1. P44.在一个二维数组中，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。

- 思路：从右上角或左下角开始找，逐行删除。或者用二分法查找。
- 代码实现：

```
function Find(target, array) {
    let n = array.length,m = array[0].length;
    let row = n - 1,col = 0;
    if (m === 0 && n === 0) {
        return false;
    }
    while (row >= 0 && col <= m - 1) {
        if (array[row][col] > target) {
            row--;
         } else if (array[row][col] < target) {
             col++;
         } else return true;
     }
     return false;
}
```

2. P51，请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

- 思路：从后往前复制，数组长度会增加。先计算空格的位置，计算替换后的长度。或者使用正则表达式。
- 代码实现：

```
function replaceSpace(str) {
    return str.replace(/\s/g, '%20');
}
```

3. P58,输入一个链表，从尾到头打印链表每个节点的值。

- 思路：用栈或者递归
- 代码实现：

```
function printListFromTailToHead(head) {
    let res = [];
    let pNode = head;
    while (pNode !== null) {
//unshift()方法将一个或多个元素添加到数组的开头，并返回该数组的新长度(该方法修改原有数组)。
        res.unshift(pNode.val);
        pNode = pNode.next;
    }
    return res;
}
```

4. P62，输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。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。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

- 思路：先找出根结点，左子树，右子树，在左子树，右子树中利用递归构造二叉树。
- 代码实现：

```
function reConstructBinaryTree(pre, vin) {
    //前序遍历和中序遍历长度为0或不匹配
    if (pre.length === 0 || vin.length === 0) {
```

```
        return null;
    }
    if(pre.length !== vin.length)
        return null;
    // 前序第一个是根节点，也是中序左右子树的分割点
    let index = vin.indexOf(pre[0]),//在中序列表中找到根结点并返回位置
        left = vin.slice(0, index),
        right = vin.slice(index + 1);
    return {
        val: pre[0],
        // 递归左右子树的前序、中序
        left: reConstructBinaryTree(pre.slice(1, index + 1), left),
        right: reConstructBinaryTree(pre.slice(index + 1), right)
    };
}
```

5. P68,用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

- 思路：队列先入先出，用两个栈，栈1压入元素，将栈1的元素逐个全部压入栈2中，栈2的栈顶元素出栈，也可以将栈1中的最后一个元素直接出栈。
- 代码实现：

```
let outStack = [],inStack = [];
function push(node) {//始终压入栈1
  inStack.push(node);
}
function pop() {
    if (!outStack.length) {//栈2为空
        if(inStack.length == 0)
            return null;
        while (inStack.length) {//栈1不为空时，逐个压入栈2
            outStack.push(inStack.pop());
        }
    }
    return outStack.pop();//栈2的栈顶元素为先入元素
}
```

6. P82,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。 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。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

- 思路：第一种方法：我们发现旋转数组在旋转后，有个分界点，而这个分界点就是最小的那个数,时间复杂度为O(n)。第二种方法：利用二分法时间复杂度是O(logn)，找到中间的数，然后和最左边的值比较，若大于最左边的数，则最左边从mid开始在右半部分继续查找；若小于最右边值，则最右边从mid开始，在左边查找。若左右中三值相等，则顺序遍历，取比前一位小的值。
- 代码实现：

```
//第一种方法
function minNumberInRotateArray1(rotateArray) {
    if (rotateArray.length === 0)
        return 0;
    for (let i = 0; i < rotateArray.length; i++) {
        if (rotateArray[i] > rotateArray[i + 1])
            return rotateArray[i + 1];
        }
    //所有的前面值都不大于后面值，说明前0个元素搬到末尾，则为原数组。此时返回第一个元素。
```

```
    return rotateArray[0];
}
//第二种方法
/*1、数组长度为0时，返回0；
2、数组长度为1时，返回这个唯一元素就好；
3、array[ 0 ] < array[ array.length - 1 ]，说明数组没有旋转，直接返回第一个元素
4、数组中没有重复元素，数组经过旋转后，前后两段子数组都是有序的，最小元素为第二个子数组的第一个元素，
用类似于二分查找的方法解决，left指向第一个元素，right指向最后一个元素，取中间数mid分别比较，
若array[mid]大于array[left]，说明array[mid]属于前一个子数组，最小元素在右边，
则将mid赋给left,在右边继续查找；
否则，若array[mid]小于array[right]，说明array[mid]属于后一个子数组，最小元素在左边，
则将mid赋给right，在左边继续查找；
如此循环，left最终将指向前一个子数组的最后一个元素，right将指向后一个子数组的第一个元素，
而right指向的这个元素就是最小元素。O(logn)
5、数组中有重复元素，如{3,3,3,2,3}，第一次循环left、mid、right指向的元素都相同，
这时我们还是需要遍历left、right之间的各数，找到最小元素。O(n)*/
function minNumberInRotateArray(rotateArray){
    if(rotateArray.length <= 1)
        return rotateArray.length <= 0 ? 0 : rotateArray[0];//情况1、2
    if(rotateArray[left] < rotateArray[right])
        return rotateArray[left];//情况3
    //正常情况（如题目中数组）,或者有相等元素的数组
    var left = 0;
    var right = rotateArray.length - 1;
    var mid;
    while(right - left >= 1){
        if(right - left === 1){//当两个下标相邻时，right指向的就是最小元素
            break;
        }
        mid = parseInt((left + right) / 2);
        //情况5，如{3,3,3,2,3}
        if(rotateArray[left] === rotateArray[mid] &&
            rotateArray[mid] === rotateArray[right]){
            var result = rotateArray[left];
            for(var i = left;i < right;i++){
                if(rotateArray[i] > rotateArray[i+1])
                    result = rotateArray[i+1];
            }
            return result;
        }else if(rotateArray[left] <= rotateArray[mid])//中间值大于等于左边{3,3,3,1,2}
            left = mid;
        else //if(rotateArray[mid] <= rotateArray[right])//中间值小于等于右边{3,3,1,1,1}
            right = mid;
    }
    return rotateArray[right];//情况4
}
```

7. P74，现在要求输入一个整数n，请你输出斐波那契数列的第n项。n<=39

- 思路：递归的效率低，用循环。动态规划的特点是：最优子结构、无后效性、子问题重叠。
- 代码实现：

```
function Fibonacci(n) {
    let f = 0,
```

```
        g = 1;
    while (n--) {
        //当前f(n)为f(n-1)+f(n-2),当n增加时，g的值变为当前的f(n-1),f的值变为当前的f(n-2)
        g += f;//直接叠加即可
        f = g - f;//f来记录f(n-1)
    }
    return f;
}
```

8. P77，一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（先后次序不同算不同的结果）。

- 思路：两种跳法，1阶或者2阶，假定第一次跳的是一阶，那么剩下的是n-1个台阶，跳法是f(n-1)；假定第一次跳的是2阶，那么剩下的是n-2个台阶，跳法是f(n-2)。可以得出总跳法为: f(n) = f(n-1) + f(n-2) 。只有一阶的时候 f(1) = 1 ,只有两阶的时候可以有 f(2) = 2。可以发现最终得出的是一个斐波那契数列。
- 代码实现：

```
function jumpFloor(number) {
    let f = 1,
        g = 2;
    while (--number) {
        //当前f(n)为f(n-1)+f(n-2),当n增加时，g的值变为当前的f(n-1),f的值变为当前的f(n-2)
        g += f;//直接叠加即可
        f = g - f;//f来记录f(n-1)
    }
    return f;
}
```

9. P78，一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。

- 思路：2^(n-1)
- 代码实现：

```
function jumpFloorII(number)
{
    let i = 1;
    while (--number) {
        i *= 2;
    }
    return i;
    //return Math.pow(2,number-1);
}
```

10. P79，我们可以用2*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*1的小矩形无重叠地覆盖一个2*n的大矩形，总共有多少种方法?

- 思路：2*1的大矩形，f(1)为1；2*2的大矩形，横着放或竖着放，f(2)为2；2*3的大矩形，先横着放，下面也只能横着，剩下的为f(1)，再竖着放，剩下的为f(2)，f(3)=f(2)+f(1)。以此类推，为斐波那契数列。
- 代码实现：

```
function rectCover(number)
{
    if(number <= 0) return 0;
    let f = 1,g = 2;
    while(--number){
        g = g + f;
```

```
        f = g - f;
    }
    return f;
}
```

11. P100，输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

- 思路：a&1!=0，判断最右边的那个位为1，然后a向右移位，但是这样输入负数时会陷入死循环，因为负数右移时，最高位补得是1，那么这样会有无数个1。
- 如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减1，那么原来处在整数最右边的1就会变为0，原来在1后面（右边）的所有的0都会变成1(如果最右边的1后面还有0的话)。其余所有位将不会受到影响。
- 举个例子：1100，减去1后，1011。我们发现减1的结果是把最右边的一个1开始的所有位都取反了。把原来的整数和减去1之后的结果做与运算，从原来整数最右边一个1那一位开始所有位都会变成0。如1100&1011=1000。a&(a-1)会把a最右边一个1变成0，直到a为0。那么一个整数的二进制有多少个1，就可以进行多少次这样的操作。
- 代码实现：

```
function NumberOf1(n)
{
    let count = 0;
    while(n != 0){
        count++;
        n = n & (n-1);
    }
    return count;
}
```

