

# 剑指 Offer 算法题解-Java实现

---

## 前言

---

题目来自《何海涛. 剑指 Offer[M]. 电子工业出版社, 2012.》，刷题网站推荐：

- [牛客网](#)
- [Leetcode](#)

## 分类

---

### 数组与矩阵

- [3. 数组中重复的数字](#)
- [4. 二维数组中的查找](#)
- [5. 替换空格](#)
- [29. 顺时针打印矩阵](#)
- [50. 第一个只出现一次的字符位置](#)

### 栈队列堆

---

- [9. 用两个栈实现队列](#)
- [30. 包含 min 函数的栈](#)
- [31. 栈的压入、弹出序列](#)
- [40. 最小的 K 个数](#)
- [41.1 数据流中的中位数](#)
- [41.2 字符流中第一个不重复的字符](#)
- [59. 滑动窗口的最大值](#)

### 双指针

---

- [57.1 和为 S 的两个数字](#)
- [57.2 和为 S 的连续正数序列](#)
- [58.1 翻转单词顺序列](#)
- [58.2 左旋转字符串](#)

### 链表

---

- [6. 从尾到头打印链表](#)
- [18.1 在 O\(1\) 时间内删除链表节点](#)
- [18.2 删除链表中重复的结点](#)
- [22. 链表中倒数第 K 个结点](#)
- [23. 链表中环的入口结点](#)
- [24. 反转链表](#)
- [25. 合并两个排序的链表](#)

- [35. 复杂链表的复制](#)
- [52. 两个链表的第一个公共结点](#)

## 树

---

- [7. 重建二叉树](#)
- [8. 二叉树的下一个结点](#)
- [26. 树的子结构](#)
- [27. 二叉树的镜像](#)
- [28. 对称的二叉树](#)
- [32.1 从上往下打印二叉树](#)
- [32.2 把二叉树打印成多行](#)
- [32.3 按之字形顺序打印二叉树](#)
- [33. 二叉搜索树的后序遍历序列](#)
- [34. 二叉树中和为某一值的路径](#)
- [36. 二叉搜索树与双向链表](#)
- [37. 序列化二叉树](#)
- [54. 二叉查找树的第 K 个结点](#)
- [55.1 二叉树的深度](#)
- [55.2 平衡二叉树](#)
- [68. 树中两个节点的最低公共祖先](#)

## 贪心思想

---

- [14. 剪绳子](#)
- [63. 股票的最大利润](#)

## 二分查找

---

- [11. 旋转数组的最小数字](#)
- [53. 数字在排序数组中出现的次数](#)

## 分治

---

- [16. 数值的整数次方](#)

## 搜索

---

- [12. 矩阵中的路径](#)
- [13. 机器人的运动范围](#)
- [38. 字符串的排列](#)

## 排序

---

- [21. 调整数组顺序使奇数位于偶数前面](#)
- [45. 把数组排成最小的数](#)
- [51. 数组中的逆序对](#)

## 动态规划

---

- [10.1 斐波那契数列](#)
- [10.2 矩形覆盖](#)
- [10.3 跳台阶](#)
- [10.4 变态跳台阶](#)
- [42. 连续子数组的最大和](#)
- [47. 礼物的最大价值](#)
- [48. 最长不含重复字符的子字符串](#)
- [49. 丑数](#)
- [60. n 个骰子的点数](#)
- [66. 构建乘积数组](#)

## 数学

---

- [39. 数组中出现次数超过一半的数字](#)
- [62. 圆圈中最后剩下的数](#)
- [43. 从 1 到 n 整数中 1 出现的次数](#)

## 位运算

---

- [15. 二进制中 1 的个数](#)
- [56. 数组中只出现一次的数字](#)

## 其它

---

- [17. 打印从 1 到最大的 n 位数](#)
- [19. 正则表达式匹配](#)
- [20. 表示数值的字符串](#)
- [44. 数字序列中的某一位数字](#)
- [46. 把数字翻译成字符串](#)
- [61. 扑克牌顺子](#)
- [64. 求  \$1+2+3+...+n\$](#)
- [65. 不用加减乘除做加法](#)
- [67. 把字符串转换成整数](#)

## 3. 数组中重复的数字

---

### 题目链接

---

[牛客网](#)

## 题目描述

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

Input:

{2, 3, 1, 0, 2, 5}

Output:

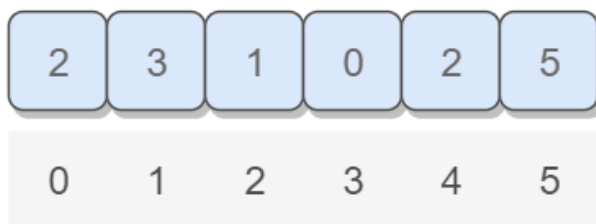
2

## 解题思路

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

对于这种数组元素在  $[0, n-1]$  范围内的问题，可以将值为  $i$  的元素调整到第  $i$  个位置上进行求解。在调整过程中，如果第  $i$  位置上已经有一个值为  $i$  的元素，就可以知道  $i$  值重复。

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



① start

 CyC2018

```
public int duplicate(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        while (nums[i] != i) {
            if (nums[i] == nums[nums[i]]) {
                return nums[i];
            }
            swap(nums, i, nums[i]);
        }
        swap(nums, i, nums[i]);
    }
    return -1;
}
```

```
private void swap(int[] nums, int i, int j) {  
    int t = nums[i];  
    nums[i] = nums[j];  
    nums[j] = t;  
}
```

## 4. 二维数组中的查找

### 题目链接

[牛客网](#)

### 题目描述

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

Consider the following matrix:

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

Given target = 5, return true.

Given target = 20, return false.

### 解题思路

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

该二维数组中的一个数，小于它的数一定在其左边，大于它的数一定在其下边。因此，从右上角开始查找，就可以根据 target 和当前元素的大小关系来快速地缩小查找区间，每次减少一行或者一列的元素。当前元素的查找区间为左下角的所有元素。

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

15 < 16

①

 CyC2018

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

## 5. 替换空格

### 题目链接

[生客网](#)

## 题目描述

将一个字符串中的空格替换成 "%20"。

Input:  
"A B"

Output:  
"A%20B"

## 解题思路

- ① 在字符串尾部填充任意字符，使得字符串的长度等于替换之后的长度。因为一个空格要替换成三个字符（%20），所以当遍历到一个空格时，需要在尾部填充两个任意字符。
- ② 令 P1 指向字符串原来的末尾位置，P2 指向字符串现在的末尾位置。P1 和 P2 从后向前遍历，当 P1 遍历到一个空格时，就需要令 P2 指向的位置依次填充 02%（注意是逆序的），否则就填充上 P1 指向字符的值。从后向前遍是为了在改变 P2 所指向的内容时，不会影响到 P1 遍历原来字符串的内容。
- ③ 当 P2 遇到 P1 时（P2 <= P1），或者遍历结束（P1 < 0），退出。



①

 CyC2018

```
public String replaceSpace(StringBuffer str) {  
    int P1 = str.length() - 1;  
    for (int i = 0; i <= P1; i++)  
        if (str.charAt(i) == ' ')  
            str.append("  ");  
  
    int P2 = str.length() - 1;  
    while (P1 >= 0 && P2 > P1) {  
        char c = str.charAt(P1--);
```

```

        if (c == ' ') {
            str.setCharAt(P2--, '0');
            str.setCharAt(P2--, '2');
            str.setCharAt(P2--, '%');
        } else {
            str.setCharAt(P2--, c);
        }
    }
    return str.toString();
}

```

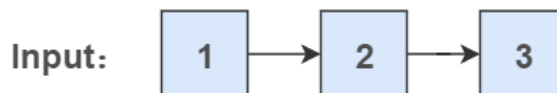
## 6. 从尾到头打印链表

### 题目链接

[牛客网](#)

### 题目描述

从尾到头反过来打印出每个结点的值。



Output: 3, 2, 1

CyC2018

### 解题思路

#### 1. 使用递归

要逆序打印链表 1->2->3 (3,2,1)，可以先逆序打印链表 2->3(3,2)，最后再打印第一个节点 1。而链表 2->3 可以看成一个新的链表，要逆序打印该链表可以继续使用求解函数，也就是在求解函数中调用自己，这就是递归函数。

```

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
    ArrayList<Integer> ret = new ArrayList<>();
    if (listNode != null) {
        ret.addAll(printListFromTailToHead(listNode.next));
        ret.add(listNode.val);
    }
    return ret;
}

```

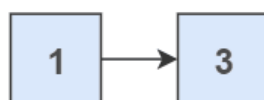


## 2. 使用头插法

头插法顾名思义是将节点插入到头部：在遍历原始链表时，将当前节点插入新链表的头部，使其成为第一个节点。

链表的操作需要维护后继关系，例如在某个节点 node1 之后插入一个节点 node2，我们可以通过修改后继关系来实现：

```
node3 = node1.next;  
node2.next = node3;  
node1.next = node2;
```



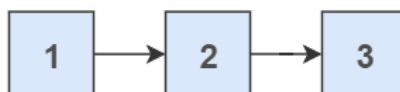
①

 CyC2018

为了能将一个节点插入头部，我们引入了一个叫头结点的辅助节点，该节点不存储值，只是为了方便进行插入操作。不要将头结点与第一个节点混起来，第一个节点是链表中第一个真正存储值的节点。

①

Old List:



New List:



 CyC2018

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
    // 头插法构建逆序链表  
    ListNode head = new ListNode(-1);  
    while (listNode != null) {
```

```

        ListNode memo = listNode.next;
        listNode.next = head.next;
        head.next = listNode;
        listNode = memo;
    }
    // 构建 ArrayList
    ArrayList<Integer> ret = new ArrayList<>();
    head = head.next;
    while (head != null) {
        ret.add(head.val);
        head = head.next;
    }
    return ret;
}

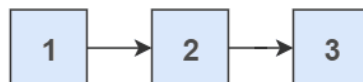
```

### 3. 使用栈

栈具有后进先出的特点，在遍历链表时将值按顺序放入栈中，最后出栈的顺序即为逆序。

①

List:



Stack:



Output:

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
    Stack<Integer> stack = new Stack<>();  
    while (listNode != null) {  
        stack.add(listNode.val);  
        listNode = listNode.next;  
    }  
    ArrayList<Integer> ret = new ArrayList<>();  
    while (!stack.isEmpty())  
        ret.add(stack.pop());  
    return ret;  
}
```

## 7. 重建二叉树

### 题目链接

[牛客网](#)

### 题目描述

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

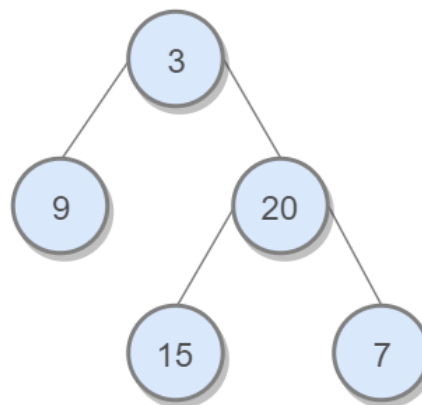
preorder:



inorder:



Output:



## 解题思路

前序遍历的第一个值为根节点的值，使用这个值将中序遍历结果分成两部分，左部分为树的左子树中序遍历结果，右部分为树的右子树中序遍历的结果。然后分别对左右子树递归地求解。

②

preorder:



inorder:



 CyC2018

```
// 缓存中序遍历数组每个值对应的索引
private Map<Integer, Integer> indexForInOrders = new HashMap<>();

public TreeNode reConstructBinaryTree(int[] pre, int[] in) {
    for (int i = 0; i < in.length; i++)
        indexForInOrders.put(in[i], i);
    return reConstructBinaryTree(pre, 0, pre.length - 1, 0);
}

private TreeNode reConstructBinaryTree(int[] pre, int preL, int preR, int inL) {
    if (preL > preR)
        return null;
    TreeNode root = new TreeNode(pre[preL]);
    int inIndex = indexForInOrders.get(root.val);
    int leftTreeSize = inIndex - inL;
    root.left = reConstructBinaryTree(pre, preL + 1, preL + leftTreeSize, inL);
    root.right = reConstructBinaryTree(pre, preL + leftTreeSize + 1, preR, inL +
leftTreeSize + 1);
    return root;
}
```

## 8. 二叉树的下一个结点

### 题目链接

[牛客网](#)

### 题目描述

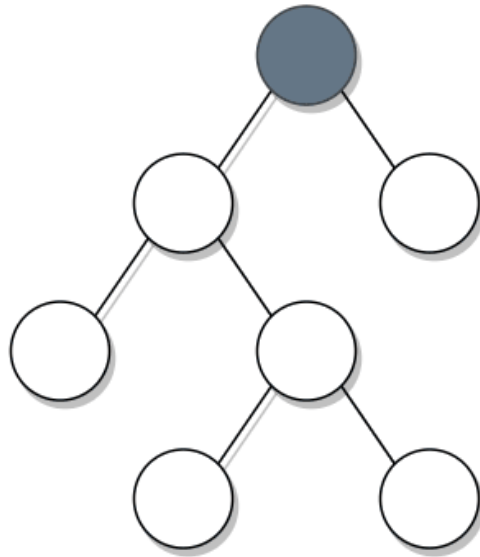
给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

```
public class TreeLinkNode {  
  
    int val;  
    TreeLinkNode left = null;  
    TreeLinkNode right = null;  
    TreeLinkNode next = null; // 指向父结点的指针  
  
    TreeLinkNode(int val) {  
        this.val = val;  
    }  
}
```

### 解题思路

我们先来回顾一下中序遍历的过程：先遍历树的左子树，再遍历根节点，最后再遍历右子树。所以最左节点是中序遍历的第一个节点。

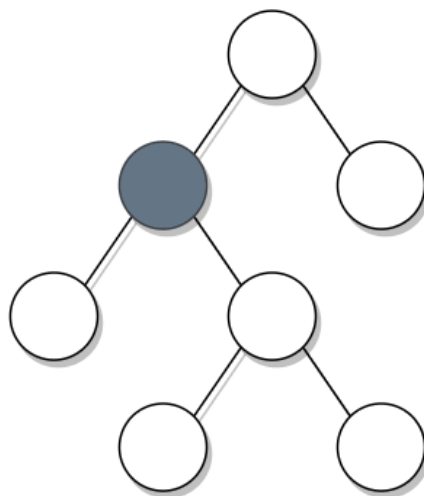
```
void traverse(TreeNode root) {  
    if (root == null) return;  
    traverse(root.left);  
    visit(root);  
    traverse(root.right);  
}
```



 CyC2018

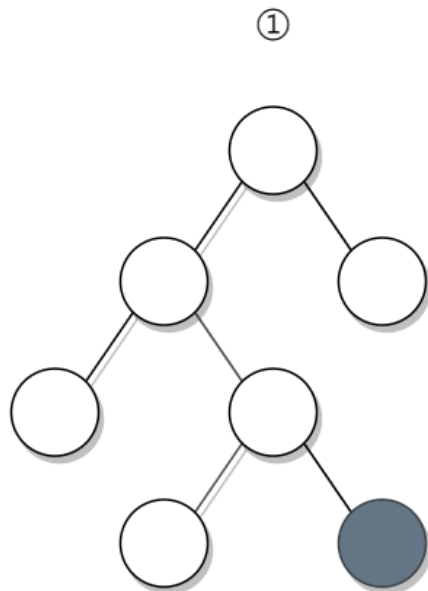
① 如果一个节点的右子树不为空，那么该节点的下一个节点是右子树的最左节点；

①



 CyC2018

② 否则，向上找第一个左链接指向的树包含该节点的祖先节点。



CyC2018

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

## 9. 用两个栈实现队列

### 题目链接

[牛客网](#)

