PeterHo的LeetCode笔记

> 本文从 Leetcode 中精选大概 200 左右的题目，去除了某些繁杂但是没有多少算法思想的题目，同时保留了面试中经常被问到的经典题目。

数据结构相关

链表

1. 找出两个链表的交点 --160--Easy

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        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，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;
    }
}
```

2. 链表反转--206--Easy

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode newHead = null;
        //使用临时节点进行交换，实际上就是改变每个节点指的方向，并将原末尾节点置为头节
点
        while (head != null) {
            ListNode nextNode = head.next;
            head.next = newHead;
            newHead = head;
            head = nextNode;
        }
        return newHead;
    }
}
```

3. 归并两个有序的链表--21--Easy

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null) {
            return l2;
        }
        if(l2 == null) {
            return l1;
        }
        //递归法解
        if(l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

4. 从有序链表中删除重复节点--83--Easy

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

5. 删除链表的倒数第 n 个节点--19--Medium

```
//思路是先让一个指针先跑n步
public class Test19 {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode node1 = head;
        ListNode node2 = head;
        for (int i = 0; i < n; i++) {
            node1 = node1.next;
        }
        if (node1 == null)
            return head.next;
        while (node1.next != null){
            node1 = node1.next;
            node2 = node2.next;
        }
        node2.next = node2.next.next;
        return head;
    }
}
```

6. 交换链表中的相邻结点--24--Medium

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        // node1与node2为需要交换的节点，pre与next节点为这两个节点的前置与后置节点。
        ListNode node1, node2, pre, next;
        // 构造一个前置节点
        ListNode node = new ListNode(-1);
        node.next = head;
        pre = node;
```

```
        while (pre.next != null && pre.next.next != null) {
            // 赋值node1, node2, next节点
            node1 = pre.next;
            node2 = pre.next.next;
            next = node2.next;
            // 交换节点
            node1.next = next;
            node2.next = node1;
            pre.next = node2;
            // 将前置节点替换为下一轮的前置节点
            pre = node1;
        }
        return node.next;
    }
}
```

7. 链表求和--445--Medium

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        Stack<Integer> stack1 = buildStack(l1);
        Stack<Integer> stack2 = buildStack(l2);
        ListNode head = new ListNode(-1);
        int carry = 0;
        //计算栈顶元素相加的结果，判断是否需要进位
        while (!stack1.isEmpty() || !stack2.isEmpty() || carry != 0) {
            int x = stack1.isEmpty() ? 0 : stack1.pop();
            int y = stack2.isEmpty() ? 0 : stack2.pop();
            int sum = x + y + carry;
            //链表前插法保留计算的每一位的值
            ListNode node = new ListNode(sum % 10);
            node.next = head.next;
            head.next = node;
            carry = sum / 10;
        }
        return head.next;
    }
    //用链表的元素构造栈
    private Stack<Integer> buildStack(ListNode l) {
        Stack<Integer> stack = new Stack<>();
        while (l != null) {
            stack.push(l.val);
            l = l.next;
        }
        return stack;
    }
}
```

8. 回文链表--234--Easy

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null) return true;
        // 用快慢指针确定链表的中点
        ListNode slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        if (fast != null) slow = slow.next;  // 偶数节点，让 slow 指向下一个节点
        cut(head, slow);                     // 切成两个链表
        return isEqual(head, reverse(slow));
    }
    //切断链表
    private void cut(ListNode head, ListNode cutNode) {
        while (head.next != cutNode) {
            head = head.next;
        }
        head.next = null;
    }
    // 链表反转
    private ListNode reverse(ListNode head) {
        ListNode newHead = null;
        while (head != null) {
            ListNode nextNode = head.next;
            head.next = newHead;
            newHead = head;
            head = nextNode;
        }
        return newHead;
    }
    //判断链表是否相等
    private boolean isEqual(ListNode l1, ListNode l2) {
        while (l1 != null && l2 != null) {
            if (l1.val != l2.val) return false;
            l1 = l1.next;
            l2 = l2.next;
        }
        return true;
    }
}
```

9. 分隔链表--725--Medium

```
class Solution {
    public ListNode[] splitListToParts(ListNode root, int k) {
        int n = 0;
        ListNode cur = root;
        //计算链表长度
        while (cur != null) {
            n++;
            cur = cur.next;
        }
        int mod = n % k;
        int size = n / k;
        ListNode[] ans = new ListNode[k];//默认赋值为null
        cur = root;
        for (int i = 0; cur != null && i < k; i++) {
            ans[i] = cur;
            //计算每一部分的链表的长度，前mod份链表长度加1
            int curSize = size + (mod-- > 0 ? 1 : 0);
            //按照计算出的curSize分割链表
            for (int j = 0; j < curSize - 1; j++) {
                cur = cur.next;
            }
            //开辟下一次分割链表的链表头，并将本次链表末尾置为null
            ListNode nextHead = cur.next;
            cur.next = null;
            cur = nextHead;
        }
        return ans;
    }
}
```

10. 链表元素按奇偶聚集--328--Medium

```
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null) {
            return head;
        }
        //思路简单，将奇数节点与偶数节点分别生成一条链表，最后再相连
        ListNode odd = head, even = head.next, evenHead = even;
        while (even != null && even.next != null) {
            odd.next = odd.next.next;
            odd = odd.next;
            even.next = even.next.next;
            even = even.next;
        }
        odd.next = evenHead;
        return head;
```

```
    }
}
```

树

递归

1. 树的高度--104--Easy

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

2. 平衡树--110--Easy

```
class Solution {
    public boolean isBalanced(TreeNode root) {
        return helper(root)>=0;
    }
    //递归法解决方案
    public int helper(TreeNode root){
        if(root == null){
            return 0;
        }
        int l = helper(root.left);
        int r = helper(root.right);
        //左子树与右子树差值大于1，则标记为不平衡（-1）
        if(l==-1 || r==-1 || Math.abs(l-r)>1)
            return -1;
        //平衡则标记为上一层的高度+1
        return Math.max(l,r) +1;
    }
}
```

3. 两节点的最长路径--543--Easy

```
class Solution {
    private int max = 0;
```

```
    public int diameterOfBinaryTree(TreeNode root) {
        //遍历每一个节点,求出此节点作为根的树的深度,那么,左子树深度加右子树深度的最大值即是答案
        depth(root);
        return max;
    }

    private int depth(TreeNode root) {
        if (root == null) return 0;
        int leftDepth = depth(root.left);
        int rightDepth = depth(root.right);
        max = Math.max(max, leftDepth + rightDepth);
        return Math.max(leftDepth, rightDepth) + 1;
    }
}
```

4. 翻转树--226--Easy

```
class Solution {
    public TreeNode invertTree(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 = invertTree(root.right);
        root.right = invertTree(tempLeft);
        return root;
    }
}
```

5. 归并两棵树--617--Easy

```
class Solution {
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        //递归相加对应元素
        if (t1 == null && t2 == null) return null;
        if (t1 == null) return t2;
        if (t2 == null) return t1;
        TreeNode root = new TreeNode(t1.val + t2.val);
        root.left = mergeTrees(t1.left, t2.left);
        root.right = mergeTrees(t1.right, t2.right);
        return root;
    }
}
```

6. 判断路径和是否等于一个数--112--Easy

```
class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        //是否为空树
        if (root == null) return false;
        //是否为叶子节点，若是，判断是否路径上的和值等于sum
        if (root.left == null && root.right == null) return sum ==
root.val;
        //递归法对左子树和右子树求解
        return hasPathSum(root.left,sum - root.val) ||
hasPathSum(root.right, sum - root.val);
    }
}
```

7. 统计路径和等于一个数的路径数量--437--Medium

```
class Solution {
    //路径不一定以 root 开头，也不一定以 leaf 结尾，但是必须连续。
    public int pathSum(TreeNode root, int sum) {
        if (root == null) return 0;
        int ans = pathSumStartWithRoot(root, sum) + pathSum(root.left,
sum) + pathSum(root.right, sum);
        return ans;
    }

    private int pathSumStartWithRoot(TreeNode root, int sum) {
        if (root == null) return 0;
        int ans = 0;
        if (root.val == sum) ans++;
        ans += pathSumStartWithRoot(root.left, sum - root.val) +
pathSumStartWithRoot(root.right, sum - root.val);
        return ans;
    }
}
```

8. 子树--572--Easy

```
class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (s == null) return false;
        return isSubtreeWithRoot(s, t) || isSubtree(s.left, t) ||
isSubtree(s.right, t);
    }

    private boolean isSubtreeWithRoot(TreeNode s, TreeNode t) {
        if (t == null && s == null) return true;
        if (t == null || s == null) return false;
        if (t.val != s.val) return false;
        return isSubtreeWithRoot(s.left, t.left) &&
isSubtreeWithRoot(s.right, t.right);
    }
}
```

9. 树的对称--101--Easy

```
class Solution {
     public boolean isSymmetric(TreeNode root) {
        if (root == null) return true;
        return compare(root.left, root.right);
    }
    public boolean compare(TreeNode node1, TreeNode node2){
        //左子节点与右子节点都为空，则树对称
        if (node1 == null && node2 == null) return true;
        //左子节点或右子节点其中之一为空，则树不对称；左自节点与右自节点值不同，则树不
对称
        if (node1 == null || node2 == null || node1.val != node2.val)
return false;
        //比较左自节点的左子树与右自节点的右子树；比较左自节点的右子树与右自节点的左子
树
        return compare(node1.left, node2.right) && compare(node1.right,
node2.left);
    }
}
```

10. 最小路径--111--Easy

```
class Solution {
    public int minDepth(TreeNode root) {
        if (root == null) return 0;
        if (root.left == null && root.right == null) return 1;
        if (root.left == null && root.right != null) return
1+minDepth(root.right);
        if (root.right == null && root.left != null) return
1+minDepth(root.left);
        return 1+Math.min(minDepth(root.left), minDepth(root.right));
    }
}
```

11. 统计左叶子节点的和--404--Easy

```
class Solution {
    public int sumOfLeftLeaves(TreeNode root) {
        if (root == null) return 0;
        //判断此节点的左子节点是否是左叶子节点，如果是则将它的和累计起来
        if (root.left != null && root.left.left == null && root.left.right
== null)
            return root.left.val + sumOfLeftLeaves(root.right);
        return sumOfLeftLeaves(root.left) + sumOfLeftLeaves(root.right);
    }
}
```

12. 相同节点值的最大路径长度--687--Easy

```
class Solution {
    private int path = 0;

    public int longestUnivaluePath(TreeNode root) {
        dfs(root);
        return path;
    }

    private int dfs(TreeNode root){
        if (root == null) return 0;
        int left = dfs(root.left);
        int right = dfs(root.right);
        int leftPath = root.left != null && root.left.val == root.val ?
left + 1 : 0;
        int rightPath = root.right != null && root.right.val == root.val ?
right + 1 : 0;
        path = Math.max(path, leftPath + rightPath);
        return Math.max(leftPath, rightPath);
```

```
    }
}
```

13. 间隔遍历--337--Medium

```
//递归法
class Solution {
    public int rob(TreeNode root) {
        //节点为空，返回0
        if (root == null) return 0;
        //实际上是求不同的两种层次遍历，奇数层与偶数层
        int val1 = root.val;
        if (root.left != null)
            val1 += rob(root.left.left) + rob(root.left.right);
        if (root.right != null)
            val1 += rob(root.right.left) + rob(root.right.right);
        int val2 = rob(root.left) + rob(root.right);
        return Math.max(val1, val2);
    }
}

//深度优先法
```

14. 找出二叉树中第二小的节点--671--Easy

```
class Solution {
    public int findSecondMinimumValue(TreeNode root) {
    //1.没有必要记录最小的值，因为最小的一定是根结点。
        //2.递归找到比根结点大的值时可以立即返回，不用再遍历当前节点下面的子节点，因为子节点的值不可能比它小。
        if (root == null) return -1;
        if (root.left == null && root.right == null)
            return -1;
        int leftVal = root.left.val;
        int rightVal = root.right.val;
        if (leftVal == root.val)
            leftVal = findSecondMinimumValue(root.left);
        if (rightVal == root.val)
            rightVal = findSecondMinimumValue(root.right);
        if (leftVal != -1 && rightVal != -1)
            return Math.min(leftVal, rightVal);
        if (leftVal != -1)
            return leftVal;
        return rightVal;
    }
}
```

层次遍历

1. 一棵树每层节点的平均数--637--Easy

```
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {
        List<Double> ans = new ArrayList<>();
        if (root == null) return ans;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            //当前层的节点个数
            int cnt = queue.size();
            double sum = 0;
            for (int i = 0; i < cnt; i++) {
                //取出节点
                TreeNode node = queue.poll();
                sum += node.val;
                //存入此节点的子节点用作下一层均值计算
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            //存入当前层的平均值
            ans.add(sum / cnt);
        }
        return ans;
    }
}
```

2. 得到左下角的节点--513--Easy

```
class Solution {
    public int findBottomLeftValue(TreeNode root) {
        //层次遍历
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
            //注意顺序不能错，先取节点，再按顺序将其右子节与左子节点点入队，
            root = queue.poll();
            if (root.right != null) queue.add(root.right);
            if (root.left != null) queue.add(root.left);
        }
        return root.val;
    }
```

```
}
```

前中后序遍历

1. 非递归实现二叉树的前序遍历--144--Medium

```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> ret = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            if (node == null) continue;
            ret.add(node.val);
            stack.push(node.right);  // 先右后左，保证左子树先遍历
            stack.push(node.left);
        }
        return ret;
    }
}
```

2. 非递归实现二叉树的后序遍历--145--Hard

```
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> ret = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            if (node == null) continue;
            ret.add(node.val);
            stack.push(node.left); //先左后右，保证右子树先遍历
            stack.push(node.right);
        }
        //从根右左转成左右根
        Collections.reverse(ret);
        return ret;
    }
}
```

3. 非递归实现二叉树的中序遍历--94--Medium

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
     List<Integer> ret = new ArrayList<>();
        if (root == null) return ret;
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null || !stack.isEmpty()){
            //先将左子节点入栈
            while (cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            //存入节点的值，下一轮遍历右子节点
            TreeNode node = stack.pop();
            ret.add(node.val);
            cur = node.right;
        }
        return ret;
    }
}
```

