# leetcode刷题分类基础（前端JavaScript版答案）

# 一、 数组类

## 1、 04. 二维数组中的查找

```
题目：在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]    给定 target = 5，返回 true。
```

注意： （1）数组的行列表达:int rows=matrix. length， columns = matrix[0].length;（2）组元素表示：matrix[i] [j] （3）异常输入判断。

```
var findNumberIn2DArray = function(matrix, target) {
    const row = matrix.length;
    if (!row)  return false;
    const col = matrix[0].length;
    let i = 0,j = col - 1;
    while (i < row && j >= 0) {
        if (matrix[i][j] === target)  return true;
        if (matrix[i][j] > target) --j;
        else  ++i;
    }
    return false;
};
```

## 2、05. 替换空格

```
输入：s = "We are happy."
输出："We%20are%20happy."//把字符串 s 中的每个空格替换成"%20"。
```

注意： （1）正则表达式，用replace（）

```
var replaceSpace = function(s) {
    return s.replace(/ /g,"%20");
};
```

## 3、11. 旋转数组的最小数字

问题描述：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。输入一个递增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。
//输入：[2,2,2,0,1]
//输出：0

注意：二分法，情况考虑周全。

```
const minArray = (nums) => {
  let left = 0,right = nums.length - 1;
  while (left < right) {
    const mid = left + right >>> 1;
     //>>>表示无符号右移 ;>>表示有符号右移;  +、-运算符的优先级高于<<、>>移运算符
    //const mid=left+(right-left)/2;
    if (nums[mid] > nums[right]) {
      left = mid + 1;
    } else if (nums[mid] == nums[right]) {
      right--;
    } else {
      right = mid;
    }
  }
  return nums[left];
};
```

## 4、17. 打印从1到最大的n位数

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

注意：Math.pow（）

```
var printNumbers = function(n) {
    let num=Math.pow(10,n)-1;
    let arr=new Array;
    for(let i=1;i<=num;i++){
        arr.push(i);
    }
    return arr;
};
```

## 5、21. 调整数组顺序使奇数位于偶数前面

```
var exchange = function(nums) {
    let len=nums.length;
    let i=0;j=len-1;n=0;
    let arr=new Array;
    while(n<len){
        if(nums[n]%2==1){
            arr[i]=nums[n];
            i++;
```

```
        }else{
            arr[j]=nums[n];
            j--;
        }
       n++;
    }
    return arr;
};
```

## 6、29. 顺时针打印矩阵

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

注意：时间复杂度o(mn)

```
var spiralOrder = function(matrix) {
    if (!matrix.length || !matrix[0].length) {
        return [];
    }
    const rows = matrix.length, columns = matrix[0].length;
    const order = [];
    let left = 0, right = columns - 1, top = 0, bottom = rows - 1;
    while (left <= right && top <= bottom) {
        //上右
        for (let column = left; column <= right; column++) {
            order.push(matrix[top][column]);
        }
        for (let row = top + 1; row <= bottom; row++) {
            order.push(matrix[row][right]);
        }
        //下左
        if (left < right && top < bottom) {
            for (let column = right - 1; column > left; column--) {
                order.push(matrix[bottom][column]);
            }
            for (let row = bottom; row > top; row--) {
                order.push(matrix[row][left]);
            }
        }
        [left, right, top, bottom] = [left + 1, right - 1, top + 1, bottom - 1];
    }
    return order;
};
```

## 7、39. 数组中出现次数超过一半的数字

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

常见方法：（1）哈希表统计法：遍历数组nums用 HashMap统计各数字的数量，可找出众数。时间和空间复杂度均为 O(N)O(N)

（2）数组排序法：将数组 nums 排序，数组中点的元素 一定为众数。

（3）摩尔投票法：核心理念为 票数正负抵消，此方法时间和空间复杂度分别为O(N)O(N)和O(1)O(1)为本题的最佳解法 */

**最佳为摩尔计数法，考虑那种不存在的情况**

```
var majorityElement = function(nums) {
    let res=0;count=0;
    for(let i=0;i<nums.length;i++){
        if(!count) {
            res=nums[i];
            count++;
        }else count+=nums[i]===res?1:-1;
    }
    return res;
};
```