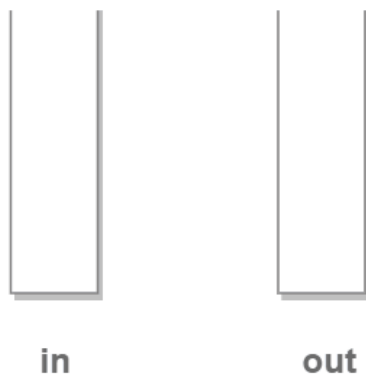
### 题目描述

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。

### 解题思路

in 栈用来处理入栈（push）操作，out 栈用来处理出栈（pop）操作。一个元素进入 in 栈之后，出栈的顺序被反转。当元素要出栈时，需要先进入 out 栈，此时元素出栈顺序再一次被反转，因此出栈顺序就和最开始入栈顺序是相同的，先进入的元素先退出，这就是队列的顺序。

①



 CyC2018

```
Stack<Integer> in = new Stack<Integer>();
Stack<Integer> out = new Stack<Integer>();

public void push(int node) {
    in.push(node);
}

public int pop() throws Exception {
    if (out.isEmpty())
        while (!in.isEmpty())
            out.push(in.pop());

    if (out.isEmpty())
        throw new Exception("queue is empty");

    return out.pop();
}
```

## 10.1 斐波那契数列

### 题目链接

[生客网](#)



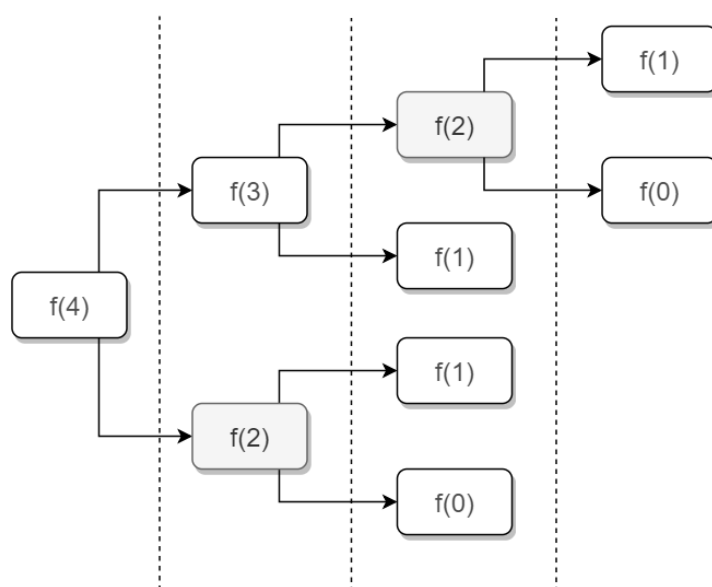
## 题目描述

求斐波那契数列的第  $n$  项， $n \leq 39$ 。

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

## 解题思路

如果使用递归求解，会重复计算一些子问题。例如，计算  $f(4)$  需要计算  $f(3)$  和  $f(2)$ ，计算  $f(3)$  需要计算  $f(2)$  和  $f(1)$ ，可以看到  $f(2)$  被重复计算了。



CyC2018

递归是将一个问题划分成多个子问题求解，动态规划也是如此，但是动态规划会把子问题的解缓存起来，从而避免重复求解子问题。

```
public int Fibonacci(int n) {  
    if (n <= 1)  
        return n;  
    int[] fib = new int[n + 1];  
    fib[1] = 1;  
    for (int i = 2; i <= n; i++)  
        fib[i] = fib[i - 1] + fib[i - 2];  
    return fib[n];  
}
```

考虑到第  $i$  项只与第  $i-1$  和第  $i-2$  项有关，因此只需要存储前两项的值就能求解第  $i$  项，从而将空间复杂度由  $O(N)$  降低为  $O(1)$ 。

```
public int Fibonacci(int n) {
    if (n <= 1)
        return n;
    int pre2 = 0, pre1 = 1;
    int fib = 0;
    for (int i = 2; i <= n; i++) {
        fib = pre2 + pre1;
        pre2 = pre1;
        pre1 = fib;
    }
    return fib;
}
```

由于待求解的  $n$  小于 40，因此可以将前 40 项的结果先进行计算，之后就能以  $O(1)$  时间复杂度得到第  $n$  项的值。

```
public class Solution {

    private int[] fib = new int[40];

    public Solution() {
        fib[1] = 1;
        for (int i = 2; i < fib.length; i++)
            fib[i] = fib[i - 1] + fib[i - 2];
    }

    public int Fibonacci(int n) {
        return fib[n];
    }
}
```

## 10.2 矩形覆盖

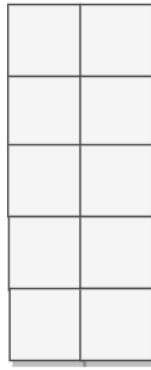
### 题目链接

[牛客网](#)

### 题目描述

我们可以用  $2 \times 1$  的小矩形横着或者竖着去覆盖更大的矩形。请问用  $n$  个  $2 \times 1$  的小矩形无重叠地覆盖一个  $2 \times n$  的大矩形，总共有多少种方法？

①



 CyC2018

## 解题思路

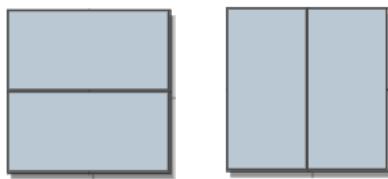
当  $n$  为 1 时，只有一种覆盖方法：



$n=1$

 CyC2018

当  $n$  为 2 时，有两种覆盖方法：



$n=2$

 CyC2018

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

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

```
public int rectCover(int n) {  
    if (n <= 2)  
        return n;  
    int pre2 = 1, pre1 = 2;  
    int result = 0;  
    for (int i = 3; i <= n; i++) {  
        result = pre2 + pre1;  
        pre2 = pre1;  
        pre1 = result;  
    }  
    return result;  
}
```

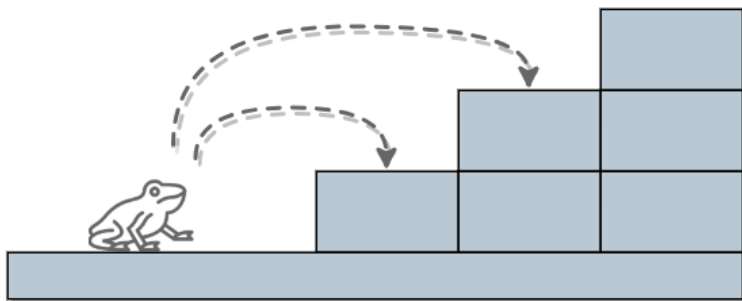
## 10.3 跳台阶

### 题目链接

[牛客网](#)

### 题目描述

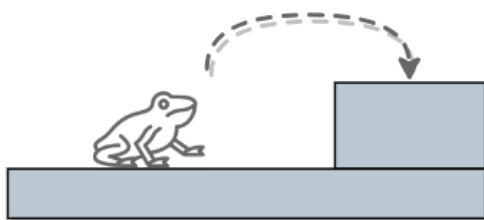
一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个  $n$  级的台阶总共有多少种跳法。



CyC2018

### 解题思路

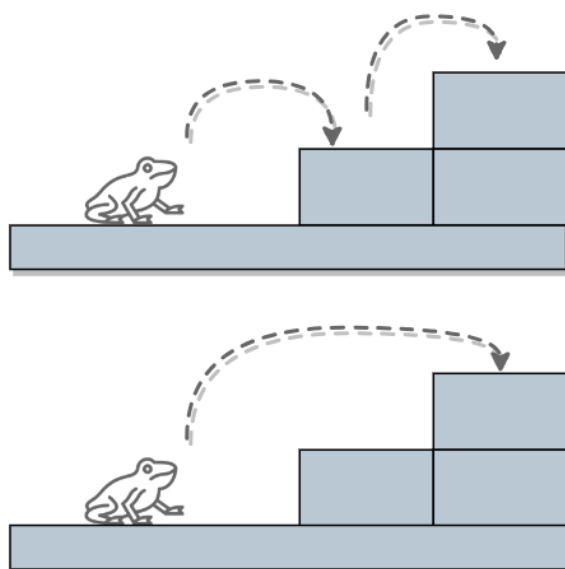
当  $n = 1$  时，只有一种跳法：



$n=1$

CyC2018

当  $n = 2$  时，有两种跳法：



$n=2$

 CyC2018

跳  $n$  阶台阶，可以先跳 1 阶台阶，再跳  $n-1$  阶台阶；或者先跳 2 阶台阶，再跳  $n-2$  阶台阶。而  $n-1$  和  $n-2$  阶台阶的跳法可以看成子问题，该问题的递推公式为：

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

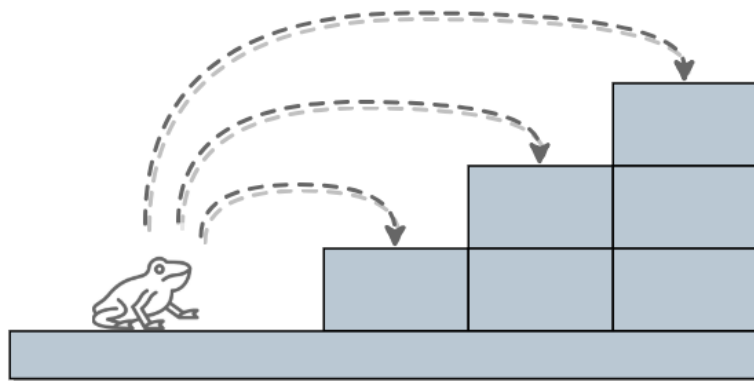
```
public int JumpFloor(int n) {  
    if (n <= 2)  
        return n;  
    int pre2 = 1, pre1 = 2;  
    int result = 0;  
    for (int i = 2; i < n; i++) {  
        result = pre2 + pre1;  
        pre2 = pre1;  
        pre1 = result;  
    }  
    return result;  
}
```

## 10.4 变态跳台阶

题目链接

## 题目描述

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



 CyC2018

## 解题思路

### 动态规划

```
public int jumpFloorII(int target) {  
    int[] dp = new int[target];  
    Arrays.fill(dp, 1);  
    for (int i = 1; i < target; i++)  
        for (int j = 0; j < i; j++)  
            dp[i] += dp[j];  
    return dp[target - 1];  
}
```

### 数学推导

跳上 n-1 级台阶，可以从 n-2 级跳 1 级上去，也可以从 n-3 级跳 2 级上去...，那么

$$f(n-1) = f(n-2) + f(n-3) + \dots + f(0)$$

同样，跳上 n 级台阶，可以从 n-1 级跳 1 级上去，也可以从 n-2 级跳 2 级上去...，那么

$$f(n) = f(n-1) + f(n-2) + \dots + f(0)$$

综上可得

$$f(n) - f(n-1) = f(n-1)$$

即

$$f(n) = 2 * f(n-1)$$

所以  $f(n)$  是一个等比数列

```
public int JumpFloorII(int target) {  
    return (int) Math.pow(2, target - 1);  
}
```

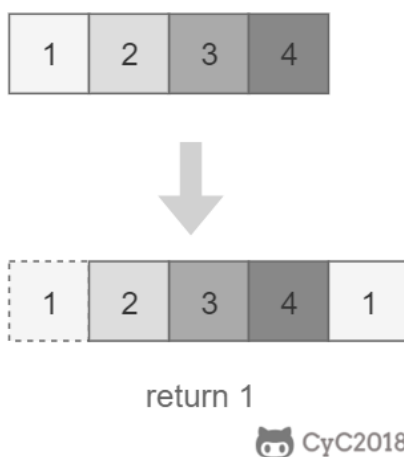
## 11. 旋转数组的最小数字

### 题目链接

[生客网](#)

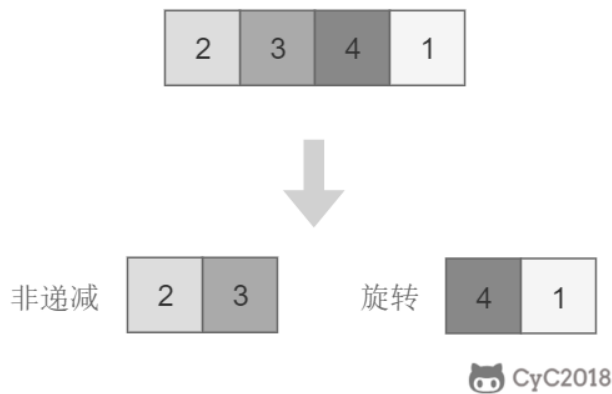
### 题目描述

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。输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。



### 解题思路

将旋转数组对半分可以得到一个包含最小元素的新旋转数组，以及一个非递减排序的数组。新的旋转数组的长度是原数组的一半，从而将问题规模减少了一半，这种折半性质的算法的时间复杂度为  $O(\log_2 N)$ 。



此时问题的关键在于确定对半分得到的两个数组哪个是旋转数组，哪个是非递减数组。我们很容易知道非递减数组的第一个元素一定小于等于最后一个元素。

通过修改二分查找算法进行求解（l 代表 low，m 代表 mid，h 代表 high）：

- 当  $\text{nums}[m] \leq \text{nums}[h]$  时，表示  $[m, h]$  区间内的数组是非递减数组， $[l, m]$  区间内的数组是旋转数组，此时令  $h = m$ ；
- 否则  $[m + 1, h]$  区间内的数组是旋转数组，令  $l = m + 1$ 。

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

如果数组元素允许重复，会出现一个特殊的情况： $\text{nums}[l] == \text{nums}[m] == \text{nums}[h]$ ，此时无法确定解在哪个区间，需要切换到顺序查找。例如对于数组  $\{1, 1, 1, 0, 1\}$ ，l、m 和 h 指向的数都为 1，此时无法知道最小数字 0 在哪个区间。

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



```
        else
            l = m + 1;
    }
    return nums[l];
}

private int minNumber(int[] nums, int l, int h) {
    for (int i = l; i < h; i++)
        if (nums[i] > nums[i + 1])
            return nums[i + 1];
    return nums[l];
}
```

## 12. 矩阵中的路径

[生客网](#)

### 题目描述

判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向上下左右移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。