12. P110，给定一个double类型的浮点数base和int类型的整数exponent，求base的exponent次方。保证base和exponent不同时为0。

- 思路：快速求幂的方法。就是说我们要算a的11次方，我们只需要算a的1次方，a的2次方，a的8次方，也就是说我们结果需要算的是这个指数对应的二进制数上有1的位。
- 和1进行按位与，可以判断二进制数最右边的位数是否为1，因此也可以判断奇偶数，因为奇数最后一位一定为1。
- 代码实现：

```
function Power(base, exponent)
{
    let res = 1,n;
    if (exponent > 0) {
        // 指数大于0的情况下
        n = exponent;
    } else if (exponent < 0) {
        // 指数小于0的情况下
        if (!base) throw new Error('分母不能为0');
            n = -exponent;
     } else {
         // 指数等于0的情况下
         return 1;
     }
     while (n) {
         // 也可以用递归做，这里采用了循环
         if (n & 1)
         // 当指数为奇数时，包括了1
             res *= base;
             base *= base;
             n >>= 1;
    }
```

```
    return exponent > 0 ? res : 1 / res;
}
```

13. P129，输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

- 思路：判断是否为奇数，统计奇数个数，然后新建数组，把所有奇数存进去数组前面，剩下的存进去数组后面。
- 代码实现：

```
function reOrderArray(array)
{
    // oddBegin主要是用作奇数的索引，oddCount是用作偶数的索引,newArray用来存储，以空间换时间，复杂度为O(n)
    let oddBegin = 0,oddCount = 0;
    let newArray = [];
    for(let i = 0;i < array.length;i++){
        if(array[i] & 1){
            oddCount++;
        }
    }
    for(let i = 0;i < array.length;i++){
        if(array[i] & 1){
            newArray[oddBegin++] = array[i];
        }else{
            newArray[oddCount++] = array[i];
        }
    }
    return newArray;
}
```

14. P134，输入一个链表，输出该链表中倒数第k个结点。

- 思路：定义一快一慢两个指针，快指针走到第k个结点，然后慢指针开始走，快指针到结尾时，慢指针就找到了倒数第k个结点。
- 代码实现：

```
function FindKthToTail(head, k)
{
    if(head == null || k <= 0){ //链表为空，k不合法时
        return null;
    }
    let fast = head,slow = head;
    while(k-- > 1){//从第一个结点开始，走k-1步
        if(fast.next != null){//判断总节点不少于k
            fast = fast.next;
        }
        else return null;
    }

    while(fast.next != null){
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```

15. P142，输入一个链表，反转链表后，输出新链表的表头。

- 思路：至少需要三个指针pPre（指向前一个结点）、pCurrent（指向当前的结点，在代码中就是pHead）、pPnext（指向后一个结点）。
- 代码实现：

```
function ReverseList(pHead)
{
    let pPre = null,pNext = null;
    while (pHead !== null) {
        pNext = pHead.next;
        pHead.next = pPre;
        pPre = pHead;
        pHead = pNext;
    }
     return pPre;
}
```

16. P145，输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

- 思路：只需要不断地比较他们的头结点，明显这是个重复的过程。可以用递归做，也可以不用递归做，不用递归做只需要用两个指针来一直指向两个链表的“头”结点就行了
- 代码实现：

```
function Merge(pHead1, pHead2)
{
    if(pHead1 === null)
        return pHead2;
    if(pHead2 === null)
        return pHead1;
    let newHead = null;
    if(pHead1.val <= pHead2.val){
        newHead = pHead1;
         newHead.next = Merge(pHead1.next, pHead2);
     }else{
         newHead = pHead2;
         newHead.next = Merge(pHead1, pHead2.next);
     }
     return newHead;
}
```

17. P148，输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

- 思路：在树A中找到和B的根结点的值一样的结点R；判断树A中以R为根结点的子树是不是包含和树B一样的结点。
- 代码实现：

```
function HasSubtree(pRoot1, pRoot2)
{
    let result = false;
    if(pRoot2 !== null && pRoot1 !== null){
         //根结点值相同，则判断子树结构是否相同，
        if(pRoot1.val === pRoot2.val){
            result = doseTree1HasTree2(pRoot1,pRoot2);
        }
        //若根结点不相同或者不是子结构，到A的左右子树中查找是否有和B根结点相同值的结点
         if(!result){
             result = HasSubtree(pRoot1.left, pRoot2) ||
```

```
                HasSubtree(pRoot1.right, pRoot2);
            }
        }
        return result;
    }
    //判断子树结构的函数
    function doseTree1HasTree2(pRoot1,pRoot2){
        //B中结点为空，则为true
        if(pRoot2 === null) return true;
        //B不为空，A中为空则为false。
        if(pRoot1 === null) return false;
        //都不为空但值不相同，返回false
        if(pRoot1.val !== pRoot2.val){
            return false;
        }
        //值相同继续比较子树
        return doseTree1HasTree2(pRoot1.left,pRoot2.left) &&
            doseTree1HasTree2(pRoot1.right,pRoot2.right);
    }
```

18. P157，操作给定的二叉树，将其变换为源二叉树的镜像。

- 思路：递归或循环交换每个节点的左右子树的位置
- 代码实现：

```
function Mirror(root)
{
    if (root === null) return;
    [root.left, root.right] = [root.right, root.left];
    Mirror(root.left);
    Mirror(root.right);
    return root;
}
```

19. P161，输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下4 X 4矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

- 思路：什么时候该继续打印下一圈即里面那一圈，也就是要找到循环的条件。那么什么是循环结束的条件呢？我们注意4X4的矩阵，只有两圈，到从第一圈到第二圈，起点从（0,0）变为了（1,1），我们发现4>1*2,类似的对于一个5X5的矩阵而言，最后一圈只有一个数字，起点坐标为（2,2），满足5>2*2，同理对于6X6的矩阵也是类似。故可以得出循环的条件就是columns>startX*2并且rows>startY*2。
- 实现打印一圈，注意边界条件，从左到右，从上到下（终止行号大于起始行号），从右到左（终止列号大于起始列号，且终止行号大于起始行号），从下到上（终止行号比起始行号至少大2，且终止列号大于起始列号）。
- 思路二：模拟魔方逆时针解法。
- 例如
  - 1 2 3
  - 4 5 6
  - 7 8 9
  - 输出并删除第一行后，再进行一次逆时针旋转，就变成：
  - 6 9
  - 5 8
  - 4 7
  - 继续重复上述操作即可
- 代码实现：

```
//第一种
```

```
function printMatrix(matrix)
{
    if(matrix === null) return;
    let start = 0,res = [];
    const row = matrix.length,col = matrix[0].length;
    //循环的条件就是columns>startX*2并且rows>startY*2。
    while(row > start * 2 && col > start * 2){
        printCircle(matrix,row,col,start,res);
        start++;
    }
    return res;
}

function printCircle(matrix,row,col,start,res){
    let endX = col - 1 - start,
        endY = row - 1 - start;
    //实现打印一圈，注意边界条件。
    //从左到右，
    for(let i = start;i <= endX;i++){
        res.push(matrix[start][i]);
    }
    //从上到下（终止行号大于起始行号），
    if(endY > start){
        for(let i = start+1;i <= endY;i++){
            res.push(matrix[i][endX]);
        }
    }
    //从右到左（终止列号大于起始列号，且终止行号大于起始行号），
    if(endY > start && endX > start){
        for(let i = endX-1;i >= start;i--){
            res.push(matrix[endY][i]);
        }
    }
    //从下到上（终止行号比起始行号至少大2，且终止列号大于起始列号）。
    if(endX > start && endY > start + 1){
        for(let i = endY-1;i > start;i--){
            res.push(matrix[i][start]);
        }
    }
}
```

20. P165，定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。

- 思路：增加了一个辅助栈，每次压入数据栈时，把当前栈里面最小的值压入辅助栈当中。这样辅助栈的栈顶数据一直是数据栈中最小的值。
- 代码实现：

```
let stack = [],minStack = [];
let temp = null;//记录当前最小值
function push(node)
{
    //入栈节点与当前最小值比较，比最小值小入最小值栈，
    if(temp !== null){
        //否则上一个最小值再次入栈
```

```
        if(node < temp){
            temp = node;
        }
        stack.push(node);
        minStack.push(temp);
    }else{//最小值栈为空时，直接入栈，
        temp = node;
        stack.push(node);
        minStack.push(temp);
    }
}
function pop()
{
    //弹出一个元素
    stack.pop();
    //最小值栈也弹出一个元素
    minStack.pop();
}
function top()
{
    return stack[stack.length - 1];//栈顶
}
function min()
{
    return minStack[minStack.length - 1];//最小值栈顶
}
```

21. P168，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可能为该栈的弹出顺序。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。（注意：这两个序列的长度是相等的）

- 思路：建立一个辅助栈按照压入顺序依次压入元素，每压入一次判断是否和弹出序列第一个元素相等，相等则弹出，弹出序列删除第一个元素，不相等则继续压入，压入了全部元素却没有和弹出序列匹配完，没有完全弹出，即辅助栈不为空则不匹配。
- 代码实现：

```
function IsPopOrder(pushV, popV)
{
    //建立一个辅助栈按照压入顺序依次压入元素
    let helpStack = [];
    let flag = false;
    while (pushV.length || helpStack.length) {
      //判断压入元素是否和弹出序列第一个元素相等，相等则弹出，弹出序列删除第一个元素
      while (helpStack[helpStack.length - 1] === popV[0] && helpStack.length) {
          helpStack.pop();
          popV.shift();
      }
      if (!popV.length) {//弹出序列全部匹配完
          flag = true;
      }
      if (!pushV.length) {//压入序列匹配完，跳出循环
          break;
      }
      helpStack.push(pushV.shift());//不相等则继续压入
```

```
    }
    return flag;
}
```

22. P171，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。

- 思路：从下打印就是按层次打印，其实也就是树的广度遍历。一般来说树的广度遍历用队列，利用先进先出的特点来保存之前节点，并操作之前的节点。树的深度遍历一般来说用栈或者递归，利用先进后出的特点来保存之前节点，把之前的节点留到后面操作。
- 代码实现：

