一、数组

1.二维数组的查找

- 题目描述：在一个二维数组中（每个一维数组的长度相同），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。
- 解答方法：
  1. 暴力解法——自己想到的解法
     - 思路：两层for遍历，如果目标值与遍历的某个值相等，则返回true
     - 代码：

```
function Find(target, array)
{
    const rowNum = array.length
    if(!rowNum){
        return false
    }
    const colNum = array[0].length
    if(!colNum){
        return false
    }
    for(let i=0;i<rowNum;i++){
        for(let j=0;j<colNum;j++){
            if(target === array[i][j]){
                return true
            }
        }
    }
    return false
}
```

     - 时间复杂度：O(N^2)，空间复杂度：O(1)
  2. 观察数组规律（最优）
     - 思路：数组的特点是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。考虑以下数组：

```
1 2 3
4 5 6
7 8 9
```

       在其中寻找 5 是否存在。过程如下：
       - 从右上角开始遍历
       - 目标元素大于目标元素(5 > 3)，根据数组特点，当前行中最大元素也小于目标元素，因此进入下一行

        - 目标元素小于当前元素(5<6)，根据数组特点，行数不变，尝试向前一列查找
        - 找到5
    - 代码：

```
function Find(target, array)
{
    const rowNum = array.length
    if(!rowNum){
        return false
    }
    const colNum = array[0].length
    if(!colNum){
        return false
    }

    let row = 0,
        col = colNum - 1
    while(row < rowNum && col >= 0){
        if(array[row][col] === target){
            return true
        }else if(target > array[row][col]){
            row++
        }else{
            col--
        }
    }
}
```

    - 时间复杂度是O(M+N)，空间复杂度是O(1)。其中M和N分别代表行数和列数。

2.调整数组顺序使奇数位于偶数前面

- 题目描述：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。
- 解答方法：
    - 方法一——自己想到的解法
        - 思路：遍历数组，判断遍历的每个值是奇数还是偶数，如果是奇数push到一个新建的左侧数组里，如果是偶数push到一个新建的右侧数组里，最后合并左右侧数组，返回即可
        - 代码：

```
function reOrderArray(array)
{
    // write code here
    const len = array.length
    if(!len){
        return false
    }
    const left_temp = []
    const right_temp = []
    for(let i=0;i<len;i++){
        //奇数
```

```
        if(array[i]%2===1){
            left_temp.push(array[i])
        }else{
            //偶数
            right_temp.push(array[i])
        }
    }
    array = left_temp.concat(right_temp)
    return array
}
```

- 时间复杂度：O(N)，空间复杂度：O(N)

- 方法二——自己想到的解法
  - 思路：采用插入排序的思想实现
  - 代码：

```
function reOrderArray(array)
{
    let k = 0 //记录已经摆好位置的奇数的个数
    for(let i=0;i<array.length;i++){
        if(array[i]%2===1){
            let j = i
            while(j>k){
                let temp = array[j]
                array[j] = array[j-1]
                array[j-1] = temp
                j--
            }
            k++
        }
    }
    return array
}
```

  - 时间复杂度：O(N^2)，空间复杂度：O(1)

3.把数组排成最小的数

- 题目描述：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为321323。
- 解答方法：
  - 思路：借助自定义排序，可以快速比较两个数的大小。比如只看[3, 32]这两个数字。它们可以拼接成 332 和 323，按照题目要求，这里应该取 323。
  - 代码：

```
function PrintMinNumber(numbers)
{
    const len = numbers.length
    if(!len){
        return ""
    }
```

```
    numbers.sort((x,y) => {
        let xx = x + "" + y
        let yy = y + "" + x
        return xx - yy
    })
    return numbers.join("")
}
```

- 时间复杂度：O(N^2)——v8用的插入排序，空间复杂度：O(1)

4.数组中的逆序对

- 题目描述：在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007。题目保证输入的数组中没有的相同的数字。例如输入：[1,2,3,4,5,6,7,0]，输出：7
- 解答方法：
  1. 暴力破解法——无法通过（牛客网会限制时间，这种办法的时间复杂度为O(N^2)，太高，不推荐）
     - 思路：遍历，从第一个数开始，每次比较这个数和这个数之后的每一个数，统计次数即可
     - 代码：

```
function InversePairs(data){
    const len = data.length
    let num = 0
    for(let i=0;i<len-1;i++){
        let newarr = data.slice(i+1)
        if(newarr.length!==0){
            for(let j=0;j<newarr.length;j++){
                if(data[i] > newarr[j]){
                    num++
                }
            }
        }
    }
    return num%1000000007
}
```

     - 时间复杂度：O(N^2)
  2. 归并排序求解
     - 思路：归并排序的改进，把数据分成前后两个数组(递归分到每个数组仅有一个数据项)，合并数组，合并时，当出现前面的数组值array[i]大于后面数组值array[j]时，则前面数组array[i]~array[mid]都是大于array[j]的，此时 count += mid+1 - i
     - 代码：

```
var sum  // 用于统计逆序对个数
function InversePairs(data){
    sum = 0  // 用于统计逆序对个数
    let start = 0
    let end = data.length - 1
    divide(start,end,data)
```

```
    return sum % 1000000007
}
function divide(start, end, data){
    if(start !== end){
        let mid = (start+end)>>1    //长度取一半
        divide(start, mid, data)  //左区间划分
        divide(mid+1, end, data)  //右区间划分
        merge(start, end, mid, data)
    }
}
function merge(start, end, mid, data){
    let i = start  //左区间的起点
    let j = mid + 1 //右区间的起点
    let temp = new Array(end-start+1)  //创建一个临时数组
    let index = 0 ///临时下标
    while(i <= mid && j <= end){
        if(data[i] > data[j]){
            temp[index++] = data[j++]
            sum += mid - i + 1  //这行是核心,去统计逆序对个数,统计的基础是在归并排序的合并过程中,合并的两个子序列都是有序的
        }else{
            temp[index++] = data[i++]
        }
    }
    while(i <= mid){
        temp[index++] = data[i++]
    }
    while(j <= end){
        temp[index++] = data[j++]
    }
    let pindex = 0
    for(let k=start; k <= end; k++){
        data[k] = temp[pindex++]
    }
}
```

- 时间复杂度： O(NlogN)
- 下面这个代码，也是归并排序思路解决，但是不知道为什么正确率只有75%

```
var sum
function InversePairs(data){
    sum = 0
    mergeSort(data)
    return sum%1000000007
}

function mergeSort(ary) {
    let len = ary.length
    if(len<2) return ary
    let middle = Math.floor(len/2)
    let left = ary.slice(0,middle)
    let right = ary.slice(middle)
    return merge(mergeSort(left),mergeSort(right))
```

```
}

function merge(left, right) {
    let temp = []
    while(left.length && right.length){
        if(left[0]>right[0]){
            sum += left.length   //统计次数
            temp.push(right.shift())
        }else{
            temp.push(left.shift())
        }
    }
    if(left.length){
        temp = temp.concat(left)
    }
    if(right.length){
        temp = temp.concat(right)
    }
    return temp
}
```

5.数组中出现次数超过一半的数字

- 题目描述：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。

1. 方法一——哈希表方法
    - 思路：遍历数组，将数组里的不同值当做一个临时对象的属性名，临时对象的属性值为数组里每一个数字出现的频率，最后输出临时对象中属性值最大的那个属性名（大于数组长度一半才输出）即可
    - 代码：

```
function MoreThanHalfNum_Solution(numbers)
{
    const len = numbers.length
    let temp = {}
    for(let i=0;i<len;i++){
        //if 里不推荐使用 Object.keys(temp).indexOf(numbers[i].toString()) !== -1
        if(temp[numbers[i]] !== undefined){
            temp[numbers[i]] ++
        }else{
            temp[numbers[i]] = 1
        }
    }
    let num = 0
    let index = null
    Object.keys(temp).forEach((item)=>{
        if (temp[item] > num){
            num = temp[item]
            index = item

        }
    })
```

```
    if(num > len/2){
        return index
    }
    return 0
}
```

- 时间复杂度： O(N)
- 另一种解法：（和方法一类似，更简洁）

```
function MoreThanHalfNum_Solution(numbers)
{
    const len = numbers.length
    if(len===1) return numbers[0]
    let temp = {}
    for(let i=0;i<len;i++){
        //if 里不推荐使用 Object.keys(temp).indexOf(numbers[i].toString()) !== -1
        if(temp[numbers[i]] !== undefined){
            temp[numbers[i]] ++
            if(temp[numbers[i]]>len/2) return numbers[i]
        }else{
            temp[numbers[i]] = 1
        }
    }
    return 0
}
```

2. 方法二——数组索引法
   - 思路：遍历数组，将遍历的每个值重新当做一个新数组的索引，如果遍历的每个值相同，则新数组索引对应的值++，否则为1。最后返回新数组里面最大的值对应索引即可。
   - 缺点：如果数组中的某个值很大很大，那个遍历新数组时，性能消耗就变大了
   - 代码：

```
function MoreThanHalfNum_Solution(numbers){
    let arr = []
    let len = numbers.length
    for(let i = 0; i < len; i++){
        let index = numbers[i]
        arr[index] !== undefined ? arr[index]++ : arr[index] = 1
    }
    var arrLen = arr.length
    var pindex = -1
    var max = -1
    for(var i = 0; i < arrLen; i++){
        if(!arr[i]) continue
        max = arr[i] > max ? (pindex = i, arr[i]) : max
    }
    return max > len / 2 ? pindex : 0
}
```

   - 时间复杂度： O(max(M,N))，M为输入数组里面的最大值，N为输入数组长度
3. 方法三——排序法

- 思路：先对数组进行排序，如果数组中数字出现的次数超过数组长度一半，那么排序后的数组的中间那个数必然是数组中出现次数最多的那个数字
- 代码：

```
function MoreThanHalfNum_Solution(numbers){
    let len = numbers.length
    if(len<1){
        return 0
    }
    let count = 0
    numbers.sort((x,y)=>{
        return x-y
    })
    let num = numbers[Math.floor(len/2)]
    for(let i=0;i<len;i++){
        if(num === numbers[i]){
            count ++
        }
    }
    if(count > len/2){
        return num
    }
    return 0
}
```

- 时间复杂度： 至少为O(NlogN)，因为排序有个时间复杂度

4. 方法四——相消法——思路很牛逼
    - 思路：用preValue记录上一次访问的值，count表明当前值出现的次数，如果下一个值和当前值相同那么count++；如果不同count--，减到0的时候就要更换新的preValue值了，因为如果存在超过数组长度一半的值，那么最后preValue一定会是该值。
    - 代码：

```
function MoreThanHalfNum_Solution(numbers){
    let  preValue = numbers[0] //用来记录上一次的记录
    let count = 1 //preValue出现的次数（相减之后）
    for(let i = 1; i < numbers.length; i++){
        if(numbers[i] === preValue){
            count ++
        }else{
            count --
            if(count === 0){
                preValue = numbers[i];
                count = 1;
            }
        }
    }
    let num = 0;//需要判断是否真的是大于1半数
    for(let i=0; i < numbers.length; i++){
        if(numbers[i] == preValue){
            num++;
        }
```

```
    }
    return num > numbers.length/2 ? preValue : 0;
}
```

- 时间复杂度： O(N)

6.数字在排序数组中出现的次数

- 题目描述：统计一个数字在排序数组中出现的次数。
- 解答方法：
  - 方法——不推荐
    - 思路：因为是有序数组，所以直接遍历，统计指定数字出现的次数即可
    - 代码：

```
function GetNumberOfK(data, k)
{
    let num = 0
    for(let i=0;i<data.length;i++){
        if(data[i]===k){
            num++
        }
        if(num&&data[i]!==k){
            break
        }
    }
    return num
}
```

    - 时间复杂度：O(N)
  - 方法二
    - 思路：利用indexOf和lastIndexOf
    - 代码：

```
function GetNumberOfK(data, k)
{
    if(data.indexOf(k) == -1) return 0
    let count = data.lastIndexOf(k)-data.indexOf(k)
    return count+1
}
```

  - 方法三——常规二分法
    - 思路：二分法
    - 代码：

```
function GetNumberOfK(data, k)
{
    let index = binarySearch(data,k)
    if(index < 0){
        return 0
    }
```

```
    let left = index - 1
    let right = index + 1
    let len = 1
    while(left>=0 && data[left--]===k) len++
    while(right<=data.length-1 && data[right++] === k) len++
    return len
}
function binarySearch(data,k){
    let start = 0, end = data.length-1
    while(start <= end){
        let mid = Math.floor((end - start)/2) + start
        if(data[mid] < k){
            start = mid + 1
        }else if(data[mid] > k){
            end = mid - 1
        }else{
            return mid
        }
    }
    return -1
}
```

- 方法四——特殊二分法
  - 思路：看到排序数组，首先应该想到二分法
  - 代码：因为数组data中都是整数，所以可以稍微变一下，不是搜索k的两个位置，而是搜索k-0.5和k+0.5这两个数应该插入的位置，然后相减即可。

```
function GetNumberOfK(data, k)
{
    return binarySearch(data, k+0.5) - binarySearch(data, k-0.5)
}
function binarySearch(data,k){
    let start = 0, end = data.length-1
    while(start <= end){
        let mid = Math.floor((end - start)/2) + start
        if(data[mid] < k){
            start = mid + 1
        }else if(data[mid] > k){
            end = mid - 1
        }
    }
    return start
}
```

7.数组中只出现一次的数字

- 题目描述：一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。
- 解答方法：
  - 方法一——自己想到的解法(和第5题方法一类似)——哈希表方法

- 思路：遍历数组，将数组里的不同值当做一个临时对象的属性名，临时对象的属性值为数组里每一个数字出现的频率，最后输出出现频率为1的数字即可
- 代码：

```
function FindNumsAppearOnce(array)
{
    const len = array.length
    let temp = {}
    for(let i=0;i<len;i++){
        // if里不推荐使用 Object.keys(temp).indexOf(array[i].toString()) !== -1
        if(temp[array[i]] !== undefined){
            temp[array[i]] ++
        }else{
            temp[array[i]] = 1
        }
    }
    let list = []
    Object.keys(temp).forEach((item)=>{
        if (temp[item] === 1){
            list.push(item)
        }
    })
    if(list.length === 2){
        return list
    }
    return null
}
```

- 时间复杂度：O(N)

◦ 方法二——有技巧性

- 思路：indexOf()和lastIndexOf()，只要两个相等，就是只出现一次的数
- 代码：

```
function FindNumsAppearOnce(array)
{
    const len = array.length
    let list = []
    for(let i=0;i<len;i++){
        if(array.indexOf(array[i]) === array.lastIndexOf(array[i])){
            list.push(array[i])
        }
    }
    if(list.length === 2){
        return list
    }
    return null
}
```