例如下面的矩阵包含了一条 bfce 路径。

a	b	t	g
c	f	c	s
j	d	e	h

 CyC2018

### 解题思路

使用回溯法（backtracking）进行求解，它是一种暴力搜索方法，通过搜索所有可能的结果来求解问题。回溯法在一次搜索结束时需要进行回溯（回退），将这一次搜索过程中设置的状态进行清除，从而开始一次新的搜索过程。例如下图示例中，从 f 开始，下一步有 4 种搜索可能，如果先搜索 b，需要将 b 标记为已经使用，防止重复使用。在这一次搜索结束之后，需要将 b 的已经使用状态清除，并搜索 c。

a	b	t	g
c	f	c	s
j	d	e	h

 CyC2018

本题的输入是数组而不是矩阵（二维数组），因此需要先将数组转换成矩阵。

```
public class Solution {
    private final static int[][] next = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};
    private int rows;
    private int cols;

    public boolean hasPath (String val, int rows, int cols, String path) {
        if (rows == 0 || cols == 0) return false;
        this.rows = rows;
        this.cols = cols;
        char[] array = val.toCharArray();
        char[][] matrix = buildMatrix(array);
        char[] pathList = path.toCharArray();
        boolean[][] marked = new boolean[rows][cols];
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                if (backtracking(matrix, pathList, marked, 0, i, j))
                    return true;

        return false;
    }

    private boolean backtracking(char[][] matrix, char[] pathList,
                                boolean[][] marked, int pathLen, int r, int c) {

        if (pathLen == pathList.length) return true;
        if (r < 0 || r >= rows || c < 0 || c >= cols
            || matrix[r][c] != pathList[pathLen] || marked[r][c]) {

            return false;
        }
        marked[r][c] = true;
        for (int[] n : next)
            if (backtracking(matrix, pathList, marked, pathLen + 1, r + n[0], c +
n[1]))
                return true;
        marked[r][c] = false;
    }
}
```

```

        return false;
    }

    private char[][] buildMatrix(char[] array) {
        char[][] matrix = new char[rows][cols];
        for (int r = 0, idx = 0; r < rows; r++)
            for (int c = 0; c < cols; c++)
                matrix[r][c] = array[idx++];
        return matrix;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        String val = "ABCESFCSADEE";
        int rows = 3;
        int cols = 4;
        String path = "ABCCED";
        boolean res = solution.hasPath(val, rows, cols, path);
        System.out.println(res);
    }
}

```

## 13. 机器人的运动范围

[牛客网](#)

### 题目描述

地上有一个  $m$  行和  $n$  列的方格。一个机器人从坐标  $(0, 0)$  的格子开始移动，每一次只能向左右上下四个方向移动一格，但是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  $3+5+3+7=18$ 。但是，它不能进入方格  $(35, 38)$ ，因为  $3+5+3+8=19$ 。请问该机器人能够达到多少个格子？

### 解题思路

使用深度优先搜索（Depth First Search, DFS）方法进行求解。回溯是深度优先搜索的一种特例，它在一次搜索过程中需要设置一些本次搜索过程的局部状态，并在本次搜索结束之后清除状态。而普通的深度优先搜索并不需要使用这些局部状态，虽然还是有可能设置一些全局状态。

```

private static final int[][] next = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};
private int cnt = 0;
private int rows;
private int cols;
private int threshold;
private int[][] digitSum;

public int movingCount(int threshold, int rows, int cols) {
    this.rows = rows;
}

```

```

        this.cols = cols;
        this.threshold = threshold;
        initDigitSum();
        boolean[][] marked = new boolean[rows][cols];
        dfs(marked, 0, 0);
        return cnt;
    }

    private void dfs(boolean[][] marked, int r, int c) {
        if (r < 0 || r >= rows || c < 0 || c >= cols || marked[r][c])
            return;
        marked[r][c] = true;
        if (this.digitSum[r][c] > this.threshold)
            return;
        cnt++;
        for (int[] n : next)
            dfs(marked, r + n[0], c + n[1]);
    }

    private void initDigitSum() {
        int[] digitSumOne = new int[Math.max(rows, cols)];
        for (int i = 0; i < digitSumOne.length; i++) {
            int n = i;
            while (n > 0) {
                digitSumOne[i] += n % 10;
                n /= 10;
            }
        }
        this.digitSum = new int[rows][cols];
        for (int i = 0; i < this.rows; i++)
            for (int j = 0; j < this.cols; j++)
                this.digitSum[i][j] = digitSumOne[i] + digitSumOne[j];
    }
}

```

## 14. 剪绳子

### 题目链接

[牛客网](#)

### 题目描述

把一根绳子剪成多段，并且使得每段的长度乘积最大。

```
n = 2
return 1 (2 = 1 + 1)

n = 10
return 36 (10 = 3 + 3 + 4)
```

## 解题思路

### 贪心

尽可能得多剪长度为 3 的绳子，并且不允许有长度为 1 的绳子出现。如果出现了，就从已经切好长度为 3 的绳子中拿出一段与长度为 1 的绳子重新组合，把它们切成两段长度为 2 的绳子。以下为证明过程。

将绳子拆成 1 和  $n-1$ ，则  $1(n-1)-n=-1<0$ ，即拆开后的乘积一定更小，所以不能出现长度为 1 的绳子。

将绳子拆成 2 和  $n-2$ ，则  $2(n-2)-n=n-4$ ，在  $n\geq 4$  时这样拆开能得到的乘积会比不拆更大。

将绳子拆成 3 和  $n-3$ ，则  $3(n-3)-n=2n-9$ ，在  $n\geq 5$  时效果更好。

将绳子拆成 4 和  $n-4$ ，因为  $4=2*2$ ，因此效果和拆成 2 一样。

将绳子拆成 5 和  $n-5$ ，因为  $5=2+3$ ，而  $5<2*3$ ，所以不能出现 5 的绳子，而是尽可能拆成 2 和 3。

将绳子拆成 6 和  $n-6$ ，因为  $6=3+3$ ，而  $6<3*3$ ，所以不能出现 6 的绳子，而是拆成 3 和 3。这里 6 同样可以拆成  $6=2+2+2$ ，但是  $3(n-3)-2(n-2)=n-5\geq 0$ ，在  $n\geq 5$  的情况下将绳子拆成 3 比拆成 2 效果更好。

继续拆成更大的绳子可以发现都比拆成 2 和 3 的效果更差，因此我们只考虑将绳子拆成 2 和 3，并且优先拆成 3，当拆到绳子长度  $n$  等于 4 时，也就是出现  $3+1$ ，此时只能拆成  $2+2$ 。

```
public int cutRope(int n) {
    if (n < 2)
        return 0;
    if (n == 2)
        return 1;
    if (n == 3)
        return 2;
    int timesOf3 = n / 3;
    if (n - timesOf3 * 3 == 1)
        timesOf3--;
    int timesOf2 = (n - timesOf3 * 3) / 2;
    return (int) (Math.pow(3, timesOf3)) * (int) (Math.pow(2, timesOf2));
}
```

### 动态规划

```

public int cutRope(int n) {
    int[] dp = new int[n + 1];
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        for (int j = 1; j < i; j++)
            dp[i] = Math.max(dp[i], Math.max(j * (i - j), dp[j] * (i - j)));
    return dp[n];
}

```

## 15. 二进制中 1 的个数

### 题目链接

[牛客网](#)

### 题目描述

输入一个整数，输出该数二进制表示中 1 的个数。

### 解题思路

$n \& (n-1)$  位运算可以将  $n$  的位级表示中最低的那一位 1 设置为 0。不断将 1 设置为 0，直到  $n$  为 0。时间复杂度： $O(M)$ ，其中  $M$  表示 1 的个数。

$n$	1	0	1	1	0	1	0	0
$n-1$	1	0	1	1	0	0	1	1
$n \& (n-1)$	1	0	1	1	0	0	0	0

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

## 16. 数值的整数次方

### 题目链接

[牛客网](#)

### 题目描述

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

### 解题思路

最直观的解法是将 x 重复乘 n 次， $x * x * x \dots * x$ ，那么时间复杂度为  $O(N)$ 。因为乘法是可交换的，所以可以将上述操作拆开成两半  $(x * x \dots * x) * (x * x \dots * x)$ ，两半的计算是一样的，因此只需要计算一次。而且对于新拆开的计算，又可以继续拆开。这就是分治思想，将原问题的规模拆成多个规模较小的子问题，最后子问题的解合并起来。

本题中子问题是  $x^{n/2}$ ，在将子问题合并时将子问题的解乘于自身相乘即可。但如果 n 不为偶数，那么拆成两半还会剩下一个 x，在将子问题合并时还需要需要多乘于一个 x。

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

因为  $(x * x)^{n/2}$  可以通过递归求解，并且每次递归 n 都减小一半，因此整个算法的时间复杂度为  $O(\log N)$ 。

```
public double Power(double x, int n) {
    boolean isNegative = false;
    if (n < 0) {
        n = -n;
        isNegative = true;
    }
    double res = pow(x, n);
    return isNegative ? 1 / res : res;
}
```

```
private double pow(double x, int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    double res = pow(x, n / 2);
    res = res * res;
    if (n % 2 != 0) res *= x;
    return res;
}
```

## 17. 打印从 1 到最大的 n 位数

### 题目描述

输入数字 n，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数即 999。

### 解题思路

由于 n 可能会非常大，因此不能直接用 int 表示数字，而是用 char 数组进行存储。

使用回溯法得到所有的数。

```
public void print1ToMaxOfNDigits(int n) {
    if (n <= 0)
        return;
    char[] number = new char[n];
    print1ToMaxOfNDigits(number, 0);
}

private void print1ToMaxOfNDigits(char[] number, int digit) {
    if (digit == number.length) {
        printNumber(number);
        return;
    }
    for (int i = 0; i < 10; i++) {
        number[digit] = (char) (i + '0');
        print1ToMaxOfNDigits(number, digit + 1);
    }
}

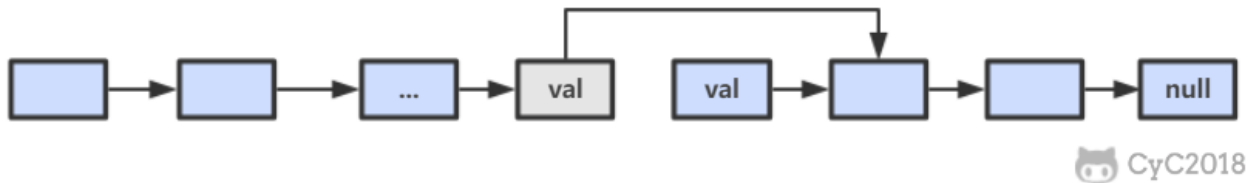
private void printNumber(char[] number) {
    int index = 0;
    while (index < number.length && number[index] == '0')
        index++;
    while (index < number.length)
        System.out.print(number[index++]);
    System.out.println();
}
```



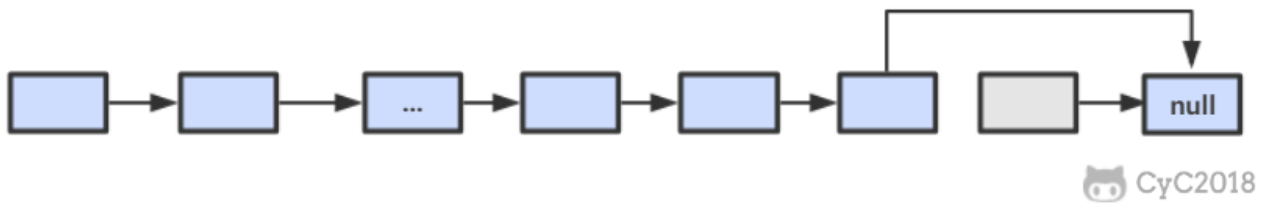
# 18.1 在 $O(1)$ 时间内删除链表节点

## 解题思路

① 如果该节点不是尾节点，那么可以直接将下一个节点的值赋给该节点，然后令该节点指向下下个节点，再删除下一个节点，时间复杂度为  $O(1)$ 。



② 否则，就需要先遍历链表，找到节点的前一个节点，然后让前一个节点指向 null，时间复杂度为  $O(N)$ 。



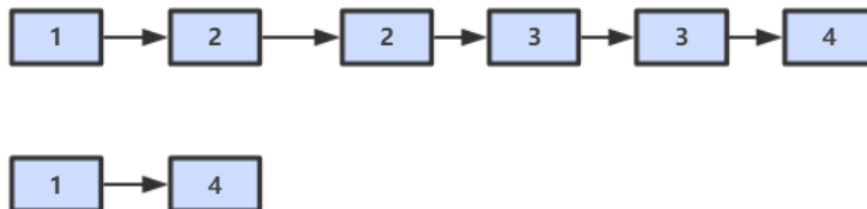
综上，如果进行  $N$  次操作，那么大约需要操作节点的次数为  $N-1+N=2N-1$ ，其中  $N-1$  表示  $N-1$  个不是尾节点的每个节点以  $O(1)$  的时间复杂度操作节点的总次数， $N$  表示 1 个尾节点以  $O(N)$  的时间复杂度操作节点的总次数。 $(2N-1)/N \sim 2$ ，因此该算法的平均时间复杂度为  $O(1)$ 。

```
public ListNode deleteNode(ListNode head, ListNode tobeDelete) {
    if (head == null || tobeDelete == null)
        return null;
    if (tobeDelete.next != null) {
        // 要删除的节点不是尾节点
        ListNode next = tobeDelete.next;
        tobeDelete.val = next.val;
        tobeDelete.next = next.next;
    } else {
        if (head == tobeDelete)
            // 只有一个节点
            head = null;
        else {
            ListNode cur = head;
            while (cur.next != tobeDelete)
                cur = cur.next;
            cur.next = null;
        }
    }
    return head;
}
```

## 18.2 删除链表中重复的结点

[生客网](#)

### 题目描述



CyC2018

### 解题描述

```
public ListNode deleteDuplication(ListNode pHead) {  
    if (pHead == null || pHead.next == null)  
        return pHead;  
    ListNode next = pHead.next;  
    if (pHead.val == next.val) {  
        while (next != null && pHead.val == next.val)  
            next = next.next;  
        return deleteDuplication(next);  
    } else {  
        pHead.next = deleteDuplication(pHead.next);  
        return pHead;  
    }  
}
```

## 19. 正则表达式匹配

[生客网](#)

### 题目描述

请实现一个函数用来匹配包括 '.' 和 '\*' 的正则表达式。模式中的字符 '.' 表示任意一个字符，而 '\*' 表示它前面的字符可以出现任意次（包含 0 次）。

在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串 "aaa" 与模式 "a.a" 和 "ab\*ac\*a" 匹配，但是与 "aa.a" 和 "ab\*a" 均不匹配。

### 解题思路

应该注意到，'.' 是用来当做一个任意字符，而 '\*' 是用来重复前面的字符。这两个的作用不同，不能把 '.' 的作用和 '\*' 进行类比，从而把它当成重复前面字符一次。

```

public boolean match(String str, String pattern) {

    int m = str.length(), n = pattern.length();
    boolean[][] dp = new boolean[m + 1][n + 1];

    dp[0][0] = true;
    for (int i = 1; i <= n; i++)
        if (pattern.charAt(i - 1) == '*')
            dp[0][i] = dp[0][i - 2];

    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (str.charAt(i - 1) == pattern.charAt(j - 1) || pattern.charAt(j - 1) ==
'.')
                dp[i][j] = dp[i - 1][j - 1];
            else if (pattern.charAt(j - 1) == '*')
                if (pattern.charAt(j - 2) == str.charAt(i - 1) || pattern.charAt(j - 2) ==
'.') {
                    dp[i][j] |= dp[i][j - 1]; // a* counts as single a
                    dp[i][j] |= dp[i - 1][j]; // a* counts as multiple a
                    dp[i][j] |= dp[i][j - 2]; // a* counts as empty
                } else
                    dp[i][j] = dp[i][j - 2]; // a* only counts as empty

    return dp[m][n];
}

```

## 20. 表示数值的字符串

[牛客网](#)

