剑指offer

1 赋值运算函数

1. 描述

为自定义类添加赋值运算符函数。

2. 思路

该题使用C++语言实现更加合适，因为Java不支持用户自定义操作符重载（尴尬了~）。不过流程以及细节都是一样的。下面是在实现时需要注意的四个细节：

（1）返回类型为该类型的引用类型。

（2）传入参数是不变的，需要使用常量引用，在Java即用 final 进行修饰。

（3）释放本实例内存（Java自带gc，故不需要主动释放内存，可以忽略）

（4）首先判断传入实例与本实例是否为同一个（在Java中需要分为引用相等、引用不等但值相等来判断），是则直接返回本实例引用，否则赋值再返回本实例引用。

3. 代码

上面也说了Java不支持用户自定义操作符重载，所以只能依赖类中方法实现赋值运算。

以下是样例实现（有些缺陷，不能达到连续赋值的效果，即如果是a=b=c，该实现是不能让b拿到c的值）：

```
// 一个缺点：此赋值从左到右进行，a=b=c等价于a=c，b不会被赋值；
public class MyAssignment {
    private String data;

    public MyAssignment(String data) {
        this.data = data;
    }

    public MyAssignment assign(final MyAssignment another) {
        if (this == another || this.data.equals(another.data))
            return this;
        else {
            this.data = another.data;
            return this;
        }
    }

    @Override
    public String toString() {
        return "MyString{" + "data='" + data + '\'' + '}';
    }
}
```

2 单例（Singleton）模式

1. 描述

设计一个类，只能生成该类的一个实例

2. 思路

单例类特点：

- 单例类只能有一个实例。
- 单例类必须自己创建自己的唯一实例。
- 单例类必须给所有其他对象提供这一实例。

单例模式实现分为懒汉式创建（需要才去创建对象）与饿汉式创建（创建类的实例时就去创建对象），同时需要考虑线程安全问题（比较重要）

- 懒汉式：线程不一定安全，可以通过一定方法（加锁，同步，线程，枚举等）实现线程安全
- 饿汉式：线程安全，分两种：属性实例化对象，静态代码块实例化对象

3. 代码

```
// 饿汉模式（通过属性创建）：线程安全，耗费资源。
public class HugerSingletonByProperty {
    private static final HugerSingletonByProperty uniqueInstance = new HugerSingletonByProperty();

    private HugerSingletonByProperty() {
    }

    public static HugerSingletonByProperty getInstance() {
        return uniqueInstance;
    }
}

// 懒汉式：非线程安全
public class LazySingletonByNormal {
    private static LazySingletonByNormal uniqueInstance;

    public static LazySingletonByNormal getInstance() {
        if (null == uniqueInstance) {
            uniqueInstance = new LazySingletonByNormal();
        }
        return uniqueInstance;
    }

    private LazySingletonByNormal() {
    }
}

// 懒汉式：线程安全(双重检验所)，可能还存在反射攻击或者反序列化攻击
// 为防止指令重排导致线程不安全，需要加上 volatile 关键字， 禁止指令进行重排序优化，同时保证变量的可见性
public class LazySingletonByDoubleCheck {
    private static volatile LazySingletonByDoubleCheck uniqueInstance;

    public static LazySingletonByDoubleCheck getInstance() {
        if (null == uniqueInstance) {
            synchronized (LazySingletonByDoubleCheck.class) {
                if (null == uniqueInstance) {
                    uniqueInstance = new LazySingletonByDoubleCheck();
```

```
                }
            }
        }
        return uniqueInstance;
    }

    private LazySingletonByDoubleCheck() {
    }
}

// 懒汉式：线程安全(静态内部类)，可能还存在反射攻击或者反序列化攻击
public class LazySingletonByStaticInnerClass {

    private static class SingletonHodler {
        private static final LazySingletonByStaticInnerClass uniqueInstance
= new LazySingletonByStaticInnerClass();
    }

    public static LazySingletonByStaticInnerClass getInstance() {
        return SingletonHodler.uniqueInstance;
    }

    private LazySingletonByStaticInnerClass() {

    }
}

// 懒汉式：枚举实现（可防止反射、反序列化）
public enum Singleton {

    INSTANCE;

    public void doSomething() {
        System.out.println("doSomething");
    }
}
```

3 数组中重复的数字

1. 描述

找出数组中重复的数字（注意需要考虑是否全部找出还是随意一个）

- 在一个长度为n的数组中，所有数字都在0到n-1的范围内，数组中的某些数字是重复的，但不知道具体的数字，也不知道重复了几次。

示例：

```
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

2. 思路

解法1：暴力求解

使用双重循环判断是否有重复元素，比较简单。

时间与空间复杂度：$O(n^2)$，$O(1)$

解法2：使用快速排序

对数组进行排序，比较数组相邻元素是否一样确定重复的数字

时间与空间复杂度：$O(nlogn)$，$O(1)$

解法3：借助哈希表

由于数组元素大小在0~n范围内，可以借助数组实现，给先与数组元素相等的key对应的value+1，如果value值大于1，则是重复元素

时间与空间复杂度：$O(n)$，$O(n)$

解法4：根据数字特点交换元素重新排序

数组元素的 索引 和 值 是 多对一 的关系。因此可遍历数组并通过交换操作，使元素的 索引 与 值 进行对应起来。

遍历数组，如果当前索引与其值不同（$nums[i]! = i$），分为两种情况：

- 第一种是$num[num[i]] == num[i]$，即代表索引 $nums[i]$ 处和索引 i 处的元素值都为 $nums[i]$，即找到一组重复值，返回此值 $nums[i]$，
- 第二种情况是$num[num[i]]! = num[i]$，交换$num[num[i]]$与$num[i]$的值，将此数字交换至对应索引位置。

时间与空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class FindRepeatNumberInArray {
    // 解法1：暴力求解
    public int findRepeatNumber(int[] nums) {
        if (nums == null && nums.length <= 0) {
            return -1;
        }
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[i] == nums[j]) {
                    return nums[i];
                }
            }
        }
        return -1;
    }

    // 解法2：排序
    public int findRepeatNumber1(int[] nums) {
        if (nums == null && nums.length <= 0) {
            return -1;
        }
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == nums[i + 1]) {
                return nums[i];
            }
        }
        return -1;
```

```
    }

    // 解法3：哈希
    public int findRepeatNumber2(int[] nums) {
        if (nums == null && nums.length <= 0) {
            return -1;
        }
        int[] hash = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            if (hash[nums[i]] >= 1) {
                return nums[i];
            } else {
                hash[nums[i]]++;
            }
        }
        return -1;
    }

    // 解法4：数字特点
    public int findRepeatNumber3(int[] nums) {
        if (nums == null && nums.length == 0) {
            return -1;
        }
        for (int i = 0; i < nums.length; i++) {
            while (nums[i] != i) {
                if (nums[nums[i]] == nums[i]) {
                    return nums[i];
                }
                int temp = nums[i];
                nums[i] = nums[temp];
                nums[temp] = temp;
            }
        }
        return -1;
    }
}
```

4 二维数组中的查找

1. 描述

在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例：

```
现有矩阵 matrix 如下：
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
给定 target = 5，返回 true。
给定 target = 20，返回 false。
```

2. 思路

有序数组，可以采用二分法解决（可以是变体）。

选取右上角或者左下角的元素作比较，如果目标值小与该值，则可以排除一列或者一行，否则排除一行或者一列，具体过程如下：

- 若数组为空，返回 false
- 初始化行下标为 0，列下标为二维数组的列数减 1
- 重复下列步骤，直到行下标或列下标超出边界
  - 获得当前下标位置的元素 num
  - 如果 num 和 target 相等，返回 true
  - 如果 num 大于 target，列下标减 1
  - 如果 num 小于 target，行下标加 1

循环体执行完毕仍未找到元素等于 target ，说明不存在这样的元素，返回 false

时间与空间复杂度：$O(m+n)$，$O(1)$，访问到的下标的行最多增加 n 次，列最多减少 m 次，因此循环体最多执行 n + m 次。

3. 代码

```java
public class FindNumberIn2DArray {
    // 二分法
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return false;
        }
        if (matrix[0][0] > target || matrix[matrix.length - 1]
[matrix[0].length - 1] < target) {
            return false;
        }
        int row = 0;
        int column = matrix[0].length - 1;
        while (row < matrix.length && column >= 0) {
            if (matrix[row][column] == target) {
                return true;
            } else if (matrix[row][column] > target) {
                column--;
            } else {
                row++;
            }
        }
        return false;
    }
```

```
}
```

5 替换空格

1. 描述

实现一个函数，把字符串中的每个空格都替换成“%20”，

示例：

```
输入：s = "We are happy."
输出："We%20are%20happy."
```

2. 思路

注意：java中String类型是不可变类型。

对原数组从后往前遍历，遇到非空格则往后移动或者直接在新的字符串上赋值，否则进行替换，或者赋值空格。

同样合并两个有序数组也可以这样来做。

时间与空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class ReplaceSpace {
    // 实现1
    public String replaceSpace(String s) {
        char[] array = s.toCharArray();
        int length = array.length;
        int spaceCount = 0;
        for (int i = 0; i < length; i++) {
            if (array[i] == ' ') {
                spaceCount++;
            }
        }
        char[] replaceArray = new char[length -spaceCount+ spaceCount * 3];
        int index = replaceArray.length - 1;
        for (int i = length - 1; i >= 0; i--) {
            if (array[i] == ' ') {
                replaceArray[index--] = '0';
                replaceArray[index--] = '2';
                replaceArray[index--] = '%';
            } else {
                replaceArray[index--] = array[i];
            }
        }

        return String.valueOf(replaceArray);
    }

    // 实现2
    public String replaceSpace1(String s){
        StringBuilder sb = new StringBuilder();
        for(int i = 0 ; i < s.length(); i++){
```

```
            char c = s.charAt(i);
            if(c == ' ') sb.append("%20");
            else sb.append(c);
        }
        return sb.toString();
    }
}
```

6 从尾到头打印链表

1. 描述

输入一个链表的头节点，从尾到头反过来打印每个节点的值，

示例：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

2. 思路

解法1：递归

由于一直是访问元素操作，可以采用递归实现，具体实现就是不断调用递归函数，然后访问节点元素。

时间与空间复杂度：$O(n)$，$O(n)$

解法2：栈

通过栈的形式存储并进行输出（先进后出）。

时间与空间复杂度：$O(n)$，$O(n)$

3. 代码

```
import java.util.Stack;

public class PrintReverseLink {
    // 解法1：递归
    public int[] reversePrint(ListNode head) {
        if (head == null) {
            return new int[0];
        }
        List<Integer> list = new ArrayList<>();
        reversePrintCore(head, list);
        int length = list.size();
        int[] result = new int[length];
        for (int i = 0; i < length; i++) {
            result[i] = list.get(i);
        }
        return result;
    }

    public void reversePrintCore(ListNode node, List<Integer> list) {
        if (node == null) {
            return;
        }
```

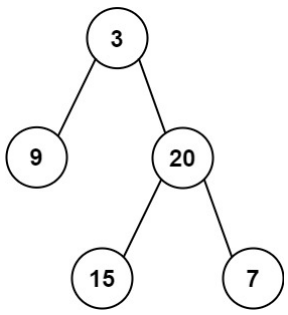
```
        reversePrintCore(node.next, list);
        list.add(node.val);
    }

    // 解法2：使用栈实现
    public int[] reversePrint2(ListNode head) {
        if (head == null) {
            return new int[0];
        }
        ListNode node = head;
        Stack<Integer> stack = new Stack<>();
        while (node != null) {
            stack.add(node.val);
            node = node.next;
        }
        int length = stack.size();
        int[] result = new int[length];
        for (int i = 0; i < length; i++) {
            result[i] = stack.pop();
        }
        return result;
    }
}
```

7 重建二叉树

1. 描述

根据二叉树的前序遍历和中序遍历，重建该二叉树。假设不包含重复数字，如前序序列{1，2，4，7，3，5，6，8}和中序序列{4，7，2，1，5，3，8，6}

示例：




```
输入：preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
输出：[3,9,20,null,null,15,7]
```

2. 思路

基础：

树有三种遍历方法：前序遍历、中序遍历、后序遍历。

- 前序遍历：根节点——左节点——右节点
- 中序遍历：左节点——根节点——右节点
- 后序遍历：左节点——右节点——根节点

此外，还有宽度/层次优先遍历，逐层访问树的节点（从第一层开始），每层都是从左到右。

前序+中序，后序+中序可以完成重建，而前序+后序无法完成

只要我们在中序遍历中定位到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有左右括号进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地对构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

时间与空间复杂度：$O(n)$，$O(n)$

3. 代码

```
public class ConstructBinaryTree {
    // 实现
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || inorder == null || preorder.length == 0 ||
inorder.length == 0 || preorder.length != inorder.length) {
            return null;
        }
        // 递归调用
        return constructCore(preorder, 0, inorder, 0, preorder.length);
    }

    public TreeNode constructCore(int[] preorder, int preStart, int[]
inorder, int inStart, int length) {
        // 左右子树集合为空，构建完成直接返回
        if (length == 0) {
            return null;
        }
        int index = -1;
        // 查找根节点在中序遍历中的索引位置
        for (int i = inStart; i < inStart + length; i++) {
            if (preorder[preStart] == inorder[i]) {
                index = i;
                break;
            }
        }
        // 左子树的节点数量
        int preorder_length = index - inStart;
        // 构建根节点
        TreeNode root = new TreeNode(preorder[preStart]);

        root.left = constructCore(preorder, preStart + 1, inorder, inStart,
preorder_length);
        root.right = constructCore(preorder, preStart + preorder_length + 1,
inorder, index + 1,
                length - preorder_length - 1);
        return root;
    }
}
```

8 二叉树的下一个节点

1. 描述

给定二叉树和其中一个节点（唯一参数），找到中序遍历序列的下一个节点。树中的节点除了有左右孩子指针，还有一个指向父节点的指针。

示例：

```
// 测试用例使用的树
//             1
//          //   \\
//         2       3
//      //  \\   // \\
//      4    5   6   7
//         // \\
//         8   9

节点4的下一个节点是2
```

2. 思路

利用二叉树中序遍历的顺序分情况讨论：

- 如果输入的当前节点有右孩子，则它的下一个节点即为该右孩子为根节点的子树的最左边的节点
- 如果输入的当前节点没有右孩子，就需要判断其与自身父节点的关系：
  - 如果当前节点没有父节点，那所求的下一个节点不存在，返回null.
  - 如果输入节点是他父节点的左孩子，那他的父节点就是所求的下一个节点,比如4->2
  - 如果输入节点是他父节点的右孩子，那就需要将输入节点的父节点作为新的当前节点，返回到（2）,判断新的当前节点与他自身父节点的关系, 比如5->1

时间与空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class NextNodeInBinaryTrees {
    public TreeNodeWithParent getNext(TreeNodeWithParent node) {
        if (node == null) {
            return null;
        }
        // 当前节点有右孩子
        if (node.right != null) {
            TreeNodeWithParent next_node = node.right;
            while (next_node.left != null) {
                next_node = next_node.left;
            }
            return next_node;
        }
        // 没有右孩子，分为几种情况
        while (true) {
            // 没有父节点
            if (node.father == null) {
                return null;
            }
            // 当前节点是父节点的左节点
            if (node.father.left == node) {
                return node.father;
            }
            // 当前节点是父节点的右节点
            if (node.father.right == node) {
                node = node.father;
```

```
            }
        }
    }
}
```

9 用两个栈实现队列

1. 描述

用两个栈，实现队列的从队尾插入元素offer()和从队头抛出元素poll()

示例：

```
输入：
["CQueue","appendTail","deleteHead","deleteHead"]
[[],[3],[],[]]
输出：[null,null,3,-1]
```

2. 思路

具体实现过程：

- 添加元素：使用栈1添加元素。
- 删除元素：将栈2元素弹出，如果连续删除，则栈2连续弹出直到栈空，接着需要判断栈1是否有元素，如果有则从栈1弹出压入栈2。

时间与空间复杂度：$O(1)$，$O(n)$

3. 代码

```java
import java.util.Stack;

public class CQueue {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

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

    public void appendTail(int value) {
        stack1.push(value);
    }

    public int deleteHead() {
        if (!stack2.isEmpty()) {
            return stack2.pop();
        } else {
            if (!stack1.isEmpty()) {
                while (!stack1.isEmpty()) {
                    stack2.push(stack1.pop());
                }
                return stack2.pop();
            } else {
                return -1;
```

```
            }
        }
    }
}
```

10.1 斐波那契数列

1. 描述

求斐波那契数列的第n项的值。

斐波那契数列的定义如下：

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
```

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

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

示例：

```
输入：n = 2
输出：1
```

2. 思路

解法1：递归

自上往下，会出现大量重复计算，效率不好（如计算f(6)与f(7)时，最后都会重复计算f(6)、f(5)等）

时间与空间复杂度：$O(n^2)$，$O(n)$

解法2：迭代

自下往上，可以使用循环实现，因此需要两个临时变量记录前面计算过的结果。

时间与空间复杂度：$O(n)$，$O(1)$

3. 代码

```
public class Fibonacci {
    // 解法1：递归解法
    public int fibonacciRecursionly(int n) {
        if (n < 0) {
            return -1;
        }
        if (n == 0 || n == 1) {
            return n;
        } else {
            return (fibonacciRecursionly(n - 1) + fibonacciRecursionly(n -
2)) % 1000000007;
        }
    }

    // 解法2：循环解法
    public int fibonacciCyclely(int n) {
```

```
        int fibonacci = 0;
        if (n < 0) {
            return -1;
        }
        if (n == 0 || n == 1) {
            return n;
        } else {
            int n_2 = 0;
            int n_1 = 1;
            for (int i = 2; i <= n; i++) {
                int temp = n_2 % 1000000007;
                n_2 = n_1 % 1000000007;
                n_1 = (temp + n_1) % 1000000007;
            }
            fibonacci = n_1;
        }
        return fibonacci;
    }
}
```

10.2 青蛙跳台阶问题

1. 描述

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

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

示例：

```
输入：n = 2
输出：2
```

2. 思路

斐波那契数列类似题目。

设跳上 n 级台阶有 f(n) 种跳法。在所有跳法中，青蛙的最后一步只有两种情况： 跳上 1 级或 2 级台阶。

- 当为 1 级台阶： 剩 n-1 个台阶，此情况共有 f(n-1) 种跳法；
- 当为 2 级台阶： 剩 n-2 个台阶，此情况共有 f(n-2) 种跳法。
- f(n) 为以上两种情况之和，即 f(n)=f(n-1)+f(n-2)，以上递推性质为斐波那契数列。本题可转化为 求斐波那契数列第 n 项的值

3. 代码

```
public class NumWays {
    public int numways(int n) {
        int fibonacci = 0;
        if (n < 0) {
            return -1;
        }
        if (n == 0 || n == 1) {
            return 1;
```