◦ 方法三——异或方法——空间复杂度较小

- 思路：对数组里的所有元素进行异或操作，最后的结果就是那两个出现 1 次的数异或的结果，然后判断这个结果里第几位是 1 ，根据这个将数组分为两个不同的子数组。这两个子数组里分别包含了一个出现次数为1的数，两个子数组分别求异或，即可得到最终结果
- 代码：

```
function FindNumsAppearOnce(array)
{
    let tmp = array[0]
    //对所有元素进行异或操作，最后的结果就是那两个出现 1 次的数异或的结果
    for(let i=1;i<array.length;i++){
        tmp = tmp ^ array[i]
    }
    if(tmp === 0) return
    let index = 0 //记录第几位是1
    while((tmp & 1) === 0){
        tmp = tmp >> 1
        index++
    }
    let num1 = 0
    let num2 = 0
    for(let i=0;i<array.length;i++){
        if(isOneAtIndex(array[i],index)) num1 = num1 ^ array[i]
        else num2 = num2 ^ array[i]
    }
    return [num1, num2]
}
function isOneAtIndex(num, index){
    num = num >> index
    return num & 1
}
```

8.数组中重复的数字

- 题目描述：在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。
- 解答方法：
  - 方法一——数组排序再扫描
    - 思路：数组先排序，然后遍历，当某个数重复次数等于2，则输出
    - 代码：

```
function duplicate(numbers, duplication)
{
    // write code here
    //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
    numbers.sort((x,y) => {
        return x-y
    })
    let k = numbers[0]
```

```
    for(let i=1;i<numbers.length;i++){
        if(numbers[i]===k){
            duplication[0] = k
            return true
        }else{
            k = numbers[i]
        }
    }
    return false

}
```

- 时间复杂度O(NlogN)——使用了sort排序的原因

- 方法二——哈希表方法
  - 思路：遍历数组，将数组里的不同值当做一个临时对象的属性名，临时对象的属性值为数组里每一个数字出现的频率，当出现频率为2，则代表找到重复的数字了
  - 代码：

```
function duplicate(numbers, duplication)
{
    // write code here
    //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
    const len = numbers.length
    let temp = {}
    for(let i=0;i<len;i++){
        //或者 typeof temp[numbers[i]] === 'undefined'
        if(temp[numbers[i]] === undefined){
            temp[numbers[i]] = 1
        }else{
            duplication[0] = numbers[i]
            return true
        }
    }
    return false

}
```

- 方法三——不需要额外的空间消耗，时间效率是O(n)
  - 思路：题目里写了数组里数字的范围保证在0 ~ n-1 之间，所以可以利用现有数组设置标志，当一个数字被访问过后，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：

```
function duplicate(numbers, duplication)
{
    // write code here
    //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
    let len = numbers.length
    for(let i=0;i<len;i++){
```

```
        let index = numbers[i]
        if(index >= len) index -= len
        if(numbers[index] >= len){
            duplication[0] = index
            return true
        }
        numbers[index] = numbers[index]+len
    }
    return false
}
```

9.旋转数组的最小数字

- 题目描述：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。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。
- 解答方法：
  - 方法一——不可取
    - 思路：数组直接从小到大排序，然后取出最小的数字即可
    - 代码：

```
function minNumberInRotateArray(rotateArray)
{
    if(rotateArray && rotateArray.length !== 0){
        let arr = [...rotateArray]
        arr.sort((x,y)=>{
            return x-y
        })
        return arr[0]
    }
    return -1
}
```

    - 注意：sort排序，10个数及以内是插入排序，10个数以后是快速排序。如果是10个数以内的排序，使用 return x > y即可。超过10个数，为了保证sort不乱序，必须 把三种情况列出来，return 1 return -1 return 0，写成 return x-y也是可以的。为了保证10个数以内和以外都可以排序，对于数字排序，推荐使用 return x-y
    - 补充：这种方法复杂度是 O(N)，并且没有利用“旋转数组”的特性
  - 方法二——这种方法复杂度比 O(N) 小一点
    - 思路：通过当前位置的值和后一位置的值进行比较
    - 代码：

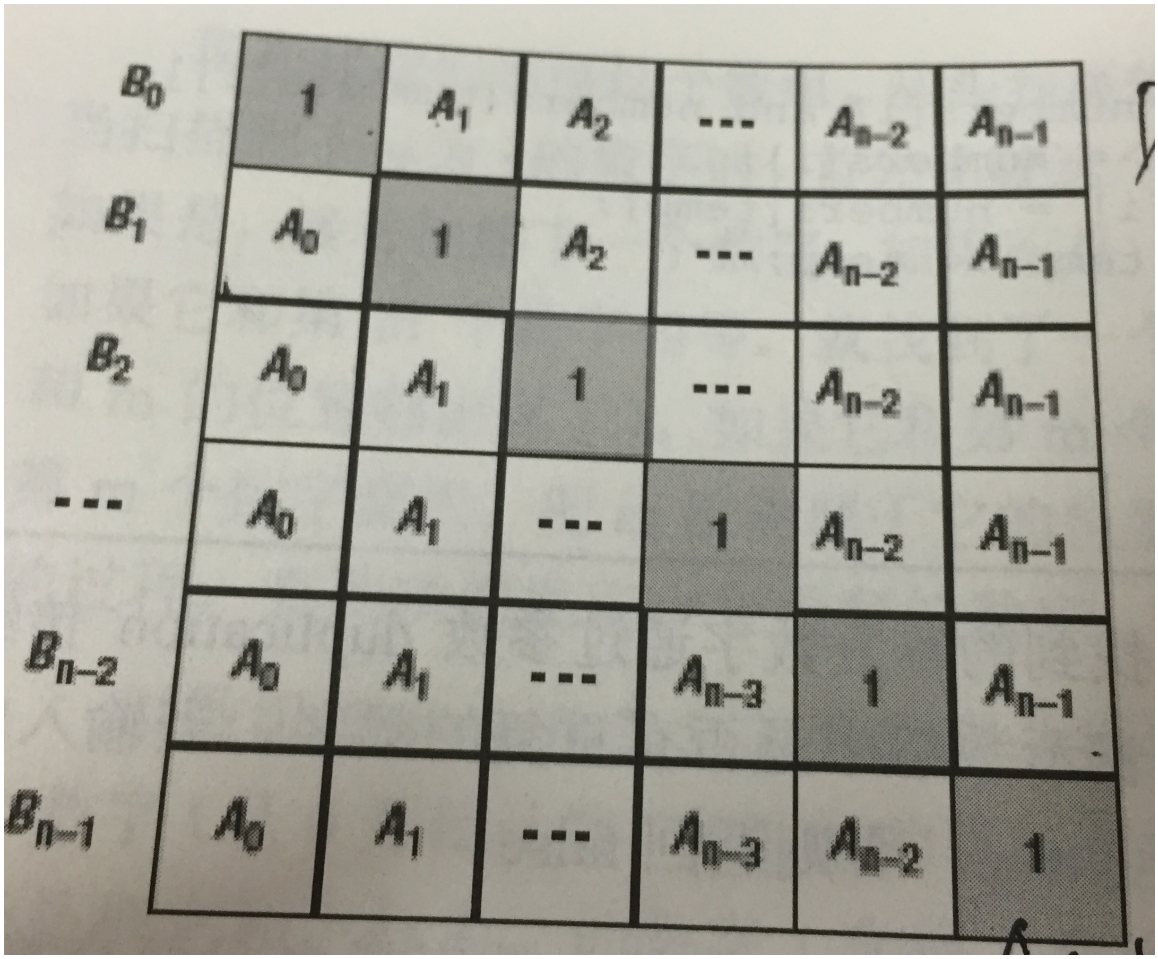
```
function minNumberInRotateArray(rotateArray)
{
  if(rotateArray.length === 0){
        return 0
    }
    let min = 0
    while(rotateArray[min]<=rotateArray[min+1]){
        min++
    }
    return rotateArray[min+1]
}
```

- 方法三——二分法——时间复杂度 O(lgN)
  - 思路：用两个指针left、right分别指向数组的第一个元素和最后一个元素，按照题目的旋转的规则，第一个元素应该是大于等于最后一个元素的。找到数组的中间元素，中间元素大于第一个元素，则中间元素位于`前面的递增子数组`，此时最小元素位于中间元素的后面，我们可以让第一个指针left指向中间元素。中间元素小于第一个元素，则中间元素位于`后面的递增子数组`，此时最小元素位于中间元素的前面，我们可以让第二个指针right指向中间元素。
  - 代码：

```
function minNumberInRotateArray(rotateArray)
{
    let left = 0,
        right = rotateArray.length - 1
    let mid
    while(left+1<right){
        mid = Math.floor((left+right)/2)
        if(rotateArray[mid] >= rotateArray[left]){
            left = mid
        }else{
            right = mid
        }
    }
    return rotateArray[right]
}
```

10.构建乘积数组

- 题目描述：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。（注意：规定B[0] = A[1] * A[2] * ... * A[n-1]，B[n-1] = A[0] * A[1] * ... * A[n-2];）
- 解答方法：
  - 方法：
    - 图示：



- 思路：B[i]的值可以看作下图的矩阵中每行的乘积。下三角用连乘可以很容易求得，先算下三角中的连乘，即先计算出B[i]中的一部分，然后将上三角中的数也乘进去。这样一来就只需要两个循环就可以解决这个问题。时间复杂度是O(n)
- 代码：

```
function multiply(array)
{
    let len = array.length
    let B = []
    B[0] = 1
    //下三角
    for(let i=1;i<len;i++){
        B[i] = B[i-1]*array[i-1]
    }
    let temp = 1
    //上三角
    for(let i=len-2;i>=0;i--){
        temp = temp*array[i+1]
        B[i] = B[i]*temp
    }
    return B
}
```

11.滑动窗口的最大值

- 题目描述：给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 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]}。
- 解答方法：
  - 方法一：暴力法
    - 思路：直接移动这个滑动窗口，每次统计窗口中的最大值即可
    - 复杂度：
      - 时间复杂度是O(kN)，其中 k 是滑动窗口的长度，N为数组中的个数
      - 空间复杂度是O(N-k+1)
    - 代码：

```
function maxInWindows(num, size)
{
    if(size === 0){
        return []
    }
    var count = num.length - size
    var result = []
    for(var i=0;i<=count;i++){
        var temp = num.slice(i,i+size)
        result.push(Math.max.apply(this,temp))
    }
    return result
}
```

  - 方法二：单调双端队列——时间复杂度是O(N)
    - 思路：（头，尾，尾，头）
      - 第一步，头部出队，清理超范围
      - 第二步，移除尾部小于当前值的元素
      - 第三步，尾部入队
      - 第四步，返回头部——当前窗口最大值
    - 复杂度：
      - 时间复杂度是O(N)，其中 N为数组中的个数
    - 代码：

```
function maxInWindows(num, size)
{
    if(size === 0){
        return []
    }

    let n = num.length
    let res = [] //存储最终结果
    let dq = [] //滑动窗口 存储的是数组的index，不是数组的值

    for(let i=0;i<n;i++){
        // 1.头：移除头部，保证窗口的长度范围
        if(dq.length!==0 && dq[0]<(i-size+1)){
```

```
            dq.shift()
        }
        // 2.尾：移除尾部小于当前值的元素
        while(dq.length!==0 && num[i]>=num[dq[dq.length-1]]){
            dq.pop()
        }
        // 3.尾：尾部加入，滑动窗口向右扩充
        dq.push(i)
        //4.头：从头部返回极大值
        if(i>=size-1){
            res.push(num[dq[0]])
        }
    }
    return res
}
```

二、字符串

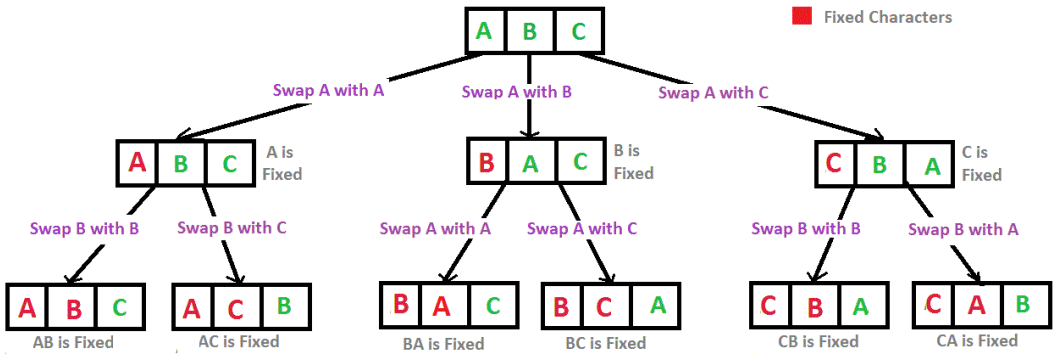
1.替换空格

- 题目描述：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。
- 解答方法：
  - 方法：
    - 思路：使用split或者正则替换
    - 代码：

```
function replaceSpace(str)
{
    //let newstr = str.split(" ").join("%20")
    let newstr = str.replace(/\s/g,"%20")
    return newstr
}
```

2.字符串的排列——回溯

- 题目描述：输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。
  - 输入描述：输入一个字符串,长度不超过9(可能有字符重复),字符只包括大小写字母。
- 解答方法：
  - 方法———回溯思想——推荐！
    - 思路：通过递归的去查找每一个位置的字符可能出现的情况。
      - 图示：



Recursion Tree for Permutations of String "ABC"

- 代码：

```
function Permutation(str){
    let a = str.split("")  //将字符串转换为数组    使用Array.from也可以
    let ans = []
    solve(ans,a,0,str.length)
    //去重操作
    ans = Array.from(new Set(ans))
    //字典排序
    ans.sort()
    return ans
}
function solve(ans,a,index,length){
    if(index === length-1){
        let res = a.join("")
        ans.push(res)
    }else{
        // 就说明现在要去确定index位置的字符了
        for(let i=index;i<length;i++){
            let temp = a[i]
            a[i] = a[index]
            a[index] = temp

            //当前index位置的字符已经通过交换找到了,那么就递归去找下一个位置的字符
            solve(ans,a,index+1,length)

            //其实就是为了消除当前层去递归的时候进行交换字符的影响
            temp = a[i]
            a[i] = a[index]
            a[index] = temp
        }
    }
}
```

- 方法二——回溯思想
  - 思路：也是回溯，但是增加了标志判断，来去除重复选择的字符
  - 代码：

```
console.log(Permutation("abc"))
```

```
function Permutation(str) {
    let arr = str.split("") //将字符串转换为数组   使用Array.from也可以
    let flg = new Array(arr.length).fill(false)
    let ans = []
    let path = []
    solve(arr, flg, 0, path, ans)
    //去重操作
    ans = Array.from(new Set(ans))
    //字典排序
    ans.sort()
    return ans
}

function solve(arr,flg, index, path, ans) {
    if (path.length === arr.length) {
        let res = path.join("")
        ans.push(res)
    } else {
        for (let i = 0; i < arr.length; i++) {
            if(!flg[i]){
                flg[i] = true
                path.push(arr[i])
                solve(arr,flg, index + 1, path, ans)
                path.pop()
                flg[i] = false
            }
        }
    }
}
```