## 8、57. 和为s的两个数字

```
输入：nums = [2,7,11,15], target = 9   输出：[2,7] 或者 [7,2]
若不存在，返回null
```

```
var twoSum = function(nums, target) {//用对撞双指针
    let left=0;right=nums.length-1;
    let arr=new Array;
    while(left<right){
        if(nums[left]+nums[right]==target) {
            arr.push(nums[left]);arr.push(nums[right]);
            return arr;
        }else if(nums[left]+nums[right]>target) right--;
        else left++;
    }
    return null;
};
```

## 9、57-II. 和为s的连续正数序列

```
问题描述：输入一个正整数 target，输出所有和为 target的连续正整数序列（至少含有两个数）。序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。
示例：输入：target = 15     输出：[[1,2,3,4,5],[4,5,6],[7,8]]
```

注意：双指针；shift() 方法用于把数组的第一个元素从其中删除，并返回第一个元素的值；分奇偶找到中间值，然后通过滑动窗口不断判断，如大于target则将第一个元素删除；

```
var findContinuousSequence = function (target) {
  let index = target % 2 === 0 ? target / 2 : (target / 2 | 0) + 1; //判断奇偶
  let res = [],temp = [],sum = 0;
  for (let i = 1; i <= index; i++) {
    temp.push(i);
    sum = sum + i;
    while (sum > target) {
      sum -= temp[0];
      temp.shift();//将数组第一个值删除
```

```
    }
    if (sum === target) {
      temp.length >= 2 && res.push([...temp]);//...是扩展运算符，...数组名
    }
  }
  return res;
};
```

## 10、 58-I. 翻转单词顺序

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

注意：trim()： 去除字符串左右两端的空格；filter(函数)：创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素；split();

```
stringObject.split(separator,howmany)//把一个字符串分割成字符串数组。
//howmany:该参数可指定返回的数组的最大长度
var str="How are you doing today?"
document.write(str.split(" ") + "<br />")
document.write(str.split("") + "<br />")
document.write(str.split(" ",3))
```

reverse()；*array*.join('separator '): 在两个元素之间插入 separator 字符串；

```
var reverseWords = function (s) {
        var str = s.trim().split(' ').filter(item => item!='').reverse().join('
');
        return str;
};
```

## 11、58-II. 左旋转字符串

```
问题描述：字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。比如输入字符
串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。
输入: s = "abcdefg", n = 2     输出: "cdefgab"
```

注意：会使用这几个函数

string.slice(start, end)　end省略时代表到尾部,不包括end。end可-1，-2

string.substring(start, end) 提取一个字符串,不包括end.end不可为负。

s.substring(n, 0)表示从0到n，不包括n

string.substr(start, len)　提取一个长度为len的字符串　从0开始。

```
var reverseLeftWords = function(s, n) {
    return s.substring(n) + s.substring(0,n);
    //return s.substring(n) + s.substring(n,0);//一样的
};
/* var reverseLeftWords = function(s, n) {
     return s.slice(n) + s.slice(0, n);
};
var reverseLeftWords = function(s, n) {
     return s.substr(n,s.length-n) + s.substr(0, n);
};  */
```

### 12、66. 构建乘积数组

问题描述：给定一个数组 A[0,1,…,n-1]，请构建一个数组 B[0,1,…,n-1]，其中 B[i] 的值是数组 A 中除了下标 i 以外的元素的积，即 B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]。不能使用除法。
//输入: [1,2,3,4,5]　　输出: [120,60,40,30,24]