```
        } else {
            int n_2 = 1;
            int n_1 = 1;
            for (int i = 2; i <= n; i++) {
                int temp = n_2 % 1000000007;
                n_2 = n_1 % 1000000007;
                n_1 = (temp + n_1) % 1000000007;
            }
            fibonacci = n_1;
        }
        return fibonacci;
    }
}
```

11 旋转数组的最小数字

1. 描述

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。

给你一个可能存在 重复 元素值的数组 numbers ，它原来是一个升序排列的数组，并按上述情形进行了一次旋转。请返回旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一次旋转，该数组的最小值为1。

示例：

```
输入：[3,4,5,1,2]
输出：1
```

2. 思路

二分查找。原本有序，进行了旋转，但在一定程度上还是有序的。

通过观察中间点 mid 与左右点（start，end）的关系，缩小范围，从而找到最小值。缩小范围时要注意值相等的情况，尤其是左右中三个值都相等的情况，这

种特殊情况是无法折半缩小范围的，只能逐步缩小，即左右点各缩进一个单位。

- 当数组第一位比最后一位小，即不存在旋转，第一位是最小的
- 当数组第一位比最后一位大时，需要判断与中间值的大小关系
  - 如果第一位小于等于中间值，则在右边，左边索引需要mid+1
  - 否则，最小值在左边，右边索引=mid
- 当数组第一位与最后一位相等时，需要判断与中间值的大小关系，特别是三者相等，需要将两边的索引进行缩进将其变成可判断的情况
  - 如果第一位小于中间值，则在右边，左边索引需要mid+1
  - 如果第一位大于中间值，则在左边，右边索引=mid
  - 如果第一位等于中间值，则左边索引=start+1，右边索引=end-1

时间与空间复杂度：$O(logn)$，$O(1)$

3. 代码

```
public class MinNumberInRotatedArray {
    public int min(int[] numbers) {
        int start = 0;
```

```
        int end = numbers.length - 1;
        if (numbers == null || numbers.length == 0) {
            return -1;
        } else {
            while (start < end) {
                int mid = (start + end) / 2;
                if (numbers[start] < numbers[end]) {        // start < end
                    return numbers[start];
                } else if (numbers[start] > numbers[end]) {  // start > end
                    if (numbers[mid] >= numbers[start])
                        start = mid + 1;
                    else
                        end = mid;
                } else {                                    // start = end
                    if (numbers[start] < numbers[mid])
                        start = mid + 1;
                    else if (numbers[start] > numbers[mid])
                        end = mid;
                    else {
                        start = start + 1;
                        end = end - 1;
                    }
                }
            }

        }
        return numbers[end];
    }
}
```

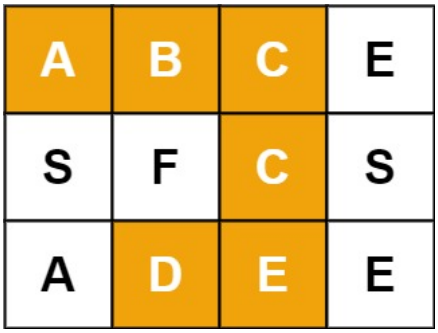
12 矩阵中的路径

1. 描述

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例：




```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
输出：true
```

2. 思路

回溯法适合解决由多个步骤组成的问题，且每个步骤都有多个选项。当在某一步选择了其中一个选项，就进入下一步，然后面临新的选项。如果当前的选项已经确定没有正确答案，就回溯到上一步，在上一步选择另一选项，重复进行。回溯法的求解过程一般都可以用树状结构表示出来。实现代码很适合用递归完成。
至于本题，需要明确递归的约束条件，回溯的终止条件，标记矩阵的标记与清除时机。

- 约束条件，标记矩阵当前位置没有走过且匹配了字符串中的当前字符
- 终止条件：匹配的字符总数超过或者等于给定的字符串长度
- 标记矩阵的标记：初始化标记矩阵，把要走的当前位置标记true，
- 标记矩阵的清除时机：当前位置的下一步返回false，说明当前路径不对，需要清除，防止影响其他路径

时间与空间复杂度：$O(mn3^k)$，$O(mn)$，k是字符串长度

3. 代码

```
public class HasPathWithBacktracking {
    public boolean hasPath(char[][] board, String str) {
        if (board == null || board.length == 0 || str == null ||
str.length() == 0) {
            return false;
        }
        int row = board.length;
        int col = board[0].length;
        boolean[][] flag = new boolean[row][col];

        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (hasPathCore(board, i, j, flag, str, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

    public boolean hasPathCore(char[][] board, int rowIndex, int colIndex,
boolean[][] flag, String str, int strIndex) {
        // 递归结束条件
        if (strIndex >= str.length()) {
            return true;
        }
        // 防止下面的索引越界
        if (rowIndex < 0 || colIndex < 0 || rowIndex >= board.length ||
colIndex >= board[0].length) {
            return false;
        }
        if (!flag[rowIndex][colIndex] && board[rowIndex][colIndex] ==
str.charAt(strIndex)) {
            flag[rowIndex][colIndex] = true;
            boolean result = hasPathCore(board, rowIndex + 1, colIndex,
flag, str, strIndex + 1) ||
                    hasPathCore(board, rowIndex - 1, colIndex, flag, str,
strIndex + 1) ||
                    hasPathCore(board, rowIndex, colIndex + 1, flag, str,
strIndex + 1) ||
```

```
                    hasPathCore(board, rowIndex, colIndex - 1, flag, str,
strIndex + 1);
            if (result == true) {
                return true;
            } else {
                // 重新设置flag值，防止对其他路径有影响
                flag[rowIndex][colIndex] = false;
                return false;
            }
        } else {
            return false;
        }
    }
}
```

13 机器人的运动范围

1. 描述

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子?

示例:

```
输入：m = 2, n = 3, k = 1
输出：3
```

2. 思路

明显回溯法求解。

本题，需要明确递归的约束条件，回溯的终止条件

- 约束条件：当前格子能进入当前格子，对下一个状态（包含上下左右四个选择）进行递归，
- 终止条件：不能进入当前格子（越界或者标记为false或者k值条件），返回count=0
- 标记矩阵的标记：初始化标记矩阵，能进入当前格子的标记true

时间与空间复杂度：$O(mn)$，$O(mn)$

3. 代码

```
public class RobotMoveWithBacktracking {
    public int moveCount(int m, int n, int k) {
        if (m <= 0 || n <= 0 || k < 0) {
            return 0;
        }
        if (k == 0) {
            return 1;
        }
        boolean[][] flag = new boolean[m][n];
        return moveCountCore(flag, 0, 0, m, n, k);
    }

```

```
    public int moveCountCore(boolean[][] flag, int rowIndex, int colIndex,
int m, int n, int k) {
        int count = 0;
        if (canIn(flag, rowIndex, colIndex, m, n, k)) {
            flag[rowIndex][colIndex] = true;
            count = 1 + moveCountCore(flag, rowIndex + 1, colIndex, m, n, k)
+
                    moveCountCore(flag, rowIndex - 1, colIndex, m, n, k) +
                    moveCountCore(flag, rowIndex, colIndex + 1, m, n, k) +
                    moveCountCore(flag, rowIndex, colIndex - 1, m, n, k);
        }
        return count;
    }

    public boolean canIn(boolean[][] flag, int rowIndex, int colIndex, int
m, int n, int k) {
        if (rowIndex < 0 || colIndex < 0 || rowIndex >= m || colIndex >= n)
{
            return false;
        }
        // 注意进入条件：满足大小要求以及没有走过
        if ((rowOrColSum(rowIndex) + rowOrColSum(colIndex) <= k) &&
!flag[rowIndex][colIndex]) {
            return true;
        }
        return false;
    }

    public int rowOrColSum(int n) {
        int sum = 0;
        while (n > 0) {
            sum = sum + n % 10;
            n = n / 10;
        }
        return sum;
    }
}
```

14 剪绳子

1. 描述

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0], k[1] ... k[m-1] 。请问 k[0] * k[1] * ... * k[m-1] 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

示例：

```
输入：2
输出：1
解释：2 = 1 + 1, 1 × 1 = 1
```

2. 思路

解法1：动态规划

我们定义长度为n的绳子剪切后的最大乘积为f(n)，剪了一刀后，f(n)=max(f(i)*f(n-i))

假设n为10，第一刀之后分为了4-6，而6也可能再分成2-4（6的最大是3-3，但过程中还是要比较2-4这种情况的），而上一步4-6中也需要求长度为4的问题的最大值，可见，各个子问题之间是有重叠的，所以可以先计算小问题，存储下每个小问题的结果，逐步往上，求得大问题的最优解。

时间与空间复杂度：$O(n^2)$，$O(n)$

解法2：贪婪法

切分规则：

- 最优： 3 。把绳子尽可能切为多个长度为 33 的片段，留下的最后一段绳子的长度可能为 0,1,2 三种情况。
- 次优： 2 。若最后一段绳子长度为 22 ； 则保留，不再拆为 1+1 。
- 最差： 1 。若最后一段绳子长度为 11 ； 则应把一份 3 + 1 替换为 2 + 2，因为 2×2>3×1。

时间与空间复杂度：$O(1)$，$O(1)$

3. 代码

```java
import java.lang.Math;

public class CuttingRope {
    // 解法1：动态规划解法
    public int cuttingRope(int n) {
        if (n < 2) return 0;
        if (n == 2) return 1;
        if (n == 3) return 2;
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        dp[2] = 2;
        dp[3] = 3;
        int max = 0;
        int temp = 0;
        for (int i = 4; i <= n; i++) {
            max = 0;
            for (int j = 1; j <= i / 2; j++) {
                temp = dp[j] * dp[i - j];
                if (temp > max)
                    max = temp;
            }
            dp[i] = max;
        }
        return dp[n];
    }
    // 解法2：贪婪算法
    public int cuttingRope1(int n) {
        if (n < 2) return 0;
        if (n == 2) return 1;
        if (n == 3) return 2;
        if (n == 4) return 4;
        int dot = -1;
        int count = n / 3;
        int m = n % 3;
        if (m == 1) {
            count = count - 1;
            dot = (int) (Math.pow(3, count)) * 4;
```

```
        } else if (m == 2) {
            dot = (int) (Math.pow(3, count)) * 2;
        } else {
            dot = (int) (Math.pow(3, count));
        }
        return dot;
    }
}
```

15 二进制中1的个数（位运算）

1. 描述

实现一个函数，输入一个int型整数，输出该数字在计算机中二进制表示形式的1的个数。
例如9->1001,输出2；-3->11111111111111111111111111111101,输出31。

示例：

```
输入：n = 11 (控制台输入 00000000000000000000000000001011)
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'
```

2. 思路

考查位运算，此题要注意负数的处理。首先要明确计算机中，数字是以补码的形式存储的。其次，明确位运算符，与&，或|，非~，异或^,<<左移位，>>带符号右移位，>>>无符号右移位(java有此符号，c++没有)

解法1：无符号右移

将n逐步无符号右移，判断最右位是否是1，直到n为0。

解法2：左移

使用一个标记，初始为1，让标记值与原输入数字异或，然后标记值左移。解法一是原数字右移，而解法二是标记左移，从java来看思路类似但换了个角度；但这个思路在C++就很关键，因为C++中没有>>>运算符，只能用解法2。

解法3：减一与

对于二进制数有如下结论：**【把一个整数减去1之后再和原来的整数做位与运算，得到的结果相当于把原整数的二进制表示形式的最右边的1变成0】**。

比如1001，执行一次上述结论，1001&1000=1000，将最右边的1改为了0；再执行一次，1000&0111=0000，第二个1也改成了0。因此能执行几次该结论，就有几个1。

举一反三：

- 使用一条语句判断一个正整数是不是2的整数次方（2的整数次方二进制只有一个1，减去1后再与其做与运算结果为0）

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

- 输入两个整数m，n，计算最少需要改变m的多少位才能得到n
  先对m，n进行异或，再统计异或结果中1的位数

3. 代码

```java
public class NumberOf1InBinary {
    // 解法1：无符号右移
    public int numberOfOne(int n) {
        int count = 0;
        while (n != 0) {
            if ((n & 1) != 0) {
                count++;
            }
            n = n >>> 1;
        }
        return count;
    }

    // 解法2：逻左移
    public int numberOfOne1(int n) {
        int count = 0;
        int flag = 1;
        while (flag != 0) {
            if ((n & flag) != 0) {
                count++;

            }
            flag = flag << 1;
        }
        return count;
    }

    // 解法3：减一与
    public int numberOfOne2(int n) {
        int count = 0;
        while (n != 0) {
            n = n & (n - 1);
            count++;
        }
        return count;
    }
}
```

16 数值的整数次方

1. 描述

实现函数double power（double base，int exponent），求base的exponent次方。不能使用库函数，不需要考虑大数问题。

示例：

```
输入：x = 2.00000, n = 10
输出：1024.00000
```

2. 思路

- 要考虑一些特殊情况，如指数为负、指数为负且底数为0、0的0次方要定义为多少。

- 底数为0的定义。对于一个double类型的数，判断它与另一个数是否相等，不能用“==”，一般都需要一个精度，见下面的equal函数。
- 对于报错的情况，比如0的负数次方，要如何处理。

具体解决有**循环求解**与**递归求解**两种，具体见代码。

循环时间空间复杂度：$O(n)$，$O(1)$

递归时间空间复杂度：$O(logn)$，$O(logn)$

3. 代码

```
public class Power {
    // 解法1：循环计算 注意特殊情况，可以抛出异常
    public double myPow(double x, int n) throws Exception {
        if (equal(x, 0)) {
            if (n == 0) {
                return 1;
            } else if (n < 0) {
                throw new Exception("分母为0");
            }
            return 0;
        }
        if (n < 0) {
            double pow = 1;
            for (int i = 0; i < -n; i++) {
                pow = pow * x;
            }
            return 1 / pow;
        } else {
            double pow = 1;
            for (int i = 0; i < n; i++) {
                pow = pow * x;
            }
            return pow;
        }
    }

    // 解法2：递归计算
    public double myPow1(double x, int n) throws Exception {
        if (equal(x, 0)) {
            if (n == 0) {
                return 1;
            } else if (n < 0) {
                throw new Exception("分母为0");
            }
            return 0;
        }
        if (n < 0) {
            // 注意 n = -max;前面加上负号还是其本身
            return 1.0 / x * powerCoreRecursionly(1.0 / x, -n - 1);
        } else {
            return powerCoreRecursionly(x, n);
        }
    }

    public double powerCoreRecursionly(double x, int n) {
        if (n == 0) {
```

```
            return 1;
        } else {
            // 奇数与偶数
            if ((n & 1) == 0) {
                double temp = powerCoreRecursionly(x, n >> 1);
                return temp * temp;
            } else {
                double temp = powerCoreRecursionly(x, n >> 1);
                return x * temp * temp;
            }
        }
    }

    // 判断double类型是否相等，主要是判断两者之差是否在很小的区间内
    public static boolean equal(double x, double y) {
        return -0.000001 < (x - y) && (x - y) < 0.000001;
    }
}
```

17 打印从1到最大的n位数

1. 描述

比如输入2，打印1, 2 ..... 98, 99；

示例：

```
输入: n = 1
输出: [1,2,3,4,5,6,7,8,9]
```

2. 思路

此题需要考虑大数问题。

解法1：字符串模拟法

使用字符串模拟数字的加法。

时间空间复杂度：$O(10^n)$，$O(n)$

解法2：递归

n位数字的全排列，注意结束与递归条件以及基础返回值

时间空间复杂度：$O(10^n)$，$O(n)$

3. 代码

```
public class Print1ToMaxOfNDigits {
    // 解法1：字符串模拟
    public void printNumbers(int n) {
        if (n <= 0) {
            return;
        }
        int length = 0;
        for (int i = 1; i <= n; i++) {
            length = length * 10 + 9;
```

```
        }
        char[] array = new char[n];
        for (int i = 0; i < n; i++) {
            array[i] = '0';
        }
        while (increase(array)) {
            printOfferBook(array);
        }
    }

    public boolean increase(char[] array) {
        for (int i = array.length - 1; i >= 0; i--) {
            if (array[i] < '9' && array[i] >= '0') {
                array[i] = (char) ((int) array[i] + 1);
                return true;
            } else if (array[i] == '9') {
                array[i] = '0';
            } else {
                return false;
            }
        }
        return false;
    }

    public void printOfferBook(char[] array) {
        int count = 0;
        boolean flag = true;
        for (int i = 0; i < array.length; i++) {
            if (flag) {
                if (array[i] != '0') {
                    flag = false;
                    System.out.print(array[i]);
                }
            } else {
                System.out.print(array[i]);
            }
        }
        if (flag == false)
            System.out.println();
    }

    // 解法2：递归解法
    public void printNumbers1(int n) {
        if (n <= 0) {
            return;
        }
        char[] array = new char[n];
        for (int i = 0; i < n; i++) {
            array[i] = '0';
        }
        for (int i = 0; i < 10; i++) {
            array[0] = (char) ('0' + i);
            printNumbersCore(array, n, 0);
        }
    }

    public void printNumbersCore(char[] array, int n, int index) {
        if (index == n - 1) {
```

```
            printOfferBook(array);
            return;
        }
        for (int i = 0; i < 10; i++) {
            array[index + 1] = (char) ('0' + i);
            printNumbersCore(array, n, index + 1);
        }
    }
}
```

18.1 删除链表的节点

1. 描述

在O（1）时间内删除单链表的节点。

给定链表的头节点和删除节点，定义一个函数在O（1）时间内删除该节点

示例：

```
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

2. 思路

此问题一般来说，需要遍历整个链表来删除节点，为什么要遍历链表？因为要获得删除节点的上一个节点，使得上一个节点的下一个指针指向删除节点的下一个节点。

因为可以有一种方法，把删除节点的下一个节点值赋值该节点，然后删除下一个节点，再把下一个节点的下一个节点指针赋值即可。

时间空间复杂度：$O(1)$，$O(1)$

3. 代码

```
public class DeleteNodeInList {
    // 节点赋值
    public ListNode<Integer> deleteNode(ListNode<Integer> head,
ListNode<Integer> node) {
        if (head == null || node == null) {
            return null;
        } else if (head == node) {
            head = head.next;
            return head;
        } else if (node.next == null) {
            ListNode<Integer> temp = head;
            while (temp.next != node) {
                temp=temp.next;
            }
            temp.next=null;
            return head;
        }else {
            node.val=node.next.val;
            node.next=node.next.next;
```

```
            return head;
        }
    }
}
```

18.2 删除排序链表中重复的节点

1. 描述

比如[1,2,2,3,3,3],删除之后为[1];

示例:

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

2. 思路

由于是已经排序好的链表，需要确定重复区域的长度，删除后还需要将被删去的前与后连接，所以需要三个节点pre, cur, post，cur到

post为重复区域，删除后将pre与post.next连接即可。

此外，要注意被删结点区域处在链表头部的情况，因为需要修改head。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```
public class DeleteRepeatNodeInList {
    public ListNode<Integer> deleteRepeatNode(ListNode<Integer> head) {
        if (head == null) {
            return null;
        } else {
            if (head.next == null) {
                return head;
            }
            ListNode<Integer> pHead = null;
            ListNode<Integer> pCurrent = head;
            ListNode<Integer> pNext = pCurrent.next;
            boolean flag = false;

