

# 剑指offer-c语言

---

## 剑指offer-c语言

- 03.数组中重复的数字
- 04.二维数组中的查找
- 05.替换空格
- 06.从头到尾反着打印链表
- 07.重建二叉树
- 09.用两个栈实现队列 (\*)
- 10.1.斐波那契数列
- 10.2.跳台阶问题
- 11.旋转数组的最小数字
- 12.矩阵中的路径 (\*)
- 13.二维数组路径元素和大于目标值 (\*)
- 14.剪绳子 (将数组分成n段求多少段有最大乘积)
- 15.位1的个数
- 16.实现x的n次幂
- 17.报数
- 18.删除链表节点
- 19.模糊搜索验证 (动态规划, 递归) (\*)
- 20.表示数值的字符串 (\*)
- 21.数组先奇数后偶数
- 22.链表中倒数第k个节点
- 24.翻转链表
- 25.合并两个排序的链表
- 26.树的子结构判断
- 27.二叉树的镜像
- 28.对称的二叉树
- 29.螺旋遍历二维数组 (\*)
- 30.包含min函数的栈 (\*)
- 31.栈的压入, 弹出
- 32.Z型打印二叉树
- 33.知后序遍历判是否为二叉搜索树
- 34.二叉树中和为某一值的路径
- 35.复杂链表的复制 (\*)
- 36.将二叉搜索数转化为排序的双向链表 (\*)
- 37.序列化二叉树 (\*)
- 38.字符串全排列 (无顺序要求, 无重复元素)
- 39.数组中出现次数超过一半的数字
- 40.数组中最小的k个数
- 41.中位数
- 42.连续子数组的最大和
- 43.数字1的个数
- 45.数字序列中的某一位数字
- 45.把数组排成最小的数
- 46.把数字翻译成字符串
- 47.路径的最大价值\*\*\*
- 48.最长不含重复字符的子字符串
- 49.丑数

50.第一次只出现一次的字符  
51.逆序对总数  
52.两个链表中的第一个  
53.在排序数组中查找  
53.2.0~n-1中缺失的数字  
54.寻找二叉搜索树中的目标节点  
55.二叉树的深度  
55.2.平衡二叉树  
56.数组中数字出现的次数  
56.2.数组中数字出现的次数  
57.和为s的两个数字  
57.2.和为s的连续正数序列  
58.翻转单词顺序  
58.2.左旋转字符串  
59.滑动窗口的最大值  
60.n个骰子的点数  
61.扑克牌中的顺子  
62.圆圈中最后剩下的数字  
63.股票的最大利润  
64.求1+2+3+...+n  
68.二叉树的最近公共祖先

## 03.数组中重复的数字

设备中存有 `n` 个文件，文件 `id` 记于数组 `documents`。若文件 `id` 相同，则定义为该文件存在副本。请返回任一存在副本的文件 `id`。

输入: `documents = [2, 5, 3, 0, 5, 0]`  
输出: 0 或 5

思路：用快速排序法`qsort`(数组，数组长度，类型，排序方式)，排序方式为`cmp`

1.额外定义一个排序函数`cmp`

2.主函数内部：先排序；遍历比较当前元素值与下一元素值是否相同，若相同输出该元素值，最后返回-1；

```
static int cmp(const void * a,const void * b){
    return *(int *)a-*(int *)b;
}
int findRepeatDocument(int* documents, int documentsSize) {
    qsort(documents,documentsSize,sizeof(int),cmp);
    for(int i=0;i<documentsSize-1;++i){
        if(documents[i]==documents[i+1]){
            return documents[i];
        }
    }
    return -1;
}
```

## 04.二维数组中的查找

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。

输入: plants = [[2,3,6,8],[4,5,8,9],[5,9,10,12]], target = 8

输出: true

思路: 遍历行和列 如果当前元素==target, 返回true, 否则返回false

```
bool findTargetIn2DPlants(int** plants, int plantsSize, int* plantsColSize, int target) {
    for(int i=0;i<plantsSize;i++)
    {
        for(int j=0;j<*plantsColSize;j++)
        {
            if(plants[i][j]==target)
            {return true;}
        }
    }
    return false;
}
```

## 05.替换空格

假定一段路径记作字符串 path，其中以 "." 作为分隔符。现需将路径加密，加密方法为将 path 中的分隔符替换为空格" "，请返回加密后的字符串。

输入: path = "a.aef.qerf.bb"

输出: "a aef qerf bb"

思路: 1.动态分配数组res记录结果, 定义n方便res记录

2.遍历数组: 如果当前元素不为 "."把他记录到res[n++]里, 否则将res[n++]变成空格

3.返回结果数组

```
char* pathEncryption(char* path) {
    char* res=(char*)malloc(sizeof(char)*(strlen(path)+1));
    int n=0;
    for(int i=0;i<strlen(path );i++)
    {
        if(path[i]!='.')
        {
            res[n++]=path[i];
        }
        else
    }
```

```
    {
        res[n++]=' ';
    }
}
res[n]='\0';
return res;
}
```

## 06.从头到尾反着打印链表

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）

输入: head = [3,6,4,1]

输出: [1,4,6,3]

思路:

- 1.要用数组返回那我们肯定要定义一个数组，但不知道长度，所以先用一个链表指针\*a去遍历数组记录长度
- 2.动态分配一个结果数组去记录逆链表
- 3.让a回到头结点
- 4.数组记录的时候从后往前记录，结点从前往后前进遍历
- 5.返回结果数组

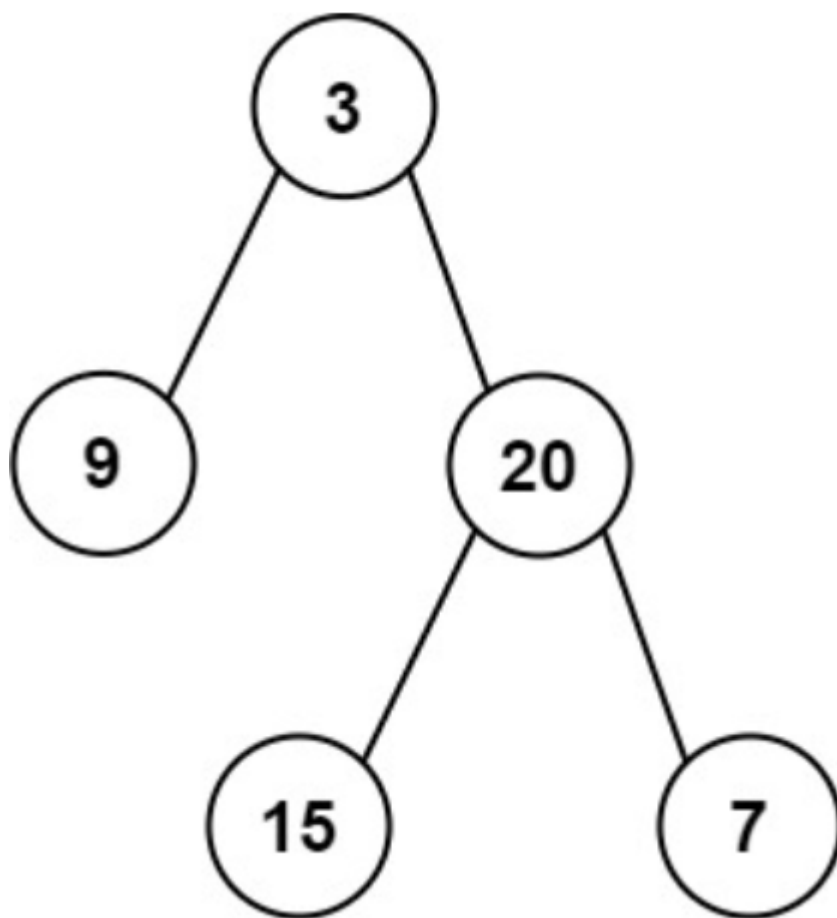
```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

int* reverseBookList(struct ListNode* head, int* returnSize) {
    struct ListNode *a=head;
    int res=0;
    while(a)
    {
        res++;
        a=a->next;
    }
    *returnSize=res;
    int i=res-1;
    int *ans=(int*)malloc(res*sizeof(int));
    a=head;
    while(a)
    {
        ans[i--]=a->val;
        a=a->next;
    }
}
```

```
}  
    return ans;  
}
```

## 07.重建二叉树

二叉树的先序遍历结果记录于整数数组 `preorder`，它的中序遍历结果记录于整数数组 `inorder`。请根据 `preorder` 和 `inorder` 的提示构造出这棵二叉树并返回其根节点。



输入: `preorder = [3,9,20,15,7]`, `inorder = [9,3,15,20,7]`

输出: `[3,9,20,null,null,15,7]`


思路: 1.判空反空

2.动态分配一个虚拟头结点root方便后续返回这个虚拟头结点就好,初始化left, right, val (根节点其实就是 `preorder[0]`)

3.遍历中序遍历数组找到这个根节点, 因为中序遍历根节点左边就是左子树, 右边就是右子树

4.调用函数遍历左子树和右子树这个地方要注意!!

5.返回root

831915607931137575

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* deduceTree(int* preorder, int preordersize, int* inorder, int inorderSize)
{
    if(!preordersize)
        return NULL;
    struct TreeNode * root = (struct TreeNode *)malloc(sizeof(struct TreeNode));
    root->val = preorder[0];
    root->left = NULL;
    root->right = NULL;

    int i = 0 ;

    for(; i < inorderSize ;i++){
        if(inorder[i] == preorder[0])
            break;
    }
    if(i) root->left = deduceTree(preorder+1,i,inorder,i);
    if(inorderSize-i-1) root->right = deduceTree(preorder+i+1,inorderSize-i-1 ,
inorder+1+i ,inorderSize-i-1);
    return root;
}

```

## 09.用两个栈实现队列 (\*)

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：

["BookQueue", "push", "push", "pop"]

[[], [1], [2], []]

输出: [null,null,null,1]

解释：

MyQueue myQueue = new MyQueue();

myQueue.push(1); // queue is: [1]

myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

myQueue.pop(); // return 1, queue is [2]

思路：这题我觉得挺难的但是leetcode里面显示他是个简单题

当调用 **push** 让元素入队时，只要把元素压入 **s1** 即可：



使用 **peek** 或 **pop** 操作队头的元素时，若 **s2** 为空，可以把 **s1** 的所有元素取出再添加进 **s2**，这时候 **s2** 中元素就是先进先出顺序了：



#首先理清我们要完成几个任务：1.创建两个栈并且初始化；2.将新入队的元素压进s1里面；3.元素从s1出来进s2里；4.释放内存

步骤：

0.预处理：MAX\_QUEUE\_SIZE 10000

1.定义结构需要2个stack和2个top

2.动态分配cq，判空；给 cq->stackA 和 cq->stackB 分别动态分配内存并且判空如果空释放cq返空，否则A和B的栈顶为-1

3.当栈顶A没满，将元素压进去

4.分两种情况：

#需要中间值tempb, tempa

1. B不为空: 让b的元素出队, 返回tempb
2. B为空, A不为空: 让a的元素出队, 将元素压进b里

5.当队列不为空, 如果a不为空释放a, 如果b不为空释放b, 最后释放队列

```
#define MAX_QUEUE_SIZE 10000

typedef struct {
    int *stackA;
    int *stackB;
    int topA;
    int topB;
} CQueue;

CQueue* CQueueCreate() {
    CQueue *cq;
    cq = (CQueue*)malloc(sizeof(CQueue));
    if (cq == NULL) { return NULL; }
    cq->stackA = (int*)malloc(MAX_QUEUE_SIZE * sizeof(int));
    if (cq->stackA == NULL) {
        free(cq);
        return NULL;
    }
    cq->topA = -1;

    cq->stackB = (int*)malloc(MAX_QUEUE_SIZE * sizeof(int));
    if (cq->stackB == NULL) {
        free(cq->stackA);
        free(cq);
        return NULL;
    }
    cq->topB = -1;
    return cq;
}

void CQueueAppendTail(CQueue* obj, int value) {
    //将新入队的元素压入栈A
    if (obj->topA != MAX_QUEUE_SIZE) {
        obj->stackA[++obj->topA] = value;
    }
}

int CQueueDeleteHead(CQueue* obj) {
    int topBElem = 0, topAElem = 0;

    //如果栈B不为空, 将在B栈顶的元素出队
    if (obj->topB != -1) {
        topBElem = obj->stackB[obj->topB--];
        return topBElem;
    }

    //如果栈B为空, 且栈A不为空, 将栈A中所有的元素压入栈B
```



```

    if (obj->topB == -1 && obj->topA != -1) {
        while (obj->topA != -1) {
            topAElem = obj->stackA[obj->topA--];
            obj->stackB[++obj->topB] = topAElem;
        }
    }
    return -1;
}

void cQueueFree(CQueue* obj) {
    if (obj != NULL) {
        if (obj->stackA != NULL) { free(obj->stackA); }
        if (obj->stackB != NULL) { free(obj->stackB); }
        free(obj);
    }
}

```

## 10.1.斐波那契数列

斐波那契数\*\*（通常用`F(n)`表示）形成的序列称为\*\*斐波那契数列\*\*。该数列由\*\*0\*\*和\*\*1\*\*开始，后面的每一项数字都是前面两项数字的和。也就是：

$F(0) = 0, F(1) = 1$   
 $F(n) = F(n - 1) + F(n - 2)$ , 其中  $n > 1$

给定`n`，请计算`F(n)`

答案需要取模  $1e9+7(1000000007)$ ，如计算初始结果为：1000000008，请返回 1。

思路：定义一个数组a继续动态规划递推

边界条件是n=0或者n=1，返回的是0或1（也就是n）

递推公式就是 $a(n) = (F(n - 1) + F(n - 2)) \% 1000000007$

```

int a[101]={0};

int fib(int n){
    if(n==0||n==1)return n;
    if(a[n]!=0)return a[n];

    a[n]=(fib(n-1)+fib(n-2))%1000000007;

    return a[n];
}

```

## 10.2.跳台阶问题

平台有 num 个小格子，每次可以选择跳一个格子或者两个格子。请返回在训练过程中，学员们共有多少种不同的跳跃方式。

结果可能过大，因此结果需要取模  $1e9+7$  (1000000007)，如计算初始结果为：1000000008，请返回 1。

输入：n = 2

输出：2

思路：定义一个数组长度为num+1和i

边界条件是：i=0和i=1：a[i]=1

递推公式是：a[i]=(a[i-1]+a[i-2])%1000000007

```
int trainways(int num) {
    int a[num+1];
    int i;
    for(i=0;i<=num;i++)
    {
        if(i==0||i==1){a[i]=1;}
        else {a[i]=(a[i-1]+a[i-2])%1000000007;}
    }
    return a[num];
}
```

## 11.旋转数组的最小数字

仓库管理员以数组 stock 形式记录商品库存表。stock[i] 表示商品 id，可能存在重复。原库存表按商品 id 升序排列。现因突发情况需要进行商品紧急调拨，管理员将这批商品 id 提前依次整理至库存表最后。请你找到并返回库存表中编号的 **最小的元素** 以便及时记录本次调拨。

输入：stock = [4,5,8,3,4]

输出：3

思路：用二分法左右指针去找最小值

1.如果最左值小于最右值则说明没旋转，最左值就是最小元素

2.定义mid，如果中间值大于最左边值，说明最小值在右边，左指针往右移动

如果中间值小于最左边值，说明最小值在左边，右指针往左移动

等于的话让左指针往右移动一个