### 题目描述

```

true

"+100"
"5e2"
"-123"
"3.1416"
"-1E-16"

```

```
false

"12e"
"1a3.14"
"1.2.3"
"+-5"
"12e+4.3"
```

## 解题思路

使用正则表达式进行匹配。

```
[ ] : 字符集合
( ) : 分组
?   : 重复 0 ~ 1 次
+   : 重复 1 ~ n 次
*   : 重复 0 ~ n 次
.   : 任意字符
\\ . : 转义后的 .
\\d : 数字
```

```
public boolean isNumeric (String str) {
    if (str == null || str.length() == 0)
        return false;
    return new String(str).matches("[+-]?\\d*(\\.\\d+)?([eE][+-]?\\d+)?");
}
```

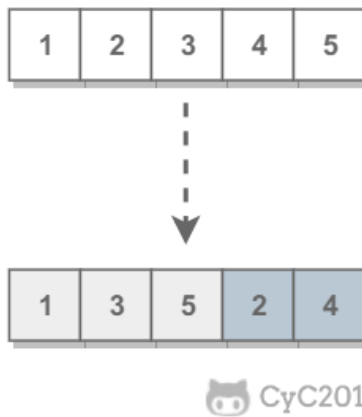
## 21. 调整数组顺序使奇数位于偶数前面

### 题目链接

[牛客网](#)

### 题目描述

需要保证奇数和奇数，偶数和偶数之间的相对位置不变，这和书本不太一样。例如对于 [1,2,3,4,5]，调整后得到 [1,3,5,2,4]，而不能是 {5,1,3,4,2} 这种相对位置改变的结果。



## 解题思路

方法一：创建一个新数组，时间复杂度  $O(N)$ ，空间复杂度  $O(N)$ 。

```
public int[] reOrderArray (int[] nums) {  
    // 奇数个数  
    int oddCnt = 0;  
    for (int x : nums)  
        if (!isEven(x))  
            oddCnt++;  
    int[] copy = nums.clone();  
    int i = 0, j = oddCnt;  
    for (int num : copy) {  
        if (num % 2 == 1)  
            nums[i++] = num;  
        else  
            nums[j++] = num;  
    }  
    return nums;  
}  
  
private boolean isEven(int x) {  
    return x % 2 == 0;  
}
```

方法二：使用冒泡思想，每次都当前偶数上浮到当前最右边。时间复杂度  $O(N^2)$ ，空间复杂度  $O(1)$ ，时间换空间。

```
public int[] reOrderArray(int[] nums) {  
    int N = nums.length;  
    for (int i = N - 1; i > 0; i--) {  
        for (int j = 0; j < i; j++) {  
            if (isEven(nums[j]) && !isEven(nums[j + 1])) {  
                swap(nums, j, j + 1);  
            }  
        }  
    }  
}
```

```

    }
    return nums;
}

private boolean isEven(int x) {
    return x % 2 == 0;
}

private void swap(int[] nums, int i, int j) {
    int t = nums[i];
    nums[i] = nums[j];
    nums[j] = t;
}

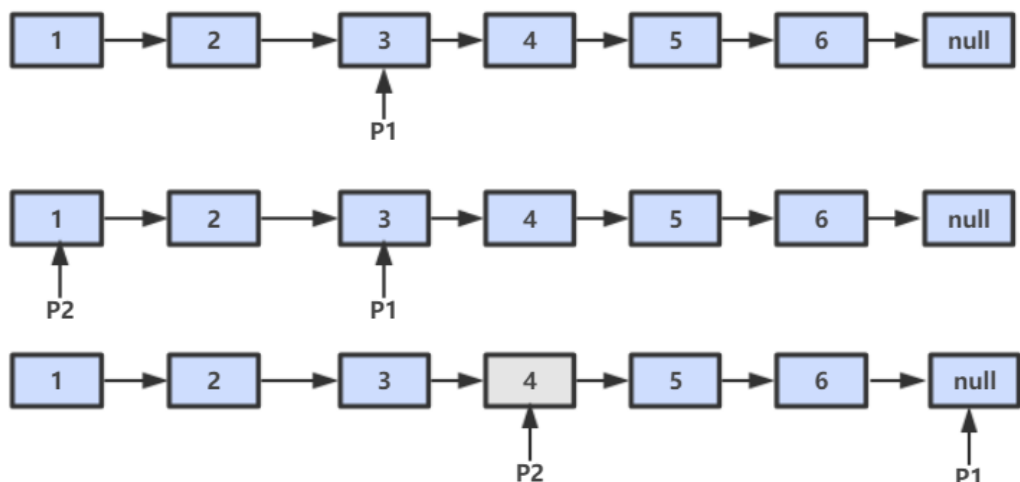
```

## 22. 链表中倒数第 K 个结点

[牛客网](#)

### 解题思路

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



CyC2018

```

public ListNode FindKthToTail(ListNode head, int k) {
    if (head == null)
        return null;
    ListNode P1 = head;
    while (P1 != null && k-- > 0)
        P1 = P1.next;
    if (k > 0)
        return null;
    ListNode P2 = head;
    while (P1 != null) {

```

```
    P1 = P1.next;
    P2 = P2.next;
}
return P2;
}
```

## 23. 链表中环的入口结点

[NowCoder](#)

### 题目描述

一个链表中包含环，请找出该链表的环的入口结点。要求不能使用额外的空间。

### 解题思路

使用双指针，一个快指针 fast 每次移动两个节点，一个慢指针 slow 每次移动一个节点。因为存在环，所以两个指针必定相遇在环中的某个节点上。

假设环入口节点为  $y_1$ ，相遇所在节点为  $z_1$ 。

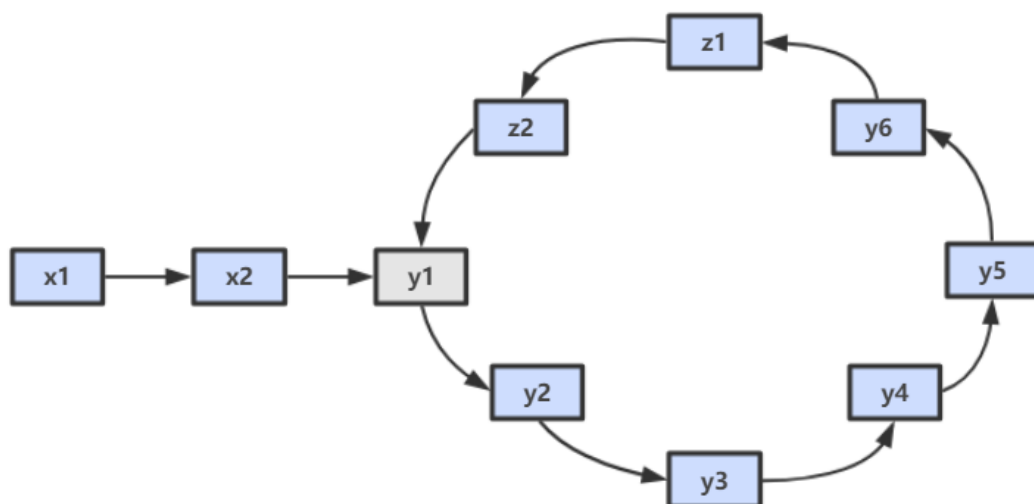
假设快指针 fast 在圈内绕了  $N$  圈，则总路径长度为  $x + Ny + (N-1)z$ 。 $z$  为  $(N-1)$  倍是因为快慢指针最后已经在  $z_1$  节点相遇了，后面就不需要再走了。

而慢指针 slow 总路径长度为  $x + y$ 。

因为快指针是慢指针的两倍，因此  $x + Ny + (N-1)z = 2(x + y)$ 。

我们要找的是环入口节点  $y_1$ ，也可以看成寻找长度  $x$  的值，因此我们先将上面的等值分解为和  $x$  有关： $x = (N-2)y + (N-1)z$ 。

上面的等值没有很强的规律，但是我们可以发现  $y + z$  就是圆环的总长度，因此我们将上面的等式再分解： $x = (N-2)(y + z) + z$ 。这个等式左边是从起点  $x_1$  到环入口节点  $y_1$  的长度，而右边是在圆环中走过  $(N-2)$  圈，再从相遇点  $z_1$  再走过长度为  $z$  的长度。此时我们可以发现如果让两个指针同时从起点  $x_1$  和相遇点  $z_1$  开始，每次只走过一个距离，那么最后他们会在环入口节点相遇。



```
public ListNode EntryNodeOfLoop(ListNode pHead) {  
    if (pHead == null || pHead.next == null)  
        return null;  
    ListNode slow = pHead, fast = pHead;  
    do {  
        fast = fast.next.next;  
        slow = slow.next;  
    } while (slow != fast);  
    fast = pHead;  
    while (slow != fast) {  
        slow = slow.next;  
        fast = fast.next;  
    }  
    return slow;  
}
```

## 24. 反转链表

[NowCoder](#)

### 解题思路

#### 递归

```
public ListNode ReverseList(ListNode head) {  
    if (head == null || head.next == null)  
        return head;  
    ListNode next = head.next;  
    head.next = null;  
    ListNode newHead = ReverseList(next);  
    next.next = head;  
    return newHead;  
}
```

#### 迭代

使用头插法。



```

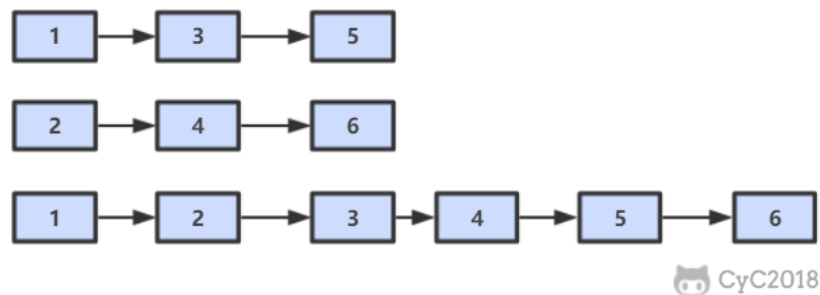
public ListNode ReverseList(ListNode head) {
    ListNode newList = new ListNode(-1);
    while (head != null) {
        ListNode next = head.next;
        head.next = newList.next;
        newList.next = head;
        head = next;
    }
    return newList.next;
}

```

## 25. 合并两个排序的链表

[NowCoder](#)

### 题目描述



### 解题思路

#### 递归

```

public ListNode Merge(ListNode list1, ListNode list2) {
    if (list1 == null)
        return list2;
    if (list2 == null)
        return list1;
    if (list1.val <= list2.val) {
        list1.next = Merge(list1.next, list2);
        return list1;
    } else {
        list2.next = Merge(list1, list2.next);
        return list2;
    }
}

```

## 迭代

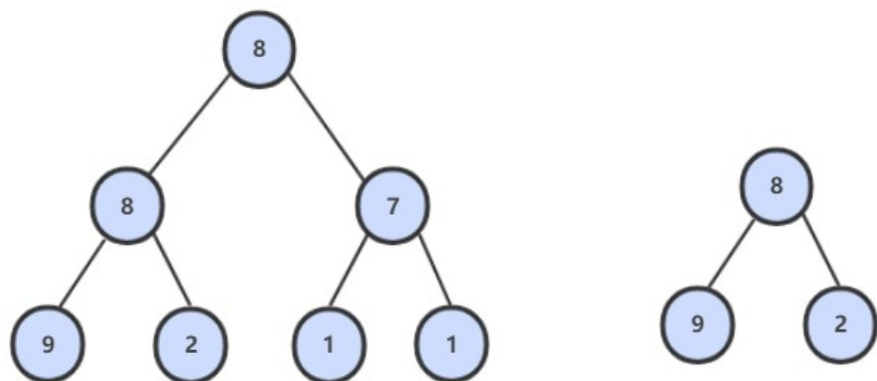
```
public ListNode Merge(ListNode list1, ListNode list2) {  
    ListNode head = new ListNode(-1);  
    ListNode cur = head;  
    while (list1 != null && list2 != null) {  
        if (list1.val <= list2.val) {  
            cur.next = list1;  
            list1 = list1.next;  
        } else {  
            cur.next = list2;  
            list2 = list2.next;  
        }  
        cur = cur.next;  
    }  
    if (list1 != null)  
        cur.next = list1;  
    if (list2 != null)  
        cur.next = list2;  
    return head.next;  
}
```

## 26. 树的子结构

### 题目链接

[牛客网](#)

### 题目描述



 CyC2018

### 解题思路

```
public boolean HasSubtree(TreeNode root1, TreeNode root2) {  
    if (root1 == null || root2 == null)
```

```

        return false;
    return isSubtreeWithRoot(root1, root2) || HasSubtree(root1.left, root2) ||
HasSubtree(root1.right, root2);
}

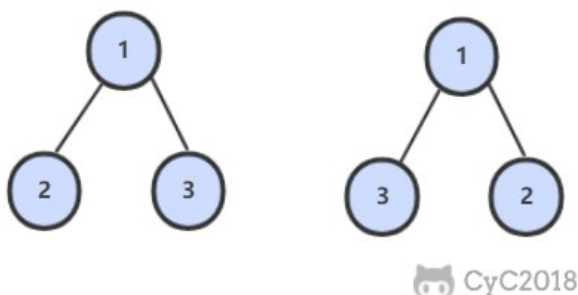
private boolean isSubtreeWithRoot(TreeNode root1, TreeNode root2) {
    if (root2 == null)
        return true;
    if (root1 == null)
        return false;
    if (root1.val != root2.val)
        return false;
    return isSubtreeWithRoot(root1.left, root2.left) && isSubtreeWithRoot(root1.right,
root2.right);
}

```

## 27. 二叉树的镜像

[牛客网](#)

### 题目描述



### 解题思路

```

public TreeNode Mirror(TreeNode root) {
    if (root == null)
        return root;
    swap(root);
    Mirror(root.left);
    Mirror(root.right);
    return root;
}

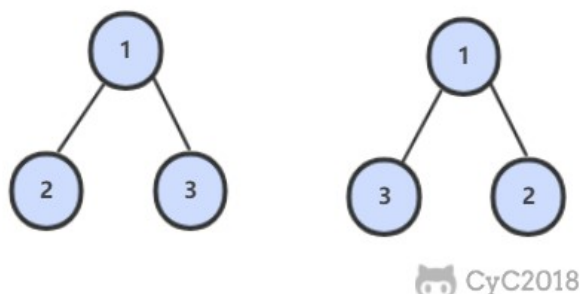
private void swap(TreeNode root) {
    TreeNode t = root.left;
    root.left = root.right;
    root.right = t;
}

```

## 28. 对称的二叉树

[NowCoder](#)

### 题目描述



### 解题思路

```
boolean isSymmetrical(TreeNode pRoot) {
    if (pRoot == null)
        return true;
    return isSymmetrical(pRoot.left, pRoot.right);
}

boolean isSymmetrical(TreeNode t1, TreeNode t2) {
    if (t1 == null && t2 == null)
        return true;
    if (t1 == null || t2 == null)
        return false;
    if (t1.val != t2.val)
        return false;
    return isSymmetrical(t1.left, t2.right) && isSymmetrical(t1.right, t2.left);
}
```