            // 注意几种情况，分类对比
            while (pNext != null) {
                if (pCurrent.val == pNext.val) {
                    flag = true;
                    pNext = pNext.next;
                } else if (flag && pCurrent.val != pNext.val) {
                    if (pHead == null) {
                        head = pNext;
                    } else {
                        pHead.next = pNext;
                    }
                    pCurrent = pNext;
                    pNext = pNext.next;
                    flag = false;
```

```
                } else {
                    pHead = pCurrent;
                    pCurrent = pNext;
                    pNext = pNext.next;
                }
            }
            if (flag && pHead != null) { // 前面有不等的，但末尾重复
                pHead.next = null;
            } else if (flag && pHead == null) { // 元素全等
                head = null;
            }
            return head;
        }
    }
}
```

19 正则表达式匹配

难

20 表示数值的字符串

1. 描述

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。

数值（按顺序）可以分成以下几个部分：

- 若干空格
- 一个 小数 或者 整数
- （可选）一个 'e' 或 'E' ，后面跟着一个 整数
- 若干空格

小数（按顺序）可以分成以下几个部分：

- （可选）一个符号字符（'+' 或 '-'）
- 下述格式之一：
  - 至少一位数字，后面跟着一个点 '.'
  - 至少一位数字，后面跟着一个点 '.' ，后面再跟着至少一位数字
  - 一个点 '.' ，后面跟着至少一位数字

整数（按顺序）可以分成以下几个部分：

- （可选）一个符号字符（'+' 或 '-'）
- 至少一位数字

部分数值列举如下：

- ["+100", "5e2", "-123", "3.1416", "-1E-16", "0123"]

部分非数值列举如下：

- ["12e", "1a3.14", "1.2.3", "+-5", "12e+5.4"]

示例：

```
输入：s = "0"
输出：true
```

2. 思路

根据题目逐部分判断是否合法即可。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class NumberStrings {
    // 实现1
    public boolean isNumber(String s) {
        // 无效数据
        if (s == null || s.length() == 0) {
            return false;
        }
        // 标记是否遇到数位、小数点、‘e’或'E'
        boolean isNum = false, isDot = false, ise_or_E = false;

        // 删除字符串头尾的空格，转为字符数组，方便遍历判断每个字符
        s = s.trim();
        int length = s.length();
        for (int i = 0; i < length; i++) {
            // 判断当前字符是否为 0~9 的数位
            if (s.charAt(i) >= '0' && s.charAt(i) <= '9') {
                isNum = true;
            } else if (s.charAt(i) == '.') {
                // 遇到小数点
                if (isDot || ise_or_E) {
                    // 小数点之前可以没有整数，但是不能重复出现小数点、或出现‘e’、'E'
                    return false;
                }
                // 标记已经遇到小数点
                isDot = true;
            } else if (s.charAt(i) == 'e' || s.charAt(i) == 'E') {
                // 遇到‘e’或'E'
                if (!isNum || ise_or_E) {
                    // ‘e’或'E'前面必须有整数，且前面不能重复出现‘e’或'E'
                    return false;
                }
                // 标记已经遇到‘e’或'E'
                ise_or_E = true;
                // 重置isNum，因为‘e’或'E'之后也必须接上整数，防止出现 123e或者123e+的非法情况
                isNum = false;
            } else if (s.charAt(i) == '-' || s.charAt(i) == '+') {
                // 正负号只可能出现在第一个位置，或者出现在‘e’或'E'的后面一个位置
                if (i != 0 && s.charAt(i - 1) != 'e' && s.charAt(i - 1) != 'E') {
                    return false;
                }
            } else {
                // 其它情况均为不合法字符
                return false;
            }
```

```
        }
        return isNum;
    }

    int index;
    // 实现2
    public boolean isNumber1(String s) {
        // 无效数据
        if (s == null || s.length() == 0) {
            return false;
        }
        index = 0;
        //字符串开始有空格，可以返回true
        while (index < s.length() && s.charAt(index) == ' ')
            ++index;
        boolean numeric = scanInteger(s);
        // 如果出现'.'，接下来是数字的小数部分
        if (index < s.length() && s.charAt(index) == '.') {
            ++index;
            // 下面一行代码用||的原因：
            // 1. 小数可以没有整数部分，例如.123等于0.123；
            // 2. 小数点后面可以没有数字，例如233.等于233.0；
            // 3. 当然小数点前面和后面可以有数字，例如233.666
            numeric = scanUnsignedInteger(s) || numeric;
        }
        // 如果出现'e'或者'E'，接下来跟着的是数字的指数部分
        if (index < s.length() && (s.charAt(index) == 'e' || s.charAt(index)
== 'E')) {
            ++index;
            // 下面一行代码用&&的原因：
            // 1. 当e或E前面没有数字时，整个字符串不能表示数字，例如.e1、e1；
            // 2. 当e或E后面没有整数时，整个字符串不能表示数字，例如12e、12e+5.4
            numeric = numeric && scanInteger(s);
        }
        //字符串结尾有空格，可以返回true
        while (index < s.length() && s.charAt(index) == ' ')
            ++index;
        // 最后看是否所有部分都符合，如1a3只会检测第一部分是整数然后是a就不会继续检测
了，index!=size，所以返回false
        return numeric && index == s.length();
    }

    public boolean scanInteger(String s) {
        if (index < s.length() && (s.charAt(index) == '+' || s.charAt(index)
== '-'))
            ++index;
        return scanUnsignedInteger(s);
    }

    public boolean scanUnsignedInteger(String s) {
        int before = index;
        while (index < s.length() && index != s.length() && s.charAt(index)
>= '0' && s.charAt(index) <= '9')
            index++;
        return index > before;
    }
}

```

21 调整数组顺序使奇数位于偶数前面

1. 描述

实现一个函数来调整数组中的数字，使得所有奇数位于数组的前半部分，偶数位于后半部分。

示例：

```
输入：nums = [1,2,3,4]
输出：[1,3,2,4]
注：[3,1,2,4] 也是正确的答案之一。
```

2. 思路

解法1：双指针

用双指针从两端向中间扫描，当前面指针指向偶数，后面指针指向奇数时交换，否则指针移动。

由于不断判断前后指针是否满足交换条件，因此可以将判断部分写成函数，这样能够提升代码的可扩展性。之后这一类问题（非负数在前，负数在后；能被3整除的在前，不能的在后；）都只需修改下判断函数即可解决。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class ReorderArray {
    // 解法1：双指针
    public static int[] exchange(int[] nums) {
        if (nums == null || nums.length == 0) {
            return nums;
        }
        int length = nums.length;
        int start = 0;
        int end = length - 1;
        while (start < end) {
            if (flag(nums, start, end) == 0) {
                int temp = nums[start];
                nums[start] = nums[end];
                nums[end] = temp;
                start++;
                end--;
            } else if (flag(nums, start, end) == 1) {
                    start++;
            }else {
                end--;
            }
        }
        return nums;
    }

    public int flag(int[] nums, int start, int end) {
        // 前面是偶数，后面是奇数，返回true调整顺序
        if ((nums[start] & 1) == 0 && (nums[end] & 1) == 1) {
            return 0;
```

```
        }else if((nums[start] & 1) == 0){
            return 1;
        }else {
            return 2;
        }
    }
}
```

22 链表中倒数第k个节点

1. 描述

求链表中倒数第k个节点。链表的尾节点定义为倒数第1个节点。

示例：

```
给定一个链表: 1->2->3->4->5, 和 k = 2.

返回链表 4->5.
```

2. 思路

解法1：双指针

如果先求链表的长度，计算后再从头遍历，需要两遍扫描。更好的方式是使用两个距离为 k 的指针向右移动，

这种方式只需扫描一遍。

需要注意链表是否为空，链表长度是否达到k，k是否是个正数

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```
public class KthNodeFromEnd {
    // 解法1：双指针
    public ListNode getKthFromEnd(ListNode head, int k) {
        if (head == null || k <= 0) {
            return null;
        }
        ListNode move = head;
        ListNode pHead = null;
        int count = 0;
        while (move != null) {
            move = move.next;
            count++;
            if (count == k) {
                pHead = head;
            } else if (count > k) {
                pHead = pHead.next;
            }
        }
        if (count < k) {
            return null;
        } else {
            return pHead;
```

```
        }
    }
}
```

23 链表中环的入口节点

1. 描述

假设一个链表中包含环，请找出入口节点。若没有环则返回null。

2. 思路

解法1：快慢指针

- 双指针第一次相遇： 设两指针 fast，slow 指向链表头部 head，fast 每轮走 2 步，slow 每轮走 1 步；
  - 第一种结果： fast 指针走过链表末端，说明链表无环，直接返回 null；

    TIPS: 若有环，两指针一定会相遇。因为每走 1 轮，fast 与 slow 的间距 +1，fast 终会追上 slow；
  - 第二种结果： 当 fast == slow时， 两指针在环中 第一次相遇 。

  下面分析此时 fast 与 slow走过的 步数关系：

  设链表共有 a+b个节点，其中 链表头部到链表入口 有 a 个节点（不计链表入口节点）， 链表环 有 b 个节点；设两指针分别走了 f ，s 步，则有：
  - fast 走的步数是 slow 步数的 2 倍，即 f = 2s； （解析： fast 每轮走 2 步）
  - fast 比 slow 多走了 n 个环的长度，即 f = s + nb； （ 解析： 双指针都走过 a 步，然后在环内绕圈直到重合，重合时 fast 比 slow 多走环的长度整数倍 ）；
  - 以上两式相减得：f = 2nb，s = nb，即fast和slow 指针分别走了 2n，n 个 环的周长

目前情况分析：

- 如果让指针从链表头部一直向前走并统计步数 k ，那么所有走到链表入口节点时的步数 是：k = a + nb（先走 a 步到入口节点，之后每绕 1 圈环（ b 步）都会再次到入口节点）。
- 而目前，slow 指针走过的步数为 nb 步。因此，我们只要想办法让 slow 再走 a 步停下来，就可以到环的入口。但是我们不知道 a 的值，该怎么办?
- 依然是使用双指针法。我们构建一个指针，此指针需要有以下性质：此指针和 slow 一起向前走 a 步后，两者在入口节点重合。那么从哪里走到入口节点需要 a 步？答案是链表头部 head 。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class EntryNodeInList {
    // 解法1：快慢指针
    public ListNode detectCycle(ListNode head) {
        ListNode fast = head, slow = head;
        while (true) {
            if (fast == null || fast.next == null) return null;
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        fast = head;
        while (slow != fast) {
```

```
            slow = slow.next;
            fast = fast.next;
        }
        return fast;
    }
}
```

24 反转链表

1. 描述

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:

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

2. 思路

使用三个指针分别指向前节点、当前节点、下一个节点，遍历链表重新指向前面节点即可。（注意一旦指向新的，链表就会段落，所以要有下一个节点的指针）。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```
public class ReverseList {
    // 实现
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode preNode = null;
        ListNode pCurrent = head;
        ListNode pNext = pCurrent.next;
        while (pNext != null) {
            if (preNode == null) {
                pCurrent.next = null;
            } else {
                pCurrent.next = preNode;
            }
            preNode = pCurrent;
            pCurrent = pNext;
            pNext = pNext.next;
        }
        // 最后一个节点注意要连上前面的节点
        pCurrent.next = preNode;
        return pCurrent;
    }
}
```

25 合并两个有序链表

1. 描述

输入两个递增排序的链表，要求合并后保持递增。

示例：

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

2. 思路

遍历两个链表，考虑将另外一个链表节点加入当前节点后面的条件（画图即可判断：另外一个节点小与当前节点，则加入，否则当前节点往下走）。

由于是一直是循环这个判断条件以及加入的逻辑，可考虑递归实现，也可以使用循环实现

循环时间空间复杂度：$O(m+n)$，$O(1)$

递归时间空间复杂度：$O(m+n)$，$O(m+n)$

3. 代码

```java
public class MergeSortedList {
    // 循环实现
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode current1 = l1;
        ListNode current2 = l2;
        ListNode pHead = null;
        ListNode pNext = pHead;
        while (current1 != null && current2 != null) {
            if (current1.val <= current2.val) {
                if (pHead == null) {
                    pHead = current1;
                    pNext = pHead;
                } else {
                    pNext.next = current1;
                    pNext = pNext.next;
                }
                current1 = current1.next;
            } else {
                if (pHead == null) {
                    pHead = current2;
                    pNext = pHead;
                } else {
                    pNext.next = current2;
                    pNext = pNext.next;
                }
                current2 = current2.next;
            }
        }
        if (current1 != null) {
            pNext.next = current1;
        }
```

```
        if (current2 != null) {
            pNext.next = current2;
        }
        return pHead;
    }

    //递归实现
    public ListNode mergeTwoLists1(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode pHead = null;
        if (l1.val <= l2.val) {
            pHead = l1;
            pHead.next = mergeTwoLists1(l1.next, l2);
        } else {
            pHead = l2;
            pHead.next = mergeTwoLists1(l1, l2.next);
        }
        return pHead;
    }
}
```

26 树的子结构

1. 描述

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

B是A的子结构， 即 A中有出现和B相同的结构和节点值。（需要考虑）

示例：

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

2. 思路

解法1：递归

由于是一直是循环这个判断条件以及加入的逻辑，可考虑递归实现。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```
public class SubstructureInTree {
    // 解法1：递归求解
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if (A == null || B == null) {
            return false;
        }
        boolean result = false;
```

```
        if (A != null && B != null) {
            // 从根节点判断
            if (A.val == B.val) {
                result = tree1HasTree2FromRoot(A, B);
            }
            // 根节点不等判断A左节点
            if (!result) {
                result = isSubStructure(A.left, B);
            }
            // 根节点与左节点都不是，判断A右节点
            if (!result) {
                result = isSubStructure(A.right, B);
            }
        }
        return result;
    }

    public boolean tree1HasTree2FromRoot(TreeNode A, TreeNode B) {
        // 判断顺序十分重要
        // B节点判断完返回true
        if (B == null) {
            return true;
        }
        // A空B不空返回false
        if (A == null) {
            return false;
        }
        // A B都不空但节点值不一样返回false
        if (A.val != B.val) {
            return false;
        }
        // 递归判断左右子节点
        return tree1HasTree2FromRoot(A.left, B.left) &&
tree1HasTree2FromRoot(A.right, B.right);
    }
}
```

27 二叉树的镜像

1. 描述

求一棵二叉树的镜像。

示例：

```
输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]
```

2. 思路

解法1：递归

二叉树的镜像，即左右子树调换。从上到下，递归完成即可。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class MirrorOfBinaryTree {
    // 解法1：递归
    public void mirrorRecursively(TreeNode<Integer> root) {
        if (root == null) {
            return;
        }
        TreeNode<Integer> temp = root.left;
        root.left = root.right;
        root.right = temp;
        mirrorRecursively(root.left);
        mirrorRecursively(root.right);
    }
}
```

28 对称的二叉树

1. 描述

判断一棵二叉树是不是对称的。如果某二叉树与它的镜像一样，称它是对称的。

示例：

```
输入：root = [1,2,2,3,4,4,3]
输出：true
```

2. 思路

比较直接的思路是比较原树与它的镜像是否一样。书中就是用的这种方式（比较二叉树的前序遍历和对称前序遍历）。但这种思路下，树

的每个节点都要读两次，也就是遍历两遍。

其实可以只遍历一次完成判断：我们可以通过判断待判断二叉树的左子树与右子树是不是对称的来得知该二叉树是否是对称的。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.LinkedList;
import java.util.Queue;

public class SymmetricalBinaryTree {
    // 实现1：递归实现
    public boolean isSymmetric(TreeNode root) {
        return isSymmetricCore(root, root);
    }

    public boolean isSymmetricCore(TreeNode A, TreeNode B) {
        // 三种情况的返回值
        if (A == null && B == null) {
            return true;
        }
        if (A == null || B == null) {
```

```
            return false;
        }
        if (A.val != B.val) {
            return false;
        }
        return isSymmetricCore(A.left, B.right) && isSymmetricCore(A.right,
B.left);
    }

    // 实现2：迭代实现
    public static boolean isSymmetrical1(TreeNode<Integer> root) {
        if (root == null)
            return true;
        if (root.left == null && root.right == null)
            return true;
        if (root.left == null || root.right == null)
            return false;
        Queue<TreeNode<Integer>> queueLeft = new LinkedList<>();
        Queue<TreeNode<Integer>> queueRight = new LinkedList<>();
        queueLeft.offer(root.left);
        queueRight.offer(root.right);
        TreeNode<Integer> tempLeft, tempRight;
        while (!queueLeft.isEmpty() || !queueRight.isEmpty()) {
            boolean left = false;
            boolean right = false;
            tempLeft = queueLeft.poll();
            tempRight = queueRight.poll();
            if (tempLeft.val.equals(tempRight.val)) {
                // 注意顺序 左-->右
                if (tempLeft.left != null)
                    queueLeft.offer(tempLeft.left);
                else
                    left = true;
                if (tempLeft.right != null)
                    queueLeft.offer(tempLeft.right);
                else
                    right = true;
                // 注意顺序 右-->左
                if (tempRight.right != null) {
                    if (left == true) {
                        return false;
                    }
                    queueRight.offer(tempRight.right);
                } else {
                    if (left == false) {
                        return false;
                    }
                }

                if (tempRight.left != null) {
                    if (right == true) {
                        return false;
                    }
                    queueRight.offer(tempRight.left);
                } else {
                    if (right == false) {
                        return false;
                    }
```

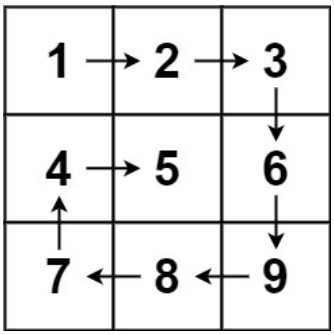
```
                }
            } else
                return false;
        }
        if (queueLeft.isEmpty() && queueRight.isEmpty())
            return true;
        else
            return false;
    }
}
```

29 顺时针打印矩阵

1. 描述

输入一个矩阵，按照从外向里以顺时针的顺序一次打印出每一个数字。

示例：




```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

2. 思路

显然，**模拟法。**

可以将矩阵看成若干层，首先输出最外层的元素，其次输出次外层的元素，直到输出最内层的元素。

对于每层，从左上方开始以顺时针的顺序遍历所有元素。假设当前层的左上角位于 (top,left)，右下角位于 (bottom,right)，按照如下顺序遍历当前层的元素。