- 方法三——回溯思想
  - 思路：也是回溯，每添加新元素，都判断该元素是不是已存在数组里了
  - 代码：

```
console.log(Permutation("abc"))
function Permutation(str) {
    let arr = str.split("") //将字符串转换为数组   使用Array.from也可以
    let ans = []
    let path = []
    solve(arr, 0, path, ans)
    //去重操作
    ans = Array.from(new Set(ans))
    //字典排序
    ans.sort()
    return ans
}
function solve(arr, index, path, ans) {
    if (path.length === arr.length) {
        let res = path.join("")
        ans.push(res)
    } else {
        for (let i = 0; i < arr.length; i++) {
            if(path.indexOf(arr[i]) === -1){
```

```
                path.push(arr[i])
                solve(arr, index + 1, path, ans)
                path.pop()
            }
        }
    }
}
```

3.左旋转字符串

- 题目描述：汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。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。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它！
- 解答方法：
  - 方法：
    - 思路：简单的字符串操作
    - 代码：

```
function LeftRotateString(str, n){
    //先进行字符判断，否则牛客无法提交成功
    if (str && str.length !== 0) {
        let len = str.length
        n = n % len
        str += str
        return str.substr(n, len)
    }
    return ""
}
```

    - 类似

```
function LeftRotateString(str, n)
{
    if(str && str.length !== 0){
        let offset = n % str.length
        let str1 = str.slice(0,offset)
        let str2 = str.slice(offset)
        return str2+str1
    }
    return ""
}
```

4.翻转单词顺序列

- 题目描述：牛客最近来了一个新员工Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事Cat对Fish写的内容颇感兴趣，有一天他向Fish借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat对一一的翻转这些单词顺序可不在行，你能帮助他么？

- 解答方法：
  - 方法：
    - 思路：字符串分割为数组，然后将新数组数组倒序，然后再拼接成字符串
    - 代码：

```
function ReverseSentence(str)
{
    if(str && str.length !== 0){
        return str.split(" ").reverse().join(" ")
    }
    return ""
}
```

5.把字符串转换成整数

- 题目描述：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
  - 输入描述：输入一个字符串,包括数字字母符号,可以为空
  - 输出描述：如果是合法的数值表达则返回该数字，否则返回0
  - 示例：
    - 输入：+2147483647 输出：2147483647
    - 输入：1a33 输出：0
- 解答方法：
  - 方法一：（自己的思路）
    - 思路：判断字符串的第一个字符，如果为 + 或者 - ，则从第二个字符开始，如果后面全是数字字符，则返回符号的整数，如果有一个非数字字符，则返回0。如果字符串的第一个字符是数字字符，则直接统计后面的所有字符是不是数字，是的话返回整数，否则返回0
    - 代码：

```
function StrToInt(str)
{
    if(str && str.length!==0){
        const len = str.length
        if(str[0] === '+' || str[0] === '-'){
            let all = 0
            let temp = null
            for(let i = 1; i < len; i++){
                temp = str[i] - '0'
                if(isNaN(temp) === false){   //数字
                    all = all*10+temp
                }else{  //非数字
                    return 0
                }
            }

            if(str[0] === '+'){
                return all
            }
```

```
            if(str[0] === '-'){
                return -all
            }
        }else if(isNaN(str[0] - '0') === false){ //数字
            let total = 0
            let number = null
            for(let i = 0; i < len; i++){
                number = str[i] - '0'
                if(isNaN(number) === false){   //数字
                    total = total*10 + number
                }else{  //非数字
                    return 0
                }
            }
            return total
        }

    }
    return 0
}
```

- 优化后的代码：

```
function StrToInt(str)
{
    if (str && str.length !== 0) {
        let res = 0,
            flag = 1 // 标识第一位是加号还是减号，即最后结果是正是负
    const n = str.length
        if (str[0] === '-') flag = -1
    for (let i = str[0] === '+' || str[0] === '-' ? 1 : 0; i < n; i++) { // 若第一位不是正负号，就从第二位开始
          if (!(str[i] >= '0' && str[i] <= '9')) return 0; // 如果不是数字0到9，直接返回0
          res = res * 10 + (str[i]-'0')
        }
        return res * flag
    }
    return 0

}
```

- 方法二——使用库函数转换
  - 思路：先使用Number函数判断字符串是不是可转换为数字，如果可以，则用parseInt转换为整数，如果为NaN，则返回0
  - 代码：

```
function StrToInt(str)
{
    return Number(str) ? parseInt(str) : 0
}
```

6.表示数值的字符串

- 题目描述：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。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。
- 解答方法：
  - 方法一：
    - 思路：使用Number函数判断
    - 代码：

```
//s字符串
function isNumeric(s)
{
    let num = Number(s)
    if(isNaN(num)){
        return false
    }else{
        return true
    }
}
```

  - 方法二：
    - 思路：使用 + 符号判断
    - 代码：

```
//s字符串
function isNumeric(s)
{
    if(+s){
        return true
    }else{
        return false
    }
}
```

7.字符流中第一个不重复的字符

- 题目描述：请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。
- 解答方法：
  - 方法一：
    - 思路：创建了一个对象map，利用对象键值对（属性：属性值）来记录每一个字符和出现次数。最后遍历对象的属性，返回出现次数为1的那一个属性即可。
    - 代码：

```
//Init module if you need
let map
function Init()
{
    // write code here
```

```
    map = {}
}
//Insert one char from stringstream
function Insert(ch)
{
    // write code here
    map[ch] = map[ch] ? map[ch]+1 : 1
}
//return the first appearence once char in current stringstream
function FirstAppearingOnce()
{
    // write code here
    for (let i in map) {
        if (map[i] === 1) {
            return i
        }
    }
    return '#'
}
```

- 注意：不是说 for in 无法按顺序打印吗？为什么这里可以用 for in 遍历
- 这种写法也是可以的

```
//Init module if you need
let arr
function Init()
{
    // write code here
    arr =[]

}
//Insert one char from stringstream
function Insert(ch)
{
    // write code here
    arr.push(ch)
}
//return the first appearence once char in current stringstream
function FirstAppearingOnce()
{
    // write code here
    for(let i=0;i<arr.length;i++){
        if(arr.indexOf(arr[i]) === arr.lastIndexOf(arr[i])){
            return arr[i]
        }
    }
    return "#"
}
```

8.第一个只出现一次的字符

- 题目描述：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）.

- 解答方法：
  - 方法一：
    - 思路：利用indexOf函数和lastIndexOf函数即可判断
    - 代码：

```
function FirstNotRepeatingChar(str)
{
    const len = str.length
    for(let i=0;i<len;i++){
        if(str.lastIndexOf(str[i]) === str.indexOf(str[i])){
            return i
        }
    }
    return -1

}
```

  - 方法二——哈希表方法
    - 思路：新建一个对象，其中key用来存放字符，value用来存放该字符出现的次数；第一次循环，将所有字符和对应出现的次数存放在map中，时间复杂度为0(n)；第二次循环找到value为1的字符所在的位置，并返回。
    - 代码：

```
function FirstNotRepeatingChar(str) {
    if(str && str.length !== 0){
        let map = {}
        const len = str.length
        for(let i=0;i<len;i++){
            map[str[i]] = map[str[i]]? map[str[i]]+1:1
        }
        for(let j=0;j<len;j++){
            if(map[str[j]] == 1){
                return j
            }
        }
    }
    return -1
}
```

9.二进制中1的个数

- 题目描述：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。
- 解答方法：
  - 方法一：
    - 思路：通过toString方法将整数转换为二进制字符串，然后遍历，统计字符为'1'的个数即可
    - 代码：

```
function NumberOf1(n){
    let str
```

```
    if(n>=0){
        //正数直接转换
        str = n.toString(2)
    }else{
        //在32位下，js负数的补码的二进制等于2的32次方减去其正数的值的二进制
        n = Math.pow(2, 32) - Math.abs(n)
        str = n.toString(2)
    }
    const len = str.length
    let num = 0
    for(let i=0;i<len;i++){
        if(str[i] === '1'){
            num++
        }
    }

    return num
}
```

- 方法二：移位
  - 思路：1和n进行位运算，结果为1则n的二进制最右边一位为1，否则为0；将n二进制形式逻辑右移（无符号右移）1位，继续与1进行位运算；由于有可能是负数，所以使用不考虑符号位的逻辑右移。
  - 代码：

```
function NumberOf1(n){
    let count = 0
    while (n != 0) {
        if ((n & 1) == 1) {
            count++
        }
        n = n >>> 1  //js中 >>>是无符号右移，>>是有符号移位
    }
    return count
}
```

- 方法三：移位
  - 思路：用1和n进行位运算，结果为1则n的二进制最右边一位为1，否则为0；将1左移一位继续进行位运算，直到左移32位截至。
  - 代码：

```
function NumberOf1(n) {
    if (n === 0) {
        return 0
    }
    var count = 0,
        flag = 1
    while (flag) {
        if (n & flag) {
            count++
        }
        flag = flag << 1
    }
```

```
    return count
}
```

10.正则表达式匹配

- 题目描述：请实现一个函数用来匹配包括'.'和`*`的正则表达式。模式中的字符'.'表示任意一个字符，而`*`表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配
- 解答方法：
  - 方法一：
    - 思路：利用js的test函数求解
    - 代码：

```
//s, pattern都是字符串
function match(s, pattern)
{
    let reg = new RegExp("^"+pattern+"$","g")
    return reg.test(s)
}
```

三、递归和循环

1.斐波那契数列

- 题目描述：大家都知道斐波那契数列，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（从0开始，第0项为0）。n<=39
- 解答方法：
  - 方法一——牛客网无法通过
    - 思路：递归计算，但是牛客无法通过。这种基于递归的解法虽然直观但时间效率很低，在实际软件开发中不会用这种方法。
    - 代码：

```
function Fibonacci(n)
{
    if(n===0){
        return 0
    }
    if(n===1){
        return 1
    }
    return Fibonacci(n-1)+Fibonacci(n-2)
}
```

  - 方法二——把递归用循环实现

- 思路：把递归用循环实现，极大地提高了时间效率
- 代码：

```
function Fibonacci(n){
    if(n<=1) {
        return n
    }
    let f0 = 0,
        f1 = 1,
        f2
    for(var i=2;i<=n;i++){
        f2=f0+f1
        f0=f1
        f1=f2
    }
    return f2
}
```

2.跳台阶——动态规划

- 题目描述：一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。
- 解答方法：
  - 方法一
    - 思路：当N=1时有一种跳法，当N=2时有两种跳法，当N=3时有三种跳法，当N=4有五种跳法，当N=5时有八种跳法，当N=6时有十三种跳法..... 这个规律符合斐波那契数列
    - 代码：

```
function jumpFloor(number){
    if(number <= 1){
        return number
    }
    let arr = [1,1]
    for(let i=2;i<=number;i++){
        arr[i] = arr[i-1]+arr[i-2]
    }
    return arr[number]
}
```

    - 注意：这道题考察的其实就是斐波那契数列，以下代码和上述效果一样

```
function jumpFloor(number){
    if(number<=1) {
        return number
    }
    let f0 = 1,
        f1 = 1,
        f2
    for(var i=2;i<=number;i++){
        f2=f0+f1
        f0=f1
```

```
        f1=f2
    }
    return f2
}
```

3.变态跳台阶

- 题目描述：一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。
- 解答方法：
  - 方法一：
    - 思路：当N=1时，有1种跳法，当N=2时，有2种跳法，当N=3时，有4种跳法，当N=4，有8种跳法，当N=5时有16种跳法..... 这个规律显而易见
    - 代码：

```
function jumpFloorII(number)
{
    // write code here
    return Math.pow(2, number - 1)
}
```

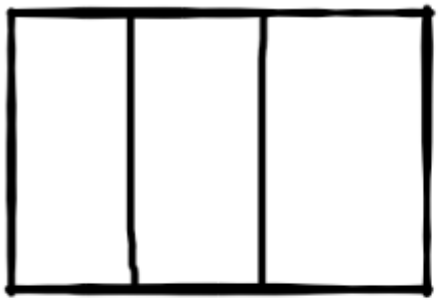
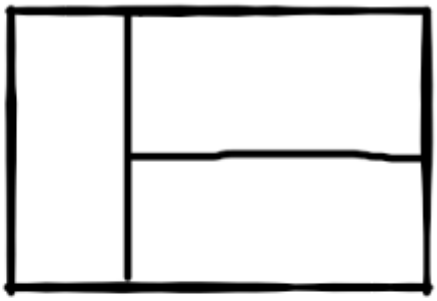
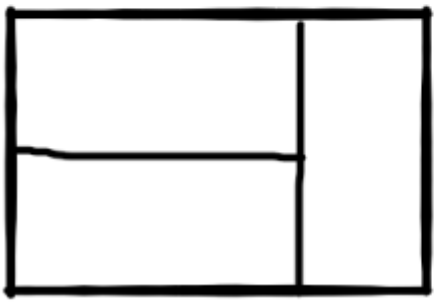
    - 另一种递归写法：

```
function jumpFloorII(number)
{
    if (number === 1){
        return 1
    }
    return 2*jumpFloorII(number-1)
}
```

4.矩形覆盖

- 题目描述：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2x1的小矩形无重叠地覆盖一个2xn的大矩形，总共有多少种方法？

  比如n=3时，2*3的矩形块有3种覆盖方法：

- 解答方法：
  - 方法一：
    - 思路：当N=1时，有1种方法，当N=2时，有2种方法，当N=3时，有3种方法，当N=4时，有5种方法，当N=5时，有8种跳法，当N=6时有12种方法..... 这个规律符合斐波那契数列
    - 代码：

```
function rectCover(number)
{
    if(number <= 2){
        return number
    }
    let f1 = 1,
        f2 = 2,
        f3
    for(let i=3;i<=number;i++){
        f3 = f1 + f2
        f1 = f2
        f2 = f3
    }
    return f3
}
```

5.数值的整数次方

- 题目描述：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。
- 解答方法：
  - 方法一：
    - 思路：将exponent个base相乘即可

        - 代码：

```
function Power(base, exponent)
{
    let val = 1
    let flag
    flag = exponent>=0 ? 1:-1,
    exponent = Math.abs(exponent)
    for(let i=1;i<=exponent;i++){
        val = val*base
    }
    if(flag === 1){
        return val
    }else{
        return 1/val
    }
}
```

