剑指Offer笔记

> 作者：**Peter Ho**
>
> 完结日期：**2020年10月6日**
>
> 说明：**本笔记记录了作者在刷《剑指Offer》题目过程中的心得体会以及代码注释的整理，便于后续复习，需结合Leetcode的剑指Offer题目使用。**
>
> 相关链接：**LeetCode** 、 **牛客网**

剑指Offer 34（树的路径和）
剑指Offer 35（链表拷贝）
剑指Offer 36（二叉搜索树与双向链表）
剑指Offer 37（序列化二叉树）
剑指Offer 38（不重复全排列）
剑指Offer 39（ 数组中出现次数超过一半的数字）
剑指Offer 40（最小的 K 个数）
剑指Offer 41（数据流中的中位数）
剑指Offer 42（连续子数组的最大和）
剑指Offer 43（1 ～n 整数中 1 出现的次数）
剑指Offer 44（数字序列中的某一位数字）
剑指Offer 45（把数组排成最小的数）
剑指Offer 46（把数字翻译成字符串）
剑指Offer 47（礼物的最大价值）
剑指Offer 48（最长不含重复字符的子字符串）
剑指Offer 49（丑数）
剑指Offer 50（第一个只出现一次的字符位置）
剑指Offer 51（数组中的逆序对）
剑指Offer 52（相交链表）
剑指Offer 53 - 1（有序数组同一元素的个数）
剑指Offer 53 - 2（0～n-1中缺失的数字）
剑指Offer 54（二叉搜索树的倒数第k大节点）
剑指Offer 55-1（树的深度）
剑指Offer 55-2（树的平衡）
剑指Offer 56-1（数组中数字出现的次数）
剑指Offer 56-2（数组中数字出现的次数）
剑指Offer 57-1（和为 S 的两个数字）
剑指Offer 57-2（和为 S 的连续正数序列）
剑指Offer 58-1（翻转单词顺序）
剑指Offer 58-2（左旋转字符串）
剑指Offer 59-1（滑动窗口的最大值）
剑指Offer 59-2（队列的最大值）
剑指Offer 60（n 个骰子的点数）
剑指Offer 61（扑克牌顺子）
剑指Offer 62（圆圈中最后剩下的数）
剑指Offer 63（股票的最大利润）
剑指Offer 64（求 1+2+3+...+n）
剑指Offer 65（不用加减乘除做加法）
剑指Offer 66（构建乘积数组）
剑指Offer 67（把字符串转换成整数）
剑指Offer 68-1（二叉搜索树最近公共父节点）
剑指Offer 68-2（二叉树最近公共父节点）

剑指Offer 03（数组中的重复数字）

```
// 只用了访问标记数组，思路简单
class Solution {
    public int findRepeatNumber(int[] nums) {
        int len = nums.length;
        boolean[] flag = new boolean[len];
        for (int num : nums) {
            if (flag[num]) {
                return num;
            } else{
                flag[num] = true;
            }
        }
        return 0;
    }
}
```

剑指Offer 04（二维数组的查找）

```
// 站在右上角看。这个矩阵其实就像是一个Binary Search Tree。然后，聪明的大家应该知道怎么做了。
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) {
            return false;
        }
        int row = matrix.length, col = matrix[0].length;
        int curRow = 0, curCol = col - 1;
        while (curRow < row && curCol >=0){
            if (matrix[curRow][curCol] == target){
                return true;
            }else if (matrix[curRow][curCol] < target){
                curRow++;
            }else {
                curCol--;
            }
        }
        return false;
    }
}
```

剑指Offer 05（替换空格）

```
// 写得有点简单了
class Solution {
    public String replaceSpace(String s) {
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == ' ')
                stringBuffer.append("%20");
            else
                stringBuffer.append(s.charAt(i));
        }
        return stringBuffer.toString();
    }
}
```

剑指Offer 06（从头到尾打印链表）

```
// 注意边界值就好
class Solution {
    public int[] reversePrint(ListNode head) {
        ListNode node = head;
        int len = 0;
        while (node != null){
            node = node.next;
            len++;
        }
        int[] ans = new int[len];
        for (int i = len - 1; i >= 0; i--) {
            ans[i] = head.val;
            head = head.next;
        }
        return ans;
    }
}
```