- 从左到右遍历上侧元素，依次为 (top,left) 到 (top,right)。
- 从上到下遍历右侧元素，依次为(top+1,right) 到 (bottom,right)。
- 如果 left<right 且 top<bottom，则从右到左遍历下侧元素，依次为 (bottom,right−1) 到 (bottom,left+1)，以及从下到上遍历左侧元素，依次为 (bottom,left) 到 (top+1,left)；**否则即只剩下一行或者一列，从右到左或者从下到上不再遍历**

3. 代码

```
public class PrintMatrix {
    // 模拟法
    public int[] spiralOrder(int[][] matrix) {
        if (matrix == null) {
            return new int[0];
        }
        if(matrix.length==0){
```

```
            return new int[0];
        }
        int length = matrix.length * matrix[0].length;
        int[] array = new int[length];
        spiralOrderCore(array, 0, matrix, 0, matrix.length - 1, 0,
matrix[0].length - 1);
        return array;
    }

    public void spiralOrderCore(int[] array, int index, int[][] matrix, int
rowStart, int rowEnd,
                                int colStart, int colEnd) {
        if (rowStart > rowEnd || colStart > colEnd) {
            return;
        }
        int count = index;
        // 只有一行
        if (rowStart == rowEnd) {
            for (int j = colStart; j <= colEnd; j++) {
                array[count++] = matrix[rowStart][j];
            }
            return;
        }
        // 只有一列
        if (colStart == colEnd) {
            for (int i = rowStart; i <= rowEnd; i++) {
                array[count++] = matrix[i][colStart];
            }
            return;
        }
        // 矩阵第一行
        for (int j = colStart; j <= colEnd; j++) {
            array[count++] = matrix[rowStart][j];
        }
        // 矩阵最后一列
        for (int i = rowStart + 1; i <= rowEnd; i++) {
            array[count++] = matrix[i][colEnd];
        }
        // 矩阵最后一行
        for (int j = colEnd - 1; j >= colStart; j--) {
            array[count++] = matrix[rowEnd][j];
        }
        // 矩阵最左一列
        for (int i = rowEnd - 1; i >= rowStart + 1; i--) {
            array[count++] = matrix[i][colStart];
        }
        spiralOrderCore(array, count, matrix, rowStart + 1, rowEnd - 1,
colStart + 1, colEnd - 1);
    }
}
```

30 包含min函数的栈

1. 描述

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

示例：

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.min();   --> 返回 -2
```

2. 思路

记录一个最小数栈，当传进来元素大于最小栈的栈顶元素时，需要把最小栈的栈顶元素再次压栈，否则把传进来的元素压进最小栈，当有出栈操作时，同时把最小栈的栈顶元素出栈

注意：栈为空的情况

3. 代码

```
import java.util.Stack;

public class StackWithMin {
    private Stack stackData;
    // 最小栈
    private Stack minData;

    public StackWithMin() {
        this.stackData = new Stack();
        this.minData = new Stack();
    }

    // 进栈操作
    public void push(int x) {
        this.stackData.push(x);
        // 当最小栈为空，或者其栈顶元素大于等于待进栈元素，将待进栈元素压入最小栈，否则将最小栈栈顶元素压入最小栈
        if (this.minData.isEmpty() || (int) this.minData.peek() >= x) {
            this.minData.push(x);
        } else {
            this.minData.push(this.minData.peek());
        }
    }

    // 出栈操作
    public void pop() {
        // 栈空，直接返回；不为空，两个栈都要出栈
        if (this.stackData.isEmpty()) {
            return;
        } else {
            this.stackData.pop();
            this.minData.pop();
        }
    }

```

```
    // 栈顶元素
    public int top() {
        return (int) this.stackData.peek();
    }

    // 最小值，最小栈为空，返回错误；否则返回最小栈栈顶元素
    public int min() {
        if (this.minData.isEmpty()) {
            return -1;
        } else {
            return (int) this.minData.peek();
        }
    }
}
```

31 栈的压入弹出序列

1. 描述

输入两个整数序列，第一个序列表示栈的压入顺序，判断第二个序列是否为该栈的弹出序序列。假设压入栈的所有数字均不相等。例如，压入序列为(1,2,3,4,5)，序列(4,5,3,2,1)是它的弹出序列，而(4,3,5,1,2)不是。

示例：

```
输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

2. 思路

对于一个给定的压入序列，由于弹出的时机不同，会出现多种弹出序列。如果是选择题，依照后进先出的原则，复现一下栈的压入弹出过程就很容易判断了。写成程序同样如此，主要步骤如下：

时间空间复杂度：$O(n)$，$O(n)$

```
步骤1：栈压入序列第一个元素，弹出序列指针指弹出序列的第一个；
步骤2：判断栈顶元素是否等于弹出序列的第一个元素：
    步骤2.1：如果不是，压入另一个元素，进行结束判断，未结束则继续执行步骤2；
    步骤2.2：如果是，栈弹出一个元素，弹出序列指针向后移动一位，进行结束判断（pop数组是否索引完），未结束则继续执行步骤2；

结束条件：如果弹出序列指针还没到结尾但已经无元素可压入，则被测序列不是弹出序列。
          如果弹出序列指针已判断完最后一个元素，则被测序列是弹出序列。
```

3. 代码

```
import java.util.Stack;

public class StackPushPopOrder {
    // 模拟栈的操作
    public boolean validateStackSequences(int[] pushed, int[] popped) {
```

```
        // 无效输入
        if (pushed == null || popped == null || pushed.length != popped.length) {
            return false;
        }
        int pushedIndex = 0;
        int poppedIndex = 0;
        // 模拟进栈出栈操作（针对进栈序列）
        Stack stack = new Stack();
        // 弹出序列判断是否结束
        while (poppedIndex < popped.length) {
            // 栈为空或者栈顶元素不等于弹出序列第一个元素，当压入序列不为空，继续将压入序列元素进栈，否则返回false
            if (stack.isEmpty() || (int) stack.peek() != popped[poppedIndex]) {
                if (pushedIndex < pushed.length) {
                    stack.push(pushed[pushedIndex++]);
                } else {
                    return false;
                }
            } else {
                // 栈不为空且栈顶元素等于弹出序列第一个元素，出栈，弹出序列索引后移，继续判断
                stack.pop();
                poppedIndex++;
            }
        }
        return true;
    }
}
```

32.1 从上到下打印二叉树I

1. 描述

从上到下打印二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

示例：

```
给定二叉树: [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
返回：
[3,9,20,15,7]
```

2. 思路

二叉树的层数遍历，可以借助List存储相关元素进行返回

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class PrintTreeFromTopToBottom1 {
    // 层序遍历1
    public int[] levelOrder(TreeNode root) {
        if (root == null) {
            return new int[0];
        }
        // 比 LinkList 好一些
        ArrayList<Integer> list = new ArrayList<Integer>();
        Queue queue = new LinkedList<>();
        queue.add(root);
        TreeNode temp = null;
        while (!queue.isEmpty()) {
            temp = (TreeNode) queue.poll();
            list.add(temp.val);
            if (temp.left != null) {
                queue.offer(temp.left);
            }
            if (temp.right != null) {
                queue.offer(temp.right);
            }
        }
        int[] array = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            array[i] = list.get(i);
        }
        return array;
    }
}
```

32.2 从上到下打印二叉树II

1. 描述

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印一行。

示例：

```
给定二叉树: [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
返回其层次遍历结果：
[
  [3],
  [9,20],
  [15,7]
]
```

2. 思路

同样是层序遍历，与上一题不同的是，此处要记录每层的节点个数，使用在层序遍历中使用循环将当前层的所有节点元素输出。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class PrintTreeFromTopToBottom2 {
    // 层序遍历2
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> list = new ArrayList();
        if (root == null) {
            return list;
        }
        Queue queue = new LinkedList();
        queue.add(root);
        while (!queue.isEmpty()) {
            List<Integer> tempList = new ArrayList<>();
            int size = queue.size();
            for (int i = size - 1; i >= 0; i--) {
                TreeNode temp = (TreeNode) queue.poll();
                tempList.add(temp.val);
                if (temp.left != null) {
                    queue.offer(temp.left);
                }
                if (temp.right != null) {
                    queue.offer(temp.right);
                }
            }
            list.add(tempList);
        }
        return list;
    }
}
```

32.3 从上到下打印二叉树III

1. 描述

请实现一个函数按照之字形打印二叉树。即第一层从左到右打印，第二层从右到左打印，第三层继续从左到右，以此类推。

示例：

```
给定二叉树: [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
返回其层次遍历结果：
[
  [3],
  [9,20],
  [15,7]
]
```

2. 思路

与上面第二个的输出很类似，只需要按照层数的奇偶性将存储元素的链表的对称元素互换即可。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class PrintTreeFromTopToBottom3 {
    // 层序遍历3
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> list = new ArrayList();
        if (root == null) {
            return list;
        }
        Queue queue = new LinkedList();
        queue.add(root);
        int count = 0;
        while (!queue.isEmpty()) {
            count++;
            List<Integer> tempList = new ArrayList<>();
            int size = queue.size();
            for (int i = size - 1; i >= 0; i--) {
                TreeNode temp = (TreeNode) queue.poll();
                tempList.add(temp.val);
                if (temp.left != null) {
                    queue.offer(temp.left);
                }
                if (temp.right != null) {
                    queue.offer(temp.right);
                }
            }
            if ((count & 1) == 0) {
                int size = tempList.size();
                for (int i = 0; i < size / 2; i++) {
                    int temp = tempList.get(i);
                    tempList.set(i, tempList.get(length - 1 - i));
                    tempList.set(length - 1 - i, temp);
                }
```

```
            }
            list.add(tempList);
        }
        return list;
    }
}
```

33 二叉搜索树的后序遍历

1. 描述

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果，假设输入数组的任意两个数都互不相同。

示例：

```
输入: [1,6,3,2,5]
输出: false
```

2. 思路

后序遍历

- 比根结点小的值都在根结点的左子树上，一旦有大于根结点的值出现，说明左子树已经遍历完。
- 找到第一个大于根结点的值，这个值就是后序遍历右子树的第一个点。
- 从第2步找到的点开始遍历完剩下的所有结点，如果在这期间发现有比根结点小的值，说明不是BST
- 左子树和右子树递归执行前三步。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```
public class verifySquenceOfBST {
    // 递归：根据二叉搜索树的性质
    public boolean verifyPostorder(int[] postorder) {
        return verifySquenceOfBSTCore(postorder, 0, postorder.length - 1);
    }

    public boolean verifySquenceOfBSTCore(int[] postorder, int start, int
end) {
        if (start >= end) return true;
        int temp = start;
        // 找到右子树结点第一次出现的地方。（或者说是遍历完整棵左子树）
        for (int i = start; i <= end; ++i) {
            if (postorder[i] < postorder[end]) {
                temp = i;
            } else break;
        }
        // 后序遍历右子树时会访问的第一个结点的下标。
        int rightTreeNode = temp + 1;
        // 验证右子树所有结点是否都大于根结点。
        for (int i = rightTreeNode; i < end; ++i) {
            if (postorder[i] > postorder[end]){
                ++rightTreeNode;
```

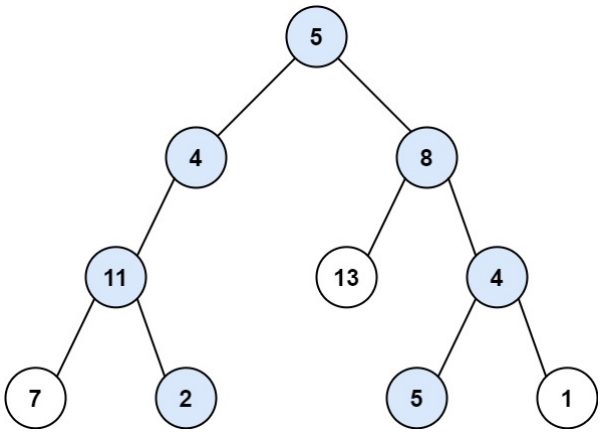
```
            }else {
                return false;
            }
        }
        return verifySquenceOfBSTCore(postorder, start, temp) &&
verifySquenceOfBSTCore(postorder, temp + 1, end - 1);
    }
}
```

34 二叉树中和为某一值的路径

1. 描述

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

示例：




```
输入：root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
输出：[[5,4,11,2],[5,8,4,5]]
```

2. 思路

解法1：深度优先遍历

注意题的条件，从根节点到叶子节点之和，而不是到某个节点之和，所以必须要遍历二叉树，可以采用深度优先遍历（先序遍历），记录每条路径之和，与期望值相等则加入list

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```
import java.util.ArrayList;
import java.util.List;

public class FindPath {
    // 回溯（深度优先遍历）
    public List<List<Integer>> pathSum(TreeNode root, int target) {
        List<List<Integer>> list = new ArrayList<>();

        if (root == null) {
            return list;
        }
        List<Integer> current = new ArrayList<>();
```

```
        pathSumCore(root, target, list, current, 0);
        return list;
    }

    public void pathSumCore(TreeNode root, int target, List<List<Integer>>
list, List<Integer> current, int value) {
        current.add(root.val);
        value = value + root.val;
        if (root.left != null)
            pathSumCore(root.left, target, list, current, value);
        if (root.right != null)
            pathSumCore(root.right, target, list, current, value);
        if (value == target && root.left == null && root.right == null) {
            // 由于current是引用类型，在下面操作后，即使加入list也会发生变化，所以要创
建新的对象
            List<Integer> addList = new ArrayList<>();
            for (int i = 0; i < current.size(); i++) {
               addList.add(current.get(i));
            }
            list.add(addList);
        }
        current.remove(current.size() - 1);
    }
}
```

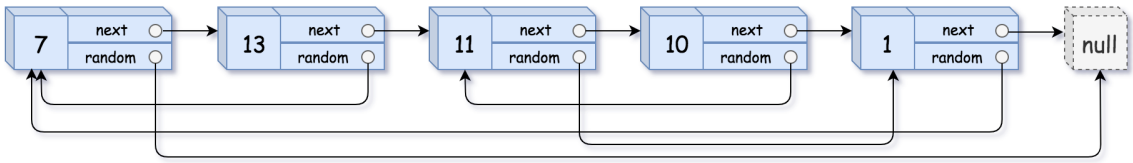
35 复杂链表的复制

1. 描述

在复杂链表中，每个节点除了有一个next指针指向下一个节点，还有一个random指针指向链表中的任意节点或null，请完成一个能够复制复杂链表的函数。

注意：复制链表是指重新开辟内存进行操作，而不是单单对原来内存的引用进行操作

示例：




```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

2. 思路

解法1：二次遍历

先复制原来链表的value和next指针，在遍历链表复制random指针（此时需要双重循环）

时间空间复杂度：$O(n^2)$，$O(1)$

解法2：哈希表

利用hash存储旧链表节点到新链表节点的映射，这样就可以直接通过hash根据旧链表的随机指针找到在新链表对应的随机指针，注意随机指针为空的情况。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```
class CopyRandomList {
    // 解法1：二次遍历
    public Node copyRandomList(Node head) {
        if (head == null) {
            return head;
        }
        Node newHead = new Node(head.val);
        Node newCurrent = newHead;
        Node current = head.next;
        // 先复制next指针
        while (current != null) {
            Node temp = new Node(current.val);
            newCurrent.next = temp;
            newCurrent = newCurrent.next;
            current=current.next;
        }
        newCurrent.next = null;
        // 再复制随机指针
        newCurrent = newHead;
        current = head;
        while (newCurrent != null) {
            Node random = current.random;
            if (random == null) {
                newCurrent.random = null;
            } else {
                Node temp = head;
                Node newTemp = newHead;
                while (random != temp) {
                    temp = temp.next;
                    newTemp = newTemp.next;
                }
                newCurrent.random = newTemp;
            }
            newCurrent = newCurrent.next;
            current = current.next;
        }
        return newHead;
    }

    //解法2：哈希表
    public Node copyRandomList(Node head) {
        if (head == null) {
            return head;
        }
        Map<Node, Node> map = new HashMap<>();
        Node newHead = new Node(head.val);
        map.put(head, newHead);
        Node newCurrent = newHead;
        Node current = head.next;
        // 先复制next指针
        while (current != null) {
```

```
            Node temp = new Node(current.val);
            map.put(current, temp);
            newCurrent.next = temp;
            newCurrent = newCurrent.next;
            current = current.next;
        }
        newCurrent.next = null;
        // 再复制随机指针
        current = head;
        newCurrent = newHead;
        while (current != null) {
            if (current.random != null) {
                newCurrent.random = map.get(current.random);
            }
            current = current.next;
            newCurrent = newCurrent.next;
        }
        return newHead;
    }
}
```

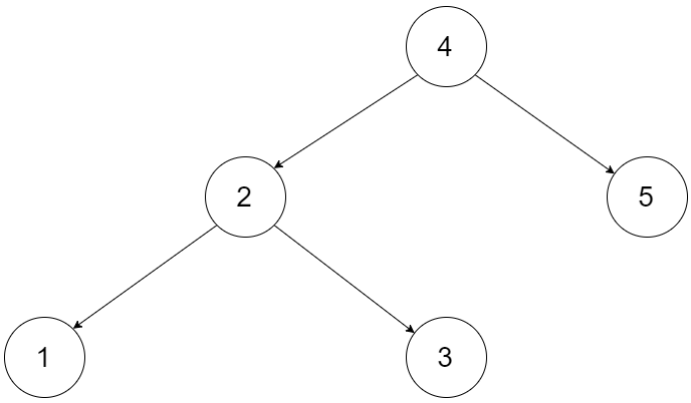
36 二叉搜索树与双向链表

1. 描述

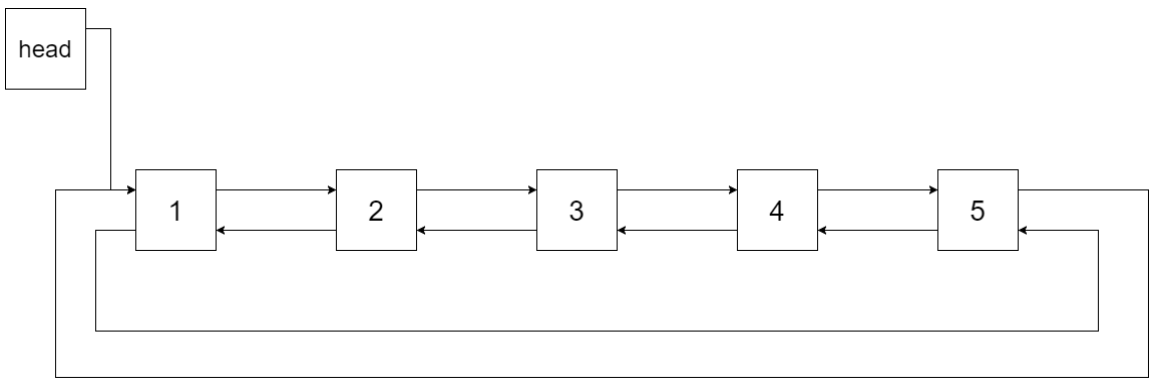
输入一颗二叉搜索树，将该二叉搜索树转换成一个排序的双向链表，不能创建任何新的节点，只能调整树中节点指针的指向。

示例:

以下面的二叉搜索树为例:



下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。



2. 思路

- 可将树的左右指针作为双向链表的前后指针
- 排序的双向链表，因为是二叉搜索树，所以其中序遍历即为排序好的
- 指针重定向，可将上步的结果存到list，循环遍历重定向指针即可（注意边缘以及特殊情况，如单节点）

另外可以使用前后两个节点指针控制左右节点，在中序遍历的同时完成指针重定向

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.ArrayList;
import java.util.List;

public class ConvertBinarySearchTree {
    // 实现
    public TreeNode treeToDoublyList(TreeNode root) {
        if (root == null) {
            return null;
        }
        if (root.left == null && root.right == null) {
            root.left = root;
            root.right = root;
            return root;
        }
        List<TreeNode> list = new ArrayList();
        getList(root, list);
        int length = list.size();
        for (int i = 0; i < length; i++) {
            if (i == 0) {
                list.get(i).left = list.get(length - 1);
                list.get(i).right = list.get(i + 1);
            } else if (i == length - 1) {
                list.get(i).left = list.get(i - 1);
                list.get(i).right = list.get(0);
            } else {
                list.get(i).left = list.get(i - 1);
                list.get(i).right = list.get(i + 1);
            }
        }
        return list.get(0);
    }

    // 中序遍历
    public void getList(TreeNode root, List<TreeNode> list) {
        if (root == null) {
            return;
        }
        getList(root.left, list);
        list.add(root);
        getList(root.right, list);
    }
}
```