        - 另一种形式：

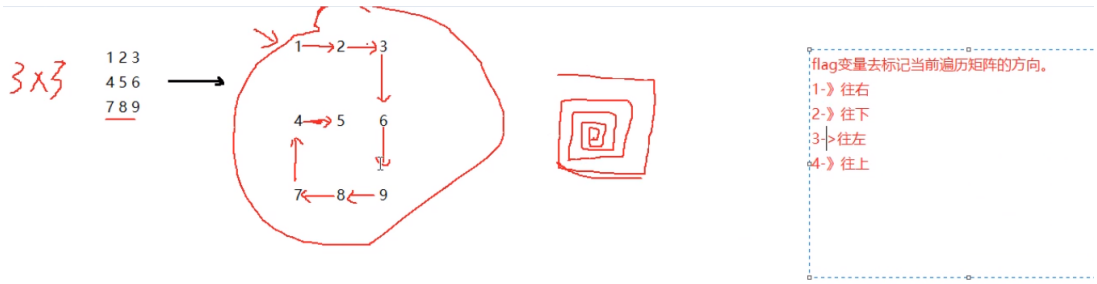
```
function Power(base, exponent)
{
    let val = 1
    for(let i=1;i<=Math.abs(exponent);i++){
        val = val*base
    }
    if(exponent<0){
        return 1/val
    }else{
        return val
    }
}
```

    - 方法二：
        - 思路：直接使用Math.pow函数返回基数（base）的指数（exponent）次幂
        - 代码：

```
function Power(base, exponent)
{
    return Math.pow(base,exponent)
}
```

6.顺时针打印矩阵

- 题目描述：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，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.
- 解答方法：
    - 方法一：
        - 思路：通过一个flag变量去不断更新遍历矩阵下标x,y的值（通过越界和当前的值）
            - 图示：



- 代码：

```
function printMatrix(matrix)
{
    const ans = []
    let flag = 1 //1->往右,2->往下,3->往左,4->往上
    let x = 0
    let y = 0
    //vis二维数组用来标记走过的点
    let vis = new Array(matrix.length)
    for(let i=0;i<matrix.length;i++){
        vis[i] = new Array(matrix[0].length)
    }

    while(ans.length < matrix.length * matrix[0].length){
        if(x<0 || x>=matrix.length || y<0 || y>=matrix[0].length || vis[x][y]){
//vis[x][y] -> 已经遍历过的位置也当作越界处理
            if(flag === 1){
                flag = 2  //往下走
                y-- //消除越界的影响
                x++ //本质上就是到达下一个位置的横坐标
            }else if(flag === 2){
                flag = 3  //往左走
                x-- //消除越界的影响
                y-- //本质上就是到达下一个位置的纵坐标
            }else if(flag === 3){
                flag = 4  //往上走
                y++ //消除越界的影响
                x-- //本质上就是到达下一个位置的横坐标
            }else{
                flag = 1  //往右走
                x++ //消除越界的影响
                y++ //本质上就是到达下一个位置的纵坐标
            }
        }else{
            ans.push(matrix[x][y])
            vis[x][y] = true  //去标记已经遍历过的位置
            //根据flag的值更新遍历矩阵的下标x,y的值
            if(flag === 1){
                y++
            }else if(flag === 2){
                x++
            }else if(flag === 3){
                y--
            }else{
```

```
                x--
            }
        }
    }
    return ans

}
```

- 方法二：
  - 思路：先右，再下，再左，再上遍历，然后范围变小，继续遍历
  - 代码：

```
function printMatrix(matrix)
{
    // write code here
    var row=matrix.length
    var col=matrix[0].length
    var res=[]
    if(row==0||col==0){
        return res
    }
    var left=0,
        top=0,
        right=col-1,
        bottom=row-1
    while(left<=right&&top<=bottom){
        for(let i=left;i<=right;i++)
            res.push(matrix[top][i])
        for(let i=top+1;i<=bottom;i++)
            res.push(matrix[i][right])
        if(top!=bottom)
            for(let i=right-1;i>=left;i--)
                res.push(matrix[bottom][i])
        if(left!=right)
            for(let i=bottom-1;i>top;i--)
                res.push(matrix[i][left])
        left++,top++,right--,bottom--
    }
    return res
}
```

7.最小的K个数

- 题目描述：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。
- 解答方法：
  - 方法一：时间复杂度最好O(N*logN)
    - 思路：先对数组从小到大排序，然后输出指定个数的最小值即可
    - 代码：

```
function GetLeastNumbers_Solution(input, k)
{
    if(input.length<k){
        return []
    }
    input.sort((x,y)=>{
        return x-y
    })
    let arr = []
    for(let i=0;i<k;i++){
        arr.push(input[i])
    }
    return arr
}
```

- 更简洁一点：

```
function GetLeastNumbers_Solution(input, k)
{
    if(input.length<k){
        return []
    }
    input.sort((x,y)=>{
        return x-y
    })

    return input.slice(0,k)
}
```

- 方法二：时间复杂度O(N)——不考虑最小K个数的顺序时，可使用此方法
  - 思路：二分思想+快排partition
  - 代码：

```
function GetLeastNumbers_Solution(arr, k)
{
    if(arr===null || arr.length===0) return []
    if(k > arr.length) return []
    let lo = 0, hi = arr.length-1
    while(lo < hi){
        let index = partition(arr,lo,hi)
        if(index===k-1) break
        else if(index<k-1) lo = index + 1
        else hi = index - 1
    }
    return arr.slice(0,k)
}
function partition(arr,lo,hi){
    let pivot = arr[lo]
    let index = lo
    for(let i=lo;i<=hi;i++){
        if(arr[i]<pivot) swap(arr,i,++index)
    }
    swap(arr,lo,index)
```

```
    return index
}
function swap(arr,i,j){
    let temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

8.整数中1出现的次数（从1到n整数中1出现的次数）

- 题目描述：求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数（从1 到 n 中1出现的次数）。
- 解答方法：
  - 方法一：
    - 思路：先构造一个判断某个数字包含多少个1的函数，然后遍历数字，累加求和即可
    - 代码：

```
function NumberOf1Between1AndN_Solution(n){
    if(n===1){
        return 1
    }
    let num = 0
    for(let i=1;i<=n;i++){
        num = num + isOne(i)
    }
    return num
}
function isOne(number){
    let bit
    let total = 0
    while(number>=1){
        bit = number%10
        if(bit === 1){
            total++
        }
        number = Math.floor(number/10)
    }
    return total
}
```

    - 类似：

```
function NumberOf1Between1AndN_Solution(n)
{
    // write code here
    let counts,num
    counts = 0
    if(n < 1){
        return 0
    }
```

```
    for(let i = 1;i <= n;i++){
        num = i
        while(num > 0){
            if(num%10 == 1){
                counts++
            }
            num = Math.floor(num/10)
        }
    }
    return counts
}
```

9.连续子数组的最大和——动态规划——！！！！！

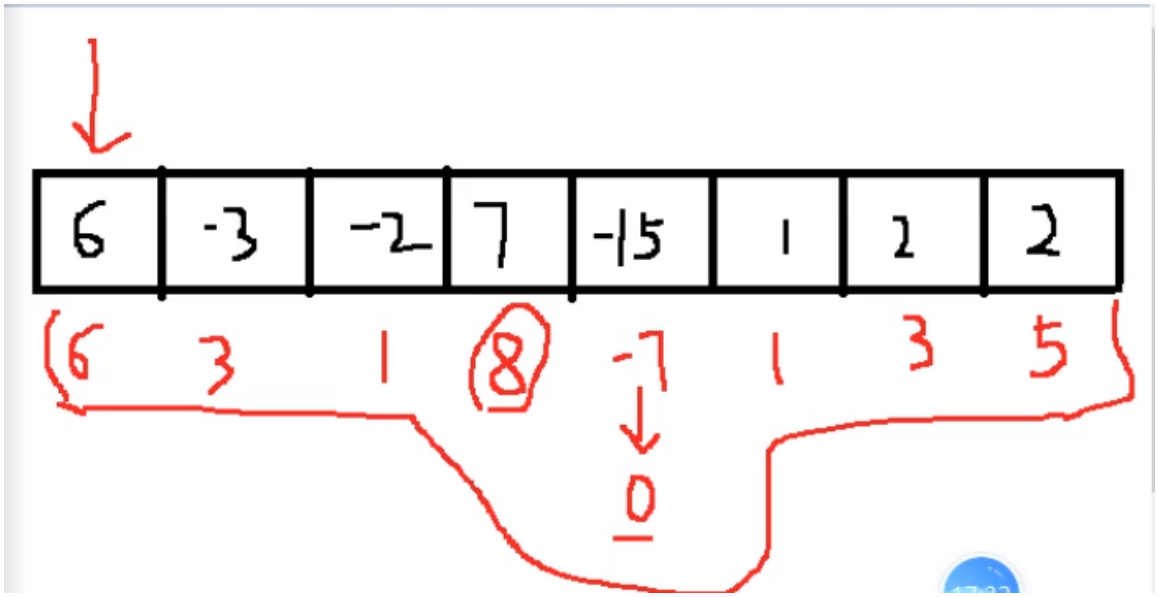
- 题目描述：HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)
- 解答方法：
  - 方法一——自己想的
    - 思路：两层遍历循环，从第一个数开始，依次求出这个数后面的数的总和，得出最大值。从第二个数开始，依次求出第二个数后面的数的总和，得出最大值。.....遍历完，即可得出最大连续子序列的和！
    - 代码：

```
function FindGreatestSumOfSubArray(array)
{
    const len = array.length
    if(len<=0){
        return 0
    }
    let temp = -Infinity
    let sum
    for(let i=0;i<len-1;i++){
        sum = array[i]
        if(temp<sum){
            temp = sum
        }
        for(let j=i+1;j<len;j++){
            sum = sum+array[j]
            if(temp<sum){
                temp = sum
            }
        }
    }
    return temp
}
```

  - 方法二——不需要动态规划

- 思路：通过sum变量去统计当前连续子序列的和，统计完后，更新Max的值，最后判断是否更新sum的值

  图示：



- 代码：

```
function FindGreatestSumOfSubArray(array)
{
    let sum = 0
    let Max = array[0]  //连续子数组最大和
    for(let i=0;i<array.length;i++){
        sum += array[i]
        Max = Math.max(Max,sum)
        if(sum < 0){
            sum = 0
        }
    }
    return Max
}
```

- 扩展——求出最大的连续子序列
  - 思路：新建一个数组存储连续子序列
  - 代码：

```
function FindGreatestSumOfSubArray(array)
{
    let sum = 0
    let max = array[0]
    let temp = [] //暂存子序列
    let ans

    for(let i=0;i<array.length;i++){
        sum = sum + array[i]
        temp.push(array[i])
        if(max < sum){
            max = sum
```

```
            ans = [...temp] //将最大子序列赋值给ans
        }

        if(sum<0){
            sum = 0
            temp = [] //清空
        }
    }
    return [max,ans]
}
let ret = FindGreatestSumOfSubArray([6,-3,-2,7,-15,1,2,2])
console.log(ret)
```

- 方法三——动态规划
  - 思路：使用动态规划，F(i):以array[i]为末尾元素的子数组的和的最大值，子数组的元素的相对位置不变；F(i)=max(F(i-1)+array[i] , array[i])；max:所有子数组的和的最大值
  - 代码：

```
function FindGreatestSumOfSubArray(array)
{
    let dp = []
    dp[0] = array[0]
    let max = array[0]
    for(let i=1;i<array.length;i++){
        dp[i] =Math.max(dp[i-1]+array[i],array[i])
        max = Math.max(max,dp[i])
    }
    return max
}
```

  - 另一种写法：

```
function FindGreatestSumOfSubArray(array)
{
    let dp  =array[0] //包含array[i]的连续数组最大值
    let max = array[0] //记录当前所有子数组的和的最大值
    for (let i = 1; i < array.length; i++) {
        dp = Math.max(dp+array[i], array[i])
        max = Math.max(dp, max)
    }
    return max
}
```

  - 扩展：
    - 思路：使用动态规划，新建一个数组存储连续子序列
    - 代码：

```
function FindGreatestSumOfSubArray(array)
{
    let dp  =array[0] //包含array[i]的连续数组最大值
    let max = array[0] //记录当前所有子数组的和的最大值
```

```
    let arr = [array[0]] ////暂存子序列
    let ans
    for (let i = 1; i < array.length; i++) {
        if(dp+array[i] < array[i]){
            arr = []  //清空
        }
        arr.push(array[i])
        dp = Math.max(dp+array[i], array[i])
        if(max < dp){
            max = dp
            ans = [...arr] //将最大子序列赋值给ans
        }
    }
    return [max,ans]
}
let ret = FindGreatestSumOfSubArray([6,-3,-2,7,-15,1,2,2])
console.log(ret)
```

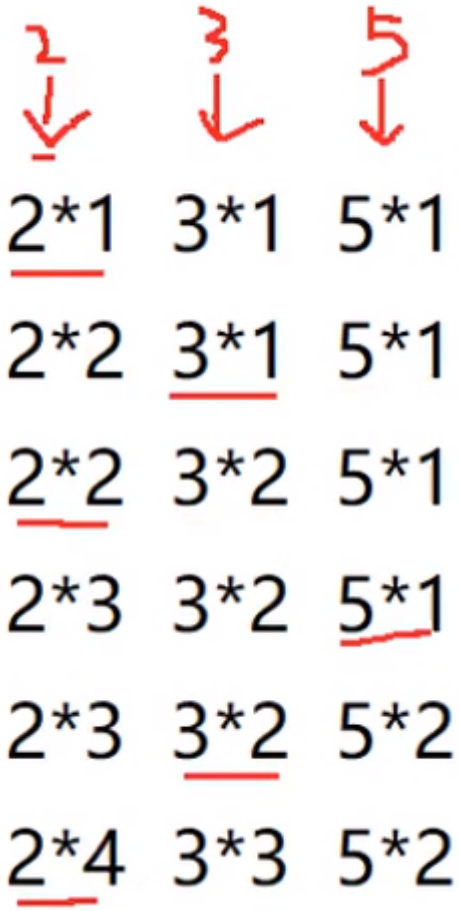
10.丑数——动态规划

- 题目描述：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。
- 解答方法：
  - 方法一——牛客网无法通过，时间复杂度过大
    - 思路：根据定义，将给定的数不断除 2、3、5，看看能不能除尽即可。
    - 代码：

```
// 判断是否符合丑数定义
function isUgly(number) {
    while (number % 2 === 0) {
        number /= 2;
    }
    while (number % 3 === 0) {
        number /= 3;
    }
    while (number % 5 === 0) {
        number /= 5;
    }
    return number === 1;
}
function GetUglyNumber_Solution(index)
{
    if(index <= 0){
        return 0
    }
    let number = 0,
        uglyFound = 0
    while(uglyFound < index){
        number++
        if(isUgly(number)){
            uglyFound++
```