剑指Offer 07（重构二叉树）

```
// 不会，没理解
class Solution {
    //利用原理,先序遍历的第一个节点就是根。在中序遍历中通过根 区分哪些是左子树的，哪些是右子树的
    //左右子树，递归
    HashMap<Integer, Integer> map = new HashMap<>();//标记中序遍历
```

```
    int[] preorder;//保留的先序遍历

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        this.preorder = preorder;
        for (int i = 0; i < preorder.length; i++) {
            map.put(inorder[i], i);
        }
        return recursive(0,0,inorder.length - 1);
    }

     /**
     * @param pre_root_idx  先序遍历的索引
     * @param in_left_idx  中序遍历的索引
     * @param in_right_idx 中序遍历的索引
     */
    public TreeNode recursive(int pre_root_idx, int in_left_idx, int
in_right_idx) {
        //相等就是自己
        if (in_left_idx > in_right_idx) {
            return null;
        }
        //root_idx是在先序里面的
        TreeNode root = new TreeNode(preorder[pre_root_idx]);
        // 有了先序的,再根据先序的，在中序中获 当前根的索引
        int idx = map.get(preorder[pre_root_idx]);

        //左子树的根节点就是 左子树的(前序遍历）第一个，就是+1,左边边界就是left，右边边界是
中间区分的idx-1
        root.left = recursive(pre_root_idx + 1, in_left_idx, idx - 1);

        //由根节点在中序遍历的idx 区分成2段,idx 就是根

        //右子树的根，就是右子树（前序遍历）的第一个,就是当前根节点 加上左子树的数量
        // pre_root_idx 当前的根  左子树的长度 = 左子树的左边-右边 (idx-1 -
in_left_idx +1) 。最后+1就是右子树的根了
        root.right = recursive(pre_root_idx + (idx-1 - in_left_idx +1)  + 1,
idx + 1, in_right_idx);
        return root;
    }
}
```

剑指Offer 09（用两个栈实现队列）

```
// 很简单的思路，但只击败10%
class CQueue {
    Stack<Integer> mainQueue;
    Stack<Integer> temp;

    public CQueue() {
        mainQueue = new Stack<>();
        temp = new Stack<>();
    }
    public void appendTail(int value) {
        while (!mainQueue.isEmpty()){
            temp.push(mainQueue.pop());
        }
        mainQueue.push(value);
        while (!temp.isEmpty()){
            mainQueue.push(temp.pop());
        }
    }

    public int deleteHead() {
        if(mainQueue.isEmpty()){
            return -1;
        }
        return mainQueue.pop();
    }
}
```

剑指Offer 10-1（斐波那契数列）

```
// 别读错题，要模的
class Solution {
    public int fib(int n) {
        if (n == 0 || n == 1) return n;
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;
        }
        return dp[n];
    }
}
```

剑指Offer 10-2（青蛙跳台阶）

```
// 斐波那契小改
class Solution {
    public int numWays(int n) {
        if (n == 0 || n == 1) return 1;
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;
        }
        return dp[n];
    }
}
```

剑指Offer 11（旋转数组中的最小数字）

```
// 无语
class Solution {
    public int minArray(int[] numbers) {
        Arrays.sort(numbers);
        return numbers[0];
    }
}
// 二分法
class Solution {
    public int minArray(int[] numbers) {
        int l = 0, r = numbers.length - 1;
        while (l < r) {
            int mid = ((r - l) >> 1) + l;
            //只要右边比中间大，那右边一定是有序数组
            if (numbers[r] > numbers[mid]) {
                r = mid;
            } else if (numbers[r] < numbers[mid]) {
                l = mid + 1;
             //去重
            } else r--;
        }
        return numbers[l];
    }
}
```

剑指Offer 12（矩阵单词查找）

```
class Solution {
    private int[][] direction = new int[][]{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
    private int row;
    private int col;

    public boolean exist(char[][] board, String word) {
        if (board == null || board.length ==0)
            return false;
        row = board.length;
        col = board[0].length;

        boolean[][] vis = new boolean[row][col];
        for (int r = 0; r < row; r++) {
            for (int c = 0; c < col; c++) {
                if (dfs(0, r, c, vis, board, word)){
                    return true;
                }
            }
        }
        return false;
    }

    private boolean dfs(int curLen, int r, int c, boolean[][] vis, char[][]
board, String word){
        // 长度相等return true
        if (curLen == word.length())
            return true;
        // 边界过滤条件
        // 判断单词的此位是否相等
        if (r < 0 || r >= row || c < 0 || c >= col || board[r][c] !=
word.charAt(curLen) || vis[r][c]) {
            return false;
        }
        // 标记为已访问
        vis[r][c] = true;
        // 对四个方向回溯
        for (int[] d : direction){
            if (dfs(curLen + 1, r + d[0], c + d[1], vis, board, word))
                return true;
        }
        // 下一步要return，所以要除去访问标记
        vis[r][c] = false;
        return false;
    }
}
```

剑指Offer 13（机器人的运动范围）

```
class Solution {
    boolean[][] visit;
    public int movingCount(int m, int n, int k) {
        // 第一步：先明确递归参数
        // 第二步：明确递归终止条件
        visit = new boolean[m][n];
        return dfs(0, 0, m, n, k);
    }

    private int dfs(int row, int col, int m, int n, int k) {
        // 第一步：先明确递归参数
        int a = sums(row);
        int b = sums(col);
        // 第二步：明确递归终止条件
        // 边界判断、位数和与k比较、当前点是否访问过
        if (row < 0 || row >= m || col < 0 || col >= n || k < a + b ||
visit[row][col]) {
            return 0;
        }
        // 第三步：递推工作
        visit[row][col] = true;
        return 1 + dfs(row + 1, col, m, n, k) + dfs(row, col + 1, m, n, k);
    }

    // 位数之和计算
    private int sums(int num) {
        int sums = 0;
        while (num != 0) {
            sums += num % 10;
            num = num / 10;
        }
        return sums;
    }
}
```