注意：从左到右遍历，再从右到左遍历。（重点）

```
var constructArr = function (a) {
    // 先乘这个数的左边，在成这个数的右边，最后相乘，避免漏掉当前数
    let n = a.length;
    let b = [];
    // p代表前i个a[i]的乘积，然后依次放入b中
    for (let i = 0, p = 1; i < n; i++) {
        b[i] = p;
        p *= a[i];
    }
    // p代表后i个a[i]的乘积，然后依次放入b中
    for (let i = n - 1, p = 1; i>=0; i--) {
        b[i] *= p;
        p *= a[i];
    }
    return b;
};
```

# 二、栈和队列

### 1、06. 从尾到头打印链表

输入：head = [1,3,2]　　输出：[2,3,1]

注意：head元素个数在遍历的时候++，或用push(),unshift()

push() 方法可向数组的末尾添加一个或多个元素，并返回新的长度。

unshift() 方法可向数组的开头添加一个或更多元素，并返回新的长度。

```
var reversePrint = function(head) {
    const stack = [],res = [];

    while(head){
        stack.push(head.val);
        head = head.next;
    }
    let t=stack.push();//unshift()
    for(let i = 0;i < t;i ++){
        res.push(stack.pop());
    }
    return res;
};
```

## 2、09.用两个栈实现队列

问题描述：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1 )

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

注意：用两个数组完成，每次插入放到数组1，当删除时，检查数组2有没有值，有值就删除，没有值将数组1的数据放入数组2中，删除数组2中第一个元素，pop()出。

```
var CQueue = function() {
this.stack1=[];
this.stack2=[];
};

CQueue.prototype.appendTail = function(value) {//  {number} value  return {void}
    this.stack1.push(value);
};

CQueue.prototype.deleteHead = function() {// @return {number}
    if(this.stack2.length) return this.stack2.pop();
    else{
        while(this.stack1.length) this.stack2.push(this.stack1.pop());
        if(this.stack2.length) return this.stack2.pop();
        else return -1;
    }
};
```

## 3、30.包含min函数的栈

```
定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.min();   --> 返回 -2.
```

```
class MinStack {
    constructor () {
        this.stack = []
    }
    push(val) {
        // 若栈不为空 则迭代对比更新最值
        this.stack.push({
            val,
            min: this.stack.length ? Math.min(this.head.min, val) : val
        })
    }
    pop() {
        this.stack.pop()
    }
    min() {
       return this.head.min
    }
    top() {
        return this.head.val
    }
    get head() {
        // 当前栈的最后一项 即栈顶头 head
        return this.stack[this.stack.length - 1]
    }
}
```

## 4、59-I. 滑动窗口

```
给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
```

...arr.slice（将数组变成数字, 即去掉中括号）

arr.slice(i, i+k)数组转数字, slice返回一个新的数组，包含从 start 到 end （不包括该元素）的 arr中的元素。

```
var maxSlidingwindow = function(nums, k) {
    if(k<=1) return nums;
    const res=[];
    for(let i=0;i<nums.length-k+1;i++){
        res.push(Math.max(...nums.slice(i,i+k)));
    }
    return res;
};
```

## 5、59-II. 队列的最大值

```
请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和
pop_front 的均摊时间复杂度都是O(1)。
若队列为空，pop_front 和 max_value 需要返回 -1
输入：["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出：[null,null,null,2,1,2]
```

注意：双栈队列

```
var MaxQueue = function() {
    this.queue1 = [];
    this.queue2 = [];//存放最大的值。
};

MaxQueue.prototype.max_value = function() {
    if (this.queue2.length) {
        return this.queue2[0];
    }
    return -1;
};

MaxQueue.prototype.push_back = function(value) {
    this.queue1.push(value);
    while (this.queue2.length && this.queue2[this.queue2.length - 1] < value)
{//循环操作，会选出最大的直到为0；
        this.queue2.pop();//如果队列2的尾部小于队列1的值，则将这个小的pop出
    }
    this.queue2.push(value);//放入插入队列1的大值。
};

MaxQueue.prototype.pop_front = function() {
    if (!this.queue1.length) {//队列1为0
        return -1;
    }
    const value = this.queue1.shift();//取出队首元素
    if (value === this.queue2[0]) {
        this.queue2.shift();
    }
    return value;
};
```