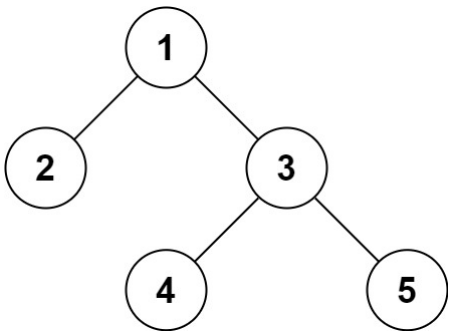
37 序列化二叉树

1. 描述

实现两个函数，分别用来序列化和反序列化二叉树。

你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例：




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

2. 思路

解法1：前序遍历

- 能让人想到重建二叉树。但二叉树序列化为前序遍历序列和中序遍历序列，然后反序列化为二叉树的思路在本题有两个关键缺点：1.全部数据都读取完才能进行反序列化。2.该方法需要保证树中节点的值各不相同（本题无法保证）。
- 其实，在遍历结果中，记录null指针后，那么任何一种遍历方式都能回推出原二叉树。但是如果期望边读取序列化数据，边反序列化二叉树，那么仅可以使用前序或层序遍历。
- 此处代码使用的是前序遍历，在恢复字符串的时候比较方便

时间空间复杂度：$O(nlogn)$，$O(n)$

3. 代码

```java
public class Codec {
    // 递归实现前序遍历，并实现序列化
    public String serialize(TreeNode root) {
        if (root == null)
            return "null,";
        StringBuilder result = new StringBuilder();
        result.append(root.val);
        result.append(",");
        result.append(serialize(root.left));
        result.append(serialize(root.right));
        return result.toString();
    }

    // 反序列化
    public TreeNode deserialize(String data) {
        StringBuilder stringBuilder = new StringBuilder(data);
        return deserializeCore(stringBuilder);
    }

```

```
    public TreeNode deserializeCore(StringBuilder stringBuilder) {
        if (stringBuilder.length() == 0)
            return null;
        // 取出根节点的值
        String num = stringBuilder.substring(0, stringBuilder.indexOf(","));
        // 为了之后方便递归，删除当前节点
        stringBuilder.delete(0, stringBuilder.indexOf(","));
        // 删除多余 ','
        stringBuilder.deleteCharAt(0);
        // 根节点为空，直接返回null
        if (num.equals("null"))
            return null;
        // 根据取出的根节点值，创建节点
        TreeNode node = new TreeNode(Integer.parseInt(num));
        // 递归创建左子树、右子树
        node.left = deserializeCore(stringBuilder);
        node.right = deserializeCore(stringBuilder);
        return node;
    }
}
```

38 字符串的排列

1. 描述

输入一个字符串，打印出该字符串中字符的所有排列。

示例：

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

2. 思路

显而易见，回溯法（深度优先遍历）——元素交换法。

- 注意字符串的重复元素
- 使用交互前后字符解决问题（注意交换后不能对之后的处理造成影响）
- 使用set集合处理重复元素的交换（即与后面字符进行交换后加入集合，加入失败则代表重复，直接跳过）

时间空间复杂度：$O(n*n!)$，$O(n)$

3. 代码

```
import java.util.*;

public class StringPermutation {
    List<String> res = new ArrayList<>();
    char[] c;

    // 回溯
    public String[] permutation(String s) {
        c = s.toCharArray();
        dfs(0);
        return res.toArray(new String[res.size()]);
```

```java
    }

    public void dfs(int x) {
        if(x == c.length - 1) {
            res.add(String.valueOf(c));       // 添加排列方案
            return;
        }
        HashSet<Character> set = new HashSet<>();
        for(int i = x; i < c.length; i++) {
            if(set.contains(c[i])) continue; // 重复，因此剪枝
            set.add(c[i]);
            swap(i, x);                      // 交换，将 c[i] 固定在第 x 位
            dfs(x + 1);                      // 开启固定第 x + 1 位字符
            swap(i, x);                      // 恢复交换
        }
    }

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

39 数组中出现次数超过一半的数字

1. 描述

找出数组中出现次数超过数组长度一半的数字。如输入{1,2,3,2,2,2,5,4,2}，则输出2。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例：

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

2. 思路

解法1：排序

对数组进行排序，最中间的数一定是多数元素。

时间空间复杂度：$O(nlogn)$，$O(logn)$

解法2：利用缓存思想

根据数字特点，利用缓存思想

```
步骤1：  缓存值value，命中次数count均初始化为0。
步骤2：  从头到尾依次读取数组中的元素，判断该元素是否等于缓存值：
      步骤2.1：如果该元素等于缓存值，则命中次数加一。
      步骤2.2：如果该元素不等于缓存值，判断命中次数是否大于1：
                步骤2.2.1：如果命中次数大于1，将命中次数减去1。
                步骤2.2.2：如果命中次数小于等于1，则令缓存值等于元素值，命中次数设为1
步骤3：  最终的缓存值value即为数组中出现次数超过一半的数字。
```

时间空间复杂度：$O(n)$，$O(1)$

解法3：哈希表

遍历数组，存储到哈希表中；找出出现次数最多的那个元素即可。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.Arrays;

class MajorityElement {
    // 解法1：排序
    public int majorityElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        Arrays.sort(nums);
        return nums[nums.length / 2];
    }

    // 解法2：缓存思想
    public int majorityElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int count = 1;
        int current = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] == current) {
                count++;
            } else {
                if (--count < 1) {
                    current = nums[i];
                    // 注意重置出现次数
                    count = 1;
                }
            }
        }
        return current;
    }

    // 解法3：哈希
    public int majorityElement(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
        for (int num : nums) {
            if (!counts.containsKey(num)) {
                counts.put(num, 1);
            } else {
                counts.put(num, counts.get(num) + 1);
            }
        }
        int sum = -1;
        int result = 0;
```

```
        for (Map.Entry<Integer, Integer> entry : counts.entrySet()) {
            if (entry.getValue() > sum) {
                sum = entry.getValue();
                result = entry.getKey();
            }
        }
        return result;
    }
}
```

40 最小的k个数

1. 描述

找出n个整数中最小的k个数。

示例：

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

2. 思路

解法1：排序

对数组进行排序，取前面k个元素。

时间空间复杂度：$O(nlogn)$，$O(1)$

解法2：大顶堆

利用大小为 k 大顶堆存储数组元素，最后留下来的元素即是所求。

时间空间复杂度：$O(nlogk)$，$O(k)$

3. 代码

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;

public class KLeastNumbers {
    // 解法1：排序，取前面k个元素
    public int[] getLeastNumbers1(int[] arr, int k) {
        if (arr == null || k <= 0 || arr.length < k) {
            return null;
        }
        Arrays.sort(arr);
        int[] array = new int[k];
        for (int i = 0; i < k; i++) {
            array[i] = arr[i];
        }
        return array;
    }

    // 解法2：最大堆
    public int[] getLeastNumbers2(int[] arr, int k) {
```

```
        if (arr == null || k <= 0 || arr.length < k) {
            return null;
        }
        int[] vec = new int[k];
        // 维护一个大小为k的顺序堆
        PriorityQueue<Integer> queue = new PriorityQueue<Integer>(new
Comparator<Integer>() {
            public int compare(Integer num1, Integer num2) {
                return num2 - num1;
            }
        });
        for (int i = 0; i < k; ++i) {
            queue.offer(arr[i]);
        }
        for (int i = k; i < arr.length; ++i) {
            if (queue.peek() > arr[i]) {
                queue.poll();
                queue.offer(arr[i]);
            }
        }
        for (int i = 0; i < k; ++i) {
            vec[i] = queue.poll();
        }
        return vec;
    }
}
```

41 数据流中的中位数

1. 描述

何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

示例：

```
输入：
["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]
[[],[1],[2],[],[3],[]]
输出：[null,null,null,1.50000,null,2.00000
```

2. 思路

解法1：利用List

采用List保存每个添加的元素，通过排序计算中位数。（超时）

时间空间复杂度：$O(nlogn)$，$O(n)$

解法2：大顶堆，小顶堆

时间空间复杂度：$O(1)$，$O(n)$

进阶 1
如果数据流中所有整数都在 0 到 100 范围内，那么我们可以利用计数排序统计每一类数的数量，并使用双指针维护中位数。

进阶 2
如果数据流中 99% 的整数都在 0 到 100 范围内，那么我们依然利用计数排序统计每一类数的数量，并使用双指针维护中位数。对于超出范围的数，我们可以单独进行处理，建立两个数组，分别记录小于 0 的部分的数的数量和大于 100 的部分的数的数量即可。当小部分时间，中位数不落在区间 [0,100] 中时，我们在对应的数组中暴力检查即可。

3. 代码

```
import java.util.*;

public class MedianFinder {
    // 解法2：大、小顶堆
    int sum = 0;
    PriorityQueue<Integer> min = new PriorityQueue<>();
    PriorityQueue<Integer> max = new PriorityQueue<>(new Comparator<Integer>
() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    });

    public MedianFinder1() {
    }

    public void addNum(int num) {
        if (sum % 2 == 0) {
            max.add(num);
            min.add(max.poll());
        } else {
            min.add(num);
            max.add(min.poll());
        }
        sum++;
    }

    public double findMedian() {
        if (sum % 2 == 0) {
            return (min.peek() + max.peek()) / 2.0;
        } else {
            return min.peek();
        }
    }
}
```

42 连续子数组的最大和

1. 描述

输入一个整型数组，数组里有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。例如输入的数组为{1,-2,3,10,-4,7,2,-5}，和最大的子数组为{3,10,-4,7,2}，因此输出为该子数组的和18。

示例：

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

2. 思路

显然是动态规划，状态转移方程如下：

dp[i] = max(dp[i-1]+nums[i], nums[i])（本身dp[i-1]代表之前最大，所以加上当前位与只有当前位最大作比较）

可通过变量进行空间优化。

时间空间复杂度：$O(n)$，$O(1)$

其实这道题可以这么想：

- 1.假如全是负数，那就是找最大值即可，因为负数肯定越加越大。
- 2.如果有正数，则肯定从正数开始计算和，不然前面有负值，和肯定变小了，所以从正数开始。
- 3.当和小于零时，这个区间就告一段落了，然后从下一个正数重新开始计算(也就是又回到 2 了)。而 dp 也就体现在这个地方。

3. 代码

```java
class MaxSubArray {
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int temp = nums[0];
        int result = temp;
        for (int i = 1; i < nums.length; i++) {
            temp = Math.max(temp + nums[i], nums[i]);
            if (temp > result) {
                result = temp;
            }
        }
        return result;
    }
}
```

43 1~n整数中1出现的次数

难

44 数字序列中某一位的数字

1. 描述

数字以0123456789101112131415... 的格式序列化到一个字符序列中。在这个序列中，第5位（从下标0开始计数）是5，第13位是1，第19位是4，等等。

示例：

```
输入：n = 3
输出：3
```

2. 思路

数学规律题。

做这道题前，我们对1位数，2位数，3位数...n位数进行分段，求出各自在数值序列的最大下标。

```
1 * 9 = 9                                   # 1位数的最大下标
1 * 9 + 2 * 90 = 189                        # 2位数的最大下标
1 * 9 + 2 * 90 + 3 * 900 = 2889             # 3位数的最大下标
1 * 9 + 2 * 90 + 3 * 900 + 4 * 9000 = 38889 # 4 位数的最大下标
```

所以，n位数的最大下标为 1×9 + 2×90 + 3×900 + ... + n×9×10^(n-1)

知道了上面的一点后，我们开始说说这道题的解题思路：

- 首先找到索引为 n 是哪个数的哪一位（例如n=11, 索引为11是10的其中一位）
  - 根据最大下标判断这个数是几位数
    若 n 为 2888，由于 （19 + 290）= 189 < 2888 < （19 + 290 + 3*900）= 2889
    所以，这个数是一个三位数。
  - 根据这个数的位数，就可以求出索引为n是哪个数的第几位
    (2888 - 189)/3 = 899 ... 2。所以，这个数是 10^(3-1) + 899 = 999 的第 2 位
    注意：当 n = 2889时，(2889-189)/3 = 900 ... 0
    10^(3-1) + 900 = 1000 - 1 = 999 的 第3位。
- 获取这个数的第m位
  例如 999 的第3位，可以将其转换为字符串，再指定索引即可。

3. 代码

```java
public class DigitsInSequence {
    // 数学规律
    public int findNthDigit(int n) {
        if (n < 0) {
            return -1;
        }
        // digit : 表示 digit 位数
        // max : 表示 dight 位数的最大下标
        // lastMax : 表示 dight - 1 位数的最大下标
        int max = 9, lastMax = 0;
        int digit = 1;
        // 找到这个数是几位数，也就是更新 digit 的过程
        while (n > max) {
            digit++;
            lastMax = max;
            max += digit * 9 * Math.pow(10, digit - 1);
        }
        // 根据这个数的位数，就可以求出索引为n是哪个数的第几位
        int num = (n - lastMax) / digit + (int) Math.pow(10, digit - 1);
        int nth = (n - lastMax) % digit;
        // 特殊情况 余数为0时，更新 nth 为这个数的最后一位
        if (nth == 0) {
            num--;
            nth = digit;
        }
```

```
        // 将求出的数 num 转化为 字符串，返回第 nth 位
        return String.valueOf(num).charAt(nth - 1) - '0';

    }
}
```

45 把数组排列成最小的数

1. 描述

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，使其为所有可能的拼接结果中最小的一个。

```
输入: [10,2]
输出: "102"
```

2. 思路

要对数组进行排序，关键点在于排序的规则需要重新定义。我们重新定义“大于”，“小于”，“等于”。如果a，b组成的数字ab的值大于ba，则称a"大于"b，小于与等于类似。比如3与32，因为332大于323，因此我们称3“大于”32。我们按照上述的“大于”，“小于”规则进行升序排列。

3. 代码

```
import java.util.Arrays;

public class SortArrayForMinNumber {
    // 实现1：快排
    public String minNumber(int[] nums) {
        String[] strs = new String[nums.length];
        for (int i = 0; i < nums.length; i++)
            strs[i] = String.valueOf(nums[i]);
        quickSort(strs, 0, strs.length - 1);
        StringBuilder res = new StringBuilder();
        for (String s : strs)
            res.append(s);
        return res.toString();
    }

    public void quickSort(String[] strs, int l, int r) {
        if (l >= r) return;
        int i = l, j = r;
        String tmp = strs[i];
        while (i < j) {
            while ((strs[j] + strs[l]).compareTo(strs[l] + strs[j]) >= 0 &&
i < j) j--;
            while ((strs[i] + strs[l]).compareTo(strs[l] + strs[i]) <= 0 &&
i < j) i++;
            tmp = strs[i];
            strs[i] = strs[j];
            strs[j] = tmp;
        }
        strs[i] = strs[l];
        strs[l] = tmp;
        quickSort(strs, l, i - 1);
```

```
        quickSort(strs, i + 1, r);
    }

    // 实现2：重写内置排序函数
    public static String sort(int[] nums) {
        if (nums == null || nums.length == 0) {
            return null;
        }
        String[] strs = new String[nums.length];
        for (int i = 0; i < nums.length; i++) {
            strs[i] = String.valueOf(nums[i]);
        }
        Arrays.sort(strs, (x, y) -> (x + y).compareTo(y + x));  // 语法糖
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < strs.length; i++) {
            sb = sb.append(strs[i]);
        }
        return sb.toString();
    }
}
```

46 把数字翻译成字符串

1. 描述

给定一个数字，按照如下规则翻译成字符串：0翻译成“a”，1翻译成“b”...25翻译成“z”。一个数字有多种翻译可能，例如12258一共有5种，分别是bccfi，bwfi，bczi，mcfi，mzi。实现一个函数，用来计算一个数字有多少种不同的翻译方法。

示例：

```
输入：12258
输出：5
解释：12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"
```

2. 思路

$num = x_1x_2\dots x_{i-2}x_{i-1}x_i\dots x_{n-1}x_n$

（例如： $12258 = x_1x_2x_3x_4x_5$ ）

⬇

设 $x_1x_2\dots x_{i-2}$ 的翻译方案数量为 $f(i-2)$

设 $x_1x_2\dots x_{i-2}x_{i-1}$ 的翻译方案数量为 $f(i-1)$

⬇

当整体翻译 $x_{i-1}x_i$ 时， $x_1x_2\dots x_{i-2}x_{i-1}x_i$ 的方案数为 $f(i-2)$

当单独翻译 x_i 时， $x_1x_2\dots x_{i-2}x_{i-1}x_i$ 的方案数为 $f(i-1)$

⬇

方案数的递推关系：

$$f(i) = \begin{cases} f(i-2) + f(i-1) & ，若数字\ x_{i-1}x_i\ 可被翻译 \\ f(i-1) & ，若数字\ x_{i-1}x_i\ 不可被翻译 \end{cases}$$

- **状态定义：** 设动态规划列表 dp ， $dp[i]$ 代表以 x_i 为结尾的数字的翻译方案数量。
- **转移方程：** 若 x_i 和 x_{i-1} 组成的两位数字可以被翻译，则 $dp[i] = dp[i-1] + dp[i-2]$ ；否则 $dp[i] = dp[i-1]$ 。
  - 可被翻译的两位数区间：当 $x_{i-1} = 0$ 时，组成的两位数是无法被翻译的（例如 $00, 01, 02, \cdots$ ），因此区间为 $[10, 25]$ 。

$$dp[i] = \begin{cases} dp[i-1] + dp[i-2] & ,10x_{i-1} + x_i \in [10, 25] \\ dp[i-1] & ,10x_{i-1} + x_i \in [0, 10) \cup (25, 99] \end{cases}$$

- **初始状态：** $dp[0] = dp[1] = 1$ ，即“无数字”和“第 1 位数字”的翻译方法数量均为 1 ；
- **返回值：** $dp[n]$ ，即此数字的翻译方案数量。

Q： 无数字情况 $dp[0] = 1$ 从何而来？

A： 当 num 第 $1, 2$ 位的组成的数字 $\in [10, 25]$ 时，显然应有 2 种翻译方法，即 $dp[2] = dp[1] + dp[0] = 2$ ，而显然 $dp[1] = 1$ ，因此推出 $dp[0] = 1$ 。

3. 代码

