You need to practice over and over again to achieve handling the questions below just like chewing gums.....

二叉树:

是否是平衡二叉树

```
def isBalanced(self, root):
    """
    :type root: TreeNode
    :rtype: bool
    """
    if not root:
        return True
    left = self.maxdepth(root.left)
    right = self.maxdepth(root.right)
    if abs(left-right) >1:
        return False
    else:
        return True and self.isBalanced(root.left) and self.isBalanced(root.right)
def maxdepth(self,root):
    if not root:
        return 0
    return max(self.maxdepth(root.left),self.maxdepth(root.right))+1
```

二叉树的前序、后序、中序遍历 递归&非递归

前序

```
def preorderTraversal(self, root):
    """
    :type root: TreeNode
    :rtype: List[int]
    """
    if not root:
        return []
    res=[]
    self.help(root,res)
    return res
def help(self,root,res):
    if not root:
        return []
    res.append(root.val)
    self.help(root.left,res)
    self.help(root.right,res)
```

非递归

```
def preorderTraversal(self, root):
"""
:type root: TreeNode
:rtype: List[int]
"""
if not root:
        return
res=[]
q = []
node = root
while len(q)>0 or node:
        while node:
                res.append(node.val)
                q.append(node)
                node = node.left
        node = q.pop()
        node = node.right
return res
```

中序

```
def inorderTraversal(self, root):
    """
    :type root: TreeNode
    :rtype: List[int]
    """
    if not root:
        return
    res=[]
    stack =[]

    node = root
    while len(stack)>0 or node:
        while node:
            stack.append(node)
            node = node.left

        node = stack.pop()
        res.append(node.val)
        node=node.right
    return res
```

后序

```
  def postorderTraversal(self, root):
    """
    :type root: TreeNode
    :rtype: List[int]
    """
    if not root:
        return
    res =[]
    stack=[]
    node = root
    while len(stack)>0 or node:
        while node:
            res.append(node.val)
            stack.append(node)
            node = node.right
        node = stack.pop()
        node= node.left
    return res[::-1]
```

由中序 + 另外一种遍历方式 重建二叉树

二叉树层序遍历 换行打印 & z字形打印

二叉树的镜像 递归&非递归

递归

```
def Mirror(self, root):
    # write code here
    if not root:
        return
    tmp = root.left
    root.left = root.right
    root.right = tmp
    self.Mirror(root.left)
    self.Mirror(root.right)
    return root
```

非递归

```
def Mirror(self, root):
        # write code here
        if not root:
            return
        stack =[root]
        while len(stack)>0:
            node = stack.pop()
            if node.left and node.right:
                tmp = node.left
                node.left = node.right
                node.right = tmp
                stack.append(node.left)
                stack.append(node.right)
            elif node.left:
                node.right = node.left
                node.left =None
                stack.append(node.right)
            elif node.right:
                node.left = node.right
                node.right =None
                stack.append(node.left)
        return root
```

判断二叉树是否为对称的 递归 & 非递归

递归

```
def isSymmetric(self, root):
    """
    :type root: TreeNode
    :rtype: bool
    """
    if not root:
        return True
    return self.ismirror(root,root)
def ismirror(self,root1,root2):
    if not root1 and not root2:
        return True
    if not root1 or not root2:
        return False

    if root1.val == root2.val:
        return self.ismirror(root1.left,root2.right) and self.ismirror(root1.right,root2.left)
    else:
        return False
```

递归

```
def isSymmetric(self, root):
    """
    :type root: TreeNode
    :rtype: bool
    """
    if not root:
        return True
    stack=[root,root]
    while len(stack)>0:
        node1 = stack.pop(0)
        node2=stack.pop(0)

        if not node1 and not node2:
            continue
        if not node1 or not node2:
            return False
        if node1.val != node2.val:
            return False
        stack.append(node1.left)
        stack.append(node2.right)
        stack.append(node1.right)
        stack.append(node2.left)
    return True
```