# 三、哈希表

## 1、03. 数组中重复的数字

找出数组中重复的数字。 输入： [2, 3, 1, 0, 2, 5, 3] 输出：2 或 3

注意：哈希表是根据键值对，学会map的has、set将键值对插入哈希表中；考虑最后没有重复值的情况；

```
var findRepeatNumber = function(nums) {
    let mymap = new Map();// [key, value] of myMap
    for(let i of nums){//i为数组里面的值
        if(mymap.has(i)) return i;
        mymap.set(i, i);//
    }
    return null;
};
```

## 2、50. 第一个只出现一次的字符

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

注意：indexOf()：indexOf() 方法可返回某个指定的字符串值在字符串中首次出现的位置。lastIndexOf();

```
var firstUniqChar = function(s) {
    for(let x of s){
        if(s.indexOf(x) === s.lastIndexOf(x)) //     返回最后一个出现的索引值,两个三个=都可以
            return x;
    }
    return ' ';
};
```

# 四、链表

## 1、18. 删除链表的节点

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]

注意：两种方法；第一种是递归思想；

```
var deleteNode = function(head, val) {
    if(head.val == val){
        return head.next;
    }
    head.next = deleteNode(head.next,val);
    return head;
};
```

```
//法二
var deleteNode = function (head, val) {
    let pre = head;
    let node = pre.next;
    if (pre.val === val) {
        return pre.next;
    }
    while (node) {
        if (node.val === val) {
            pre.next = node.next;
        }
        pre = node;
        node = node.next;
    }
    return head;
};
```

## 2、22. 链表中倒数第k个节点

```
给定一个链表: 1->2->3->4->5, 和 k = 2.   返回链表 4->5.
```

注意：法一用栈，法二让指针p先走k步，再一起走

```
//法一 ：栈
var getKthFromEnd = function(head, k) {
    var stack = [];
    var ans = [];
    while(head){ //所有节点入栈
        stack.push(head);
        head = head.next;
    }
    //出栈第k个节点
    while(k > 0){
        ans = stack.pop();
        k--;
    }
    return ans;
};
//法二：让指针p先走k步，再一起走
var getKthFromEnd = function(head, k) {
    let p = head, q = head;
    let i = 0;
    while (p) {
        if (i >= k) {
            q = q.next;
        }
        p = p.next;//先走k步
        i++;
    }
    return i < k ? null : q;
};
```

## 3、24.反转链表

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

注意：迭代

```
var reverseList = function(head) {
    let prev = null;
    let curr = head;
    //let [p, curr] = [null, head];
    while (curr) {
        const next=curr.next;//记录curr的下一个
        curr.next = prev;
        prev = curr;//保存反序的结果5，
        curr = next;//指向指针下一个
    }
    return prev;
};
```

## 4、 25.合并两个排序的链表

```
输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

注意：递归思想

```
var mergeTwoLists = function(l1, l2) {
    if (!l1 || !l2) //二者有一个为空，则停止递归；
        return l1 || l2;
    if (l1.val < l2.val) {
        l1.next = mergeTwoLists(l1.next, l2)
        return l1;
    } else {
        l2.next = mergeTwoLists(l1, l2.next)
        return l2;
    }
}
```

## 5、focus 52. 两个链表的第一个公共节点

```
输入两个链表，找出它们的第一个公共节点。
```

注意：两种方法，法一为快慢指针；法二为哈希表

```
//法一：双指针法(快慢指针追赶问题)
var getIntersectionNode = function(headA, headB) {
    let A = headA, B = headB
    while (A !== B ) {
        A  = A ? A.next : headB//A不为空，则A等于A.next
        B  = B  ? B .next : headA
    }
```