剑指Offer 14-1（剪绳子1）

```
// 动态规划
class Solution {
    public int cuttingRope(int n) {
        int[] dp = new int[n + 1];
        dp[1] = 1;
```

```
        for (int i = 2; i <= n; i++)
            for (int j = 1; j < i; j++)
                dp[i] = Math.max(dp[i], Math.max(j * (i - j), dp[j] * (i -
j)));
        return dp[n];
    }
}
//递归，反而更快
class Solution {
    public int cuttingRope(int n) {
        if(n == 2) return 1;
        if(n == 3) return 2;
        if(n == 4) return 4;
        if(n == 5) return 6;
        if(n == 6) return 9;
        return 3 * cuttingRope(n - 3);
    }
}
```

剑指Offer 14-2（剪绳子2）*

```
// 再看看吧
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        long res = 1L;
        int p = (int) 1e9+7;
        //贪心算法，优先切三，其次切二
        while(n>4){
            res = res * 3 % p;
            n -= 3;
        }
        //出来循环只有三种情况，分别是n=2、3、4
        return (int)(res * n % p);
    }
}
```

剑指Offer 15（二进制中 1 的个数）

```
// 位运算
class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
```

```
        while (n != 0) {
            // 如果是0结尾，向前借1，就会导致'与'运算之后相较于运算之前相比少一个1，如果是1同样也导致运算之后相较于运算之前少一个1。循环往复。
            count += n & 1;
            n = n >>> 1;
        }
        return count;
    }
}

class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        return Integer.bitCount(n);
    }
}
```

剑指Offer 16（数值的整数次方）

```
// cv大法
// 递归
class Solution {
    public double myPow(double x, int n) {
        if(n == 0) return 1;
        if(n == 1) return x;
        if(n == -1) return 1 / x;
        double half = myPow(x, n / 2);
        double mod = myPow(x, n % 2);
        return half * half * mod;
    }
}

//
class Solution {
    public double myPow(double x, int n) {
        if(x == 0) return 0;
        long b = n;
        double res = 1.0;
        if(b < 0) {
            x = 1 / x;
            b = -b;
        }
        while(b > 0) {
            if((b & 1) == 1) res *= x;
            x *= x;
            b >>= 1;
        }
```

```
        return res;
    }
}
```

剑指Offer 17（打印数组）*

```
// 这个太简单，考虑一下大数问题
class Solution {
    public int[] printNumbers(int n) {
        int limit = (int) Math.pow(10, n) - 1;
        int[] ans = new int[limit];
        for (int i = 0; i < limit; i++) {
            ans[i] = i + 1;
        }
        return ans;
    }
}
```

剑指Offer 18（在 O(1) 时间内删除链表节点）

```
// 利用了两个指针记录
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        ListNode curr = head;
        ListNode pre = new ListNode(-1);
        if (head.val == val)
            return head.next;
        while (curr != null){
            if (curr.val == val){
                pre.next = curr.next;
                break;
            }
            pre = curr;
            curr = curr.next;
        }
        return head;
    }
}
```

剑指Offer 19（正则表达式匹配）

```

```

剑指Offer 20（表示数值的字符串）

```
// cv大法
class Solution {
    public boolean isNumber(String s) {
        if(s == null || s.length() == 0){
            return false;
        }
        //标记是否遇到相应情况
        boolean numSeen = false;
        boolean dotSeen = false;
        boolean eSeen = false;
        char[] str = s.trim().toCharArray();
        for(int i = 0;i < str.length; i++){
            if(str[i] >= '0' && str[i] <= '9'){
                numSeen = true;
            }else if(str[i] == '.'){
                //.之前不能出现.或者e
                if(dotSeen || eSeen){
                    return false;
                }
                dotSeen = true;
            }else if(str[i] == 'e' || str[i] == 'E'){
                //e之前不能出现e，必须出现数
                if(eSeen || !numSeen){
                    return false;
                }
                eSeen = true;
                numSeen = false;//重置numSeen，排除123e或者123e+的情况,确保e之后也出
现数
            }else if(str[i] == '-' || str[i] == '+'){
                //+-出现在0位置或者e/E的后面第一个位置才是合法的
                if(i != 0 && str[i-1] != 'e' && str[i-1] != 'E'){
                    return false;
                }
            }else{//其他不合法字符
                return false;
            }
        }
        return numSeen;
    }
}
```

剑指Offer 21（调整数组顺序使奇数位于偶数前面）

```
// 双指针法交换，太慢了
class Solution {
    public int[] exchange(int[] nums) {
        // 双指针法
        int left = 0, right = nums.length - 1;
        int temp;
        while (left < right){
             if (isOdd(nums[left])) {// 前奇
                left++;
            }else if (!isOdd(nums[right])){// 后偶
                right--;
            }else {// 前偶后奇
                temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
                left++;
                right--;
            }
        }
        return nums;
    }

    private boolean isOdd(int num){
        return num % 2 == 1;
    }
}
```

剑指Offer 22（链表中倒数第 K 个结点）

```
// 典型的双指针问题
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode p1 = head, p2 = head;
        for (int i = 1; i < k; i++) {
            p1 = p1.next;
        }
        while (p1.next != null){
            p1 = p1.next;
            p2 = p2.next;
        }
        return p2;
    }
```

```
}
```