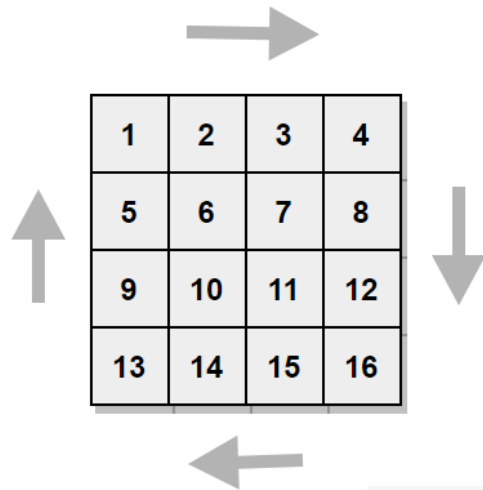
## 29. 顺时针打印矩阵

### 题目链接

[牛客网](#)

### 题目描述

按顺时针的方向，从外到里打印矩阵的值。下图的矩阵打印结果为：1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10



CyC2018

## 解题思路

一层一层从外到里打印，观察可知每一层打印都有相同的处理步骤，唯一不同的是上下左右的边界不同了。因此使用四个变量  $r1, r2, c1, c2$  分别存储上下左右边界值，从而定义当前最外层。打印当前最外层的顺序：从左到右打印最上一行→从上到下打印最右一行→从右到左打印最下一行→从下到上打印最左一行。应当注意只有在  $r1 \neq r2$  时才打印最下一行，也就是在当前最外层的行数大于 1 时才打印最下一行，这是因为当前最外层只有一行时，继续打印最下一行，会导致重复打印。打印最左一行也要做同样处理。

1	2	3	4
5			8
9			12
13	14	15	16

6	7
10	11

CyC2018

```
public ArrayList<Integer> printMatrix(int[][] matrix) {
    ArrayList<Integer> ret = new ArrayList<>();
    int r1 = 0, r2 = matrix.length - 1, c1 = 0, c2 = matrix[0].length - 1;
    while (r1 <= r2 && c1 <= c2) {
        // 上
        for (int i = c1; i <= c2; i++)
            ret.add(matrix[r1][i]);
        // 右
        for (int i = r1 + 1; i <= r2; i++)
            ret.add(matrix[i][c2]);
```

```

    if (r1 != r2)
        // 下
        for (int i = c2 - 1; i >= c1; i--)
            ret.add(matrix[r2][i]);
    if (c1 != c2)
        // 左
        for (int i = r2 - 1; i > r1; i--)
            ret.add(matrix[i][c1]);
    r1++; r2--; c1++; c2--;
}
return ret;
}

```

## 30. 包含 min 函数的栈

### 题目链接

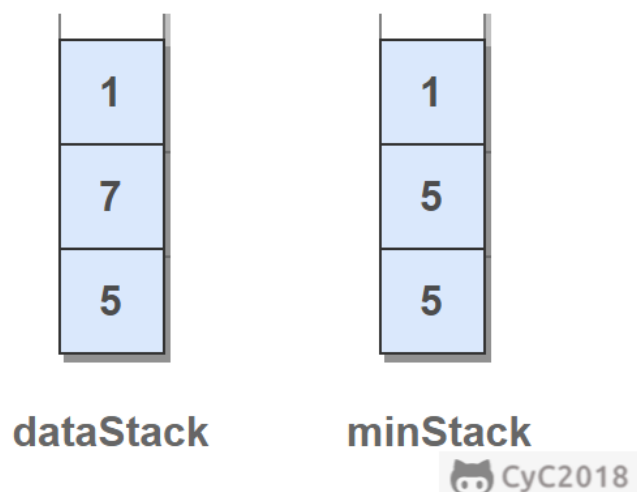
[牛客网](#)

### 题目描述

实现一个包含 min() 函数的栈，该方法返回当前栈中最小的值。

### 解题思路

使用一个额外的 minStack，栈顶元素为当前栈中最小的值。在对栈进行 push 入栈和 pop 出栈操作时，同样需要对 minStack 进行入栈出栈操作，从而使 minStack 栈顶元素一直为当前栈中最小的值。在进行 push 操作时，需要比较入栈元素和当前栈中最小值，将值较小的元素 push 到 minStack 中。



```

private Stack<Integer> dataStack = new Stack<>();
private Stack<Integer> minStack = new Stack<>();

public void push(int node) {

```

```
dataStack.push(node);
minStack.push(minStack.isEmpty() ? node : Math.min(minStack.peek(), node));
}

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

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

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

## 31. 栈的压入、弹出序列

### 题目链接

[牛客网](#)

### 题目描述

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否是该栈的弹出顺序。假设压入栈的所有数字均不相等。

例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4,5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。

### 解题思路

使用一个栈来模拟压入弹出操作。每次入栈一个元素后，都要判断一下栈顶元素是不是当前出栈序列 popSequence 的第一个元素，如果是的话则执行出栈操作并将 popSequence 往后移一位，继续进行判断。

```
public boolean IsPopOrder(int[] pushSequence, int[] popSequence) {
    int n = pushSequence.length;
    Stack<Integer> stack = new Stack<>();
    for (int pushIndex = 0, popIndex = 0; pushIndex < n; pushIndex++) {
        stack.push(pushSequence[pushIndex]);
        while (popIndex < n && !stack.isEmpty()
            && stack.peek() == popSequence[popIndex]) {
            stack.pop();
            popIndex++;
        }
    }
    return stack.isEmpty();
}
```

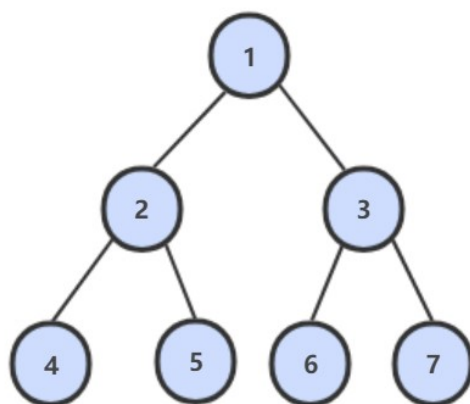
## 32.1 从上往下打印二叉树

[NowCoder](#)

### 题目描述

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

例如，以下二叉树层次遍历的结果为：1,2,3,4,5,6,7



 CyC2018

### 解题思路

使用队列来进行层次遍历。

不需要使用两个队列分别存储当前层的节点和下一层的节点，因为在开始遍历一层的节点时，当前队列中的节点数就是当前层的节点数，只要控制遍历这么多节点数，就能保证这次遍历的都是当前层的节点。

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



```

while (!queue.isEmpty()) {
    int cnt = queue.size();
    while (cnt-- > 0) {
        TreeNode t = queue.poll();
        if (t == null)
            continue;
        ret.add(t.val);
        queue.add(t.left);
        queue.add(t.right);
    }
}
return ret;
}

```

## 32.2 把二叉树打印成多行

[NowCoder](#)

### 题目描述

和上题几乎一样。

### 解题思路

```

ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(pRoot);
    while (!queue.isEmpty()) {
        ArrayList<Integer> list = new ArrayList<>();
        int cnt = queue.size();
        while (cnt-- > 0) {
            TreeNode node = queue.poll();
            if (node == null)
                continue;
            list.add(node.val);
            queue.add(node.left);
            queue.add(node.right);
        }
        if (list.size() != 0)
            ret.add(list);
    }
    return ret;
}

```

## 32.3 按之字形顺序打印二叉树

## 题目描述

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

## 解题思路

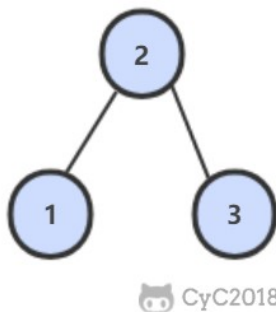
```
public ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(pRoot);
    boolean reverse = false;
    while (!queue.isEmpty()) {
        ArrayList<Integer> list = new ArrayList<>();
        int cnt = queue.size();
        while (cnt-- > 0) {
            TreeNode node = queue.poll();
            if (node == null)
                continue;
            list.add(node.val);
            queue.add(node.left);
            queue.add(node.right);
        }
        if (reverse)
            Collections.reverse(list);
        reverse = !reverse;
        if (list.size() != 0)
            ret.add(list);
    }
    return ret;
}
```

## 33. 二叉搜索树的后序遍历序列

## 题目描述

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的结果。假设输入的数组的任意两个数字都互不相同。

例如，下图是后序遍历序列 1,3,2 所对应的二叉搜索树。



## 解题思路

```
public boolean VerifySequenceOfBST(int[] sequence) {
    if (sequence == null || sequence.length == 0)
        return false;
    return verify(sequence, 0, sequence.length - 1);
}

private boolean verify(int[] sequence, int first, int last) {
    if (last - first <= 1)
        return true;
    int rootVal = sequence[last];
    int cutIndex = first;
    while (cutIndex < last && sequence[cutIndex] <= rootVal)
        cutIndex++;
    for (int i = cutIndex; i < last; i++)
        if (sequence[i] < rootVal)
            return false;
    return verify(sequence, first, cutIndex - 1) && verify(sequence, cutIndex, last - 1);
}
```

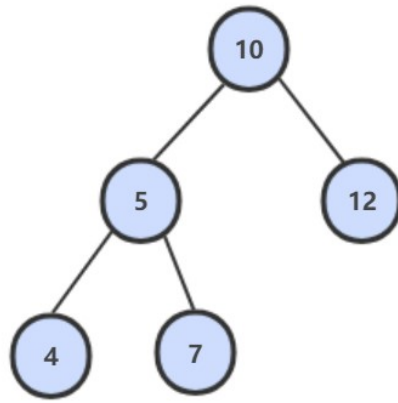
## 34. 二叉树中和为某一值的路径

[NowCoder](#)

### 题目描述

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

下图的二叉树有两条和为 22 的路径：10, 5, 7 和 10, 12



CyC2018

## 解题思路

```
private ArrayList<ArrayList<Integer>> ret = new ArrayList<>();

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root, int target) {
    backtracking(root, target, new ArrayList<>());
    return ret;
}

private void backtracking(TreeNode node, int target, ArrayList<Integer> path) {
    if (node == null)
        return;
    path.add(node.val);
    target -= node.val;
    if (target == 0 && node.left == null && node.right == null) {
        ret.add(new ArrayList<>(path));
    } else {
        backtracking(node.left, target, path);
        backtracking(node.right, target, path);
    }
    path.remove(path.size() - 1);
}
```

## 35. 复杂链表的复制

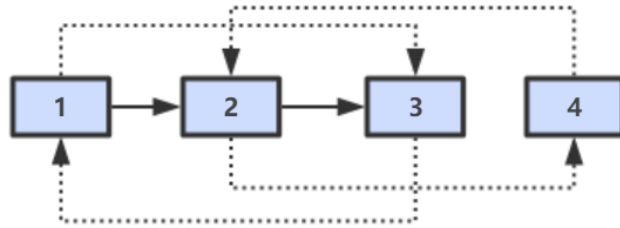
[NowCoder](#)

### 题目描述

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

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

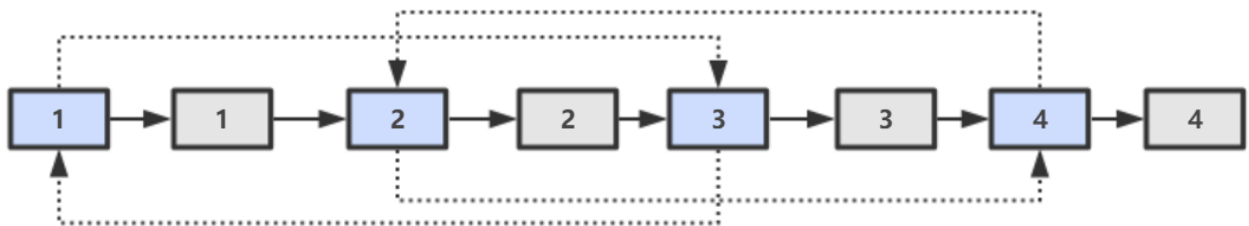
    RandomListNode(int label) {
        this.label = label;
    }
}
```



CyC2018

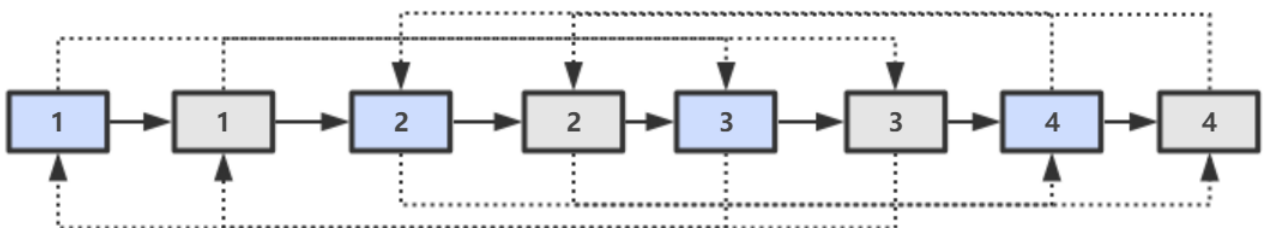
## 解题思路

第一步，在每个节点的后面插入复制的节点。



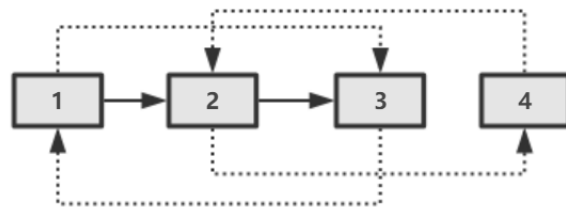
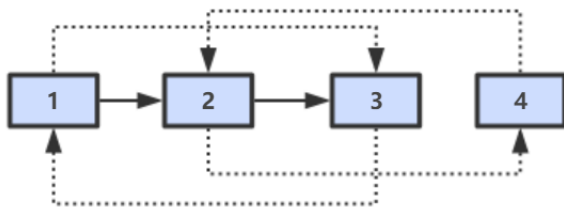
CyC2018

第二步，对复制节点的 random 链接进行赋值。



CyC2018

第三步，拆分。



CyC2018

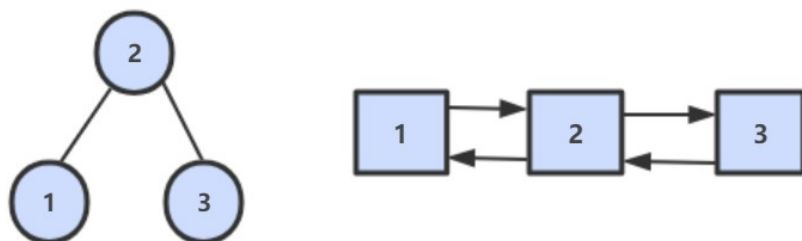
```
public RandomListNode Clone(RandomListNode pHead) {
    if (pHead == null)
        return null;
    // 插入新节点
    RandomListNode cur = pHead;
    while (cur != null) {
        RandomListNode clone = new RandomListNode(cur.label);
        clone.next = cur.next;
        cur.next = clone;
        cur = clone.next;
    }
    // 建立 random 链接
    cur = pHead;
    while (cur != null) {
        RandomListNode clone = cur.next;
        if (cur.random != null)
            clone.random = cur.random.next;
        cur = clone.next;
    }
    // 拆分
    cur = pHead;
    RandomListNode pCloneHead = pHead.next;
    while (cur.next != null) {
        RandomListNode next = cur.next;
        cur.next = next.next;
        cur = next;
    }
    return pCloneHead;
}
```