```
int stockManagement(int* stock, int stockSize) {
    int left=0,right=stockSize-1;

    while(left<right)
    {
        if(stock[left]<stock[right]){return stock[left];}
        int mid=left+(right-left)/2;
```

```
if(stock[mid]>stock[left]){left=mid+1;}
else if(stock[mid]<stock[left]){right=mid;}
else{left++;}
}
return stock[left];
}
```

## 12.矩阵中的路径 (\*)

字母迷宫游戏初始界面记作  $m \times n$  二维字符串数组 `grid`，请判断玩家是否能在 `grid` 中找到目标单词 `target`。  
注意：寻找单词时 **必须** 按照字母顺序，通过水平或垂直方向相邻的单元格内的字母构成，同时，同一个单元格内的字母 **不允许** 被重复使用。

A	B	C	E
S	F	C	S
A	D	E	E

输入: `grid = [ ["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"] ]`, `target = "ABCCED"`  
输出: `true`

思路：虽然是中等题但是二维数组还是不熟练所以标\*

- 1.首先宏定义最大行数列数，全局变量行数列数
- 2.深度优先函数dfs去检查从当前位置出发的路径能不能组成目标字符串，并临时标记已访问过的字符防止重复访问
- 3.主函数判断遍历网格中的每个位置，如果从某个位置出发可以找到目标字符串则返回true

详细说说每个思路里要做什么步骤：

·dfs函数：

1.超出边界条件返回false, 遍历完了返回true

2.定义一个temp=数组当前元素, 并标记为\0

3.bool res等于遍历四个方向

4.恢复当前元素不标记

5.返回res

·主函数

1.外部遍历行内部遍历列, 若调用函数成功, 返回true, 否则false

```
#define MAX_ROWS 100
#define MAX_COLS 100
int rows, cols;

bool dfs(char grid[MAX_ROWS][MAX_COLS], char* target, int i, int j, int k) {
    if (i >= rows || i < 0 || j >= cols || j < 0 || grid[i][j] != target[k])
    {
        return false;
    }
    if (k == strlen(target) - 1)
    {
        return true;
    }

    char temp = grid[i][j];
    grid[i][j] = '\0';
    bool res = dfs(grid, target, i + 1, j, k + 1) ||
                dfs(grid, target, i - 1, j, k + 1) ||
                dfs(grid, target, i, j + 1, k + 1) ||
                dfs(grid, target, i, j - 1, k + 1);
    grid[i][j] = temp;

    return res;
}

bool wordPuzzle(char grid[MAX_ROWS][MAX_COLS], int gridRows, int gridCols, char* target) {
    rows = gridRows;
    cols = gridCols;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (dfs(grid, target, i, j, 0)) {
                return true;
            }
        }
    }
    return false;
}
```

## 13.二维数组路径元素和大于目标值 (\*)

家居整理师将待整理衣橱划分为  $m \times n$  的二维矩阵 `grid`，其中 `grid[i][j]` 代表一个需要整理的格子。整理师自 `grid[0][0]` 开始 **逐行逐列** 地整理每个格子。

整理规则为：在整理过程中，可以选择 **向右移动一格** 或 **向下移动一格**，但不能移动到衣柜之外。同时，不需要整理  $\text{digit}(i) + \text{digit}(j) > \text{cnt}$  的格子，其中  $\text{digit}(x)$  表示数字  $x$  的各数位之和。

请返回整理师 **总共需要整理多少个格子**。

输入:  $m = 4, n = 7, \text{cnt} = 5$   
输出: 18

思路: flag数组记录格子是否被抵达了

遍历行: 计算横坐标的和

遍历列: 计算纵坐标的和

当满足边界条件的时候把当前格子标记,  $\text{res}++$

最后返回res

```
int wardrobeFinishing(int m, int n, int cnt) {
    int i, j, sumi, sumj, res=0, resold=0;
    int flag[m][n];
    for(int i=0; i<m; i++)
    {
        if(i%10==0){sumi=i/10;}
        else{sumi++;}

        for(int j=0; j<n; j++)
        {
            if(j%10==0){sumj=j/10;}
            else{sumj++;}
            if((j==0 || j>0&&flag[i][j-1]==1 || i>0&&flag[i-1][j]==1)&&sumi+sumj<=cnt)
            {
                flag[i][j]=1;
                res++;
            }
            else
            {
                flag[i][j]=0;
            }
        }
        if(res!=resold){resold=res;}
        else{break;}
    }
    return res;
}
```

## 14.剪绳子 (将数组分成n段求多少段有最大乘积)

现需要将一根长为正整数 `bamboo_len` 的竹子砍为若干段，每段长度均为正整数。请返回每段竹子长度的最大乘积是多少。

输入: bamboo\_len = 12  
输出: 81

思路：动态递归嘛就是，看是分成两段的乘积比较大还是分成多段的乘积比较大

多段乘积又分为两端和多段

所以步骤就是定义一个dp数组并初始化，从i=2开始遍历（因为一整段没必要遍历），j从1递归到i，更新最大值，然后赋值给dp。

```
int cuttingBamboo(int bamboo_len) {
    int dp[bamboo_len+1];
    memset(dp,0,sizeof(dp));
    for(int i=2;i<=bamboo_len;i++)
    {
        int curMax=0;
        for(int j=1;j<i;j++)
        {
            curMax=fmax(curMax,fmax(j*(i-j),j*dp[i-j]));
        }
        dp[i]=curMax;
    }
    return dp[bamboo_len];
}
```

### 15.位1的个数

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为 [汉明重量](#)）。

[illegible]

思路：其实这个题乍一看很难其实很简单，题目的n是uint32\_t类型所以输入的已经是000...

当遇到一个1就ret记录一次，然后把这个1变成0，免得重复记录，最后返回ret就好

```
int hammingweight(uint32_t n) {
    int ret=0;
    while(n)
    {
        n&=n-1;
        ret++;
    }
    return ret;
}
```

---

## 16.实现x的n次幂

实现 `pow(x, n)`，即计算 x 的 n 次幂函数

```
输入: x = 2.00000, n = 10
输出: 1024.00000
```

思路：只要注意有几种情况就好了

第一种：n=0，谁的0次方都是1

第二种：如果是偶次方，则res等于res乘x

第三种：如果是奇次方，则x等于x乘x

第四种：如果次方是复数则要返回的是倒数

```
double myPow(double x, int n){
    if(n==0){return 1.0;}
    double res=1.0;
    for(int i=n;i!=0;i/=2)
    {
        if(i%2!=0)
        {
            res*=x;
        }
        x*=x;
    }
    return n<0?1/res:res;
}
```

---

## 17.报数

实现一个十进制数字报数程序，请按照数字从小到大的顺序返回一个整数数列，该数列从数字 1 开始，到最大的正整数 cnt 位数字结束。

```
输入: cnt = 2
输出:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,6
4,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94
,95,96,97,98,99]
```

思路：如果cnt是1，则到9；如果是2，则到99；如果是3，则到999；

所以说我们得先判断他是到nums这个nums为几；

知道为几之后动态分配一个数组来储存这个报数

遍历到nums，储存值

```

int* countNumbers(int cnt, int* returnSize) {
    int nums=1;
    while(cnt>0)
    {
        nums=nums*10;
        cnt--;
    }

    int *ans=(int*)malloc(sizeof(int)*(nums-1));
    for(int i=1;i<nums;i++)
    {
        ans[i-1]=i; //因为报数是从1开始的如果是从0开始的则元素等于他的下标
    }
    *returnSize=nums-1;
    return ans;
    free(ans);
}

```

## 18.删除链表节点

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。

返回删除后的链表的头节点。

输入：head = [4,5,1,9], val = 5

输出：[4,1,9]

解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9。

思路：需要注意的点是删除的可能是第一个节点

先判空返空，如果删除的是第一个节点那就返回从下一个节点开始的链表；

定义链表指针p，当p的下一个不为空且下一个的值为目标值，则删除节点，否则前进；

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

struct ListNode* deleteNode(struct ListNode* head, int val){
    if(head==NULL)return NULL;
    if(head->val == val)
    {
        return head->next;
    }

    struct ListNode* p=head;
    while((p->next!=NULL)&&(p->next->val!=val))

```



```

{
    p = p->next;
}
if(p->next != NULL)
{
    p->next = p->next->next;
}

return head;
}

```

## 19.模糊搜索验证（动态规划，递归）（\*）

请设计一个程序来支持用户在文本编辑器中的模糊搜索功能。用户输入内容中可能使用到如下两种通配符：

- '.' 匹配任意单个字符。
- '\*' 匹配零个或多个前面的那一个元素。

请返回用户输入内容 `input` 所有字符是否可以匹配原文字符串 `article`。

输入: `article = "aa", input = "a"`  
 输出: `false`  
 解释: "a" 无法匹配 "aa" 整个字符串。

思路：我觉得这题挺难的因为理解题目就已经很难了还要看怎么匹配

我们可以先注意到\*号是可以匹配前面的元素

所以讨论如果第二个字符为\*的话，可以分成两种情况：匹配0次和一次或多次

两种情况：

// 1. '\*' 匹配零次，跳过模式的前两个字符继续匹配

// 2. '\*' 匹配一次或多次，且字符串的第一个字符与模式的第一个字符匹配（或者模式的第一个字符是 '.'），继续匹配字符串的下一个字符

```

bool articleMatch(char* s, char* p) {
    int ls=strlen(s),lp=strlen(p);
    if(lp==0){return ls==0?true:false;}
    if(p[1]=='*')// 如果模式的第二个字符是 '*'
    {
        // 两种情况:
        // 1. '*' 匹配零次，跳过模式的前两个字符继续匹配
        // 2. '*' 匹配一次或多次，且字符串的第一个字符与模式的第一个字符匹配（或者模式的第一个字符是 '.'），继续匹配字符串的下一个字符
        return articleMatch(s,p+2) || (ls!=0&&(s[0]==p[0] || p[0]=='.')&&articleMatch(s+1,p));
    }
    // 如果模式的第二个字符不是 '*'
    // 检查字符串第一个字符与模式第一个字符是否匹配（或模式第一个字符是 '.'），并继续匹配后续字符
    return ls!=0&&(s[0]==p[0] || p[0]=='.')&&articleMatch(s+1,p+1);
}

```

## 20.表示数值的字符串 (\*)

**有效数字** (按顺序) 可以分成以下几个部分:

1. 若干空格
2. 一个 **小数** 或者 **整数**
3. (可选) 一个 `'e'` 或 `'E'` , 后面跟着一个 **整数**
4. 若干空格

**小数** (按顺序) 可以分成以下几个部分:

1. (可选) 一个符号字符 (`'+'` 或 `'-'`)
2. 下述格式之一:
  1. 至少一位数字, 后面跟着一个点 `'.'`
  2. 至少一位数字, 后面跟着一个点 `'.'` , 后面再跟着至少一位数字
  3. 一个点 `'.'` , 后面跟着至少一位数字

**整数** (按顺序) 可以分成以下几个部分:

1. (可选) 一个符号字符 (`'+'` 或 `'-'`)
2. 至少一位数字

部分有效数字列举如下: `["2", "0089", "-0.1", "+3.14", "4.", "-.9", "2e10", "-90E3", "3e+7", "+6e-1", "53.5e93", "-123.456e789"]`

部分无效数字列举如下: `["abc", "1a", "1e", "e3", "99e2.5", "--6", "-+3", "95a54e53"]`

给你一个字符串 `s` , 如果 `s` 是一个 **有效数字** , 请返回 `true` 。

这题略了这题说实话题目就特别难, 遇到了也做不出来

拜拜这个分给了!

---

## 21.数组先奇数后偶数

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, 使得所有奇数在数组的前半部分, 所有偶数在数组的后半部分。

输入: `actions = [1,2,3,4,5]`

输出: `[1,3,5,2,4]`

解释: 为正确答案之一

思路: 用二分法左右指针: 左指针找奇数, 右指针找偶数, 如果不对的左右指针所指元素交换

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* trainingPlan(int* actions, int actionsSize, int* returnSize) {
    int left=0,right=actionsSize-1;
    *returnSize=actionsSize;

    while(left<right)
```

```

{
    while(left<actionsSize&&actions[left]%2==1)
    {
        left++; //左指针指的元素就是奇数
    }
    while(right>=0&&actions[right]%2==0)
    {
        right--; //右指针指的元素就是偶数
    }
    if(left>=right)
    {
        break;
    }
    //循环到这说明左指针指的是偶数，右指针指的是奇数，那就交换他们俩
    int temp=actions[left];
    actions[left]=actions[right];
    actions[right]=temp;
    left++;
    right--;
}
return actions;
}

```

## 22.链表中倒数第k个节点

给定一个头节点为 `head` 的链表用于记录一系列编号，请查找并返回倒数第 `cnt` 个编号。

输入: `head = [2,4,7,8]`, `cnt = 1`  
 输出: 8

思路: 创建两个指针，让p1先走cnt步，然后再让p1, p2同时走，当p1走到空，p2就走到倒数第k个节点的位置了

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* trainingPlan(struct ListNode* head, int cnt) {
    struct ListNode* p1=head;
    struct ListNode* p2=head;

    for(int i=0;i<cnt;i++)
    {
        p1=p1->next;
    }

    while(p1!=NULL)
    {

```

```

        p1=p1->next;
        p2=p2->next;
    }
    return p2;
}

```

## 24.翻转链表

给定一个头节点为 `head` 的单链表用于记录一系列核心肌群训练编号，请将该系列训练编号 **倒序** 记录于链表并返回。

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

思路：首先肯定是判空返空，先定义一个链表指针p和虚拟头结点a，a的下一个指向空为了存放倒序链表，让p开始遍历链表，然后新建一个q指针为p的下一位，交换他们的位置，使得a为倒叙

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* trainningPlan(struct ListNode* head) {
    if(head==NULL)return head;
    struct ListNode*p;
    p=head;
    struct ListNode*a=(struct ListNode*)malloc(sizeof(struct ListNode));
    a->next=NULL;

    while(p)
    {
        struct ListNode*q=p->next;
        p->next=a->next;
        a->next=p;
        p=q;
    }
    return a->next;
}

```

## 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: `l1 = [1,2,4]`, `l2 = [1,3,4]`  
 输出: `[1,1,2,3,4,4]`

思路：如果l1为空返l2，如果l2为空返l1

如果l1当前值小于l2当前值，那么l1下一位为调用函数合并l1下一位和l2

反之同理

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* trainningPlan(struct ListNode* l1, struct ListNode* l2) {
    if(l1==NULL)return l2;
    if(l2==NULL)return l1;

    if(l1->val<l2->val)
    {
        l1->next= trainningPlan(l1->next,l2);
        return l1;
    }
    else{
        l2->next=trainningPlan(l2->next,l1);
        return l2;
    }
}
```