BST

> 二叉查找树（BST）：根节点大于等于左子树所有节点，小于等于右子树所有节点。
>
> 二叉查找树中序遍历有序。

1. 修剪二叉查找树--669--Easy

```
class Solution {
    public TreeNode trimBST(TreeNode root, int L, int R) {
        if (root == null) return null;
        if (root.val > R)
            return trimBST(root.left, L, R);
        if (root.val < L)
            return trimBST(root.right, L, R);
        root.left = trimBST(root.left, L, R);
        root.right = trimBST(root.right, L, R);
        return root;
    }
}
```

2. 寻找二叉查找树的第 k 个元素--230--Easy

```
class Solution {
    //记录当前遍历的节点
```

```
    private int cnt = 0;
    private int val;

    public int kthSmallest(TreeNode root, int k) {
        inOrder(root, k);
        return val;
    }

    //中序遍历
    private void inOrder(TreeNode node, int k) {
        if (node == null) return;
        inOrder(node.left, k);
        cnt++;
        if (cnt == k) {
            val = node.val;
            return;
        }
        inOrder(node.right, k);
    }
}
```

3. 把二叉查找树每个节点的值都加上比它大的节点的值--538--Easy

```
class Solution {
    int sum = 0;
    public TreeNode convertBST(TreeNode root) {
        if (root != null) {
            //遍历右子树
            convertBST(root.right);
            //遍历顶点
            root.val = root.val + sum;
            sum = root.val;
            //遍历左子树
            convertBST(root.left);
            return root;
        }
        return null;
    }
}
```

4. 二叉查找树的最近公共祖先--235--Easy

```
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
        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节点都大于root，则说明p和q为root的右子节点
            return lowestCommonAncestor(root.left,p,q);
        }else{//其他情况均为root节点为最大父节点
            return root;
        }
    }
}
```

5. 二叉树的最近公共祖先--236--Medium

```
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;
    }
}
```

6. 从有序数组中构造二叉查找树--108--Easy

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        // 左右等分建立左右子树，中间节点作为子树根节点，递归该过程
        return nums == null ? null : buildTree(nums, 0, nums.length - 1);
    }

    private TreeNode buildTree(int[] nums, int l, int r) {
        if (l > r) {
            return null;
        }
        int m = l + (r - l) / 2;
        TreeNode root = new TreeNode(nums[m]);
        root.left = buildTree(nums, l, m - 1);
        root.right = buildTree(nums, m + 1, r);
        return root;
    }
}
```

7. 根据有序链表构造平衡的二叉查找树--109--Medium

```
class Solution {
    public TreeNode sortedListToBST(ListNode head) {
        if(head == null) return null;
        else if(head.next == null) return new TreeNode(head.val);
        ListNode pre = head;
        ListNode p = pre.next;
        ListNode q = p.next;
        //利用快慢指针法，找到链表的中点p
        while(q!=null && q.next!=null){
            pre = pre.next;
            p = pre.next;
            q = q.next.next;
        }
        //将中点左边的链表分开
        pre.next = null;
        //递归求两边的子树
        TreeNode root = new TreeNode(p.val);
        root.left = sortedListToBST(head);
        root.right = sortedListToBST(p.next);
        return root;
    }
}·
```

8. 在二叉查找树中寻找两个节点，使它们的和为一个给定值--653--Easy

```
class Solution {
    //使用中序遍历得到有序数组之后，再利用双指针对数组进行查找。
    public boolean findTarget(TreeNode root, int k) {
        List<Integer> nums = new ArrayList<>();
        inOrder(root, nums);
        int i = 0, j = nums.size() - 1;
        while (i < j) {
            int sum = nums.get(i) + nums.get(j);
            if (sum == k) return true;
            if (sum < k) i++;
            else j--;
        }
        return false;
    }

    private void inOrder(TreeNode root, List<Integer> nums) {
        if (root == null) return;
        inOrder(root.left, nums);
        nums.add(root.val);
        inOrder(root.right, nums);
    }
}
```

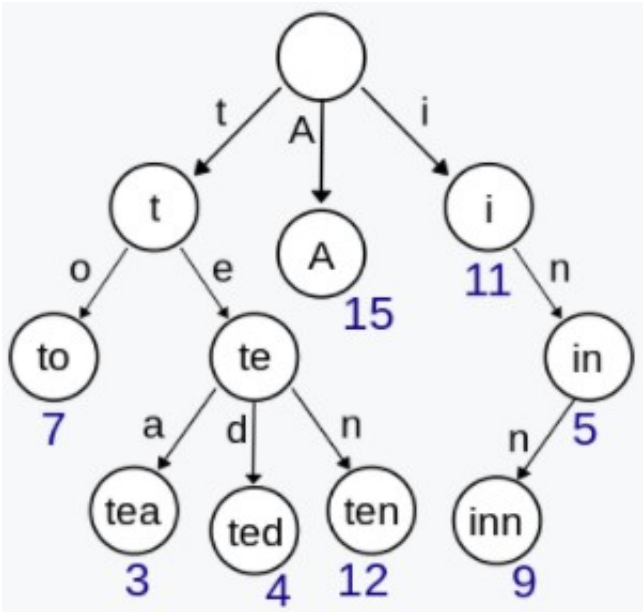
9. 在二叉查找树中查找两个节点之差的最小绝对值--530--Easy

```
class Solution {
    private int minDiff = Integer.MAX_VALUE;
    private TreeNode preNode = null;

    public int getMinimumDifference(TreeNode root) {
        inOrder(root);
        return minDiff;
    }
    //中序遍历得到升序数列，每次计算两数之间最小值，取最小
    private void inOrder(TreeNode node) {
        if (node == null) return;
        inOrder(node.left);
        if (preNode != null)
            minDiff = Math.min(minDiff, node.val - preNode.val);
        preNode = node;
        inOrder(node.right);
    }
}
```

10. 寻找二叉查找树中出现次数最多的值--501--Easy

```
class Solution {
    private int curCnt = 1;
private int maxCnt = 1;
    private TreeNode preNode = null;

    public int[] findMode(TreeNode root) {
        List<Integer> maxCntNums = new ArrayList<>();
        //引用maxCntNums
        inOrder(root, maxCntNums);
        int[] ret = new int[maxCntNums.size()];
        //转成答案形式
        for (int i = 0; i < maxCntNums.size(); i++)
            ret[i] = maxCntNums.get(i);
        return ret;
    }
    //中序遍历
    private void inOrder(TreeNode node, List<Integer> nums) {
        if (node == null) return;
        inOrder(node.left, nums);
        if (preNode != null) {
            if (preNode.val == node.val) // 当前值与上一个结点值相同，当前值的出现次数增加。
                curCnt++;
            else // 当前值与上一个结点值不同，重置计数
                curCnt = 1;
        }
        if (curCnt > maxCnt) {// 出现次数更多，清空之前的出现少的数，更新最大出现次数
            maxCnt = curCnt;
            nums.clear();//清空
            nums.add(node.val);
        } else if (curCnt == maxCnt) {// 不止一个众数
            nums.add(node.val);
        }
        preNode = node;
        inOrder(node.right, nums);
    }
}
```

Trie



Trie，又称前缀树或字典树，用于判断字符串是否存在或者是否具有某种字符串前缀。

1. 实现一个 Trie--208--Medium

```
class Trie {

    private class Node {
        Node[] childs = new Node[26];
        boolean isLeaf;
    }

    private Node root = new Node();

    public Trie() {
    }

    public void insert(String word) {
        insert(word, root);
    }

    private void insert(String word, Node node) {
        if (node == null) return;
        if (word.length() == 0) {
            node.isLeaf = true;
            return;
        }
        int index = indexForChar(word.charAt(0));
        if (node.childs[index] == null) {
            node.childs[index] = new Node();
        }
        insert(word.substring(1), node.childs[index]);
    }
```

```
    public boolean search(String word) {
        return search(word, root);
    }

    private boolean search(String word, Node node) {
        if (node == null) return false;
        if (word.length() == 0) return node.isLeaf;
        int index = indexForChar(word.charAt(0));
        return search(word.substring(1), node.childs[index]);
    }

    public boolean startsWith(String prefix) {
        return startWith(prefix, root);
    }

    private boolean startWith(String prefix, Node node) {
        if (node == null) return false;
        if (prefix.length() == 0) return true;
        int index = indexForChar(prefix.charAt(0));
        return startWith(prefix.substring(1), node.childs[index]);
    }

    private int indexForChar(char c) {
        return c - 'a';
    }
}
```

2. 实现一个 Trie，用来求前缀和--677--Medium

```
class MapSum {

 private class Node {
        Node[] child = new Node[26];
        int value;
    }

    private Node root = new Node();

    public MapSum() {

    }

    public void insert(String key, int val) {
        insert(key, root, val);
    }

    private void insert(String key, Node node, int val) {
```

```
        if (node == null) return;
        if (key.length() == 0) {
            node.value = val;
            return;
        }
        int index = indexForChar(key.charAt(0));
        if (node.child[index] == null) {
            node.child[index] = new Node();
        }
        insert(key.substring(1), node.child[index], val);
    }

    public int sum(String prefix) {
        return sum(prefix, root);
    }

    private int sum(String prefix, Node node) {
        if (node == null) return 0;
        if (prefix.length() != 0) {
            int index = indexForChar(prefix.charAt(0));
            return sum(prefix.substring(1), node.child[index]);
        }
        int sum = node.value;
        for (Node child : node.child) {
            sum += sum(prefix, child);
        }
        return sum;
    }

    private int indexForChar(char c) {
        return c - 'a';
    }
}
```

栈和队列

1. 用栈实现队列--232--Easy

```
class MyQueue {

    private Stack<Integer> a;// 输入栈
    private Stack<Integer> b;// 输出栈

    public MyQueue() {
        a = new Stack<>();
        b = new Stack<>();
```

```
    }

    public void push(int x) {
        a.push(x);
    }

    public int pop() {
        // 如果b栈为空，则将a栈全部弹出并压入b栈中，然后b.pop()
        if(b.isEmpty()){
            while(!a.isEmpty()){
                b.push(a.pop());
            }
        }
        return b.pop();
    }

    public int peek() {
        if(b.isEmpty()){
            while(!a.isEmpty()){
                b.push(a.pop());
            }
        }
        return b.peek();
    }

    public boolean empty() {
        return a.isEmpty() && b.isEmpty();
    }
}
```

2. 用队列实现栈--225--Easy

```
class MyStack {

    private Queue<Integer> a;//输入队列
    private Queue<Integer> b;//输出队列

    public MyStack() {
        a = new LinkedList<>();
        b = new LinkedList<>();
    }

    public void push(int x) {
        a.offer(x);
        // 将b队列中元素全部转给a队列
        while(!b.isEmpty())
            a.offer(b.poll());
```

```
        // 交换a和b,使得a队列没有在push()的时候始终为空队列
        Queue temp = a;
        a = b;
        b = temp;
    }

    public int pop() {
        return b.poll();
    }

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

    public boolean empty() {
        return b.isEmpty();
    }
}
```

3. 最小值栈--155--Easy

```
// 链表解决
class MinStack {

    private Node head;

    public void push(int x) {
        if(head == null)
            head = new Node(x, x);
        else
            head = new Node(x, Math.min(x, head.min), head);
    }

    public void pop() {
        head = head.next;
    }

    public int top() {
        return head.val;
    }

    public int getMin() {
        return head.min;
    }
    //定义私有节点类
    private class Node {
        int val;
```

```
        int min;
        Node next;

        private Node(int val, int min) {
            this(val, min, null);
        }

        private Node(int val, int min, Node next) {
            this.val = val;
            this.min = min;
            this.next = next;
        }
    }
}

//用栈解决
class MinStack {

    private Stack<Integer> dataStack;
    private Stack<Integer> minStack;
    private int min;

    public MinStack() {
        dataStack = new Stack<>();
        minStack = new Stack<>();
        min = Integer.MAX_VALUE;
    }


    public void push(int x) {//最主要注意push进来的时候更新min的值
        dataStack.add(x);
        min = Math.min(min, x);
        minStack.add(min);
    }

    public void pop() {
        dataStack.pop();
        minStack.pop();
        min = minStack.isEmpty() ? Integer.MAX_VALUE : minStack.peek();
    }

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

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

4. 用栈实现括号匹配--20--Easy

```
class Solution {
    public boolean isValid(String s) {
        if (s.isEmpty()){
            return true;
        }
        Stack<Character> stack = new Stack<Character>();
        for (Character c : s.toCharArray()) {
            //当字符分别为({[，向栈中存入)}],如果字符串括号字符匹配，则栈顶字符会与其一致
            if(c=='(')
                stack.push(')');
            else if(c=='{')
                stack.push('}');
            else if(c=='[')
                stack.push(']');
            else if(stack.empty()||c!=stack.pop())
                return false;
        }
        return stack.empty();
    }
}
```

5. 数组中元素与下一个比它大的元素之间的距离--739--Medium

```
//暴力，用时击败19%
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int len = T.length;
        int[] ans = new int[len];
        for (int i = 0; i < len; i++) {
            for (int j = i; j < len; j++) {
                if (T[j] > T[i]) {
                    ans[i] = j - i;
                    break;
                }
            }
        }
        return ans;
    }
}
//结合栈，用时击败69%,Java中new一个数组，其中元素的默认值为0
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int n = T.length;
```

```
        int[] dist = new int[n];
        Stack<Integer> indexs = new Stack<>();
        for (int curIndex = 0; curIndex < n; curIndex++) {
            //下标栈不为空，且当前温度大于上一次下标的温度
            while (!indexs.isEmpty() && T[curIndex] > T[indexs.peek()]) {
                int preIndex = indexs.pop();
                dist[preIndex] = curIndex - preIndex;
            }
            indexs.add(curIndex);
        }
        return dist;
    }
}
```

6. 循环数组中比当前元素大的下一个元素--503--Medium

```
class Solution {
    //思路:
    //1.将数组中所有元素全部置为-1
    //2.遍历两次，相当于循环遍历
    //3.第一遍遍历，入栈索引i
    //4.只要后面元素比栈顶索引对应的元素大，索引出栈，ans[pre.pop()]的数值
    //5.最后栈里面剩余的索引对应的数组值，都为默认的-1（因为后面未找到比它大的值）
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] ans = new int[n];
        Arrays.fill(ans, -1);
        Stack<Integer> pre = new Stack<>();
        for (int i = 0; i < n * 2; i++) {
            int num = nums[i % n];
            while (!pre.isEmpty() && nums[pre.peek()] < num) {
                ans[pre.pop()] = num;
            }
            if (i < n){
                pre.push(i);
            }
        }
        return ans;
    }
}
```