二叉树的最大深度 递归 & 非递归

递归

```
def maxDepth(root):
    if not root:
        retrun 0
    return max(maxDepth(root.left),maxDepth(root.right))+1
```

非递归

```
def maxdepth(root):
    if not root:
        return 0
    stack=[]
    stack.append([1,root])   ### 记录层数
    res =0
    while len(stack)>0:
        cur,node = stack.pop()
        if node.left:
            stack.append([cur+1,node.left])
        if node.right:
            stack.append([cur+1,node.right])
        res = max(res,cur)
    return res
```

二叉搜索树的最近公共祖先

递归

```
def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        if not root:
            return
        if root == p:
            return p
        if root == q:
            return q
        if p.val<root.val and q.val < root.val:
            return self.lowestCommonAncestor(root.left,p,q)
        if p.val > root.val and q.val >root.val:
            return self.lowestCommonAncestor(root.right,p,q)
        else:
            return root
```

非递归

```
def lowestCommonAncestor(self, root, p, q):
    if not root:
        return
    if root == p:
        return p
    if root == q:
        return q
    while root:
        if p.val<root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val >root.val:
            root = root.right
        else:
            return root
```

二叉树的最近公共祖先

递归

```
def lowestCommonAncestor(self, root, p, q):
    if not root or p== root or q==root:
        return root
    left = self.lowestCommonAncestor(root.left,p,q)
    right = self.lowestCommonAncestor(root.right,p,q)
    if left and right:
        return root
    elif left:
        return left
    elif right:
        return right
```

DP 动态规划

最小编辑距离

dp[i][j] = min(dp[i-1][j]+1,dp[i][j-1]+1,dp[i-1][j-1]+1) (word1[i] != word2[j])

背包问题 （0-1 背包、完全背包（物品个数不限）、多背包（物品个数有限个）

```
dp[i][j] = max(dp[i-1][j],dp[i-1][j-w[i]]+c[i])(j>=w[i])
dp[i][j] = max(dp[i-1][j],dp[i-1][j-k*w[i]]+k*c[i] ) for k in q[i]
dp[i][j] = max[dp[i-1][j],dp[i][j-w[i]]+c[i])
```

给定数组不能获取相邻的两个数，求和最大

dp[i] = max(dp[i-2]+nums[i],dp[i-1])

最大子序列和（子串和），输出序列?

子序列：dp[i] = max(dp[i-1]+arr[i],dp[i-1],arr[i])
子串：dp[i] = max(dp[i-1]+arr[i],arr[i])

乘积最大的子序列（子串）

mindp[i] = min(maxdp[i-1]*nums[i],mindp[i-1]*nums[i],nums[i])
maxdp[i] = max(mindp[i-1]*nums[i],maxdp[i-1]*nums[i],nums[i])
return max(maxdp)

最长上升子序列的长度（子串） 子序列：

for i in range(1,n):
for j in range(i,-1,-1):
if nums[j]<nums[i]:
dp[i] = max(dp[i],dp[j]+1)
子串：
for i in range(1,n):
if nums[i]>nums[i-1]:
dp[i] = (dp[i-1]+1,1)
class Solution(object):
"""
子序列

```
 """
    def f(self,arr):
        n = len(arr)
        dp=[1]*n

        for i in range(1,n):
            for j in range(i,-1,-1):
                if arr[i] >=arr[j]:
                    dp[i]= max(dp[j]+1,dp[i])
        return max(dp)
    """
    子串
    """
    def f2(self,arr):
        n = len(arr)
        dp=[1]*n
        for i in range(1,n):
            if arr[i]>arr[i-1]:
                dp[i] = max(dp[i-1]+1,1)
        return max(dp)
```