## 26.树的子结构判断

给定两棵二叉树 `tree1` 和 `tree2`，判断 `tree2` 是否以 `tree1` 的某个节点为根的子树具有 **相同的结构和节点值**。  
注意，**空树** 不会是以 `tree1` 的某个节点为根的子树具有 **相同的结构和节点值**。

输入: `tree1 = [1,7,5]`, `tree2 = [6,1]`  
输出: `false`  
解释: `tree2` 与 `tree1` 的一个子树没有相同的结构和节点值。

输入: `tree1 = [3,6,7,1,8]`, `tree2 = [6,1]`  
输出: `true`  
解释: `tree2` 与 `tree1` 的一个子树拥有相同的结构和节点值。即 `6 -> 1`。

思路：先构造一个辅助函数去帮忙如何判定子结构，主函数去调用递归遍历左子树和右子树

辅助函数要注意的是，如果2为空他是可以作为1的字结构的，但1为空就是错的，然后就看当前值是否相等，不相等也是错的，调用辅助函数递归左右子树

主函数里呢如果a为空或者b为空都返空（题目），如果调用辅助函数成立，返true。然后调用主函数递归左右子树

```
/**
 * Definition for a binary tree node.
```

```

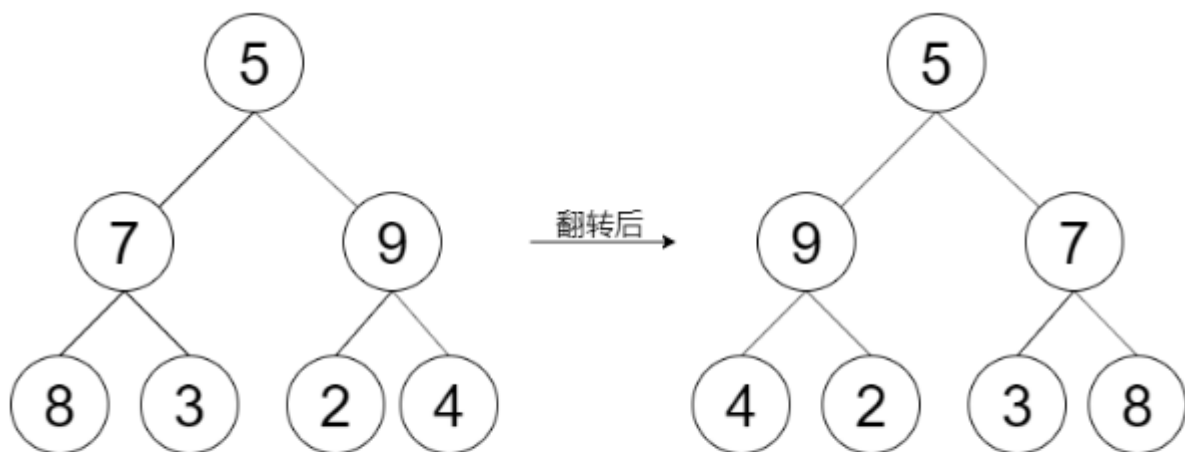
* struct TreeNode {
*     int val;
*     struct TreeNode *left;
*     struct TreeNode *right;
* };
*/
static bool issametree(struct TreeNode*A,struct TreeNode*B)
{
    if(B==NULL)return true;
    if(A==NULL)return false;
    if(A->val!=B->val)return false;
    return issametree(A->left,B->left)&&issametree(A->right,B->right);
}

bool isSubStructure(struct TreeNode* A, struct TreeNode* B){
    if(A==NULL||B==NULL)return false;
    if(issametree(A,B))return true;
    return isSubStructure(A->left,B)||isSubStructure(A->right,B);
}

```

## 27.二叉树的镜像

给定一棵二叉树的根节点 `root`，请左右翻转这棵二叉树，并返回其根节点。



输入: `root = [5,7,9,8,3,2,4]`

输出: `[5,9,7,4,2,3,8]`

思路：这个题就很简单啦，定义左树指针和右树指针去调用函数递归左右子树，左子树赋值右指针，右子树赋值左指针，返回根。

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;

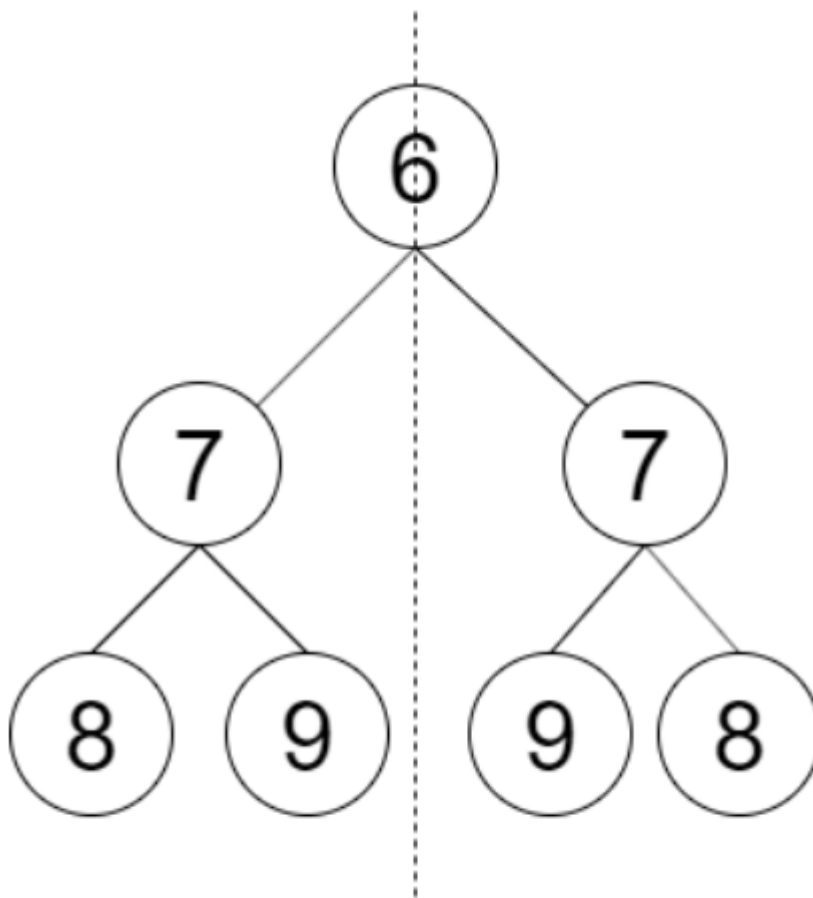
```

```
*      struct TreeNode *right;
* };
*/

struct TreeNode* mirrorTree(struct TreeNode* root){
    if(root==NULL)return NULL;
    struct TreeNode*left=mirrorTree(root->left);
    struct TreeNode*right=mirrorTree(root->right);
    root->left=right;
    root->right=left;
    return root;
}
```

## 28.对称的二叉树

请设计一个函数判断一棵二叉树是否 **轴对称**。



输入: root = [6,7,7,8,9,9,8]

输出: true

解释: 从图中可看出树是轴对称的。

思路：把左右子树当作是两棵树，用辅助函数去构造怎么判断两个树是不是对称，然后主函数调用递归左右子树就好了

辅助函数要怎么判断呢？有三种情况：1和2都是空树返true；1或者2是空树返false；1当前值不等于2当前值；调用递归判断（1左子树和2右子树）和（1右子树和2左子树）

主函数判空然后调用递归左右子树就好了

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool issymmetric(struct TreeNode* root1, struct TreeNode* root2){
    //递归结束条件
    if(root1 == NULL && root2 == NULL){
        return true;
    }
    else if(root1 == NULL || root2 == NULL){
        return false;
    }
    if(root1 -> val != root2 -> val){
        return false;
    }
    return issymmetric(root1 -> left, root2 -> right) && issymmetric(root1 -> right, root2 -> left);
}

bool checkSymmetricTree(struct TreeNode* root) {
    if(root == NULL){
        return true;
    }
    return issymmetric(root -> left, root -> right);
}
```

## 29.螺旋遍历二维数组 (\*)

给定一个二维数组 `array`，请返回「螺旋遍历」该数组的结果。

**螺旋遍历**：从左上角开始，按照 **向右、向下、向左、向上** 的顺序 **依次** 提取元素，然后再进入内部一层重复相同的步骤，直到提取完所有元素。

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

思路：

先定义四个方向directions [4][2]



一些特殊情况：若行或者列为0，结果数组长度为0，返回0；

准备正事：

- 1.需要一个visit二维数组来标记这个地方走过啦（新建行列数据为原数组的数据，还要记得初始化）；
- 2.需要动态分配一个order一维数组来存放螺旋遍历的结果（长度为行\*列），返回长度也为行\*列；

开始正式遍历咯：

- 1.定义row和column为0，开始位置index为0
- 2.遍历total（也就是行\*列）先记录order开始存元素，visited开始标记走过的为true
- 3.定义nextrow为row+方向[index]，nextcol为column+方向 [index][1]
- 4.如果（满足五个超出边界：nextrow或者nextcol小于0或者大于原数组边界或者visited过了）那么index更新
- 5.否则更新row和column（根据方向）
- 6.返回order

```
int directions[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

int* spiralArray(int** array, int arraySize, int* arrayColSize, int* returnSize) {
    if (arraySize == 0 || arrayColSize[0] == 0) {
        *returnSize = 0;
        return NULL;
    }

    int rows = arraySize, columns = arrayColSize[0];
    int visited[rows][columns];
    memset(visited, 0, sizeof(visited));
    int total = rows * columns;
    int* order = malloc(sizeof(int) * total);
    *returnSize = total;

    int row = 0, column = 0;
    int directionIndex = 0;
    for (int i = 0; i < total; i++) {
        order[i] = array[row][column];
        visited[row][column] = true;
        int nextRow = row + directions[directionIndex][0], nextColumn = column +
        directions[directionIndex][1];
        if (nextRow < 0 || nextRow >= rows || nextColumn < 0 || nextColumn >= columns ||
        visited[nextRow][nextColumn]) {
            directionIndex = (directionIndex + 1) % 4;
        }
        row += directions[directionIndex][0];
        column += directions[directionIndex][1];
    }
    return order;
}
```

## 30.包含min函数的栈 (\*)

请你设计一个 **最小栈**。它提供 `push` , `pop` , `top` 操作, 并能在常数时间内检索到最小元素的栈。

输入:

```
["MinStack","push","push","push","getMin","pop","top","getMin"]  
[[],[-2],[2],[-3],[],[],[],[]]
```

输出:

```
[null,null,null,null,-3,null,2,-2]
```

解释:

```
MinStack minStack = new MinStack();  
minStack.push(-2);  
minStack.push(2);  
minStack.push(-3);  
minStack.getMin();   --> 返回 -3.  
minStack.pop();  
minStack.top();       --> 返回 2.  
minStack.getMin();    --> 返回 -2.
```

思路: 该死的栈又是你啊!

先看我们要实现哪些功能: 创建最小栈, 入栈, 出栈, 栈顶, 栈的最小值, 释放内存

我感觉这个题我做不出来要是考到这个题要不给了吧我把代码放过来

简单来说就是每个元素入栈时, 还要记录下来当前栈的最小值。

```
typedef struct Node{  
    int nodeVal;  
    int minVal;  
    struct Node* next;  
} stackNode;  
typedef struct {  
    stackNode* head;  
} MinStack;  
  
/** initialize your data structure here. */  
  
MinStack* minStackCreate() {  
    MinStack* myMinStack = (MinStack*)malloc(sizeof(MinStack));  
    myMinStack->head = (stackNode*)malloc(sizeof(stackNode));  
    myMinStack->head->next = NULL;  
    return myMinStack;  
}  
  
void minStackPush(MinStack* obj, int x) {  
    if (obj == NULL) return;  
    // Create new node:  
    stackNode* newNode = (stackNode*)malloc(sizeof(stackNode));  
    newNode->nodeVal = x;  
    newNode->minVal = x;  
    newNode->next = NULL;
```

```

// first node direct add
if(obj->head->next == NULL) {
    obj->head->next = newNode;

} else{
    //compare value and min to head first.
    newNode->next = obj->head->next;
    obj->head->next = newNode;
    if (x < obj->head->next->next->minVal) {
        obj->head->next->minVal = x;
    } else {
        obj->head->next->minVal = obj->head->next->next->minVal;
    }
}
}

void minStackPop(MinStack* obj) {
    if(obj == NULL) return;
    stackNode* tempNode = obj->head->next;
    obj->head->next = tempNode->next;
    free(tempNode);
}

int minStackTop(MinStack* obj) {
    if(obj == NULL) return -1;

    return obj->head->next->nodeVal;
}

int minStackGetMin(MinStack* obj) {
    if(obj == NULL) return -1;

    return obj->head->next->minVal;
}

void minStackFree(MinStack* obj) {

    while(obj->head!=NULL) {
        stackNode* tempNode = obj->head;
        obj->head = obj->head->next;
        free(tempNode);
        tempNode = NULL;
    }
}

/**
 * Your MinStack struct will be instantiated and called as such:
 * MinStack* obj = minStackCreate();
 * minStackPush(obj, x);

 * minStackPop(obj);

 * int param_3 = minStackTop(obj);

```

```
* int param_4 = minStackMin(obj);

* minStackFree(obj);
*/
```

## 31.栈的压入，弹出

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否是该栈的弹出顺序。

输入: putIn = [6,7,8,9,10,11], takeOut = [9,11,10,8,7,6]  
输出: true  
解释: 我们可以按以下操作放入并拿取书籍:  
push(6), push(7), push(8), push(9), pop() -> 9,  
push(10), push(11), pop() -> 11, pop() -> 10, pop() -> 8, pop() -> 7, pop() -> 6

思路: 用数组假装是栈来实行操作

动态分配stack数组长度就是putInSize, 定义一个top假装栈顶-1, index索引0;