剑指Offer 24（反转链表）

```
// 三指针法，需注意指针初始化为null
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode cur = head;
        ListNode next = null;
        while (cur != null){
            next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
}
```

剑指Offer 25（合并有序链表）

```
// 链表类问题，设置dummyHead是一个常规操作，主要是为了避免讨论头节点，倒不一定是头节点丢失。
// 这个题如果你不用dummyHead，你就需要去讨论头节点到底是l1还是l2。
// 而如果是删除倒数第n个节点那个题，如果不采用dummyHead，你就需要单独去讨论如果删除的是倒数第n个节点，也就是head被删除的情况。
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode(-1), pre = dummyHead;
        while (l1 != null && l2 != null){
            if (l1.val <= l2.val) {
                pre.next = l1;
                pre = l1;
                l1 = l1.next;
            }else {
                pre.next = l2;
                pre = l2;
                l2 = l2.next;
            }
        }
        if (l1 != null) {
            pre.next = l1;
        }
        if (l2 != null) {
```

```
            pre.next = l2;
        }
        return dummyHead.next;
    }
}
```

剑指Offer 26（树的子结构）*

```
// 自己写不出啊
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(A == null || B == null) return false;
        return dfs(A, B) || isSubStructure(A.left, B) ||
isSubStructure(A.right, B);
    }
    public boolean dfs(TreeNode A, TreeNode B){
        if(B == null) return true;
        if(A == null) return false;
        return A.val == B.val && dfs(A.left, B.left) && dfs(A.right, B.right);
    }
}
```

剑指Offer 27（树的镜像）

```
// 递归，注意一下其他的方法，用栈或队列
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if (root == null) return null;
        TreeNode tempLeft = root.left;//后面的操作会改变 left 指针，因此先保存下来
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(tempLeft);
        return root;
    }
}
```

剑指Offer 28（树的对称）

> 做递归思考三步：
>
> 1. 递归的函数要干什么？
>
> - 函数的作用是判断传入的两个树是否镜像。
> - 输入：TreeNode left, TreeNode right

- 输出：是：true，不是：false

2. 递归停止的条件是什么？

- 左节点和右节点都为空 -> 倒底了都长得一样 ->true
- 左节点为空的时候右节点不为空，或反之 -> 长得不一样-> false
- 左右节点值不相等 -> 长得不一样 -> false

3. 从某层到下一层的关系是什么？

- 要想两棵树镜像，那么一棵树左边的左边要和二棵树右边的右边镜像，一棵树左边的右边要和二棵树右边的左边镜像
- 调用递归函数传入左左和右右
- 调用递归函数传入左右和右左
- 只有左左和右右镜像且左右和右左镜像的时候，我们才能说这两棵树是镜像的

4. 调用递归函数，我们想知道它的左右孩子是否镜像，传入的值是root的左孩子和右孩子。这之前记得判个root==null

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

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

剑指Offer 29（顺时针打印矩阵）

```
class Solution {
    // 第一位表示行变化，第二位表示列变化
    // 0代表不变，1代表增加，-1代表减少
    // 四行分别代表：右，下，左，上
    int[][] d = {{0,1}, {1,0}, {0, -1}, {-1, 0}};
    public int[] spiralOrder(int[][] matrix) {
        int n = matrix.length;
        if (n == 0)
            return new int[0];
        int m = matrix[0].length;
        int[] res = new int[n * m];
```

```
        boolean[][] vis = new boolean[n][m];
        int r = 0, c = 0;
        int index = 0;
        // 初始化 方向。往左
        int dir = 0;
        while (index < n * m) {
            res[index++] = matrix[r][c];
            // 标记当前点已经访问过
            vis[r][c] = true;
            int nextR = r + d[dir % 4][0];
            int nextC = c + d[dir % 4][1];
            // 边界判断
            // 最外圈，0 <= nextR < n-1，0 <= nextC <= m-1
            // 其余位置根据vis数据判断，只要访问过就算达到边界
            if (nextR == n || nextR < 0 || nextC == m || nextC < 0 ||
vis[nextR][nextC]) {
                // 达到边界转方向
                dir += 1;
                nextR = r + d[dir % 4][0];
                nextC = c + d[dir % 4][1];
            }
            // 更新当前点
            r = nextR;
            c = nextC;
        }
        return res;
    }
}
```

剑指Offer 30（包含 min 函数的栈）

```
class MinStack {

    // 数据栈和最小值栈
    Stack<Integer> dataStack, minStack;
    public MinStack() {
        dataStack = new Stack<>();
        minStack = new Stack<>();
    }

    public void push(int x) {
        // 数据栈添加数据
        dataStack.add(x);
        // 若最小值栈为空 或者
        // 加入的值小于等于当前最小值，添加最小值
        if(minStack.empty() || minStack.peek() >= x)
            minStack.add(x);
```

```
    }

    public void pop() {
        // 数据栈pop，若数据栈pop的值是最小值则最小值栈也要pop
        if(dataStack.pop().equals(minStack.peek()))
            minStack.pop();
    }

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

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

剑指Offer 31（栈的压入、弹出序列）