```
        }
    }
    return number
}
```

- 方法二——动态规划
  - 思路：前面速度慢是因为计算了太多非丑数。根据丑数定义，每一个丑数都是根据前面一个丑数乘以 2、3 或者 5 得到的。在确保顺序的情况下，逐步计算即可。
    - 图示：



  - 代码：

```
function GetUglyNumber_Solution(index)
{
    if(index<=0){
        return 0
    }
    const uglyNum = [1] //存放丑数
    let pointer2 = 0, //遍历丑数*2的队列
        pointer3 = 0, //遍历丑数*3的队列
        pointer5 = 0  //遍历丑数*5的队列
    for(let i=1;i<index;i++){
        // 找出下一个丑数，确保顺序
        uglyNum[i] = Math.min(uglyNum[pointer2] * 2, uglyNum[pointer3] * 3,
uglyNum[pointer5] * 5)
        // 如果结果相同，移动指针，防止下次重复计算
```

```
        if (uglyNum[i] === uglyNum[pointer2] * 2) ++pointer2
        if (uglyNum[i] === uglyNum[pointer3] * 3) ++pointer3
        if (uglyNum[i] === uglyNum[pointer5] * 5) ++pointer5
    }
    return uglyNum[index - 1]
}
```

- 时间复杂度是O(N)，空间复杂度是O(N)

11.和为S的连续正数序列——滑动窗口

- 题目描述：小明很喜欢数学,有一天他在做数学作业时,要求计算出9~16的和,他马上就写出了正确答案是100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为100(至少包括两个数)。没多久,他就得到另一组连续正数和为100的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为S的连续正数序列? Good Luck!
- 解答方法：
  - 方法一——自己的想法
    - 思路：多层遍历，外层循环从1开始，以S/2的为终点；内层循环从当前数字开始，以S/2的为终点；以当前数字连续向后累加，如果和刚好为S，则保存此时的连续正数序列。如果和大于S，则退出内层循环，开始下一轮循环查找。最后将保存的连续正数序列输出即可
    - 代码：

```
function FindContinuousSequence(sum)
{
    if(sum === 1) return []
    let num = Math.ceil(sum/2) //右侧最大值
    let total = 0
    let ans = []  //保存符合条件的所有序列
    let temp = []  //保存符合条件的当前序列
    for(let i=1;i<=num;i++){
        total = 0
        for(let j=i;j<=num;j++){
            total += j
            temp.push(j)
            if(total===sum){
                ans.push([...temp])
            }else if(total>sum){
                temp = []
                break
            }
        }
    }
    return ans
}
```

    - 缺点：代码的时间复杂度过高，虽然牛客网可以通过，但是不推荐
  - 方法二——滑动窗口

- 思路：题目求的是连续正数序列，而且至少含有两个数，那么我们可以从1,2这两个数开始。以求和为9的所有连续序列为例，假设两个指针left和right，分别指向正数序列的首尾，curSum表示序列之和，一开始left=1，right=2，curSum=3<9，序列需要包含更多的数，于是right+1，此curSum=6，依旧小于9，于是right+1，此时curSum=10，大于9，序列需要删除一些数，于是left+1，curSum=9，找到第一个满足条件的序列；接着right+1，按照前面的方法继续查找满足条件的序列，直到right等于Math.ceil(sum/2)
- 代码：

```
function FindContinuousSequence(sum) {
    if(sum<3){
        return []
    }
    let result = []
    let left = 1
    let right = 2
    let curSum = left + right
    while(left<right && right <= Math.ceil(sum/2)){
        while(curSum>sum){
            curSum -= left
            left++
        }
        if(curSum === sum){
            result.push(createSequence(left,right))
        }
        right++
        curSum+=right
    }
    return result
}
function createSequence(left,right){
    let temp = []
    for(let i=left;i<=right;i++){
        temp.push(i)
    }
    return temp
}
```

- 另一种方式初始化temp数组可以直接存储连续子序列

```
function FindContinuousSequence(sum) {
    if(sum<3){
        return []
    }
    let result = []
    let left = 1
    let right = 2
    let temp = [left,right]
    let curSum = left + right
    while(left<right && right <= Math.ceil(sum/2)){
        while(curSum>sum){
            temp.shift()
            curSum -= left
            left++
```

```
        }
        if(curSum === sum){
            result.push([...temp])
        }
        right++
        temp.push(right)
        curSum+=right
    }
    return result
}
```

- 可能这种写法更容易理解（while里面只有if）——标准写法：

```
function FindContinuousSequence(sum) {
    if (sum < 3) {
        return []
    }
    let result = []
    let left = 1
    let right = 2
    let curSum = left + right
    while (left < right && right <= Math.ceil(sum / 2)) {
        if (curSum > sum) {
            curSum -= left
            left++
        } else if (curSum === sum) {
            result.push(createSequence(left, right))
            right++
            curSum += right
        } else{
            right++
            curSum += right
        }
    }
    return result
}
function createSequence(left,right){
    let temp = []
    for(let i=left;i<=right;i++){
        temp.push(i)
    }
    return temp
}
```

12.和为S的两个数字——滑动窗口

- 题目描述：输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。
- 解答方法：
    - 方法一——滑动窗口

- 思路：左右夹逼法！！！只需要2个指针！ left开头，right指向结尾。如果和小于sum，说明太小了，left右移寻找更大的数。如果和大于sum，说明太大了，right左移寻找更小的数。和相等，把left和right的数返回
- 代码：（自己想的，还可以优化，其实最外层的乘积最小，下次千万别被题目误导了）

```
function FindNumbersWithSum(array, sum)
{
    if(array.length === 1){
        return false
    }
    let len = array.length
    let left = 0
    let right = len-1
    let curSum = array[left]+array[right]
    let min = Infinity
    let temp = []
    while(left<right){
        while(curSum < sum){
            left++
            curSum = array[left]+array[right]
        }
        if(curSum === sum){
            if(min > array[left]*array[right]){
                min = array[left]*array[right]
                temp = [array[left],array[right]]
            }
            left++
            curSum = array[left]+array[right]
        }
        while(curSum > sum){
            right--
            curSum = array[left]+array[right]
        }

    }
    return temp
}
```

- 优化代码后——标准写法：

```
function FindNumbersWithSum(array, sum)
{
    // write code here
    if(array.length < 2)
        return []
    var left = 0,
        right = array.length-1
    while(left < right){
        if(array[left]+array[right] < sum){
            left++
        }else if(array[left]+array[right] > sum){
            right--
        }else{
```

```
                return [array[left],array[right]]//外层最先找到的两个数，乘积必然是最小的
            }
        }
        return []
    }
```

13.扑克牌顺子

- 题目描述：LL今天心情特别好,因为他去买了一副扑克牌,发现里面居然有2个大王,2个小王(一副牌原本是54张^_^)...他随机从中抽出了5张牌,想测测自己的手气,看看能不能抽到顺子,如果抽到的话,他决定去买体育彩票,嘿嘿！！“红心A,黑桃3,小王,大王,方片5”,“Oh My God!”不是顺子.....LL不高兴了,他想了想,决定大\小 王可以看成任何数字,并且A看作1,J为11,Q为12,K为13。上面的5张牌就可以变成“1,2,3,4,5”(大小王分别看作2和4),“So Lucky!”。LL决定去买体育彩票啦。 现在,要求你使用这幅牌模拟上面的过程,然后告诉我们LL的运气如何， 如果牌能组成顺子就输出true，否则就输出false。为了方便起见,你可以认为大小王是0。
- 解答方法：
  - 方法一：
    - 思路：先统计0的个数sum，然后对数组从小到大排序，接着用sum个0去填充（相邻位置不连续的），如果剩余的0的个数为负数，则返回false；否则返回true
    - 代码：

```
function IsContinuous(numbers)
{
    if(numbers.length === 0){
        return false
    }
    let sum = 0
    for(let x of numbers){  //统计数组里 数字0 的个数
        if(x===0){
            sum++
        }
    }
    numbers.sort((x,y)=>{   //数组排序
        return x-y
    })
    for(let i=sum+1;i<numbers.length;i++){
        sum -= numbers[i]-numbers[i-1]-1
        if(sum<0 || numbers[i]===numbers[i-1]){
            return false
        }
    }
    return true

}
```

  - 方法二：
    - 思路：
      1. max - min < 5
      2. 除0外，没有重复的数
    - 代码：

```
function IsContinuous(numbers)
{
    if(numbers.length!==5) return false
    let arr = numbers.filter(x => {
        return x>0
    })
    let min = Math.min(...arr)
    let max = Math.max(...arr)
    if(max-min>=5) return false
    for(let i=0;i<arr.length;i++){
        if(arr.indexOf(arr[i]) !== i){
            return false
        }
    }
    return true
}
```

14.求1+2+3+...+n

- 题目描述：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）。
- 解答方法：
  - 方法一：
    - 思路：使用递归
    - 代码：

```
function Sum_Solution(n)
{
    if(n===1){
        return 1
    }
    return n + Sum_Solution(n-1)
}
```

    - 改进：去掉if判断

```
function Sum_Solution(n)
{
  let ans = n;
    ans && (ans += Sum_Solution(n-1))
    return ans
}
```

  - 方法二：
    - 思路：使用reduce方法。reduce() 方法对数组中的每个元素执行一个由您提供的reducer函数(升序执行)，将其结果汇总为单个返回值。
    - 代码：

```
function Sum_Solution(n)
{
    var arr = new Array(n).fill(1) //fill() 方法用一个固定值填充一个数组中从起始索引到终止索引内的全部元素。不包括终止索引。
    var newArray = arr.map(function(item,index){ //map返回一个新数组
        return item+index
    })
    var result = newArray.reduce(function(pre,next){ //reduce() 方法对数组中的每个元素执行一个由您提供的reducer函数(升序执行)，将其结果汇总为单个返回值
        return pre+next
    })
    return result
}
```

15.不用加减乘除做加法

- 题目描述：写一个函数，求两个整数之和，要求在函数体内不得使用+、-、*、/四则运算符号。
- 解答方法：
  - 方法：
    - 思路：
      - 第一步：我们试想，对于二进制相加，不考虑进位的话，如下：0+1=1，1+0=1；1+1=0，0+0=0；可以观察到，这和异或的结果是一样的
      - 我们再来考虑进位的过程，上面的过程，只有1+1会产生进位，此时我们可以想象成两个数先做位与运算，然后再向左移动一位，只有两个数都是1的时候，位与得到的结果是1，其余都是0
      - 相加，重复前两步，直到不产生进位为止
    - 代码：

```
function Add(num1, num2){
    //将加法分为进位的部分和不进位的部分，和为两部分之和
    while(num2 !== 0){
        let temp = num1 ^ num2  //不用进位的部分
        num2 = (num1&num2)<<1  //进位的部分
        num1 = temp
    }
    return num1
}
```

16.孩子们的游戏(圆圈中最后剩下的数)

- 题目描述：每年六一儿童节,牛客都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF作为牛客的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的:首先,让小朋友们围成一个大圈。然后,他随机指定一个数m,让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,从他的下一个小朋友开始,继续0...m-1报数....这样下去....直到剩下最后一个小朋友,可以不用表演,并且拿到牛客名贵的“名侦探柯南”典藏版(名额有限哦!!^_^)。请你试着想下,哪个小朋友会得到这份礼品呢？(注：小朋友的编号是从0到n-1)

  如果没有小朋友，请返回-1
- 解答方法：
  - 方法一：

- 思路：假设当前删除的位置是del ,下一个删除的数字的位置是 del + m 。但是，由于把当前位置的数字删除了，后面的数字会前移一位，所以实际的下一个位置是 del + m - 1。由于存在m>arr.length的情况，所以最后取余一下，就是 (del + m - 1) % arr.length。
  - 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。
- 代码：

```
function LastRemaining_Solution(n, m){
    if(n===0){
        return -1
    }
    //快速生成0到指定数的数组
    let arr = new Array(n).fill(0).map((item,index)=>{
        return item+index
    })
    let head = 0
    while(arr.length > 1){
        //构造新数组
        let len = arr.length
        head = (head + m - 1) % len
        arr.splice(head,1)
    }
    return arr[0]
}
```

- 方法二——递归
  - 思路：当n=1时，留下的肯定是序号为0的人。可以得出递推公式：F(n)=(F(n-1)+m)%n。这里是求保留下来的人，而第一种方法是求删除的人，所以这里是m而不是m-1。
  - 代码：

```
function LastRemaining_Solution(n, m)
{
    if(n===0) return -1
    // 旧编号 0    1    ...    m-1    m    m+1    ...    n-1
    // 新编号                          0    1
    // 即旧编号规律为：  新编号+m
    if(n===1) return 0
    return (LastRemaining_Solution(n-1,m)+m)%n
}
```

  - 循环的写法

```
function LastRemaining_Solution(n, m){
    if(n===0){
        return -1
    }
    let last = 0
    for(let i=2;i<=n;i++){
        last = (last + m)%i
    }
    return last
}
```

17.矩阵中的路径——回溯法

- 题目描述：

  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 $\begin{bmatrix} a & b & c & e \\ s & f & c & s \\ a & d & e & e \end{bmatrix}$ 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。

- 解答方法：
  - 方法：
    - 思路：回溯
      - 根据给定数组，初始化一个标志位数组，初始化为false，表示未走过，true表示已经走过，不能走第二次
      - 根据行数和列数，遍历数组，先找到一个与str字符串的第一个元素相匹配的矩阵元素，进入judge
      - 根据i和j先确定一维数组的位置，因为给定的matrix是一个一维数组
      - 确定递归终止条件：越界，当前找到的矩阵值不等于数组对应位置的值，已经走过的，这三类情况，都直接false，说明这条路不通
      - 若k，就是待判定的字符串str的索引已经判断到了最后一位，此时说明是匹配成功的
      - 下面就是本题的精髓，递归不断地寻找周围四个格子是否符合条件，只要有一个格子符合条件，就继续再找这个符合条件的格子的四周是否存在符合条件的格子，直到k到达末尾或者不满足递归条件就停止。
      - 走到这一步，说明本次是不成功的，我们要还原一下标志位数组index处的标志位，进入下一轮的判断
    - 代码：

```
function hasPath(matrix, rows, cols, str) {
    let flag = new Array(matrix.length).fill(null);
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            //循环遍历二维数组，找到起点等于str第一个元素的值，再递归判断四周是否有符合条件的--回溯法
            if (judge(matrix, i, j, rows, cols, flag, str, 0)) {
                return true
            }
        }
    }
    return false
}

function judge(matrix, i, j, rows, cols, flag, str, k) {
    //先根据i和j计算匹配的第一个元素转为一维数组的位置
    let index = i * cols + j;
    //递归终止条件
    if (i < 0 || j < 0 || i >= rows || j >= cols || matrix[index] != str[k] || flag[index] == true) {
```