```java
public class TranslateNumbersToStrings {
    // 解法1：递归
    public int translateNum(int num) {
        if (num < 0) {
            return -1;
        }
        return f(num);
    }

    public int f(int num) {
        if (num < 10) {
            return 1;
        }
```

```
        // 当相邻两位数字在10-25才能被翻译，其他都不行
        if (num % 100 > 9 && num % 100 < 26) {
            return f(num / 10) + f(num / 100);
        } else {
            return f(num / 10);
        }
    }

    // 解法2：动态规划
    public int translateNumDp(int num) {
        if (num < 0) {
            return -1;
        }
        char[] ch = String.valueOf(num).toCharArray();
        int length = ch.length;
        int[] dp = new int[length + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= length; i++) {
            int n = (ch[i - 2] - '0') * 10 + (ch[i - 1] - '0');
            if (n > 9 && n < 26) {
                dp[i] = dp[i - 1] + dp[i - 2];
            } else {
                dp[i] = dp[i - 1];
            }
        }
        return dp[length];
    }
}
```

47 礼物的最大值

1. 描述

在一个m*n的棋盘的每一个格都放有一个礼物，每个礼物都有一定价值（大于0）。从左上角开始拿礼物，每次向右或向下移动一格，直到右下角结束。给定一个棋盘，求拿到礼物的最大价值。

示例:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 12
解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物
```

2. 思路

- 动态规划
  - 设 f(i, j)为从棋盘左上角走至单元格 (i ,j)的礼物最大累计价值，易得到以下递推关系：f(i,j) 等于 f(i,j-1) 和 f(i-1,j)中的较大值加上当前单元格礼物价值 grid(i,j) 。
  - f(i, j)=max[f(i, j−1), f(i−1, j)]+ grid(i, j)

动态规划解析：

- **状态定义：** 设动态规划矩阵 dp ， $dp(i,j)$ 代表从棋盘的左上角开始，到达单元格 (i,j) 时能拿到礼物的最大累计价值。
- **转移方程：**
  1. 当 $i=0$ 且 $j=0$ 时，为起始元素；
  2. 当 $i=0$ 且 $j\neq 0$ 时，为矩阵第一行元素，只可从左边到达；
  3. 当 $i\neq 0$ 且 $j=0$ 时，为矩阵第一列元素，只可从上边到达；
  4. 当 $i\neq 0$ 且 $j\neq 0$ 时，可从左边或上边到达；

$$dp(i,j)=\begin{cases} grid(i,j) & ,i=0,j=0 \\ grid(i,j)+dp(i,j-1) & ,i=0,j\neq 0 \\ grid(i,j)+dp(i-1,j) & ,i\neq 0,j=0 \\ grid(i,j)+\max[dp(i-1,j),dp(i,j-1)] & ,i\neq 0,j\neq 0 \end{cases}$$

- **初始状态：** $dp[0][0]=grid[0][0]$ ，即到达单元格 $(0,0)$ 时能拿到礼物的最大累计价值为 $grid[0][0]$ ；
- **返回值：** $dp[m-1][n-1]$ ， m,n 分别为矩阵的行高和列宽，即返回 dp 矩阵右下角元素。

空间复杂度优化：

- 由于 $dp[i][j]$ 只与 $dp[i-1][j]$, $dp[i][j-1]$, $grid[i][j]$ 有关系，因此可以将原矩阵 $grid$ 用作 dp 矩阵，即直接在 $grid$ 上修改即可。
- 应用此方法可省去 dp 矩阵使用的额外空间，因此空间复杂度从 $O(MN)$ 降至 $O(1)$ 。

3. 代码

```
public class MaxValueOfGifts {
    // 实现1：动态规划
    public int maxValue(int[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }
        int[][] dp = new int[grid.length][grid[0].length];

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (i == 0 && j == 0) {
                    dp[i][j] = grid[i][j];
                }
                if (i == 0 && j != 0) {
                    dp[i][j] = dp[i][j - 1] + grid[i][j];
                }
                if (i != 0 && j == 0) {
                    dp[i][j] = dp[i - 1][j] + grid[i][j];
                }
                if (i != 0 && j != 0) {
                    dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]) +
grid[i][j];
                }
            }
        }
        return dp[grid.length - 1][grid[0].length - 1];
    }

    // 实现2：优化，先初始化一行一列
    public static int maxValue1(int[][] grid) {
        if (grid == null || grid.length == 0) {
```

```
            return 0;
        }
        int[][] dp = new int[grid.length][grid[0].length];
        dp[0][0] = grid[0][0];
        for (int i = 1; i < grid.length; i++) {
            dp[i][0] = grid[i][0] + dp[i - 1][0];
        }
        for (int j = 1; j < grid[0].length; j++) {
            dp[0][j] = grid[0][j] + dp[0][j - 1];
        }

        for (int i = 1; i < grid.length; i++) {
            for (int j = 1; j < grid[0].length; j++) {
                dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]) + grid[i]
[j];
            }
        }
        return dp[grid.length - 1][grid[0].length - 1];
    }

    // 实现3：优化，由于dp[i][j]之和前面有关系，不会影响后面，所以可以直接在原来的数组上
进行改动，减少空间消耗
    public static int maxValue2(int[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }

        // 行列初始化
        for (int i = 1; i < grid.length; i++) {
            grid[i][0] += grid[i - 1][0];
        }
        for (int j = 1; j < grid[0].length; j++) {
            grid[0][j] += grid[0][j - 1];
        }

        for (int i = 1; i < grid.length; i++) {
            for (int j = 1; j < grid[0].length; j++) {
                grid[i][j] = Math.max(grid[i][j - 1], grid[i - 1][j]) +
grid[i][j];
            }
        }
        return grid[grid.length - 1][grid[0].length - 1];
    }

    // 实现4：通过扩展一行一列空间，解决矩阵边界问题以及代码简洁
    public static int maxValue3(int[][] grid){
        if (grid == null || grid.length == 0) {
            return 0;
        }
        int row = grid.length;
        int column = grid[0].length;
        //dp[i][j]表示从grid[0][0]到grid[i - 1][j - 1]时的最大价值
        int[][] dp = new int[row + 1][column + 1];
        for (int i = 1; i <= row; i++) {
            for (int j = 1; j <= column; j++) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) + grid[i -
1][j - 1];
            }
```

```
        }
        return dp[row][column];
    }
}
```

48 最长不含重复字符的子字符串

1. 描述

输入一个字符串（只包含a~z的字符），求其最长不含重复字符的子字符串的长度。

示例：

```
输入：“abcabcbb”
输出：3
解释：因为无重复字符的最长子串是 “abc”，所以其长度为 3。
```

2. 思路

解法1：动态规划

- 状态定义： 设动态规划列表 dp ， dp[j] 代表以字符 s[j] 为结尾的 “最长不重复子字符串” 的长度。
- 转移方程： 固定右边界 j ， 设字符 s[j] 左边距离最近的相同字符为 s[i] ， 即 s[i] = s[j] 。
  - 当 i < 0 即 字符 s[j] 之前没有出现过，则 dp[j] = dp[j-1] + 1 ；
  - 当 dp[j - 1] < j - i ， 说明字符 s[i] 在子字符串 dp[j-1] **区间之外** ， 则 dp[j] = dp[j - 1] + 1 ；
  - dp[j−1] ≥ j−i ， 说明字符 s[i] 在子字符串 dp[j-1] **区间之中** ， 则 dp[j] 的左边界由 s[i] 决定，即 dp[j] = j - i ；

  其中第一与第二种情况是一样的，可以合并。当 i < 0 时，由于 dp[j−1]≤j 恒成立，因而 dp[j - 1] < j - i 恒成立，故状态转移方程：

  $$dp[j] = \begin{cases} dp[j-1] + 1, dp[j-1] < j - i \\ j - i, \ dp[j-1] >= j - i \end{cases}$$
- **返回值：** max(dp) ， 即全局的 “最长不重复子字符串” 的长度。

由于返回值是取 dp 数组最大值，因此可借助变量 tmp 存储 dp[j] ， 变量 res 每轮更新最大值即可。

时间空间复杂度：$O(n)$，$O(n)$

解法2：双指针（滑动窗口）

借助解法1的思想，只要判断两个相邻的相同字符之间隔了多少字符，取其中最大值，如果之前没有出现过，索引按 -1 处理即可。

具体也是采用哈希表存储当前字符上一次出现的索引位置，计算当前位置与上一个位置差值，更新结果，并更新当前字符在哈希表的索引位置。

该方法也可以视为滑动窗口解法。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```
import java.util.HashMap;
import java.util.Map;

public class LongestSubstringWithoutDup {
```

```java
    // 解法1：动态规划
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> dic = new HashMap<>();
        int res = 0, tmp = 0;
        for (int j = 0; j < s.length(); j++) {
            int i = dic.getOrDefault(s.charAt(j), -1); // 获取索引 i
            dic.put(s.charAt(j), j); // 更新哈希表
            tmp = tmp < j - i ? tmp + 1 : j - i; // dp[j - 1] -> dp[j]
            res = Math.max(res, tmp); // max(dp[j - 1], dp[j])
        }
        return res;
    }

    // 解法2：双指针
    public int lengthOfLongestSubstring1(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }
        int length = s.length();
        Map<Character, Integer> map = new HashMap<>();
        int max = 0;
        int left = -1;
        for (int i = 0; i < length; i++) {
            if (map.containsKey(s.charAt(i))) {
                left = Math.max(left, map.get(s.charAt(i)));
            }
            map.put(s.charAt(i), i);
            max = Math.max(max, i - left);
        }
        return max;
    }
}
```

49 丑数

1. 描述

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

示例：

```
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
```

2. 思路

计算出丑数：因为每个丑数都可以看成是由1去乘以2、3、5，再乘以2、3、5而衍生出来的。可以用三个指针指向第一个丑数1，三个指针分别表示乘2，乘3，乘5，将三个指针计算出来的最小的丑数放在数组中，并将该指针向后移动一个位置。为了得到第1500个丑数，需要一个长度1500的数组来记录已经计算出来的丑数。因此这个思路也可以说是用空间换时间。

3. 代码

```java
public class Ugly {
```

```java
    // 直接求丑数
    public int getUglyNumber(int num) {
        if (num <= 0)
            return 0;
        int[] dp = new int[num];
        // 第一个丑数
        dp[0] = 1;
        // 分别计算当前丑数有多少个2，3，5相乘组成
        int mul2 = 0;
        int mul3 = 0;
        int mul5 = 0;
        for (int i = 1; i < num; i++) {
            // 计算前面三个丑数乘以2，3，5后哪个最小。
            dp[i] = min(dp[mul2] * 2, dp[mul3] * 3, dp[mul5] * 5);
            // 2*3=6，3*2=6，会有重复值
            if (dp[mul2] * 2 == dp[i]) {
                mul2++;
            }
            if (dp[mul3] * 3 == dp[i]) {
                mul3++;
            }
            if (dp[mul5] * 5 == dp[i]) {
                mul5++;
            }
        }
        return dp[num - 1];
    }

    public int min(int a, int b, int c) {
        int x = a > b ? b : a;
        return x > c ? c : x;
    }
}
```

50 第一个只出现一次的字符

1. 描述

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

示例：

```
输入：s = "abaccdeff"
输出：'b'
```

2. 思路

解法1：哈希表

由于只有小写字母，可以使用26位数组实现哈希，存储字符出现的次数。再次遍历字符串，判断当前字符的出现次数，如果是1直接返回。

时间空间复杂度：$O(n)$，$O(n)$

解法2：哈希表优化

由于只有小写字母，可以使用26位数组实现哈希，不再存储字符出现的次数，当多次出现时存储-1，只需要一次存储字符索引位置。

再次遍历26位数组返回索引不是-1且最小的即可。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
public class FirstUniqChar {
    // 解法1：哈希表
    public int firstUniqChar(String s) {
        if (s == null || s.length() == 0) {
            return -1;
        }
        int[] map = new int[26];
        for (int i = 0; i < s.length(); i++) {
            int index = s.charAt(i) - 'a';
            map[index]++;
        }
        for (int i = 0; i < s.length(); i++) {
            int index = s.charAt(i) - 'a';
            if (map[index] == 1) {
                return i;
            }
        }
        return -1;
    }

    // 解法2：哈希表
    public int firstUniqChar1(String s) {
        if (s == null || s.length() == 0) {
            return -1;
        }
        int[] map = new int[26];
        for (int i = 0; i < s.length(); i++) {
            int index = s.charAt(i) - 'a';
            if (map[index] > 0 || map[index] == -1) {
                map[index] = -1;
            } else {
                map[index] = i + 1;
            }
        }
        int result = -1;
        for (int i = 0; i < 26; i++) {
            if (map[i] > 0) {
                if (result == -1 || result > map[i]) {
                    result = map[i];
                }
            }
        }
        return result == -1 ? -1 : result - 1;
    }
}
```

51 数组中的逆序对

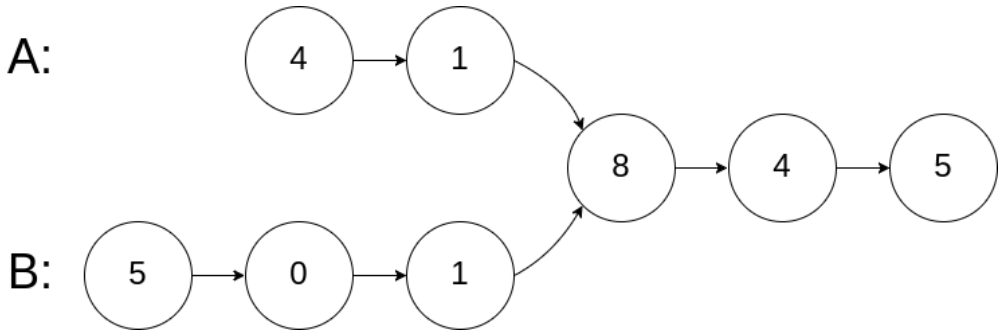
难

52 两个链表的第一个公共节点

1. 描述

输入两个链表，找出它们的第一个公共节点。

示例：




```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

2. 思路

解法1：栈

使用两个栈存储两个链表的节点，然后不断出栈比较节点是否相等即可。

时间空间复杂度：$O(m+n)$，$O(m+n)$

解法2：链表等长

根据题意，长链表的前面节点肯定不是相交节点，因此删除长链表的前面节点使得其与链表相等，然后两个链表同时往下走进行判断。

时间空间复杂度：$O(max(m,n))$，$O(1)$

解法3：双指针

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

时间空间复杂度：$O(m+n)$，$O(1)$

这样，当它们相遇时，所指向的结点就是第一个公共结点。

```
你变成我，走过我走过的路。
我变成你，走过你走过的路。
然后我们便相遇了..
```

3. 代码

```java
public class GetIntersectionNode {
    // 解法1：栈
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        Stack<ListNode> stack1 = new Stack<>();
        Stack<ListNode> stack2 = new Stack<>();
        ListNode node = headA;
        while (node != null) {
            stack1.push(node);
            node = node.next;
        }
        node = headB;
        while (node != null) {
            stack2.push(node);
            node = node.next;
        }
        ListNode result = null;
        while (!stack1.isEmpty() && !stack2.isEmpty()) {
            node = stack1.pop();
            if (node == stack2.pop()) {
                result = node;
            }
        }
        return result;
    }

    // 解法2：链表等长
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        int length1 = 0;
        int length2 = 0;
        ListNode node = headA;
        while (node != null) {
            length1++;
            node = node.next;
        }
        node = headB;
        while (node != null) {
            length2++;
            node = node.next;
        }
        ListNode node1 = headA;
        ListNode node2 = headB;
        if (length1 > length2) {
            for (int i = 0; i < length1 - length2; i++) {
                node1 = node1.next;
            }
        } else {
            for (int i = 0; i < length2 - length1; i++) {
                node2 = node2.next;
            }
```

```
        }
        while (node1 != null) {
            if (node1 == node2) {
                return node1;
            }
            node1 = node1.next;
            node2 = node2.next;
        }
        return null;
    }

    // 解法3：双指针（你的名字）
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode node1 = headA;
        ListNode node2 = headB;
        while (node1 != node2) {
            node1 = node1 == null ? headB : node1.next;
            node2 = node2 == null ? headA : node2.next;
        }
        return node1;
    }
}
```

53.1 数字在排序数组中出现的次数

1. 描述

统计一个数字在排序数组中出现的次数。

示例：

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

2. 思路

- 利用二分法求出小于等于target的位置start，如果num[start]不等于target，则数组不存在target直接返回，否则存在，且target首次出现位置等于start
- 当target存在时，利用二分法再次求出小于等于target+1的位置end。并判断结束位置end是否是最后一个元素以及是否与target相等，防止数组没有大于等于target+1的元素。
  - 如果是最后一个元素且等于target，则end是target最后出现的位置
  - 否则最后一个元素不是target，end-1是target最后出现的位置

时间空间复杂度：$O(logn)$，$O(1)$

3. 代码

```
public class SearchRange {
    // 解法1：暴力求解
    public int[] searchRange(int[] nums, int target) {
        int[] result = new int[]{-1, -1};
```

```
        if (nums == null || nums.length == 0) {
            return result;
        }
        int count = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == target) {
                if (result[0] == -1) {
                    result[0] = i;
                }
                count++;
            }
        }
        if (result[0] != -1) result[1] = result[0] + count - 1;
        return result;
    }

    // 解法2：二分求解
    public int[] searchRange1(int[] nums, int target) {
        int[] result = new int[]{-1, -1};
        if (nums == null || nums.length == 0) {
            return result;
        }
        // 查找元素的开始位置
        int start = search(nums, target);
        if (nums[start] != target) return result; //查找失败
        result[0] = start;
        // 查找元素的结束位置
        int end = search(nums, target + 1);
        // 判断结束位置，并判断结束位置是否是最后一个元素以及是否与target相等，防止数组没有大于等于target+1的元素
        if (end == nums.length - 1 && nums[end] == target) {
            result[1] = end;
        } else {
            result[1] = end - 1;
        }

        return result;
    }

    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        // 查找元素的开始位置
        while (left < right) {
            int mid = (left + right) >> 1;
            // 判断条件变体（如果有target则是最左边的，否则是大于target最小的数位置）
            if (nums[mid] >= target) right = mid;
            else left = mid + 1;
        }
        return left;
    }
}
```

53.2 0~n中缺失的数字

1. 描述

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例：

