**旋转数组找target**

class Solution(object):

def search(self, nums, target):

if not nums:

return -1

ll = len(nums)

l,r = 0, ll-1

while l<=r:

while l<r and nums[l]==nums[l+1]: # 因为数组可以有重复的数

l += 1

while l<r and nums[r]==nums[r-1]:

r -= 1

mid = (l+r)/2

if nums[mid] == target:

return mid # return mid就是返回target的下标

# mid 和 l 比较 得到两边哪一边有序

if nums[mid]>=nums[l]: # 左有序

# target在有序的段内

if target < nums[mid] and target >= nums[l]:

r = mid - 1

else: # 不在有序的段内，也就是在右边咯，那把l加大

l = mid + 1

else: # 右有序

# 在有序的段内

if target > nums[mid] and target <= nums[r]: # target在有序的这段

l = mid + 1

else: # 不在，那就是在左段咯，把r减少

r = mid - 1

return -1

**旋转数组找最小**

class Solution(object):

def findMin(self, nums):

l,r = 0, len(nums)-1

while l<r:

mid = (l+r)/2

if nums[mid] < nums[r]:

r = mid

else: # nums[mid]>=nums[r]证明最小值在后面那段

l = mid+1 # 所以要把l变大

return nums[l]

**非连续递增子序列**  O(n^2) dp

class Solution(object):

def lengthOfLIS(self, nums):

if not nums:

return 0

l = len(nums)

dp = [1]\*l

res = 1

for i in range(l):

**for j in range(i): # 注意是i**

if **dp[j]+1>dp[i]** and nums[i]>nums[j]: # 状态转移方程

dp[i] = dp[j]+1

if res < dp[i]:

res = dp[i]

return res

**dp+二分 时复nlogn**

def fun(nums):

l = len(nums)

dp = [0]\*l # dp[i]:递增子串长度为i中，最尾元素最小的那个，的尾元素值

maxlen = 0

for i in range(l): # 最外层时复是O(n) 内部的二分时复是O(logn)

lo, hi = 0, **maxlen** # 所以总的时复是O(nlogn)

while lo < hi:

mid = (lo+hi)/2

if dp[mid] < nums[i]:

lo = mid + 1

else:

hi = mid

dp[lo] = nums[i] # 以上代码是二分查找的把nums填入dp

if lo == maxlen: # 如果lo!=maxlen的话 证明插入的值在0～maxlen的话之间

# 新来的值不比dp[-1]大，最大递增串也没能+1

maxlen += 1

return maxlen

**统计二进制表示中1的个数**

def fun(n):

count = 0

if n < 0: n = n & 0xffffffff # 考虑负数的情况

while n:

n = n&(n-1) # n和n-1与运算，消除n中最后边的1

count += 1

return count

**滑动窗口最大值**

class Solution(object):

def maxSlidingWindow(self, nums, k):

win, res = [], [] # win存放没过期的，最大值的，index

for i, v in enumerate(nums):

if i >= k and win[0] <= i - k: # win[0] <= i - k 证明窗已经划滑过了

win.pop(0) # pop(0)把最早的index剔除

while win and nums[win[-1]] <= v:

win.pop()

win.append(i) # 把大的值的index加进来

if i >= k - 1:

res.append(nums[win[0]]) # 注意append的是win[0]

return res

**快速判断是不是2的幂**

def fun(a):

**res = a&(a-1)**  # 消除a中最右边的1

if not res:

return True

return False

**行列有序找第k小/大 左到右、上到下、递增**

class Solution(object):

def kthSmallest(self, matrix, k):

m, n = len(matrix), len(matrix[0])

lo,hi = matrix[0][0], matrix[-1][-1]

while lo<=hi:

mid = (lo+hi)/2

i,j=m-1, 0 **# 左下角开始找 注意！！**

count = 0

while i>=0 and j<n:

if matrix[i][j]<=mid:

**count += (i+1)** # 这一列上面所有的数都是比mid小

**j += 1** # 列右移

else:

**i -= 1** # 行上移

if count < k: # mid选小了

lo = mid+1

else:

hi = mid-1

return lo

**行列递增数组，寻找是否存在某值target 逐行+行内二分**

class Solution(object):

def searchMatrix(self, matrix, target):

if not matrix or not matrix[0]:

return False

m = len(matrix)

n = len(matrix[0])

for i in range(m):

l,r = 0,n-1

while l<=r:

if matrix[i][r] < target: # 这一行的最大值都比target小? 那肯定不在这行

break # break出来，往下一行看

else: # 这部分用二分，可以使得当矩阵维度很大的时候，时复更友好

mid = (l+r)/2

if matrix[i][mid] == target:

return True

elif matrix[i][mid] < target:

l = mid + 1

else:

r = mid - 1

return False

**0-1矩阵内全1 最大矩形面积**

def maximalRectangle(matrix):

if not matrix or not matrix[0]:

return 0

n = len(matrix[0])

height = [0] \* (n + 1)

ans = 0

for row in matrix:

for j in range(n): # [i][j]==1则height+1,否则height清0

height[j] = height[j] + 1 if row[j] == 1 else 0

stack = [-1] # stack存放的是width信息

# stack会依次压入列的index，当height[stack[-1]]<height[r]

# 证明后一列下方的高度减小了，这个时候把之前那个较大的height作为h

# 宽度就是当前列index r-1 - stack[-1] 很好理解啊这个算法

for r in range(n + 1):

while height[r] < height[stack[-1]]:

h = height[stack.pop()]

w = r - 1 - stack[-1]

ans = max(ans, h \* w)

stack.append(r) # 不管上面的while循环,每次反正会把r压进来

# 缺失的r就是上面被pop掉了

return ans

**和为target的连续子数组**  的个数 **时复O(n) 用dict**

class Solution(object):

def subarraySum(self, nums, k):

count = 0

if not nums:

return count

l = len(nums)

Sum= 0

dic = {}

**dic[0]=1**  # 这句也不能漏且不能随便写成别的 会bug...

for i in range(l):

Sum += nums[i]

if (Sum-k) in dic:

count += dic[Sum-k] # 这里注意是直接dic[Sum-k]

# 重复出现了几次就加几

**if Sum not in dic:**

**dic[Sum] = 0**

**dic[Sum] += 1** # 这里容易出错

# 不管s-target值如何，sum都要在dic中+=1的，所以要先判断是不是存在，然后再自加1

return count

**时复O(n^2)：**

count = 0

for i in range(l):

ssum = 0

**for j in range(i, l): # 注意这里是i~l**

ssum += nums[j]

if ssum == k:

count += 1

print count

**扩展：连续子数组和为k的倍数**，则**dict存sum%k** 即可。时复也是O(n)

**Top k 最小堆做 还可以二分**

**用最小堆是因为：堆顶最小，数组中其他的数每次和堆顶比较就ok，使得这个堆一直是保持最大状态。最后返回这个堆，就是topk。**

def heap\_adjust(data, root):

if 2\*root+1 < len(data):

# 下面四行 找到更小的子

if 2\*root+2 < len(data) and data[2\*root+2] < data[2\*root+1]:

k = 2\*root+2

else:

k = 2\*root+1

# 把堆顶换成最小

if data[k] < data[root]:

data[k],data[root] = data[root],data[k]

heap\_adjust(data,k) # 递归维护下一个层子

# 创建堆

def min\_heap(data):

ind = len(data)/2 - 1

for i in range(ind, -1, -1):

heap\_adjust(data, i)

return data

def main\_topk(nums, k):

nums\_k = min\_heap(nums[:k]) # 先用前k个数组成一个堆

for i in range(k, len(nums)):

if nums[i] > nums\_k[0]:

nums\_k[0] = nums[i]

nums\_k = min\_heap(nums\_k) # 跟新维护这个堆

return nums\_k

res = main\_topk(s, 5)