哈希表

1. 数组中两个数的和为给定值--1--Easy

```java
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]，
        //如果存在说明 target - nums[i] 所在的索引和 i 就是要找的两个数。
        HashMap<Integer, Integer> indexForNum = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (indexForNum.containsKey(target - nums[i])) {
                return new int[]{indexForNum.get(target - nums[i]), i};
            } else {
                indexForNum.put(nums[i], i);
            }
        }
        return null;
    }
}
```

2. 判断数组是否含有重复元素--217--Easy

```java
class Solution {
    public boolean containsDuplicate(int[] nums) {
        //HashSet自带去重，所以HashSet的size小于数组的length的话就存在重复元素
        Set<Integer> res = new HashSet<>();
        for(int i:nums)
            res.add(i);
        return res.size()<nums.length;
    }
}
```

3. 最长和谐序列--594--Easy

```java
class Solution {
    public int findLHS(int[] nums) {
        //和谐序列中最大数和最小数之差正好为 1，应该注意的是序列的元素不一定是数组的连续元素。
        Map<Integer, Integer> countForNum = new HashMap<>();
        for (int num : nums) {
            //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
            countForNum.put(num, countForNum.getOrDefault(num, 0) + 1);
        }
        int longest = 0;
        for (int num : countForNum.keySet()) {
```

```
            //当前元素是num，判断Map中是否存在num+1的元素，longset取当前longest与这两个元素和的最大值
            if (countForNum.containsKey(num + 1)) {
                longest = Math.max(longest, countForNum.get(num + 1) + countForNum.get(num));
            }
        }
        return longest;
    }
}
```

4. 最长连续序列--128--Hard

```
class Solution {
    public int longestConsecutive(int[] nums) {
        Map<Integer, Integer> countForNum = new HashMap<>();
        for (int num : nums) {
            countForNum.put(num, 1);
        }
        for (int num : nums) {
            forward(countForNum, num);
        }
        return maxCount(countForNum);
    }

    private int forward(Map<Integer, Integer> countForNum, int num) {
        if (!countForNum.containsKey(num)) {
            return 0;
        }
        int cnt = countForNum.get(num);
        if (cnt > 1) {
            return cnt;
        }
        cnt = forward(countForNum, num + 1) + 1;
        countForNum.put(num, cnt);
        return cnt;
    }

    private int maxCount(Map<Integer, Integer> countForNum) {
        int max = 0;
        for (int num : countForNum.keySet()) {
            max = Math.max(max, countForNum.get(num));
        }
        return max;
    }
}
```

```java
class Solution {
    public int longestConsecutive(int[] nums) {
        //思路是将所有元素用HashMap存储
        //以每一个元素为原点，向左右两边扩张，保存当前元素已经存在的最大连续子序列
        //相同元素的最大连续子序列是相等的
        int n = nums.length;
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        int ans = 0;
        for (int num : nums) {
            if (!map.containsKey(num)) {
                //判断是否有num-1这个元素，如果有令left值等于此元素的当前最大连续子序列长度
                int left = map.get(num - 1) == null ? 0 : map.get(num - 1);
                //判断是否有num+1这个元素，如果有令right值等于此元素的当前最大连续子序列长度
                int right = map.get(num + 1) == null ? 0 : map.get(num + 1);
                int cur = 1 + left + right;
                //判断当前元素的最大扩张是否最大
                if (cur > ans) {
                    ans = cur;
                }
                map.put(num, cur);
                //更新与之相关的左右元素的最大子序列
                map.put(num - left, cur);
                map.put(num + right, cur);
         }
        }
        return ans;
    }
}
```

字符串

1. 两个字符串包含的字符是否完全相同--242--Easy

```
class Solution {
    public boolean isAnagram(String s, String t) {
        char[] sChars = s.toCharArray();
        char[] tChars = t.toCharArray();
        Arrays.sort(sChars);
        Arrays.sort(tChars);
        return String.valueOf(sChars).equals(String.valueOf(tChars));
    }
}
```

2. 计算一组字符集合可以组成的回文字符串的最大长度--409--Easy

```
class Solution {
    public boolean isIsomorphic(String s, String t) {
        int[] cnts = new int[256];
        for (char c : s.toCharArray()) {
            cnts[c]++;
        }
        int palindrome = 0;
        for (int cnt : cnts) {
            palindrome += (cnt / 2) * 2;
        }
        if (palindrome < s.length()) {
            palindrome++;   // 这个条件下 s 中一定有单个未使用的字符存在，可以把这个字符放到回文的最中间
        }
        return palindrome;
    }
}
```

3. 字符串同构--205--Easy

```
class Solution {
    //记录一个字符上次出现的位置，如果两个字符串中的字符上次出现的位置一样，那么就属于同构。
    public boolean isIsomorphic(String s, String t) {
        int[] preIndexOfS = new int[256];
        int[] preIndexOfT = new int[256];
        for (int i = 0; i < s.length(); i++) {
            char sc = s.charAt(i), tc = t.charAt(i);
            if (preIndexOfS[sc] != preIndexOfT[tc]) {
                return false;
            }
            preIndexOfS[sc] = i + 1;
            preIndexOfT[tc] = i + 1;
```

```
        }
        return true;
    }
}
```

4. 回文子字符串个数--647--Medium

```
class Solution {
    private int ans = 0;
    //从字符串的某一位开始，尝试着去扩展子字符串
    public int countSubstrings(String s) {
        for (int i = 0; i < s.length(); i++) {
            extendSubstrings(s, i, i);     // 从当前字符开始奇数长度的回文数
            extendSubstrings(s, i, i + 1); // 从当前字符与当前下一字符开始偶数长
度的回文数
        }
        return ans;
    }

    private void extendSubstrings(String s, int start, int end) {
        while (start >= 0 && end < s.length() && s.charAt(start) ==
s.charAt(end)) {
            start--;
            end++;
            ans++;
        }
    }
}
```

5. 判断一个整数是否是回文数--9--Easy

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

6. 统计二进制字符串中连续 1 和连续 0 数量相同的子字符串个数--696--Easy

```
class Solution {
    public int countBinarySubstrings(String s) {
     int  n = s.length();
        int pre = 0;
        int curr = 1;
        int ans = 0;
        for (int i = 0; i < n - 1; i++) {
            //记录当前相同字符个数
            if (s.charAt(i) == s.charAt(i+1)) {
                curr++;
            } else {
                pre = curr;
                curr = 1;
            }
            //当前相同字符个数与前一相同字符个数比较，若小于，则累加答案个数
            if (pre >= curr)
                ans++;
        }
        return ans;
    }
}
```

数据与矩阵

1. 把数组中的 0 移到末尾--283--Easy

```
class Solution {
    public void moveZeroes(int[] nums) {
        int idx = 0;
        for (int num : nums) {
            if (num != 0) {
                nums[idx++] = num;
            }
        }
        while (idx < nums.length) {
            nums[idx++] = 0;
        }
    }
}
```

2. 改变矩阵维度--566--Easy

```
class Solution {
    public int[][] matrixReshape(int[][] nums, int r, int c) {
        int m = nums.length, n = nums[0].length;
        if (m * n != r * c) {
            return nums;
        }
        int index = 0;
        int[][] ans = new int[r][c];
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                //nums数组中第index个元素为nums[index / n][index % n]
                ans[i][j] = nums[index / n][index % n];
                index++;
            }
        }
        return ans;
    }
}
```

3. 找出数组中最长的连续 1--485--Easy

```
class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        //暴力
        int max = 0, cur = 0;
        for (int x : nums) {
            cur = x == 0 ? 0 : cur + 1;
            max = Math.max(max, cur);
        }
        return max;

    }
}
```

4. 有序矩阵查找--240--Medium

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return false;
        int m = matrix.length, n = matrix[0].length;
        int row = 0, col = n-1;
        //col--比col++更合适处理此情况
```

```
        while (row < m && col >= 0){
            if (target == matrix[row][col]) return true;
            else if (target < matrix[row][col]) col--;
            else
                row++;
        }
        return false;
    }
}
```

5. 有序矩阵的 Kth Element--378--Medium

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        //直接排序法
        int rows = matrix.length, columns = matrix[0].length;
        int[] sorted = new int[rows * columns];
        int index = 0;
        for (int[] row : matrix) {
            for (int num : row) {
                sorted[index++] = num;
            }
        }
        Arrays.sort(sorted);
        return sorted[k - 1];
    }
}



class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        //二分查找法
        int m = matrix.length, n = matrix[0].length;
        int low = matrix[0][0], high = matrix[m - 1][n - 1];
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int count = 0;
            //计算数组中小于当前mid的元素个数
            for (int i = 0; i < m; i++)
                for (int j = 0; j < n && matrix[i][j] <= mid; j++)
                    count++;
            //数组中小于mid的个数小于k，调整下界，令low等于mid+1
            //反之，调整上界，令high等于mid-1
            if (count < k)
                low = mid + 1;
            else
```

```
                high = mid - 1;
            }
            //经计算的下界元素，即数组中第k小的元素
            return low;
        }
    }
```

6. 一个数组元素在 [1, n] 之间，其中一个数被替换为另一个数，找出重复的数和丢失的数--645--Easy

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        // 用一个布尔数组来记录元素是否出现，布尔数组默认值为false
        boolean[] showed = new boolean[nums.length + 1];
        int[] ans = new int[2];
        for (int num : nums) {
            // 出现了重复的数字 置为true
            if (showed[num])
                ans[0] = num;
            else
                showed[num] = true;
        }
        for (int i = 1; i <= nums.length; i++) {
            // 找到没有出现的那一位
            if (!showed[i])
                ans[1] = i;
        }
        return ans;
    }
}
```

7. 找出数组中重复的数，数组值在 [1, n] 之间--287--Medium

```
class Solution {
    public int findDuplicate(int[] nums) {
        //双指针解法，类似于有环链表中找出环的入口：
        int slow = nums[0], fast = nums[nums[0]];
        while (slow != fast) {
            slow = nums[slow];
            fast = nums[nums[fast]];
        }
        fast = 0;
        while (slow != fast) {
            slow = nums[slow];
            fast = nums[fast];
        }
        return slow;
```

```
    }
}
//上述解法过于炫技，我选择hashset
class Solution {
    public int findDuplicate(int[] nums) {
        Set<Integer> set = new HashSet<>();
        int ans = 0;
        for (int i = 0; i < nums.length; i++) {
            if (!set.contains(nums[i])) {
                set.add(nums[i]);
            }else {
                ans = nums[i];
            }
        }
        return ans;
    }
}
```

8. 数组相邻差值的个数--667--Medium

```
class Solution {
    public int[] constructArray(int n, int k) {
        //题目描述：数组元素为 1\~n 的整数，要求构建数组，使得相邻元素的差值不相同的个数为 k。
        //让前 k+1 个元素构建出 k 个不相同的差值，序列为：1、k+1、 2、k、 3、k-1 ... k/2、k/2+1.
        int[] ret = new int[n];
        ret[0] = 1;
        for (int i = 1, interval = k; i <= k; i++, interval--) {
            ret[i] = i % 2 == 1 ? ret[i - 1] + interval : ret[i - 1] - interval;
        }
        for (int i = k + 1; i < n; i++) {
            ret[i] = i + 1;
        }
        return ret;
    }
}
```

9. 数组的度--697--Easy

```
class Solution {
    public int findShortestSubArray(int[] nums) {
        //题目意思是在不改变原数组顺序的情况下求出一个度与原数组一样的子数组
        //用来记录每个元素出现的次数
        Map<Integer, Integer> numsCnt = new HashMap<>();
```

```
        //记录每个元素的最后一次出现的下标
        Map<Integer, Integer> numsLastIndex = new HashMap<>();
        //记录每个元素的第一次出现的下标
        Map<Integer, Integer> numsFirstIndex = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int num = nums[i];
            numsCnt.put(num, numsCnt.getOrDefault(num, 0) + 1);
            numsLastIndex.put(num, i);
            if (!numsFirstIndex.containsKey(num)) {
                numsFirstIndex.put(num, i);
            }
        }
        //求出数组的度
        int maxCnt = 0;
        for (int num : nums) {
            maxCnt = Math.max(maxCnt, numsCnt.get(num));
        }
        int ans = nums.length;
        for (int i = 0; i < nums.length; i++) {
            int num = nums[i];
            int cnt = numsCnt.get(num);
            if (cnt != maxCnt) continue;
            ans = Math.min(ans, numsLastIndex.get(num) -
numsFirstIndex.get(num) + 1);
        }
        return ans;
    }
}
```

10. 对角元素相等的矩阵--766--Easy

```
//方法一：判断上一行去掉最后一个元素 和 下一行去掉第一个元素是否相等

//方法二：暴力
class Solution {
    public boolean isToeplitzMatrix(int[][] matrix) {
        for (int i = 1; i < matrix.length; i++) {
            for (int j = 1; j < matrix[0].length; j++) {
                if (matrix[i][j] != matrix[i - 1][j - 1])
                    return false;
            }
        }
        return true;
    }
}
```

11. 嵌套数组--565--Medium

```
class Solution {
    public int arrayNesting(int[] nums) {
        int max = 0;
        for (int i = 0; i < nums.length; i++) {
            int cnt = 0;
            for (int j = i; nums[j] != -1; ) {
                cnt++;
                int t = nums[j];
                nums[j] = -1; // 标记该位置已经被访问
                j = t;
            }
            max = Math.max(max, cnt);
        }
        return max;
    }
}
```

12. 分隔数组--769--Medium

```
class Solution {
    public int maxChunksToSorted(int[] arr) {
    //思路
        //首先找到从左块开始最小块的大小。
        //如果前 k 个元素为 [0, 1, ..., k-1]，可以直接把他们分为一个块。
        //当我们需要检查 [0, 1, ..., n-1] 中前 k+1 个元素是不是 [0, 1, ..., k]
的时候，只需要检查其中最大的数是不是 k 就可以了。
        int ans = 0, max = 0;
        for (int i = 0; i < arr.length; ++i) {
            max = Math.max(max, arr[i]);
            if (max == i) ans++;
        }
        return ans;
    }
}
```

图 - 未写

二分图

> 如果可以用两种颜色对图中的节点进行着色，并且保证相邻的节点颜色不同，那么这个图就是二分图。

1. 判断是否为二分图--785--Medium

拓扑排序

1. 课程安排的合法性--207--Medium

2. 课程安排的顺序--210--Medium

并查集

1. 冗余连接--684--Medium

位运算

http://c.biancheng.net/view/784.html

0. 原理

   基本原理

运算符	含义	实例	结果
&	按位进行与运算（AND）	4 & 5	4
|	按位进行或运算（OR）	4 | 5	5
^	按位进行异或运算（XOR）	4 ^ 5	1
~	按位进行取反运算（NOT）	~ 4	-5

运算符	含义	实例	结果
»	右移位运算符	8»1	4
«	左移位运算符	9«2	36

0s 表示一串 0，1s 表示一串 1。