```
输入：[0,1,3]
输出：2
```

2. 思路

解法1：暴力求解

遍历数组，判断当前元素与其索引是否相等。

时间空间复杂度：$O(n)$，$O(1)$

解法2：二分查找

先利用二分找到与目标值相等的索引，然后左边界=mid+1，否则有边界=mid-1，返回left

时间空间复杂度：$O(logn)$，$O(1)$

解法3：数学求和

计算0~n之间的和，再计算数组所有元素和，两者之差即为缺失的元素。

时间空间复杂度：$O(n)$，$O(1)$

3. 代码

```java
public class GetMissingNumber {
    // 解法1：暴力
    public int missingNumber(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int length = nums.length;
        // 注意初始化，缺失的可能是 n
        int index = nums.length;
        for (int i = 0; i < length; i++) {
            if (nums[i] != i) {
                index = i;
                break;
            }
        }
        return index;
    }

    // 解法2：二分求解
    public int missingNumber2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int length = nums.length;
        int left = 0;
        int right = length - 1;
        while (left <= right) {
            // 注意加减法优先级高于位运算
```

```
            int mid = left + ((right - left) >> 1);
            if (mid == nums[mid]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return left;
    }

    // 解法3：数学求和公式
    public int missingNumber(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int sum1 = (1 + nums.length) * nums.length / 2;
        int sum2 = 0;
        for(int i = 0; i < nums.length; i++){
            sum2 += nums[i];
        }
        return sum1 - sum2;
    }
}
```

53.3 数组中数值和下标相等的元素

1. 描述

假设一个单调递增的数组里的每个元素都是整数且是唯一的。编写一个程序，找出数组中任意一个数值等于其下标的元素。

示例：

```
输入：[-3,-1,1,3,5]
输出：3
```

2. 思路

解法1：暴力求解

遍历数组，判断当前元素与其索引是否相等。

时间空间复杂度：$O(n)$，$O(1)$

解法2：二分查找

先利用二分找到与目标值相等的索引，即返回，否则判断大小关系更新左右边界的值

时间空间复杂度：$O(logn)$，$O(1)$

3. 代码

```
public class IntegerIdenticalToIndex {
    // 解法1：暴力求解
    public int integerIdenticalToIndex(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
```

```
        }
        int length = nums.length;
        int index = -1;
        for (int i = 0; i < length; i++) {
            if (nums[i] == i) {
                index = i;
                break;
            }
        }
        return index;
    }

    // 解法2：二分
    public int integerIdenticalToIndex2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int length = nums.length;
        int left = 0;
        int right = length - 1;
        while (left <= right) {
            // 注意加减法优先级高于位运算
            int mid = left + ((right - left) >> 1);
            if (nums[mid] == mid) {
                left = mid;
                break;
            } else if (nums[mid] > mid) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

54 二叉搜索树的第k大节点

1. 描述

找出二叉搜索树的第k大节点。

示例：

```
输入：root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出：4
```

2. 思路

解法1：中序遍历+list

由于是二叉搜索树，其中序遍历结果即是从小到大排序好的结果，将结果保存到list中，之间返回第k大个元素即可。

时间空间复杂度：$O(n)$，$O(n)$

解法2：优化

由于是二叉搜索树，其中序遍历结果即是从小到大排序好的结果，在递归的时候判断是第几个元素即可。

利用二叉搜索树的中序遍历的倒序（左中右----右中左），通过全局变量控制访问的次数，直到达到第k大元素

时间空间复杂度：$O(n)$，$O(logn)$

3. 代码

```java
import java.util.ArrayList;
import java.util.List;

public class KthNodeInBST {
    // 解法1：中序遍历+list
    public int kthLargest(TreeNode root, int k) {
        if (root == null || k < 0) {
            return -1;
        }
        List<TreeNode> list = new ArrayList<>();
        traverse(root, list);
        int length = list.size();
        if (k > length) {
            return -1;
        }
        return list.get(length - k).val;
    }

    public void traverse(TreeNode node, List<TreeNode> list) {
        if (node == null) {
            return;
        }
        traverse(node.left, list);
        list.add(node);
        traverse(node.right, list);
    }

    int count = 0;
    int res = -1;

    // 解法2：递归优化，不使用额外空间，第k大，则每次减一，直到为0
    public int kthLargest1(TreeNode root, int k) {
        if (root == null || k < 0) {
            return -1;
        }
        traverse1(root, k);
        return res;
    }

    public void traverse1(TreeNode node, int k) {
        if (node == null) {
            return;
```

```
        }
        traverse1(node.right, k);
        count++;
        if (k == count) {
            res = node.val;
            return;
        }
        traverse1(node.left, k);
    }
}
```

55 二叉树的深度

1. 描述

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

示例：

```
给定二叉树 [3,9,20,null,null,15,7]，
    3
   / \
  9  20
    /  \
   15   7
返回它的最大深度 3 。
```

2. 思路

解法1：层次遍历

注意每次需要将每层的节点poll

时间空间复杂度：$O(n), O(n)$

解法2：递归

当前节点的深度比子节点大1，并判断左右子树哪个最大取哪个

时间空间复杂度：$O(n), O(n)$

3. 代码

```
class MaxDepth {
    // 解法1：层次遍历
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int depth = 0;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            depth++;
            int size = queue.size();
            for (int i = 0; i < size; i++) {
```

```
                TreeNode node = queue.poll();
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
            }
        }
        return depth;
    }

    // 解法2：递归
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        return maxDepthCore(root);
    }

    public int maxDepthCore(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 计算当前节点深度，其等于左右节点深度+1
        int left = maxDepthCore(root.left) + 1;
        int right = maxDepthCore(root.right) + 1;
        // 最后比较哪个最大取哪个
        return left > right ? left : right;
    }
}
```

56.1 数组中数字出现的次数 I

1. 描述

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

示例：

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

2. 思路

注意：自身与自身异或是0，0与任何数字异或还是该数字，某个数字与其相反数进行与操作可得最低位为1的值（如4 （0100）& -4 =4）

解法1：哈希

利用哈希存储数组元素以及数组元素出现次数，最后找到出现一次的两个数字即可。

时间空间复杂度：$O(n), O(n)$

解法2：异或

- 可以看成“数组中只出现一次的一个数字”的延伸。如果所有数字都出现两次，只有一个数字是出现1次，那么可以通过把所有所有进行异或运算解决。因为x^x = 0。
- 但如果有两个数字出现一次，能否转化成上述问题？依旧把所有数字异或，最终的结果就是那两个出现一次的数字a,b异或的结果。因为a，b不相等，因此结果肯定不为0，那么结果的二进制表示至少有一位为1，找到那个1的位置p，然后我们就可以根据第p位是否为1将所有的数字分成两堆，这样我们就把所有数字分成两部分，且每部分都是只包含一个只出现一次的数字、其他数字出现两次，从而将问题转化为最开始我们讨论的“数组中只出现一次的一个数字”问题。

实例分析(以2,4,3,6,3,2,5,5为例)：

```
相关数字的二进制表示为：
2 = 0010        3 = 0011        4 = 0100
5 = 0101        6 = 0110

步骤1 全体异或：2^4^3^6^3^2^5^5 = 4^6 = 0010
步骤2 确定位置：对于0010，从右数的第二位为1，因此可以根据倒数第2位是否为1进行分组
步骤3 进行分组：分成[2,3,6,3,2]和[4,5,5]两组
步骤4 分组异或：2^3^6^3^2 = 6，4^5^5 = 4，因此结果为4，6。
```

时间空间复杂度：$O(n), O(1)$

3. 代码

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class NumberAppearOnce {
    // 解法1：哈希
    public int[] singleNumbers(int[] nums) {
        if (nums == null || nums.length == 0) {
            return new int[0];
        }
        int[] array = new int[2];
        Map<Integer, Integer> map = new HashMap<>();
        int length = nums.length;
        for (int i = 0; i < length; i++) {
            if (map.containsKey(nums[i])) {
                map.put(nums[i], -1);
            } else {
                map.put(nums[i], 1);
            }
        }
        Iterator<Map.Entry<Integer, Integer>> it =
map.entrySet().iterator();
        int index = 0;
        while (it.hasNext()) {
            Map.Entry<Integer, Integer> entry = it.next();
            if (entry.getValue() == 1) {
                array[index++] = entry.getKey();
            }
            if (index >= 2) {
                break;
            }
        }
        return array;
```

```
    }

    // 解法2：利用异或
    public int[] singleNumbers2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return new int[0];
        }
        int temp = nums[0];
        int length = nums.length;
        for (int i = 1; i < length; i++) {
            temp = temp ^ nums[i];
        }
        int[] result = new int[]{0, 0};
        // 快速获得最低位为1的值
        int index = temp & -temp;
        for (int i = 0; i < length; i++) {
            // 同位不是1
            if ((index & nums[i]) == 0){
                result[0] = result[0] ^ nums[i];
            } else{
                result[1] = result[1] ^ nums[i];
            }
        }
        return result;
    }
}
```

56.2 数组中数字出现的次数 II

1. 描述

在一个数组 `nums` 中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

示例：

```
输入：nums = [3,4,3,3]
输出：4
```

2. 思路

解法1：位运算

仍然利用位运算，只是不能用异或，把所有数字的位进行相加，除以3，所得等于0，则只出现一次的数字该位为0，否则是1

时间空间复杂度：$O(n), O(1)$

解法2：哈希

仍然可以使用哈希解决。

时间空间复杂度：$O(n), O(n)$

3. 代码

```
public class NumberAppearOneInThree {
```

```
    // 解法1：位运算
    public int singleNumber(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        int result = 0;
        int length = nums.length;
        int[] bit = new int[32];
        for (int i = 0; i < length; i++) {
            int temp = nums[i];
            for (int j = 31; j >= 0; j--) {
                if ((temp & 1) == 1) {
                    bit[j] += 1;
                }
                temp = temp >>> 1;
            }
        }
        for (int i = 0; i < 32; i++) {
            result = result + (int) ((bit[i] % 3) * Math.pow(2, (31 - i)));
        }
        return result;
    }
}
```

57.1 和为s的数字

1. 描述

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使它们的和为s。如果有多对和为s，输入任意一对即可。

示例：

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

2. 思路

解法1：双指针

双指针，判断前后指针对应的数字之和与target关系，等于直接返回，小于left++，大于right++。

时间空间复杂度：$O(n), O(1)$

3. 代码

```
public class SumS {
    // 解法1：双指针
    public int[] twoSum(int[] nums, int target) {
        if (nums == null || nums.length == 0 || target <= 0) {
            return new int[2];
        }
        int[] result = new int[2];
        int left = 0;
        int right = nums.length - 1;
        while (left <= right) {
```

```
            if (nums[left] + nums[right] == target) {
                result[0] = nums[left];
                result[1] = nums[right];
                break;
            } else if (nums[left] + nums[right] > target) {
                right--;
            } else if (nums[left] + nums[right] < target) {
                left++;
            }
        }
        return result;
    }
}
```

57.2 和为s的连续正数序列

1. 描述

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

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例：

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

2. 思路

解法1：暴力求解

计算当前节点与后面节点之和sum，并与target比较，相等之间返回，如果大于继续下一个节点，如果小于后面节点后移。

时间空间复杂度：$O(target\sqrt{target}), O(1)$

解法2：双指针

计算前后指针 left 与 right （初始化时是相邻的）之间的和 sum：

- 如果 sum<target ，说明指针 right 还可以向右拓展使得 sum 增大，此时指针 right 向右移动，即 right+=1
- t如果 sum>target 则说明以 left 为起点不存在一个 right 使得 sum=target ，此时要枚举下一个起点，指针 left 向右移动，即 left+=1
- 如果 sum=target 则说明我们找到了以 left 为起点得合法解 [left, right] 并添加到结果中，以 left 为起点的合法解最多只有一个，所以需要枚举下一个起点，指针 left 向右移动，即 left += 1

时间空间复杂度：$O(target), O(1)$

解法3：数学公式

根据当前索引 left 直接求解和为 target 的右边界 right ， right需要满足两个条件，

- 判别式 b^2-4ac 开根需要为整数。
- 最后的求根公式的分子需要为偶数，因为分母为 2

时间空间复杂度：$O(target), O(1)$

3. 代码

```java
import java.util.ArrayList;
import java.util.List;

public class ContinuousSequenceWithSum {
    // 解法1：暴力
    public int[][] findContinuousSequence(int target) {
        if (target <= 0) {
            return null;
        }
        int count = 0;
        int index = 1;
        List<int[]> list = new ArrayList<int[]>();
        // 由于至少两个数字，所以访问到目标值一半即可。
        while (index <= target / 2) {
            int sum = 0;
            int right = -1;
            int temp = index;
            while (true) {
                sum = sum + temp;
                if (sum == target) {
                    right = temp;
                    break;
                } else if (sum > target) {
                    break;
                } else {
                    temp++;
                }
            }
            if (right != -1 && right != index) {
                int[] res = new int[right - index + 1];
                for (int i = index; i <= right; i++) {
                    res[i - index] = i;
                }
                count++;
                list.add(res);
            }
            index++;
        }
        return list.toArray(new int[count][]);
    }

    // 解法2：双指针
    public int[][] findContinuousSequence2(int target) {
        if (target <= 0) {
            return null;
        }
        int count = 0;
        List<int[]> list = new ArrayList<int[]>();
        //
        for(int left=1, right=2; left<right;) {
            int sum = (left + right) * (right - left + 1) / 2;
            if (sum == target) {
                int[] res = new int[right - left + 1];
                for (int i = left; i <= right; i++) {
                    res[i - left] = i;
```

```
                }
                count++;
                list.add(res);
                left++;
            } else if (sum < target) {
                right++;
            } else {
                left++;
            }
        }
        return list.toArray(new int[count][]);
    }

    // 解法3：数学优化
    public int[][] findContinuousSequence3(int target) {
        int i = 1;
        double j = 2.0;
        List<int[]> res = new ArrayList<>();
        while (i < j) {
            j = (-1 + Math.sqrt(1 + 4 * (2 * target + (long) i * i - i))) /
2;
            if (i < j && j == (int) j) {
                int[] ans = new int[(int) j - i + 1];
                for (int k = i; k <= (int) j; k++)
                    ans[k - i] = k;
                res.add(ans);
            }
            i++;
        }
        return res.toArray(new int[0][]);
    }
}
```

58.1 翻转单词顺序

1. 描述

输入一个英文句子，翻转单词顺序，单词内字符不翻转，标点符号和普通字母一样处理。

示例：

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

2. 思路

- 利用trim删除前后空格，split分割，遇到空直接跳过（使用StringBuilder效率高一些）
- 利用trim删除前后空格，双指针，记录每个单词的首尾

3. 代码

```
public class ReverseWordsInSentence {
    // 实现1
    public String reverseWords(String s) {
        if (s == null || s.length() == 0) {
```

```
            return "";
        }
        // 删除首尾单词
        s = s.trim();
        String[] strs = s.split(" ");
        StringBuilder sb = new StringBuilder();
        for (int i = strs.length - 1; i >= 0; i--) {
            if (strs[i].length() != 0) {
                sb.append(strs[i] + " ");
            }
        }
        return sb.toString().trim();
    }

    // 实现2：双指针
    public String reverseWords2(String s) {
        s = s.trim(); // 删除首尾空格
        int j = s.length() - 1, i = j;
        StringBuilder res = new StringBuilder();
        while(i >= 0) {
            while(i >= 0 && s.charAt(i) != ' ') i--; // 搜索首个空格
            res.append(s.substring(i + 1, j + 1) + " "); // 添加单词
            while(i >= 0 && s.charAt(i) == ' ') i--; // 跳过单词间空格
            j = i; // j 指向下个单词的尾字符
        }
        return res.toString().trim(); // 转化为字符串并返回
    }
}
```

58.2 左旋转字符串

1. 描述

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。

示例：

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

2. 思路

由于是字符串类型，在Java中属于不可变类型，因此需要重新申请内存存储。如果是数组类型，除了下面方面，还可以通过反转方法求解。

- 暴力解法，先遍历后面字符串，再遍历前面部分
- 利用求余运算优化代码
- 利用substring进行切片，最后相加即可

3. 代码

```
public class LeftRotateString {
    // 实现1
    public String reverseLeftWords(String s, int n) {
```

```
        if (s == null || s.length() == 0 || n <= 0 || n > s.length()) {
            return "";
        }
        StringBuilder sb = new StringBuilder();
        int length = s.length();
        for (int i = n; i < length; i++) {
            sb.append(s.charAt(i));
        }
        for (int i = 0; i < n; i++) {
            sb.append(s.charAt(i));
        }

        return sb.toString();
    }

    // 实现2：利用求余运算，优化遍历操作
    public static String reverseLeftWords1(String s, int n) {
        if (s == null || s.length() == 0 || n <= 0 || n > s.length()) {
            return "";
        }
        StringBuilder res = new StringBuilder();
        int length = s.length();
        for (int i = n; i < n + length; i++)
            res.append(s.charAt(i % length));
        return res.toString();
    }

    // 实现3：切片操作
    public static String reverseLeftWords2(String s, int n) {
        if (s == null || s.length() == 0 || n <= 0 || n > s.length()) {
            return "";
        }
        return s.substring(n, s.length()) + s.substring(0, n);
    }
}
```

59.1 滑动窗口的最大值

1. 描述

给定一个数组和滑动窗口的大小，请找出所有滑动窗口的最大值。

示例：

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:
  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

2. 思路

解法1：暴力优化

暴力优化，记录上一个的滑动窗口的最大值（注意k=1的情况）

时间空间复杂度：$O(n^2)$，$O(1)$

解法2：双向有序队列

单调队列 注意：

- 队列按从大到小放入
- 如果首位值（即最大值）不在窗口区间，删除首位
- 如果新增的值小于队列尾部值，加到队列尾部
- 如果新增值大于队列尾部值，删除队列中比新增值小的值，把新增值加入到队列中
- 如果新增值大于队列中所有值，删除所有，然后把新增值放到队列首位，保证队列一直是从大到小

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

```java
import java.util.Deque;
import java.util.LinkedList;

public class MaxInSlidingWindow {
    // 解法1：暴力优化
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || nums.length == 0 || nums.length < k || k <= 0) {
            return new int[0];
        }
        if (k == 1) {
            return nums;
        }
        int length = nums.length;
        int len = length - k + 1;
        int[] array = new int[len];
        array[0] = nums[0];
        for (int j = 0; j < k; j++) {
            array[0] = (array[0] < nums[j]) ? nums[j] : array[0];
        }
        for (int i = 1; i < len; i++) {
            // 前一个最大值与前一个滑动窗口最左边值关系，
            if (array[i - 1] != nums[i - 1]) {
                if (array[i - 1] >= nums[i + k - 1]) {
                    array[i] = array[i - 1];
                } else {
                    array[i] = nums[i + k - 1];
                }
            } else {
                int tempLength = i + k;
                array[i] = nums[i];
                for (int j = i + 1; j < tempLength; j++) {
                    array[i] = (array[i] < nums[j]) ? nums[j] : array[i];
                }
            }
        }
        return array;
    }
```

