cur<length && chars[cur] == ' '){
            cur++;
        }
        if(cur==length){
            return 0;
        }
        boolean isNeg = false;
        if(chars[cur]=='-'){
            isNeg = true;
            cur++;
        }else if(chars[cur]=='+'){
            cur++;
        }else if(!Character.isDigit(chars[cur])){
            return 0;
        }

        //与leetcode不同的一点  字符串不符合数字规范 即返回0
        for(int i = cur;i<length;i++){
            if(!Character.isDigit(chars[i])){
                return 0;
            }
        }

        int ans = 0;
        while(cur<length && Character.isDigit(chars[cur])){
            int digit = chars[cur]-'0';
            if(ans>(Integer.MAX_VALUE-digit)/10){
                return ans = isNeg? Integer.MIN_VALUE : Integer.MAX_VALUE;
            }
```

```
            ans = ans*10+digit;
            cur ++;
        }
        return isNeg? -ans:ans;
    }
}
```

51: 构建乘积数组(☆)

给定一个数组 A[0,1,...,n-1]，请构建一个数组 B[0,1,...,n-1]，其中 B 中的元素 B[i]=A[0]×A[1]×...×A[i-1]×A[i+1]×...×A[n-1]。不能使用除法。

示例:

```
输入：[1,2,3,4,5]
输出：[120,60,40,30,24]
```

i >

		A[0]	A[1]	A[2]	A[3]	A[4]
B[0] =	120 =	1	2	3	4	5
B[1] =	60 =	1	1	3	4	5
B[2] =	40 =	1	2	1	4	5
B[3] =	30 =	1	2	3	1	5
B[4] =	24 =	1	2	3	4	1

```
class Solution {
    public int[] constructArr(int[] a) {
        if(a.length == 0) return new int[0];
        int[] b = new int[a.length];
        b[0] = 1;
        int tmp = 1;
        for(int i = 1; i < a.length; i++) {
            b[i] = b[i - 1] * a[i - 1];
        }
        for(int i = a.length - 2; i >= 0; i--) {
            tmp *= a[i + 1];
            b[i] *= tmp;
        }
        return b;
    }
}
```

52: 第一个只出现一次的字符

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

示例:

```
s = "abaccdeff"
返回 "b"

s = ""
返回 " "
```

```
class Solution {
    public char firstUniqChar(String s) {
        char[] chars = s.toCharArray();
        Map<Character,Boolean> map = new LinkedHashMap<>();
        for(int i=0;i<chars.length;i++){
            if(!map.containsKey(chars[i])){
                map.put(chars[i],true);
            }else{
                map.put(chars[i],false);
            }
        }

        for(char c:chars){
            if(map.get(c))
                return c;
        }

        return ' ';
    }
}
```

53:链表中环的入口节点

题目描述

给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。

思路:

1：先找到相遇节点

2：找到相遇节点后，将慢指针放到链表头，然后头指针和相遇节点指针同时移动，相遇 即环的入口节点。

```
public class Solution {
    public ListNode EntryNodeOfLoop(ListNode pHead)
    {
        ListNode slow = pHead;
        ListNode fast = pHead;
        while(fast!=null && fast.next!=null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow == fast){
                break;
```

```
            }
        }
        if(fast == null || fast.next==null){
            return null;
        }

        slow = pHead;
        while(slow!=fast){
            slow = slow.next;
            fast = fast.next;
        }
        return slow;
    }
}
```

54：删除链表中重复的节点(☆)

题目描述

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

```
public class Solution {
    public ListNode deleteDuplication(ListNode pHead)
    {
        if(pHead == null || pHead.next == null){
            return pHead;
        }

        ListNode head = new ListNode(-1);
        head.next = pHead;
        ListNode pre = head;
        ListNode cur = head.next;

        while(cur!=null){
            if(cur.next!=null && cur.next.val == cur.val){
                while(cur.next!=null  && cur.next.val == cur.val){
                    cur = cur.next;
                }
                cur = cur.next;
                pre.next = cur;
            }else{
                pre = cur;
                cur = cur.next;
            }
        }
        return head.next;
    }
}
```

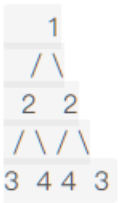
55: 对称二叉树(☆)

剑指 Offer 28. 对称的二叉树

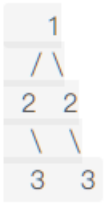
难度 简单 74

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。



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