遍历putIn数组, 把putIn数组元素放进stack数组里, 当top不为-1同时top元素等于takeOut的索引元素, 就让top--, index++

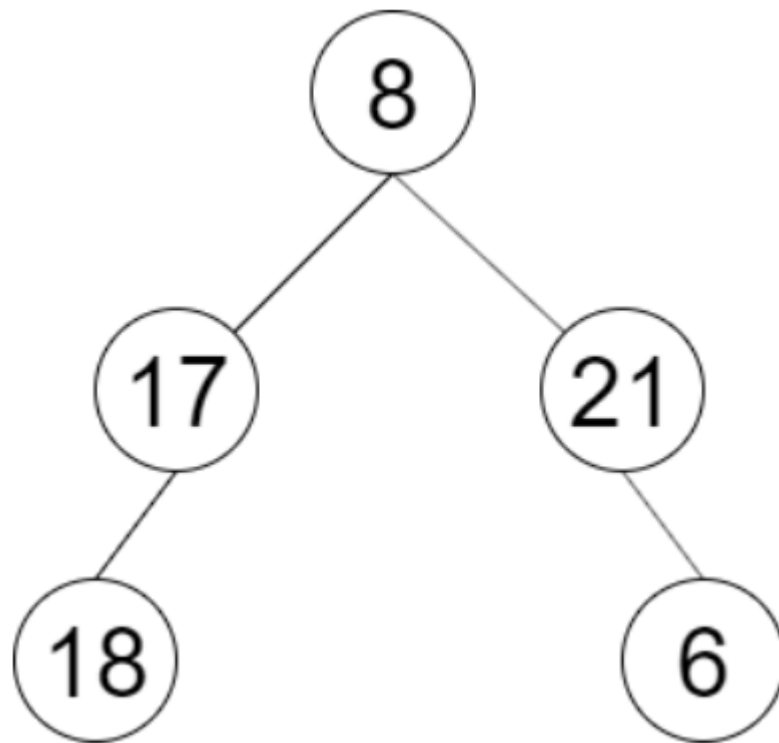
如果假栈能空返说明true, 否则false;

```
bool validateBookSequences(int* putIn, int putInSize, int* takeOut, int takeOutSize) {
    int *stack=(int*)malloc(sizeof(int)*putInSize);
    int top=-1,index=0;
    for(int i=0;i<putInSize;i++)
    {
        stack[++top]=putIn[i];
        while((top!=-1)&&stack[top]==takeOut[index])
        {
            top--;
            index++;
        }
    }
    if(top===-1)
    {
        return true;
    }
    return false;
}
```

## 32.Z型打印二叉树

一棵圣诞树记作根节点为 `root` 的二叉树, 节点值为该位置装饰彩灯的颜色编号。请按照如下规则记录彩灯装饰结果:

- 第一层按照从左到右的顺序记录
- 除第一层外每一层的记录顺序均与上一层相反。即第一层为从左到右，第二层为从右到左。



输入: root = [8,17,21,18,null,null,6]

输出: [[8],[21,17],[18,6]]

思路: 层序遍历整个树, 用k判断该层是奇层还是偶层将偶层翻转

首先我们知道翻转数组要先定义一个reverse函数, 用左右指针三步走翻转

而层序遍历中需要定义返回结果二维数组, 定义队列用于层序遍历

```

void reverse(int *nums, int n){
    int left=0, right=n-1;
    while(left<right){
        int temp=nums[left];
        nums[left]=nums[right];
        nums[right]=temp;
        left++;
        right--;
    }
}

int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes){
    if(!root){ // 如果树为空, 返回空
        *returnSize=0;
        return NULL;
    }

    int **res=(int**)malloc(sizeof(int*)*1001); // 分配返回结果的内存
    *returnSize=0;
    *returnColumnSizes=(int**)malloc(sizeof(int)*1001); // 分配返回列大小的内存
  
```

```

struct TreeNode* queue[1100]; // 定义队列用于层序遍历
memset(queue, 0, 1100*sizeof(struct TreeNode*)); // 初始化队列
memset(*returnColumnSizes, 0, 1001*sizeof(int)); // 初始化列大小

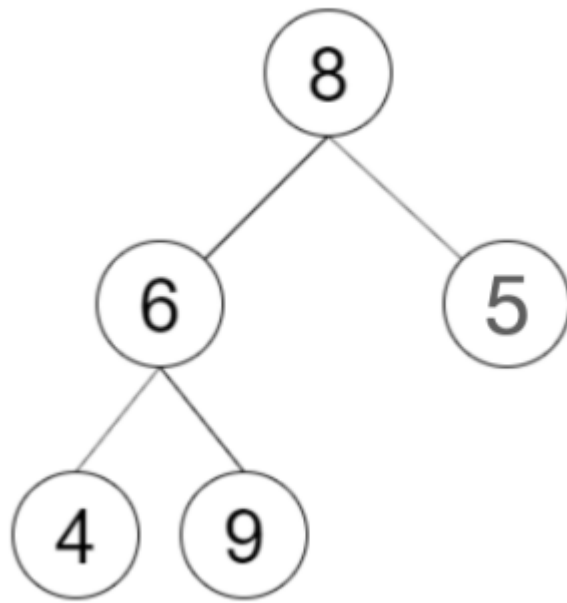
int front=0, rear=0; // 队列的前后指针
struct TreeNode* last=root; // 保存当前层的最后一个节点
queue[rear++]=root; // 根节点入队

int k=0; // 层数计数
while(front!=rear){
    int size=rear-front; // 当前层的节点数
    int *temp=(int*)malloc(sizeof(int)*size); // 临时数组保存当前层的节点值
    for(int i=0; i<size; i++){
        struct TreeNode* node=queue[front++]; // 从队列中取出节点
        temp[returnColumnSizes[0][*returnSize]++]=node->val; // 保存节点值
        if(node->left) queue[rear++]=node->left; // 左子节点入队
        if(node->right) queue[rear++]=node->right; // 右子节点入队
    }
    if(k%2){ // 如果当前层数为奇数, 反转该层节点值
        reverse(temp, size);
    }
    res[(*returnSize)++]=temp; // 保存当前层的结果
    k++;
}
return res; // 返回最终结果
}

```

### 33. 知后序遍历判是否为二叉搜索树

请实现一个函数来判断整数数组 `postorder` 是否为二叉搜索树的后序遍历结果



输入: postorder = [4,9,6,5,8]  
输出: false  
解释: 从上图可以看出这不是一颗二叉搜索树

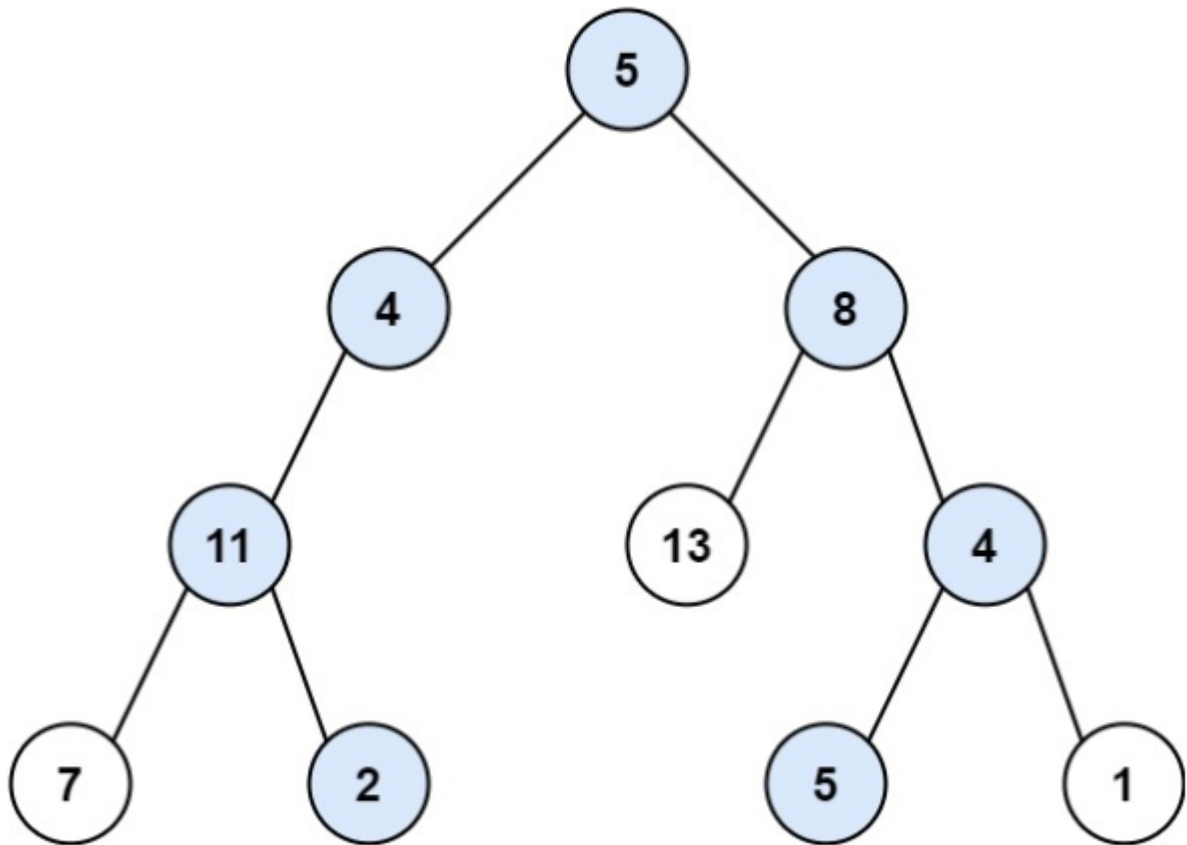
思路: 因为对于二叉搜索树来说就是右节点大于左节点, 而且数组最后一个值就是根节点  
因此我们可以先用一个i遍历整个后序遍历的数组, 如果找了一个数大于根节点跳出循环  
再用n从i下一个数开始遍历, 如果但凡出现一个数小于根节点的就说明他不是一颗二叉搜索树  
当i为0或者i遍历完了这一次就调用函数去判断那个我们已经判断好的左右子树返回结果

```
bool verifyTreeOrder(int* postorder, int postordersize) {  
    if(postordersize==0||postordersize==1)return true;  
    int i=0,j=0;  
    for(i=0;i<postordersize-1;i++)  
    {  
        if(postorder[i]>postorder[postordersize-1])break;  
    }  
    for(j=i;j<postordersize-1;j++)  
    {  
        if(postorder[j]<postorder[postordersize-1]){return false;}  
    }  
    if(i==0||i==postordersize-1)return verifyTreeOrder(postorder,postordersize-1);  
    else{  
        return verifyTreeOrder(postorder,i)&&verifyTreeOrder(&postorder[i],postordersize-i-1);  
    }  
}
```

## 34.二叉树中和为某一值的路径

给你二叉树的根节点 `root` 和一个整数目标和 `targetSum`，找出所有 **从根节点到叶子节点** 路径总和等于给定目标和的路径。

**叶子节点** 是指没有子节点的节点。



输入: `root = [5,4,8,11,null,13,4,7,2,null,null,5,1]`, `targetSum = 22`  
输出: `[[5,4,11,2],[5,8,4,5]]`

思路：需要一个二维结果数组存储和为`targetsum`的各个路径，需要一个路径数组存储路径（行），各路径的个数（列）。

开始干正事：两步走路径函数找和为`target`的路，主函数调用

判空为空，和怎么求，路径怎么记录。左右子树都不为空的时候，如果找到了和为`target`的路则定义一个`temp`数组遍历路径记录路径，然后在循环外让结果数据定义行和列。递推操作左右子树，恢复`sum`。

主函数的时候先定义结果数组行列和路径索引还有和，递归调用数组

```
int** Result = NULL;
int Result_Index = 0; //结果数组，放置和为target的各路径;

int Path[1000] = {0}; //该数组不需要动态分配，因此直接在全局变量处申请静态数组，极端示例需要1000，因此申请1000 (2023.5.14)
int Path_Index = 0;
int Sum = 0;
```



```

int* RecolSizes = NULL; //各路径的个数信息数组;

void Back_Track(struct TreeNode* root, int target){
    if(!root){
        return;
    } //如果节点为空, 则直接返回 (既不是叶子节点, 也不需要进行加和)

    Sum += root->val;
    Path[Path_Index] = root->val;
    Path_Index++; //该层的操作;

    if(!root->left && !root->right){
        if(Sum == target){
            int* temi = (int*)malloc(sizeof(int) * Path_Index);
            for(int i = 0; i < Path_Index; i++){
                temi[i] = Path[i];
            }
            Result[Result_Index] = temi;
            RecolSizes[Result_Index] = Path_Index;
            Result_Index++;
        }
    } //递推终止的条件: root && !root->left && !root->right;

    Back_Track(root->left, target);
    Back_Track(root->right, target); //递推操作;

    Sum -= root->val;
    Path_Index--; //回溯操作;
}

int** pathSum(struct TreeNode* root, int target, int* returnSize, int** returnColumnSizes){
    Result = (int**)malloc(sizeof(int*) * 290);
    RecolSizes = (int*)malloc(sizeof(int) * 290); //测试用例最大使用量290 (2023.5.14)
    Result_Index = 0;

    Path_Index = 0;
    Sum = 0;

    Back_Track(root, target);

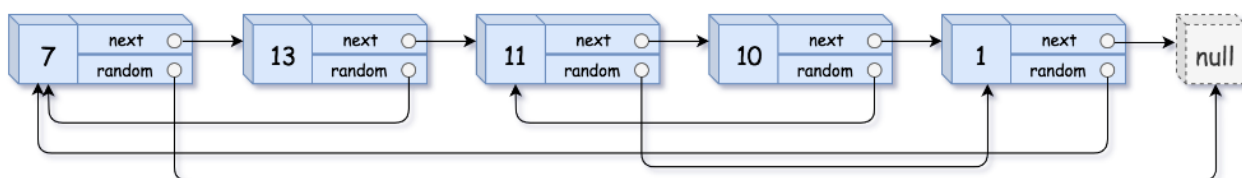
    *returnSize = Result_Index;
    *returnColumnSizes = RecolSizes;
    return Result;
}

```

## 35.复杂链表的复制 (\*)

请实现 `copyRandomList` 函数, 复制一个复杂链表。在复杂链表中, 每个节点除了有一个 `next` 指针指向下一个节点, 还有一个 `random` 指针指向链表中的任意节点或者 `null`。

示例 1:



输入: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

输出: [[7,null],[13,0],[11,4],[10,2],[1,0]]