```java

    // 解法2：双向有序队列
    public int[] maxSlidingWindow1(int[] nums, int k) {
        if(nums.length == 0 || k == 0) return new int[0];
        Deque<Integer> deque = new LinkedList<>();
        int[] res = new int[nums.length - k + 1];
        // 未形成窗口
        for(int i = 0; i < k; i++) {
            while(!deque.isEmpty() && deque.peekLast() < nums[i])
                deque.removeLast();
            deque.addLast(nums[i]);
        }
        res[0] = deque.peekFirst();
        // 形成窗口后
        for(int i = k; i < nums.length; i++) {
            // 首位值（即最大值）不在窗口区间，删除首位
            if(deque.peekFirst() == nums[i - k])
                deque.removeFirst();
            // 新增值大于队列尾部值，删除队列中比新增值小的值，把新增值加入到队列中
            while(!deque.isEmpty() && deque.peekLast() < nums[i])
                deque.removeLast();
            deque.addLast(nums[i]);
            res[i - k + 1] = deque.peekFirst();
        }
        return res;
    }
}
```

59.2 队列的最大值

1. 描述

请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

示例：

```
输入：
["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出: [null,null,null,2,1,2
```

2. 思路

- 维护一个常规队列与一个单调的双向队列
  - max：双向队列不为空，返回队首元素
  - push：常规队列直接加入，双向队列比较队尾元素与加入元素的大小，如果小与则弹出，直到不小于，将新元素加入
  - pop：直接返回常规队列的元素，如果其元素等于双向队列的队首，则把双向队列的队首弹出

3. 代码

```java
import java.util.Deque;
```

```java
import java.util.LinkedList;
import java.util.Queue;

public class MaxQueue {
    Queue<Integer> q;
    Deque<Integer> d;

    public MaxQueue() {
        q = new LinkedList<Integer>();
        d = new LinkedList<Integer>();
    }

    public int max_value() {
        if (d.isEmpty()) {
            return -1;
        }
        return d.peekFirst();
    }

    public void push_back(int value) {
        while (!d.isEmpty() && d.peekLast() < value) {
            d.pollLast();
        }
        d.offerLast(value);
        q.offer(value);
    }

    public int pop_front() {
        if (q.isEmpty()) {
            return -1;
        }
        int ans = q.poll();
        if (ans == d.peekFirst()) {
            d.pollFirst();
        }
        return ans;
    }
}
```

60 n个骰子的点数

1. 描述

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例：

```
输入: 1
输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]
```

2. 思路

- 动态规划（待理解）

3. 代码

```java
import java.util.Arrays;

public class DicesProbability {
    // 解法1：动态规划
    public double[] dicesProbability(int n) {
        if (n <= 0) {
            return new double[0];
        }
        double[] dp = new double[6];
        Arrays.fill(dp, 1.0 / 6.0);
        for (int i = 2; i <= n; i++) {
            double[] tmp = new double[5 * i + 1];
            for (int j = 0; j < dp.length; j++) {
                for (int k = 0; k < 6; k++) {
                    tmp[j + k] += dp[j] / 6.0;
                }
            }
            dp = tmp;
        }
        return dp;
    }
}

```

61 扑克牌中的顺子

1. 描述

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

示例:

```
输入: [1,2,3,4,5]
输出: True
```

2. 思路

- 考虑顺子的条件：除了0（大小王），不能有重复的数字、最大值与最小值的差值不能超过4
    - 第一种做法，利用map记录是否重复以及额外两个变量记录最大最小值
    - 第二种做法，数组排序，记录0的个数，然后判断相邻数字的差值，如果是0是重复，如果是1是连续，超过1则消耗0填充，最后判断0的个数是否小与0

3. 代码

```java
import java.util.Arrays;

public class ContinousCards {
    // 实现1
    public boolean isStraight(int[] nums) {
        if (nums == null || nums.length != 5) {
            return false;
```

```
        }
        Arrays.sort(nums);
        int zero = 0;
        int index = -1;
        for (int i = 0; i < 5; i++) {
            if (nums[i] == 0) {
                zero++;
            } else {
                if (index == -1) {
                    index = nums[i];
                } else {
                    int del = nums[i] - index;
                    if (del == 0) {
                        return false;
                    } else if (del == 1) {
                        index = nums[i];
                    } else {
                        zero = zero - (del - 1);
                        if (zero < 0) {
                            return false;
                        }
                        index = nums[i];
                    }
                }
            }
        }
        return true;
    }

    // 实现2
    public boolean isStraight(int[] nums) {
        if (nums == null || nums.length != 5) {
            return false;
        }
        Arrays.sort(nums);
        int zero = 0;
        for (int i = 0; i < 4; i++) {
            if (nums[i] == 0) {
                zero++;
            } else if(nums[i] == nums[i + 1]){
                return false;
            }
        }
        return nums[4] - nums[zero] < 5;
    }
}
```

62 圆圈中最后剩下的数字

1. 描述

0,1,⋯,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。

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

示例：

```
输入：n = 5，m = 3
输出：3
```

2. 思路

- 递归
- 动态规划
- 总结规律
  - 第一轮是 [0, 1, 2, 3, 4] ，所以是 [0, 1, 2, 3, 4] 这个数组的多个复制。这一轮 2 删除了。

    第二轮开始时，从 3 开始，所以是 [3, 4, 0, 1] 这个数组的多个复制。这一轮 0 删除了。

    第三轮开始时，从 1 开始，所以是 [1, 3, 4] 这个数组的多个复制。这一轮 4 删除了。

    第四轮开始时，还是从 1 开始，所以是 [1, 3] 这个数组的多个复制。这一轮 1 删除了。

    最后剩下的数字是 3。

    图中的绿色的线指的是新的一轮的开头是怎么指定的，每次都是固定地向前移位 mm 个位置。

    然后我们从最后剩下的 3 倒着看，我们可以反向推出这个数字在之前每个轮次的位置。

    最后剩下的 3 的下标是 0。

    第四轮反推，补上 m 个位置，然后模上当时的数组大小 2，位置是(0 + 3) % 2 = 1。

    第三轮反推，补上 m 个位置，然后模上当时的数组大小 3，位置是(1 + 3) % 3 = 1。

    第二轮反推，补上 m 个位置，然后模上当时的数组大小 4，位置是(1 + 3) % 4 = 0。

    第一轮反推，补上 m 个位置，然后模上当时的数组大小 5，位置是(0 + 3) % 5 = 3。

    所以最终剩下的数字的下标就是3。因为数组是从0开始的，所以最终的答案就是3。

    总结一下反推的过程，就是 (当前index + m) % 上一轮剩余数字的个数。

3. 代码

```java
public class LastNumberInCircle {
    public int lastRemaining(int n, int m) {
        if (n <= 0 || m <= 0) {
            return -1;
        }
        int x = 0;
        for (int i = 2; i <= n; i++) {
            x = (x + m) % i;
        }
        return x;
    }
}
```

63 股票的最大利润

1. 描述

假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少?

示例：

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格。
```

2. 思路

动态规划以及可优化（利用单变量）

时间空间复杂度：$O(n), O(n)/O(1)$

3. 代码

```java
public class MaxProfit {
    // 动态规划
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 1) {
            return 0;
        }
        int[] result = new int[prices.length];
        int current = prices[0];
        for (int i = 1; i < prices.length; i++) {
            if ((prices[i] - current) > result[i - 1]) {
                result[i] = prices[i] - current;
            } else {
                result[i] = result[i - 1];
            }
            if (prices[i] < current) {
                current = prices[i];
            }
        }
        return result[prices.length - 1];
    }

    // 动态规划优化
    public int maxProfit1(int[] prices) {
        if (prices == null || prices.length < 1) {
            return 0;
        }
        int result = 0;
        int current = prices[0];
        for (int i = 1; i < prices.length; i++) {
            // 更新结果最大值
            if ((prices[i] - current) > result) {
                result = prices[i] - current;
            }
            // 更新当前最小值
            if (prices[i] < current) {
                current = prices[i];
            }
        }
        return result;
    }
}
```

64 求 1+2+...+n

1. 描述

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

示例:

```
输入: n = 3
输出: 6
```

2. 思路

- 递归实现，但是递归有结束条件，想办法利用逻辑操作判定结束条件

3. 代码

```
public class SumNums {
    // 递归运算
    public int sumNums(int n) {
        boolean flag = (n > 0) && (n += sumNums(n - 1)) > 0;
        return n;
    }
}
```

65 不用加减乘除做加法

1. 描述

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

示例:

```
输入: a = 1, b = 1
输出: 2
```

2. 思路

- 位运算

设两数字的二进制形式 a, b ， 其求和 s = a + b ， a(i) 代表 a 的二进制第 i 位，则分为以下四种情况：

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

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

- n=a⊕b 非进位和：异或运算
- c=a&b<<1 进位：与运算+左移一位

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

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

3. 代码

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

66 构建乘积数组

1. 描述

给定一个数组 A[0,1,...,n-1]，请构建一个数组 B[0,1,...,n-1]，其中 B[i] 的值是数组 A 中除了下标 i 以外的元素的积, 即 B[i]=A[0]×A[1]×...×A[i-1]×A[i+1]×...×A[n-1]。不能使用除法。

示例：

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

2. 思路

解法1：暴力求解

将要删除节点的值更新后其后面节点的值，然后再将删除节点的下一个指针指向下一个节点的下一个节点。

时间空间复杂度：$O(n^2)$，$O(1)$

解法2：模拟二维数组乘积

将所要乘积的数组行列对应，分为上下三角两部分，两者相乘即可。

时间空间复杂度：$O(n)$，$O(n)$

解法3：解法2优化

先计算下三角保存到结果数组，计算上三角时可使用常量保存中间结果再与结果数组相乘

时间空间复杂度：$O(n)$，$O(1)$

$$B[i] = A[0]\times A[1]\times\ldots\times A[i-1]\times A[i+1]\times\ldots\times A[n-1]\times A[n]$$

⬇ 列表格

B[0] =	1	A[1]	A[2]	···	A[n-1]	A[n]
B[1] =	A[0]	1	A[2]	···	A[n-1]	A[n]
B[2] =	A[0]	A[1]	1	···	A[n-1]	A[n]
... =	···	···	···	···	···	···
B[n-1] =	A[0]	A[1]	A[2]	···	1	A[n]
B[n] =	A[0]	A[1]	A[2]	···	A[n-1]	1

⬇ 解法

通过两轮循环，分别计算 下三角 和 上三角 的乘积，
即可在不使用除法的前提下获得所需结果。

3. 代码

```java
public class ProductExceptSelf {
    // 解法1：暴力（超时）
    public int[] productExceptSelf(int[] nums) {
        int[] result = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            result[i] = 1;
            for (int j = 0; j < nums.length; j++) {
                if (i != j) {
                    result[i] *= nums[j];
                }
            }
        }
        return result;
    }

    // 解法2：模拟二维数组乘积
    public int[] productExceptSelf1(int[] nums) {
        int[] result1 = new int[nums.length];
        int[] result2 = new int[nums.length];

        for (int i = 0; i < nums.length; i++) {
            result1[i] = result2[i] = 1;
        }
        for (int i = 1; i < nums.length; i++) {
            result1[i] = result1[i - 1] * nums[i - 1];
        }
        for (int i = nums.length - 2; i >= 0; i--) {
            result2[i] = result2[i + 1] * nums[i + 1];
        }
        for (int i = 0; i < nums.length; i++) {
```

```
            result1[i] *= result2[i];
        }
        return result1;
    }

    // 解法3：模拟二维数组乘积优化
    public int[] productExceptSelf2(int[] nums) {
        int[] result = new int[nums.length];
        result[0] = 1;
        // 计算下三角
        for (int i = 1; i < nums.length; i++) {
            result[i] = result[i - 1] * nums[i - 1];
        }
        // 再计算上三角
        int temp = 1;
        for (int i = nums.length - 2; i >= 0; i--) {
            temp *= nums[i + 1];
            result[i] *= temp;
        }
        return result;
    }
}
```

67 把字符串转换成整数

1. 描述

写一个函数 StrToInt，实现把字符串转换成整数这个功能。不能使用 atoi 或者其他类似的库函数。

示例：

```
输入：“42”
输出：42
```

2. 思路

- 首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。
- 当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。
- 该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。
- 注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。
- 在任何情况下，若函数不能进行有效的转换时，请返回 0。
- 说明：
    - 假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−2^31，2^31 − 1]。如果数值超过这个范围，请返回 INT_MAX (2^31 − 1) 或 INT_MIN (−2^31) 。

3. 代码

```
public class MyAtoi {
```

```java
    // 使用库函数去除空格
    public int myAtoi(String str) {
        if (str == null || str.length() == 0) {
            return 0;
        }
        // 去除空格
        str = str.trim();
        if (str.length() == 0) {
            return 0;
        }
        int len = str.length();
        long temp = 0;
        boolean flag = true;
        int index = 0;
        // 判断正负，默认是正数
        if (str.charAt(0) == '+' || str.charAt(0) == '-') {
            flag = str.charAt(0) == '+' ? true : false;
            index++;
        }
        for (int i = index; i < len; i++) {
            if (str.charAt(i) >= '0' && str.charAt(i) <= '9') {
                temp = temp * 10 + (int) (str.charAt(i) - '0');
                // 判断是否溢出
                if (flag && temp > Integer.MAX_VALUE) {
                    return Integer.MAX_VALUE;
                } else if (-1 * temp < Integer.MIN_VALUE) {
                    return Integer.MIN_VALUE;
                }
            } else break;
        }
        if (flag)
            return (int) temp;
        else
            return -(int) temp;
    }

    // 不使用trim
    public int myAtoi1(String str) {
        if (str == null || str.length() == 0) {
            return 0;
        }
        int len = str.length();
        long temp = 0;
        boolean flag = true;
        int index = 0;
        while (index < len) {
            if (str.charAt(index) != ' ') {
                break;
            }
            index++;
        }
        if (index == len) {
            return (int) temp;
        }
        if (str.charAt(index) == '+' || str.charAt(index) == '-') {
            flag = str.charAt(0) == '+' ? true : false;
            index++;
        }
```

```
        for (int i = index; i < len; i++) {
            if (str.charAt(i) >= '0' && str.charAt(i) <= '9') {
                temp = temp * 10 + (int) (str.charAt(i) - '0');
                if (flag && temp > Integer.MAX_VALUE) {
                    return Integer.MAX_VALUE;
                } else if (-1 * temp < Integer.MIN_VALUE) {
                    return Integer.MIN_VALUE;
                }
            } else break;
        }
        if (flag)
            return (int) temp;
        else
            return -(int) temp;
    }

    public static void main(String[] args) {
//        System.out.println(strToInt("42"));
//        System.out.println(strToInt("      -42"));
//        System.out.println(strToInt("4193 with words"));
//        System.out.println(strToInt("words and 987"));
//        System.out.println(strToInt("21111111111111111111112187"));
//        System.out.println(strToInt("-312321434523452335345 4"));
        System.out.println(strToInt2("211111asdas"));
        System.out.println(strToInt2("-3+4-"));
    }
}
```

68 树中两个节点的最低公共祖先

1. 描述

输入一棵树的根节点，输入两个被观察节点，求这两个节点的最低(最近)公共祖先。

示例:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

2. 思路

对于树没有做明确说明，所以原书中就对树的可能情况做了假设，然后就衍生出多种思路。

- 如果是二叉搜索树
  - 遍历找到比第一个节点大，比第二个节点小的节点即可
- 如果是父子间有双向指针的树
  - 由下往上看，转化为找两个链表的第一个公共节点问题
- 如果只是一个包含父到子的指针的普通二叉树
  - 终止条件:
    - 当越过叶节点，则直接返回 null;
    - 当 root 等于 p, q，则直接返回 root;
  - 递推工作:

        - 开启递归左子节点，返回值记为 left；
        - 开启递归右子节点，返回值记为 right；
    - 返回值： 根据 left 和 right，可展开为四种情况；
        - 当 left 和 right 同时为空： 说明 root 的左 / 右子树中都不包含 p,q，返回 null；
        - 当 left 和 right 同时不为空： 说明 p, q 分列在 root 的 异侧 （分别在 左 / 右子树），因此 root 为最近公共祖先，返回 root；
        - 当 left 为空 ， right 不为空： p,q 都不在 root 的左子树中，直接返回 right 。具体可分为两种情况：
            - p,q 其中一个在 root 的 右子树 中，此时 right 指向 p（假设为 p ）；
            - p,q 两节点都在 root 的 右子树 中，此时的 right 指向 最近公共祖先节点 ；
        - 当 left 不为空 ， right 为空： p,q 都不在 root 的右子树中，直接返回 left， 与情况 3. 同理；
        - 观察发现， 情况 1. 可合并至 3. 和 4. 内，详见代码。

时间空间复杂度：$O(n)$，$O(n)$

3. 代码

二叉搜索树：

```java
// 二叉搜索树
public class LowestCommonAncestor {
    public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || p == null || q == null) {
            return null;
        }
        if (p.val > q.val)
            return lowestCommonAncestorCore(root, q, p);
        else
            return lowestCommonAncestorCore(root, p, q);
    }

    public static TreeNode lowestCommonAncestorCore(TreeNode root, TreeNode p, TreeNode q) {
        if (root.val >= p.val && root.val <= q.val) {
            return root;
        } else if (root.val > p.val && root.val > q.val) {
            return lowestCommonAncestorCore(root.left, p, q);
        } else {
            return lowestCommonAncestorCore(root.right, p, q);
        }
    }
}
```

普通二叉树：

```java
// 普通搜索树
public class LowestCommonAncestor2 {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) return root;
        // 递归遍历左子树，只要在左子树中找到了p或q，则先找到谁就返回谁
        TreeNode left = lowestCommonAncestor(root.left, p, q);
```

```
7
8           // 递归遍历右子树，只要在左子树中找到了p或q，则先找到谁就返回谁
9           TreeNode right = lowestCommonAncestor(root.right, p, q);
10          if (left == null && right == null) return null; // 1.
11
12          // 如果在左子树中 p和 q都找不到，则 p和 q一定都在右子树中，右子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
13          if (left == null) return right; // 3.
14
15          // 如果 left不为空，在左子树中有找到节点（p或q），这时候要再判断一下右子树中的情况，如果在右子树中，p和q都找不到，则 p和q一定都在左子树中，左子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
16          if (right == null) return left; // 4.
17
18          // 否则，当 left和 right均不为空时，说明 p、q节点分别在 root异侧，最近公共祖先即为 root
19          return root; // 2. if(left != null and right != null)
20      }
21  }
```