```
// 根据两个数组模拟栈的出入
class Solution {
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        Stack<Integer> stack = new Stack<>();
        int i = 0;
        for(int num : pushed) {
            stack.push(num); // num 入栈
            // 只有当此时栈顶元素等于此时poped[i]，出栈，并且i++
            while(!stack.isEmpty() && stack.peek() == popped[i]) { // 循环判断与
出栈
                stack.pop();
                i++;
            }
        }
        // 模拟成功则栈为空
        return stack.isEmpty();
    }
}
```

剑指Offer 32-1（从上往下打印二叉树）

```
// 就一个层次遍历，不是很懂为什么是medium
class Solution {
    public int[] levelOrder(TreeNode root) {
        if (root == null)
```

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

剑指Offer 32-2（把二叉树打印成多行）

```
// 自己的方法用了两个队列
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ansList = new ArrayList<>();
        if (root == null)
            return ansList;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
            List<Integer> list = new ArrayList<>();
            Queue<TreeNode> tempQueue = new LinkedList<>();
            while (!queue.isEmpty()){
                tempQueue.add(queue.poll());
            }
            while (!tempQueue.isEmpty()){
                if (tempQueue.peek().left != null)
                    queue.add(tempQueue.peek().left);
                if (tempQueue.peek().right != null)
                    queue.add(tempQueue.peek().right);
                list.add(tempQueue.poll().val);
            }
            ansList.add(new ArrayList<>(list));
        }
        return ansList;
    }
```

```
}
```

剑指Offer 32-3（按之字形顺序打印二叉树）

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ansList = new ArrayList<>();
        if (root == null)
            return ansList;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
            List<Integer> list = new ArrayList<>();
            Queue<TreeNode> tempQueue = new LinkedList<>();
            while (!queue.isEmpty()){
                tempQueue.add(queue.poll());
            }
            while (!tempQueue.isEmpty()){
                if (tempQueue.peek().left != null)
                    queue.add(tempQueue.peek().left);
                if (tempQueue.peek().right != null)
                    queue.add(tempQueue.peek().right);
                list.add(tempQueue.poll().val);
            }
            ansList.add(new ArrayList<>(list));
        }
        // 与上一题的主要区别
        for (int i = 1; i < ansList.size(); i += 2) {
            Collections.reverse(ansList.get(i));
        }
        return ansList;
    }
}
```

剑指Offer 33（二叉搜索树的后序遍历序列）

```

```

剑指Offer 34（树的路径和）

```
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> ans = new ArrayList<>();
        dfs(root, sum, 0, new ArrayList<>(), ans);
        return ans;
    }

    private void dfs(TreeNode root, int sum, int curSum, List<Integer> list,
List<List<Integer>> ans){
        // 节点为空直接返回
        if (root == null)
            return;
        // 将当前节点的值加入到list中
        list.add(root.val);
        // 每往下走一步就要计算走过的路径和
        curSum += root.val;
        // 如果到达叶子节点，就不能往下走了，直接return
        if (root.left == null && root.right == null){
            if (sum == curSum)
                // 此处一定要new
                ans.add(new ArrayList<>(list));
            // 需要将最后加入的节点给移除掉，
            // 因为下一步直接return了，不会再走最后一行的remove了，
            // 所以这里在return之前提前把最后一个结点的值给remove掉。
            list.remove(list.size() - 1);
            return;
        }
        // 如果没到达叶子节点，就继续从他的左右两个子节点往下找
        dfs(root.left, sum, curSum, list, ans);
        dfs(root.right, sum, curSum, list, ans);
        // 我们要理解递归的本质，当递归往下传递的时候他最后还是会往回走，
        // 我们把这个值使用完之后还要把它给移除，这就是回溯
        list.remove(list.size() - 1);
    }
}
```

剑指Offer 35（链表拷贝）

> 太菜了，没明白题意。
>
> 浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内
>
> 存，修改新对象不会改到原对象。

```
class Solution {
    public Node copyRandomList(Node head) {
        if (head == null) {
            return head;
        }
        //map中存的是(原节点，拷贝节点)的一个映射
        Map<Node, Node> map = new HashMap<>();
        for (Node cur = head; cur != null; cur = cur.next) {
            map.put(cur, new Node(cur.val));
        }
        //将拷贝的新的节点组织成一个链表
        for (Node cur = head; cur != null; cur = cur.next) {
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
        }

        return map.get(head);
    }
}
```

剑指Offer 36（二叉搜索树与双向链表）

```

```

剑指Offer 37（序列化二叉树）

```
// 不会
```

剑指Offer 38（不重复全排列）

```
class Solution {
    List<String> res = new ArrayList<>();
    char[] c;
    public String[] permutation(String s) {
        c = s.toCharArray();
        dfs(0);
        return res.toArray(new String[res.size()]);
    }
    void dfs(int x) {
        if(x == c.length - 1) {
            res.add(String.valueOf(c)); // 添加排列方案
            return;
```

```
        }
        HashSet<Character> set = new HashSet<>();
        for(int i = x; i < c.length; i++) {
            if(set.contains(c[i])) continue; // 重复，因此剪枝
            set.add(c[i]);
            swap(i, x); // 交换，将 c[i] 固定在第 x 位
            dfs(x + 1); // 开启固定第 x + 1 位字符
            swap(i, x); // 恢复交换
        }
    }
    void swap(int a, int b) {
        char tmp = c[a];
        c[a] = c[b];
        c[b] = tmp;
    }
}
```