最长回文子序列（子串）

子序列：dp[i][j] = max(dp[i+1][j-1]+2,dp[i+1][j],dp[i][j-1])
子串：dp[i][j] 表示 s[i:j]是不是回文 是的话就标为1 不是的话就标为0
for j in range(n): ####这个遍历的方式很不一样！！
for i in range(j-1,-1,-1):
if j-i+1 == 2 and s[i] == s[j]:
dp[i][j]=1
if j-i+1>2:
if s[i]==s[j]:
dp[i][j] = dp[i+1][j-1]
else:
dp[i][j] = 0

最长公共子序列：

dp[i][j] = dp[i-1][j-1]+1 if word1[i]==word2[j]
dp[i][j] = max(dp[i-1][j],dp[i][j-1])

股票每天的价格，只进行一次买入卖出，求最大收益

dp[i] = max(dp[i-1],prices[i]-minp) minp 表示 前i-1天中的最小的价格

0-1 矩阵 求只包含1的最大正方形

dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1

0-1矩阵 找出每个元素到最近的0的距离

数塔问题（三角形 从上往下的最小路径和）

dp[i][j] = min(dp[i-1][j-1],dp[i-1][j])+nums[i][j] 注意边界 也可以用自底向上法 就是遍历要反过来

```
def f(arr):
    dp=arr[:][:]
    n = len(arr)
    for i in range(1,n):
        for j in range(len(arr[i])):
            if j == 0:
                dp[i][j] = dp[i-1][j]+arr[i][j]
            elif j == len(arr)-1:
                dp[i][j] = dp[i-1][j-1]+arr[i][j]
            else:
                dp[i][j] = min(dp[i-1][j-1]+arr[i][j],dp[i-1][j]+arr[i][j])
    return min(dp[-1])
```

零钱兑换 （有一个目标值，一个arr ， arr里的coin可以使用任意多次，求凑成目标值的方法数）

设零钱有n种，目标值为amount dp：（n）x amount+1）

```
def f(amount,arr):
    n = len(arr)
    dp = [[0]*(amount+1) for _ in range(n)]
    for i in range(n):
        dp[i][0] =1
    for j in range(amount+1):
        if amount % arr[0] ==0:
            dp[0][j] =1
    for i in range(1,n):
        for j in range(1,amount+1):
            if j>=arr[i]:
                # 不用上这个coin ， 和用这个coin
                dp[i][j] = dp[i-1][j] + dp[i][j-arr[i]]
            else:
                dp[i][j] = dp[i-1][j]
    return dp[-1][-1]
```

链表：

两两交换链表中结点

递归

```
def swapPairs(self, head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if not head:
        return


    if  head.next:
        tmp = head.next
        head.next = self.swapPairs(tmp.next)
        tmp.next = head
        return tmp
    else:
        return head
```

找到链表中环的入口 （hash表 / O(1)）

```
def EntryNodeOfLoop(self, pHead):
    # write code here
    if pHead==None or pHead.next==None or pHead.next.next==None:
        return
    pre = pHead.next.next
    cur = pHead.next
    while pre != cur:
        if pre.next==None or pre.next.next==None:
            return None
        pre = pre.next.next
        cur = cur.next
    pre = pHead
    while pre!=cur:
        pre = pre.next
        cur = cur.next

    return cur
```

反转链表

```
def reverse(head):
    if not head:
        return
    p = None
    while head:
        q = head.next
        head.next = p
        p = head
        head = q
    return p
```

两个链表的第一个公共结点

双指针

```
def getIntersectionNode(self, headA, headB):
    """
    :type head1, head1: ListNode
    :rtype: ListNode
    """
    if not headA or not headB:
        return
    pa = headA
    pb = headB

    while pa!=pb:
        if pa:
            pa = pa.next
        else:
            pa = headB
        if pb:
            pb = pb.next
        else:
            pb = headA
    return pa
```

计算长度法

```
def getIntersectionNode(self, headA, headB):
  """
  :type head1, head1: ListNode
  :rtype: ListNode

  """
  if not headA or not headB:
      return
  a = headA
  b = headB
  n = 0
  m = 0
  while a:
      a = a.next
      n = n+1
  while b:
      b = b.next
      m = m+1
  a = headA
  b = headB
  while a or b:
      if a == b:
          return a
      if n ==m:
          a=a.next
          b = b.next
      elif n>m:
          a=a.next
          n -=1
      else:
          b = b.next
          m -=1
  return
```