```
function PrintFromTopToBottom(root)
{
    let res = [];
    let queue = [];
    //利用队列先进先出的特点
    if(root === null) return res;
    //先将根结点入队
    queue.push(root);
    //当队列不为空时，队头出队，每出队一个节点，都将他的左右子节点入队
     while(queue.length){
         let node = queue.shift();
         if(node.left !== null) queue.push(node.left);
         if(node.right !== null) queue.push(node.right);
         res.push(node.val);
    }
    return res;
}
```

23. P179，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。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

- 思路：二叉搜索树的特点是有序，后序遍历最后一个值是根节点r。通过根节点r我们可以判断左子树和右子树。判断左子树中的每个值是否小于r,右子树的每个值是否大于r。对左、右子树递归判断。
- 代码实现：

```
function VerifySquenceOfBST(sequence)
{
    if (!sequence.length) return false;
    return judge(sequence, 0, sequence.length - 1);
}
function judge(arr, left, root) {
    if (left >= root) return true;
    let i = root;
    //从后往前找，比根大的都是右子树中的节点，找到右子树的起始点i
     while (arr[i - 1] > arr[root] && i > left) i--;
     //左子树从后往前找，左子树中有大于根的节点，则返回false
     for (let j = i - 1; j >= left; j--) {
         if (arr[j] > arr[root]) return false;
     }
     //递归左右子树
     return judge(arr, left, i - 1) && judge(arr, i, root - 1);
}
```

24. P182，输入一颗二叉树的根节点和一个整数，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。(注意: 在返回值的list中，数组长度大的数组靠前)

- 思路：深度遍历整个树，并且把已经走过的节点的和与期望值作比较，如果走到底还不符合要求的话，就要回退值。
- 代码实现：

```
function FindPath(root, expectNumber)
{
    const list = [],
        listAll = [];
    return findpath(root, expectNumber, list, listAll);
}
function findpath(root, expectNumber, list, listAll) {
    if (root === null) {
        return listAll;
     }
     list.push(root.val);
     const x = expectNumber - root.val;
     if (root.left === null && root.right === null && x === 0) {
         listAll.push(Array.of(...list));
         //Array.of()是静态方法，也返回一个数组。
         //Array.of(...elements) 创建一个具有可变数量参数的新的数组实例。
     }
     findpath(root.left, x, list, listAll);
     findpath(root.right, x, list, listAll);
     list.pop();//每返回上一层就回退一个节点
     return listAll;
}
```

25. P187，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。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

- 思路：解法一：先复制节点，用p.next连接起来，然后再去设置p.random指针指向，不过这个设置又需要从头节点开始查。总的时间复杂度为O（n2）
- 解法二：用map来保存<N，N`>，这样就很容易设置p.random了，比如我们在节点S处和节点S`处，我们通过S可以得到N，那么<N，N`>对应，就可以使得S`的next指向N`了。这是通过空间换时间
- 解法三：比较复杂些，但是不用空间换时间也能达到O（n）。第一步，对链表的每个节点N创建N‘，并链接在N的后面。设置复制出来的p.random。节点N指向S，那么N'指向S'，而N和N'相邻，那么S和S'也相邻。把长链表拆开成两个链表。取偶数位置的节点为复制的节点。
- 代码实现：

```
function Clone(pHead)
{
    cloneNodes(pHead);
    cloneRandom(pHead);
    return reconnectNodes(pHead);
}
//1、复制每个结点，如复制结点A得到A1，将结点A1插到结点A后面；
function cloneNodes(pHead) {
    let currentNode = pHead;
    while(currentNode !== null){
        //currentNode.label为当前节点的值，复制当前节点
        let cloneNode = new RandomListNode(currentNode.label);
        cloneNode.next = currentNode.next;//复制的节点指向原链表下一个节点
        currentNode.next = cloneNode;//链接到原节点之后
        currentNode = cloneNode.next;
```

```
    }
}
function cloneRandom(pHead){
    let currentNode = pHead;
    //2、重新遍历链表，复制老结点的随机指针给新结点，如A1.random = A.random.next
    while(currentNode !== null) {
        //如果currentNode.random不为null,
        //那么将currentNode.next.random赋值为currentNode.random.next，建立引用关系
        if(currentNode.random !== null){
            currentNode.next.random = currentNode.random.next;
        }
        currentNode = currentNode.next.next;
    }
}

//3、拆分链表，将链表拆分为原链表和复制后的链表
function reconnectNodes(pHead){
    let currentNode = pHead;
    let pCloneHead = null;
    let cloneNode = null;
    if(currentNode !== null){
        pCloneHead = pHead.next;//新链表的头节点为原链表的头节点的next
        cloneNode = pHead.next;//指向第一个复制节点
        currentNode.next = cloneNode.next;//原链表第一个连接，复制节点的next为原链表的next
        currentNode = cloneNode.next;//当前指向第二个原链表的节点
    }
    while(currentNode !== null) {
         //在原来未拆分之前的链表上进行拆分的操作
         cloneNode.next = currentNode.next;//复制链表的连接，原节点的next为复制节点next
         cloneNode = cloneNode.next;//复制链表的指针后移
         currentNode.next = cloneNode.next;//复制节点的next为原链表的next
         currentNode = currentNode.next;//当前节点后移
     }
     return pCloneHead;
}
```

26. P191，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。

- 思路：要生成排序的双向列表，只能是中序遍历，因为中序遍历才能从小到大，所以需要递归，先对左子数调整为双向链表，并用变量pLast指向最后一个节点。再将根结点和pLast连起来。再去调整右子树，将右子树的第一个节点和根连起来。
- 代码实现：

```
function Convert(pRootOfTree)
{
    if (pRootOfTree === null) return null;
    let pLast = null;//定义一个指针指向双向链表的最后一个节点
    pLast = ConvertNode(pRootOfTree, pLast);
    let pHead = pLast;
    //返回的结果pHead指向双向链表的最后一个节点
    while (pHead && pHead.left) {//链表不为null且向左循环不为空，.left为空时，pHead指向头节点
        pHead = pHead.left;
    }
    return pHead;
```

```
}

function ConvertNode(pNode, pLast) {
    if (pNode === null) return;

    if (pNode.left) {//左子树不为空，递归排序左子树
        pLast = ConvertNode(pNode.left, pLast);
    }
    pNode.left = pLast;//根连在左子树之后
    if (pLast) {
        pLast.right = pNode;
    }
    pLast = pNode;//如果右子树为空，最后一个节点为根结点
    if (pNode.right) {//右子树不为空，递归排序右子树
        pLast = ConvertNode(pNode.right, pLast);
    }
    return pLast;//返回最后一个节点
}
```

27. P197，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。长度不超过9(可能有字符重复),字符只包括大小写字母。

- 思路：递归全排列法：把字符串分为两部分：第一部分为第一个字符，第二部分为第一个字符以后的字符串。然后求后面那部分的全排列。再将第一个字符与后面的那部分字符逐个交换
- 回溯法：利用树去尝试不同的可能性，不断地去字符串数组里面拿一个字符出来拼接字符串，当字符串数组被拿空时，就把结果添加进结果数组里，然后回溯上一层。（通过往数组加回去字符以及拼接的字符串减少一个来回溯。）
- 代码实现：

```
function Permutation(str) {
    let res = [];
    if (str.length <= 0) return res;
    let arr = str.split(''); // 将字符串转化为字符数组
    permutate(arr, 0, res);
    //ES6 提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
    res = [...new Set(res)]; //去重
    res.sort(); // 排序
    return res;
}

function permutate(arr, index, res) {
    if (index === arr.length - 1) {//递归的出口是全排列到最后一个字符
        let s = '';
        for (let i = 0; i < arr.length; i++) {
            s += arr[i];
        }
        res.push(s);
      }
    for (let i = index; i < arr.length; i++) {//index和剩下的每一个字符交换
        //if(strArr[i] === strArr[index] && i !== index)//去重的另一种方法
            //continue;//不同位置的元素和index相等，跳过该元素
        [arr[index], arr[i]] = [arr[i], arr[index]]; // es6支持交换
        permutate(arr, index + 1, res);//固定index，递归全排列剩下的字符
        [arr[index], arr[i]] = [arr[i], arr[index]]; // 交换，回退
    }
```

```
}
```

28. P205，数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

- 思路：第一种：基于快排思想中的partition函数来做，排序后的数组中间的数就是那个出现次数超过一半的数，验证中间的数是不是超过长度一半。修改原数组。
- 第二种：根据数组特点来做，数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现的次数的和还要多。注意检查是不是符合要求。
- 代码实现：

```
//第一种
function MoreThanHalfNum_Solution(numbers)
{
    //基于partition
    const len = numbers.length;
    let index = 0,mid = len >> 2;
    index = partition(numbers,0,len - 1);
    while(index !== mid){
        if(index > mid){
            index = partition(numbers,0,index - 1);
        }else{
            index = partition(numbers,index + 1,len-1);
        }
    }
    if(checkMoreThanHalf(numbers,mid))
        return numbers[mid];
    return 0;
}
function partition(arr,start,end){
    let key = arr[start];// 一开始让key为第一个数
    while(start < end){//扫描一遍
        // 如果key小于arr[end]，则end递减，继续比较
        while(key <= arr[end] && start < end) end--;
        [arr[end],arr[start]] = [arr[start], arr[end]];//arr[end]小于key,交换
        // 如果key大于arr[start]，则start递增，继续比较
        while(key >= arr[start] && start < end) start++;
        [arr[end],arr[start]] = [arr[start], arr[end]];//arr[start]大于key，交换
    }
    return end;//把key现在所在的下标返回。start也可以。因为循环结束，start和end指向同一位置
}
function checkMoreThanHalf(numbers,mid){
    let counts = 0;
    for(let i = 0;i < numbers.length;i++){
        if(numbers[i] === numbers[mid]){
            counts++;
        }
    }
    if(counts*2 > numbers.length )
       return true;
    return false;
}
//第二种
function MoreThanHalfNum_Solution(numbers){
```