```
//递归方法
public class Solution {
    boolean isSymmetrical(TreeNode pRoot)
    {
        if(pRoot == null){
            return true;
        }
        return helper(pRoot.left,pRoot.right);
    }

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

```
//非递归方法
public static boolean isSymmetric(TreeNode root){
        if(root == null){
            return true;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root.left);
        queue.add(root.right);
        while (!queue.isEmpty()){
            TreeNode left = queue.poll();
            TreeNode right = queue.poll();
            if(left == null && right == null){
                continue;
```

```
            }
            if(left == null || right==null){
                return false;
            }
            if(left.val != right.val){
                return false;
            }

            queue.add(left.left);
            queue.add(right.right);
            queue.add(left.right);
            queue.add(right.left);
        }
        return true;
    }
```

华为Leetcode题目备考

H01:版本号匹配

比较两个版本号 *version1* 和 *version2*。
如果 *version1* > *version2* 返回 `1`，如果 *version1* < *version2* 返回 `-1`， 除此之外返回 `0`。

你可以假设版本字符串非空，并且只包含数字和 `.` 字符。

`.` 字符不代表小数点，而是用于分隔数字序列。

例如，`2.5` 不是“两个半”，也不是“差一半到三”，而是第二版中的第五个小版本。

你可以假设版本号的每一级的默认修订版号为 `0`。例如，版本号 `3.4` 的第一级（大版本）和第二级（小版本）修订号分别为 `3` 和 `4`。其第三级和第四级修订号均为 `0`。

示例 1:

```
输入：version1 = "0.1", version2 = "1.1"
输出：-1
```

示例 2:

```
输入：version1 = "1.0.1", version2 = "1"
输出：1
```

```
class Solution {
    public int compareVersion(String version1, String version2) {
        String[] nums1 = version1.split("\\.");
        String[] nums2 = version2.split("\\.");
        int n1 = nums1.length;
        int n2 = nums2.length;
        int i1 = 0;
        int i2 = 0;
        for(int i=0;i<Math.max(n1,n2);i++){
```

```
            i1 = i<n1? Integer.parseInt(nums1[i]):0;
            i2 = i<n2? Integer.parseInt(nums2[i]):0;
            if(i1!=i2){
                return i1>i2? 1:-1;
            }
        }
        return 0;
    }
}
```

H02:有效括号

难度 简单 1826

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

有效字符串需满足：

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

注意空字符串可被认为是有效字符串。

示例 1:

```
输入：()
输出：true
```

示例 2:

```
输入：()[]{}
输出：true
```

```
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        char[] chars = s.toCharArray();
        for(char c:chars){
            if(stack.size()==0){
                stack.push(c);
            }else if(isPair(stack.peek(),c)){
                stack.pop();
            }else{
                stack.push(c);
            }
        }
        return stack.size()==0;
    }

    public boolean isPair(char a,char b){
        if(a=='(' && b==')' || a=='{' && b=='}' || a=='[' && b==']'){
            return true;
        }
        return false;
    }
```

```
}
```

H03：出现频率前K高的元素

347. 前 K 个高频元素

难度 中等 446

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]
```

```java
import java.util.*;

public class Main03 {
    public static void main(String[] args) {
        List<Integer> list;
        list = topKNums(new int[]{1,1,1,2,2,3},2);
        System.out.println(list.toString());
    }

    public static List<Integer> topKNums(int[] nums,int k){
        Map<Integer,Integer> map = new HashMap<Integer, Integer>();
        for(int i=0;i<nums.length;i++){
            if(map.containsKey(nums[i])){
                int curCount = map.get(nums[i])+1;
                map.put(nums[i],curCount);
            }else {
                map.put(nums[i],1);
            }
        }