删除链表中的重复元素，使得每个元素只出现一次

```
def deleteDuplicates(head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if not head:
        return
    p = head
    while p:
        if p.next:
            if p.val == p.next.val:
                if p.next.next:
                    p.next = p.next.next
                else:
                    p.next = None
            else:
                p =p.next
        else:
            p = p.next
    return head
```

删除重复元素，使其不出现

```
class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head:
            return
        p = ListNode(-99999)
        p.next = head
        pre = p
        cur = head

        while cur:
            flag=False
            while cur.next and cur.val == cur.next.val:
                flag =True
                cur = cur.next
            if flag == True:
                pre.next = cur.next

            else:
                pre = cur
            cur = cur.next
        return p.next
```

合并两个有序链表

递归 非递归的话类似于 归并排序的那部分

```
def Merge(self, pHead1, pHead2):
    # write code here
    if not pHead1:
        return pHead2
    if not pHead2:
        return pHead1
    head = ListNode(0)
    if pHead1.val<pHead2.val:
        head = pHead1
        head.next = self.Merge(pHead1.next,pHead2)
    else:
        head = pHead2
        head.next = self.Merge(pHead1,pHead2.next)
    return head
```

链表的倒数第k个结点

```
def FindKthToTail(self, head, k):
    # write code here
    if not head:
        return
    if k==0:
        return
    low  =head
    fast = head
    i = 0
    while i<k-1:
        if fast.next:
            fast=fast.next
            i = i+1
        else:
            return
    while fast.next:
        low = low.next
        fast = fast.next
    return low
```

删除倒数第N个结点

```
def removeNthFromEnd(self, head, n):
    """
    :type head: ListNode
    :type n: int
    :rtype: ListNode
    """

    if not head:
        return

    low = head
    fast = head
    i = 0
    while i<n-1:
        if fast.next:
            fast = fast.next
            i +=1
    if fast.next:
        while fast.next.next:
            fast = fast.next
            low = low.next
        low.next = low.next.next
        return head
    else: ## 说明 fast已经走到最后一个结点了 那就是要删去第一个结点
        return head.next
```

双/三指针：

三数之和

荷兰国旗问题

def f(arr):

```
cur = 0
low = 0
hi = len(arr)-1
while cur<= hi:
    if  arr[cur] == 0 and cur == low:
        cur += 1
        low +=1
    elif arr[cur] == 0 and low<cur:
        tmp = arr[cur]
        arr[cur] =arr[low]
        arr[low] = tmp
        low += 1
        cur +=1
    elif arr[cur] ==1 :
        cur +=1
    elif arr[cur] == 2:
        tmp = arr[cur]
        arr[cur] = arr[hi]
        arr[hi] = tmp
        hi -=1
print(arr)
```

有效三角形的个数：

```
def triangleNumber(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    if not nums:
        return
    n = len(nums)
    nums.sort()
    res=0
    for i in range(n-1,1,-1):
        k = 0
        j = i-1
        while k<j:
            if nums[k] + nums[j] > nums[i]:
                res += j-k
                j = j-1
            else:
                k +=1
    return res
```

排序：

每种的最好最坏情况的复杂度（quick sort and merge sort and heap sort

##'''直接插入排序'''''

将数组中的所有元素依次和前面已经排好的元素进行比较，如果选择的元素比已经排序的元素小，则交换，直到全部元素都比较过

arr = [5, 6, 3, 1, 8, 7, 2, 4]

```
def insertsort(arr):
    for i in range(1,len(arr)):  # 遍历数组中的所有元素
        for j in range(i,0,-1):  # range(start,stop,step)  将该元素依次和前面的元素比较
            if arr[j] < arr[j-1] :
                temp = arr[j]
                arr[j] = arr[j-1]
                arr[j-1] = temp
            else:
                break
    return arr
```

print(insertsort(arr))