```
    let num = numbers[0];
    let count = 1;
    for(let i = 1;i < numbers.length;i++){
        if(numbers[i] !== num){
            if(count === 0){
                num = numbers[i];
                count++;
            }else{
                count--;
            }
        }else{
            count++;
        }
    }
    if(checkMoreThanHalf(numbers,num))
        return num;
    return 0;
}
function checkMoreThanHalf(numbers,num){
    let counts = 0;
    for(let i = 0;i < numbers.length;i++){
        if(numbers[i] === num){
            counts++;
        }
    }
    if(counts*2 > numbers.length )
       return true;
    return false;
}
```

29. P209，输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

- 思路：第一种，基于快排Partition方法，index为k。比k小的放前面，比k大的放后面。输出前k个数。改变数组O(n)
- 第二种，堆或红黑树。先将前k个数放入数组，进行排序，之后的数与最大值比较，若比他小则替换。O(nlogk)适合处理海量数据，不改变原有数据，只要容纳k的辅助空间，适合n比较大k比较小的问题
- 代码实现：

```
//第一种，基于快排Partition方法
function GetLeastNumbers_Solution(input, k)
{
    if (input.length === 0 || k > input.length || k < 1) return [];
    let res = [];
    let index = Partition(input,0,input.length - 1);
    while(index !== k - 1){
        if(index > k - 1){
            index = Partition(input,0,index - 1);
        }else{
            index = Partition(input,index + 1,input.length - 1);
        }
    }
    res = input.slice(0, index + 1);
    res.sort((a, b) => a - b);
    return res;
}
```

```
function Partition(input,start,end){
    let key = input[start];
    while(start < end){
        while(start < end && input[end] >= key) end--;
        [input[start],input[end]] = [input[end],input[start]];
        while(start < end && input[start] <= key) start++;
        [input[start],input[end]] = [input[end],input[start]];
    }
    return end;
}
//排序方法
function GetLeastNumbers_Solution(input, k){
    if(input.length === 0 || k > input.length || k < 1) return [];
    input.sort((a,b)=>(a-b));
    let res = input.slice(0,k);
    return res;
}
//第二种，最大堆
function GetLeastNumbers_Solution(input, k){
    if(input.length === 0 || k > input.length || k < 1) return [];
    let heap = input.slice(0,k);//先将前k个数放入数组。
    buildHeap(heap);//建立最大堆
    for(let i = k;i < input.length;i++){//遍历剩下的数字
        if(input[i] < heap[0]){//如果比堆顶小
            heap[0] = input[i];//替换堆顶
            maxHeap(heap,0);//调整最大堆
        }
    }
    heap.sort((a,b)=>(a-b));
    return heap;
}
function buildHeap(arr){
    for(let i = arr.length / 2 - 1;i>=0;i--){//从第一个非叶子节点向上调整
        maxHeap(arr,i);
    }
}
function maxHeap(arr,i){
    let left = 2*i + 1,//左孩子
        right = left + 1,//右孩子
        largest = 0;//记录最大值

    if(left < arr.length && arr[left] > arr[i]){//存在左孩子且大于根
        largest = left;//最大为左孩子
    }else largest = i;//否则最大值为根

    if(right < arr.length && arr[right] > arr[largest]){//存在右孩子且大于最大值
        largest = right;//最大为右孩子
    }

    if(largest !== i){//最大值不是根，把最大值换到根的位置
        [arr[largest],arr[i]] = [arr[i],arr[largest]];
        maxHeap(arr,largest);//交换后向下调整
```

```
    }
}
```

30. P218，求连续子数组的最大和，包含负数。

- 思路：如果用函数sum(i)表示以第i个数字结尾的子数组的最大和，那么我们只需要求出max[sum(i)]，其中可以用如下递归公式。当sum(i-1)<=0时，sum(i)=array[i];sum(i-1)>0时，sum(i)=sum(i-1)+array[i];当前子数组的和为负值，则将最大值置为下一个值。否则累加。
- 代码实现：

```
function FindGreatestSumOfSubArray(array)
{
    if(array === null ||array.length === 0) return 0;
    let cur = array[0],//当前子数组的和
        max = array[0];//最大和

    for(let i = 1;i < array.length;i++){
        if(cur <= 0){
            cur = array[i];
         }else{
             cur += array[i];
         }
         if(cur > max){
             max = cur;
         }
     }
     return max;
}
```

31. P221，求从1到整数n中1出现的次数。

- 思路：若当前位上的数字为零，则当前位出现1的次数为：高位数字*当前位；若当前位上的数字为1，则当前位出现1的次数为：高位数字*当前位+低位数字+1；若当前位上的数字大于1，则当前位出现1的次数为：（高位数字+1）*当前位；
- 方法二：公式法
- 代码实现：

```
function NumberOf1Between1AndN_Solution(n)
{
    let count = 0,
        i = 1,//当前位，从个位开始
        before = 0,current = 0,after = 0;//高位数字，当前位数字，低位数字

    while(parseInt(n/i) !== 0){
        before = parseInt(n/(i*10));
        current = parseInt(n/i%10);
         after = parseInt(n%i);

         if(current === 0){
             count += before * i;
         }else if(current === 1){
             count += before * i + after + 1;
         }else{
             count += (before + 1) * i;
         }
         i = i * 10;//向前移一位
     }
```

```
    return count;
}
//第二种，本质和第一种一样
//~是按位取反运算，~~是取反两次，~~的作用是去掉小数部分
//因为位运算的操作值要求是整数，其结果也是整数，所以经过位运算的都会自动变成整数
//除了~~n 还可以用n<<0，n>>0，n|0取整
//与Math.floor()不同的是，它只是单纯的去掉小数部分，不论正负都不会改变整数部分
function NumberOf1Between1AndN_Solution(n){
    if(n<=0) return 0;
    let count=0;
    for(let i=1;i<=n;i*=10){//向前移位
    //ab即为n，和上述高低位不太一样，a等于上面的高位+当前位数字，b为低位
        let a = ~~(n/i),b = n%i;
        //分为0，1和>=2三种情况
        //0：a/10*i
        //1：a/10*i+b+1
        //>=2:(a+1)/10*i
        //提取共同点，a+8为了让>=2的情况进一位，当前位为1时还要加上b+1
        count = count + ~~((a+8)/10)*i + (a%10===1)*(b+1);
    }
    return count;
}
```

32. P227，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。

- 思路：先把整型数组替换成字符串数组，对字符串数组按指定规则排序，将排好序的字符串数组拼接出来。排序规则a和b：ab大于ba,则a>b；ab小于ba,则a<b；ab等于ba,则a=b。
- 代码实现：

```
function PrintMinNumber(numbers) {
    numbers.sort(function(s1, s2) {//定义排序规则
        let c1 = `${s1}${s2}`;
        let c2 = `${s2}${s1}`;
        return c1 > c2;
      });
    let min = '';
    numbers.forEach((i) => min += i);
    return min;
}
```

33. P240，把只包含质因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含质因子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。

- 思路：动态规划的思想，把前面的丑数存着，生成后面的丑数。t2,t3,t5是判断点，用于判断从何处开始选出并乘以对应因子肯定会大于当前数组中最大丑数，而前面的丑数不用考虑。
- 将每个丑数分别乘以2，3，5得到后面的丑数。将结果按从小到大排序。记当前最大的丑数为M,则下一个丑数则是前面某一个丑数分别乘以2，3，5，大于M的三个数中的最小者。不需要把当前已有的所有丑数都乘以2，3，5；因为肯定存在某一个丑数T2，他前面的所有丑树乘以2都小于M,他之后的丑数乘以2都太大,所以记这个丑数为T2，同理记T3,T5。之后只要更新T2,T3,T5即可
- 代码实现：

```
function GetUglyNumber_Solution(index)
{
    if(index <= 0) return 0;
```

```
    let res = [];
    res[0] = 1;
    let t2 = 0,
        t3 = 0,
        t5 = 0;

     for(let i = 1;i < index;i++){
         let min = Math.min(res[t2]*2, res[t3]*3, res[t5]*5);
         res[i] = min;
         if(min === res[t2] * 2) t2++;
         if(min === res[t3] * 3) t3++;
         if(min === res[t5] * 5) t5++;
         }
     }
     return res[index-1];
}
```

34. P243，在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）.

- 思路：用map保存字符和出现的次数
- 代码实现：

```
function FirstNotRepeatingChar(str) {
    if (str.length < 1 || str.length > 10000) return -1;
    let map = {};
    for (let i = 0; i < str.length; i++) {
        if (!map[str[i]]) {
            map[str[i]] = 1;
        } else {
            map[str[i]]++;
        }
     }
     for (let i = 0; i < str.length; i++) {
         if (map[str[i]] === 1) {
             return i;
         }
     }
     return -1;
}
```

35. P249，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P。并将P对1000000007取模的结果输出。 即输出P%1000000007

- 思路：基于归并排序，参数的问题，这里很巧妙地用了一个copy数组作为data参数。合并时，count如何计数。
- 代码实现：