```
    return B;//存储结果，即节点
};
//法二：哈希表
var getIntersectionNode = function(headA, headB) {
    const map = new Map();
    let node = headA;
    while (node) {
        map.set(node, true);
        node = node.next;
    }
    node = headB;
    while (node) {
        if (map.has(node)) return node;
        node = node.next;
    }
    return null;
};
```

# 五 DFS深度优先搜索

```
function TreeNode(val) {
this.val = val;
this.left = this.right = null;
}
```

**二叉搜索树，中序遍历的数组结果刚好是排好序的。**

**前序遍历（中左右）、中序遍历（左中右）、后序遍历（左右中）**

## 1、（递归）27. 二叉树的镜像

```
请完成一个函数，输入一个二叉树，该函数输出它的镜像。
```

注意：递归

```
var mirrorTree = function(root) {
    if(!root) return null;//如果root为空，则返回null。
    [root.left, root.right] = [mirrorTree(root.right), mirrorTree(root.left)];
    return root;//return {TreeNode}
};
```

## 2、28. 对称的二叉树（箭头函数）

```
请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。
输入：root = [1,2,2,3,4,4,3]
输出：true
输入：root = [1,2,2,null,3,null,3]
输出：false
```

注意：递归；三种情况：（1）左右为空（2）其中有一个为空（3）两个不为空判断值

**箭头函数： const check = (left, right) => { }**

```
var isSymmetric = function(root) {
   if (!root) return true;//如果根为空
    const check = (left, right) => {//箭头函数

        if (!left && !right) return true  // 左右子树同时为空      则为镜像
        if (!left || !right) return false // 左右子树有一个为空   则不为镜像
        if (left.val !== right.val) return false // 左右子树同时存在 但值不同 也不为镜像
        return check(left.left, right.right) && check(left.right, right.left)
    }
    return check(root.left, root.right)//真正的调用
};
```

## 3、 (递归) 54. 二叉搜索树的第k大节点

```
给定一棵二叉搜索树，请找出其中第k大的节点。
输入: root = [3,1,4,null,2], k = 1   输出: 4
   3
  / \
 1   4
  \
   2
```

注意：两种方法。（第二种好）

1、二叉搜索树，中序遍历的数组结果刚好是排好序的，则反中序遍历,遍历结束后直接获取数组第 K-1 位的数值;

2、第二种解法上利用反中序遍历，最容易想到的优化就是直接遍历到第 k 大的值就停止遍历，直接返回需要的值.

```
//2、反中序遍历结果是直接获取数组第 K-1 位的数值
var kthLargest = function(root, k) {
    // 反中序遍历，记录数值到数组，获取第k -1 个
    let resArr = [];//数组存储反中序遍历结果，
    const dfs = function(node) {//const dfs=(root)=>{//箭头函数
        if (!node ) return
        dfs(node.right)
        setArray.push(node.val)
        dfs(node.left)
    }
    dfs(root);      //函数调用
    return resArr[k - 1];
};
```

## 4、 (递归) 55 二叉树的深度

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

    3
   / \
  9  20
    /  \
   15   7
```

注意：从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

```
var maxDepth = function (root) {
    if (!root) return 0;//如果为空,返回0;
    return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
};
```

# 六、BFS宽度优先搜索

## 1、32-I.从上到下打印二叉树

```
从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。
给定二叉树: [3,9,20,null,null,15,7],返回[3,9,20,15,7]

    3
   / \
  9  20
    /  \
   15   7
```

注意：BFS一般指宽度优先搜索。宽度优先搜索算法（又称广度优先搜索）

定义两个变量，把树放入数组中，队列用数组实现。

```
var levelOrder = function(root) {
    if (!root) return [];//如果根为空，即树为空
    const res = [];//存放结果
    const queue = [root];//将 root 放入队列,最初queue的长度为1
    while (queue.length) {//当queue为空，返回
        const head = queue.shift();//数组的操作shift
        res.push(head.val);//取出队首元素，将 val 放入返回的数组中
        head.left && queue.push(head.left);
        //检查队首元素的子节点，若不为空，则将子节点放入队列
        head.right && queue.push(head.right);
        //检查队列是否为空，为空，结束并返回数组；不为空，回到第二步
    }
    return res;
};
```