'''希尔排序'''

将待排序数组按照步长gap进行分组，然后将每组的元素利用直接插入排序的方法进行排序；

每次将gap折半减小，循环上述操作；当gap=1时，利用直接插入，完成排序。

arr = [5, 6, 3, 1, 8, 7, 2, 4]

```
def shell_sort(arr):
    gap = int(len(arr)/2)

    while gap >=1:
        # 在根据gap分组后的组内进行插入排序，一组的元素不一定是只有两个的 所以要遍历
        for i in range(gap,len(arr)): # 遍历组
            for j in range(i-gap,-1,-gap):
                if arr[j] > arr[j+gap]:
                    temp =arr [j]
                    arr[j] = arr[j+gap]
                    arr[j+gap] = temp
                    print(arr)

        gap =int(gap/2)

    return arr
```

print(shell_sort(arr))

或者

```
def shell(arr):
    gap = int(len(arr)/2)
    while gap >=1:
        for i in range(1,len(arr)):
            for j in range(i,0,-gap):
                if arr[j] < arr[j-gap]:
                    temp = arr[j]
                    arr[j] =arr[j-gap]
                    arr[j-gap] = temp
        gap = int(gap/2)
    return arr
print(shell(arr))
```

'''简单选择排序'''

从待排序序列中，找到关键字最小的元素；

如果最小元素不是待排序序列的第一个元素，将其和第一个元素互换；

从余下的 N - 1 个元素中，找出关键字最小的元素，重复(1)、(2)步，直到排序结束。

arr = [5, 6, 3, 1, 8, 7, 2, 4]

```
def select_sort(arr):

    for i in range(len(arr)):
        min=i
        for j in range(i,len(arr)):

            if arr[min] > arr[j]:
                min = j  # 查找从 i 到 n 之间的 最小数的
        if min != i:  # 如果最小数不是i（最前面的那个）  那么将最小的这个数 和 i 位置的数交换
            temp = arr[i]
            arr[i] = arr[min]
            arr[min] = temp
    return arr
```

'''堆排序'''

'''冒泡排序''' ## easy

将序列当中的左右元素，依次比较，保证右边的元素始终大于左边的元素；

（ 第一轮结束后，序列最后一个元素一定是当前序列的最大值；）

对序列当中剩下的n-1个元素再次执行步骤1。

对于长度为n的序列，一共需要执行n-1轮比较

（利用while循环可以减少执行次数）

```
def bubble_sort(arr):
    i = len(arr)
    while i > 0:
        for j in range(1,i):
            if arr[j] < arr[j -1]:
                temp = arr[j]
                arr[j] = arr[j - 1]
                arr[j - 1] = temp
        i = i - 1

    return arr
```

'''快速排序'''

快速排序的基本思想：挖坑填数+分治法

从序列当中选择一个基准数(pivot)

在这里我们选择序列当中第一个数最为基准数

将序列当中的所有数依次遍历，比基准数大的位于其右侧，比基准数小的位于其左侧

重复步骤1.2，直到所有子集当中只有一个元素为止。

```
def quick_sort(array, left, right):
    if left >= right:
        return
    low = left
    high = right
    key = array[low]
    while left < right:
        while left < right and array[right] > key: #当key右边的数大于key时 从最右边往里走，形成遍历
            right=right -1
        array[left] = array[right]     #如果不满足条件大于key的条件，则将该数字赋给左边

        while left < right and array[left] <= key: #当key左边的数小于等于key时，从最左边往里走，遍历
            left =left +1
        array[right] = array[left]  # 如果不满足小于等于key的条件，则将该数字赋值给右边
    array[right] = key
    quick_sort(array, low, left - 1)
    quick_sort(array, left + 1, high)

def quick(arr,start,end):
    if start < end :
        i,j,key = start,end,arr[start]
        while i<j:
            while i<j and arr[j] >= key:
                j = j-1
            if i<j:
                arr[i] = arr[j]
                i = i + 1
            while i<j and arr[i] < key:
                i = i+1
            if i<j:
                arr[j] = arr[i]
                j = j-1
        arr[i] = key
        quick(arr,start,i-1)
        quick(arr,i+1,end)
    return arr
print(quick(arr,0,len(arr)-1))
```

