PeterHo的LeetCode之旅

> 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--678--Easy

```

```

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

```

```

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

```

```

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

```

```

10. 寻找二叉查找树中出现次数最多的值--501--Easy

```

```

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

```

```

2. 实现一个 Trie，用来求前缀和--677--Medium

```

```

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;
    }
}
```

```java
//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

```java
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

```
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

```
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

```
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;
```

```
    }
}


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. 字符串循环移位包含--编程之美3.1

```

```

2. 字符串循环移位--编程之美2.17

3. 字符串中单词的翻转--程序员代码面试指南

4. 两个字符串包含的字符是否完全相同--242--Easy

5. 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

6. 字符串同构--205--Easy

7. 回文子字符串个数--647--Medium

8. 判断一个整数是否是回文数--9--Easy

9. 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

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

```java
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

```java
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

```java
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

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

0. 原理

   基本原理

   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

2. 数组中唯一一个不重复的元素--136--Easy

3. 找出数组中缺失的那个数--268--Easy

4. 数组中不重复的两个元素--260--Easy

5. 翻转一个数的比特位--190--Easy

6. 不用额外变量交换两个整数--程序员面试指南--P317

7. 判断一个数是不是 2 的 n 次方--231--Easy

8. 判断一个数是不是 4 的 n 次方--342--Easy

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

10. 求一个数的补码--476--Easy

11. 实现整数的加法--461--Easy

12. 字符串数组最大乘积--371--Easy

13. 统计从 0 ~ n 每个数的二进制表示中 1 的个数--318--Medium

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。
```

```java
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

```
```

排序

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

1. 分配饼干--455--Easy

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

6. 买卖股票的最大收益 II--122--Easy

7. 种植花朵--605--Easy

8. 判断是否为子序列--329--Medium

9. 修改一个数成为非递减数组--665--Easy

10. 子数组最大的和--53--Easy

11. 分隔字符串使同种字符出现在一起--763--Medium

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

```

```

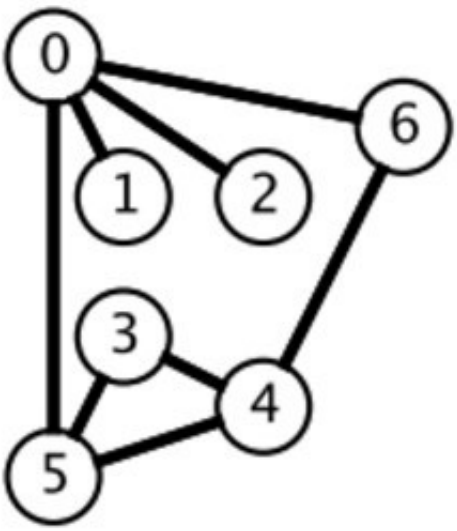
分治

1. 给表达式加括号--241--Medium

2. 不同的二叉搜索树--95--Medium

搜索

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

```

```

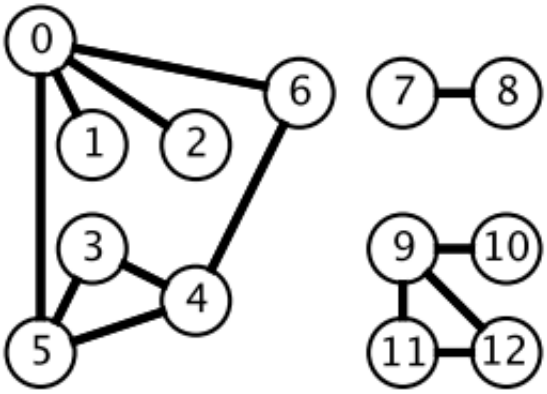
2. 组成整数的最小平方数数量--279--Medium

```

```

3. 最短单词路径--127--Medium

```

```

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

```
```

2. 矩阵中的连通分量数目--200--Medium

```
```

3. 好友关系的连通分量数目--547--Medium

```
```

4. 填充封闭区域--130--Medium

```
```

5. 能到达的太平洋和大西洋的区域--417--Medium

```
```

Backtracking

> Backtracking（回溯）属于 DFS。
>
> - 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

```

```

2. IP 地址划分--93--Medium

```

```

3. 在矩阵中寻找字符串--79--Medium

```

```

4. 输出二叉树中所有从根到叶子的路径--257--Easy

```

```

5. 排列--46--Medium

```

```

6. 含有相同元素求排列--47--Medium

```

```

7. 组合--77--Medium

```

```

8. 组合求和--39--Medium

```

```

9. 含有相同元素的组合求和--40--Medium

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

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：

$$dp[i] = (i-1) * dp[i-2] + (i-1) * dp[i-1]$$

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：

$$dp[i] = dp[i-1] + dp[i-3]$$

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

```

```

2. 数组中等差递增子区间的个数--413--Medium

```

```

分割整数

1. 分割整数的最大乘积--343--Medium

```

```

2. 按平方数来分割整数--279--Medium

```

```

3. 分割整数构成字母字符串--91--Medium

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

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

2. 16 进制--405--Easy

3. 26 进制--168--Easy

阶乘

1. 统计阶乘尾部有多少个 0--172--Easy

字符串加法减法

1. 二进制加法--67--Easy

2. 字符串加法--415--Easy

相遇问题

1. 改变数组元素使所有的数组元素都相等--462--Medium

多数投票问题

1. 数组中出现次数多于 n / 2 的元素--169--Easy

其它

1. 平方数--367--Easy

2. 3 的 n 次方--326--Easy

3. 乘积数组--238--Medium

4. 找出数组中的乘积最大的三个数--628--Easy