## 36. 二叉搜索树与双向链表

[NowCoder](#)

### 题目描述

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



CyC2018

## 解题思路

```
private TreeNode pre = null;
private TreeNode head = null;

public TreeNode Convert(TreeNode root) {
    inOrder(root);
    return head;
}

private void inOrder(TreeNode node) {
    if (node == null)
        return;
    inOrder(node.left);
    node.left = pre;
    if (pre != null)
        pre.right = node;
    pre = node;
    if (head == null)
        head = node;
    inOrder(node.right);
}
```

## 37. 序列化二叉树

[NowCoder](#)

### 题目描述

请实现两个函数，分别用来序列化和反序列化二叉树。

### 解题思路

```
private String deserializeStr;

public String Serialize(TreeNode root) {
    if (root == null)
        return "#";
}
```

```

        return root.val + " " + Serialize(root.left) + " " + Serialize(root.right);
    }

    public TreeNode Deserialize(String str) {
        deserializeStr = str;
        return Deserialize();
    }

    private TreeNode Deserialize() {
        if (deserializeStr.length() == 0)
            return null;
        int index = deserializeStr.indexOf(" ");
        String node = index == -1 ? deserializeStr : deserializeStr.substring(0, index);
        deserializeStr = index == -1 ? "" : deserializeStr.substring(index + 1);
        if (node.equals("#"))
            return null;
        int val = Integer.valueOf(node);
        TreeNode t = new TreeNode(val);
        t.left = Deserialize();
        t.right = Deserialize();
        return t;
    }

```

## 38. 字符串的排列

[NowCoder](#)

### 题目描述

输入一个字符串，按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc，则打印出由字符 a, b, c 所能排列出来的所有字符串 abc, acb, bac, bca, cab 和 cba。

### 解题思路

```

private ArrayList<String> ret = new ArrayList<>();

public ArrayList<String> Permutation(String str) {
    if (str.length() == 0)
        return ret;
    char[] chars = str.toCharArray();
    Arrays.sort(chars);
    backtracking(chars, new boolean[chars.length], new StringBuilder());
    return ret;
}

private void backtracking(char[] chars, boolean[] hasUsed, StringBuilder s) {
    if (s.length() == chars.length) {
        ret.add(s.toString());
    }
}

```



```

        return;
    }
    for (int i = 0; i < chars.length; i++) {
        if (hasUsed[i])
            continue;
        if (i != 0 && chars[i] == chars[i - 1] && !hasUsed[i - 1]) /* 保证不重复 */
            continue;
        hasUsed[i] = true;
        s.append(chars[i]);
        backtracking(chars, hasUsed, s);
        s.deleteCharAt(s.length() - 1);
        hasUsed[i] = false;
    }
}

```

## 39. 数组中出现次数超过一半的数字

[NowCoder](#)

### 解题思路

多数投票问题，可以利用 Boyer-Moore Majority Vote Algorithm 来解决这个问题，使得时间复杂度为  $O(N)$ 。

使用 cnt 来统计一个元素出现的次数，当遍历到的元素和统计元素相等时，令 cnt++，否则令 cnt--。如果前面查找了 i 个元素，且 cnt == 0，说明前 i 个元素没有 majority，或者有 majority，但是出现的次数少于  $i/2$ ，因为如果多于  $i/2$  的话 cnt 就一定不会为 0。此时剩下的  $n - i$  个元素中，majority 的数目依然多于  $(n - i)/2$ ，因此继续查找就能找出 majority。

```

public int MoreThanHalfNum_Solution(int[] nums) {
    int majority = nums[0];
    for (int i = 1, cnt = 1; i < nums.length; i++) {
        cnt = nums[i] == majority ? cnt + 1 : cnt - 1;
        if (cnt == 0) {
            majority = nums[i];
            cnt = 1;
        }
    }
    int cnt = 0;
    for (int val : nums)
        if (val == majority)
            cnt++;
    return cnt > nums.length / 2 ? majority : 0;
}

```

## 40. 最小的 K 个数

### 题目链接

## 解题思路

### 大小为 K 的最小堆

- 复杂度:  $O(N\log K) + O(K)$
- 特别适合处理海量数据

维护一个大小为 K 的最小堆过程如下: 使用大顶堆。在添加一个元素之后, 如果大顶堆的大小大于 K, 那么将大顶堆的堆顶元素去除, 也就是将当前堆中值最大的元素去除, 从而使得留在堆中的元素都比被去除的元素来得小。

应该使用大顶堆来维护最小堆, 而不能直接创建一个小顶堆并设置一个大小, 企图让小顶堆中的元素都是最小元素。

Java 的 PriorityQueue 实现了堆的能力, PriorityQueue 默认是小顶堆, 可以在初始化时使用 Lambda 表达式  $(o1, o2) \rightarrow o2 - o1$  来实现大顶堆。其它语言也有类似的堆数据结构。

```
public ArrayList<Integer> GetLeastNumbers_Solution(int[] nums, int k) {
    if (k > nums.length || k <= 0)
        return new ArrayList<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>((o1, o2) -> o2 - o1);
    for (int num : nums) {
        maxHeap.add(num);
        if (maxHeap.size() > k)
            maxHeap.poll();
    }
    return new ArrayList<>(maxHeap);
}
```

### 快速选择

- 复杂度:  $O(N) + O(1)$
- 只有当允许修改数组元素时才可以使用

快速排序的 partition() 方法, 会返回一个整数 j 使得  $a[l..j-1]$  小于等于  $a[j]$ , 且  $a[j+1..h]$  大于等于  $a[j]$ , 此时  $a[j]$  就是数组的第 j 大元素。可以利用这个特性找出数组的第 K 个元素, 这种找第 K 个元素的算法称为快速选择算法。

```
public ArrayList<Integer> GetLeastNumbers_Solution(int[] nums, int k) {
    ArrayList<Integer> ret = new ArrayList<>();
    if (k > nums.length || k <= 0)
        return ret;
    findKthSmallest(nums, k - 1);
    /* findKthSmallest 会改变数组, 使得前 k 个数都是最小的 k 个数 */
    for (int i = 0; i < k; i++)
        ret.add(nums[i]);
    return ret;
}

public void findKthSmallest(int[] nums, int k) {
```

```

int l = 0, h = nums.length - 1;
while (l < h) {
    int j = partition(nums, l, h);
    if (j == k)
        break;
    if (j > k)
        h = j - 1;
    else
        l = j + 1;
}
}

private int partition(int[] nums, int l, int h) {
    int p = nums[l];      /* 切分元素 */
    int i = l, j = h + 1;
    while (true) {
        while (i != h && nums[++i] < p) ;
        while (j != l && nums[--j] > p) ;
        if (i >= j)
            break;
        swap(nums, i, j);
    }
    swap(nums, l, j);
    return j;
}

private void swap(int[] nums, int i, int j) {
    int t = nums[i];
    nums[i] = nums[j];
    nums[j] = t;
}
}

```

## 41.1 数据流中的中位数

### 题目链接

[牛客网](#)

### 题目描述

如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。

### 解题思路

```

/* 大顶堆，存储左半边元素 */
private PriorityQueue<Integer> left = new PriorityQueue<>((o1, o2) -> o2 - o1);
/* 小顶堆，存储右半边元素，并且右半边元素都大于左半边 */

```

```

private PriorityQueue<Integer> right = new PriorityQueue<>();
/* 当前数据流读入的元素个数 */
private int N = 0;

public void Insert(Integer val) {
    /* 插入要保证两个堆存于平衡状态 */
    if (N % 2 == 0) {
        /* N 为偶数的情况下插入到右半边。
        * 因为右半边元素都要大于左半边，但是新插入的元素不一定比左半边元素来的大，
        * 因此需要先将元素插入左半边，然后利用左半边为大顶堆的特点，取出堆顶元素即为最大元素，此时插入
        右半边 */
        left.add(val);
        right.add(left.poll());
    } else {
        right.add(val);
        left.add(right.poll());
    }
    N++;
}

public Double GetMedian() {
    if (N % 2 == 0)
        return (left.peek() + right.peek()) / 2.0;
    else
        return (double) right.peek();
}

```

## 41.2 字符流中第一个不重复的字符

### 题目描述

[牛客网](#)

### 题目描述

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符 "go" 时，第一个只出现一次的字符是 "g"。当从该字符流中读出前六个字符 "google" 时，第一个只出现一次的字符是 "l"。

### 解题思路

使用统计数组来统计每个字符出现的次数，本题涉及到的字符都为 ASCII 码，因此使用一个大小为 128 的整型数组就能完成次数统计任务。

使用队列来存储到达的字符，并在每次有新的字符从字符流到达时移除队列头部那些出现次数不再是一次的元素。因为队列是先进先出顺序，因此队列头部的元素为第一次只出现一次的字符。

```
private int[] cnts = new int[128];
private Queue<Character> queue = new LinkedList<>();

public void Insert(char ch) {
    cnts[ch]++;
    queue.add(ch);
    while (!queue.isEmpty() && cnts[queue.peek()] > 1)
        queue.poll();
}

public char FirstAppearingOnce() {
    return queue.isEmpty() ? '#' : queue.peek();
}
```

## 42. 连续子数组的最大和

[NowCoder](#)

### 题目描述

{6, -3, -2, 7, -15, 1, 2, 2}, 连续子数组的最大和为 8（从第 0 个开始，到第 3 个为止）。

### 解题思路

```
public int FindGreatestSumOfSubArray(int[] nums) {
    if (nums == null || nums.length == 0)
        return 0;
    int greatestSum = Integer.MIN_VALUE;
    int sum = 0;
    for (int val : nums) {
        sum = sum <= 0 ? val : sum + val;
        greatestSum = Math.max(greatestSum, sum);
    }
    return greatestSum;
}
```

## 43. 从 1 到 n 整数中 1 出现的次数

[NowCoder](#)

### 解题思路

```

public int NumberOf1Between1AndN_Solution(int n) {
    int cnt = 0;
    for (int m = 1; m <= n; m *= 10) {
        int a = n / m, b = n % m;
        cnt += (a + 8) / 10 * m + (a % 10 == 1 ? b + 1 : 0);
    }
    return cnt;
}

```

[Leetcode : 233. Number of Digit One](#)

## 44. 数字序列中的某一位数字

### 题目描述

数字以 0123456789101112131415... 的格式序列化到一个字符串中，求这个字符串的第 index 位。

### 解题思路

```

public int getDigitAtIndex(int index) {
    if (index < 0)
        return -1;
    int place = 1; // 1 表示个位, 2 表示 十位...
    while (true) {
        int amount = getAmountOfPlace(place);
        int totalAmount = amount * place;
        if (index < totalAmount)
            return getDigitAtIndex(index, place);
        index -= totalAmount;
        place++;
    }
}

/**
 * place 位数的数字组成的字符串长度
 * 10, 90, 900, ...
 */
private int getAmountOfPlace(int place) {
    if (place == 1)
        return 10;
    return (int) Math.pow(10, place - 1) * 9;
}

/**
 * place 位数的起始数字
 * 0, 10, 100, ...
 */
private int getBeginNumberOfPlace(int place) {

```

```

    if (place == 1)
        return 0;
    return (int) Math.pow(10, place - 1);
}

/**
 * 在 place 位数组成的字符串中，第 index 个数
 */
private int getDigitAtIndex(int index, int place) {
    int beginNumber = getBeginNumberOfPlace(place);
    int shiftNumber = index / place;
    String number = (beginNumber + shiftNumber) + "";
    int count = index % place;
    return number.charAt(count) - '0';
}

```

## 45. 把数组排成最小的数

### 题目链接

[牛客网](#)

### 题目描述

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，打印能拼接出的所有数字中最小的一个。例如输入数组 {3, 32, 321}，则打印出这三个数字能排成的最小数字为 321323。

### 解题思路

可以看成是一个排序问题，在比较两个字符串 S1 和 S2 的大小时，应该比较的是 S1+S2 和 S2+S1 的大小，如果 S1+S2 < S2+S1，那么应该把 S1 排在前面，否则应该把 S2 排在前面。

```

public String PrintMinNumber(int[] numbers) {
    if (numbers == null || numbers.length == 0)
        return "";
    int n = numbers.length;
    String[] nums = new String[n];
    for (int i = 0; i < n; i++)
        nums[i] = numbers[i] + "";
    Arrays.sort(nums, (s1, s2) -> (s1 + s2).compareTo(s2 + s1));
    String ret = "";
    for (String str : nums)
        ret += str;
    return ret;
}

```

## 46. 把数字翻译成字符串

## 题目描述

给定一个数字，按照如下规则翻译成字符串：1 翻译成“a”，2 翻译成“b”... 26 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 abbeh, lbeh, aveh, abyh, lyh。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。

## 解题思路

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

## 47. 礼物的最大价值

## 题目描述

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。例如，对于如下棋盘

1	10	3	8
12	2	9	6
5	7	4	11
3	7	16	5

礼物的最大价值为 1+12+5+7+7+16+5=53。

## 解题思路



应该用动态规划求解，而不是深度优先搜索，深度优先搜索过于复杂，不是最优解。

```
public int getMost(int[][] values) {
    if (values == null || values.length == 0 || values[0].length == 0)
        return 0;
    int n = values[0].length;
    int[] dp = new int[n];
    for (int[] value : values) {
        dp[0] += value[0];
        for (int i = 1; i < n; i++)
            dp[i] = Math.max(dp[i], dp[i - 1]) + value[i];
    }
    return dp[n - 1];
}
```

## 48. 最长不含重复字符的子字符串

### 题目描述

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。例如对于 arabcacfr，最长不含重复字符的子字符串为 acfr，长度为 4。

### 解题思路

```
public int longestSubStringWithoutDuplication(String str) {
    int curLen = 0;
    int maxLen = 0;
    int[] preIndexs = new int[26];
    Arrays.fill(preIndexs, -1);
    for (int curI = 0; curI < str.length(); curI++) {
        int c = str.charAt(curI) - 'a';
        int preI = preIndexs[c];
        if (preI == -1 || curI - preI > curLen) {
            curLen++;
        } else {
            maxLen = Math.max(maxLen, curLen);
            curLen = curI - preI;
        }
        preIndexs[c] = curI;
    }
    maxLen = Math.max(maxLen, curLen);
    return maxLen;
}
```

## 49. 丑数

## 题目描述

把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 N 个丑数。

## 解题思路

```
public int GetUglyNumber_Solution(int N) {
    if (N <= 6)
        return N;
    int i2 = 0, i3 = 0, i5 = 0;
    int[] dp = new int[N];
    dp[0] = 1;
    for (int i = 1; i < N; i++) {
        int next2 = dp[i2] * 2, next3 = dp[i3] * 3, next5 = dp[i5] * 5;
        dp[i] = Math.min(next2, Math.min(next3, next5));
        if (dp[i] == next2)
            i2++;
        if (dp[i] == next3)
            i3++;
        if (dp[i] == next5)
            i5++;
    }
    return dp[N - 1];
}
```

## 50. 第一个只出现一次的字符位置

## 题目链接

[牛客网](#)

## 题目描述

在一个字符串中找到第一个只出现一次的字符，并返回它的位置。字符串只包含 ASCII 码字符。

Input: abacc  
Output: b

## 解题思路

最直观的解法是使用 HashMap 对出现次数进行统计：字符做为 key，出现次数作为 value，遍历字符串每次都将会 key 对应的 value 加 1。最后再遍历这个 HashMap 就可以找出出现次数为 1 的字符。