用C++新建哈希表来完成

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if(head == nullptr) return nullptr;
        Node* cur = head;
        unordered_map<Node*, Node*> map;
        // 3. 复制各节点, 并建立 "原节点 -> 新节点" 的 Map 映射
        while(cur != nullptr) {
            map[cur] = new Node(cur->val);
            cur = cur->next;
        }
        cur = head;
        // 4. 构建新链表的 next 和 random 指向
        while(cur != nullptr) {
            map[cur]->next = map[cur->next];
            map[cur]->random = map[cur->random];
            cur = cur->next;
        }
        // 5. 返回新链表的头节点
        return map[head];
    }
};
```

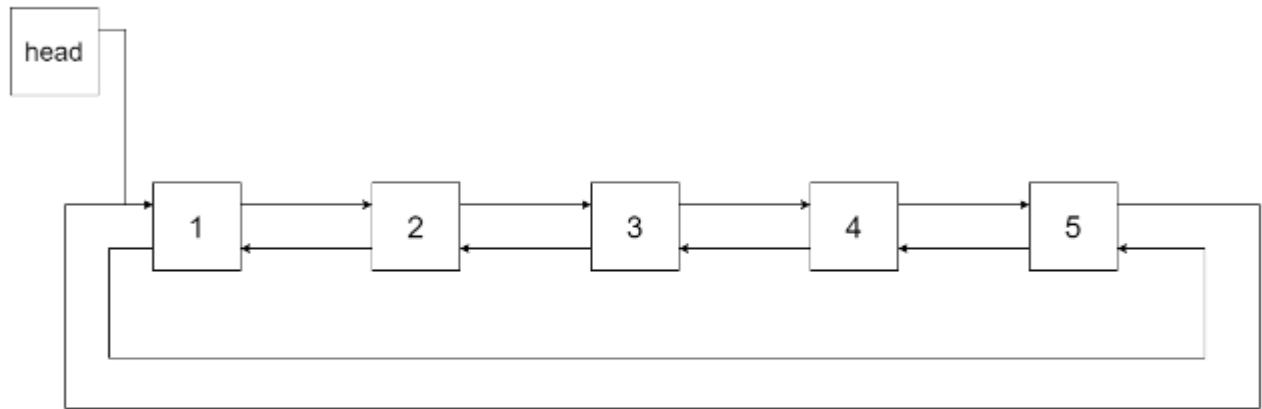
## 36.将二叉搜索数转化为排序的双向链表 (\*)

将一个 **二叉搜索树** 就地转化为一个 **已排序的双向循环链表**。

对于双向循环列表，你可以将左右孩子指针作为双向循环链表的前驱和后继指针，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

特别地，我们希望可以 **就地** 完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中最小元素的指针。

**输入:** root = [4,2,5,1,3]

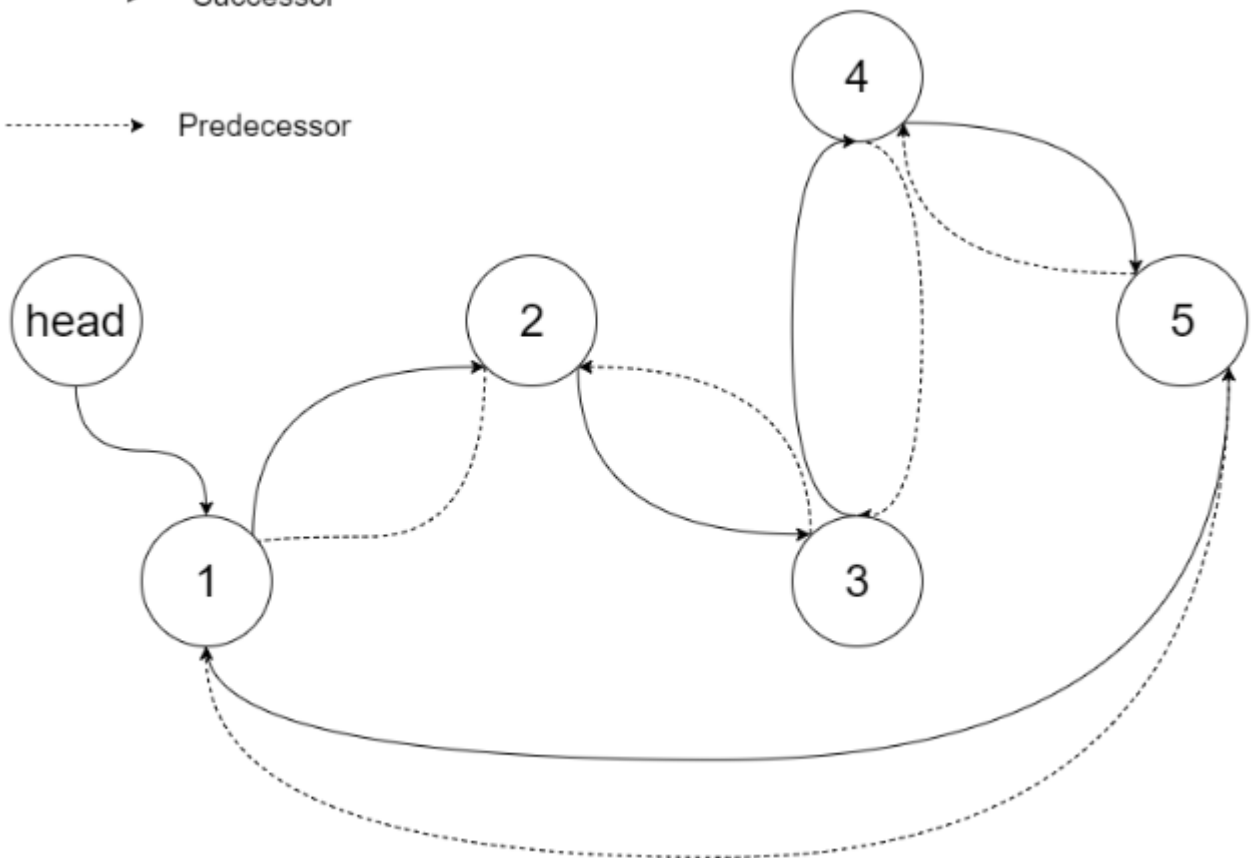


**输出:** [1,2,3,4,5]

**解释:** 下图显示了转化后的二叉搜索树，实线表示后继关系，虚线表示前驱关系。

————→ Successor

-----→ Predecessor

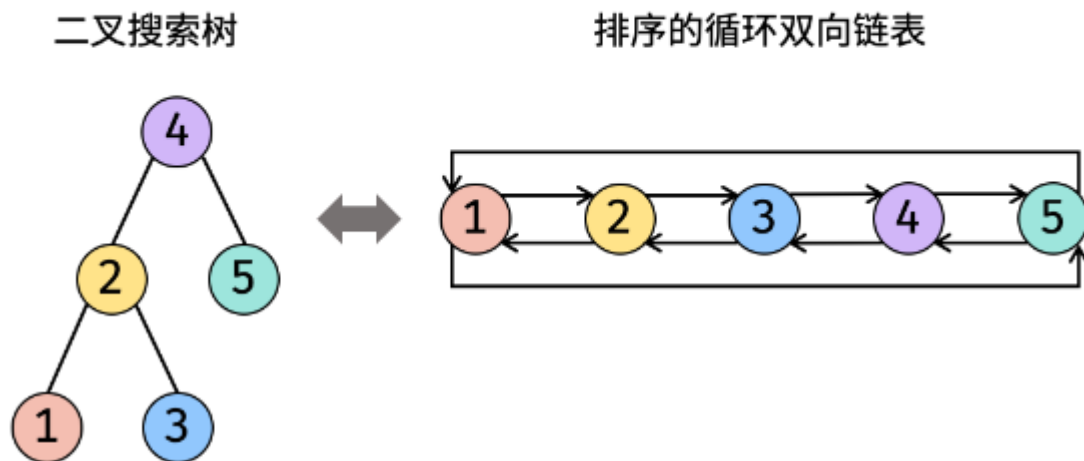


思路：中序遍历--双向链表--循环链表

因为要排序链表所以中序遍历二叉搜索树就是排序链表

在构建相邻节点的引用关系时，定义前驱节点为pre，当前节点为cur，不仅应构建pre.right=cur，也应该构建cur.left=pre。

定义链表头head和tail，最后应该要循环起来所以要加head.left=tail和tail.right=head。



排序：树的中序遍历

1	2	3	4	5
---	---	---	---	---

双向：不仅 ②.right = ③，还有 ③.left = ②

循环：⑤.right = ①，①.left = ⑤

```
class Solution {
public:
    Node* treeToDoublyList(Node* root) {
        if(root == nullptr) return nullptr; //判空
        dfs(root); //深度搜索调用函数
        head->left = pre; //为了双向链表准备的
        pre->right = head; //同上
        return head;
    }
private:
    Node *pre, *head;
    void dfs(Node* cur) { //定义深度优先搜索函数
        if(cur == nullptr) return; //判空
        dfs(cur->left); //先搞左边子树
        if(pre != nullptr) pre->right = cur;
        else head = cur;
        cur->left = pre;
        pre = cur;
        dfs(cur->right); //再搞右边子树
    }
};
```

## 37.序列化二叉树 (\*)

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

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

思路: 递归->深度优先搜索 (先中后序遍历)

先序遍历整棵树，而树是整型我们要转换成字符串型，对于空节点我们可以自定义一个字符来表示，而两个节点间的数据用，隔开。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

void dfs(struct TreeNode * root, char * str)
{
    if(root == NULL)//空节点, 用@代替
    {
        strcat(str, "@");
        return;
    }
    char s[7] = ""; //临时保存树元素值
    sprintf(s, "%d", root->val); //类型转换
    strcat(str, s); //加入字符串数组中
    dfs(root->left, str); //递归遍历左右子节点
    dfs(root->right, str);
    return;
}

/** Encodes a tree to a single string. */
char* serialize(struct TreeNode* root) {
    char * str = calloc(500000, sizeof(char)); //申请额外空间。并对其初始化为0
    dfs(root, str); //先序遍历
    return str;
}

struct TreeNode * dfsTree(char * data, int * inode) //先序遍历反构造树
{
    if(data[*inode] == '@') //空节点, 构造
    {
        (*inode) += 2; 跳过 '@和, '
        return NULL;
    }
}
```

```

}
char s[7] = ""; //临时保存元素值
int node = 0;
for(*inode; data[*inode] != ','; (*inode)++) //读取树元素值
{
    s[node++] = data[*inode];
}
(*inode)++; //跳过','
struct TreeNode * root = malloc(sizeof(struct TreeNode)); //构造节点
root->val = atoi(s); //元素转换
root->left = dfsTree(data, inode); //构造左右子节点
root->right = dfsTree(data, inode);
return root;
}
/** Decodes your encoded data to tree. */
struct TreeNode* deserialize(char* data) {
    int inode = 0;
    return dfsTree(data, &inode);
}

// Your functions will be called as such:
// char* data = serialize(root);
// deserialize(data);

```

## 38.字符串全排列（无顺序要求，无重复元素）

某店铺将用于组成套餐的商品记作字符串 `goods`，其中 `goods[i]` 表示对应商品。请返回该套餐内所含商品的 **全部排列方式**。

返回结果 **无顺序要求**，但不能含有重复的元素。

输入: `goods = "agew"`

输出:

`["aegw", "aewg", "agew", "agwe", "aweg", "awge", "eagw", "eawg", "egaw", "egwa", "ewag", "ewga", "gaew", "gawe", "geaw", "gewa", "gwae", "gwea", "waeg", "wage", "weag", "wega", "wgae", "wgea"]`

思路：可以把这个问题看作有  $n$  个空位，然后我们从左往右去填入题目给定的  $n$  个商品，每个商品只能使用一次。（回溯法）

定义一个回溯函数：`backtrack(i, perm)` 表示当前排列为 `perm`，下一个待填入的空位是 `i`

#第一种情况  $i=n$  说明我们填完了，那么就将 `perm` 放入结果数组里

#第二种情况  $i < n$  说明还没填完，没填完的话该填哪些元素呢，每个商品只能只用一次，所以可以定义一个标记数组 `vis` 来标记已经填过的商品。

```

void backtrack(char** rec, int* recSize, int* vis, char* goods, int i, int n, char* perm) {
    if (i == n) { //说明填完了
        char* tmp = malloc(sizeof(char) * (n + 1));
        strcpy(tmp, perm);
        rec[(*recSize)++] = tmp;
    }
}

```

```

        return;
    }
    for (int j = 0; j < n; j++) { //说明还没填完
        if (vis[j] || (j > 0 && !vis[j - 1] && goods[j - 1] == goods[j])) {
            continue; //判断该商品是否被标记过, 如果没标记过则继续
        }
        vis[j] = true; //标记
        perm[i] = goods[j]; //将元素填入空里
        backtrack(rec, recSize, vis, goods, i + 1, n, perm); //回溯看下一个元素
        vis[j] = false; //恢复标记
    }
}

int cmp(char* a, char* b) { //定义比较函数方便后续排序
    return *a - *b;
}

char** goodsorder(char* goods, int* returnSize) {
    int n = strlen(goods);
    int recMaxSize = 1;
    for (int i = 2; i <= n; i++) {
        recMaxSize *= i;
    }
    char** rec = malloc(sizeof(char*) * recMaxSize); //定义结果数组
    *returnSize = 0;
    int vis[n]; //定义标记数组
    memset(vis, 0, sizeof(vis));
    char perm[n + 1];
    perm[n] = '\0';
    qsort(goods, n, sizeof(char), cmp);
    backtrack(rec, returnSize, vis, goods, 0, n, perm);
    return rec;
}

```

### 39. 数组中出现次数超过一半的数字

仓库管理员以数组 `stock` 形式记录商品库存表。 `stock[i]` 表示商品 `id`，可能存在重复。请返回库存表中数量大于 `stock.length / 2` 的商品 `id`。

输入: `stock = [6, 1, 3, 1, 1, 1]`  
 输出: `1`

思路: 先让变量 `num` 初始化为 `stock[0]`，也就是例子中的 6，然后循环遍历数组

如果遇到相同的比如说 `num` 是 6 遍历第一次 `stock[0]`，他俩相等 `count++`，遍历第二次的时候 `num` 还是 6 但是 `stock[1]` 是 1，不相等了，那就让 `count--`。如果 `count` 为 0 了就让 `num` 等于 1，`count=1` 继续遍历。

```
int inventoryManagement(int* stock, int stockSize) {
    int count=1;
    int num=stock[0];

    for(int i=0;i<stockSize;i++)
    {
        if(stock[i]==num)count++;
        else{count--;if(count==0)num=stock[i],count=1;}
    }
    return num;
}
```

## 40.数组中最小的k个数

仓库管理员以数组 `stock` 形式记录商品库存表，其中 `stock[i]` 表示对应商品库存余量。请返回库存余量最少的 `cnt` 个商品余量，返回 **顺序不限**。

输入: `stock = [2,5,7,4]`, `cnt = 1`  
 输出: `[2]`

思路：我第一反应是排序然后输出前k个说干就干

排序肯定需要一个比较函数，定义！

然后开始干正事了，先判空如果输出最小的0个数或者数组长度为0，都为空

输出前k个数肯定要放到一个结果数组里，所以先动态分配一个结果数组，记得要判空

在这里有个小细节，复制stock数组避免修改原数组，所以我们要动态分配一个sortedstock数组并初始化，如果是空释放内存，返回长度为0，返空。遍历stock数组，赋值每一个元素给sortedstock数组。

我们对复制的数组来进行排序然后让i遍历到cnt，把前cnt个元素给结果数组，释放复制数组的内存。

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int* inventoryManagement(int* stock, int stockSize, int cnt, int* returnSize) {
    if (cnt == 0 || stockSize == 0) {
        *returnSize = 0;
        return NULL;
    }

    // 动态分配内存给结果数组
    int* arr = (int*)malloc(cnt * sizeof(int));
    if (arr == NULL) {
        *returnSize = 0;
        return NULL;
    }
}
```



```

// 复制 stock 数组以避免修改原数组
int* sortedStock = (int*)malloc(stockSize * sizeof(int));
if (sortedStock == NULL) {
    free(arr);
    *returnSize = 0;
    return NULL;
}
for (int i = 0; i < stockSize; i++) {
    sortedStock[i] = stock[i];
}

// 对复制的数组进行排序
qsort(sortedStock, stockSize, sizeof(int), compare);