```
let count = 0;
function InversePairs(data) {
    if (!data || data.length < 2) return 0
    count = mergeCount(data,0, data.length - 1);
    count %= 1000000007;
    return count;
}

function mergeCount(data,start,end){
```

```
    let mid = (start + end) >> 1;
    if(start < end){
        mergeCount(data,start,mid);//递归分割子数组
        mergeCount(data,mid+1,end);
        merge(data,start,mid,end);//递归从最小的子数组归并排序
    }
    return count;
}

//归并操作，把两个子数组按序复制到一个结果数组的归并过程。空间O(n),时间O(nlgn)
function merge(data,start,mid,end){
    let res = [];//存储归并结果
    let c = 0,s = start,index = mid+1;
    while(start <= mid && index <= end){//两个有序子数组的范围
        //前一个子数组值比后一个子数组中最小值都小，所以没有逆序对
        if(data[start] < data[index]){
            res[c++] = data[start++];//前一个子数组的值复制到结果数组中
        }else{//前一个子数组最小值比后一个子数组中最小值大，
        //说明该值之后的每个值和后一个子数组最小值都构成逆序对，逆序对的个数为第一个子数组的长度
            res[c++] = data[index++];
            count += mid + 1 - start;
        }
    }

    //一个子数组已经归并完，另一个子数组还有数据，顺序放入结果数组即可
    while(start <= mid){
        res[c++] = data[start++];
    }
    while(index <= end){
        res[c++] = data[index++];
    }

    for(let d of res){
        data[s++] = d;
    }
}
```

36. P253，输入两个链表，找出它们的第一个公共结点。

- 思路：两种情况：
- 1，当两个链表长度相同时，索引比较对应节点，找到第一个相同节点返回，否则返回空
- 2，当两个链表长度不同时，索引比较对应节点，直到短的链表为空，短链表指针p1指向长链表头节点，继续和长链表指针p2向下比较，直到长链表为空，此时p1正好先走了n步，这里的n为两个链表的长度差。p2指向短链表的头节点，此时同时向下比较，有相同节点返回，没有返回空。
- 代码实现：

```
function FindFirstCommonNode(pHead1, pHead2)
{
    let p1 = pHead1,
        p2 = pHead2;
    while(p1 !== p2){
        p1 = p1 === null ? pHead2 : p1.next;
        p2 = p2 === null ? pHead1 : p2.next;
    }
```

```
    return p1;
}
```

37. P263，统计一个数字在排序数组中出现的次数。

- 思路：因为数组有序，所以k集中在某一段，二分查找+递归。从中间开始找k，若中间值大于k，则递归在左边查找；中间值小于k，递归在右边查找；中间值等于k，则从中间到数组结尾，遍历到不为k为止，并计数。从中间到数组开始遍历到不为k为止，并计数
- 代码实现：

```
function GetNumberOfK(data, k)
{
    let result = 0;
    if(data.length <= 0) return 0;
    if(data.length === 1){
        if(data[0] === k)
            return 1;
        return 0;
    }

     let len = data.length,
         mid = len >> 1;
     if(k < data[mid]){
         result = GetNumberOfK(data.slice(0,mid), k);
     }else if(k > data[mid]){
         result = GetNumberOfK(data.slice(mid+1,len), k);
     }else{
         for(let i = mid;i >= 0;i--){
             if(data[i] === k)
                 result++;
             else break;
         }
         for(let i = mid+1;i <= len;i++){
             if(data[i] === k)
                 result++;
             else break;
         }
     }
     return result;
}
```

38. P271，输入一棵二叉树，求该树的深度。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。

- 思路：深度为左右子树中深度较大的值+1
- 代码实现：

```
function TreeDepth(pRoot)
{
    if(pRoot === null) return 0;
    let left = TreeDepth(pRoot.left);
    let right = TreeDepth(pRoot.right);
    return left > right ? left + 1 : right + 1;
}
```

39. P273，输入一棵二叉树，判断该二叉树是否是平衡二叉树。

- 思路：第一种：正常思路，应该会获得节点的左子树和右子树的高度，然后比较高度差是否小于1。节点重复遍历，影响效率。第二种：在求高度的同时判断是否平衡，如果不平衡就返回-1，否则返回树的高度。并且当左子树高度为-1时，就没必要去求右子树的高度了，可以直接一路返回到最上层。
- 代码实现：

```
function IsBalanced_Solution(pRoot) {
    return TreeDepth(pRoot) !== -1;
}

function TreeDepth(pRoot) {
    if (pRoot == null) return 0;
    let leftLen = TreeDepth(pRoot.left);
    if (leftLen == -1) return -1;
    let rightLen = TreeDepth(pRoot.right);
     if (rightLen == -1) return -1;
     return Math.abs(leftLen - rightLen) > 1 ? -1 : Math.max(leftLen, rightLen) + 1;
}
```

40. P275，一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。请写程序找出这两个只出现一次的数字。

- 思路：第一种：使用js中的indexOf()和lastIndexOf()，只要两个相等，就是只出现一次的数。
- 第二种：使用map记录下每个数的次数，占空间。和34题类似。
- 第三种：两个相同的数异或之后为0，把数组中所有的数异或运算，得到的结果是两个只出现一次的数的异或结果，因为出现两次的都抵消了。这个结果中的1表示两个数不同的位异或出的1。求出异或结果的1在第index位。根据index是0还是1把数组分成两个子数组。出现两次的数肯定被分在同一个数组，两个落单的数一定会被分开，因为index位不可能相同。将两个子数组所有数异或。出现两次的数抵消，剩下的是出现一次的数。
- 代码实现：

```
//第一种
function FindNumsAppearOnce(array)
{
    // return list, 比如[a,b], 其中ab是出现一次的两个数字
    let res = [];
    for(let i = 0;i < array.length;i++){
        if(array.indexOf(array[i]) === array.lastIndexOf(array[i]))
            res.push(array[i]);
    }
     return res;
}
//第二种
function FindNumsAppearOnce(array)
{
    let map = {};
    let res = [];
    for(let i = 0;i < array.length;i++){
        if(!map[array[i]]){
            map[array[i]] = 1;
        }else{
            map[array[i]]++;
        }
    }

    for(let i = 0;i <array.length;i++){
        if(map[array[i]] === 1){
```

```
            res.push(array[i]);
        }
    }
    return res;
}
//第三种
function FindNumsAppearOnce(array)
{
    let num = array[0];
    //所有数异或得出一个值
    for(let i= 1;i < array.length;i++){
        num = num ^ array[i];
    }
    if(num === 0) return;

    //求结果值1所在的index
    let index = 0;
    while((num&1) === 0){
        num = num >> 1;
        index++;
    }
    //根据index位上为1/0分为两组，异或
    let num1 = 0,num2 = 0;
    for(let i = 0;i < array.length;i++){
        let flag = ((array[i] >> index) & 1) === 1;
        if(flag){
            num1 = num1 ^ array[i];
        }else{
            num2 = num2 ^ array[i];
        }
    }
    return [num1,num2];
}
```

41. P282，输出所有和为S的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

- 思路：定义两个指针small指向第一个数字，big指向第二个数字；和小于sum时，big++，使和增大；较大时，small++，使和减少；因为序列至少有两个数字，所以small增加到（sum+1）/2为止。
- 代码实现：

```
function FindContinuousSequence(sum)
{
    let small = 1,big = 2,curSum = small + big ;
    let res = [],temp;
    if(sum < 3) return res;
    while(small < (sum + 1) / 2){
        if(curSum > sum){
            curSum -= small;
            small++;
        }else if(curSum < sum){
            big++;
            curSum += big;
        }else{
            temp = [];//把上一次符合的序列清空
```

```
            for(let i = small;i <= big;i++){
                temp.push(i);
            }
            res.push(temp);
            curSum -= small;//去掉最低值，继续向前找其他符合要求的序列
            small++;
        }
    }
    return res;
}
```

42. P280，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，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。

- 思路：定义两个指针，分别从前面和后面遍历，间隔越远乘积越小，最先出现的两个数乘积最小。
- 代码实现：

```
function FindNumbersWithSum(array, sum)
{
    let left = 0,right = array.length - 1;
    let res = [];
    let curSum;
    if(array.length < 2) return [];

    while(left < right){
        curSum = array[left] + array[right];
        if(curSum === sum){//输出第一对即可
            res.push(array[left],array[right]);
            return res;
        }else{
            if(curSum < sum) left++;
            else right--;
        }
    }
    return res;
}
```

43. P286，左旋字符串，对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。

- 思路：slice()方法，截取字符串前k位，拼接在剩下字符后面
- 代码实现：

```
function LeftRotateString(str, n)
{
    if (str === null || str.length === 0) return '';
    n = n % str.length;//左旋的位数大于字符串长度，取余
    return str.slice(n) + str.slice(0, n);
}
```

44. P284，翻转字符串，翻转单词顺序

- 思路：split() 方法使用指定的分隔符字符串将一个String对象分割成字符串数组，以将字符串分隔为子字符串，以确定每个拆分的位置。 reverse() 方法将数组中元素的位置颠倒，并返回该数组。该方法会改变原数组。join() 方法将一个数组（或一个类数组对象）的所有元素连接成一个字符串并返回这个字符串。如果数组只有一个项目，那么将返回该项目而不使用分隔符。
- 代码实现：

```
function ReverseSentence(str) {
    return str
    .split(' ')//以空格分割单词返回数组
    .reverse()
    .join(' ');//以空格连接所有单词
}
```

45. P298，扑克牌的顺子，从扑克牌中随机抽5张，判断是不是一个顺子，即是否连续，A为1，2-10位数字本身，J为11，Q为12，K为13。大小王可以是任意数字。 如果牌能组成顺子就输出true，否则就输出false。为了方便起见，可以认为大小王是0。

- 思路：满足题目要求我们需要满足两个条件：最大和最小之差不超过5。这5个数当中不能有重复的数字。利用位运算的技巧来判断是否有数字重复。因为我们的数字范围为1-13，用每一个bit对应一个数字，如果出现过，那么flag上这个bit就为1，不然就是0。
- 代码实现：