## 2、 32-II从上到下打印二叉树

```
从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。
结果:[
  [3],
  [9,20],
  [15,7]
]
```

注意：结果为数组中每个元素依然为数组，需要加入的时候初始化；

```
var levelOrder = function(root) {
```

```
    //  实现BFS 构造队列queue实现
    if (!root) return [];
    const queue = [[root, 0]], res = [];
    while (queue.length) {
        const [node, level] = queue.shift();//移除首元素
        // 判断当前层是否已经初始化设置 [] 若无则初始化一下
        if (!res[level]) res[level] = [];//定义数组中每个元素为数组
        res[level].push(node.val);

        node.left && queue.push([node.left, level + 1]);
        node.right && queue.push([ node.right, level + 1 ]);
    }
    return res;
};
```

## 3、 ok32-III从上到下打印二叉树

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

```
[
  [3],
  [20,9],
  [15,7]
]
```

注意：不能采用第二种方法的思想，同理，将数据放入queue中，然后设置一个标志levelNum控制每层输出完毕，判断反转。

```
var levelOrder = function(root) {
    if (!root) return [];
    const queue = [root];
    const res = [];
    let level = 0; // 代表当前层数
    while (queue.length) {
        res[level] = []; // 第level层的初始化
        let levelNum = queue.length; // 第level层的节点数量
        while (levelNum--) {
            const head = queue.shift();
            res[level].push(head.val);
            head.left && queue.push(head.left);
            head.right && queue.push(head.right);
        }

        if (level % 2)  res[level].reverse();// 行号是奇数时，翻转当前层的遍历结果
        level++;
    }
    return res;
};
```

# 七、动态规划

```
写一个函数，输入 n ，求斐波那契（Fibonacci）数列的第 n 项（即 F(N)）。斐波那契数列的定义如下：
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
```

## 1、10-I斐波那契数列

注意：动态规划思想。

```
var fib = function(n) {
    let a=0,b=1;
    while(n--){
        res=(a+b)% 1000000007;
        a=b;b=res;
    }
    return a;
};
```

## 2、10-II青蛙跳台阶

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

注意：初始条件可以用简单值确认结果

```
var numWays = function(n) {
    let n1=1;n2=2;
    while(n-->1){
      res=(n1+n2)%1000000007;
        n1=n2;n2=res;
    }
    return n1;
};
```

## 3、42.连续子数组的最大和

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

注意：动态规划算法，找到最佳的转移方程（时间on，空间o1）

```
var maxSubArray = function(nums) {
    let pre = nums[0];
    let cur = nums[0];
    let sum = nums[0];      //存储结果
    for(let i = 1; i < nums.length; i++) {
        cur = pre >= 0? pre + nums[i] : nums[i];
        pre = cur;//pre保存目前的值
        sum = sum > pre? sum:pre    //返回大的那个值
    }
    return sum
};
```

## 4、46 .把数字翻译成字符串（动态规划）

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a” ，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。
输入: 12258
输出: 5
解释: 12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"

注意：动态规划，找到状态转移方程，dp(i) 表示前 i个数字的翻译方法数

当x[i-1]x[i]不可以组合时，dp[i]=dp[i-1]

当x[i-1]x[i]可以组合时，dp[i]=dp[i-2]+dp[i-1]

```
const translateNum = function(num)  {
  const str = num.toString();
  let prev = 1,cur = 1;
  for (let i = 2; i < str.length + 1; i++) {
    const temp=str.substr(i-2, 2)//temp =str[i - 2] + str[i - 1];
    if (temp>= '10' && temp <= '25') {
      const t = cur;    // 缓存上个状态
      cur = prev + cur; // 当前状态 = 上上个状态 + 上个状态
      prev = t;         // 更新上上个状态
    } else       prev = cur;
  }
  return cur;
}
```