        List<Map.Entry<Integer,Integer>> list = new ArrayList<>(map.entrySet());
//初始化
        Collections.sort(list, new Comparator<Map.Entry<Integer, Integer>>() {
            @Override
            public int compare(Map.Entry<Integer, Integer> o1,
Map.Entry<Integer, Integer> o2) {
                return o2.getValue()-o1.getValue(); //逆序
            }
        });

        List<Integer> ans = new ArrayList<>();
        for(int i=0;i<k;i++){
            ans.add(list.get(i).getKey());
        }
        return ans;
    }
```

```
}
```

H04：网格中的最短路径

给你一个 `m * n` 的网格，其中每个单元格不是 `0`（空）就是 `1`（障碍物）。每一步，您都可以在空白单元格中上、下、左、右移动。

如果您 最多 可以消除 `k` 个障碍物，请找出从左上角 `(0, 0)` 到右下角 `(m-1, n-1)` 的最短路径，并返回通过该路径所需的步数。如果找不到这样的路径，则返回 -1。

示例 1：

```
输入：
grid =
[[0,0,0],
 [1,1,0],
 [0,0,0],
 [0,1,1],
 [0,0,0]],
k = 1
输出：6
解释：
不消除任何障碍的最短路径是 10。
消除位置 (3,2) 处的障碍后，最短路径是 6 。该路径是 (0,0) -> (0,1) -> (0,2) -> (1,2) -> (2,2) -> (3,2) -> (4,2).
```

```
public int shortestPath(int[][] grid, int k) {
        int m = grid.length;
        int n = grid[0].length;
        // 非法参数处理
        if (validateInputParams(k, m, n)) {
            return -1;
        }
        // 特殊场景处理
        if (m == 1 && n == 1) {
            return 0;
        }
        // BFS搜索节点访问标识，此题要求有k个消除障碍的机会，所以每个节点除了标记是否被访问过
        // 还要记录搜索到此节点时消除了几个障碍。消除相同障碍的下一层节点 可以剪枝（因为有相同代价更早的节点了）
        // 例子：k=1，BFS是按层级来的，绕道的层级扩展越多
        // 坐标(0,2)可以为消除(0,1)障碍过来的 visited[0][2][1] = 1，搜索层级为2
        // 也可能为不消除任何障碍过来的 visited[0][2][0] = 1，层级为6，为扩展搜索不通障碍消除数提供区分
        // 0 1 0 0 0 1 0 0
        // 0 1 0 1 0 1 0 1
        // 0 0 0 1 0 0 1 0

        // 二维标记位置，第三维度标记 到此节点的路径处理障碍总个数
        int[][][] visited = new int[m][n][k+1];
        // 初始步数为0，m=n=1的特殊场景已处理
```

```
        int minSteps = 0;
        // 初始位置标记已访问
        visited[0][0][0] = 1;
        Queue<Point> queue = new LinkedList<>();
        Point startPoint = new Point(0, 0, 0);
        queue.offer(startPoint);

        // 定义四个方向移动坐标
        int[] dx = {1, -1, 0, 0};
        int[] dy = {0, 0, 1, -1};
        // BFS搜索-队列遍历
        while (!queue.isEmpty()) {
            minSteps++;
            // BFS搜索-遍历相同层级下所有节点
            // 当前队列长度一定要在循环外部定义，循环内部有插入对列操作
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                Point current = queue.poll();
                int x = current.x;
                int y = current.y;
                int oneCount = current.oneCount;