```
        return false
    }
    //若k已经到达str末尾了，说明之前的都已经匹配成功了，直接返回true即可
    if (k == str.length - 1) {
        return true
    }
    //要走的第一个位置为true，表示已经走过了
    flag[index] = true;
    //回溯，递归寻找。每次找到了就给k加一，找不到，还原
    if (judge(matrix, i - 1, j, rows, cols, flag, str, k + 1) ||
        judge(matrix, i + 1, j, rows, cols, flag, str, k + 1) ||
        judge(matrix, i, j - 1, rows, cols, flag, str, k + 1) ||
        judge(matrix, i, j + 1, rows, cols, flag, str, k + 1)) {
        return true
    }
    //走到这，说明这一条路不通，还原，再试其他的路径
    flag[index] = false;
    return false;
}
```

18.机器人的运动范围——回溯法

- 题目描述：地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子？
- 解答方法：
  - 方法一——回溯法（推荐）
    - 思路：
      - 从(0,0)开始走，探索时，判断当前节点是否可达的标准为：
        - 当前节点在矩阵内
        - 当前节点未被访问过
        - 当前节点满足limit限制
      - 每成功走一步标记当前位置为true，然后从当前位置往四个方向探索
    - 代码：

```
let sum
function movingCount(threshold, rows, cols){
    sum= 0  //牛客提交时，必须在这个函数里初始化为0
    let vis = new Array(rows*cols).fill(null)
    slove(0,0,rows,cols,threshold,vis)
    return sum
}
function slove(i,j,rows,cols,threshold,vis){
    if(i<0||j<0||i>=rows||j>=cols||vis[i*cols+j]===true||(cul(i)+cul(j))>threshold){
        return
    }
    //当前位置(i,j)可走，那么就从当前位置往四个方向移动即可
    vis[i*cols+j] = true
    sum++
```

```
    slove(i+1,j,rows,cols,threshold,vis)
    slove(i-1,j,rows,cols,threshold,vis)
    slove(i,j+1,rows,cols,threshold,vis)
    slove(i,j-1,rows,cols,threshold,vis)

}
//求位数和
function cul(x){
    let res = 0
    //第一种方法--求余
    // while(x!==0){
    //  res += x%10
    //  x = parseInt(x/10)
    // }
    //第二种方法--字符串操作
    let temp = x + ""
    for (let i = 0; i < temp.length; i++) {
        res += temp.charAt(i) / 1
    }
    return res
}
```

- 方法二——暴力破解法——不推荐
  - 思路：依次遍历方格中的每一格，判断每一个方格的下标数位之和是否小于指定值，如果小于指定值，再判断该方格是否可以由上下左右推出
  - 代码：

```
function movingCount(threshold, rows, cols){
    if(threshold < 0){
    return 0
    }
    let sum= 1
    let vis = new Array(rows*cols).fill(null)
    vis[0] = true
    for(let i=0;i<rows;i++){
        for(let j=0;j<cols;j++){
            if(cul(i)+cul(j) <= threshold){
                if(i-1>=0 && vis[(i-1)*cols+j]){
                    //下
                    sum++
                    vis[i*cols+j] = true
                }else if(i+1<rows && vis[(i+1)*cols+j]){
                    //上
                    sum++
                    vis[i*cols+j] = true
                }else if(j-1>=0 && vis[i*cols+j-1]){
                    //左
                    sum++
                    vis[i*cols+j] = true
                }else if(j+1<cols && vis[i*cols+j+1]){
                    //右
                    sum++
                    vis[i*cols+j] = true
```

```
                }
            }
        }
    }
    return sum
}
//求位数和
function cul(x){
    let res = 0
    //第一种方法--求余
    // while(x!==0){
    //  res += x%10
    //  x = parseInt(x/10)
    // }
    //第二种方法--字符串操作
    let temp = x + ""
    for (let i = 0; i < temp.length; i++) {
        res += temp.charAt(i) / 1
    }
    return res
}
```

- 补充：这种方法时间复杂度更高

19.剪绳子

- 题目描述：给你一根长度为n的绳子，请把绳子剪成整数长的m段（ m、n都是整数，n>1并且m>1 ），每段绳子的长度记为k[0],k[1],...,k[m]。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。
  - 输入描述：输入一个数n，意义见题面。（ 2 <= n <= 60 ）
  - 示例1：
    - 输入：8
    - 输出：18
- 解答方法：
  - 方法一——动态规划
    - 思路：
      - F(i)=F(i-j)+F(j)
    - 代码：

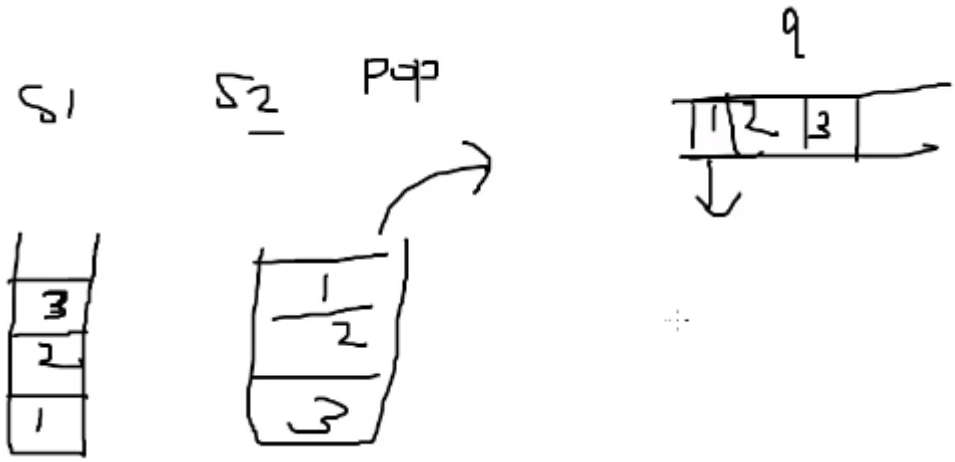
```
function cutRope(number){
    if(number === 2){
        return 1
    }
    if(number === 3){
        return 2
    }
    let dp = new Array(number+1).fill(0)
    dp[1] = 1
    dp[2] = 2
    dp[3] = 3
```

```
    let val = 0
    for(let i=4;i<=number;i++){
        for(let j=2;j<=parseInt(i/2);j++){
            dp[i] = Math.max(dp[i-j]*dp[j],dp[i])
        }
    }
    return dp[number]
}
```

四、栈和队列

1.用两个栈实现队列

- 题目描述：用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。
- 解答方法：
  - 思路：两个“先进后出”实现一个“先进先出”，两个栈，一个为入队栈stack1，一个为出队栈stack2。队列插入一个新的元素，压入stack1。队列弹出一个队列头元素，当stack2为空时，把stack1中的元素逐个压入stack2，此时stack2顶端元素就是队列头元素，可以直接弹出；当stack2不为空时，它的顶端元素可以直接弹出。
    - 示意图：



  - 代码：

```
let outStack = [] //出栈
let inStack = [] //入栈
function push(node)
{
    // write code here
    inStack.push(node)
}
function pop()
{
    // write code here
    if(!outStack.length){
        while(inStack.length){
```

```
            outStack.push(inStack.pop())
        }
    }
    return outStack.pop()

}
```

2.包含min函数的栈

- 题目描述：定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。
  - 注意：保证测试中不会当栈为空的时候，对栈调用pop()或者min()或者top()方法。
- 解答方法：
  - 方法一：——不推荐，时间复杂度不为O（1）
    - 思路：min方法时直接调用Math.min实现，但是这里时间复杂度不为O（1），所以不推荐该方法
    - 代码：

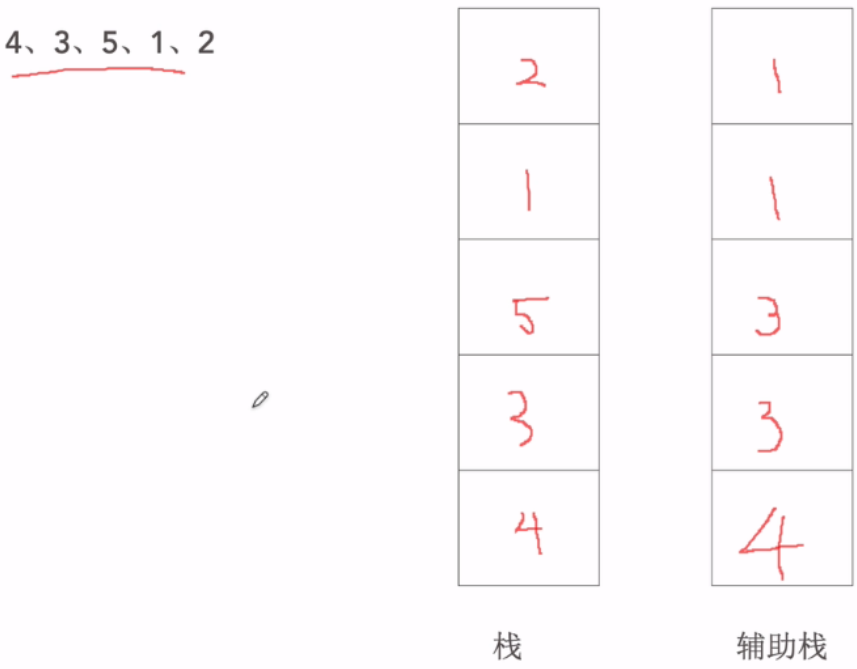
```
let stack = []
function push(node)
{
    // write code here
    stack.push(node)

}
function pop()
{
    // write code here
    if(stack.length>0){
        return stack.pop()
    }else{
         return null
    }
}
function top()
{
    // write code here
    if(stack.length>0){
        return stack[stack.length-1]
    }else{
        return null
    }

}
function min()
{
    // write code here
    if(stack.length>0){
        return Math.min(...stack)
        // 或者 return Math.min.apply(this,stack) 可以实现相同效果
    }else{
        return null
```

```
    }
}
```

- 方法二：——推荐
  - 思路：利用辅助栈，时间复杂度为O（1）
    - 对原栈和辅助栈的处理过程如下：
    - 元素压入原栈的时候，如果辅助栈为空，或者元素 <= 辅助栈的栈顶元素，那么将元素也压入辅助栈
      - 元素弹出原栈的时候，如果元素等于辅助栈的栈顶元素，辅助栈也弹出元素



  - 代码：

```
let stack = [] //定义一个栈
let minStack = [] //定义一个辅助栈
function push(node)
{
    if(stack.length===0){
        stack.push(node)
        minStack.push(node)
        return
    }
    let topVal = minStack[minStack.length-1]
    if(node>topVal){
        minStack.push(topVal)
    }else{
        minStack.push(node)
    }
    stack.push(node)
}
function pop()
{
    stack.pop()
```

```
    minStack.pop()
}
function top()
{
    return stack[minStack.length-1]
}
function min()
{
    return minStack[minStack.length-1]
}
```

3.栈的压入、弹出序列

- 题目描述：输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否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）

> 一开始都看不懂题目，后来才好像明白是什么个意思。假设有一串数字要将他们压栈: 1 2 3 4 5，可以先把1,2,3,4按顺序压入栈，然后将4出栈，接着将5入栈，然后依次出栈，最后结果就是4,5,3,2,1

- 解答方法：
  - 思路：借用一个辅助的栈，将原数列依次压入辅助栈，栈顶元素与所给的出栈队列相比，如果相同则出栈；如果不同则继续压栈，直到原数列中所有的数字压栈完毕；最后，检测辅助栈中是否为空， 若空，则该序列是压栈序列对应的一个弹出序列。否则，说明序列不是该栈的弹出序列。
  - 代码：

```
function IsPopOrder(pushV, popV)
{
    // write code here
    let stack = []  //定义一个辅助栈作用的数组
    let idx = 0 //弹出序列的下标初始值
    for(let i=0;i<pushV.length;i++){
        stack.push(pushV[i])
        while(stack.length && stack[stack.length-1] === popV[idx]){
            stack.pop() //当栈不为空，且栈顶元素等于弹出序列当前下标所指向的值时，出栈
            idx++  //弹出序列下标后移
        }
    }
    return stack.length === 0
}
```

五、链表

1.链表中环的入口结点

- 题目描述：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。
- 解答方法：

- 方法一——简单快速——有问题，`虽然牛客可以通过，但是如果存在多个节点的value值一样，这种方法也行不通`
  - 思路：遍历链表，如果其中包含环，则某个节点一定会出现两次
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function EntryNodeOfLoop(pHead)
{
    // write code here
    let node = pHead
    let map = {}
    while(node !== null){
        let val = node.val
        if(map[val] === undefined){
            map[val] = 1
            node = node.next
        }else{
            return node
        }
    }
    return null
}
```

- 方法二
  - 思路：数组法
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function EntryNodeOfLoop(pHead)
{
    // 数组法
    if(!pHead){
        return null
    }
    var arr = [],
        node = pHead
    while(node !== null){
        if(arr.indexOf(node) !== -1){
            return node
        }else{
            arr.push(node)
            node = node.next
        }
    }
    return null
}
```

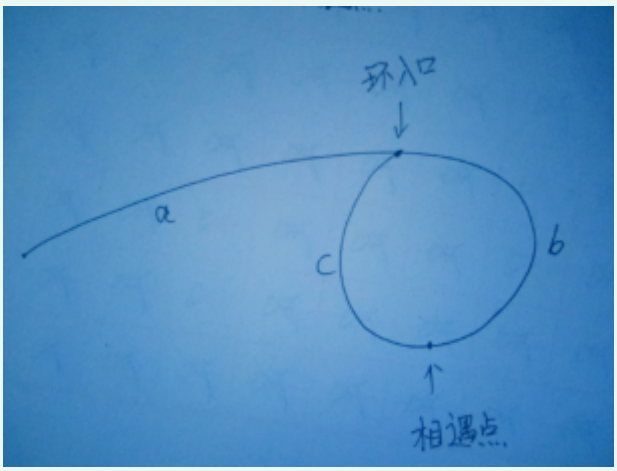
- 方法三
  - 思路：用Set对象，Set 对象类似于数组
  - 代码：

```
Set 对象类似于数组/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function EntryNodeOfLoop(pHead)
{
    // write code here
    let node = pHead
    let set = new Set()
    while(node !== null){
        if(!set.has(node)){
            set.add(node)
            node = node.next
        }else{
            return node
        }
    }
    return null
}
```

- 方法四
  - 思路：用Map对象
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function EntryNodeOfLoop(pHead)
{
    // write code here
    let node = pHead
    let map = new Map()
    while(node !== null){
        if(!map.has(node)){
            map.set(node,1)
            node = node.next
        }else{
            return node
        }
    }
    return null
}
```