考虑到要统计的字符范围有限，也可以使用整型数组代替 HashMap。ASCII 码只有 128 个字符，因此可以使用长度为 128 的整型数组来存储每个字符出现的次数。

```

public int FirstNotRepeatingChar(String str) {
    int[] cnts = new int[128];
    for (int i = 0; i < str.length(); i++)
        cnts[str.charAt(i)]++;
    for (int i = 0; i < str.length(); i++)
        if (cnts[str.charAt(i)] == 1)
            return i;
    return -1;
}

```

以上实现的空间复杂度还不是最优的。考虑到只需要找到只出现一次的字符，那么需要统计的次数信息只有 0,1,更大，使用两个比特位就能存储这些信息。

```

public int FirstNotRepeatingChar2(String str) {
    BitSet bs1 = new BitSet(128);
    BitSet bs2 = new BitSet(128);
    for (char c : str.toCharArray()) {
        if (!bs1.get(c) && !bs2.get(c))
            bs1.set(c); // 0 0 -> 0 1
        else if (bs1.get(c) && !bs2.get(c))
            bs2.set(c); // 0 1 -> 1 1
    }
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (bs1.get(c) && !bs2.get(c)) // 0 1
            return i;
    }
    return -1;
}

```

## 51. 数组中的逆序对

[NowCoder](#)

### 题目描述

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

### 解题思路

```

private long cnt = 0;
private int[] tmp; // 在这里声明辅助数组，而不是在 merge() 递归函数中声明

public int InversePairs(int[] nums) {
    tmp = new int[nums.length];
    mergeSort(nums, 0, nums.length - 1);
}

```

```

    return (int) (cnt % 1000000007);
}

private void mergeSort(int[] nums, int l, int h) {
    if (h - l < 1)
        return;
    int m = l + (h - l) / 2;
    mergeSort(nums, l, m);
    mergeSort(nums, m + 1, h);
    merge(nums, l, m, h);
}

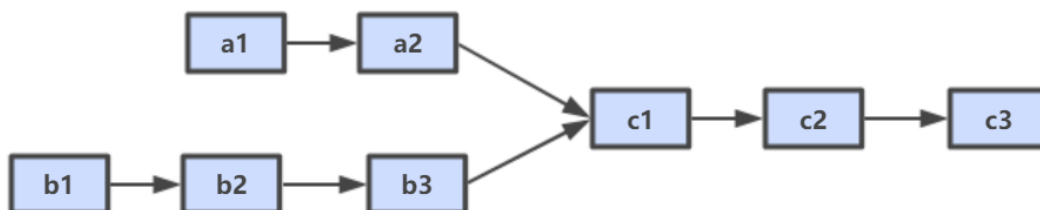
private void merge(int[] nums, int l, int m, int h) {
    int i = l, j = m + 1, k = l;
    while (i <= m || j <= h) {
        if (i > m)
            tmp[k] = nums[j++];
        else if (j > h)
            tmp[k] = nums[i++];
        else if (nums[i] <= nums[j])
            tmp[k] = nums[i++];
        else {
            tmp[k] = nums[j++];
            this.cnt += m - i + 1; // nums[i] > nums[j], 说明 nums[i...mid] 都大于
nums[j]
        }
        k++;
    }
    for (k = l; k <= h; k++)
        nums[k] = tmp[k];
}

```

## 52. 两个链表的第一个公共结点

[NowCoder](#)

### 题目描述



CyC2018

### 解题思路

设 A 的长度为  $a + c$ ，B 的长度为  $b + c$ ，其中  $c$  为尾部公共部分长度，可知  $a + c + b = b + c + a$ 。

当访问链表 A 的指针访问到链表尾部时，令它从链表 B 的头部重新开始访问链表 B；同样地，当访问链表 B 的指针访问到链表尾部时，令它从链表 A 的头部重新开始访问链表 A。这样就能控制访问 A 和 B 两个链表的指针能同时访问到交点。

```
public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
    ListNode l1 = pHead1, l2 = pHead2;
    while (l1 != l2) {
        l1 = (l1 == null) ? pHead2 : l1.next;
        l2 = (l2 == null) ? pHead1 : l2.next;
    }
    return l1;
}
```

## 53. 数字在排序数组中出现的次数

### 题目链接

[牛客网](#)

### 题目描述

Input:  
nums = 1, 2, 3, 3, 3, 3, 4, 6  
K = 3

Output:  
4

### 解题思路

只要能找出给定的数字  $k$  在有序数组第一个位置和最后一个位置，就能知道该数字出现的次数。

先考虑如何实现寻找数字在有序数组的第一个位置。正常的二分查找如下，在查找到给定元素  $k$  之后，立即返回当前索引下标。

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

```

    }
    return -1;
}

```

但是在查找第一个位置时，找到元素之后应该继续往前找。也就是当 `nums[m] >= k` 时，在左区间继续查找，左区间应该包含 `m` 位置。

```

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

```

查找最后一个位置可以转换成寻找 `k+1` 的第一个位置，并再往前移动一个位置。

```

public int GetNumberOfK(int[] nums, int K) {
    int first = binarySearch(nums, K);
    int last = binarySearch(nums, K + 1);
    return (first == nums.length || nums[first] != K) ? 0 : last - first;
}

```

需要注意以上实现的查找第一个位置的 `binarySearch` 方法，`h` 的初始值为 `nums.length`，而不是 `nums.length - 1`。先看以下示例：

```

nums = [2,2], k = 2

```

如果 `h` 的取值为 `nums.length - 1`，那么在查找最后一个位置时，`binarySearch(nums, k + 1) - 1 = 1 - 1 = 0`。这是因为 `binarySearch` 只会返回 `[0, nums.length - 1]` 范围的值，对于 `binarySearch([2,2], 3)`，我们希望返回 3 插入 `nums` 中的位置，也就是数组最后一个位置再往后一个位置，即 `nums.length`。所以我们需要将 `h` 取值为 `nums.length`，从而使得 `binarySearch` 返回的区间更大，能够覆盖 `k` 大于 `nums` 最后一个元素的情况。

## 54. 二叉查找树的第 K 个结点

[NowCoder](#)

### 解题思路

利用二叉查找树中序遍历有序的特点。

```

private TreeNode ret;
private int cnt = 0;

```

```
public TreeNode KthNode(TreeNode pRoot, int k) {
    inOrder(pRoot, k);
    return ret;
}

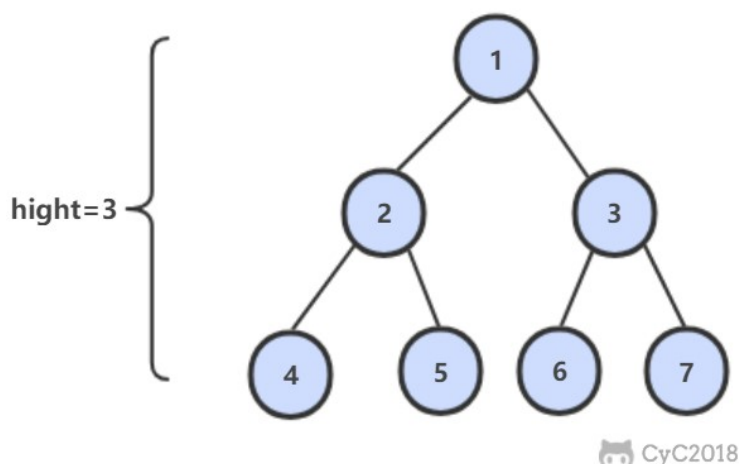
private void inOrder(TreeNode root, int k) {
    if (root == null || cnt >= k)
        return;
    inOrder(root.left, k);
    cnt++;
    if (cnt == k)
        ret = root;
    inOrder(root.right, k);
}
```

## 55.1 二叉树的深度

[NowCoder](#)

### 题目描述

从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。



### 解题思路

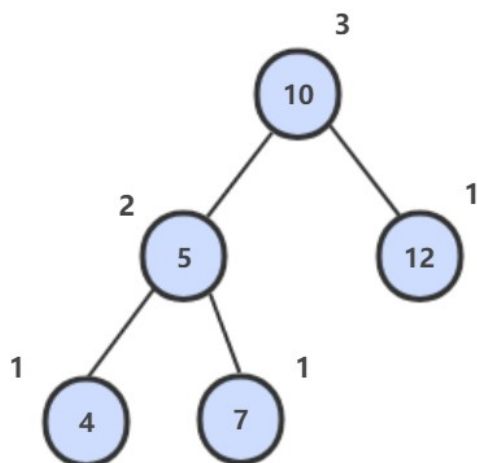
```
public int TreeDepth(TreeNode root) {
    return root == null ? 0 : 1 + Math.max(TreeDepth(root.left),
    TreeDepth(root.right));
}
```

## 55.2 平衡二叉树

[NowCoder](#)

## 题目描述

平衡二叉树左右子树高度差不超过 1。



CyC2018

## 解题思路

```
private boolean isBalanced = true;

public boolean IsBalanced_Solution(TreeNode root) {
    height(root);
    return isBalanced;
}

private int height(TreeNode root) {
    if (root == null || !isBalanced)
        return 0;
    int left = height(root.left);
    int right = height(root.right);
    if (Math.abs(left - right) > 1)
        isBalanced = false;
    return 1 + Math.max(left, right);
}
```

## 56. 数组中只出现一次的数字

### 题目链接

[牛客网](#)

### 题目描述

一个整型数组里除了两个数字之外，其他的数字都出现了两次，找出这两个数。



## 解题思路

两个相等的元素异或的结果为 0，而 0 与任意数  $x$  异或的结果都为  $x$ 。

对本题给的数组的所有元素执行异或操作，得到的是两个不存在重复的元素异或的结果。例如对于数组  $[x, x, y, y, z, k]$ ， $x \oplus x \oplus y \oplus y \oplus z \oplus k = 0 \oplus y \oplus y \oplus z \oplus k = y \oplus y \oplus z \oplus k = 0 \oplus z \oplus k = z \oplus k$ 。

两个不相等的元素在位级表示上一定会有所不同，因此这两个元素异或得到的结果  $\text{diff}$  一定不为 0。位运算  $\text{diff} \& -\text{diff}$  能得到  $\text{diff}$  位级表示中最右侧为 1 的位，令  $\text{diff} = \text{diff} \& -\text{diff}$ 。将  $\text{diff}$  作为区分两个元素的依据，一定有一个元素对  $\text{diff}$  进行异或的结果为 0，另一个结果非 0。设不相等的两个元素分别为  $z$  和  $k$ ，遍历数组所有元素，判断元素与  $\text{diff}$  的异或结果是否为 0，如果是的话将元素与  $z$  进行异或并赋值给  $z$ ，否则与  $k$  进行异或并赋值给  $k$ 。数组中相等的元素一定会同时与  $z$  或者与  $k$  进行异或操作，而不是一个与  $z$  进行异或，一个与  $k$  进行异或。而且这些相等的元素异或的结果为 0，因此最后  $z$  和  $k$  只是不相等的两个元素与 0 异或的结果，也就是不相等两个元素本身。

下面的解法中， $\text{num1}$  和  $\text{num2}$  数组的第一个元素是用来保持返回值的... 实际开发中不推荐这种返回值的方式。

```
public int[] FindNumsAppearOnce (int[] nums) {
    int[] res = new int[2];
    int diff = 0;
    for (int num : nums)
        diff ^= num;
    diff &= -diff;
    for (int num : nums) {
        if ((num & diff) == 0)
            res[0] ^= num;
        else
            res[1] ^= num;
    }
    if (res[0] > res[1]) {
        swap(res);
    }
    return res;
}

private void swap(int[] nums) {
    int t = nums[0];
    nums[0] = nums[1];
    nums[1] = t;
}
```

## 57.1 和为 S 的两个数字

### 题目链接

[牛客网](#)

### 题目描述

在有序数组中找出两个数，使得和为给定的数  $S$ 。如果有多对数字的和等于  $S$ ，输出两个数的乘积最小的。

## 解题思路

使用双指针，一个指针指向元素较小的值，一个指针指向元素较大的值。指向较小元素的指针从头向尾遍历，指向较大元素的指针从尾向头遍历。

- 如果两个指针指向元素的和  $sum == target$ ，那么这两个元素即为所求。
- 如果  $sum > target$ ，移动较大的元素，使  $sum$  变小一些；
- 如果  $sum < target$ ，移动较小的元素，使  $sum$  变大一些。

```
public ArrayList<Integer> FindNumbersWithSum(int[] nums, int target) {
    int i = 0, j = nums.length - 1;
    while (i < j) {
        int cur = nums[i] + array[j];
        if (cur == target)
            return new ArrayList<>(Arrays.asList(nums[i], nums[j]));
        if (cur < target)
            i++;
        else
            j--;
    }
    return new ArrayList<>();
}
```

## 57.2 和为 $S$ 的连续正数序列

### 题目描述

[牛客网](#)

### 题目描述

输出所有和为  $S$  的连续正数序列。例如和为 100 的连续序列有：

```
[9, 10, 11, 12, 13, 14, 15, 16]
[18, 19, 20, 21, 22]。
```

### 解题思路

```
public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();
    int start = 1, end = 2;
    int curSum = 3;
    while (end < sum) {
        if (curSum > sum) {
            curSum -= start;
        }
    }
}
```

```

        start++;
    } else if (curSum < sum) {
        end++;
        curSum += end;
    } else {
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++)
            list.add(i);
        ret.add(list);
        curSum -= start;
        start++;
        end++;
        curSum += end;
    }
}
return ret;
}

```

## 58.1 翻转单词顺序列

### 题目描述

[牛客网](#)

### 题目描述

Input:  
"I am a student."