**N个数组中的topk N个数组均是降序的了**

**维护一个大顶堆，别和上面混淆了…：因为大顶堆使得堆顶最大，每次弹出堆顶，然后后续补充新数进来，再维护堆使得堆顶依旧最大。返回的是每次弹出的堆顶值，弹出k次。**

**数组除自身乘积 左右乘子**

class Solution(object):

def productExceptSelf(self, nums):

left,right = 1,1

l = len(nums)

res = [0]\*l

for i in range(l):

res[i] = left # 现在的res存放的是：每个位置上元素，其左边需要乘的值

left \*= nums[i] # left init=1 第一个值的左边乘1啊 没问题的

for j in range(l-1,-1,-1):

res[j] \*= right # 这里res开始依次把每个位置上元素，其右边需要乘的值乘上

# 所以是\*=right right init=1，最后一个值右边可不就是乘1嘛

right \*= nums[j]

return res

**连续子数组乘积最大**

class Solution(object):

def maxProduct(self, nums):

l = len(nums)

mmax = nums[0]

a = 1

for i in range(l):

a \*= nums[i]

mmax = a if a > mmax else mmax

if nums[i] == 0:

a = 1 # 遇到0 重新乘

# 负数为偶数个，则整个数组的各个值相乘为最大值

# 负数为奇数个，则从左边开始，乘到最后一个负数停止有一个“最大值”，(中间的负负可得正)

# 同理从右边开始乘到最后一个负数，也有一个“最大值”，比较这俩最大值即可

# 所以只要左右分别遍历一回就ok

a = 1

for j in range(l-1,-1,-1):

a \*= nums[j]

mmax = mmax if mmax > a else a

if nums[j] == 0:

a = 1

return mmax

**01矩阵最接近的1的距离**

class Solution(object):

def updateMatrix(self, matrix):

m = len(matrix)

n = len(matrix[0])

ners = [[-1,0],[1,0],[0,1],[0,-1]]

queue = []

for i in range(m):

for j in range(n):

if matrix[i][j] == 0:

queue.append([i,j])

else:

matrix[i][j] = m+n # 如果矩阵值是1，给成最大值tag一下

while queue:

cur = queue.pop(0)

for ner in ners:

x,y=cur[0]+ner[0],cur[1]+ner[1]

# 因为xy 和 cur 是相邻的，所以value的理论距离是1

# 那 matrix[x][y] > matrix[cur[0]][cur[1]]+1 则说明xy位置上值为1 所以xy位置上距离值在cur上+1

if x>=0 and y>=0 and x<m and y<n and matrix[x][y] > matrix[cur[0]][cur[1]]+1:

matrix[x][y] = matrix[cur[0]][cur[1]] + 1

queue.append([x,y])

return matrix

**最长不重复子串长度**

**连续的数组题，多用dict，不连续的多用dp**

# O(n)的方法

class Solution(object):

def lengthOfLongestSubstring(self, s):

res = 0

if not s:

return res

d = {} # key 是s中的字符 value是这个字符出现的index index会后序被刷新

start = 0

for i in range(len(s)):

if s[i] in d and d[s[i]] >= start:

start = d[s[i]] + 1 # 新的不重复子串的头要在d[s[i]]的基础上后移一位

tmp = i + 1 - start

d[s[i]] = i

res = tmp if tmp > res else res

return res

**上dp 一个数，被最少的平方数之和组成**

class Solution(object):

def numSquares(self, n):

f = [i for i in range(n+1)]

for i in range(n+1): # i为需要被组合的值

j = 1

while j\*j <= i:

f[i] = min(f[i], f[i-j\*j]+1)

j+=1

return f[-1]

**几乎一样的题 最少的零钱兑换张数**

class Solution(object):

def coinChange(self, coins, amount):

dp = [amount+100] \* (amount+1) # 这里先把最坏的兑换可能预设好

dp[0]=0

for i in range(1, amount+1): # i为需要兑换的金额值

for coin in coins:

if i >= coin:

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

return dp[-1] if dp[-1] != amount+100 else -1

# **上面两题，第一个循环是要拼凑的值，第二个循环是组合序列**

# **下面的题，第一个循环是组合序列，第二个循环是需要组合的值**

**继续dp 拼凑金额的最多可能性**

# a = [1, 5, 10, 20, 50, 100]

def dp(num):

dp = [1 for i in range(0, num + 1)]

for i in range(1, 6): # 这个6是共有6种面额 a数组的长度

for j in range(1, num + 1): # j为需要拼凑的金额值

if j >= a[i]:

dp[j] = dp[j] + dp[j - a[i]]

return dp[-1]

**数组中的 最长斐波拉契序列**

a[i][j]用于记录A[?]+A[i]是否能得到A[j]

def lenLongestFibSubseq(self, A): # A是递增序列

n, res = len(A), 0

a = [[0] \* n for i in range(n)]

for i, v in enumerate(A):

**lo, hi = 0, i - 1**

while lo < hi:

if A[lo] + A[hi] < v:

lo += 1

elif A[lo] + A[hi] > v:

hi -= 1

else: # A[lo] + A[hi] == v

if a[lo][hi]:

**a[hi][i]** = a[lo][hi] + 1 # v可以添加在a[lo][hi]构成的序列后

else:

**a[hi][i]** = 3 # lo+hi=i

res = max(a[hi][i], res)

lo += 1

hi -= 1

return res

**数据流的中位数 左右分别维护最大/小堆**

def max\_min\_dui(a, left, right):

# left right 是左右最大/小堆 a是新进来的数

if (len(left) + len(right)) & 1 == 0: # &1==0则是偶数

if left and a < max(left):

left.append(a)

a = max(left)

left.remove(a)

right.append(a) # 左边加一个，则把最大的弹出给右边 左右个数相等

else: # 当前个数为奇数，默认左边多一个 则新的数考虑放右边去

if right and a > min(right):

right.append(a)

a = min(right)

right.remove(a) # 右边把最小值弹出，给到左边去

left.append(a)

return left, right # 返回一个数进来后，调整好的左右大小堆

def main(arrs):

left, right = [],[]

for i in range(len(arrs)):

left, right = max\_min\_dui(int(arrs[i]), left, right)

if len(arrs) & 1 == 0:

res = (max(left) + min(right))/2

else:

**res = min(right)** # 注意是右边的最小堆会多一个数

# 应该是因为最先是right开始add数进来吧...

return res

**两有序链表合并**

class Solution(object):

def mergeTwoLists(self, pHead1, pHead2):

**tmp = ListNode(0)**

**phead = tmp**

while pHead1 and pHead2:

if pHead1.val < pHead2.val:

tmp.next = pHead1

pHead1 = pHead1.next

else:

tmp.next = pHead2

pHead2 = pHead2.next

tmp = tmp.next

if pHead1:

tmp.next = pHead1

if pHead2:

tmp.next = pHead2

**return phead.next**

**合并k个有序链表 用堆 时复nlogk n是k个链表所有节点的个数**

**import heapq**

**# 堆，找到k个节点中的最小值**

class Solution(object):

def mergeKLists(self, lists):

head = ListNode(0) # init

current = head

heap = []

for node in lists:

if node:

heap.append((node.val, node))

heapq.**heapify**(heap) # 堆维护，时复klogk

while heap:

\_, nod = heapq.**heappop**(heap) # value and node

current.next = nod

current = current.next

**if nod.next:**  # 入堆的那个链表的下一个节点, 压入heap

heapq.**heappush**(heap, (nod.next.val, nod.next))

return head.next

**两堆实现队列 一个入数据 一个出数据**

class Solution:

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

self.stack1 = [] # 入数据

self.stack2 = [] # 出数据，靠stack1弹出的数据弹入stack2

def push(self, node):

self.stack1.append(node)

def pop(self):

if self.stack2:

return self.stack2.pop()

elif not self.stack1: # stack1空，那就直接none了，没数据可删

return None

else: # stack1有数据，stack2为空

while self.stack1:

self.stack2.append(self.stack1.pop())

return self.stack2.pop()