```
function IsContinuous(numbers) {
    let max = 0,
        min = 14,
        flag = 0;
    if (numbers.length !== 5) return false;
    for (let i = 0; i < numbers.length; i++) {
        if (numbers[i] > 13 || numbers[i] < 0) return false;//数字不合法
        if (numbers[i] === 0) continue;//是0就跳过
        //位运算判断数字是否重复
         //将1左移numbers[i]位，相当于将第numbers[i]置为1，然后与flag进行按位与运算
         //若flag上的第numbers[i]已为1，说明该位已被占用，数字重复了，运算的结果大于0,返回false
         if(((1 << numbers[i]) & flag) > 0) return false;
         flag = flag | 1 << numbers[i];//该位上不为1，通过“按位或”填充flag该位的1.
         //更新除0以外的最值
         if (numbers[i] > max) max = numbers[i];
         if (numbers[i] < min) min = numbers[i];
     }
     //最大与最小之差不超过5
     if (max - min >= 5) return false;
     return true;
}
```

46. P300，约瑟夫环，从0到n-1，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，求出这个圆圈里剩下的最后一个数字。

- 思路：注意下标的处理，也可以用约瑟夫环公式
- 代码实现：

```
function LastRemaining_Solution(n, m) {
    if (n === 0 || m === 0) return -1;
    const child = [];
    let del = 0;
    //将n个数加入数组
    for (let i = 0; i < n; i++) {
        child[i] = i;
    }
    while (child.length > 1) {
         const k = m - 1;//下标为零开始，第m个数下标为m-1
         //第一个数走k步，从删除的下一个数开始继续找第m个数，
```

```
        //因为删除了一个元素，所以下标表现为走了m-1步也就是k步，即在上一个下标上加k
        del = (del + k) % child.length;//超过数组长度取余
        child.splice(del, 1);//删除该下标的数字
    }
    return child[0];
}
```

47. P307，求1+2+3+...+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）。

- 思路：不能用乘除也就不能用公式，并且不能用循环，只能用递归。递归要终止条件，不能用if-else终止，那么只能用逻辑运算符。逻辑运算符当中的短路运算符有&&和||，这里只能用&&。A&&B的短路：对于非Boolean类型的数值，会转换为Boolean类型来判断，如果条件A不成立，则返回条件A的数值本身；如果条件A成立，不管条件B成不成立都返回条件B数值本身。
- 代码实现：

```
function Sum_Solution(n) {
    return n && Sum_Solution(n - 1) + n;//n减到0时判断n为0，直接返回0，向上返回。
}
```

48. P310，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。

- 思路：利用位运算。第一步，不考虑进位对每一位相加，1加1，0加0都为0，1加0，0加1都为1，即异或的结果。第二步，只有1加1会产生进位，两个数先做与运算，再左移1位。第三步，两个步骤结果相加，相加的过程也要重复前两步。
- 代码实现：

```
function Add(num1, num2)
{
    let sum = 0;
    while(num2 != 0){//直到没有进位为止
        sum = num1 ^ num2;
        num2 = (num1 & num2) << 1;
        num1 = sum;
    }
    //不能返回sum的原因，如果num2开始就为0，不会进入循环，不会给sum重新赋值
    return num1;
}
```

49. P318，输入一个字符串，包括数字字母符号，可以为空，将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数， 数值为0或者字符串不是一个合法的数值则返回0。

- 思路：判断第一个字符是不是正负号，若是负数输出负值，非法数值返回0。
- 代码实现：

```
function StrToInt(str) {
    let res = 0,
        flag = 1;//标记是否是负数
    const n = str.length;
    if (!n) return 0;
    if (str[0] === '-') {
        flag = -1;
    }
    //有符号从第1位开始，没有符合从第0位开始
    for (let i = str[0] === '+' || str[0] === '-' ? 1 : 0; i < n; i++) {
        if (!(str[i] >= '0' && str[i] <= '9')) return 0;
        res = res * 10 + (str[i] - '0');
    }
```

```
    return res * flag;
}
```

50. 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。

- 思路：用辅助空间数组或者哈希表记录出现的次数。
- 第二种：由于长度为n的数组里的所有数字都在0到n-1，也就是值也在数组的下标范围里。利用现有数组设置标志，遍历数组，将当前位置i上的值（a[i]）作为下标index去找对应的值a[index]（a[a[i]]）；当访问过这个数字后，可以设置对应位上的数 + n，之后再遇到相同的数时，会发现对应位上的数已经大于等于n了，那么直接返回这个数即可。
- 代码实现：

```
//第一种
function duplicate(numbers, duplication)
{
    //这里要特别注意~找到任意重复的一个值并赋值到duplication[0]
    //函数返回True/False
    //用辅助空间数组或者哈希表记录出现的次数
    let map = {};
    for(let i = 0;i < numbers.length;i++){
        if(!map[numbers[i]]){
            map[numbers[i]] = 1;
        }else{
            duplication[0] = numbers[i];
            return true;
        }
    }
    return false;
}
//第二种
function duplicate(numbers, duplication)
{
    for (let i = 0; i < numbers.length; i++) {
        let index = numbers[i];
        if (index >= numbers.length) {//下标大于length，减去length
            index -= numbers.length;
        }
        //下标对应的值大于length，说明已经被访问过，是重复值
        if (numbers[index] >= numbers.length) {
            duplication[0] = index;
            return true;
        }
        //当访问过这个数字后，可以设置对应位上的数 + length
        numbers[index] = numbers[index] + numbers.length;
    }
    return false;
}
```

51. P312，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中的元素B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]。不能使用除法。

- 思路：利于上三角矩阵和下三角矩阵的乘积。
- 代码实现：

```
function multiply(array) {
```

```
    //B[i] = C[i] * D[i]
    //自上而下C[i] = C[i-1]* A[i-1]
    //自下而上D[i] = D[i+1]* A[i+1]
    const B = [],
        len = array.length;
    B[0] = 1;
    // 计算前i - 1个元素的乘积
    for (let i = 1; i < len; i++) {
        B[i] = array[i - 1] * B[i - 1];//相当于C[i]
    }
    let tmp = 1;
    // 计算后N - i个元素的乘积并连接
    for (let i = len - 2; i >= 0; i--) {
        tmp *= array[i + 1];
        B[i] *= tmp;
    }
    return B;
}
```

52. P124，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

- 思路：
  - 当模式中的第二个字符不是'*'时
    - 字符串第一个字符和模式中的第一个字符相匹配或模式中的第一个字符是'.'，字符串和模式后后移一个字符
    - 字符串第一个字符和模式中的第一个字符不匹配，直接返回false
  - 当模式中的第二个字符是'*'时
    - 字符串第一个字符和模式中的第一个字符不匹配，模式后移两个字符，相当于忽略'*'和它之前的字符(模式中'*'之前的字符在字符串中出现了0次)
    - 字符串第一个字符和模式中的第一个字符相匹配或模式中的第一个字符是'.'，
      - 模式后移两位，类似于上一种情况，被忽略
      - 字符串后移一位，模式后移两位，相当于匹配了一个字符(模式中'*'之前的字符在字符串中出现了1次)
      - 字符串后移一位，模式不变，匹配下一个字符，相当于判断多位(模式中'*'之前的字符在字符串中可以出现多次)
- 代码实现：

```
//s, pattern都是字符串
function match(s, pattern)
{
    if(s === null || pattern === null) return false;
    return matchIndex(s,0,pattern,0);
}

function matchIndex(s,sIndex,pattern,pIndex){
    //s,pattern同时到达末尾则为true
    if(sIndex === s.length && pIndex === pattern.length) return true;
    //pattern先到末尾，而s没到，则为false
    if(sIndex !== s.length && pIndex === pattern.length) return false;

    //pattern第二个字符是*
    if(pIndex + 1 < pattern.length && pattern[pIndex + 1] === '*'){
        //s第一个字符和pattern的第一个字符相匹配或模式中的第一个字符是'.'
        if((sIndex !== s.length && s[sIndex] === pattern[pIndex]) ||
```

```
            (sIndex !== s.length && pattern[pIndex] === '.')){
             return (
                 matchIndex(s,sIndex,pattern,pIndex+2) ||
                 matchIndex(s,sIndex + 1,pattern,pIndex+2) ||
                 matchIndex(s,sIndex + 1,pattern,pIndex)
             );
         }else{//s第一个字符和pattern的第一个字符不匹配
             return matchIndex(s,sIndex,pattern,pIndex+2);
         }
     }
     //pattern第二个字符不是*
     //s第一个字符和pattern的第一个字符相匹配或模式中的第一个字符是'.'
     if((sIndex !== s.length && s[sIndex] === pattern[pIndex]) ||
        (sIndex !== s.length && pattern[pIndex] === '.')){
         return matchIndex(s,sIndex+1,pattern,pIndex+1);
     }else{
         return false;
     }
 }
```

53. P127，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）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

- 思路：逐个字符判断，A[.[B]][E|eC]，或者用正则表达式
  - 开头可以有符号也可以没有。
  - A不是必须的。小数可以没有整数部分（.123相当于0.123）
  - B不是必须的。小数点后面也可以没有数字（123.相当于123.0）
  - e或E和小数点最多出现一次，e或E前面必须是数字，后面必须是整数（可以有符号）。
- 代码实现：