## 5、47. 礼物的最大价值

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？
输入:  输出: 12  解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物
```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```

注意：两种表示方法，第二种为第一种的优化

数组：行数grid.length；列数grid[0].length；一维数组：`dp = Array(cols).fill(0)`

箭头函数初始化二维数组：`dp = Array(rows).fill(0).map(i => Array(cols).fill(0))`

递归，经典动态规划问题。每个位置的最优解是由它的上一位或者左一位决定的。即 `dp[i][j] = max( dp[i-1][j], dp[i][j-1] ) + gird[i][j]` ，我们按照状态方程，写代码

状态转移方程：`f(i,j)=max[f(i,j-1),f(i-1,j)]+grid(i,j);`

```
//法一：二维dp
 /*var maxValue = function(grid) {
    let rows = grid.length;
    if (!rows) return 0;//如果行为空，返回0;
```

```
    let cols = grid[0].length;
    //fill将一个固定值替换数组的元素。新建二维度的数组
    let dp = Array(rows).fill(0).map(i => Array(cols).fill(0))
    for(let i = rows-1; i>=0; i--) {
        for(let j = cols-1; j>=0; j--) {
            let a = i >= rows-1 ? 0 : dp[i+1][j];
            let b = j >= cols-1 ? 0 : dp[i][j+1];
            dp[i][j] = grid[i][j] + Math.max(a, b)
        }
    }
    return dp[0][0];//存放最高
}; */
//法二：一维dp，空间复杂度降低
var maxValue = function(grid) {
    let rows = grid.length;
    if (!rows) return 0
    let cols = grid[0].length;
    let dp = Array(cols).fill(0)
    for(let i = rows-1; i>=0; i--) {
        for(let j = cols-1; j>=0; j--) {
            //let a = i+1 >= rows ? 0 : dp[j];
            //let b = j+1 >= cols ? 0 : dp[j+1];
            let a = i >= rows-1 ? 0 : dp[j];
            let b = j >= cols-1 ? 0 : dp[j+1];
            dp[j] = grid[i][j] + Math.max(a, b);
        }
    }
    return dp[0]
};
```

## 6、49.丑数

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。

```
var nthUglyNumber = function(n) {
    var res=new Array(n);
    res[0]=1;
    let ptr2=0,ptr3=0,ptr5=0;// 下个数字永远 * 2/3/5
    for(let i=1;i<n;i++){
        res[i]=Math.min(res[ptr2]*2,res[ptr3]*3,res[ptr5]*5);
        // 说明前ptr2个丑数*2也不可能产生比i更大的丑数了
        // 所以移动ptr2
        if(res[i]==res[ptr2]*2) ptr2++;
        if(res[i]==res[ptr3]*3) ptr3++;
        if(res[i]==res[ptr5]*5) ptr5++;
    }
    return res[n - 1];//res从小到大排序
};
```

## 7、62. 圆圈中最后剩下的数字

0,1,···,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。
例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。
输入: n = 5, m = 3
输出: 3

注意：迭代+递归。经典的约瑟夫环。

//n个人：编号m%n被删除。从2开始反递归。n=1,f(1,m)=0;

// `f(2,m)=[f(1,m)+m]%2;` //恢复到两个人时最后结果的编号

// `f(3,m)=[f(2,m)+m]%3;` //恢复到三个人时最后结果的编号

// `f(n,m)=[f(n-1,m)+m]%n;` 删掉m个人将上次结果右移m个对原长度取余可恢复这次索引。

//主要思想，从最后结果恢复不断到最初的状态，即结果编号

```
var lastRemaining = function(n, m) {
    let res = 0;
    for(let i = 2; i <= n; i++) {
        res = (m + res) % i;
    }
    return res;//最后结果的索引号的递归。
};
```

## 8、63.股票的最大利润

假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少？
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。