Output:  
"student. a am I"

### 解题思路

先翻转每个单词，再翻转整个字符串。

题目应该有一个隐含条件，就是不能用额外的空间。虽然 Java 的题目输入参数为 String 类型，需要先创建一个字符数组使得空间复杂度为  $O(N)$ ，但是正确的参数类型应该和原书一样，为字符数组，并且只能使用该字符数组的空间。任何使用了额外空间的解法在面试时都会大打折扣，包括递归解法。

```

public String ReverseSentence(String str) {
    int n = str.length();
    char[] chars = str.toCharArray();
    int i = 0, j = 0;
    while (j <= n) {
        if (j == n || chars[j] == ' ') {
            reverse(chars, i, j - 1);

```

```

        i = j + 1;
    }
    j++;
}
reverse(chars, 0, n - 1);
return new String(chars);
}

private void reverse(char[] c, int i, int j) {
    while (i < j)
        swap(c, i++, j--);
}

private void swap(char[] c, int i, int j) {
    char t = c[i];
    c[i] = c[j];
    c[j] = t;
}

```

## 58.2 左旋转字符串

### 题目链接

[牛客网](#)

### 题目描述

将字符串 S 从第 K 位置分隔成两个子字符串，并交换这两个子字符串的位置。

Input:  
S="abcXYZdef"  
K=3

Output:  
"XYZdefabc"

### 解题思路

先将 "abc" 和 "XYZdef" 分别翻转，得到 "cbafedZYX"，然后再把整个字符串翻转得到 "XYZdefabc"。

```

public String LeftRotateString(String str, int n) {
    if (n >= str.length())
        return str;
    char[] chars = str.toCharArray();
    reverse(chars, 0, n - 1);
    reverse(chars, n, chars.length - 1);
    reverse(chars, 0, chars.length - 1);
}

```

```
    return new String(chars);
}

private void reverse(char[] chars, int i, int j) {
    while (i < j)
        swap(chars, i++, j--);
}

private void swap(char[] chars, int i, int j) {
    char t = chars[i];
    chars[i] = chars[j];
    chars[j] = t;
}
```

## 59. 滑动窗口的最大值

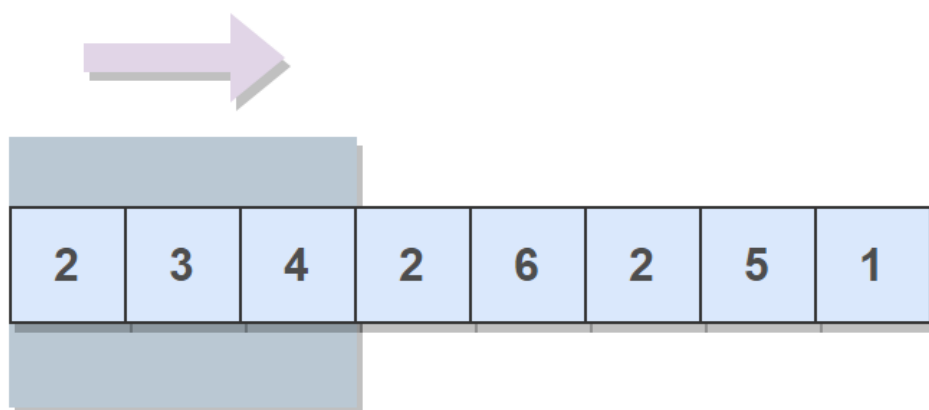
### 题目链接

[牛客网](#)

### 题目描述

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。

例如，如果输入数组 {2, 3, 4, 2, 6, 2, 5, 1} 及滑动窗口的大小 3，那么一共存在 6 个滑动窗口，他们的最大值分别为 {4, 4, 6, 6, 6, 5}。



 CyC2018

### 解题思路

维护一个大小为窗口大小的大顶堆，顶堆元素则为当前窗口的最大值。

假设窗口的大小为  $M$ ，数组的长度为  $N$ 。在窗口向右移动时，需要先在堆中删除离开窗口的元素，并将新到达的元素添加到堆中，这两个操作的时间复杂度都为  $\log_2 M$ ，因此算法的时间复杂度为  $O(N \log_2 M)$ ，空间复杂度为  $O(M)$ 。

```
public ArrayList<Integer> maxInWindows(int[] num, int size) {
    ArrayList<Integer> ret = new ArrayList<>();
    if (size > num.length || size < 1)
        return ret;
    PriorityQueue<Integer> heap = new PriorityQueue<>((o1, o2) -> o2 - o1); /* 大顶堆 */
    for (int i = 0; i < size; i++)
        heap.add(num[i]);
    ret.add(heap.peek());
    for (int i = 0, j = i + size; j < num.length; i++, j++) { /* 维护一个大小为 size 的大顶堆 */
        heap.remove(num[i]);
        heap.add(num[j]);
        ret.add(heap.peek());
    }
    return ret;
}
```

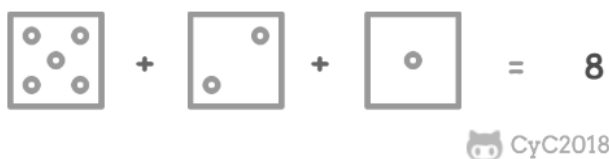
## 60. n 个骰子的点数

### 题目链接


[Lintcode](#)

### 题目描述

把  $n$  个骰子扔在地上，求点数和为  $s$  的概率。



The diagram illustrates the sum of three dice faces. The first die shows 4 dots, the second shows 2 dots, and the third shows 1 dot. These are summed together to equal 8.

 CyC2018

### 解题思路

#### 动态规划

使用一个二维数组  $dp$  存储点数出现的次数，其中  $dp[i][j]$  表示前  $i$  个骰子产生点数  $j$  的次数。

空间复杂度： $O(N^2)$

```
public List<Map.Entry<Integer, Double>> dicesSum(int n) {
    final int face = 6;
```

```

final int pointNum = face * n;
long[][] dp = new long[n + 1][pointNum + 1];

for (int i = 1; i <= face; i++)
    dp[1][i] = 1;

for (int i = 2; i <= n; i++)
    for (int j = i; j <= pointNum; j++)        /* 使用 i 个骰子最小点数为 i */
        for (int k = 1; k <= face && k <= j; k++)
            dp[i][j] += dp[i - 1][j - k];

final double totalNum = Math.pow(6, n);
List<Map.Entry<Integer, Double>> ret = new ArrayList<>();
for (int i = n; i <= pointNum; i++)
    ret.add(new AbstractMap.SimpleEntry<>(i, dp[n][i] / totalNum));

return ret;
}

```

## 动态规划 + 旋转数组

空间复杂度:  $O(N)$

```

public List<Map.Entry<Integer, Double>> dicesSum(int n) {
    final int face = 6;
    final int pointNum = face * n;
    long[][] dp = new long[2][pointNum + 1];

    for (int i = 1; i <= face; i++)
        dp[0][i] = 1;

    int flag = 1;                                /* 旋转标记 */
    for (int i = 2; i <= n; i++, flag = 1 - flag) {
        for (int j = 0; j <= pointNum; j++)
            dp[flag][j] = 0;                    /* 旋转数组清零 */

        for (int j = i; j <= pointNum; j++)
            for (int k = 1; k <= face && k <= j; k++)
                dp[flag][j] += dp[1 - flag][j - k];
    }

    final double totalNum = Math.pow(6, n);
    List<Map.Entry<Integer, Double>> ret = new ArrayList<>();
    for (int i = n; i <= pointNum; i++)
        ret.add(new AbstractMap.SimpleEntry<>(i, dp[1 - flag][i] / totalNum));

    return ret;
}

```

# 61. 扑克牌顺子

## 题目链接

[NowCoder](#)

## 题目描述

五张牌，其中大小鬼为癞子，牌面为 0。判断这五张牌是否能组成顺子。

1	0	3	4	5
---	---	---	---	---

return true

1	2	4	6	7
---	---	---	---	---

return false

 CyC2018

## 解题思路

```
public boolean isContinuous(int[] nums) {  
  
    if (nums.length < 5)  
        return false;  
  
    Arrays.sort(nums);  
  
    // 统计癞子数量  
    int cnt = 0;  
    for (int num : nums)  
        if (num == 0)  
            cnt++;  
  
    // 使用癞子去补全不连续的顺子  
    for (int i = cnt; i < nums.length - 1; i++) {  
        if (nums[i + 1] == nums[i])  
            return false;  
        cnt -= nums[i + 1] - nums[i] - 1;  
    }  
  
    return cnt >= 0;  
}
```

# 62. 圆圈中最后剩下的数



## 题目链接

[NowCoder](#)

## 题目描述

让小朋友们围成一个大圈。然后，随机指定一个数  $m$ ，让编号为 0 的小朋友开始报数。每次喊到  $m-1$  的那个小朋友要出列唱首歌，然后可以在礼品箱中任意的挑选礼物，并且不再回到圈中，从他的下一个小朋友开始，继续 0... $m-1$  报数 .... 这样下去 .... 直到剩下最后一个小朋友，可以不用表演。

## 解题思路

约瑟夫环，圆圈长度为  $n$  的解可以看成长度为  $n-1$  的解再加上报数的长度  $m$ 。因为是圆圈，所以最后需要对  $n$  取余。

```
public int LastRemaining_Solution(int n, int m) {  
    if (n == 0)        /* 特殊输入的处理 */  
        return -1;  
    if (n == 1)        /* 递归返回条件 */  
        return 0;  
    return (LastRemaining_Solution(n - 1, m) + m) % n;  
}
```

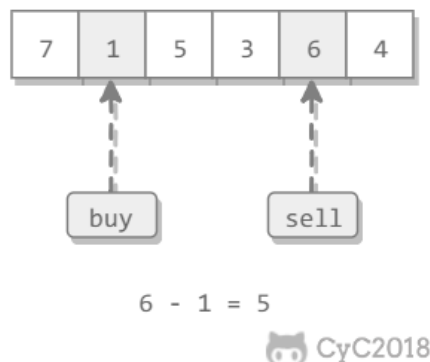
## 63. 股票的最大利润

### 题目链接

[Leetcode: 121. Best Time to Buy and Sell Stock](#)

### 题目描述

可以有一次买入和一次卖出，买入必须在前。求最大收益。



### 解题思路

使用贪心策略，假设第  $i$  轮进行卖出操作，买入操作价格应该在  $i$  之前并且价格最低。因此在遍历数组时记录当前最低的买入价格，并且尝试将每个位置都作为卖出价格，取收益最大的即可。

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    int soFarMin = prices[0];
    int maxProfit = 0;
    for (int i = 1; i < prices.length; i++) {
        soFarMin = Math.min(soFarMin, prices[i]);
        maxProfit = Math.max(maxProfit, prices[i] - soFarMin);
    }
    return maxProfit;
}
```

## 64. 求 1+2+3+...+n

### 题目链接

[NowCoder](#)

### 题目描述

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。

### 解题思路

使用递归解法最重要的是指定返回条件，但是本题无法直接使用 if 语句来指定返回条件。

条件与 && 具有短路原则，即在第一个条件语句为 false 的情况下不会去执行第二个条件语句。利用这一特性，将递归的返回条件取非然后作为 && 的第一个条件语句，递归的主体转换为第二个条件语句，那么当递归的返回条件为 true 的情况下就不会执行递归的主体部分，递归返回。

本题的递归返回条件为  $n \leq 0$ ，取非后就是  $n > 0$ ；递归的主体部分为  $sum += \text{Sum\_Solution}(n - 1)$ ，转换为条件语句后就是  $(sum += \text{Sum\_Solution}(n - 1)) > 0$ 。

```
public int Sum_Solution(int n) {
    int sum = n;
    boolean b = (n > 0) && ((sum += Sum_Solution(n - 1)) > 0);
    return sum;
}
```

## 65. 不用加减乘除做加法

### 题目链接

[NowCoder](#)

### 题目描述

写一个函数，求两个整数之和，要求不得使用 +、-、\*、/ 四则运算符号。

## 解题思路

$a \oplus b$  表示没有考虑进位的情况下两数的和， $(a \& b) \ll 1$  就是进位。

递归会终止的原因是  $(a \& b) \ll 1$  最右边会多一个 0，那么继续递归，进位最右边的 0 会慢慢增多，最后进位会变为 0，递归终止。

```
public int Add(int a, int b) {  
    return b == 0 ? a : Add(a ^ b, (a & b) << 1);  
}
```

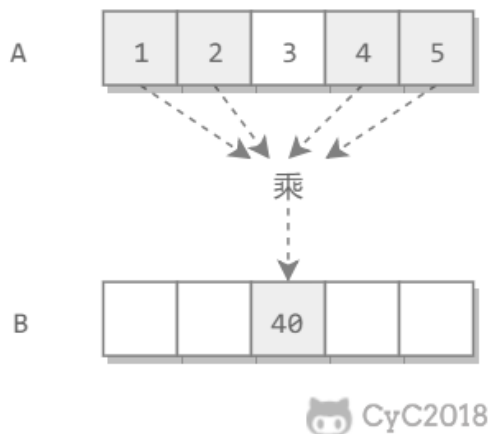
## 66. 构建乘积数组

### 题目链接

[NowCoder](#)

### 题目描述

给定一个数组  $A[0, 1, \dots, n-1]$ ，请构建一个数组  $B[0, 1, \dots, n-1]$ ，其中 B 中的元素  $B[i]=A[0]*A[1]*\dots*A[i-1]*A[i+1]*\dots*A[n-1]$ 。要求不能使用除法。



CyC2018

## 解题思路

```
public int[] multiply(int[] A) {  
    int n = A.length;  
    int[] B = new int[n];  
    for (int i = 0, product = 1; i < n; product *= A[i], i++) /* 从左往右累乘 */  
        B[i] = product;  
    for (int i = n - 1, product = 1; i >= 0; product *= A[i], i--) /* 从右往左累乘 */  
        B[i] *= product;  
    return B;  
}
```

## 67. 把字符串转换成整数

### 题目链接

[NowCoder](#)

### 题目描述

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

```
Input:
+2147483647
1a33

Output:
2147483647
0
```

### 解题思路

```
public int StrToInt(String str) {
    if (str == null || str.length() == 0)
        return 0;
    boolean isNegative = str.charAt(0) == '-';
    int ret = 0;
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (i == 0 && (c == '+' || c == '-')) /* 符号判定 */
            continue;
        if (c < '0' || c > '9') /* 非法输入 */
            return 0;
        ret = ret * 10 + (c - '0');
    }
    return isNegative ? -ret : ret;
}
```

## 68. 树中两个节点的最低公共祖先

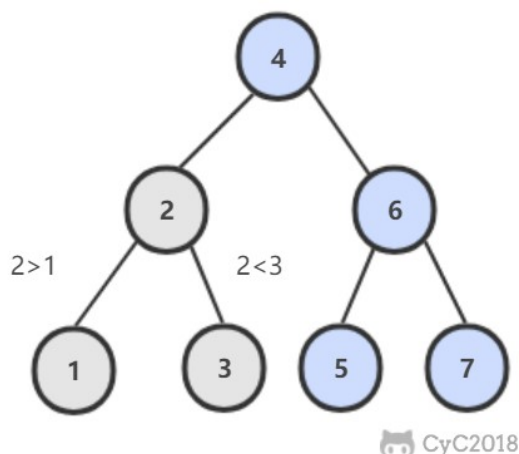
### 68.1 二叉查找树

#### 题目链接

[Leetcode : 235. Lowest Common Ancestor of a Binary Search Tree](#)

## 解题思路

在二叉查找树中，两个节点  $p, q$  的公共祖先  $root$  满足  $root.val \geq p.val \ \&\& \ root.val \leq q.val$ 。



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

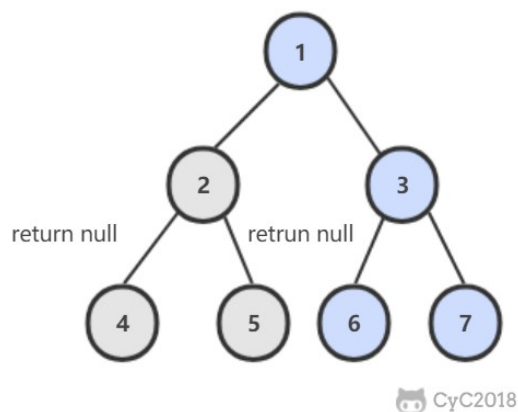
## 68.2 普通二叉树

### 题目链接

[Leetcode : 236. Lowest Common Ancestor of a Binary Tree](#)

### 解题思路

在左右子树中查找是否存在  $p$  或者  $q$ ，如果  $p$  和  $q$  分别在两个子树中，那么就说明根节点就是最低公共祖先。



```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
    if (root == null || root == p || root == q)  
        return root;  
    TreeNode left = lowestCommonAncestor(root.left, p, q);  
    TreeNode right = lowestCommonAncestor(root.right, p, q);  
    return left == null ? right : right == null ? left : root;  
}
```