                // 对当前节点四个方向进行识别处理
                for (int j = 0; j < 4; j++) {
                    int xNew = x + dx[j];
                    int yNew = y + dy[j];
                    // 越界
                    if (xNew < 0 || xNew >= m || yNew < 0 || yNew >= n) {
                        continue;
                    }
                    // 搜索到目标节点则直接返回结果
                    if (xNew == m - 1 && yNew == n - 1) {
                        return minSteps;
                    }
                    // 穿越障碍次数已满
                    if (grid[xNew][yNew] == 1 && oneCount >= k) {
                        continue;
                    }
                    int oneCountNew = grid[xNew][yNew] == 1 ? oneCount + 1 :
oneCount;
                    // 四个方向节点是否被访问过（第三维度）
                    if (visited[xNew][yNew][oneCountNew] == 1) {
                        continue;
                    } else {
                        // 未被访问过且可以走的节点标记为访问过，对下一步节点确认状态非常重
要
                        // 将下一层级节点入队列标记为已访问，可以剪枝更多节点，节省计算耗时
                        visited[xNew][yNew][oneCountNew] = 1;
                    }
                    queue.offer(new Point(xNew, yNew, oneCountNew));
                }
            }
        }
        // BFS没搜索到目标，返回-1
        return -1;
    }

    private boolean validateInputParams(int k, int m, int n) {
```

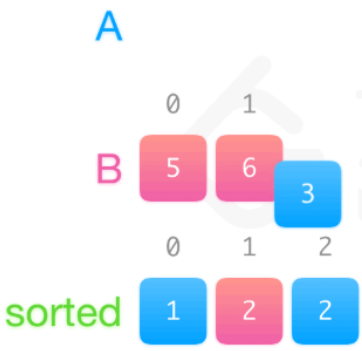
```
        return m > 40 || m < 1 || n > 40 || n < 1 || k < 1 || k > m * n;
    }

    class Point {
        int x;
        int y;
        int oneCount;

        public Point(int x, int y, int oneCount) {
            this.x = x;
            this.y = y;
            this.oneCount = oneCount;
        }
    }
```

H05:合并两个排序数组

方法 2: 双指针

算法

方法 1 没有利用数组 `A` 与 `B` 已经被排序的性质。为了利用这一性质，我们可以使用双指针方法。这一方法将两个数组看作队列，每次从两个数组头部取出比较小的数字放到结果中。如下面的动画所示：




```
class Solution {
    public void merge(int[] A, int m, int[] B, int n) {
        int pa = 0;
        int pb = 0;
        int[] sorted = new int[m+n];
        int curVal = 0;
        while(pa<m || pb<n){
            if(pa==m){              //注意判断其中一个是否到达尾部
                curVal = B[pb++];
            }else if(pb==n){
                curVal = A[pa++];
            }else if(A[pa]<B[pb]){
                curVal = A[pa++];
            }else{
                curVal = B[pb++];
            }
           sorted[pa+pb-1] = curVal;
        }
        for(int i=0;i<m+n;i++){
```

```
            A[i] = sorted[i];
        }
    }
}
```

H06:和为s的连续正数序列

```
class Solution {
    public int[][] findContinuousSequence(int target) {
        //滑动窗口解决问题
        int i = 1;  //窗口最左端
        int j = 1;  //窗口最右端  [i,j)
        List<int[]> ans = new ArrayList<>();
        int sum = 0;
        while(i<=target/2){
            if(sum<target){
                sum += j;
                j++;
            }else if(sum>target){
                sum -= i;
                i++;
            }else{
                int[] temp = new int[j-i];
                for(int k=i;k<j;k++){
                    temp[k-i] = k;
                }
                ans.add(temp);
                sum -= i;
                i++;
            }
        }
        return ans.toArray(new int[ans.size()][]);
    }
}
```

H07:最大子序和

53. 最大子序和

难度 简单 1716

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

示例:

```
输入：[-2,1,-3,4,-1,2,1,-5,4],
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6。
```

```
class Solution {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int[] dp = new int[nums.length];
```

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

}
```

H08:无重复字符的最长子串

```
public static int maxSubString(String s){
    int n = s.length();
    int ans = 0;
    int start = 0;
    Map<Character,Integer> map = new HashMap<>();
    for(int end = 0;end<n;end++){
        if(map.containsKey(s.charAt(end))){
            start = Math.max(map.get(s.charAt(end)),start);
        }
        ans = Math.max(ans,end-start+1);
        map.put(s.charAt(end),end+1);
    }
    return ans;
}
```