// 取前 cnt 个最小的元素
for (int i = 0; i < cnt; i++) {
    arr[i] = sortedStock[i];
}

*returnSize = cnt;
free(sortedStock); // 释放临时数组的内存
return arr;
}

```

## 41.中位数

**中位数** 是有序整数列表中的中间值。如果列表的大小是偶数，则没有中间值，中位数是两个中间值的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是  $(2 + 3) / 2 = 2.5$

设计一个支持以下两种操作的数据结构：

- `void addNum(int num)` - 从数据流中添加一个整数到数据结构中。
- `double findMedian()` - 返回目前所有元素的中位数。

**示例 1:**

输入：

`["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]`

`[[],[1],[2],[],[3],[ ]]`

输出: `[null,null,null,1.50000,null,2.00000]`

本来的思路是先排序然后取中间值但是qsort会超时

用gnum存放数字，每加入新数字之前按顺序存储：当现在小于gcnt时且指针指向存进去，如果相等所有数字都比新数字小则放在最后，否则将后面的数字copy到放好之后。然后就可以按照中位数的概念放了。

```
typedef struct {
```

```

    int num; // 没什么用
} MedianFinder;
#define MAX_BUF 20000
#define DIV_NO 2
double g_num[MAX_BUF] = {0}; // 存放所有数字
int g_cnt = 0;
MedianFinder* medianFinderCreate() {
    MedianFinder *tmp = (MedianFinder *)calloc(1, sizeof(MedianFinder)); // 没用
    return tmp;
}

// 本题每次用qsort会超时，因此每次加入新数字前必须按顺序存储，查找时会很快
void medianFinderAddNum(MedianFinder* obj, int num) {
    int tmp = 0;
    while (tmp < g_cnt && g_num[tmp] < num) { // 因为全局数组按顺序存储，因此找到新数字应该存放的索引位置
        tmp++;
    }
    if (tmp == g_cnt) {
        g_num[g_cnt++] = (double)num; // 如果所有数字都比新数字小，则放在最后
    } else {
        double tmpNum[MAX_BUF] = {0};
        memcpy(tmpNum, &g_num[tmp], sizeof(double) * (g_cnt - tmp)); // 先将其后的数字拷贝出来
        g_num[tmp] = num; // 放入新数字
        memcpy(&g_num[tmp + 1], tmpNum, sizeof(double) * (g_cnt - tmp)); // 再将其后的数字拷贝回来
        g_cnt++;
    }
}

int Cmp(const void *a, const void *b)
{
    return *(double *)a - *(double *)b;
}

// 本题每次用qsort会超时，因此每次加入新数字前必须按顺序存储，查找时会很快
double medianFinderFindMedian(MedianFinder* obj) {
    if ((g_cnt % DIV_NO) == 0 && g_cnt > 1) { // 如果数字个数大于1且是2的整数倍则取中间2个数字取平均
        return (double)((g_num[g_cnt / DIV_NO - 1] + g_num[g_cnt / DIV_NO]) / DIV_NO);
    } else {
        if (g_cnt == 1) {
            return (double)g_num[0]; // 如果只有一个数字返回即可
        }
        return (double)g_num[g_cnt / DIV_NO]; // 否则取奇数数字个数的中间一个即可
    }
    return 0;
}

void medianFinderFree(MedianFinder* obj) {
    memset(g_num, 0, sizeof(g_num));
    g_cnt = 0;
    free(obj);
}

```

## 42.连续子数组的最大和

某公司每日销售额记于整数数组 `sales`，请返回所有 **连续** 一或多天销售额总和的最大值。

要求实现时间复杂度为  $O(n)$  的算法。

示例 1:

输入: `sales = [-2,1,-3,4,-1,2,1,-5,4]`  
输出: 6  
解释: `[4,-1,2,1]` 此连续四天的销售总额最高，为 6

思路：动态规划嘛，跟抢劫是一个思路，就是前面的和与加上当前数的和取大的那个

```
int maxSales(int* sales, int salesSize) {  
    int pre = 0, maxAns = sales[0];  
    for (int i = 0; i < salesSize; i++) {  
        pre = fmax(pre + sales[i], sales[i]);  
        maxAns = fmax(maxAns, pre);  
    }  
    return maxAns;  
}
```

## 43.数字1的个数

给定一个整数 `num`，计算所有小于等于 `num` 的非负整数中数字 `1` 出现的个数。

示例 1:

输入: `num = 0`  
输出: 0

思路:

每一位的1出现的次数相加，例如521求1的个数：对于个位1来说1的个数是1，对于21来说1的个数是  $2*1+10+1=13$ ，对于521来说1的个数是  $5*20+100+13$

一共就可能到10次方，公式就出来了：`res+record【count】*num`

```
int countDigitOne(int n){  
    int record[] = {0, 1, 20, 300, 4000, 50000, 600000, 7000000, 80000000, 900000000};  
    int count = 0;  
    int res = 0;  
    long flag = 1;  
    int tmp = n;  
    while(tmp != 0){  
        int num = tmp % 10;  
        tmp /= 10;  
        res = res + record[count] * num;  
        if(num > 1){
```

```
        res += flag;
    }else if(num == 1){
        res += n % flag + 1;
    }
    count++;
    flag *= 10;
}
return res;
}
```

## 45.数学序列中的某一位数字

某班级学号记录系统发生错乱，原整数学号序列 `[0,1,2,3,4,...]` 分隔符丢失后变为 `01234...` 的字符序列。请实现一个函数返回该字符序列中的第 `k` 位数字。

**示例 1:**

输入: `k = 5`  
输出: `5`

**示例 2:**

输入: `k = 12`  
输出: `1`  
解释: 第 12 位数字在序列 `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...` 里是 `1`，它是 `11` 的一部分。

思路:

1. 将  $101112\dots$  中的每一位称为 **数位**，记为  $k$ ；
2. 将  $10, 11, 12, \dots$  称为 **数字**，记为  $num$ ；
3. 数字  $10$  是一个两位数，称此数字的 **位数** 为  $2$ ，记为  $digit$ ；
4. 每  $digit$  位数的起始数字（即： $1, 10, 100, \dots$ ），记为  $start$ ；

数字范围	位数	数字数量	数位数量
1~9	1	9	9
10~99	2	90	180
100~999	3	900	2700
...	...	...	...
start~end	digit	$9 \times start$	$9 \times start \times digit$



位数递推公式  $digit = digit + 1$   
 起始数字递推公式  $start = start \times 10$   
 数位数量计算公式  $count = 9 \times start \times digit$

也就是说我们先确定  $k$  所在的数字的位数记为  $digit$ ，所在的数字几位  $num$ ，确定  $k$  是  $num$  的哪一位返回结果

```

#include<stdio.h>
#include<stdlib.h>
int findkthnumber(int k)
{
    int digit=1;
    long start=1;
    long count=9;
    while(k>count)
    {
        k-=count;
        start*=10;
    }
}

```

```

        digit+=1;
        count=digit*start*9;
    }
    long num=start+(k-1)/digit;
    long res=(k-1)%digit;
    return res;
}

```

## 45.把数组排成最小的数

闯关游戏需要破解一组密码，闯关组给出的有关密码的线索是：

- 一个拥有密码所有元素的非负整数数组 `password`
- 密码是 `password` 中所有元素拼接后得到的最小的一个数

请编写一个程序返回这个密码。

**示例 1:**

```

输入：password = [15, 8, 7]
输出："1578"

```

思路：

定义比较函数为了方便后续排序，但这个比较函数要加上合并两个字符串的功能

动态定义res数组和指针，排序之后遍历。

```

int compare(const void *a, const void *b)
{
    char num1[24];
    char num2[24];

    sprintf(num1, "%d%d", *(int *)a, *(int *)b);
    sprintf(num2, "%d%d", *(int *)b, *(int *)a);
    return strcmp(num1, num2);
}

char* minNumber(int* nums, int numsSize){
    char *res = (char *)malloc(sizeof(char) * 1200);
    char *p = res;

    qsort(nums, numsSize, sizeof(int), compare);
    for (int i = 0; i < numsSize; i++)
        p += sprintf(p, "%d", nums[i]);
    *p = '\0';
    return res;
}

```

## 46.把数字翻译成字符串

现有一串神秘的密文 `ciphertext`，经调查，密文的特点和规则如下：

- 密文由非负整数组成
- 数字 0-25 分别对应字母 a-z

请根据上述规则将密文 `ciphertext` 解密为字母，并返回共有多少种解密结果。

输入：ciphertext = 216612

输出：6

解释：216612 解密后有 6 种不同的形式，分别是 "cbggbc", "vggbc", "vggm", "cbggm", "cqqgbc" 和 "cqqgm"

思路：

这道题就是跳台阶进阶版

跳台阶的递推方程是 $dp[i]=dp[i-2]+dp[i-1]$

但我们只考虑数字大于9的时候的情况，如果大于9我们将每一位都变成单独的个位数存进nums里，添加剩余的数字，因为我们存进nums的时候是倒序排列的所以要用双指针交换三步走翻转一下开始搞dp，剩余部分的动态规划

```
int translateNum(int num){
    int* nums = (int *) malloc (sizeof(int) * 20), numSize = 0;
    int* dp = (int *) malloc (sizeof(int) * 20);

    while (num > 9) {
        nums[numSize++] = num % 10;
        num = (num - num % 10) / 10;
    }
    nums[numSize++] = num;
    for(int left = 0, right = numSize - 1; left < right; left++, right--) {
        int tmp = nums[left]; nums[left] = nums[right]; nums[right] = tmp;
    }

    dp[0] = 1;
    dp[1] = (nums[0] * 10 + nums[1] < 26 && nums[0] * 10 + nums[1] > 9) ? dp[0] + 1 : dp[0];
    for(int i = 2; i < numSize; i++)
        dp[i] = (nums[i - 1] * 10 + nums[i] < 26 && nums[i - 1] * 10 + nums[i] > 9) ? dp[i - 2] + dp[i - 1] : dp[i - 1];
    return dp[numSize - 1];
}
```

## 47.路径的最大价值\*\*\*

现有一个记作二维矩阵 `frame` 的珠宝架，其中 `frame[i][j]` 为该位置珠宝的价值。拿取珠宝的规则为：

- 只能从架子的左上角开始拿珠宝
- 每次可以移动到右侧或下侧的相邻位置
- 到达珠宝架子的右下角时，停止拿取

注意：珠宝的价值都是大于 0 的。除非这个架子上没有任何珠宝，比如 `frame = [[0]]`。

#### 示例 1:

输入: `frame = [[1,3,1],[1,5,1],[4,2,1]]`

输出: 12

解释: 路径 1→3→5→2→1 可以拿到最高价值的珠宝

思路：用动态规划求最大和，写出往右移动和往下移动的状态方程也就是将求最大的f注意不要出界。

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int maxValue(int** grid, int gridSize, int* gridColSize) {
    int m = gridSize, n = gridColSize[0];
    int f[m][n];
    memset(f, 0, sizeof(f));
    for (int i = 0; i < m; ++i) { //遍历行
        for (int j = 0; j < n; ++j) { //遍历列
            if (i > 0) {
                f[i][j] = MAX(f[i][j], f[i - 1][j]); //向右走
            }
            if (j > 0) {
                f[i][j] = MAX(f[i][j], f[i][j - 1]); //向下走
            }
            f[i][j] += grid[i][j];
        }
    }
    return f[m - 1][n - 1];
}
```

## 48.最长不含重复字符的子字符串

某套连招动作记作序列 `arr`，其中 `arr[i]` 为第 `i` 个招式的名字。请返回 `arr` 中最多可以出连续不重复的多少个招式。

#### 示例 1:

输入: `arr = "dbascDdad"`

输出: 6

解释: 因为连续且最长的招式序列是 `"dbascD"` 或 `"bascDd"`，所以其长度为 6。

#### 示例 2:

输入: `arr = "KKK"`

输出: 1

解释: 因为无重复字符的最长子串是 `"K"`，所以其长度为 1。

#### 示例 3:



输入: arr = "pwwkew"

输出: 3

解释: 因为连续且最长的招式序列是 "wke", 所以其长度为 3。

请注意区分 子串 与 子序列 的概念: 你的答案必须是 连续招式 的长度, 也就是 子串。而 "pwke" 是一个非连续的 子序列, 不是 子串。

思路: 利用数字储存字符最后出现的位置, 默认都没出现-1初始化状态, 遍历字符串

```
int lengthOfLongestSubstring(char* s){
    int len = strlen(s);
    int max = 0, temp = 0;
    int appear[128]; // 定义数组存储字符最后出现的位置 (ASCII中的所有字符)
    for (int i = 0; i < 128; i++)
    {
        appear[i] = -1; // 默认将所有字符的显示次数设为-1, 即默认都未出现
    }

    for (int j = 0; j < len; j++)
    {
        int i = appear[s[j]];
        appear[s[j]] = j; // 存储字符最后出现的位置
        temp = temp < j - i ? temp + 1 : j - i; // 状态转移方程 dp[j - 1] -> dp[j], 更新当前字符串商都, 如果当前字符串长度temp小于当前字符到上一次出现该字符位置的距离则temp+1, 表示当前字符没有重复, 可以延续, 否则temp更新为j-i, 表示重复了需要重新计算。
        max = max > temp ? max : temp; // max(dp[j - 1], dp[j]), 更新最大字符串商都, 如果temp+1了就会比max大, 这个时候就更新max
    }

    return max;
}
```

## 49.丑数

给你一个整数 **n** , 请你找出并返回第 **n** 个丑数。

**说明:** 丑数是只包含质因数 2、3 和/或 5 的正整数; 1 是丑数。

**示例 1:**

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。

思路: 定义一个数组长度为n+1, 定义三个指针p2,p3,p5,遍历数组定义丑数是当前指针指向的抽数乘以对应的质因数 num2, num3, num5

```
int nthUglyNumber(int n) {
    int dp[n + 1];
    dp[1] = 1;
    int p2 = 1, p3 = 1, p5 = 1;
```

```

for (int i = 2; i <= n; i++) {
    int num2 = dp[p2] * 2, num3 = dp[p3] * 3, num5 = dp[p5] * 5;
    dp[i] = fmin(fmin(num2, num3), num5);
    if (dp[i] == num2) {
        p2++;
    }
    if (dp[i] == num3) {
        p3++;
    }
    if (dp[i] == num5) {
        p5++;
    }
}
return dp[n];
}

```

## 50.第一次只出现一次的字符

某套连招动作记作仅由小写字母组成的序列 `arr`，其中 `arr[i]` 第 `i` 个招式的名字。请返回第一个只出现一次的招式名称，如不存在请返回空格。

示例 1:

输入: `arr = "abbccdeff"`  
 输出: `'a'`

示例 2:

输入: `arr = "ccdd"`  
 输出: `' '`

思路：用一个数组存储每一个字母出现的次数，然后遍历一边字符串，如果在对应的储存数字里出现一次直接返回该字符

```

char firstUniqChar(char* s){
    int i,hash[26]={0};
    for(i=0;i<strlen(s);i++){
        hash[s[i]-'a']++;
    }
    for(i=0;i<strlen(s);i++){
        if(hash[s[i]-'a']==1)
            return s[i];
    }
    return ' ';
}