'''归并排'''

采用分治法，将已有序的子序列合并得到一个完全有序的序列，即先使每个子序列有序，再使子序列段有序。

现将数列分成两个有序的子序列

再将两个子序列合并成一个序列

```
arr=[2,4,5,7,1,3]
def wayMerge(a,b):
    merseries =[]
    i =0
    j =0
    while i < len(a) and j < len(b):
        if a[i] >= b[j]:
            merseries.append(b[j])
            j = j+1
        else:
            merseries.append(a[i])
            i  = i+1
    merseries.extend(a[i:])
    merseries.extend(b[j:])
    return merseries
def mergeSort(series):
    if len(series) <=1:
        return series
    i = len(series) //2
    l = mergeSort(series[:i])
    r = mergeSort(series[i:])
    return wayMerge(l,r)

print(mergeSort(arr))
```

"""

堆排

"""

大顶堆 ## 顺序排列

```
def adjust(low,high):
    i = low ## i 为想要调整的结点
    j = i*2+1  ### j 是它的左孩子
    while j<high:  ### 存在孩子
        if j+1<high and arr[j+1]>arr[j]:  ## 如果右孩子存在，且右孩子比左孩子要大
            j = j+1       ### 让j 存的是右孩子的下标
        if arr[j] > arr[i]:    ## 如果孩子中最大的那个 比 欲调整的结点i大
            tmp=arr[j]        ##  将 最大权值的那个孩子j 和欲调整结点i 进行交换
            arr[j] = arr[i]
            arr[i] = tmp
            i= j       ### 保持i为欲调整结点
            j = i*2 +1  ### j为其左孩子
        else:
            break    ### 孩子的值都比欲调整结点i的值小，调整结束

def heapsort_(arr):
    n = len(arr)
    ## 建堆 从第一个有孩子的结点开始调整
    for i in range(n//2,-1,-1):
        adjust(i,n)  ## 和下面的对比 做调整
    ## 每建完一次堆 根结点就是最大的那个数字 把它和最后的那个数字进行交换
    ## 调整堆
    for i in range(n-1,0,-1):
        tmp = arr[0]   ## 根结点的数字
        arr[0] = arr[i]  ## 将其和最后的数字进行交换
        arr[i] = tmp
        adjust(0,i) ## 对arr[0:i] 的进行排序
    return arr

print(heapsort_(arr))
```

栈：

栈的压入、弹出序列是否合法

```
def IsPopOrder(self, pushV, popV):
    # write code here
    tmp=[]
    pop = popV[:]
    for i in pushV:
        tmp.append(i)

        if i == pop[0]:

            while tmp[-1] == pop[0]:
                tmp.pop()
                pop.pop(0)
                if not tmp and not pop:
                    return True
    if not tmp and not pop:
        return True
    else:
        return False
```

二分法：（似乎蛮容易错的）

递归

```
def bs(arr,key,left,right):
    if left>right:
        return -1
    mid = (left+right)//2
    if arr[mid] == key:
        return mid
    if arr[mid] > key:
        return bs(arr,key,left,mid-1)
    else:
        return bs(arr,key,mid+1,right)

print(bs([1,2,3,4,5,6],7,0,5))
```

非递归

```
def bs(arr,key):
    n = len(arr)
    i = 0
    j = n-1
    while i<=j:
        mid = (i+j)//2
        if arr[mid] == key:
            return mid
        if arr[mid] > key:
            j = mid-1
        if arr[mid] < key:
            i = mid+1
    return -1
```