```
x ^ 0s = x      x & 0s = 0      x | 0s = x
x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
x ^ x = 0       x & x = x       x | x = x
```

利用 x ^ 1s = ~x 的特点，可以将一个数的位级表示翻转；利用 x ^ x = 0 的特点，可以将三个数中重复的两个数去除，只留下另一个数。

```
1^1^2 = 2
```

利用 x & 0s = 0 和 x & 1s = x 的特点，可以实现掩码操作。一个数 num 与 mask：00111100 进行位与操作，只保留 num 中与 mask 的 1 部分相对应的位。

```
01011011 &
00111100
--------
00011000
```

利用 x | 0s = x 和 x | 1s = 1s 的特点，可以实现设值操作。一个数 num 与 mask：00111100 进行位或操作，将 num 中与 mask 的 1 部分相对应的位都设置为 1。

```
01011011 |
00111100
--------
01111111
```

位与运算技巧

n&(n-1) 去除 n 的位级表示中最低的那一位 1。例如对于二进制表示 01011011，减去 1 得到 01011010，这两个数相与得到 01011010。

```
01011011 &
01011010
--------
01011010
```

n&(-n) 得到 n 的位级表示中最低的那一位 1。-n 得到 n 的反码加 1，也就是 -n=~n+1。例如对于二进制表示 10110100，-n 得到 01001100，相与得到 00000100。

```
10110100 &
01001100
--------
00000100
```

n-(n&(-n)) 则可以去除 n 的位级表示中最低的那一位 1，和 n&(n-1) 效果一样。

移位运算

>> n 为算术右移，相当于除以 2n，例如 -7 >> 2 = -2。

```
11111111111111111111111111111001  >> 2
--------
11111111111111111111111111111110
```

>>> n 为无符号右移，左边会补上 0。例如 -7 >>> 2 = 1073741822。

```
11111111111111111111111111111001  >>> 2
--------
00111111111111111111111111111111
```

<< n 为算术左移，相当于乘以 2n。-7 << 2 = -28。

```
11111111111111111111111111111001  << 2
--------
11111111111111111111111111100100
```

mask 计算

要获取 111111111，将 0 取反即可，~0。

要得到只有第 i 位为 1 的 mask，将 1 向左移动 i-1 位即可，1<<(i-1) 。例如 1<<4 得到只有第 5 位为 1 的 mask： 00010000。

要得到 1 到 i 位为 1 的 mask，(1<<i)-1 即可，例如将 (1<<4)-1 = 00010000-1 = 00001111。

要得到 1 到 i 位为 0 的 mask，只需将 1 到 i 位为 1 的 mask 取反，即 ~((1<<i)-1)。

Java 中的位操作

```
static int Integer.bitCount();            // 统计 1 的数量
static int Integer.highestOneBit();       // 获得最高位
static String toBinaryString(int i);      // 转换为二进制表示的字符串
```

1. 统计两个数的二进制表示有多少位不同--461--Easy

```
class Solution {
    public int hammingDistance(int x, int y) {
        //位的异或运算，两数相等时，x^y = 0
        int z = x ^ y;
        int cnt = 0;
        while(z != 0) {
            //位与操作
            cnt += z & 1;
            z = z>>1;
        }
        return cnt;
    }
}
```

2. 数组中唯一一个不重复的元素--136--Easy

```
class Solution {
    public int singleNumber(int[] nums) {
        int ans = 0;
        //对全部元素进行异或计算，异或满足交换律，得出的答案为单个元素
        for(int num: nums) {
            ans ^= num;
        }
        return ans;
    }
}
```

3. 找出数组中缺失的那个数--268--Easy

```
class Solution {
    public int missingNumber(int[] nums) {
        int ans = 0;
        //对所有元素进行异或操作，相同元素的异或答案为0，所以结果最后为缺少的数
        for (int i = 0; i < nums.length; i++) {
            ans = ans ^ i ^ nums[i];
        }
        return ans ^ nums.length;
    }
}
```

4. 数组中不重复的两个元素--260--Easy

```
class Solution {
```

```
    public int[] singleNumber(int[] nums) {
        //两个不相等的元素在位级表示上必定会有一位存在不同。
        //将数组的所有元素异或得到的结果为不存在重复的两个元素异或的结果。
        //diff &= -diff 得到出 diff 最右侧不为 0 的位，
        //也就是不存在重复的两个元素在位级表示上最右侧不同的那一位，利用这一位就可以将两个元素区分开来。
        int diff = 0;
        for (int num : nums) diff ^= num;
        diff &= -diff;  // 得到最右一位
        int[] ret = new int[2];
        for (int num : nums) {
            if ((num & diff) == 0) ret[0] ^= num;
            else ret[1] ^= num;
        }
        return ret;
    }
}
```

5. 翻转一个数的比特位--190--Easy

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

6. 判断一个数是不是 2 的 n 次方--231--Easy

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        //递归求解
        if(n==1) return true;
        if(n==0 || n%2!=0) return false;
        return isPowerOfTwo(n/2);
    }

    public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;
    }
}
```

7. 判断一个数是不是 4 的 n 次方--342--Easy

```
class Solution {
    public boolean isPowerOfFour(int num) {
        //这种数在二进制表示中有且只有一个奇数位为 1，例如 16（10000）。
        return num > 0 && (num & (num - 1)) == 0 && (num &
0b01010101010101010101010101010101) != 0;
    }
}
```

8. 判断一个数的位级表示是否不会出现连续的 0 和 1--693--Easy

```
class Solution {
    public boolean hasAlternatingBits(int n) {
        //对于 1010 这种位级表示的数，把它向右移动 1 位得到 101，这两个数每个位都不
同，因此异或得到的结果为 1111。
        int a = (n ^ (n >> 1));
        return (a & (a + 1)) == 0;
    }
}
```

9. 求一个数的补码--476--Easy

```
class Solution {
    //对于 00000101，要求补码可以将它与 00000111 进行异或操作。那么问题就转换为求掩
码 00000111
    public int findComplement(int num) {
        if (num == 0) return 1;
        int mask = 1 << 30;
        while ((num & mask) == 0) mask >>= 1;
        mask = (mask << 1) - 1;
        return num ^ mask;
    }

    //可以利用 Java 的 Integer.highestOneBit() 方法来获得含有首 1 的数。
    public int findComplement(int num) {
        if (num == 0) return 1;
        int mask = Integer.highestOneBit(num);
        mask = (mask << 1) - 1;
        return num ^ mask;
    }
}
```

10. 实现整数的加法--371--Easy

```java
class Solution {
    public int getSum(int a, int b) {
        if (b == 0)
            return a;
        return getSum((a ^ b), (a & b) << 1);
    }
}
```

11. 字符串数组最大乘积--318--Medium

```java
 class Solution {
     public int maxProduct(String[] words) {
         /**
         全是小写字母，可以用一个32为整数表示一个word中出现的字母，
         hash[i]存放第i个单词出现过的字母，a对应32位整数的最后一位，
         b对应整数的倒数第二位，依次类推. 时间复杂度O(N^2)
         判断两两单词按位与的结果，如果结果为0且长度积大于最大积则更新
         **/
         int n = words.length;
         int[] hash = new int[n];
         int max = 0;
         for(int i = 0; i < n; ++i) {
             for(char c : words[i].toCharArray())
                 hash[i] |= 1 << (c-'a');
         }

         for(int i = 0; i < n-1; ++i) {
             for(int j = i+1; j < n; ++j) {
                 if((hash[i] & hash[j]) == 0)
                     max = Math.max(words[i].length() * words[j].length(),
max);
             }
         }
         return max;
     }
 }
```

12. 统计从 0 ~ n 每个数的二进制表示中 1 的个数--338--Medium

```
class Solution {
    public int[] countBits(int num) {
    //对于数字 6(110)，它可以看成是 4(100) 再加一个 2(10)，
        //因此 dp[i] = dp[i&(i-1)] + 1;
        int[] ret = new int[num + 1];
        for(int i = 1; i <= num; i++){
            ret[i] = ret[i&(i-1)] + 1;
        }
        return ret;
    }
}
```

算法思想

双指针

1. 有序数组的 Two Sum--167--Easy

```
//暴力
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int[] ans = new int[2];
        int flag = 0;
        for (int i = 0; i < numbers.length; i++) {
            for (int j = i+1; j < numbers.length; j++) {
                if (numbers[i]+numbers[j] == target && i!=j) {
                    ans[0]=i+1;
                    ans[1]=j+1;
                    flag = 1;
                    break;
                }
            }
            if (flag==1){
                break;
            }
        }
        return ans;
    }
}
//双指针
```

2. 两数平方和--633--Easy

```
class Solution {
    public boolean judgeSquareSum(int c) {
        if (c < 0) return false;
        int i = 0, j = (int) Math.sqrt(c);
        while (i <= j) {
            int powSum = i * i + j * j;
            if (powSum == c) {
                return true;
            } else if (powSum > c) {
                j--;
            } else {
                i++;
            }
        }
        return false;
    }
}
```

3. 反转字符串中的元音字符--345--Easy

```
class Solution {
    public String reverseVowels(String s) {
        char[] letters = s.toCharArray();
        //使用双指针
        List<Character> letterList = new ArrayList<>();
        char[] letter=new char[]
{'a','e','i','o','u','A','E','I','O','U'};
        int len = s.length();
        int left = 0, right = len-1;
        for(int i=0;i<10;i++)
            letterList.add(letter[i]);
        char temp;
        while (left < right){
            if (letterList.contains(letters[left]) &&
letterList.contains(letters[right])) {
                temp = letters[left];
                letters[left] = letters[right];
                letters[right] = temp;
                left++;
                right--;
            }
            else if (!letterList.contains(letters[left]) &&
letterList.contains(letters[right])) {
                left++;
            }
```

```
            else if (letterList.contains(letters[left]) &&
!letterList.contains(letters[right])) {
                right--;
            }
            else {
                left++;
                right--;
            }
        }
        s = String.valueOf(letters);
        return s;
    }
}
```

4. 回文字符串--680--Easy

```
class Solution {
    public boolean validPalindrome(String s) {
        for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
            //如果不想等，则分别删除两个字符查看剩余未检测的字符串是否回文
            if (s.charAt(i) != s.charAt(j)) {
                return isPalindrome(s, i, j - 1) || isPalindrome(s, i + 1,
j);
            }
        }
        //s是回文直接返回true
        return true;
    }
    //双指针判断是否回文
    private boolean isPalindrome(String s, int i, int j) {
        while (i < j) {
            if (s.charAt(i++) != s.charAt(j--)) {
                return false;
            }
        }
        return true;
    }
}
```

5. 归并两个有序数组--88--Easy

```
class Solution {
     public void merge(int[] nums1, int m, int[] nums2, int n) {
        //先合并再排序
        int len1 =nums1.length;
        int k = 0;
        for (int i = m; i < len1; i++) {
           nums1[i] = nums2[k++];
        }
        Arrays.sort(nums1);
    }
}
```

6. 判断链表是否存在环--141--Easy

```
//hashset解法
class Solution {
    public boolean hasCycle(ListNode head) {
        HashSet<ListNode> nodesSeen = new HashSet<>();
        while (head != null) {
            //遍历每个节点，若此节点之前遍历过，则是环形链表
            if (nodesSeen.contains(head)) {
                return true;
            } else {
                nodesSeen.add(head);
            }
            head = head.next;
        }
        return false;
    }
}
//使用双指针，一个指针每次移动一个节点，一个指针每次移动两个节点，如果存在环，那么这两个指针一定会相遇。
class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null) {
        return false;
    }
    ListNode l1 = head, l2 = head.next;
    while (l1 != null && l2 != null && l2.next != null) {
        if (l1 == l2) {
            return true;
        }
        l1 = l1.next;
        l2 = l2.next.next;
    }
    return false;
    }
```

```
}
```

7. 最长子序列--524--Medium

```
class Solution {
    public String findLongestWord(String s, List<String> d) {
     //初始化最长子串为空
        String longestWord = "";
        for (String target : d) {
            int l1 = longestWord.length(), l2 = target.length();
            //l1大于l2则不比较
            //或长度相同情况下字典顺序更大，注意：字典顺序是按照字母ASCII顺序大小比较
            //String元素，compareTo：
            //如果参数字符串等于此字符串，则返回值 0；
            //如果此字符串小于字符串参数，则返回一个小于 0 的值；
            //如果此字符串大于字符串参数，则返回一个大于 0 的值。
            if (l1 > l2|| (l1 == l2 && longestWord.compareTo(target) < 0))
{
                continue;
            }
            if (isSubstr(s, target)) {
                longestWord = target;
            }
        }
        return longestWord;
    }

    //双指针判断是否是子串
    private boolean isSubstr(String s, String target) {
        int i = 0, j = 0;
        while (i < s.length() && j < target.length()) {
            if (s.charAt(i) == target.charAt(j)) {
                j++;
            }
            i++;
        }
        return j == target.length();
    }
}
```

排序 - 未写

快速选择

用于求解 **Kth Element** 问题，也就是第 K 个元素的问题。

可以使用快速排序的 partition() 进行实现。需要先打乱数组，否则最坏情况下时间复杂度为 $O(N^2)$。

堆

用于求解 **TopK Elements** 问题，也就是 K 个最小元素的问题。可以维护一个大小为 K 的最小堆，最小堆中的元素就是最小元素。最小堆需要使用大顶堆来实现，大顶堆表示堆顶元素是堆中最大元素。这是因为我们要得到 k 个最小的元素，因此当遍历到一个新的元素时，需要知道这个新元素是否比堆中最大的元素更小，更小的话就把堆中最大元素去除，并将新元素添加到堆中。所以我们需要很容易得到最大元素并移除最大元素，大顶堆就能很好满足这个要求。

堆也可以用于求解 Kth Element 问题，得到了大小为 k 的最小堆之后，因为使用了大顶堆来实现，因此堆顶元素就是第 k 大的元素。

快速选择也可以求解 TopK Elements 问题，因为找到 Kth Element 之后，再遍历一次数组，所有小于等于 Kth Element 的元素都是 TopK Elements。