- 方法五
  - 思路：——前四种方法类似，第四种用快慢指针

> 设置快慢指针，都从链表头出发，快指针每次**走两步**，慢指针一次**走一步**，假如有环，一定相遇于环中某点(结论1)。接着让两个指针分别从相遇点和链表头出发，两者都改为每次**走一步**，最终相遇于环入口(结论2)。以下是两个结论证明：
>
> 两个结论： 1、设置快慢指针，假如有环，他们最后一定相遇。 2、两个指针分别从链表头和相遇点继续出发，每次走一步，最后一定相遇与环入口。
>
> **证明结论1**：设置快慢指针fast和low，fast每次走两步，low每次走一步。假如有环，两者一定会相遇（因为low一旦进环，可看作fast在后面追赶low的过程，每次两者都接近一步，最后一定能追上）。
>
> **证明结论2：**
>
> 设：
>
> 链表头到环入口长度为--**a**
>
> 环入口到相遇点长度为--**b**
>
> 相遇点到环入口长度为--**c**
>
> 
>
>
> 则：相遇时
>
> **快指针路程=a+(b+c)k+b** ，k>=1 其中b+c为环的长度，k为绕环的圈数（k>=1,即最少一圈，不能是0圈，不然和慢指针走的一样长，矛盾）。
>
> **慢指针路程=a+b**
>
> 快指针走的路程是慢指针的两倍，所以：
>
> **（a+b）*2=a+(b+c)k+b**
>
> 化简可得：
>
> a=(k-1)(b+c)+c 这个式子的意思是： **链表头到环入口的距离=相遇点到环入口的距离+（k-1）圈环长度**。其中k>=1,所以**k-1>=0**圈。所以两个指针分别从链表头和相遇点出发，最后一定相遇于环入口。

- 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function EntryNodeOfLoop(pHead)
{
    // write code here
```

```
    let fast = pHead
    let slow = pHead
    while(fast && fast.next){
        fast = fast.next.next
        slow = slow.next
        if(fast === slow) break
    }
    if(!fast || !fast.next) return null
    fast = pHead
    while(fast!==slow){
        fast = fast.next
        slow = slow.next
    }
    return slow
}
```

2.从尾到头打印链表

- 题目描述：输入一个链表，按链表从尾到头的顺序返回一个ArrayList。
- 解答方法：
  - 思路：遍历链表，从数组头部插入遍历节点的值即可
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function printListFromTailToHead(head)
{
    // write code here
    let cur = head
    let list = []
    while(cur !== null){
        list.unshift(cur.val)
        cur = cur.next
    }
    return list
}
```

3.链表中倒数第k个结点

- 题目描述：输入一个链表，输出该链表中倒数第k个结点。
- 解答方法：
  - 思路：遍历链表，将遍历到的每个节点插入数组头部，最后返回数组里面k-1位置的值即可
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
```

```
function FindKthToTail(head, k)
{
    // write code here
    let cur = head
    let list = []
    while(cur != null){
        list.unshift(cur)
        cur = cur.next
    }
    return list[k-1]
}
```

或者(思路一样)：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function FindKthToTail(head, k)
{
    // write code here
    let cur = head
    let list = []
    while(cur != null){
        list.push(cur)
        cur = cur.next
    }
    return list[list.length-k]
}
```

4.反转链表

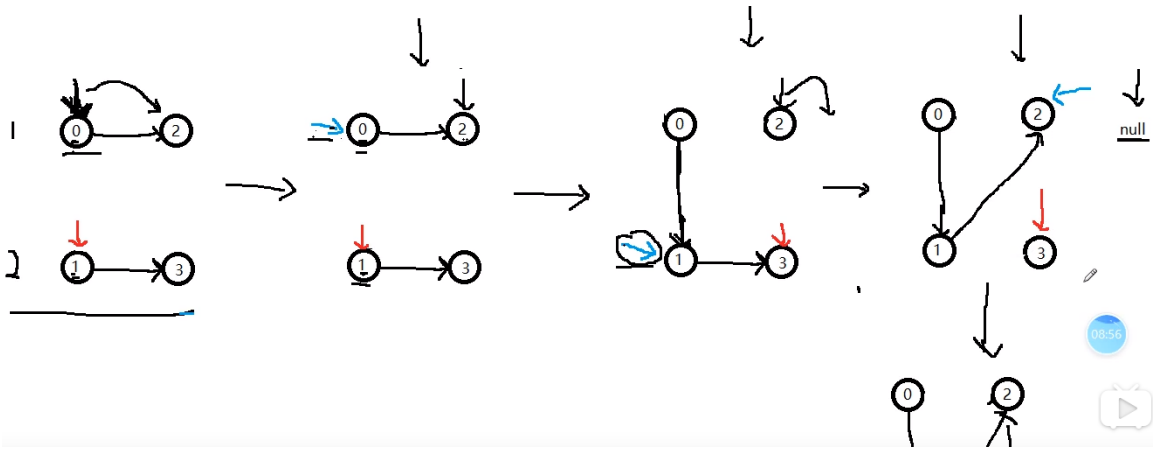
- 题目描述：输入一个链表，反转链表后，输出新链表的表头。
- 解答方法：
  - 思路：三个指针，p、head和q，每次让【head指向的节点】指向【p指向的节点】，然后遍历，即可完成
  - 代码：——时间复杂度O(n)，空间复杂度O(1)

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function ReverseList(pHead)
{
    // write code here
    let p = null
    let q
    while(pHead !== null){
        q = pHead
        pHead = q.next
        q.next = p
        p = q
    }
```

```
    pHead = p
    return pHead
}
```

5.合并两个排序的链表

- 题目描述：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。
- 解答方法：
  - 方法一：
    - 思路：不断去比较两个链表中节点的val值，然后取判断哪个节点优先需要合成到链表的尾部
    - 图示：



    - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function Merge(pHead1, pHead2)
{
    // write code here
    if(!pHead1){
        return pHead2
    }
    if(!pHead2){
        return pHead1
    }
    let headNode   //最终合成链表的头节点
    if(pHead1.val<pHead2.val){
        headNode = pHead1
        pHead1 = pHead1.next
    }else{
        headNode = pHead2
        pHead2 = pHead2.next
    }
    let removeNode = headNode //在当前位置就是合成链表的长度为1，此时的头节点和尾节点是一样的
    while(pHead1!==null && pHead2!==null){
        if(pHead1.val > pHead2.val){
```

```
            removeNode.next = pHead2 //将合成链表的尾部节点添加到链表2中当前所指向的节点
            removeNode = removeNode.next //去更新合成链表的尾部节点
            pHead2 = pHead2.next
        }else{
            removeNode.next = pHead1 //将合成链表的尾部节点添加到链表2中当前所指向的节点
            removeNode = removeNode.next //去更新合成链表的尾部节点
            pHead1 = pHead1.next
        }
    }
    //将剩余的链表1中的节点放入到合成链表中
    if(pHead1!==null){
        removeNode.next = pHead1
    }else{ //将剩余的链表2中的节点放入到合成链表中
        removeNode.next = pHead2
    }
    return headNode
}
```

- 方法二：（推荐）
  - 思路：和一类似，但是代码更简洁
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function Merge(pHead1, pHead2)
{
    // write code here
    //建立虚拟头节点
    let headNode = {}
    let curNode = headNode
    while(pHead1!==null && pHead2!==null){
        if(pHead1.val > pHead2.val){
            curNode.next = pHead2 //curNode指向更小的
            curNode = pHead2 //curNode起到串联作用
            pHead2 = pHead2.next
        }else{
            curNode.next = pHead1 //curNode指向更小的
            curNode = pHead1  //curNode起到串联作用
            pHead1 = pHead1.next
        }
    }
    //将剩余的链表1中的节点放入到合成链表中
    if(pHead1!==null){
        curNode.next = pHead1
    }else{ //将剩余的链表2中的节点放入到合成链表中
        curNode.next = pHead2
    }
    return headNode.next
}
```

- 方法三：（递归，代码简洁，但是可能更难懂一点）

- 思路：和前面思想类似，使用递归，代码更加简洁
- 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function Merge(pHead1, pHead2)
{
    // write code here
    if(pHead1 === null){
        return pHead2
    } else if (pHead2 === null) {
        return pHead1
    }
    if(pHead1.val > pHead2.val){
        pHead2.next = Merge(pHead1,pHead2.next)
        return pHead2
    } else {
        pHead1.next = Merge(pHead1.next,pHead2)
        return pHead1
    }
}
```

6.复杂链表的复制

- 题目描述：输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针random指向一个随机节点），请对此链表进行深拷贝，并返回拷贝后的头结点。（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）
- 解答方法：
  - 思路：
    - 递归思想：把大问题转换为若干小问题
    - 将复杂链表分为头结点和剩余结点两部分，剩余部分采用递归方法
  - 代码：

```
/*function RandomListNode(x){
    this.label = x;
    this.next = null;
    this.random = null;
}*/
function Clone(pHead)
{
    // write code here
    if(!pHead){
        return null
    }
    let head = new RandomListNode(pHead.label)
    head.random = pHead.random
    head.next = Clone(pHead.next)
    return head
}
```

7.两个链表的第一个公共结点

- 题目描述：输入两个链表，找出它们的第一个公共结点。（注意因为传入数据是链表，所以错误测试数据的提示是用其他方式显示的，保证传入数据是正确的）
- 解答方法：
  - 方法一：
    - 分析：从链表的定义可以看出，这两个链表是单向链表。如果两个单项链表有公共节点，那么这两个链表从某一节点开始，他们的next都指向同一个节点。但由于是单向链表的节点，每一个节点都只有一个next，因此从第一个公共节点开始，之后她们的所有节点都是重合的，不可能再出现分叉，所以两个有公共节点而部分重合的链表，其拓扑图像是一个Y，而不可能是X。
    - 思路：首先找到两个链表的长度len1和len2，然后确定哪条为长链表哪条为短链表，之后确定长度差，长的链表先走长度差步数，这样能保证两个链表能同时走在尾。一层循环直到两条链表走到相同的结点就返回。
    - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function FindFirstCommonNode(pHead1, pHead2){
    // write code here
    function getListLength(pHead){
        let length = 0
        let pNode = pHead
        while(pNode != null){
            length++
            pNode = pNode.next
        }
        return length
    }
    let pLength1 = getListLength(pHead1)
    let pLength2 = getListLength(pHead2)
    let pLong = pLength1 > pLength2 ? pHead1:pHead2
    let pShort = pLength1 > pLength2 ? pHead2:pHead1
    var nLengthDif = Math.abs(pLength1 - pLength2)
    for(var i = 0 ; i < nLengthDif ; i++){
        pLong = pLong.next
    }
    while(pLong != null && pShort != null && pLong != pShort){
        pShort = pShort.next
        pLong = pLong.next
    }
    return pLong
}
```

  - 方法二
    - 思路：先将一个链表里面的节点存入数组，然后遍历另一个链表的每一个节点，若某个节点存在于数组中，则可以判断该节点为公共节点
    - 代码：

```
/*function ListNode(x){
this.val = x;
    this.next = null;
}*/
function FindFirstCommonNode(pHead1, pHead2)
{
    if(pHead1==null||pHead2==null) return null
    let arr=[]
    let node1=pHead1
    let node2=pHead2
    while(node1){
        arr.push(node1)
        node1=node1.next
    }
    while(node2){
        if(arr.indexOf(node2)>=0){
            return node2
        }
        node2=node2.next
    }
    return null
}
```

- 方法三——推荐——最简洁
  - 思路：用两个指针扫描”两个链表“，最终两个指针到达 null 或者到达公共结点。
    - 长度相同有公共结点，第一次就遍历到；没有公共结点，走到尾部NULL相遇，返回NULL
    - 长度不同有公共结点，第一遍差值就出来了，第二遍一起到公共结点；没有公共，一起到结尾NULL。
  - 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function FindFirstCommonNode(pHead1, pHead2)
{
    // write code here
    var p1 = pHead1
    var p2 = pHead2
    while(p1 !== p2){
        p1 = (p1 === null ? pHead2 : p1.next)
        p2 = (p2 === null ? pHead1 : p2.next)
    }
    return p1
}
```

8.删除链表中重复的结点

- 题目描述：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
- 解答方法：

- 思路：
  - 首先添加一个头节点，以方便碰到第一个，第二个节点就相同的情况
- 设置两个指针，一直往后面搜索
- 代码：

```
/*function ListNode(x){
    this.val = x;
    this.next = null;
}*/
function deleteDuplication(pHead)
{
    // write code here
    let newHead = new ListNode('head')
    newHead.next = pHead
    pHead = newHead
    let qHead = pHead.next
    while(qHead){
        while(qHead.next!==null && qHead.val === qHead.next.val) {
            qHead = qHead.next
        }
        //没移动
        if(pHead.next === qHead) {
            pHead = qHead
            qHead = qHead.next
        }
        //移动了
        else {
            pHead.next = qHead.next
            qHead = qHead.next

        }
    }
    return newHead.next
}
```

六、二叉树

1.重建二叉树

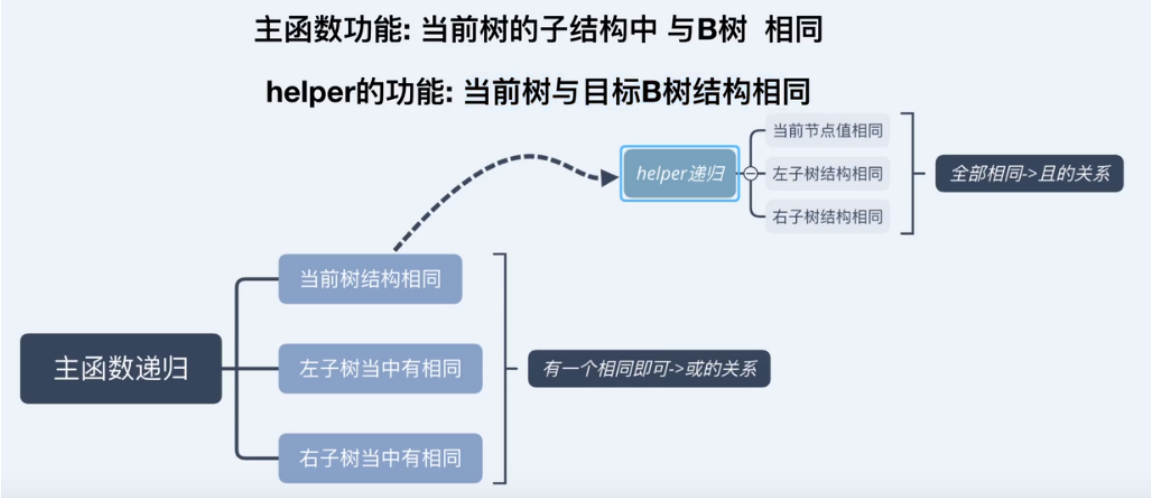
- 题目描述：输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。
- 解答方法：
  - 方法
    - 思路：
      - 构建一个找根节点的index的函数
  - 写递归表达式