剑指Offer 39（ 数组中出现次数超过一半的数字）

```
class Solution {
    public int majorityElement(int[] nums) {
        // 需要的数字出现次数多于一半 那么排序后必定在中间
        Arrays.sort(nums);
        return nums[nums.length / 2];
    }
}
```

剑指Offer 40（最小的 K 个数）

```
// Arrays.sort()
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        Arrays.sort(arr);
        int[] ans = new int[k];
        for (int i = 0; i < k; i++) {
            ans[i] = arr[i];
        }
        return ans;
    }
}
```

剑指Offer 41（数据流中的中位数）

```

```

剑指Offer 42（连续子数组的最大和）

```
class Solution {
    public int maxSubArray(int[] nums) {
        // 动态规划
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        int max = dp[0];
        for(int i = 1; i < nums.length; i++){
            dp[i] = Math.max(dp[i-1] + nums[i], nums[i]);
            max = Math.max(max, dp[i]);
        }
        return max;
    }
}
```

剑指Offer 43（1 ～n 整数中 1 出现的次数）

```

```

剑指Offer 44（数字序列中的某一位数字）

```

```

剑指Offer 45（把数组排成最小的数）

```

```

剑指Offer 46（把数字翻译成字符串）

```

```

剑指Offer 47（礼物的最大价值）

```

```

剑指Offer 48（最长不含重复字符的子字符串）

```
// 哈希表解法，滑动窗口
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) return 0;
        Map<Character, Integer> map = new HashMap<>();
        int ans = 0;
        for (int end = 0, start = 0; end < s.length(); end++) {
            // 如果当前出现重复字母，窗口左指针右移到重复字母上次出现的右一位
            if (map.containsKey(s.charAt(end))) {
                start = Math.max(map.get(s.charAt(end))+1, start);
            }
            // 不断更新的窗口长度与原来所求结果的最大值
            ans = Math.max(ans, end - start + 1);
            // 将出现字母与最新出现的位置存入HashMap
            map.put(s.charAt(end), end);
        }
        return ans;
    }
}
```

剑指Offer 49（丑数）

```

```

剑指Offer 50（第一个只出现一次的字符位置）

```
class Solution {
    public char firstUniqChar(String s) {
        Map<Character, Integer> map = new HashMap<>();
        for (Character c : s.toCharArray()) {
            //getOrDefault意思就是当Map集合中有这个key时，就使用这个key值，如果没有就使用默认值defaultValue
            //此处存入的是每个元素出现的次数
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        // 按照字符出现顺序，查找出现近一次的字符。用foreach 顺序不对
        for (int i = 0; i < s.length(); i++) {
            if (map.get(s.charAt(i))==1)
                return s.charAt(i);
        }
        return ' ';
    }
}
```

剑指Offer 51（数组中的逆序对）

```

```

剑指Offer 52（相交链表）

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode pA = headA, pB = headB;
        //判断两个指针所指节点的值是否相同，若不相同则执行循环语句
        //总体思路是指针各自遍历一遍链表，遍历完成后，然后遍历另一条链表，当指针指向同一元素时，表明此处为相交节点
        //若无相交节点，则遍历完两条链表，pA与pB都为null，此时 pA == pB，跳出循环
        while(pA != pB) {
            //若pA为空，则pA指向headB，否则指向下一点
            pA = pA == null ? headB : pA.next;
            //若pB为空，则pB指向headA，否则指向下一点
            pB = pB == null ? headA : pB.next;
        }
        return pA;
    }
```

```
}
```

剑指Offer 53 - 1（有序数组同一元素的个数）

```
class Solution {
    public int search(int[] nums, int target) {
        // 非空判断
        if(nums == null || nums.length == 0) {
            return 0;
        }
        int ans = 0;
        // 二分查找
        int low = 0, high = nums.length - 1;
        int mid = (low + high) / 2;
        while(low <= high) {
            if(nums[mid] == target) {
                ans = count(nums, mid, target);
                break;
            }
            if(nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
            mid = (low + high) / 2;
        }
        return ans;
    }

    // 向两边扩张计算出个数
    private int count(int[] nums, int cur, int target){
        int l = cur - 1, h = cur + 1;
        int ans = 1;
        while(l >= 0 && nums[l] == target) {
            l--;
            ans++;
        }
        while(h < nums.length && nums[h] == target) {
            h++;
            ans++;
        }
        return ans;
    }
}
```

剑指Offer 53 - 2（0～n-1中缺失的数字）

```
class Solution {
    public int missingNumber(int[] nums) {
        // 二分法，最后返回low或者high都可
        int low = 0, high = nums.length - 1;
        while(low <= high) {
            int mid = (low + high) / 2;
            if(nums[mid] == mid)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return low;
    }
}