可以看到，快速选择和堆排序都可以求解 Kth Element 和 TopK Elements 问题。

1. Kth Element--215--Medium

桶排序

1. 出现频率最多的 k 个元素--347--Medium

2. 按照字符出现次数对字符串排序--451--Medium

荷兰国旗问题

1. 按颜色进行排序--75--Medium

贪心思想 - 未写完

1. 分配饼干--455--Easy

```
class Solution {
    //贪心的思想是，用尽量小的饼干去满足小需求的孩子，所以需要进行排序先
    public int findContentChildren(int[] g, int[] s) {
        int child = 0;
        int cookie = 0;
        Arrays.sort(g);  //先将饼干 和 孩子所需大小都进行排序
        Arrays.sort(s);
        while (child < g.length && cookie < s.length ){ //当其中一个遍历就结束
            if (g[child] <= s[cookie]){ //当用当前饼干可以满足当前孩子的需求，可以满足的孩子数量+1
                child++;
            }
            cookie++; // 饼干只可以用一次，因为饼干如果小的话，就是无法满足被抛弃，满足的话就是被用了
        }
        return child;
    }
}
```

2. 不重叠的区间个数--435--Medium

```

```

3. 投飞镖刺破气球--425--Medium

```

```

4. 根据身高和序号重组队列--406--Medium

```

```

5. 买卖股票最大的收益--121--Easy

```java
class Solution {
    public int maxProfit(int[] prices) {
        //寻找最小的值，并在它之后找一个最大的值，取差值即为答案
        int ans = 0;
        int minPrice = Integer.MAX_VALUE;
        for (int i = 0; i < prices.length; i++) {
            if (prices[i] < minPrice)
                minPrice = prices[i];
            ans = Math.max(ans, prices[i]-minPrice);
        }
        return ans;
    }
}
```

6. 买卖股票的最大收益 II--122--Easy

```java
class Solution {
    public int maxProfit(int[] prices) {
        //把问题看作每天都对股票买卖，有利益就在总数上累加
        int sum = 0;
        int temp = 0;
        for (int i = 1; i < prices.length; i++) {
            temp = prices[i]-prices[i-1];
            if(temp > 0)
                sum += temp;
        }
        return sum;
    }
}
```

7. 种植花朵--605--Easy

```java
class Solution {
    public boolean canPlaceFlowers(int[] flowerbed, int n) {
        int count = 0, len = flowerbed.length, pre, next;
        for (int i = 0; i < len; i++) {
            if (flowerbed[i] == 1)
                continue;
            //判断此时的前一位与后一位是否越界
            //判断此时的前一位与后一位是否种花
            pre = i == 0 ? 0 : flowerbed[i - 1];
            next = i == len - 1 ? 0 : flowerbed[i + 1];
            if (pre == 0 && next == 0){
                //将此处种花，并计数
                count++;
```

```
                flowerbed[i] = 1;
            }
        }
        return count >= n;
    }
}
```

8. 判断是否为子序列--329--Medium

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int cur=0;
        //按顺序相等的字符数若等于s的长度，则为子序列
        for (int i = 0; i < t.length()&& cur<s.length(); i++) {
            if(t.charAt(i)==s.charAt(cur)) cur++;
        }
        return cur==s.length();
    }
}
```

9. 修改一个数成为非递减数组--665--Easy

```
class Solution {
    public boolean checkPossibility(int[] nums) {
        int cnt = 0;
        for (int i = 1; i < nums.length && cnt < 2; i++) {
            //非递减情况直接跳下一位
            if (nums[i] >= nums[i - 1]) {
                continue;
            }
            //递减情况计数
            cnt++;
            //判断前两位是否大于当前位，
            //是：修改i位为i-1
            //否：修改i-1位为i
            if (i - 2 >= 0 && nums[i - 2] > nums[i]) {
                nums[i] = nums[i - 1];
            } else {
                nums[i - 1] = nums[i];
            }
        }
        return cnt <= 1;
    }
}
```

10. 子数组最大的和--53--Easy

```
class Solution {
    public int maxSubArray(int[] nums) {
        int ans = nums[0];
        int sum = 0;
        for(int num: nums) {
            //遍历每个元素，当sum大于0时与sum累加
            //否则令sum等于num，意思是若最大自序和为负数，判断当前负元素是否为最大负
元素
            if(sum > 0) {
                sum += num;
            } else {
                sum = num;
            }
            ans = Math.max(ans, sum);
        }
        return ans;
    }
}
```

11. 分隔字符串使同种字符出现在一起--763--Medium

```
class Solution {
    public List<Integer> partitionLabels(String S) {
     // 先用HashMap存每个字符对应的最后index;
        List<Integer> res = new ArrayList<>();
        HashMap<Character,Integer> map = new HashMap();
        for(int i = 0; i < S.length(); i++){
            map.put(S.charAt(i),i);
        }
        int firstIndex = 0;
        while(firstIndex < S.length()){
            //初始化当前的lastIndex，取当前字符的最后index
            int lastIndex = map.get(S.charAt(firstIndex));
            int currIndex = firstIndex;
            //当某一区间内的字符的lastIndex小于当前的lastIndex
            while(currIndex < lastIndex){
                //当前index表示的字符的最后index若大于当前lastIndex
                //则更新lastIndex为此字符的lastIndex
                if(map.get(S.charAt(currIndex)) > lastIndex){
                    lastIndex = map.get(S.charAt(currIndex));
                }
                currIndex++;
            }
            //添加区间长度
```

```
                res.add(lastIndex - firstIndex + 1);
                //赋值下一区间的firstIndex
                firstIndex = lastIndex + 1;
            }
            return res;
        }
    }
```

二分查找

> **正常实现**

```
Input : [1,2,3,4,5]
key : 3
return the index : 2
```

```
public int binarySearch(int[] nums, int key) {
int l = 0, h = nums.length - 1;
while (l <= h) {
    int m = l + (h - l) / 2;
    if (nums[m] == key) {
        return m;
    } else if (nums[m] > key) {
        h = m - 1;
    } else {
        l = m + 1;
    }
}
return -1;
}
```

> **时间复杂度**

二分查找也称为折半查找，每次都能将查找区间减半，这种折半特性的算法时间复杂度为 O(logN)。

> **m 计算**

有两种计算中值 m 的方式：

- m = (l + h) / 2
- m = l + (h - l) / 2

l + h 可能出现加法溢出，也就是说加法的结果大于整型能够表示的范围。但是 l 和 h 都为正数，因此 h - l 不会出现加法溢出问题。所以，最好使用第二种计算法方法。

> **未成功查找的返回值**

循环退出时如果仍然没有查找到 key，那么表示查找失败。可以有两种返回值：

- -1：以一个错误码表示没有查找到 key
- l：将 key 插入到 nums 中的正确位置

变种

二分查找可以有很多变种，实现变种要注意边界值的判断。例如在一个有重复元素的数组中查找 key 的最左位置的实现如下：

```
public int binarySearch(int[] nums, int key) {
    int l = 0, h = nums.length - 1;
    while (l < h) {
        int m = l + (h - l) / 2;
        if (nums[m] >= key) {
            h = m;
        } else {
            l = m + 1;
        }
    }
    return l;
}
```

该实现和正常实现有以下不同：

- h 的赋值表达式为 h = m
- 循环条件为 l < h
- 最后返回 l 而不是 -1

在 nums[m] >= key 的情况下，可以推导出最左 key 位于 [l, m] 区间中，这是一个闭区间。h 的赋值表达式为 h = m，因为 m 位置也可能是解。

在 h 的赋值表达式为 h = m 的情况下，如果循环条件为 l <= h，那么会出现循环无法退出的情况，因此循环条件只能是 l < h。以下演示了循环条件为 l <= h 时循环无法退出的情况：

```
nums = {0, 1, 2}, key = 1
l   m   h
0   1   2  nums[m] >= key
0   0   1  nums[m] < key
1   1   1  nums[m] >= key
1   1   1  nums[m] >= key
...
```

当循环体退出时，不表示没有查找到 key，因此最后返回的结果不应该为 -1。为了验证有没有查找到，需要在调用端判断一下返回位置上的值和 key 是否相等。

1. 求开方--69--Easy

```
class Solution {
    public int mySqrt(int x) {
```

```
        //定义左右指针。
        int l = 0, r = x, ans = -1;
        while (l <= r) {
            //定义中点
            int mid = l + (r - l) / 2;
            //若中点的平方大于x，令右指针为中点-1，反之暂令所求结果为中点
            if ((long)mid * mid <= x) {
                ans = mid;
                l = mid + 1;
            }
            else {
                r = mid - 1;
            }
        }
        return ans;
    }
}
```

2. 大于给定元素的最小元素--744--Easy

```
class Solution {
    public char nextGreatestLetter(char[] letters, char target) {
        int len = letters.length;
        int l = 0, r = len - 1;
        while (l <= r){
            int mid = l + (r - l)/2;
            if (letters[mid] <= target)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return letters[l % len];
    }
}
```

3. 有序数组的 Single Element--540--Easy

```
class Solution {
    public int singleNonDuplicate(int[] nums) {
        int l = 0, r = nums.length - 1;//nums长度必为奇数
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (mid % 2 == 1) {
                mid--;   // 保证 l/h/m 都在偶数位，使得查找区间大小一直都是奇数
            }
            if (nums[mid] == nums[mid + 1]) {
```

```
                l = mid + 2;
            } else {
                r = mid;
            }
        }
        return nums[l];
    }
}
```

4. 第一个错误的版本--278--Easy

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int l = 1, r = n;
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (isBadVersion(mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l;
    }
}
```

5. 旋转数组的最小数字--153--Easy

```
class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int m = l + (r - l) / 2;
            //由于元素是连续递增的，不重复且只有一处断点，用此条件可判断
            if (nums[m] <= nums[r]) {
                r = m;
            } else {
                l = m + 1;
            }
        }
        return nums[l];
    }
}
```

6. 查找区间--34--Easy

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
         int [] result = new int[2];
            if(nums.length == 0){
                result[0] = -1;
                result[1] = -1;
                return  result;
            }
            result[0] = findFirst(nums, nums.length, target);
            result[1] = findLast(nums, nums.length, target);
            return result;
        }

        public int findFirst(int [] a, int len, int key) {
            if (len < 1)
                return - 1;
            int low = 0;
            int high = len - 1;
            while(low <= high) {
                int mid = (low + high)/2;
                //寻找到目标值，且当mid-1越界或a[mid-1]不等于目标值
                if (a[mid] == key && (mid - 1 < 0 || a[mid - 1] != key)) {
                    return mid;
                } else if (a[mid] >= key) {
                    high = mid - 1;
                } else {//a[mid] < key 或 a[mid] == key && (mid - 1 >= 0 ||
a[mid - 1] == key))
                    low = mid + 1;

                }
                System.out.println("findF   low的变化:"+low);
            }
            return -1;
        }

        public int findLast(int [] a, int len, int key) {
            if (len < 1)
                return - 1;
            int low = 0;
            int high = len - 1;
            while(low <= high){
                int mid = (low + high)/2;
                //寻找到目标值，且当mid+1越界或a[mid+1]不等于目标值
                if (a[mid] == key && (mid + 1 >= len || a[mid + 1] !=
key)) {
                    return mid;
                } else if (a[mid] <= key) {
```

```
                    low = mid + 1;
                } else {//a[mid] > key 或 a[mid] == key && (mid + 1 < len
|| a[mid + 1] == key)
                    high = mid - 1;

                }
                System.out.println("findL   high的变化:"+high);
            }
            return -1;
        }
}
```

分治 - 未写

1. 给表达式加括号--241--Medium

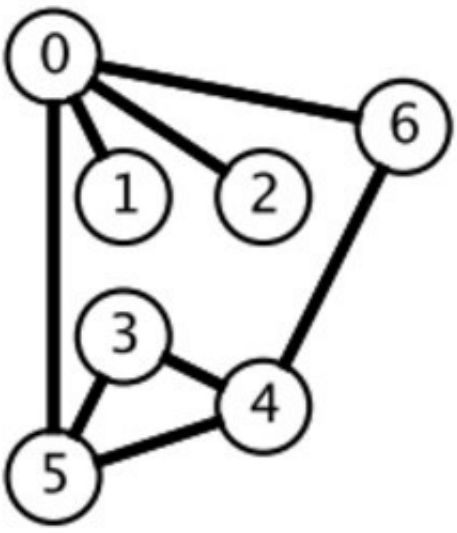
```

```

2. 不同的二叉搜索树--95--Medium

```

```

搜索 - 未写完

BFS



> 广度优先搜索一层一层地进行遍历，每层遍历都是以上一层遍历的结果作为起点，遍历一个距离能访问到的所有节点。需要注意的是，遍历过的节点不能再次被遍历。
>
> 第一层：
>
> - 0 -> {6,2,1,5}
>
> 第二层：
>
> - 6 -> {4}
> - 2 -> {}
> - 1 -> {}
> - 5 -> {3}
>
> 第三层：
>
> - 4 -> {}
> - 3 -> {}
>
> 每一层遍历的节点都与根节点距离相同。设 d_i 表示第 i 个节点与根节点的距离，推导出一个结论：对于先遍历的节点 i 与后遍历的节点 j，有 $d_i <= d_j$。利用这个结论，可以求解最短路径等 **最优解** 问题：第一次遍历到目的节点，其所经过的路径为最短路径。应该注意的是，使用 BFS 只能求解无权图的最短路径，无权图是指从一个节点到另一个节点的代价都记为 1。
>
> 在程序实现 BFS 时需要考虑以下问题：
>
> - 队列：用来存储每一轮遍历得到的节点；
> - 标记：对于遍历过的节点，应该将它标记，防止重复遍历。

1. 计算在网格中从原点到特定点的最短路径长度--1091--Medium

```
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return -1;
        }
        int ans = 0;
        int[][] direction = {{0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1},
{-1, -1}, {-1, 0}, {-1, 1}};
        int m = grid.length, n = grid[0].length;
        //bfs的老套路 来个队列
        Queue<int[]> queue = new LinkedList<>();
        queue.add(new int[]{0, 0});//把起点扔进去
        while (!queue.isEmpty()){
            int size =queue.size();
            ans++;
            while (size-- > 0){
                int[] cur = queue.poll();
                int crrX = cur[0], curY = cur[1];
                if (grid[crrX][curY] == 1){
                    continue;
                }
```