```

## 51.逆序对总数

在股票交易中，如果前一天的股价高于后一天的股价，则可以认为存在一个「交易逆序对」。请设计一个程序，输入一段时间内的股票交易记录 `record`，返回其中存在的「交易逆序对」总数。

#### 示例 1:

输入: `record = [9, 7, 5, 4, 6]`

输出: 8

解释: 交易中的逆序对为 (9, 7), (9, 5), (9, 4), (9, 6), (7, 5), (7, 4), (7, 6), (5, 4)。

思路: 归并排序类似于二叉树的方法

```
// 归并排序 (二路归并)
int mergeSort(int l, int r, int *nums, int *tmp)
{
    // 退出条件
    if (l >= r)
        return 0;

    // 递归划分
    int m = (l + r) / 2;
    int res = mergeSort(l, m, nums, tmp) + mergeSort(m + 1, r, nums, tmp);

    // 合并阶段
    int i = l, j = m + 1;
    for (int k = l; k <= r; k++) // 注意k的边界 <=
    {
        tmp[k] = nums[k];
    }
    for (int k = l; k <= r; k++) // 注意k的边界 <=
    {
        if (i == m + 1)
        { // 左子数组合并结束，将右边剩余元素加入数组
            nums[k] = tmp[j++];
        }
        else if (j == r + 1 || tmp[i] <= tmp[j])
        { // 右边数组合并完，或右边>左边，将元素加入结果数组
            nums[k] = tmp[i++];
        }
        else
        { // 左边元素>右边元素，此时将贡献逆序对
            nums[k] = tmp[j++];
            res += m - i + 1; // 统计逆序数
        }
    }

    return res;
}

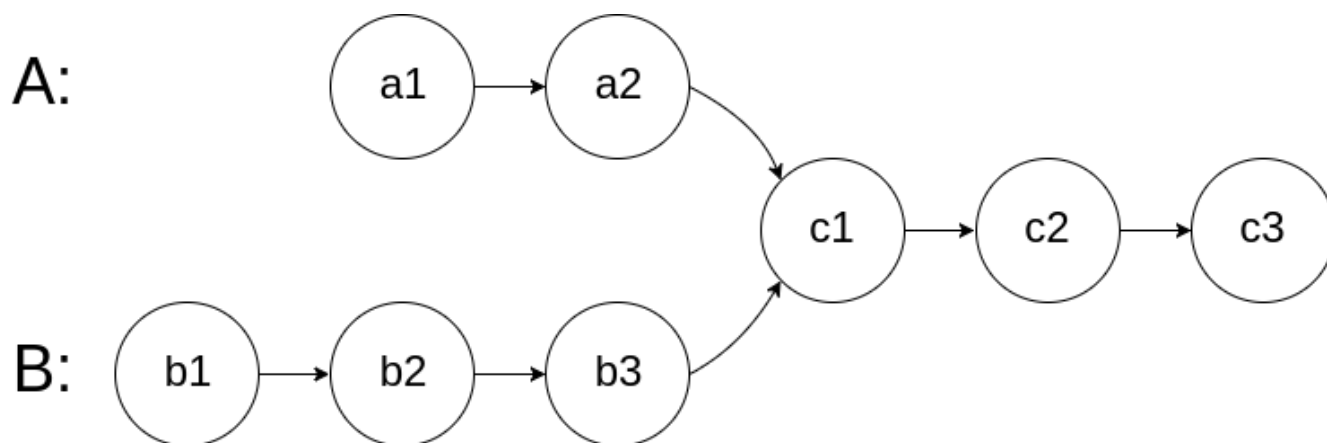
int reversePairs(int *nums, int numsSize)
{
    int *tmp = malloc(sizeof(int) * numsSize);
    return mergeSort(0, numsSize - 1, nums, tmp);
}
```

```
}
```

## 52.两个链表中的第一个

某教练同时带教两位学员，分别以链表 `l1`、`l2` 记录了两套核心肌群训练计划，节点值为训练项目编号。两套计划仅有前半部分热身项目不同，后续正式训练项目相同。请设计一个程序找出并返回第一个正式训练项目编号。如果两个链表不存在相交节点，返回 `null`。

如下面的两个链表：



在节点 `c1` 开始相交。

输入说明：

`intersectVal` - 相交的起始节点的值。如果不存在相交节点，这一值为 0

`l1` - 第一个训练计划链表

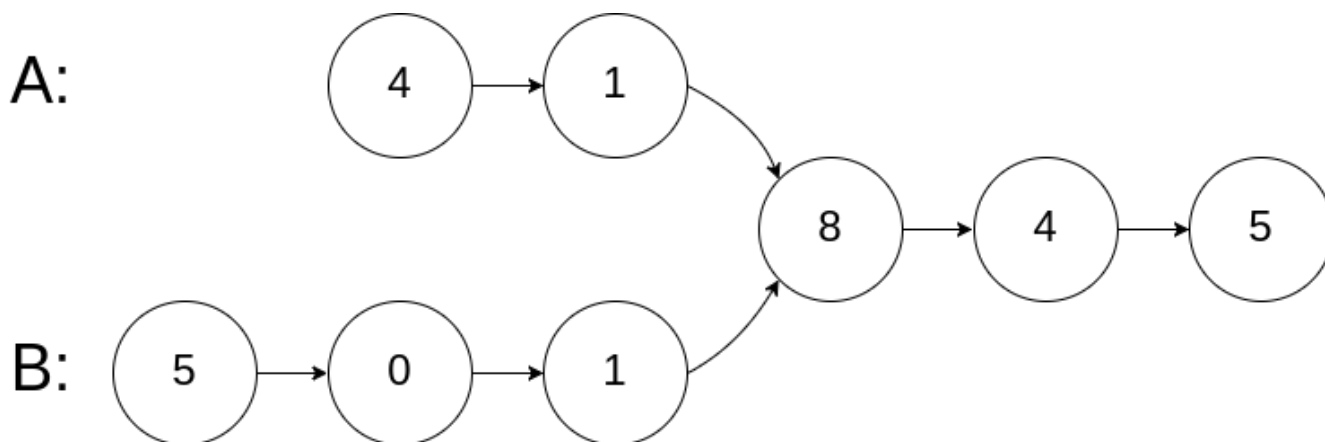
`l2` - 第二个训练计划链表

`skip1` - 在 `l1` 中（从头节点开始）跳到交叉节点的节点数

`skip2` - 在 `l2` 中（从头节点开始）跳到交叉节点的节点数

程序将根据这些输入创建链式数据结构，并将两个头节点 `head1` 和 `head2` 传递给你的程序。如果程序能够正确返回相交节点，那么你的解决方案将被视作正确答案。

**示例 1：**



输入: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

输出: Reference of the node with value = 8

解释: 第一个正式训练项目编号为 8 (注意, 如果两个列表相交则不能为 0)。从各自的表头开始算起, 链表 A 为 [4,1,8,4,5], 链表 B 为 [5,0,1,8,4,5]。在 A 中, 相交节点前有 2 个节点; 在 B 中, 相交节点前有 3 个节点。

思路: 两个表连接起来当节点相同, 指针所指的就是相交的第一个节点

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
    struct ListNode* p1 = headA;
    struct ListNode* p2 = headB;
    while (p1 != p2) {
        if (p1 == NULL) {
            p1 = headB;
        } else {
            p1 = p1->next;
        }
        if (p2 == NULL) {
            p2 = headA;
        } else {
            p2 = p2->next;
        }
    }
    return p1;
}

```

## 53.在排序数组中查找

某班级考试成绩按非严格递增顺序记录于整数数组 `scores`, 请返回目标成绩 `target` 的出现次数。

### 示例 1:

输入: scores = [2, 2, 3, 4, 4, 4, 5, 6, 6, 8], target = 4  
输出: 3

### 示例 2:

输入: scores = [1, 2, 3, 5, 7, 9], target = 6  
输出: 0

思路: 直接遍历数组然后遇到target, 设置变量count++, 最后返回count

```
#include <stdio.h>

int countTargetOccurrences(int* scores, int scoresSize, int target) {
    int count = 0;
    for (int i = 0; i < scoresSize; i++) {
        if (scores[i] == target) {
            count++;
        }
    }
    return count;
}
```

也可以用二分法来做

```
int binarySearch(int* nums, int numssize, int target, bool lower) {
    int left = 0, right = numssize - 1, ans = numssize;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (nums[mid] > target || (lower && nums[mid] >= target)) {
            right = mid - 1;
            ans = mid;
        } else {
            left = mid + 1;
        }
    }
    return ans;
}

int countTarget(int* scores, int numssize, int target) {
    int leftIdx = binarySearch(scores, numssize, target, true);
    int rightIdx = binarySearch(scores, numssize, target, false) - 1;
    int ret = 0;
    if (leftIdx <= rightIdx && rightIdx < numssize && scores[leftIdx] == target &&
        scores[rightIdx] == target) {
        ret = rightIdx - leftIdx + 1;
    }
    return ret;
}
```

---

## 53.2.0~n-1中缺失的数字

某班级  $n$  位同学的学号为  $0 \sim n-1$ 。点名结果记录于升序数组 `records`。假定仅有一位同学缺席，请返回他的学号。

示例 1:

输入: `records = [0,1,2,3,5]`  
输出: 4

示例 2:

输入: `records = [0, 1, 2, 3, 4, 5, 6, 8]`  
输出: 7

思路: 本来就是升序数组, 那就直接如果下标不等于对应的值就说明他缺席

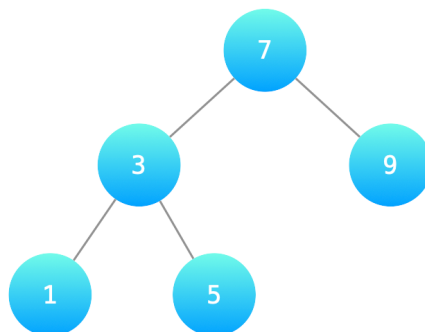
```
int takeAttendance(int* records, int recordsSize) {  
    for (int i = 0; i < recordsSize; ++ i)  
        if (i != records[i]) return i;  
    return recordsSize;  
}
```

---

## 54.寻找二叉搜索树中的目标节点

某公司组织架构以二叉搜索树形式记录, 节点值为处于该职位的员工编号。请返回第 `cnt` 大的员工编号。

示例 1:



输入: root = [7, 3, 9, 1, 5], cnt = 2

```
      7
     /\
    3  9
   /\
  1  5
```

输出: 7

思路: 二叉搜索树的中序遍历就是升序, 遍历到size-cnt就输出

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
void inorder(struct TreeNode *root, int *temp, int *size){
    if(root == NULL){
        return;
    }
    inorder(root -> left, temp, size);
    temp[(*size)++] = root -> val;
    inorder(root -> right, temp, size);
}

int findTargetNode(struct TreeNode* root, int cnt) {
    int *temp = (int*)malloc(sizeof(int)*10000);
    int size = 0;
    inorder(root, temp, &size);
    return temp[size - cnt];
}
```

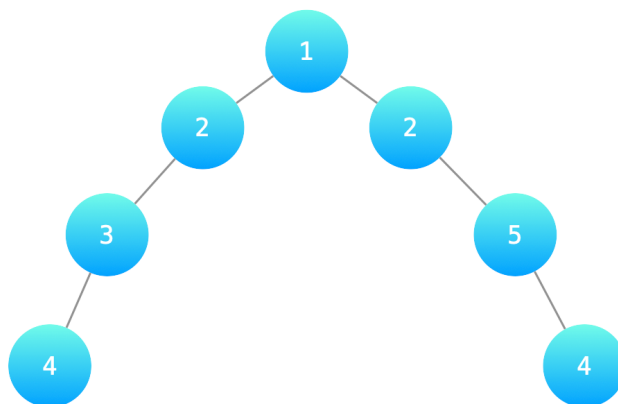
---

## 55. 二叉树的深度

某公司架构以二叉树形式记录, 请返回该公司的层级数。

示例 1:





输入: root = [1, 2, 2, 3, null, null, 5, 4, null, null, 4]

输出: 4

解释: 上面示例中的二叉树的最大深度是 4, 沿着路径 1 -> 2 -> 3 -> 4 或 1 -> 2 -> 5 -> 4 到达叶节点的最长路径上有 4 个节点。

思路: 树为空深度为0; 只有根节点深度为1; 只有左子树, 深度为左子树深度+1; 只有右子树, 深度为右子树深度+1; 既有左子树又有右子树, 深度为左右子树深度的谁大谁+1

```
int maxDepth(struct TreeNode* root){
    if (root == NULL) {
        return 0;
    }

    /* 当前节点的左子树的深度 */
    int lenLeft = maxDepth(root->left);

    /* 当前节点的右子树的深度 */
    int lenRight = maxDepth(root->right);

    /* 二叉树的深度等于左右子树深度的的较大者加 1 (当前节点的深度) */
    return lenLeft > lenRight ? lenLeft + 1 : lenRight + 1;
}
```

## 55.2.平衡二叉树

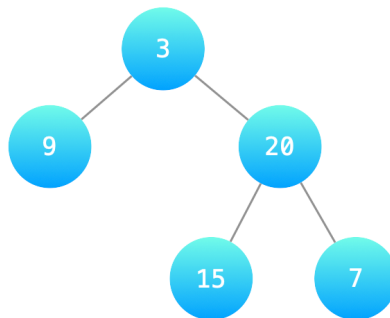
输入一棵二叉树的根节点, 判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1, 那么它就是一棵平衡二叉树。

### 示例 1:

输入: root = [3,9,20,null,null,15,7]

输出: true

解释: 如下图



思路: 计算左右子树的深度, 两者相减小于1就是平衡, 判断每个节点

```
int height(struct TreeNode* root) {
    if (root == NULL) {
        return 0;
    } else {
        return fmax(height(root->left), height(root->right)) + 1;
    }
}

bool isBalanced(struct TreeNode* root) {
    if (root == NULL) {
        return true;
    } else {
        return fabs(height(root->left) - height(root->right)) <= 1 && isBalanced(root->left) && isBalanced(root->right);
    }
}
```

## 56.数组中数字出现的次数

整数数组 `sockets` 记录了一个袜子礼盒的颜色分布情况，其中 `sockets[i]` 表示该袜子的颜色编号。礼盒中除了一款撞色搭配的袜子，每种颜色的袜子均有两只。请设计一个程序，在时间复杂度  $O(n)$ ，空间复杂度  $O(1)$  内找到这双撞色搭配袜子的两个颜色编号。