// 理论的和减去实际的和即为缺少的数字
class Solution {
    public int missingNumber(int[] nums) {
        //计算出0-n的和  n*(n+1)/2
        int sum = nums.length * (nums.length+1)/2;
        return  sum - Arrays.stream(nums).sum() ;
    }
}
```

剑指Offer 54（二叉搜索树的倒数第k大节点）

```
class Solution {
    // 先中序遍历，存入list，再取倒数第k项
    public int kthLargest(TreeNode root, int k) {
        List<Integer> treeList = new ArrayList<>();
        helper(root, treeList);
        return treeList.get(treeList.size() - k);
    }

    private void helper(TreeNode root, List<Integer> treeList){
        if (root == null) return;
        if (root.left != null) helper(root.left, treeList);
        treeList.add(root.val);
        if (root.right != null) helper(root.right, treeList);
    }
}

// 做了点小变化，反中序遍历，直接得到逆序列
class Solution {
```

```
     public int kthLargest(TreeNode root, int k) {
        List<Integer> treeList = new ArrayList<>();
        helper(root, treeList);
        return treeList.get(k - 1);
    }

    private void helper(TreeNode root, List<Integer> treeList){
        if (root == null) return;
        if (root.right != null) helper(root.right, treeList);
        treeList.add(root.val);
        if (root.left != null) helper(root.left, treeList);
    }
}

// 思路同上，但直接return答案，不再用list
class Solution {
    private int ans = 0, cnt = 0;
    public int kthLargest(TreeNode root, int k) {
        helper(root, k);
        return ans;
    }

    private void helper(TreeNode root,  int k){
        if (root == null) return;
        if (root.right != null) helper(root.right, k);
        if (++cnt == k){
            ans = root.val;
        }
        if (root.left != null) helper(root.left, k);
    }
}
```

剑指Offer 55-1（树的深度）

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

//层次遍历，AC速度还不如递归。但是面试可能考
class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;
```

```
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        int depth = 0;
        while(!queue.isEmpty()){
            // 需要先记录当前层的长度，因为queue.size()会变
            int curSize = queue.size();
            for(int i = 0; i < curSize; i++){
                TreeNode temp = queue.poll();
                if(temp.left != null) queue.add(temp.left);
                if(temp.right != null) queue.add(temp.right);
            }
            depth++;
        }
        return depth;
    }
}
```

剑指Offer 55-2（树的平衡）

```
// 结合55-1的经典递归
class Solution {
    public boolean isBalanced(TreeNode root) {
        if(root==null) return true;
        if(Math.abs(maxDepth(root.left) - maxDepth(root.right)) <= 1){
            return isBalanced(root.left) && isBalanced(root.right);
        }
        return false;
    }

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

剑指Offer 56-1（数组中数字出现的次数）

```
// 哈希表 思路简单，可考虑位运算
class Solution {
    public int[] singleNumbers(int[] nums) {
        int[] ans = new int[2];
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
```

```
            map.put(nums[i],map.getOrDefault(nums[i],0) + 1);
        }
        int i = 0;
        for (int num : map.keySet()) {
            if (map.get(num) == 1) {
                ans[i] = num;
                i++;
            }
        }
        return ans;
    }
}
```

剑指Offer 56-2（数组中数字出现的次数）

```
// 哈希表 思路简单，可考虑位运算
class Solution {
    public int singleNumber(int[] nums) {
        int ans = 0;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i],map.getOrDefault(nums[i],0) + 1);
        }
        for (int num : map.keySet()) {
            if (map.get(num) == 1) {
                ans = num;
                break;
            }
        }
        return ans;
    }
}
```

剑指Offer 57-1（和为 S 的两个数字）

```
// 本题是有序的，可优化。可用双指针
class Solution {
    public int[] twoSum(int[] nums, int target) {
        // 用 HashMap 存储数组元素和索引的映射，
        // 在访问到 nums[i] 时，判断 HashMap 中是否存在 target - nums[i]，
        HashMap<Integer, Integer> indexForNum = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (indexForNum.containsKey(target - nums[i])) {
```

```
                return new int[]{target - nums[i], nums[i]};
            } else {
                indexForNum.put(nums[i], i);
            }
        }
        return null;
    }
}
```

剑指Offer 57-2（和为 S 的连续正数序列）

```
class Solution {
    public int[][] findContinuousSequence(int target) {
        List<int[]> list = new ArrayList<>();

        // 脑子里要有一个区间的概念，这里的区间是(1, 2, 3, ..., target - 1)
        // 套滑动窗口模板，l是窗口左边界，r是窗口右边界，窗口中的值一定是连续值。
        // 当窗口中数字和小于target时，r右移；大于target时，l右移；等于target时就获得了
一个解
        for (int l = 1, r = 1, sum = 0; r <= target/2 + 1; r++) {
            sum += r;
            // 左指针右移
            while (sum > target) {
                sum -= l;
                l++;
            }
            if (sum == target) {
                int[] temp = new int[r - l + 1];
                for (int i = 0; i < temp.length; i++) {
                    temp[i] = l + i;
                }
                list.add(temp);
            }
        }

        int[][] res = new int[list.size()][];
        for (int i = 0; i < res.length; i++) {
            res[i] = list.get(i);
        }
        return res;
    }
}
```

剑指Offer 58-1（翻转单词顺序）