```
                // 能放进队列里的都是为0可以走的（这一点在后面保证了）
                // 如果等于终点则返回
                if (crrX == m - 1 && curY == n - 1){
                    return ans;
                }
                grid[crrX][curY] = 1;//标记
                //开始八个方向的判断
                for (int[] d : direction) {
                    int newX = crrX + d[0], newY = curY + d[1];
                    //这里开始过滤
                    if (newX < 0 || newX >= m || newY < 0 || newY >= n){
                        continue;
                    }
                    //把在数组范围内 并且为0不阻塞的放入队列中
                    queue.add(new int[]{newX, newY});
                }
            }
        }
        return -1;
    }
}
```

2. 组成整数的最小平方数数量--279--Medium

```
//动态规划
class Solution {
    // dp[i]：表示完全平方数和为i的 最小个数
    // 初始状态dp[i]均取最大值i，即 1+1+...+1，i个1；dp[0] = 0
    // dp[i] = min(dp[i], dp[i-j*j]+1)，其中，j是平方数，j=1~k,其中k*k要保证
<= i
    // 意思就是：完全平方数和为i的 最小个数 等于 当前完全平方数和为i的 最大个数
    //   与 （完全平方数和为 i - j * j 的 最小个数 + 完全平方数和为 j * j的 最小个
数）
    //   可以看到 dp[j*j] 是等于1的
    public int numSquares(int n) {
        int[] dp = new int[n + 1]; // 默认初始化值都为0
        for (int i = 1; i <= n; i++) {
            dp[i] = i; // 最坏的情况就是每次+1
            for (int j = 1; i - j * j >= 0; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1); // 动态转移方程
            }
        }
        return dp[n];
    }
}

//BFS
```

```java
class Solution {
    public int numSquares(int n) {
        //生成小于n平方数序列
        List<Integer> squares = generateSquares(n);
        //BFS套路队列
        Queue<Integer> queue = new LinkedList<>();
        //标记数组
        boolean[] marked = new boolean[n + 1];
        //扔进起点
        queue.add(n);
        marked[n] = true;
        int level = 0;
        while (!queue.isEmpty()) {
            int size = queue.size();
            level++;
            while (size-- > 0) {
                int cur = queue.poll();
                //遍历每个平方数,
                for (int s : squares) {
                    int next = cur - s;
                    //next小于0，不符合条件
                    if (next < 0) {
                        break;
                    }
                    //减至为0，返回level
                    if (next == 0) {
                        return level;
                    }
                    //此平方数已被标记，进行下一次循环
                    if (marked[next]) {
                        continue;
                    }
                    //标记当前数减去所有平方数的剩余数
                    //并加入队列
                    //将次数的所有剩余数加入队列，BFS思想
                    marked[next] = true;
                    queue.add(next);
                }
            }
        }
        return n;
    }

    /**
     * 生成小于 n 的平方数序列
     * @return 1,4,9,...
     */
    private List<Integer> generateSquares(int n) {
        List<Integer> squares = new ArrayList<>();
```

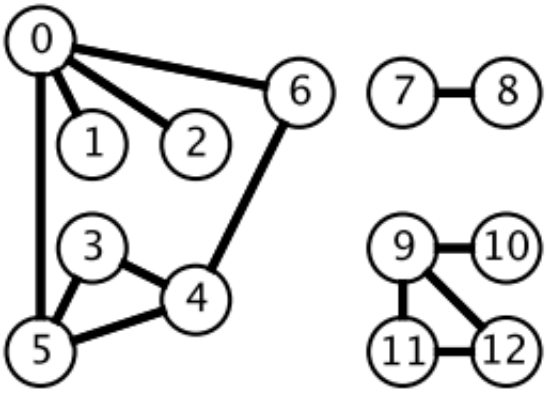
```
        int square = 1;
        int diff = 3;
        while (square <= n) {
            squares.add(square);
            square += diff;
            diff += 2;
        }
        return squares;
    }
}
```

3. 最短单词路径--127--Medium

```
class Solution {
    //BFS的思想
 public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
        Queue<String> queue = new LinkedList<>();//少不了队列
        queue.add(beginWord);//加入起始词
        boolean[] marked = new boolean[wordList.size() + 1];//少不了标记
        int layer = 1;//注意返回的是层数+1.所以这里直接放1了
        while(!queue.isEmpty()) {//固定的层数记录形式
            layer++;
            int size = queue.size();
            while (size-- > 0) {
                String cur = queue.poll();
                for (int i = 0; i < wordList.size(); i++) {
                    if(marked[i])
                        continue;
                    String dic = wordList.get(i);
                    if(canChange(dic, cur)) {
                        //先到达endWord就是最快换取法
                        if(dic.equals(endWord))
                            return layer;
                        //将当前词能改的词都存入队列
                        queue.add(dic);
                        marked[i] = true;
                    }
                }
            }
        }
        return 0;
    }
    //是否可以转换的辅助函数
    public boolean canChange(String s, String t) {
        int diff = 0;
        for (int i = 0; i < s.length(); i++) {
```

```
            if(s.charAt(i) != t.charAt(i))
                diff++;
        }
        return diff == 1;
    }
}
```

DFS



广度优先搜索一层一层遍历，每一层得到的所有新节点，要用队列存储起来以备下一层遍历的时候再遍历。

而深度优先搜索在得到一个新节点时立即对新节点进行遍历：从节点 0 出发开始遍历，得到到新节点 6 时，立马对新节点 6 进行遍历，得到新节点 4；如此反

复以这种方式遍历新节点，直到没有新节点了，此时返回。返回到根节点 0 的情况是，继续对根节点 0 进行遍历，得到新节点 2，然后继续以上步骤。

从一个节点出发，使用 DFS 对一个图进行遍历时，能够遍历到的节点都是从初始节点可达的，DFS 常用来求解这种 **可达性** 问题。

在程序实现 DFS 时需要考虑以下问题：

- 栈：用栈来保存当前节点信息，当遍历新节点返回时能够继续遍历当前节点。可以使用递归栈。
- 标记：和 BFS 一样同样需要对已经遍历过的节点进行标记。

1. 查找最大的连通面积--695--Medium

```
class Solution {
    private int m, n;
    private int[][] direction = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public int maxAreaOfIsland(int[][] grid) {
        if (grid == null || grid.length == 0 ||grid[0].length ==0 )
            return 0;
        int maxArea = 0;
```

```
        m = grid.length;
        n = grid[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                maxArea = Math.max(maxArea, dfs(grid, i, j));
            }
        }
        return maxArea;
    }

    public int dfs(int[][] grid, int x, int y){
        if (x < 0 || x >= m ||y < 0 || y >= n || grid[x][y] == 0)
            return 0;
        //置为0表示已经访问过
        grid[x][y] = 0;
        int area = 1;
        //对四个方向进行dfs遍历
        for (int[] d : direction) {
            area += dfs(grid, x + d[0], y + d[1]);
        }
        return area;
    }
}
```

2. 矩阵中的连通分量数目--200--Medium

```
class Solution {
    private int m, n;
    private int[][] direction = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0)
            return 0;
        int ans  = 0;
        m = grid.length;
        n = grid[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1'){
                    dfs(grid, i, j);
                    ans++;
                }
            }
        }
        return ans;
    }
```

```
    private void dfs(char[][] grid, int x, int y){
        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] != '1')
            return;//不再符合岛的条件，回溯
        //置为2标记为已经访问
        grid[x][y] = '2';
        for (int[] d : direction) {
            dfs(grid, x + d[0], y + d[1]);
        }
    }
}
```

3. 好友关系的连通分量数目--547--Medium

```
class Solution {
    private int m;

    public int findCircleNum(int[][] M) {
        //注意：M[i][j]表达的意思是i和j是朋友
        //可以看作无向图
        if (M == null || M.length == 0 || M[0].length == 0)
            return 0;
        int ans  = 0;
        m = M.length;
        //使用一个visited数组，依次判断每个节点，
        //如果其未访问，朋友圈数加1并对该节点进行dfs搜索标记所有访问到的节点
        boolean[] isVisited = new boolean[m];
        for (int i = 0; i < m; i++) {
            if (!isVisited[i]){
                dfs(M, i, isVisited);
                ans++;
            }
        }
        return ans;
    }

    private void dfs(int[][] M, int i, boolean[] isVisited){
        isVisited[i] = true;
        for (int j = 0; j < m; j++) {
            if (M[i][j] == 1 && !isVisited[j])
                dfs(M, j, isVisited);
        }
    }
}
```

4. 填充封闭区域--130--Medium

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

    public void solve(char[][] board) {
        if (board == null || board.length == 0 || board[0].length == 0) {
            return;
        }

        row = board.length;
        col = board[0].length;
        //遍历边界，第0列和最后一列，标记所有未为x包围的o
        for (int i = 0; i < row; i++) {
            dfs(board, i, 0);
            dfs(board, i, col - 1);
        }
        //遍历边界，第0行和最后一行，标记所有未为x包围的o
        for (int i = 0; i < col; i++) {
            dfs(board, 0, i);
            dfs(board, row - 1, i);
        }
        //将未被标记的o全部改为x
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (board[i][j] == 'T') {
                    board[i][j] = 'O';
                } else if (board[i][j] == 'O') {
                    board[i][j] = 'X';
                }
            }
        }
    }

    private void dfs(char[][] board, int r, int c) {
        if (r < 0 || r >= row || c < 0 || c >= col || board[r][c] != 'O')
{
            return;
        }
        //标记为'T'表示为已经访问过
        board[r][c] = 'T';
        //四个方向的遍历
        for (int[] d : direction) {
            dfs(board, r + d[0], c + d[1]);
        }
    }
}
```

5. 能到达的太平洋和大西洋的区域--417--Medium

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

    public List<List<Integer>> pacificAtlantic(int[][] matrix) {
        List<List<Integer>> ansList = new ArrayList<>();
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return ansList;
        row = matrix.length;
        col = matrix[0].length;
        this.matrix = matrix;//延伸至全局可用
        boolean[][] canReachP = new boolean[row][col];
        boolean[][] canReachA = new boolean[row][col];
        //遍历边界，第0列和最后一列，
        for (int i = 0; i < row; i++) {
            dfs(canReachP, i, 0);//太平洋
            dfs(canReachA, i, col - 1);//大西洋
        }
        //遍历边界，第0行和最后一行，
        for (int i = 0; i < col; i++) {
            dfs(canReachP, 0, i);//太平洋
            dfs(canReachA, row - 1, i);//大西洋
        }
        //遍历所有点，将能流向大西洋和太平洋的存入ansList
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (canReachP[i][j] && canReachA[i][j])
                    ansList.add(Arrays.asList(i, j));
            }
        }
        return ansList;
    }

    private void dfs(boolean[][] canReach, int r, int c){
        if (canReach[r][c])//标记过则回溯
            return;
        //标记
        canReach[r][c] = true;
        //对四个不同方向进行dfs遍历
        for (int[] d : direction) {
            int nextR = d[0] + r;
            int nextC = d[1] + c;
            //条件过滤
            if (nextR < 0 || nextR >= row || nextC < 0 || nextC >= col ||
matrix[r][c] > matrix[nextR][nextC])
                continue;
            dfs(canReach, nextR, nextC);
        }
```

```
    }
}
```

Backtracking

> Backtracking（回溯）属于 DFS。回溯算法事实上就是在一个树形问题上做深度优先遍历，因此**首先需要把问题转换为树形问题**。
>
> - 普通 DFS 主要用在 **可达性问题** ，这种问题只需要执行到特点的位置然后返回即可。
> - 而 Backtracking 主要用于求解 **排列组合** 问题，例如有 { 'a','b','c' } 三个字符，求解所有由这三个字符排列得到的字符串，这种问题在执行到特定的位置返回之后还会继续执行求解过程。
>
> 因为 Backtracking 不是立即返回，而要继续求解，因此在程序实现时，需要注意对元素的标记问题：
>
> - 在访问一个新元素进入新的递归调用时，需要将新元素标记为已经访问，这样才能在继续递归调用时不用重复访问该元素；
> - 但是在递归返回时，需要将元素标记为未访问，因为只需要保证在一个递归链中不同时访问一个元素，可以访问已经访问过但是不在当前递归链中的元素。

1. 数字键盘组合--17--Medium

```
class Solution {
    private String[] keys = {"", "", "abc", "def", "ghi", "jkl", "mno",
"pqrs", "tuv", "wxyz"};

    public List<String> letterCombinations(String digits) {
        List<String> ans = new ArrayList<>();
        //判断临界情况
        if (digits == null || digits.length() == 0)
            return ans;
        doCombination(new StringBuilder(), ans, digits);
        return ans;
    }

    private void doCombination(StringBuilder prefix, List<String> ans,
String digits){
        //当前字符穿长度等于数字组合长度
        //表示当前是一种组合的答案，存入答案后回溯
        if (prefix.length() == digits.length()) {
            ans.add(prefix.toString());
            return;
        }
        //利用ascii码，将当前位转成数字
        int curDigits = digits.charAt(prefix.length()) - '0';
        //取出数字代表的字符
        String letters = keys[curDigits];
```

```
        for (char c : letters.toCharArray()) {
            prefix.append(c); //添加
            doCombination(prefix, ans, digits);
            //回溯去掉最后一位，接着起始前进一位
            prefix.deleteCharAt(prefix.length() - 1); //删除
        }
        //注意，遍历完当前letters后回溯到上一层的letters
        //如，String为"234"，再遍历完"4"的首轮"ghi"后，自动回溯到"3"的"def"
        //此时，"d"删去，存入"e"，然后接着遍历"4"的"ghi"
    }
}
```

2. IP 地址划分--93--Medium

```
class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> addresses = new ArrayList<>();
        StringBuilder tempAddress = new StringBuilder();
        doRestore(0, tempAddress, addresses, s);
        return addresses;
    }

    private void doRestore(int k, StringBuilder tempAddress, List<String>
addresses, String s) {
        //判断是否已经存了三位ip或剩余的s是否为0
        if (k == 4 || s.length() == 0) {
            //ip末尾
            if (k == 4 && s.length() == 0) {
                addresses.add(tempAddress.toString());
            }
            return;
        }
        for (int i = 0; i < s.length() && i <= 2; i++) {
            // 非0位存在第一位为0字符为非法ip
            if (i != 0 && s.charAt(0) == '0') {
                break;
            }
            //取出后i位字符串
            String part = s.substring(0, i + 1);
            //转为整形判断是否小于等于255，为合法IP地址
            if (Integer.valueOf(part) <= 255) {
                if (tempAddress.length() != 0) {
                    part = "." + part;
                }
                tempAddress.append(part); // 添加
                doRestore(k + 1, tempAddress, addresses, s.substring(i +
1));
```