```
//方法二：时间复杂度o(n)
var maxProfit = function(prices) {
    let minprice =Number.MAX_VALUE;  // JavaScript 中可表示的最大的数
    let maxprofit = 0;//没有买卖，结果为0；
    for (const price of prices) {
        maxprofit = Math.max(price - minprice, maxprofit);//保存最大利润，记录每天价格与最小值的差，即利润。比较每天的利润和目前最大利润的大小。
        minprice = Math.min(price, minprice);//保存数组中最小的值
    }
    return maxprofit;
};
```

# 八、树

## 68-I．二叉搜索树的最近公共祖先

```
给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

```
var lowestCommonAncestor = function (root, p, q) {
    if(!root) return null;
    if(p.val===q.val) return p;
    while(root){
        if(root.val < q.val && root.val < p.val) root = root.right;
        if(root.val > q.val && root.val > p.val) root = root.left;
        else return root;
    }
};
```

## 68-II．二叉树的最近公共祖先

```
给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。
```

注意：递归

```
var lowestCommonAncestor = function(root, p, q) {
    let ans;
    const dfs = (root, p, q) => {
        if (root === null) return false;
        const lson = dfs(root.left, p, q);
        const rson = dfs(root.right, p, q);
        if ((lson && rson) || ((root.val === p.val || root.val === q.val) &&
(lson || rson))) {
            ans = root;
        }
        return lson || rson || (root.val === p.val || root.val === q.val);
    }
    dfs(root, p, q);
    return ans;
};
```

# 九、位运算

## 15. 二进制中1 的个数

```
输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

注意：学会match的正则表达式，还有`toString()`的用法。

```
//法一
var hammingWeight = function(n) {
     const r =  n.toString(2).match(/1/g);//2表示基数，默认10；
     //正则表达式：/pattern/attributes。
     //"g"、"i" 和 "m"，分别用于指定全局匹配、区分大小写的匹配和多行匹配。
    return r ? r.length : 0
};
```

## 56-I 数组中数字出现的次数

```
一个整型数组 nums 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。
输入：nums = [1,2,10,4,1,4,3,3]
输出：[2,10] 或 [10,2]
```

注意：<<左移运算符；&按位与；异或(^)运算。任何数和本身异或则为 0；任何数和 0 异或是 本身

```
var singleNumbers = function(nums) {
  let a = 0;
  let b = 0;
  let c = 0;
  nums.forEach((item) => {
    c ^= item;
  });
  let mark = 1;              // mark就是分组凭据
  while((mark & c) === 0) { // 一直到找到第一个1为止
    mark <<= 1;
  }
  nums.forEach((item) => {
    if ((mark & item) === 0) { // 分组
      a ^= item
    } else {
      b ^= item;
    }
  });
  return [a, b];
};
```

## 65. 不用加减乘除做加法

```
输入: a = 1, b = 1
输出: 2
```

注意：充分理解异或，与运算；**无进位和** 与 **异或运算** 规律相同，**进位** 和 **与运算** 规律相同（并需左移一位）

```
var add = function(a, b) {
    while (b) {//直至进位为0.输出结果非进位
        let c = (a & b) << 1 // 进位 = 与操作，再左移1位
        a ^= b               // 非进位 =异或操作
        b = c                // 进位
    }
    return a;
};
```

# 十、递归

## 64. 求1+2+...+n

```
求 1+2+...+n ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。
输入: n = 3    输出: 6
```

注意：只有加减法、赋值、位运算符以及逻辑运算符；前n项和公式；

```
//将乘除转加减
var sumNums = function (n) {
    return Math.round(Math.exp(Math.log(n) + Math.log(n + 1) - Math.log(2)));
}; */

/* //递归
var sumNums = function(n) {
    return n && sumNums(n - 1) + n;
}; */
//幂运算加移位
var sumNums = function (n) {
    return (n ** 2 + n) >> 1;// **幂次方
};
```