#### 示例 1:

输入: `sockets = [4, 5, 2, 4, 6, 6]`

输出: `[2,5]` 或 `[5,2]`

思路: 通过异或的知识点做

```
int* singleNumbers(int* nums, int numsSize, int* returnSize){
    int ret = 0;
    int i = 0;
    int num1 = 0;
    int num2 = 0;
    int pos = 0;
    int *arr = (int *)malloc(2 * sizeof(int)); //要动态开辟内存空间存储返回值，返回类型为int*，
    不能使用临时数组
    //通过0对自身依次异或，将两个单独的数的异或后的结果找出，再通过结果将两个值分离开来
    for(i = 0; i < numsSize; i++)
    {
        ret ^= nums[i];
    }
    //找到异或后结果的二进制序列，找出为1的位数（如果为1，则说明两个单独数在这个位不相同，可凭此将两个
    数分离）
    for(i = 0; i < 32; i++)
    {
        if((ret >> i) & 1 == 1)
        {
            pos = i;
            break;
        }
    }
    //找到位数后，将按其他数的这个位数是否为1，还是为0分离，这就形成了两组数
    //各带一个单独数

    //在分别自身异或，最后剩下的就是单独数
    for(i = 0; i < numsSize; i++)
    {
        if((nums[i] >> pos) & 1 == 1)
        {
            num1 ^= nums[i];
        }
        else
        {
            num2 ^= nums[i];
        }
    }
    arr[0] = num1;
    arr[1] = num2;
    *returnSize = 2;
}
```

```
        return arr;
    }
```

## 56.2.数组中数字出现的次数

教学过程中，教练示范一次，学员跟做三次。该过程被混乱剪辑后，记录于数组 `actions`，其中 `actions[i]` 表示做出该动作的人员编号。请返回教练的编号。

示例 1:

输入: `actions = [5, 7, 5, 5]`  
输出: 7

思路：把每个数字的每一位相加起来并且%3，因为只有一个数字出现一次其他数字出现三次，所以取模后会剩下只出现一次的数字

```
int singleNumber(int* nums, int numSize)
{
    int ans = 0;
    int bit = 0;
    for(int i = 31; i >= 0; i--)
    {
        for(int j = 0; j < numSize; j++)
        {
            bit += nums[j] >> i & 1;
        }
        ans = ans * 2 + bit % 3;
        bit = 0;
    }
    return ans;
}
```

## 57.和为s的两个数字

购物车内的商品价格按照升序记录于数组 `price`。请在购物车中找到两个商品的价格总和刚好是 `target`。若存在多种情况，返回任一结果即可。

示例 1:

输入: `price = [3, 9, 12, 15]`, `target = 18`  
输出: `[3,15]` 或者 `[15,3]`

思路：可以采用双指针，一左一右，如果和大于target让右边--，如果小于让左边++

```
int* twoSum(int* nums, int numSize, int target, int* returnSize){
    int left=0,right=numSize-1; //左右指针
```

```

int *res=(int *)malloc(2*sizeof(int));
*returnSize=2;
while(1){
    if(nums[left]+nums[right]>target)
        right--;
    else if(nums[left]+nums[right]<target)
        left++;
    else
    {
        res[0]=nums[left];
        res[1]=nums[right];
        return res;
    }
}

return;
}

```

## 57.2.和为s的连续正数序列

待传输文件被切分成多个部分，按照原排列顺序，每部分文件编号均为一个 **正整数**（至少含有两个文件）。传输要求为：连续文件编号总和为接收方指定数字 `target` 的所有文件。请返回所有符合该要求的文件传输组合列表。

**注意**，返回时需遵循以下规则：

- 每种组合按照文件编号 **升序** 排列；
- 不同组合按照第一个文件编号 **升序** 排列

**示例 1：**

输入：target = 12

输出：[[3, 4, 5]]

解释：在上述示例中，存在一个连续正整数序列的和为 12，为 [3, 4, 5]。

思路：利用双指针遍历1到n/2查找，其实就是滑动窗口

```

/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller calls free().
 */
int** findContinuousSequence(int target, int* returnSize, int** returnColumnSizes) {
    if (target < 1) {
        return NULL;
    }
    int** res = (int**)malloc(sizeof(int*) * target);
    *returnColumnSizes = (int*)malloc(sizeof(int) * target);
    int left = 1, right = 1, sum = 0, len = target / 2;
    *returnSize = 0;

```

```

while (left <= len) {
    if (sum == target) {
        (*returnColumnSizes)[*returnSize] = right - left;
        res[*returnSize] = (int*)malloc(sizeof(int) * (right - left));
        for (int i = left; i < right; i++) {
            res[*returnSize][i - left] = i;
        }
        (*returnSize)++;
        sum -= left++;
    }
    if (sum < target) {
        sum += right++;
    }
    if (sum > target){
        sum -= left++;
    }
}
return res;
}

```

## 58.翻转单词顺序

你在与一位习惯从右往左阅读的朋友发消息，他发出的文字顺序都与正常相反但单词内容正确，为了和他顺利交流你决定写一个转换程序，把他所发的消息 `message` 转换为正常语序。

注意：输入字符串 `message` 中可能会存在前导空格、尾随空格或者单词间的多个空格。返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。

### 示例 1:

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

思路：去掉空格，将整体字符串翻转，再逐一将单词反转即可

```

void swap(char *s,int st,int ed)
{
    int len=ed-st+1;
    for(int i=st;i<st+len/2;i++)
    {
        char c=s[ed-(i-st)];
        s[ed-(i-st)]=s[i];
        s[i]=c;
    }
}

char* reverseWords(char* s){
    int n=strlen(s),pre=0,last=n-1,idx=0,st=0;
    if(n==0) return s;
    char *str=(char*)malloc(sizeof(char)*(n+1));

```

```

while(pre<n&&s[pre]==' ') pre++;
while(last>=0&&s[last]==' ') last--;
if(pre==n||last==-1) return "";
for(int i=pre;i<=last;i++)
{
    if(s[i]!=' ') str[idx++]=s[i];
    else
    {
        if(i-1>0&&s[i-1]==' ') continue;
        else str[idx++]=s[i];
    }
}
str[idx]='\0';
int len=strlen(str);
swap(str,0,len-1);
//printf("%s\n",str);
for(int i=0;i<len;i++)
{
    if(str[i]==' '||i==len-1)
    {
        if(i==len-1) swap(str,st,i);
        else swap(str,st,i-1);
        if(i==len-1) break;
        while(i<len&&str[i]==' ') i++;
        st=i;
        i-=1;
        //printf("%s\n",str);
    }
}
return str;
}

```

## 58.2.左旋转字符串

某公司门禁密码使用动态口令技术。初始密码为字符串 `password`，密码更新均遵循以下步骤：

- 设定一个正整数目标值 `target`
- 将 `password` 前 `target` 个字符按原顺序移动至字符串末尾

请返回更新后的密码字符串。

**示例 1：**

输入：password = "s3cur1tyC0d3", target = 4  
输出："r1tyC0d3s3cu"

思路：申请一个数组按照要求填入字符

```

char* reverseLeftwords(char* s, int n){
    int len = strlen(s);

```

```

char* result = malloc(sizeof(char) * len + 1);
int count = 0;
for (int i = n; i < len; i++) {
    result[count++] = s[i];
}
for (int i = 0; i < n; i++) {
    result[count++] = s[i];
}
result[count] = '\0';
return result;
}

```

## 59.滑动窗口的最大值

科技馆内有一台虚拟观景望远镜，它可以用来观测特定纬度地区的地形情况。该纬度的海拔数据记于数组 `heights`，其中 `heights[i]` 表示对应位置的海拔高度。请找出并返回望远镜视野范围 `limit` 内，可以观测到的最高海拔值。

示例 1:

输入: heights = [14,2,27,-5,28,13,39], limit = 3

输出: [27,27,28,28,39]

解释:

滑动窗口的位置	最大值
-----	-----
[14 2 27] -5 28 13 39	27
14 [2 27 -5] 28 13 39	27
14 2 [27 -5 28] 13 39	28
14 2 27 [-5 28 13] 39	28
14 2 27 -5 [28 13 39]	39

思路：没啥好说的遇到也写不出来队列问题

```

int* maxAltitude(int* heights, int numSize, int limit, int* returnSize) {
    if(numSize==0){
        *returnSize=0;
        return NULL;
    }
    int q[numSize];
    int left = 0, right = 0;
    for (int i = 0; i < limit; ++i) {
        while (left < right && heights[i] >= heights[q[right - 1]]) {
            right--;
        }
        q[right++] = i;
    }
    *returnSize = 0;
    int* ans = malloc(sizeof(int) * (numSize - limit + 1));
    ans[(*returnSize)++] = heights[q[left]];
    for (int i = limit; i < numSize; ++i) {

```



```

        while (left < right && heights[i] >= heights[q[right - 1]]) {
            right--;
        }
        q[right++] = i;
        while (q[left] <= i - limit) {
            left++;
        }
        ans[(*returnSize)++] = heights[q[left]];
    }
    return ans;
}

```

## 60.n个骰子的点数

你选择掷出 `num` 个色子，请返回所有点数总和的概率。

你需要用一个浮点数数组返回答案，其中第 `i` 个元素代表这 `num` 个骰子所能掷出的点数集合中第 `i` 小的那个的概率。

**示例 1:**

输入: `num = 3`

输出:

[0.00463,0.01389,0.02778,0.04630,0.06944,0.09722,0.11574,0.12500,0.12500,0.11574,0.09722,0.06944,0.04630,0.02778,0.01389,0.00463]

思路: 动态规划

```

double* statisticsProbability(int num, int* returnSize) {
    double* dp = (double*)malloc(sizeof(double) * 6);
    int dpsize=6;
    for(int i =0;i<6;i++)
    {
        dp[i]=1.0/6.0;
    }

    for(int i=2;i<=num;i++)
    {
        double*temp=(double*)calloc(5*i+1,sizeof(double));
        for(int j=0;j<dpsize;j++)
        {
            for(int k=0;k<6;k++)
            {
                temp[j+k]+=(double)dp[j]*(1.0/6.0);
            }
        }
        dp=(double*)realloc(dp,sizeof(double)*(5*i+1));
        for(int m=0;m<5*i+1;m++)
        {dp[m]=temp[m];}
        dpsize=5*i+1;
    }
}

```

```
        free(temp);
    }
    *returnSize=5*num+1;
    return dp;
}
```

## 61.扑克牌中的顺子

展览馆展出来自 13 个朝代的文物，每排展柜展出 5 个文物。某排文物的摆放情况记录于数组 `places`，其中 `places[i]` 表示处于第 `i` 位文物的所属朝代编号。其中，编号为 0 的朝代表示未知朝代。请判断并返回这排文物的所属朝代编号是否连续（如遇未知朝代可算作连续情况）

示例 1:

输入: `places = [0, 6, 9, 0, 7]`  
输出: `True`

思路：先排序，用一个数组记录这些文物，遍历数组如果当前值等于下一个的值返false，然后更新sum值

```
int cmp(int* a, int* b){
    return *a - *b;
}

bool isStraight(int* nums, int numsSize){
    qsort(nums, numsSize, sizeof(int), cmp);
    int res = 0;
    while(nums[res] == 0){
        res++;
    }
    int sum = 0;
    for(int i = res; i < numsSize - 1; i++){
        if(nums[i] == nums[i + 1]) return false;
        sum += nums[i + 1] - nums[i] - 1;
    }
    printf("%d", sum);
    return res >= sum ? true : false;
}
```

## 62.圆圈中最后剩下的数字

社团共有 `num` 位成员参与破冰游戏，编号为 `0 ~ num-1`。成员们按照编号顺序围绕圆桌而坐。社长抽取一个数字 `target`，从 0 号成员起开始计数，排在第 `target` 位的成员离开圆桌，且成员离开后从下一个成员开始计数。请返回游戏结束时最后一位成员的编号。

示例 1:

输入: num = 7, target = 4  
输出: 1

```
int iceBreakingGame(int num, int target) {  
    if (num == 1) {  
        return 0;  
    }  
    int prevRemaining = iceBreakingGame(num - 1, target);  
    return (prevRemaining + target) % num;  
}
```

## 63.股票的最大利润

数组 `prices` 记录了某芯片近期的交易价格, 其中 `prices[i]` 表示的 `i` 天该芯片的价格。你只能选择 **某一天** 买入芯片, 并选择在 **未来的某一个不同的日子** 卖出该芯片。请设计一个算法计算并返回你从这笔交易中能获取的最大利润。

如果你不能获取任何利润, 返回 0。

示例 1:

输入: prices = [3, 6, 2, 9, 8, 5]  
输出: 7  
解释: 在第 3 天 (芯片价格 = 2) 买入, 在第 4 天 (芯片价格 = 9) 卖出, 最大利润 = 9 - 2 = 7。

思路: 从最后一天开始向前遍历, 计算profit最高的时候

```
int maxProfit(int* prices, int pricesSize){  
    int ans = 0;  
  
    if(pricesSize>=2){  
        int max = prices[pricesSize-1],  
        int profit;  
        for(int i = pricesSize-2;i>=0;i--){  
            profit = max-prices[i];  
            if(profit<=0)  
                max = prices[i];  
            else if(profit>ans)  
                ans = profit;  
        }  
    }  
  
    return ans;  
}
```

## 64.求1+2+3+...+n

请设计一个机械累加器，计算从 1、2... 一直累加到目标数值 `target` 的总和。注意这是一个只能进行加法操作的程序，不具备乘除、if-else、switch-case、for 循环、while 循环，及条件判断语句等高级功能。

#### 示例 1:

输入: `target = 5`  
输出: 15

思路: 注意&和&&的区别

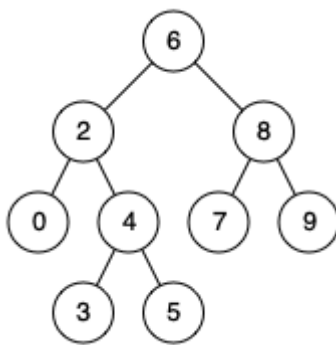
```
int sumNums(int n){  
    n && (n += sumNums(n - 1)); //可以查询 &和&&的区别//&&如果第一个就为假就不会在看之后的条件。同理||  
    中第一个为真也不会再执行后面的条件。  
    return n; //根据这个实现了判断的效果，这样，当n为0的时候就会停止调用函数。同理也可以借此过滤，自增自减运算符。  
    //上一个句中说的n是递归内层的形参的n。最外层的n是一个 n（初始状态）+（循环体）的形式。所以不会影响返回的值  
}
```

## 68. 二叉树的最近公共祖先

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。

[百度百科](#)中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树: `root = [6,2,8,0,4,7,9,null,null,3,5]`



#### 示例 1:

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。

思路: 从根节点开始遍历

如果当前节点的值大于 p 和 q 的值，说明 p 和 q 应该在当前节点的左子树，因此将当前节点移动到它的左子节点；

如果当前节点的值小于 p 和 q 的值，说明 p 和 q 应该在当前节点的右子树，因此将当前节点移动到它的右子节点；

如果当前节点的值不满足上述两条要求，那么说明当前节点就是「分岔点」。此时，p 和 q 要么在当前节点的不同的子树中，要么其中一个就是当前节点。

```
struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct
TreeNode* q) {
    struct TreeNode* ancestor = root;
    while (true) {
        if (p->val < ancestor->val && q->val < ancestor->val) {
            ancestor = ancestor->left;
        } else if (p->val > ancestor->val && q->val > ancestor->val) {
            ancestor = ancestor->right;
        } else {
            break;
        }
    }
    return ancestor;
}
```

---