```
                tempAddress.delete(tempAddress.length() - part.length(),
tempAddress.length()); // 删除
            }
        }
    }
}

//暴力
class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> r = new ArrayList<>();
        if(s.length() > 12) return r;
        for(int i = 1; i < s.length() && i <= 3; ++i){
            for(int j = i; j < s.length() && j <= i + 3; ++j){
                for(int k = j; k < s.length() && k <= j + 3; ++k){
                    String s1 = s.substring(0, i);
                    String s2 = s.substring(i, j);
                    String s3 = s.substring(j, k);
                    String s4 = s.substring(k);
                    if(f(s1) && f(s2) && f(s3) && f(s4)){
                        StringBuilder sb = new StringBuilder();
                        sb.append(s1); sb.append(".");
                        sb.append(s2); sb.append(".");
                        sb.append(s3); sb.append(".");
                        sb.append(s4);
                        r.add(sb.toString());
                    }
                }
            }
        }
        return r;
    }

    boolean f(String s){
        if(s.length() == 0) return false;
        if(s.length() == 1) return true;
        if(s.length() > 3) return false;
        if(s.charAt(0) == '0') return false;
        if(Integer.parseInt(s) <= 255) return true;
        return false;
    }
}
```

3. 在矩阵中寻找字符串--79--Medium

```
class Solution {
    private int[][] direction = {{1, 0}, {-1, 0}, {0, 1}, {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[][] visited = new boolean[row][col];
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (backTracking(0, i, j, visited, board, word)) {
                    return true;
                }
            }
        }
        return false;
    }

    private boolean backTracking(int curLen, int r, int c, boolean[][]
visited, char[][] board, String word) {
        //curLen从0开始，若与word长度相等，则证明前面各位字母相等，返回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) || visited[r][c]) {
            return false;
        }
        //标记为已经访问
        visited[r][c] = true;
        //四个方向的回溯
        for (int[] d : direction) {
            if (backTracking(curLen + 1, r + d[0], c + d[1], visited,
board, word)) {
                return true;
            }
        }
        //下个起点可能需要再次访问此点，所以去除标记
        visited[r][c] = false;
        return false;
    }
}
```

4. 输出二叉树中所有从根到叶子的路径--257--Easy

```java
class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> ans = new ArrayList<>();
        if (root == null)
            return ans;
        List<Integer> values = new ArrayList<>();
        backTracking(root, values, ans);
        return ans;
    }

    //回溯
    private void backTracking(TreeNode node, List<Integer> values,
List<String> ans){
        if (node == null) // 节点为空则回溯
            return;
        values.add(node.val);// 添加此时节点值
        if (isLeft(node)){
            ans.add(buildPath(values)); // 添加已有值的生成路径
        } else {
            backTracking(node.left, values, ans); // 回溯左子树
            backTracking(node.right, values, ans); // 回溯右子树
        }
        values.remove(values.size() - 1); //删除此时节点值
    }

    //判断是否是叶子节点
    private boolean isLeft(TreeNode node){
        return node.left == null && node.right == null;
    }

    //构造答案所需的路径结构
    private String buildPath(List<Integer> values){
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < values.size(); i++) {
            sb.append(values.get(i));
            if (i != values.size() - 1) {
                sb.append("->");
            }
        }
        return sb.toString();
    }
}
```

5. 排列--46--Medium

```java
class Solution {
    public List<List<Integer>> permute(int[] nums) {
```

```
        List<List<Integer>> res = new ArrayList<>();
        boolean[] visited = new boolean[nums.length];
        backtrack(res, nums, new ArrayList<Integer>(), visited);
        return res;
    }

    private void backtrack(List<List<Integer>> res, int[] nums,
ArrayList<Integer> tmp, boolean[] visited) {
        if (tmp.size() == nums.length) {
            res.add(new ArrayList<>(tmp));// 重新构造一个 List
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (visited[i]) continue;
            visited[i] = true; //标记为已经访问
            tmp.add(nums[i]); // 添加元素
            backtrack(res, nums, tmp, visited); // 回溯
            visited[i] = false; //去除标记
            tmp.remove(tmp.size() - 1); //删除元素
        }
        //注意，遍历完当前nums后回溯到上一层的nums
        //如 1 2 3 到 1 3 2
    }
}
```

6. 含有相同元素求排列--47--Medium

```
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        Set<List<Integer>> listSet = new HashSet<>(); // 利用hashset去重
        List<List<Integer>> ans = new ArrayList<>();
        boolean[] visited = new boolean[nums.length];
        backtrack(listSet, nums, new ArrayList<Integer>(), visited);
        for (List<Integer> tmp : listSet) {
            ans.add(tmp);
        }
        return ans;
    }

    private void backtrack(Set<List<Integer>> listSet, int[] nums,
ArrayList<Integer> tmp, boolean[] visited) {
        if (tmp.size() == nums.length) {
            listSet.add(new ArrayList<>(tmp));// 重新构造一个 List
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (visited[i]) continue;
```

```
            visited[i] = true; //标记为已经访问
            tmp.add(nums[i]); // 添加元素
            backtrack(listSet, nums, tmp, visited); // 回溯
            visited[i] = false; //去除标记
            tmp.remove(tmp.size() - 1); //删除元素
        }
        //注意，遍历完当前nums后回溯到上一层的nums
        //如 1 2 3 到 1 3 2
    }
}
```

7. 组合--77--Medium

```
class Solution {
    private List<List<Integer>> ans = new ArrayList<>();
    private List<Integer> temp = new ArrayList<>();
    private int k, n;

    public List<List<Integer>> combine(int n, int k) {
        this.k = k;
        this.n = n;
        backtrack(temp, 0);
        return ans;
    }

    private void backtrack(List<Integer> tmp, int x) {
        if (tmp.size() == k) {
            ans.add(new ArrayList<>(tmp));// 重新构造一个 List
            return;
        }
        for (int i = x + 1; i <= n; i++) {
            tmp.add(i); // 添加元素
            backtrack(tmp, i); // 起始元素往下，并对起始元素后对元素回溯
            tmp.remove(tmp.size() - 1); //删除元素
        }
    }
}
```

8. 组合求和--39--Medium

```
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        List<List<Integer>> ans = new ArrayList<>();
        backtracking(new ArrayList<>(), ans, 0, target, candidates);
        return ans;
```

```
    }

    private void backtracking(List<Integer> temp, List<List<Integer>> ans,
int start, int target, int[] candidates) {

        if (target == 0) {
            ans.add(new ArrayList<>(temp));
            return;
        }
        for (int i = start; i < candidates.length; i++) {
            if (candidates[i] <= target) {
                temp.add(candidates[i]); // 添加
                // target减去添加的数，注意此处i不需+1，因为candidate可重复使用
                backtracking(temp, ans, i, target - candidates[i],
candidates);
                temp.remove(temp.size() - 1); // 删除
            }
        }
    }
}
```

9. 含有相同元素的组合求和--40--Medium

```
// HashSet 解决一切，但是效率低
class Solution {
    private Set<List<Integer>> ansSet = new HashSet<>();

    public List<List<Integer>> combinationSum2(int[] candidates, int
target) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(candidates);
        boolean[] isVisited = new boolean[candidates.length];
        backtracking(new ArrayList<>(), 0, target, candidates, isVisited);
        for (List<Integer> temp : ansSet) {
            ans.add(temp);
        }
        return ans;
    }

    private void backtracking(List<Integer> temp, int start, int target,
int[] candidates, boolean[] isVisited) {

        if (target == 0) {
            ansSet.add(new ArrayList<>(temp));
            return;
        }
        for (int i = start; i < candidates.length; i++) {
```

```
            if (isVisited[i]) //
                continue;
            if (candidates[i] <= target) {
                temp.add(candidates[i]); // 添加
                isVisited[i] = true; // 标记
                // target减去添加的数，注意此处i+1，因为candidate不可重复使用
                backtracking(temp, i + 1, target - candidates[i],
candidates, isVisited);
                temp.remove(temp.size() - 1); // 删除
                isVisited[i] = false; // 去除标记
            }
        }
    }
}

//20%和90%的区别
class Solution {
   public List<List<Integer>> combinationSum2(int[] candidates, int
target) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(candidates);
        boolean[] isVisited = new boolean[candidates.length];
        backtracking(new ArrayList<>(), ans, 0, target, candidates,
isVisited);
        return ans;
    }

    private void backtracking(List<Integer> temp, List<List<Integer>> ans,
int start, int target, int[] candidates, boolean[] isVisited) {

        if (target == 0) {
            ans.add(new ArrayList<>(temp));
            return;
        }
        for (int i = start; i < candidates.length; i++) {
            // 某个candidates未被标记，但是与之前的candidates值相等，进行下一次循
环
            if (i != 0 && candidates[i] == candidates[i - 1] &&
!isVisited[i - 1])
                continue;
            if (candidates[i] <= target) {
                temp.add(candidates[i]); // 添加
                isVisited[i] = true; // 标记
                // target减去添加的数，注意此处i+1，因为candidate不可重复使用
                backtracking(temp, ans, i + 1, target - candidates[i],
candidates, isVisited);
                temp.remove(temp.size() - 1); // 删除
                isVisited[i] = false; // 去除标记
            }
```

```
            }
        }
    }
```

10. 1-9 数字的组合求和--216--Medium

```

```

11. 子集--78--Medium

```

```

12. 含有相同元素求子集--90--Medium

```

```

13. 分割字符串使得每个部分都是回文数--131--Medium

```

```

14. 数独--37--Hard

```

```

15. N 皇后--51--Hard

```

```

动态规划 - 未写完

递归和动态规划都是将原问题拆成多个子问题然后求解，他们之间最本质的区别是，动态规划保存了子问题的解，避免重复计算。

斐波那契数列

1. 爬楼梯--70--Easy

```
class Solution {
    public int climbStairs(int n) {
        //总体思路，第n阶的方法数等于第n-1阶的方法数加第n-2阶的方法数
        if (n==1)
            return 1;
        int[] dp = new int[n+1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i < n + 1; i++) {
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[n];
    }
}
//考虑到 dp[i] 只与 dp[i - 1] 和 dp[i - 2] 有关，因此可以只用两个变量来存储 dp[i - 1] 和 dp[i - 2]，使得原来的 O(N) 空间复杂度优化为 O(1) 复杂度。
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) {
            return n;
        }
        int pre2 = 1, pre1 = 2;
        for (int i = 2; i < n; i++) {
            int cur = pre1 + pre2;
            pre2 = pre1;
            pre1 = cur;
        }
        return pre1;
    }
}
```

2. 强盗抢劫--198--Easy

```
class Solution {
    public int rob(int[] nums) {
        int len = nums.length;
        if(len == 0)
            return 0;
        int[] dp = new int[len + 1];
        dp[0] = 0;
        dp[1] = nums[0];
        for(int i = 2; i <= len; i++) {
            //动态规划方程
            dp[i] = Math.max(dp[i-1], dp[i-2] + nums[i-1]);
        }
```

```
            return dp[len];
        }
    }
```

3. 强盗在环形街区抢劫--213--Medium

```
class Solution {
    //其实就是把环拆成两个队列，一个是从0到n-2，另一个是从1到n-1，然后返回两个结果最大的。
    //选择0到n-2与1到n-1可以避开首末相连的情况
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int n = nums.length;
        if (n == 1) {
            return nums[0];
        }
        return Math.max(rob(nums, 0, n - 2), rob(nums, 1, n - 1));
    }

    private int rob(int[] nums, int first, int last) {
        int pre2 = 0, pre1 = 0;
        for (int i = first; i <= last; i++) {
            int cur = Math.max(pre1, pre2 + nums[i]);
            pre2 = pre1;
            pre1 = cur;
        }
        return pre1;
    }
}
```

4. 信件错排

题目描述：有 N 个 信 和 信封，它们被打乱，求错误装信方式的数量。

定义一个数组 dp 存储错误方式数量，dp[i] 表示前 i 个信和信封的错误方式数量。假设第 i 个信装到第 j 个信封里面，而第 j 个信装到第 k 个信封里面。根据 i 和 k 是否相等，有两种情况：

- i==k，交换 i 和 j 的信后，它们的信和信封在正确的位置，但是其余 i-2 封信有 dp[i-2] 种错误装信的方式。由于 j 有 i-1 种取值，因此共有 (i-1)*dp[i-2] 种错误装信方式。
- i != k，交换 i 和 j 的信后，第 i 个信和信封在正确的位置，其余 i-1 封信有 dp[i-1] 种错误装信方式。由于 j 有 i-1 种取值，因此共有 (i-1)*dp[i-1] 种错误装信方式。

综上所述，错误装信数量方式数量为：

5. 母牛生产

   题目描述：假设农场中成熟的母牛每年都会生 1 头小母牛，并且永远不会死。第一年有 1 只小母牛，从第二年开始，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。给定整数 N，求 N 年后牛的数量。

   第 i 年成熟的牛的数量为：

矩阵路径

1. 矩阵的最小路径和--64--Medium

```
class Solution {
    public int minPathSum(int[][] grid) {
        //求从矩阵的左上角到右下角的最小路径和，每次只能向右和向下移动。
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(i == 0 && j == 0)
                    continue;
                else if(i == 0)  // 只能从上侧走到该位置
                    grid[i][j] = grid[i][j - 1] + grid[i][j];
                else if(j == 0)  // 只能从上侧走到该位置
                    grid[i][j] = grid[i - 1][j] + grid[i][j];
                else
                    grid[i][j] = Math.min(grid[i - 1][j], grid[i][j - 1])
+ grid[i][j];
            }
        }
        return grid[grid.length - 1][grid[0].length - 1];
    }
}
```

2. 矩阵的总路径数--198--Easy

```
//动态规划
class Solution {
 public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0 || j == 0)//对于第0行和第0列。机器人只能右走或往下走，
所以路径数都为1
                    dp[i][j] = 1;
```

```
                else {
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                }
            }
        }
        return dp[m - 1][n - 1];
    }
}
//数学方法
class Solution {
    public int uniquePaths(int m, int n) {
        int S = m + n - 2;  // 总共的移动次数
        int D = m - 1;      // 向下的移动次数
        long ret = 1;
        for (int i = 1; i <= D; i++) {
            ret = ret * (S - D + i) / i;
        }
        return (int) ret;
    }
}
```

数组区间

1. 数组区间和--303--Easy