```
class Solution {
    public String reverseWords(String s) {
        String[] strings = s.trim().split(" "); // 删除首尾空格，分割字符串
        StringBuilder res = new StringBuilder();
        for(int i = strings.length - 1; i >= 0; i--) { // 倒序遍历单词列表
            if(strings[i].equals("")) continue; // 遇到空单词则跳过
            res.append(strings[i] + " "); // 将单词拼接至 StringBuilder
        }
        return res.toString().trim(); // 转化为字符串，删除尾部空格，并返回
    }
}
```

剑指Offer 58-2（左旋转字符串）

```
class Solution {
    public String reverseLeftWords(String s, int n) {
        StringBuilder ans = new StringBuilder();
        ans.append(s.substring(n, s.length()));
        ans.append(s.substring(0, n));
        return ans.toString();
    }
}
```

剑指Offer 59-1（滑动窗口的最大值）

```
// 自己的太慢了
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums == null || nums.length == 0) {
            return new int[0];
        }
        int len = nums.length;
        int[] ans = new int[len - k + 1];
        for (int l = 0, r = k - 1; l < len - k + 1 ; l++) {
            ans[l] = getMax(nums, l, r);
            r++;
        }
        return ans;
    }

    private int getMax(int[] nums, int l, int r){
        int max = Integer.MIN_VALUE;
```

```
        for (int i = l; i <= r; i++) {
            max = Math.max(nums[i], max);
        }
        return max;
    }
}

// LC上需要在线性时间复杂度内解决此题
```

剑指Offer 59-2（队列的最大值）

```

```

剑指Offer 60（n 个骰子的点数）

```

```

剑指Offer 61（扑克牌顺子）

```

```

剑指Offer 62（圆圈中最后剩下的数）

```

```

剑指Offer 63（股票的最大利润）

```

```

剑指Offer 64（求 1+2+3+...+n）

```

```

剑指Offer 65（不用加减乘除做加法）

```
// 位运算真不会，得看
class Solution {
    public int add(int a, int b) {
        return b == 0 ? a : add(a ^ b, (a & b) << 1);
    }
}
```

剑指Offer 66（构建乘积数组）

```

```

剑指Offer 67（把字符串转换成整数）

```

```

剑指Offer 68-1（二叉搜索树最近公共父节点）

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        //当p和q节点等于root节点，直接返回root
        if (p.val == root.val || q.val == root.val){
            return root;
        }
        //递归求解，利用二叉搜索树性质，切记
        if (p.val > root.val && q.val > root.val) {//p和q节点都大于root，则说明p和q
为root的右子节点
            return lowestCommonAncestor(root.right, p, q);
        }else if (p.val < root.val && q.val < root.val) {//p和q节点都小于root，则
说明p和q为root的左子节点
            return lowestCommonAncestor(root.left, p, q);
        }else{//其他情况均为root节点为最大父节点
            return root;
        }
    }
}
```

剑指Offer 68-2（二叉树最近公共父节点）

```
// 递归解法
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        /**
         注意p,q必然存在树内，且所有节点的值唯一!!!
         递归思想，对以root为根的(子)树进行查找p和q，如果root == null || p || q 直接返
回root
         表示对于当前树的查找已经完毕，否则对左右子树进行查找，根据左右子树的返回值判断：
         1. 左右子树的返回值都不为null，由于值唯一左右子树的返回值就是p和q，此时root为LCA
         2. 如果左右子树返回值只有一个不为null，说明只有p和q存在与左或右子树中，最先找到的
那个节点为LCA
         3. 左右子树返回值均为null，p和q均不在树中，返回null
         **/
        if (root == null || root == p || root == q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left != null && right != null) {
            return root;
        } else if (left != null) {
            return left;
        } else if (right != null) {
            return right;
        }
        return null;
    }
}

// 别人的代码注解
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        if(root == null) return null; // 如果树为空，直接返回null
        if(root == p || root == q) return root; // 如果 p和q中有等于 root的，那么它
们的最近公共祖先即为root（一个节点也可以是它自己的祖先）
        TreeNode left = lowestCommonAncestor(root.left, p, q); // 递归遍历左子树，
只要在左子树中找到了p或q，则先找到谁就返回谁
        TreeNode right = lowestCommonAncestor(root.right, p, q); // 递归遍历右子
树，只要在右子树中找到了p或q，则先找到谁就返回谁
        if(left == null) return right; // 如果在左子树中 p和 q都找不到，则 p和 q一定
都在右子树中，右子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
        else if(right == null) return left; // 否则，如果 left不为空，在左子树中有找
到节点（p或q），这时候要再判断一下右子树中的情况，如果在右子树中，p和q都找不到，则 p和q一定都
在左子树中，左子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
        else return root; //否则，当 left和 right均不为空时，说明 p、q节点分别在 root
异侧，最近公共祖先即为 root
    }
```

```
}
```