要解决这个问题，实际中会怎么做，先不考虑代码，注意分模块，先把精力放在一个分模块上，一点点攻克，先易后难，最后思路出来了，在挑选数据结构使之能完成你的想法，最后在写代码，思路清晰在写代码，思考完了在写代码！！！真的很重要（这是自顶向下的设计模式）C++中面向对象的设计模式可以解决在这种自顶向下设计中出现的问题（如串联改变）其基本单元是类。通过对类的设计来完成对实体建模任务。所以无论是软件开发还是算法的设计最重要的都是对模型的理解，即从数据或者实际问题中理解并建立模型。

1.在二叉树中寻找值最大的节点并返回。

思路：1.查找结点：递归遍历二叉树，2.找到最大值并要保存该节点。

class Solution:

# @param {TreeNode} root the root of binary tree

# @return {TreeNode} the max node

def maxNode(self, root):

# Write your code here

a=findmax(root,maxval=-8888,a=[])

if len(a)==0:

return None

a=sorted(a,key=lambda x:x[0])

return a[-1][1]

def findmax(root,maxval,a):

if root!=None:

if maxval<root.val:

maxval=root.val

a.append([maxval,root])

findmax(root.left,maxval,a)

findmax(root.right,maxval,a)

return a

2. 查找斐波纳契数列中第 N 个数。

所谓的斐波纳契数列是指：

* 前2个数是 0 和 1 。
* 第 *i* 个数是第 *i*-1 个数和第*i*-2 个数的和。

斐波纳契数列的前10个数字是：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

思路：1.需要知道前面的节点，构造一维数组去做，算法复杂度为0(n)。

思路：2.我只需要前两相即可，没必要保存那摩多，因此考虑算法前后都是两相，所以考虑矩阵乘法，根据矩阵乘法性质，n^2\*n^2=n^4，这样分治算法就下来.

class Solution:

# @param n: an integer

# @return an integer f(n)

def fibonacci(self, n):

# write your code here

'''算法时间复杂度是O(n)

a=[0,1]

if n<2:

return a[n-1]

for i in range(2,n+1):

a.append(a[i-1]+a[i-2])

return a[n-1]

'''

#分治法算法复杂度时o(lg(n))大致如此，未检测

a=[[1,1],[1,0]]

real=n-1

if real%2==0:

temp=a\*\*(real//2)

return temp\*temp

else:

temp=a\*\*((real-1)//2)

return temp\*temp

3. 删除链表中等于给定值val的所有节点。

**样例**

给出链表 1->2->3->3->4->5->3, 和 val = 3, 你需要返回删除3之后的链表：1->2->4->5

思路：前后顺序不变，只要递归找然后相同删除即可，即相同就把pre的next指针指向cur的next，这样可以删除cur。不相同就往下走

Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.next = None

class Solution:

# @param head, a ListNode

# @param val, an integer

# @return a ListNode

def removeElements(self, head, val):

# Write your code here

if head == None:

return head

dummy = ListNode(0)

dummy.next = head

pre = dummy

while head:

if head.val == val:

pre.next = head.next

head = pre

pre = head

head = head.next

return dummy.next

4.给一组整数，按照升序排序，使用选择排序，冒泡排序，插入排序或者任何 O(n2) 的排序算法。

冒泡：

class Solution:

# @param {int[]} A an integer array

# @return nothing

def sortIntegers(self, A):

# Write your code here

for i in range(len(A)-1):

for j in range(len(A)-1):

if A[j]>A[j+1]:

A[j],A[j+1]=A[j+1],A[j]

return A

5. 在一个排序数组中找一个数，返回该数出现的任意位置，如果不存在，返回-1

思路：二分查找（非递归）也可以递归不过递归对内存要求高，有时候内存会对算法的时间复杂度产生影响。（具体原因未知）

class Solution:

# @param {int[]} A an integer array sorted in ascending order

# @param {int} target an integer

# @return {int} an integer

def findPosition(self, A, target):

# Write your code here

length=len(A)

pre=0

end=length-1

while pre<=end:

mid=(pre+end)//2

if A[mid]>target:

end=mid-1

elif A[mid]<target:

pre=mid+1

else:

return mid

return -1

6. 给出两个整数a和b, 求他们的和, 但不能使用 + 等数学运算符。

思路：不知道怎么想出来的，位运算永远是最快的，据说sqrt函数最牛的算法就是用位运算做出来的。

class Solution:

"""

@param a: The first integer

@param b: The second integer

@return: The sum of a and b

"""

def aplusb(self, a, b):

# write your code here, try to do it without arithmetic operators

if a==~b+1:

return 0

j=1

while(j):

t=a^b（异或）

j=(a&b)<<1（进位标志）

a=t

b=j

return t

7. 给定两个二进制字符串，返回他们的和（用二进制表示）a = 11 b = 1 a+b=100

思路：一种方法就是化成加法，而后算其二进制，显然这不是最优算法。二进制与十进制一样就是进位标志为2，按小学加法运算处理空位补零，每做一位确定进位。

class Solution:

# @param {string} a a number

# @param {string} b a number

# @return {string} the result

def addBinary(self, a, b):

# Write your code here

lena=len(a);lenb=len(b)

if lena>lenb:

b="0"\*(lena-lenb)+b

elif lenb>lena:

a="0"\*(lenb-lena)+a

i=len(a)-1;j=len(b)-1

result=[]

h=0

s=""

while i>=0 and j>=0:

t=int(a[i])+int(b[j])+h

h=0

if t>=2:

t=t%2

h=1

i-=1;j-=1

result.append(t)

#print(result)

if h !=0:

result.append(h)

for i in range(len(result)):

s=s+str(result[len(result)-i-1])

return s

8. 给出一个非负整数 num，反复的将所有位上的数字相加，直到得到一个一位的整数

这个需要判断是否是一位数，然后还要把整数化成字符串然后按序号取（以余数方式也可以）。

class Solution:

# @param {int} num a non-negative integer

# @return {int} one digit

def addDigits(self, num):

# Write your code here

if num<10:

return num

while num>=10:

num=jishuan(num)

return num

def jishuan(num):

h=0

while(num>=10):

h=h+num

num=num//10

sum=h+num

return sum

9. 你有两个用链表代表的整数，其中每个节点包含一个数字。数字存储按照在原来整数中相反的顺序，使得第一个数字位于链表的开头。写出一个函数将两个整数相加，用链表形式返回和。

Example 给出两个链表 3->1->5->null 和 5->9->2->null，返回 8->0->8->null

思路直接取然后逆过来放到字符串里然后变成整数，在想加变成链表即可（也可以按下面思路）

# Definition for singly-linked list.

# class ListNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.next = Non

class Solution:

# @param l1: the first list

# @param l2: the second list

# @return: the sum list of l1 and l2

def addLists(self, l1, l2):

# write your code here

h=0

a=[]

while l1!=None and l2!=None:

sum=l1.val+l2.val+h

h=0

if sum>=10:

h=sum//10

sum=sum%10

a.append(sum)

l1=l1.next

l2=l2.next

if l1==None and l2!=None:

a.append(l2.val+h)

if l1!=None and l2==None:

a.append(l1.val+h)

length=len(a)

#print(a)

l3=ListNode(a[0])

insertnode(l3,a[1:],length,i=1)

return l3

def insertnode(list1,node,length,i):

if i< length:

#if list1.next!=None:

list1.next=ListNode(node[0])

insertnode(list1.next,node[1:],length,i+1)

10.给定一个二叉树,确定它是高度平衡的。对于这个问题,一棵高度平衡的二叉树的定义是：一棵二叉树中每个节点的两个子树的深度相差不会超过1。判断是否平衡

思路：如果能有数据接口直接能保存每个节点树深就好了，但不能修改原本数据结构，因此要做出每个节点深度，但其实只需做出两个即左右节点，递归实现，每出一层深度加一，这里深度应该是该节点左右子节点的最大深度。

"""

Definition of TreeNode:

class TreeNode:

def \_\_init\_\_(self, val):

self.val = val

self.left, self.right = None, None

"""

class Solution:

"""

@param root: The root of binary tree.

@return: True if this Binary tree is Balanced, or false.

"""

'''

第一种方法时间花费506ms

def isBalanced(self, root):

# write your code here

if root == None:

return True

leftdepth=treedepth(root.left)

rightdepth=treedepth(root.right)

dif=leftdepth-rightdepth

if dif>1 or dif<-1:

return False

return Solution.isBalanced(self,root.left) and Solution.isBalanced(self,root.right)

def treedepth(root):

if root==None:

return 0

leftdepth=treedepth(root.left)

rightdepth=treedepth(root.right)

if leftdepth >rightdepth:

leftdepth+=1

else:

rightdepth+=1

return max(leftdepth,rightdepth)

'''

第二种

#花费时间为353ms

def isBalanced(self, root):

# write your code here

if root==None:

depth=0

return treedepth(root,depth=0)[0]

def treedepth(root,depth):

if root==None:

depth=0

return True,depth

#leftflag,leftitems=treedepth(root.left,0)

#print(leftitems)

#rightflag,rightitems=treedepth(root.right,0)

if treedepth(root.left,0)[0] and treedepth(root.right,0)[0]:

leftitems=treedepth(root.left,0)[1]

rightitems=treedepth(root.right,0)[1]

#print(leftitems,rightitems)

diff=leftitems-rightitems

if diff>=-1 and diff<=1:

depth=max(leftitems,rightitems)+1

return True,depth

return False,depth