```
class NumArray {

    private int[] sums;
    //NumArray
    //重点在多次调用
    public NumArray(int[] nums) {
        sums = new int[nums.length + 1];
        for (int i = 1; i <= nums.length; i++) {
            sums[i] = sums[i - 1] + nums[i - 1];
        }
    }

    public int sumRange(int i, int j) {
        return sums[j + 1] - sums[i];
    }
}
```

2. 数组中等差递增子区间的个数--413--Medium

```
class Solution {
```

```
    public int numberOfArithmeticSlices(int[] A) {
     //例子:
        //dp[2] = 1
        //    [0, 1, 2]
        //dp[3] = dp[2] + 1 = 2
        //    [0, 1, 2, 3], // [0, 1, 2] 之后加一个 3
        //    [1, 2, 3]     // 新的递增子区间
        //dp[4] = dp[3] + 1 = 3
        //    [0, 1, 2, 3, 4], // [0, 1, 2, 3] 之后加一个 4
        //    [1, 2, 3, 4],    // [1, 2, 3] 之后加一个 4
        //    [2, 3, 4]        // 新的递增子区间
        //综上，在 A[i] - A[i-1] == A[i-1] - A[i-2] 时，dp[i] = dp[i-1] + 1。
        //
        //因为递增子区间不一定以最后一个元素为结尾，可以是任意一个元素结尾，因此需要返回 dp 数组累加的结果。
        if (A == null || A.length == 0)
            return 0;
        int len = A.length;
        int[] dp = new int[len];
        for (int i = 2; i < len; i++) {
            if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
                dp[i] = dp[i - 1] + 1;
            }
        }
        int ans = 0;
        for (int cur:dp) {
            ans += cur;
        }
        return ans;
    }
}
```

分割整数

1. 分割整数的最大乘积--343--Medium

```
class Solution {
    public int integerBreak(int n) {
        //通过数学方法可证明，几何平均不等式，求导
//         最优： 33 。把数字 nn 可能拆为多个因子 33 ，余数可能为 0,1,20,1,2 三种情况。
//         次优： 22 。若余数为 22 ；则保留，不再拆为 1+11+1 。
//         最差： 11 。若余数为 11 ；则应把一份 3 + 13+1 替换为 2 + 22+2，因为 2 \times 2 > 3 \times 12×2>3×1。
        if(n <= 3) return n - 1;
        int a = n / 3, b = n % 3;
        if(b == 0) return (int)Math.pow(3, a);
```

```
        if(b == 1) return (int)Math.pow(3, a - 1) * 4;
        return (int)Math.pow(3, a) * 2;
    }
}

class Solution {
    public int integerBreak(int n) {
         //数组默认值为0
        int[] dp = new int[n + 1];
        dp[1] = 1;
        for (int i = 2; i < n + 1; i++) {
            for (int j = 1; j < i; j++) {
                //dp方程
                dp[i] = Math.max(dp[i], Math.max(j * dp[i - j], j * (i -
j)));
            }
        }
        return dp[n];
    }
}
```

2. 按平方数来分割整数--279--Medium

```
class Solution {
    // dp[i]：表示完全平方数和为i的 最小个数
    // 初始状态dp[i]均取最大值i，即 1+1+...+1，i个1；dp[0] = 0
    // dp[i] = min(dp[i], dp[i-j*j]+1)，其中，j是平方数，j=1~k,其中k*k要保证
<= i
    // 意思就是：完全平方数和为i的 最小个数 等于 当前完全平方数和为i的 最大个数
    //   与 （完全平方数和为 i - j * j 的 最小个数 + 完全平方数和为 j * j的 最小个
数）
    //   可以看到 dp[j*j] 是等于1的
    public int numSquares(int n) {
        int[] dp = new int[n + 1]; // 默认初始化值都为0
        for (int i = 1; i <= n; i++) {
            dp[i] = i; // 最坏的情况就是每次+1
            for (int j = 1; i - j * j >= 0; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1); // 动态转移方程
            }
        }
        return dp[n];
    }
}
```

3. 分割整数构成字母字符串--91--Medium

```java
class Solution {
    public int numDecodings(String s) {
     if (s == null || s.length() == 0) {
            return 0;
        }
        int n = s.length();
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = s.charAt(0) == '0' ? 0 : 1;
        for (int i = 2; i <= n; i++) {
            int one = Integer.valueOf(s.substring(i - 1, i));
            if (one != 0) {
                dp[i] += dp[i - 1];
            }
            if (s.charAt(i - 2) == '0') {
                continue;
            }
            int two = Integer.valueOf(s.substring(i - 2, i));
            if (two <= 26) {
                dp[i] += dp[i - 2];
            }
        }
        return dp[n];
    }
}
```

最长递增子序列

1. 最长递增子序列--300--Medium

```

```

2. 一组整数对能够构成的最长链--646--Medium

```

```

3. 最长摆动子序列--366--Medium

```

```

最长公共子序列

1. 最长公共子序列--1143--Medium

0-1背包

1. 划分数组为和相等的两部分--416--Medium

2. 改变一组数的正负号使得它们的和为一给定数--494--Medium

3. 01 字符构成最多的字符串--474--Medium

4. 找零钱的最少硬币数--322--Medium

5. 找零钱的硬币数组合--518--Medium

6. 字符串按单词列表分割--139--Medium

7. 组合总和--377--Medium

股票交易

1. 需要冷却期的股票交易--309--Medium

2. 需要交易费用的股票交易--714--Medium

3. 只能进行两次的股票交易--123--Hard

4. 只能进行 k 次的股票交易--188--Medium

字符串编辑

1. 删除两个字符串的字符使它们相等--583--Medium

2. 编辑距离--72--Hard

3. 复制粘贴字符--650--Medium

数学

素数分解

每一个数都可以分解成素数的乘积，例如 $84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * ...$

整除

令 $x = 2^{m0} * 3^{m1} * 5^{m2} * 7^{m3} * 11^{m4} * ...$

令 $y = 2^{n0} * 3^{n1} * 5^{n2} * 7^{n3} * 11^{n4} * ...$

如果 x 整除 y（y mod x == 0），则对于所有 i，mi <= ni。

最大公约数最小公倍数

x 和 y 的最大公约数为：$gcd(x,y) = 2^{min(m0,n0)} * 3^{min(m1,n1)} * 5^{min(m2,n2)} * ...$

x 和 y 的最小公倍数为：$lcm(x,y) = 2^{max(m0,n0)} * 3^{max(m1,n1)} * 5^{max(m2,n2)} * ...$

1. 生成素数序列--204--Easy

```java
class Solution {
    public int countPrimes(int n) {
      //不是素数为true，默认为false
        boolean[] notPrimes = new boolean[n + 1];
        int count = 0;
        for (int i = 2; i < n; i++) {
            if (notPrimes[i]) {
                continue;
            }
            count++;
            // 从 i * i 开始，因为如果 k < i，那么 k * i 在之前就已经被去除过了
            for (long j = (long) (i) * i; j < n; j += i) {
                notPrimes[(int) j] = true;
            }
        }
        return count;
    }
}
```

2. 最大公约数

```
//最大公约数
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
//最小公倍数为两数的乘积除以最大公约数。
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

3. 使用位操作和减法求解最大公约数--编程之美--2.7

   对于 a 和 b 的最大公约数 f(a, b)，有：

   - 如果 a 和 b 均为偶数，f(a, b) = 2*f(a/2, b/2);

- 如果 a 是偶数 b 是奇数，f(a, b) = f(a/2, b);
  - 如果 b 是偶数 a 是奇数，f(a, b) = f(a, b/2);
  - 如果 a 和 b 均为奇数，f(a, b) = f(b, a-b);

  乘 2 和除 2 都可以转换为移位操作。

```
public int gcd(int a, int b) {
    if (a < b) {
        return gcd(b, a);
    }
    if (b == 0) {
        return a;
    }
    boolean isAEven = isEven(a), isBEven = isEven(b);
    if (isAEven && isBEven) {
        return 2 * gcd(a >> 1, b >> 1);
    } else if (isAEven && !isBEven) {
        return gcd(a >> 1, b);
    } else if (!isAEven && isBEven) {
        return gcd(a, b >> 1);
    } else {
        return gcd(b, a - b);
    }
}
```

进制转换

1. 7 进制--504--Easy

```
class Solution {
    public String convertToBase7(int num) {
        if (num == 0) {
```

```
            return "0";
        }
        StringBuilder sb = new StringBuilder();
        //负数先转为整数求解
        boolean isNegative = num < 0;
        if (isNegative) {
            num = -num;
        }
        while (num > 0) {
            sb.append(num % 7);
            num /= 7;
        }
        String ret = sb.reverse().toString();
        return isNegative ? "-" + ret : ret;
    }
//    public String convertToBase7(int num) {
//        return Integer.toString(num, 7);
//    }
}
```

2. 16 进制--405--Easy

```
class Solution {
    public String toHex(int num) {
         //位运算
        char[] map = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'a', 'b', 'c', 'd', 'e', 'f'};
        if (num == 0) return "0";
        StringBuilder sb = new StringBuilder();
        while (num != 0) {
            sb.append(map[num & 0b1111]);
            num >>>= 4; // 因为考虑的是补码形式，因此符号位就不能有特殊的意义，需要
使用无符号右移，左边填 0
        }
        return sb.reverse().toString();
    }
}
```

3. 26 进制--168--Easy

```
class Solution {
    public String convertToTitle(int n) {
     StringBuilder str = new StringBuilder();
        //转字符前插法
        while (n != 0) {
            n--;
            str.insert(0, (char) (n % 26 + 65));
            n /= 26;
        }
        return str.toString();
    }
}
```

阶乘

1. 统计阶乘尾部有多少个 0--172--Easy

```
class Solution {
    //算一下乘法因子里有多少个5
 public int trailingZeroes(int n) {
        int count = 0;
        while(n >= 5) {
            count += n / 5;
            n /= 5;
        }
        return count;
    }
}
```

字符串加法减法

1. 二进制加法--67--Easy

```
class Solution {
    public String addBinary(String a, String b) {
         int n = a.length(), m = b.length();
        //长的放在前面
        if (n < m)
            return addBinary(b, a);
        int L = Math.max(n, m);
        StringBuilder sb = new StringBuilder();
        int carry = 0, j = m - 1;
        //计算过程
        for(int i = L - 1; i >= 0; --i) {
            if (a.charAt(i) == '1')
```

```
                ++carry;
            if (j > -1 && b.charAt(j--) == '1')
                ++carry;
            if (carry % 2 == 1)
                sb.append('1');
            else
                sb.append('0');
            carry /= 2;
        }
        //若最后多出一位carry，加入字符串
        if (carry == 1)
            sb.append('1');
        //反转字符串得到答案
        sb.reverse();
        return sb.toString();
    }
}
```

2. 字符串加法--415--Easy

```
//字符串加法、链表加法以及二进制加法之类的都可以这么写
class Solution {
 public String addStrings(String num1, String num2) {
       StringBuilder sb = new StringBuilder();
        int carry = 0, i = num1.length() - 1, j = num2.length() - 1;
        //从末位开始相加，当两个字符串数下标>=0时，或者进位不为0时，
        //计算当前位相加结果存入StringBuilder
        while(i >= 0 || j >= 0 || carry != 0){
            if(i >= 0)
                carry += num1.charAt(i--) - '0';
            if(j >= 0)
                carry += num2.charAt(j--) - '0';
            sb.append(carry%10);
            carry /= 10;
        }
        return sb.reverse().toString();
    }
}
```

相遇问题

1. 改变数组元素使所有的数组元素都相等--462--Medium

```
class Solution {
     public int minMoves2(int[] nums) {
```

```
        //转成中位数的步数最少，可用数学证明
        Arrays.sort(nums);
        int move = 0;
        int l = 0, h = nums.length - 1;
        while (l <= h) {
            move += nums[h] - nums[l];
            l++;
            h--;
        }
        return move;
    }
}
```

多数投票问题

1. 数组中出现次数多于 n / 2 的元素--169--Easy

```
class Solution {
    public int majorityElement(int[] nums) {
     //利用哈希表求解
        HashMap<Integer,Integer> hashMap = new HashMap<>();
        int count = 0;
        int len = nums.length;
        int ans = nums[0];
        for (int i = 0; i < nums.length; i++) {
            if (!hashMap.containsKey(nums[i])){
                hashMap.put(nums[i],1);
            }else {
                count = hashMap.get(nums[i]);
                hashMap.put(nums[i],count+1);
            }
            if (hashMap.get(nums[i]) > len/2)
                ans = nums[i];
        }
        return ans;
    }
}
```

其它

1. 平方数--367--Easy

```
class Solution {
    public boolean isPerfectSquare(int num) {
        //牛顿迭代法
```

```
        long r = num;
        while (r * r > num) {
            r = (r + num/r) /2;
        }
        return r * r == num;
    }

    public boolean isPerfectSquare(int num) {
        int subNum = 1;
        while (num > 0) {
            num -= subNum;
            subNum += 2;
        }
        return num == 0;
    }
}
```

2. 3 的 n 次方--326--Easy

```
class Solution {
    public boolean isPowerOfThree(int n) {
        //通过查看相关解析，发现了这个解法，用到了数论的知识，
        //3的幂次的质因子只有3，而所给出的n如果也是3的幂次，
        //故而题目中所给整数范围内最大的3的幂次的因子只能是3的幂次，
        //1162261467是3的19次幂，是整数范围内最大的3的幂次
        return n > 0 && (1162261467 % n == 0);
    }
}
```

3. 乘积数组--238--Medium

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n=nums.length;
        int left=1,right=1;     //left：从左边累乘，right：从右边累乘
        int[] ans = new int[n];
        Arrays.fill(ans, 1);
        for(int i=0;i<n;++i){
            //最终每个元素其左右乘积进行相乘得出结果
            ans[i] *= left;        //乘以其左边的乘积
            left *= nums[i];
            ans[n - 1 - i] *= right;  //乘以其右边的乘积
            right *= nums[n - 1 - i];
        }
        return ans;
    }
```

```
}
```

4. 找出数组中的乘积最大的三个数--628--Easy

```
class Solution {
    public int maximumProduct(int[] nums) {
     int ans = Integer.MIN_VALUE;
        int len = nums.length;
        Arrays.sort(nums);
        //最大的三位数相乘
        ans = Math.max(ans, nums[len-1] * nums[len-2] * nums[len-3]);
        //如果有最小两个的负数的绝对值更大，那就负负得正再乘最大的数
        ans = Math.max(ans, nums[0] * nums[1] * nums[len-1]);
        return ans;
    }
}
```