```
//s字符串
function isNumeric(s)
{   //match()返回值：如果使用g标志，则将返回与完整正则表达式匹配的所有结果（Array），未匹配 null。
    //return s.match(/[+-]?\d*(\.\d*)?([eE][+-]?\d+)?/g)[0] === s;
    //search()返回值：如果匹配成功，则返回正则表达式在字符串中首次匹配项的索引;否则，返回 -1。
    if(s === null || !s.length) return false;
    return s.search(/^[+-]?\d+(\.\d*)?$/) === 0 //没有指数
    || s.search(/^[+-]?\d*\.\d+$/) === 0
    || s.search(/^[+-]?\d+(\.\d*)?[Ee]{1}[+-]?\d+$/) === 0;
}
```

54. P247，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"。如果当前字符流没有存在出现一次的字符，返回#字符。

- 思路：一般遇到次数问题，就可以想到用哈希表来统计次数
- 代码实现：

```
let map = {};
function Init() {
  map = {};
}
function Insert(ch) {
  if (map[ch]) {
    map[ch] += 1;
  } else {
```

```
    map[ch] = 1;
  }
}
//for in遍历的是数组的索引（即键名），而for of遍历的是数组元素值。
//for of遍历的只是数组内的元素，而不包括数组的原型属性method和索引name
//for..of适用遍历数/数组对象/字符串/map/set等拥有迭代器对象的集合.但是不能遍历对象，
//因为没有迭代器对象.与forEach()不同的是，它可以正确响应break、continue和return语句
//for-of循环不支持普通对象，但如果你想迭代一个对象的属性，
//你可以用for-in循环（这也是它的本职工作）或内建的Object.keys()方法：
function FirstAppearingOnce() {
  for (const i in map) {//遍历数组的索引（即键名）,就是传入的字符
    if (map[i] === 1) {
      return i;
    }
  }
  return '#';
}
```

55. P139，给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。

- 思路：一快（两步）一慢（一步）指针，先找到碰撞点，然后碰撞点到入口节点的距离就是头结点到入口节点的距离。快指针指向头节点，继续相同速度一起走，相遇即为入口节点。详细分析：相遇p1==p2时，p2经过2x个节点，p1经过x个，设环中有n个节点，p2比p1多走一圈环，2x=n+x，n=x，所以p1实际走了一个环的步数，这时p2指向头节点（p1停在相遇点，即相当于p1p2从头节点开始，p1先走了一个环的步数（n）），p1p2一起每次走一步，直到p1==p2，此时p1指向环的入口
- 代码实现：

```
//相遇p1==p2时，p1实际走了一个环的步数，这时p2指向头节点，
//p1p2一起每次走一步，直到p1==p2，此时p1指向环的入口
function EntryNodeOfLoop(pHead)
{
    let p1 = pHead,p2 = pHead;
    while(p2 !== null && p2.next !== null){
        p1 = p1.next;
        p2 = p2.next.next;
        if(p1 === p2){//相遇说明包含环
            p2 = pHead;
            while(p2 !== p1){
                p1 = p1.next;
                p2 = p2.next;
            }
            return p1;
        }
    }
    return null;
}
```

56. P122，在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

- 思路：因为链表是单向的，如果是第一个、第二个节点就重复的话，删除就比较麻烦。因此我们可以额外添加头节点来解决。因为重复的节点不一定是重复两个，可能重复很多个，需要循环处理。
- 代码实现：

```
/*
function ListNode(x){
```

```
    this.val = x;
    this.next = null;
}
*/
function deleteDuplication(pHead) {
    if (pHead === null || pHead.next === null) {
        return pHead;
    }
    //新建一个头节点，以防头节点被删除，方便处理第一个、第二个节点就是相同的情况。
    let Head = new ListNode(0);
    Head.next = pHead;
    let pre = Head;
    let cur = Head.next;
    while (cur !== null && cur.next !== null) {
        if (cur.val === cur.next.val) {
        // 找到最后的一个相同节点,因为相同节点可能重复多个
            let val = cur.val;
            while (cur !== null && cur.val === val) {
                cur = cur.next;
            }
            //上一个非重复值指向下一个非重复值，即删除重复值
            pre.next = cur;
        } else {
            pre = cur;
            cur = cur.next;
        }
    }
    return Head.next;
}
```

57. P65，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。

- 思路：
  - 如果一个节点有右子树，那么它的下一个节点就是它的右子树中的最左子节点。也就是说，从右子节点出发一直沿着指向左子节点的指针，找到叶子结点就是下一个节点。
  - 如果没有右子树，又可以分为两种情况
    - 如果节点是它父节点的左子节点，那么它的下一个节点就是它的父节点。
    - 如果一个节点既没有右子树，并且它还是父节点的右子节点，那么需要沿着指向父节点的指针一直向上遍历，直到找到一个是它父节点的左子节点的节点。这个节点的父节点就是下一个节点。
- 代码实现：

```
function GetNext(pNode)
{
    if(pNode === null) return null;
    //如果一个节点有右子树，那么它的下一个节点就是它的右子树中的最左子节点。
    //也就是说，从右子节点出发一直沿着指向左子节点的指针，找到叶子结点就是下一个节点。
    if(pNode.right !== null){
        pNode = pNode.right;
        while(pNode.left !== null){
            pNode = pNode.left;
        }
        return pNode;
    }
```

```
    //如果没有右子树，又可以分为两种情况
    //如果节点是它父节点的左子节点，那么它的下一个节点就是它的父节点。
    //如果一个节点既没有右子树，并且它还是父节点的右子节点，那么需要沿着指向父节点的指针一直向上遍历，
    //直到找到一个是它父节点的左子节点的节点。返回该节点的父节点
    while(pNode.next !== null){
        if(pNode === pNode.next.right){
            pNode = pNode.next;
        }else{
            return pNode.next;
        }
    }
    return null;
}
```

58. P159，请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

- 思路：递归判断，左子树的左孩子等于右子树的右孩子且左子树的右孩子等于右子树的左孩子，并且左右子树节点值相等，则对称。注意空的情况。
- 代码实现：

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

function isCommon(left,right){
    if(left === null) return right === null;
    if(right === null) return false;
     if(left.val !== right.val) return false;
     return isCommon(left.left,right.right) && isCommon(left.right,right.left);
}
```

59. P176，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。

- 思路：在奇数行从左至右，在偶数行从右至左，因为是先进后出，分析可得我们需要的数据结构是栈。为了避免顺序错误，要用两个栈分别存储奇偶层，然后一行打印完后交替。也可以在广度遍历的基础上加个临时数组，然后对偶数行的数据reverse。
- 代码实现：

```
function Print(pRoot)
{
    let res = [];
    if(pRoot === null) return res;

    let stack1 = [],
        stack2 = [];
    stack1.push(pRoot);
    let i = 1;
     while(stack1.length !== 0 || stack2.length !== 0){
        let list = [];
         //奇数层
        if( (i&1) === 1){
            while(stack1.length !== 0){
```

```
                let node = stack1[stack1.length - 1];
                stack1.pop();
                list.push(node.val);
                if(node.left !== null) stack2.push(node.left);
                if(node.right !== null) stack2.push(node.right);
            }
        }else{//偶数层
            while(stack2.length !== 0){
                let node = stack2[stack2.length - 1];
                stack2.pop();
                list.push(node.val);
                if(node.right !== null) stack1.push(node.right);
                if(node.left !== null) stack1.push(node.left);
            }
        }
        res.push(list);
        i++;
    }
    return res;
}
```

60. P174，从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

- 思路：从上到下打印二叉树用队列可以实现，但是如果多行打印需要在行与行之间进行分割。如何分割呢？肯定要知道个数才能分割。又如何知道这一行有多少个呢？我们可以通过遍历上一层，通过它们的子树就可以知道这一层有多少个节点了，所以需要一个变量记录下一层节点的数目。还需要有一个变量来记录这一层已经打印了多少个节点了。所以我们需要一个队列+两个变量。
- 代码实现：

```
function Print(pRoot) {
  const queue = [],
    res = [];
  if (pRoot === null) {
    return res;
  }
  queue.push(pRoot);
  let nextLevel = 0; // 下一层节点个数
  let toBePrinted = 1; // 这一层还有多少个节点要打印
   let list = []; // 存放每一层节点
   while (queue.length) {
     const pNode = queue.shift();
     list.push(pNode.val);
     if (pNode.left !== null) {
       queue.push(pNode.left);
       nextLevel++;
     }
     if (pNode.right !== null) {
       queue.push(pNode.right);
       nextLevel++;
     }
     toBePrinted--;
     if (toBePrinted === 0) {
       res.push(list);
       list = [];
```

```
        toBePrinted = nextLevel;
        nextLevel = 0;
      }
    }
    return res;
}
```

61. P194，请实现两个函数，分别用来序列化和反序列化二叉树。二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。二叉树的反序列化是指：根据某种遍历顺序得到的序列化结果，重构二叉树。

- 思路：序列化，前序遍历二叉树存入数组，当我们遍历到null时，我们需要用换位符（比如$）代表，方便反序列化。反序列化，根据数组重构二叉树。
- 代码实现：

```
let arr = [];
function Serialize(pRoot) {
    if (pRoot === null) {//节点为空用$表示
        arr.push('$');
    } else {
        arr.push(pRoot.val);//不为空加入数组
        Serialize(pRoot.left);//递归左右子树
        Serialize(pRoot.right);
    }
}
function Deserialize() {
    let node = null;
    if (arr.length < 1) {
         return null;
    }
    let number = arr.shift();
    if (typeof number === 'number') {//数组中值是数字
        node = new TreeNode(number);
        node.left = Deserialize();
        node.right = Deserialize();
    }
    return node;
}
```

