链滴

剑指 Offer 算法题 (1)

作者：ellenbboe
原文链接：https://ld246.com/article/1565227474130
来源网站：链滴

数组中重复的数字

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。

要求是时间复杂度 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。

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 的了，因此可以知道 2 重复

```
public boolean duplicate(int numbers[],int length,int [] duplication) {

        if(numbers==null || length ==0)
        {
            return false;
        }
        for(int i=0;i<numbers.length;i++)
        {
            while(numbers[i] != i)
            {
                if(numbers[i] == numbers[numbers[i]])
                {
                    duplication[0] =numbers[i];
                    return true;
                }
                Util.swap(numbers,i,numbers[i]);
            }
        }
        return false;
    }
```

二维数组中的查找

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，判断这个数否在该二维数组中。

该二维数组中的一个数，它左边的数都比它小，下边的数都比它大。因此，从右上角开始查找，就可根据 target 和当前元素的大小关系来缩小查找区间，当前元素的查找区间为左下角的所有元素。

```
 public boolean Find(int target, int [][] array) {

        if(array==null||array.length==0||array[0].length == 0)
        {
            return false;
        }
        int rows = array.length-1;
        int cols = array[0].length-1;
        int row = 0,col = cols;
        while(row<=rows&&col>=0)
        {
            if(target < array[row][col])
            {
```

```
            col--;
        }else if(target >array[row][col])
        {
            row++;
        }else{
            return true;
        }
    }
    return false;
}
```

替换空格

将一个字符串中的空格替换成 "%20"。

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（%20），因此当遍历到一个空格时，需要在尾部填充两个任意字符。

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 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。

```
public String replaceSpace(StringBuffer str) {

    int perlength = str.length()-1;

    for(int i=0;i<=perlength;i++)
    {
        if(str.charAt(i) == ' ')
        {
            str.append("  ");
        }
    }
    int nowlength = str.length()-1;
    while(perlength>=0&&nowlength>=perlength)
    {
        if(str.charAt(perlength)==' ')
        {
            str.setCharAt(nowlength--,'0');
            str.setCharAt(nowlength--,'2');
            str.setCharAt(nowlength--,'%');
        }else{
            str.setCharAt(nowlength--,str.charAt(perlength));
        }
        perlength--;
    }

    return str.toString();
}
```

从尾到头打印链表

从尾到头反过来打印出每个结点的值。

使用递归

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

        ArrayList<Integer> ret = new ArrayList<>();
        if(listNode==null)
        {
            return ret;
        }
        if(listNode.next == null)
        {
            ret.add(listNode.val);
        }else{
            ret = printListFromTailToHead(listNode.next);
            ret.add(listNode.val);
        }
        return ret;
    }
```

使用头插法

```
public class solution6_2 {
    //链接新的节点的时候还是通过new来构建一个新的节点吧
    //像这样ListNode node = new ListNode(listNode.val); 直接赋值list的话会晕-.-
    public ArrayList<Integer> printListFromTailHToHead(ListNode listNode)
    {
     ListNode head = new ListNode(-1);

     while(listNode!=null)
     {
         ListNode node = new ListNode(listNode.val);
         if(head.next!=null)
         {
             node.next = head.next;
         }
         head.next = node;
         listNode = listNode.next;
     }
     ArrayList<Integer> ret = new ArrayList<>();
     head = head.next;
     while(head!=null)
     {
         ret.add(head.val);
         head = head.next;
     }
     return ret;
    }
}
```

使用栈(最简单)

```
public class solution6_3 {
    //倒叙可以优先考虑栈
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        Stack<ListNode> stack = new Stack<>();
        ArrayList<Integer> ret = new ArrayList<>();
        while(listNode!=null)
        {
            stack.push(listNode);
            listNode = listNode.next;
        }
        while(!stack.isEmpty())
        {
            ret.add(stack.pop().val);
        }
        return ret;
    }
}
```

重建二叉树

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都不含重复的数字。

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遍历结果，右部分为树的右子树中序遍历的结果。

```
HashMap<Integer,Integer> hashMap = new HashMap<>();//存放中序节点的下标
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        for(int i = 0;i<in.length;i++)
        {
            hashMap.put(in[i],i);
        }
        return reConstructBinaryTree(pre,0,pre.length-1,0);
    }
    public TreeNode reConstructBinaryTree(int [] pre,int preL,int preR,int inL) {
        if(preL>preR)
        {
            return null;
        }
        TreeNode root = new TreeNode(pre[preL]);
        int index = hashMap.get(root.val);
        int LeftSize = index - inL;//左子树的大小,为了获得右子树的开始
        root.left = reConstructBinaryTree(pre,preL+1,preL+LeftSize,inL);
        root.right =reConstructBinaryTree(pre,preL+LeftSize+1,preR,inL+LeftSize+1);
        return root;
    }
```

二叉树的下一个结点

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，树中的结点仅包含左右子结点，同时包含指向父结点的指针。

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

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。

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

        }
    }
```