- 写递归出口
- 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function reConstructBinaryTree(pre, vin)
{
    //边界条件 && 递归出口
    if(pre === null || pre.length === 0){
        return null
    }
    //1.获取到树根节点的value值
    let rootNode = new TreeNode(pre[0])
    let index = findIndex(pre,vin)
    // 2.构建left左子树，right右子树
    rootNode.left = reConstructBinaryTree(pre.slice(1,index+1),vin.slice(0,index))
    rootNode.right = reConstructBinaryTree(pre.slice(index+1),vin.slice(index+1))
    return rootNode
}
//构造一个找根的index的函数
function findIndex(pre,vin){
    let val = pre[0]
    for(let i=0;i<vin.length;i++){
        if(val === vin[i]){
            return i
        }
    }
}
```

2.树的子结构

- 题目描述：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）
- 解答方法：
  - 方法
    - 思路：



- 代码：
  - 边界条件

主函数中:

-B == null不符合题目说明

-A == null不能包含任意子结构

helper函数中:

- B== null 是递归的终止条件
- B已经为null, 则A为null, 说明结构不同

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function HasSubtree(pRoot1, pRoot2)
{
    if(pRoot1 === null || pRoot2 === null) return false
    return helper(pRoot1,pRoot2) || HasSubtree(pRoot1.left,pRoot2) ||
HasSubtree(pRoot1.right,pRoot2)
}
function helper(pRoot1,pRoot2){
    // pRoot2是空，不管pRoot1是否为空，返回true
    if(pRoot2 === null) return true
    // pRoot2不是空，pRoot1为空
    if(pRoot1 === null) return false
    return pRoot1.val === pRoot2.val && helper(pRoot1.left,pRoot2.left) &&
helper(pRoot1.right,pRoot2.right)
}
```

3.二叉树的镜像

- 题目描述：操作给定的二叉树，将其变换为源二叉树的镜像。
  - 输入描述：

> ```
> 二叉树的镜像定义：源二叉树
>         8
>       /  \
>      6   10
>     / \  / \
>    5  7 9 11
>    镜像二叉树
>         8
>       /  \
>      10   6
>     / \  / \
>    11 9 7  5
> ```

- 解答方法：
  - 方法
    - 思路：递归遍历，交换左右子树
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function Mirror(root)
{
    if(root === null) return
    let temp = root.left
    root.left = root.right
    root.right = temp
    Mirror(root.left)
    Mirror(root.right)
}
```

    - 也可以写成下面的形式

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function Mirror(root)
{
    if(root === null) return root
    let l = Mirror(root.left)
    let r = Mirror(root.right)
```

```
    root.left = r
    root.right = l
    return root
}
```

4.从上往下打印二叉树

- 题目描述：从上往下打印出二叉树的每个节点，同层节点从左至右打印
- 解答方法：
  - 方法
    - 思路：广度优先遍历
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function PrintFromTopToBottom(root)
{
    if(!root) return []
    let queue = [root] //当前队列
    let res = [] //依次存放所有层节点
    while(queue.length){
        let len = queue.length
        for(let i=0;i<len;i++){
            let cur = queue.shift()
            res.push(cur.val)
            if(cur.left) queue.push(cur.left)
            if(cur.right) queue.push(cur.right)
        }
    }
    return res
}
```

5.二叉搜索树的后序遍历序列

- 题目描述：输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。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
- 解答方法：
  - 方法
    - 思路：先找到根节点，然后根据二叉搜索树特点（非空左子树的所有键值小于其根节点的键值；非空右子树的所有键值大于其根节点的键值）来进行判断
    - 代码：

```
function VerifySquenceOfBST(sequence)
{
    // write code here
    if(!sequence.length) return false
```

```
    return adjustSequence(sequence,0,sequence.length-1)
}
function adjustSequence(sequence,left,right){
    if(left>=right) return true
    let root = sequence[right]
    let k = left
    while(k<right && sequence[k]<root) k++
    for(let i=k;i<right;i++){
        if(sequence[i]<root) return false
    }
    return adjustSequence(sequence,left,k-1) && adjustSequence(sequence,k,right-1)
}
```

6.二叉树中和为某一值的路径

- 题目描述：输入一颗二叉树的根节点和一个整数，按字典序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
- 解答方法：
  - 方法
    - 思路：深度优先遍历递归
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
let res
let path
function FindPath(root, sum)
{
    // write code here
    res = [] //存储所有的可能路径
    path = [] //存储当前路径
    if(!root) return []
    dfs(root,sum)
    return res
}
function dfs(node,sum){
    path.push(node.val)
    sum = sum - node.val
    if(!sum && !node.left && !node.right){
        res.push(path.slice())
    }else{
        if(node.left !== null)  dfs(node.left,sum)
        if(node.right !== null)  dfs(node.right,sum)
    }
    path.pop() //这一步很关键，把所有push进去的每一个元素在递归执行完成之时都弹出来，使得path每次都是重头来过
}
```

7.二叉搜索树与双向链表

- 题目描述：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向
- 解答方法：
  - 方法
    - 思路：利用中序遍历的递归写法，调整树中节点指针的指向
    - 代码：

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

let realHead //最终输出的头节点
let tail  //定义的尾节点
function Convert(pRootOfTree)
{
    realHead = null
    tail = null
    ConvertSub(pRootOfTree)
    return realHead
}
function ConvertSub(pRootOfTree){
    if(!pRootOfTree) return
    ConvertSub(pRootOfTree.left)
    if(!tail){
        tail = pRootOfTree
        realHead = pRootOfTree
    }else{
        tail.right = pRootOfTree
        pRootOfTree.left = tail
        tail = pRootOfTree
    }
    ConvertSub(pRootOfTree.right)
}
```

8.二叉树的深度

- 题目描述：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
- 解答方法：
  - 方法一——推荐
    - 思路：利用递归思路，树的深度等于树的高度
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function TreeDepth(pRoot)
{
    // 利用树的深度等于树的高度来解决
    if(!pRoot) return 0
    let left = TreeDepth(pRoot.left)
    let right = TreeDepth(pRoot.right)
    return Math.max(left+1,right+1)
}
```

- 方法二——不太推荐
  - 思路：广度优先遍历，记录遍历的层数
  - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function TreeDepth(pRoot)
{
    if(pRoot === null){
        return 0
    }
    let count = 0
    let stack = [pRoot]
    while(stack.length !== 0){
        let len =  stack.length
        for(let i=0;i<len;i++){
            let cur = stack.shift()
            if(cur.left) stack.push(cur.left)
            if(cur.right) stack.push(cur.right)
        }
        count++
    }
    return count
}
```

8.平衡二叉树

- 题目描述：输入一棵二叉树，判断该二叉树是否是平衡二叉树。在这里，我们只需要考虑其平衡性，不需要考虑其是不是排序二叉树
- 解答方法：
  - 方法
    - 思路：结合前面求二叉树深度的方法，再利用平衡二叉树的特性（任意一节点的左子树高度减去其右子树的高度的绝对值小于等于1）

- 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function IsBalanced_Solution(pRoot)
{
    if(!pRoot) return true
    let l = TreeDepth(pRoot.left)
    let r = TreeDepth(pRoot.right)
    return Math.abs(l-r)<=1 && IsBalanced_Solution(pRoot.left) &&
IsBalanced_Solution(pRoot.right)
}

function TreeDepth(root)
{
    if(!root) return 0
    let left = TreeDepth(root.left)
    let right = TreeDepth(root.right)
    return Math.max(left+1,right+1)
}
```

9.二叉树的下一个结点

- 题目描述：给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针
- 解答方法：
  - 方法
    - 思路：
      - 若该节点有右子树，则查找该右子树的最左节点（若没有最左节点，则返回右子树顶节点）
      - 若该节点没有右子树，则查找第一个当前节点是父节点左孩子的节点
    - 代码：

```
/*function TreeLinkNode(x){
    this.val = x;
    this.left = null;
    this.right = null;
    this.next = null;
}*/
function GetNext(pNode)
{
    if(pNode === null) return null
    //如果有右子树，则找右子树的最左节点
    if(pNode.right!== null){
        pNode = pNode.right
        while(pNode.left!== null){
            pNode = pNode.left
        }
        return pNode
```

```
    }else{
        //没右子树，则找第一个当前节点是父节点左孩子的节点
        while(pNode.next!== null){
            if(pNode.next.left === pNode) return pNode.next
            pNode = pNode.next
        }
    }
    return null //退到了根节点仍没找到，则返回null
}
```

10.对称的二叉树

- 题目描述：请实现一个函数，用来判断一棵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。
- 解答方法：
  - 方法
    - 思路：和前面树的子结构思路类似，因为这是是判断对称，所以除了需要当前节点值相同，还需要A的左与B的右相同，A的右与B的左相同
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function isSymmetrical(pRoot)
{
    if(!pRoot) return true
    return helper(pRoot.left,pRoot.right)
}
function helper(nodeA,nodeB){
    if(nodeA===null && nodeB === null) return true
    if(nodeA===null || nodeB === null) return false
    return nodeA.val === nodeB.val && helper(nodeA.left,nodeB.right) &&
helper(nodeA.right,nodeB.left)
}
```

11.按之字形顺序打印二叉树

- 题目描述：请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。
- 解答方法：
  - 方法
    - 思路：利用广度遍历优先的思路，奇数层不反转，偶数层反转
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
```

```
    this.right = null;
} */
function Print(pRoot)
{
    // 广度优先遍历
    if(!pRoot) return []
    let ans = []
    let stack = [pRoot]
    let num = 1
    while(stack.length!==0){
        let path = []
        let len = stack.length
        for(let i=0;i<len;i++){
            let cur = stack.shift()
            path.push(cur.val)
            if(cur.left) stack.push(cur.left)
            if(cur.right) stack.push(cur.right)
        }
        if(num%2===0){
            ans.push(path.reverse())
        }else{
            ans.push(path)
        }
        num++
    }
    return ans
}
```

- 改进：不再每一层循环时反转数组，而是奇数层时从前向后遍历，偶数层时从后向前遍历

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function Print(pRoot)
{
    // 广度优先遍历
    if(!pRoot) return []
    let ans = []
    let stack = [pRoot]
    let num = 1
    while(stack.length!==0){
        let path = []
        let len = stack.length
        if(num%2===0){
            //偶数层，从后向前遍历
            for(let i=0;i<len;i++){
                let cur = stack.shift()
                path.unshift(cur.val)
                if(cur.left) stack.push(cur.left)
                if(cur.right) stack.push(cur.right)
            }
        }else{
```

```
            //奇数层，从前向后遍历
            for(let i=0;i<len;i++){
                let cur = stack.shift()
                path.push(cur.val)
                if(cur.left) stack.push(cur.left)
                if(cur.right) stack.push(cur.right)
            }
        }

        ans.push(path)
        num++
    }
    return ans
}
```

12.把二叉树打印成多行

- 题目描述：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。
- 解答方法：
  - 方法
    - 思路：利用广度遍历优先的思路
    - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function Print(pRoot)
{
    // 广度优先遍历
    if(!pRoot) return []
    let ans = []
    let stack = [pRoot]
    while(stack.length!==0){
        let path = []
        let len = stack.length
        for(let i=0;i<len;i++){
            let cur = stack.shift()
            path.push(cur.val)
            if(cur.left) stack.push(cur.left)
            if(cur.right) stack.push(cur.right)
        }
        ans.push(path)
    }
    return ans
}
```

13.序列化二叉树

- 题目描述：请实现两个函数，分别用来序列化和反序列化二叉树

    - 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。序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，序列化的结果是一个字符串，序列化时通过 某种符号表示空节点（#），以 ！ 表示一个结点值的结束（value!）。
    - 二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。
    - 例如，我们可以把一个只有根节点为1的二叉树序列化为"1,"，然后通过自己的函数来解析回这个二叉树
- 解答方法：
    - 方法
        - 思路：选择深度优先遍历(先序)，
        - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
let arr = []
function Serialize(pRoot)
{
    if(pRoot === null){
        arr.push('#')
    }else{
        arr.push(pRoot.val)
        Serialize(pRoot.left)
        Serialize(pRoot.right)
    }
}
function Deserialize()
{
    let node = null
    if(arr.length===0){
        return null
    }
    let number = arr.shift()
    if(typeof number == 'number'){
        node = new TreeNode(number)
        node.left = Deserialize()
        node.right = Deserialize()
    }
    return node
}
```

14.二叉搜索树的第k个结点

- 题目描述：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。
- 解答方法：
    - 方法一——不推荐
        - 思路：使用中序递归遍历，但是这里会把所有节点都遍历完，时间复杂度高

- 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
let arr
function KthNode(pRoot, k)
{
    if(!pRoot) return
    arr = []
    inorder(pRoot)
    return arr[k-1]
}
function inorder(pRoot){
    if(pRoot!== null){
        inorder(pRoot.left)
        arr.push(pRoot)
        inorder(pRoot.right)
    }
}
```

- 方法二——改进中序递归遍历
  - 思路：依然使用中序递归遍历，但是在递归时增加边界判断条件，使得在递归到第k小的节点就停止递归了
  - 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
let arr
let key
function KthNode(pRoot, k)
{
    if(!pRoot) return
    arr = []
    key = k
    inorder(pRoot)
    return arr[key-1]
}
function inorder(pRoot){
    if(arr.length === key) return
    if(pRoot!== null){
        inorder(pRoot.left)
        arr.push(pRoot)
        inorder(pRoot.right)
    }
}
```

- 方法三

- 思路：使用中序非递归遍历
- 代码：

```
/* function TreeNode(x) {
    this.val = x;
    this.left = null;
    this.right = null;
} */
function KthNode(pRoot, k)
{
    let res = []
    let temp = pRoot  //temp为当前遍历的节点，初始为根
    let stack = []  // stack作为栈
    while(temp || stack.length!==0){
        if(temp){
            // 遍历左子树
            stack.push(temp)
            // 每遇到非空二叉树先向左走
            temp = temp.left
        }else{
            // temp为空，出栈
            temp = stack.pop()
            // 访问该节点
            res.push(temp)
            if(res.length === k){
                break
            }
            // 向右走一次
            temp = temp.right
        }
    }
    return res[k-1]
}
```

七、堆

1.数据流中的中位数

- 题目描述：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。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。
- 解答方法：
  - 方法一——不推荐，时间复杂度高
    - 思路：每次都对数组进行排序
    - 复杂度：时间复杂度至少为O(n logn)
    - 代码：

```
let arr = []
function Insert(num)
{
    // write code here
    arr.push(num)
}
function GetMedian(){
    //先进行排序
    arr.sort((x,y)=>{
        return x-y
    })
    let len = arr.length
    if(len%2){
        return arr[parseInt(len/2)]
    }else{
        return (arr[len/2-1]+arr[len/2])/2
    }

}
```