62. P269，给定一棵二叉搜索树，请找出其中的第k小的结点。例如，（5，3，7，2，4，6，8）中，按结点数值大小顺序第三小结点的值为4。

- 思路：二叉搜索树的中序遍历序列就是从小到大排列，也就是我们可以直接中序遍历，同时计数，就可以得到我们想要的节点了。需要注意的是计数变量k应该在函数外面，不然递归后回来是无法获得已经改变了的k值的。
- 代码实现：

```
function KthNode(pRoot, k) {
    if(pRoot === null || k < 1) return null;
    // 为了能追踪k，应该把KthNodeCore函数定义在这里面，k应该在KthNodeCore函数外面
    function KthNodeCore(pRoot) {
        let target = null;
        if(pRoot.left !== null){
            target = KthNodeCore(pRoot.left);
        }
        if(target === null){
            if(k === 1)
```

```
                target = pRoot;
            k--;
        }
        if(target === null && pRoot.right !== null){
            target = KthNodeCore(pRoot.right);
        }
        return target;
    }
    return KthNodeCore(pRoot);
}
```

63. P214，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。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。

- 思路：第一种：插入的时候保持数组一直有序，插入的时间复杂度O(n)，得到中位数的时间复杂度O(1)；第二种：最大堆和最小堆插入的时间复杂度O(logn)，得到中位数的时间复杂度O(1)。最小堆的值都大于最大堆，奇数个值的中位数为最大堆堆顶，偶数个时中位数为两个堆顶的平均数。
- 其他数据容器
  - 数组没有排序，用partition函数可以找出中位数，插入的时间复杂度O(1)，得到中位数的时间复杂度O(n)。
  - 排序的链表，插入的时间复杂度O(n)，得到中位数的时间复杂度O(1)。
  - 二叉搜索树，插入的时间复杂度平均O(lgn)，当二叉搜索树极度不平衡时，看起来像一个排序的链表，插入新数据的时间为O(n)，得到中位数的时间复杂度O(lgn)，最差为O(n)。
  - AVL树，平衡的二叉搜索树，插入的时间复杂度O(lgn)，得到中位数的时间复杂度O(1)。和堆一样，但是实现此数据结构相对复杂。
- 代码实现：

```
//第一种
let array = [];
function Insert(num) {
    array.push(num);//插在末尾
    for (let i = array.length - 2; array[i] > num; i--) {//从倒数第二个元素开始比较
        [array[i], array[i + 1]] = [array[i + 1], array[i]];//小于前面的值就交换
    }
}
function GetMedian() {
    if (array.length & 1 === 1) {
        return array[(array.length - 1) / 2];
    }
    return (array[array.length / 2] + array[array.length / 2 - 1]) / 2;
}
```

64. P288，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}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

- 思路：题目描述的是滑动窗口在数组上移动。如果我们以滑动窗口为对象，那么就是数组在滑动窗口上移动。显然，可以看出滑动窗口就是一个队列，数组中的一个一个的数先进去，先出来。此外这题还有一个可以优化的一点就是不一定需要把所有数字存进去队列里，只需要把以后有可能成为最大值的数字存进去。还有一点要注意的是队列里保存的是下标，而不是实际的值，因为窗口移动主要是下标的变化。当然还有其他解法，比如利用两个栈去实现这个队列，从而使得查询时间复杂度降低到O(n)
- 代码实现：

```
function maxInWindows(num, size)
{
```

```
    let res = [];
    if(num === null || size < 1 || num.length < size){
        return res;
    }
    let begin;
    let queue = [];
    for(let i = 0; i < num.length; i++){
        begin = i - size + 1;//代表滑动窗口的左边界
        if(queue.length == 0){
            queue.push(i);//队列里保存的是下标，而不是实际的值，因为窗口移动主要是下标的变化
        }else if(begin > queue[0]){
            queue.shift();
        }
        while((queue.length != 0) && (num[queue[queue.length - 1]] <= num[i])){
            queue.pop();
        }
        queue.push(i);
        if(begin >= 0){
            res.push(num[queue[0]])
        }
    }
    return res;
}
```

65. P89，请设计一个函数，用来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左，向右，向上，向下移动一个格子。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 a b c e s f c s a d e e 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

- 思路：一个一个字母的匹配，当发现不行时，就要回溯上个步骤，选择另一步。回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。
- 回溯法的基本思想：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。字典树也有点和回溯法的解空间树类似。
- 代码实现：

```
function hasPath(matrix, rows, cols, path) {
    const pathLength = 0;
    const visited = new Array(rows * cols);
    for (let row = 0; row < rows; row++) {
        for (let col = 0; col < cols; col++) {
            // 遍历，遍历的点为起点。
            if (hasPathCore(matrix, rows, cols, row, col, path, pathLength, visited)) {
                return true;
            }
        }
    }
    return false;
}
function hasPathCore(matrix, rows, cols, row, col, path, pathLength, visited) {
    let hasPath = false;
```

```
    if (pathLength === path.length) return true;
    if(
        row >= 0 && row < rows &&
        col >= 0 && col < cols &&
        matrix[row * cols + col] === path[pathLength] &&
        !visited[row * cols + col]
      ) {
        ++pathLength;
        visited[row * cols + col] = true;
        // 因为||为短路运算符，只要第一个满足就会返回，而不会去计算后面的，所以有些路径可以不用去走
        hasPath =
          hasPathCore(matrix, rows, cols, row - 1, col, path, pathLength, visited) ||
          hasPathCore(matrix, rows, cols, row, col - 1, path, pathLength, visited) ||
          hasPathCore(matrix, rows, cols, row + 1, col, path, pathLength, visited) ||
          hasPathCore(matrix, rows, cols, row, col + 1, path, pathLength, visited);
        if (!hasPath) {
            --pathLength;
            visited[row * cols + col] = false;
        }
    }
    return hasPath;
}
```

66. P92，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?

- 思路：回溯法和暴力法有点类似，不过他会用数组或变量去记录已经遍历过的解，避免重复遍历，从而减少了计算量。
- 代码实现：

```
function movingCount(threshold, rows, cols) {
    let visited = [];//标记是否能进入
    for (let i = 0; i < rows; i++) {
        visited.push([]);
        for (let j = 0; j < cols; j++) {
            visited[i][j] = false;
        }
    }
    return move(0, 0, rows, cols, visited, threshold);
}

function move(i, j, rows, cols, visited, threshold) {
    let count = 0;
    if (i < 0 || i === rows || j < 0 || j === cols || visited[i][j]) {
        return 0;
    }
    let sum = 0;
    let tmp = `${i}${j}`;//坐标转成字符串
    for (let k = 0; k < tmp.length; k++) {
        sum += tmp.charAt(k) / 1; // 转成数字
    }
    if (sum > threshold) {
        return 0;
    }
```

```
    visited[i][j] = true;
    count++;
    return (
            count+
            move(i + 1, j, rows, cols, visited, threshold) +
            move(i, j + 1, rows, cols, visited, threshold) +
            move(i - 1, j, rows, cols, visited, threshold) +
            move(i, j - 1, rows, cols, visited, threshold)
          );
}
```

67. P96，给你一根长度为n的绳子，请把绳子剪成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]。请问k[0]xk[1]x...xk[m]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。

- 思路：动态规划和贪心算法。求一个问题的最优解，当一个大问题能分解成若干个小问题，这些小问题之间还有互相重叠的更小的子问题，整体问题的最优解依赖各个子问题的最优解，从上往下分析问题，从下往上求解问题。贪心选择需要用数学方式证明正确性。
- 动态规划：定义函数f(n)为乘积最大值，剪第一刀时有n-1种选择，剪完之后长度可能为1,2,...,n-1，因此f(n)=max(f(i)*f(n-i))，其中0<i<n，这是一个从上到下的递归公式，会有很多重复的子问题，所以从下往上求解。时间复杂度O(n*n)，空间复杂度O(n)。
- 贪心算法：当n>=5时，尽可能多剪长度为3的绳子；当剩下的绳子长度为4时，剪成两段长度为2的绳子。
- 代码实现：

```
//动态规划
function cutRope(number)
{
    //f(n)=max(f(i)*f(n-i))
    if(number < 2) return 0;
    if(number === 2) return 1;
    if(number === 3) return 2;
    let products = [0,1,2,3];

     for(let i = 4;i <= number;i++){
        let max = 0;
        for(let j = 1;j <= i/2;j++){
            let product = products[j] * products[i - j];//f(i)*f(n-i), 1<i<=n/2
            if(max < product) max = product;
        }
        products[i] = max;//记录每一个f(i)的最优解
    }
    return products[number];
}
//贪心算法
function cutRope(number){
    if(number < 2) return 0;
    if(number === 2) return 1;
    if(number === 3) return 2;

    //尽可能多剪出长度为3的绳子
    let timesOf3 = Math.floor(number/3);
    //当剩下的绳子长度为4时，就不再剪为3的绳子，而是剪两段长度为2的绳子，因为2*2>3*1
    if(number - timesOf3*3 === 1){
        timesOf3 -= 1;//剪出长度为3的次数减1
```

```
    }
    let timesOf2 = (number - timesOf3*3)/2;
    return Math.pow(3,timesOf3) * Math.pow(2,timesOf2);
}
//下面证明这种贪心选择的正确性
//首先，当n>=5时，证明2(n-2)>n,且3(n-3)>n。也就是说，当绳子剩下的长度>=5的时候，
//就把它剪成3或2的绳子。当n>=5时，3(n-3)>=2(n-2),因此应该尽可能多的减出3的绳子
//当绳子为4的时候，2*2>3*1，其实2*2=4，长度为4的绳子没必要剪，但是题目要求至少剪一刀
```