用两个栈实现队列

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。

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。

```
package nowcoder.retry;

import java.util.Stack;

public class solution9 {
    Stack<Integer> in = new Stack<>();
    Stack<Integer> out = new Stack<>();
    public void push(int node) {
        in.push(node);
    }

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

        return out.pop();
    }
```

```
}
```

斐波那契数列

求斐波那契数列的第 n 项，n <= 39。

如果使用递归求解，会重复计算一些子问题。例如，计算 f(10) 需要计算 f(9) 和 f(8)，计算 f(9) 需要算 f(8) 和 f(7)，可以看到 f(8) 被重复计算了。

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，从而避免重复求解子问题。

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，从而将空间杂度由 O(N) 降低为 O(1)。

```
public int Fibonacci(int n)
    {
        int[] pre =new int[2];
        pre[0] = 0;
        pre[1] = 1;
        if(n<2)
        {
            return pre[n];
        }
        for(int i = 2;i<=n;i++)
        {
            pre[0] = pre[0] + pre[1];
            Util.swap(pre,0,1);
        }
        return pre[1];
    }
```

矩形覆盖

我们可以用 21 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 21 的小矩形无重叠地覆盖一个 *n 的大矩形，总共有多少种方法？找规律 f(n) = f(n-1)+f(n-2)

```
package nowcoder.retry;

public class solution10_1 {
    public int Fibonacci(int n)
    {
        int[] pre =new int[2];
        pre[0] = 1;
        pre[1] = 2;
        if(n<2)
        {
            return pre[n];
        }
        for(int i = 3;i<=n;i++)
        {
            pre[0] = pre[0] + pre[1];
            Util.swap(pre,0,1);
        }
```

```
        return pre[1];
    }
}
```

跳台阶

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。

找规律 f(n) = f(n-1)+f(n-2)

```
package nowcoder.retry;

public class solution10_2 {
    public int Fibonacci(int n)
    {
        int[] pre =new int[2];
        pre[0] = 1;
        pre[1] = 2;
        if(n<2)
        {
            return pre[n];
        }
        for(int i = 3;i<=n;i++)
        {
            pre[0] = pre[0] + pre[1];
            Util.swap(pre,0,1);
        }
        return pre[1];
    }
}
```

变态跳台阶

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 级的阶总共有多少种跳法。

动态规划

```
package nowcoder.retry;

import java.util.Arrays;

public class solution10_3 {
    public int JumpFloorII(int target) {
        int[] array = new int[target];
        Arrays.fill(array,1);
        for(int i = 1;i<target;i++)
        {
            for(int j =i-1;j>=0;j--)
            {
```

```
                array[i] = array[j]+array[i];
            }
        }
        return array[target-1];
    }
}
```

数学推导

```
package nowcoder.retry;

public class solution10_3_1 {
    public int JumpFloorII(int target) {
        return (int) Math.pow(2, target - 1);
    }
}
```

旋转数组的最小数字

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的一个旋转，输出旋转数组的最小元素。

例如数组 {3, 4, 5, 1, 2} 为 {1, 2, 3, 4, 5} 的一个旋转，该数组的最小值为 1。

在一个有序数组中查找一个元素可以用二分查找，二分查找也称为折半查找，每次都能将查找区间减，这种折半特性的算法时间复杂度都为 O(logN)。

本题可以修改二分查找算法进行求解：

当 nums[m] <= nums[h] 的情况下，说明解在 [l, m] 之间，此时令 h = m；(由于之前有顺序性)

否则解在 [m + 1, h] 之间，令 l = m + 1。

如果数组元素允许重复的话，那么就会出现一个特殊的情况：nums[l] == nums[m] == nums[h]，么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 指向的都为 1，此时无法知道最小数字 0 在哪个区间。

```
package nowcoder.retry;

public class solution11 {
    public int minNumberInRotateArray(int[] nums) {
        if (nums.length == 0)
            return 0;
        int low = 0;
        int high = nums.length-1;
        while(low!=high)
        {
            int m = (high + low)/2;
            if(nums[m] > nums[low])
            {
                low = m;
            }else{
```

```
                high = m;
            }
        }
        return nums[low+1];
    }
}
```

本文参照该CyC的CS-Notes学习.代码在我的Github仓库中 heart