**a + b + c closest to target 三指针法**

class Solution(object):

def threeSumClosest(self, nums, target):

nums.sort() # 数组先排序

res = sum(nums[0: 3])

diff = abs(res - target)

for i in range(len(nums) - 2):

j, k = i + 1, len(nums) - 1 # ijk三指针

if nums[i] \* 3 - target > diff:

break

while j < k:

r = nums[i] + nums[j] + nums[k]

if abs(r - target) < diff:

res = r

diff = abs(r - target)

if r > target: # 和大了，把k往前移

k = k - 1

elif r < target: # j往后移

j = j + 1

else: # r==target

return r

return res

**数组中，重复/消失 的数 置负处理 时复O(n)**

class Solution(object):

def findDuplicates(self, numbers):

res = []

if numbers == None:

return res

l = len(numbers)

out = [0]\*l # 没访问的到，标记为0

for i in range(l):

if out[numbers[i]-1] == 0:

**out[numbers[i]-1] = -numbers[i]**  # 这里给别的值其实也可以

# 为了延续数组消失数的思想，还是用-nums[i]吧～ 表示index numbers[i]-1处已经被访问好了

**else:**

**res.append(numbers[i])**

return res #返回的就是重复的值 因为不重复的数，以他的值作为index，只可能出现一次是0，不然就是重复出现的值

**或者可以用二分法，但是需要一个count函数统计，counts = count(arr, start, mid) arr数组中，值在start~mid之间，元素的个数**

**数组中消失的数 出现的数，以他为index，把对应位置上的元素置负，O(n)遍历完后，值>0的index值就是消失的数值 因为这几个位置上没有被置负**

class Solution(object):

def findDisappearedNumbers(self, nums):

res = []

l = len(nums)

for i in range(l):

nums[abs(nums[i])-1] = -(abs(nums[abs(nums[i])-1])) # 注意abs的使用

for j in range(l):

if **nums[j] > 0: # >0处就是消失的值 index和value之间是差1的 so:j+1**

**res.append(j+1)**

return res

**nms**

def non\_max\_suppress(predicts\_dict, thr):

for object\_name, bbox in predicts\_dict.items(): # class and it's bbox: key, value

bbox\_array = np.array(bbox, dtype=np.float)

x1 = bbox\_array[:, 0]

y1 = bbox\_array[:, 1]

x2 = bbox\_array[:, 2]

y2 = bbox\_array[:, 3]

scores = bbox\_array[:, 4]

order = scores.argsort()[::-1] # 逆序排序score

areas = (x2 - x1 + 1) \* (y2 - y1 + 1)

keep = []

while order.size > 0:

i = order[0] # i 是box的编号

keep.append(i) # cur max's index

# calue the iou:

xx1 = np.maximum(x1[i], x1[order[1:]]) # x1[i] is the cur max, x1[order[1:]] is the other bbox

yy1 = np.maximum(y1[i], y1[order[1:]])

xx2 = np.minimum(x2[i], x2[order[1:]])

yy2 = np.minimum(y2[i], y2[order[1:]])

inter = np.maximum(0.0, xx2 - xx1 + 1) \* np.maximum(0.0, yy2 - yy1 + 1)

iou = inter / (areas[i] + areas[order[1:]] - inter) # 广播机制

inds = np.where(iou <= thr)[0] # save the bbox if the iou <= thr

order = order[inds + 1] #将order中的第inds+1处的值重新赋值给order；即更新保留下来的索引，加1是因为没有计算与自身的IOU，所以索引相差１，需要加上

bbox = bbox\_array[keep]

predicts\_dict[object\_name] = bbox.tolist()

return predicts\_dict

**书写卷积**

Input = [[40,24,135],[200,239,238],[90,34,94]]

kernel = [[0.0,0.6],[0.1,0.3]]

def my\_conv(input,kernel):

output\_size = (len(input)-len(kernel)+1)

res = [[0]\*output\_size for i in range(output\_size)]

for i in range(len(res)):

for j in range(len(res)):

res[i][j] = int(compute\_conv(input,kernel,i,j))

return res

def compute\_conv(input,kernel,i,j):

res = 0

for kk in range(len(kernel)):

for k in range(len(kernel[0])):

res +=int(input[i+kk][j+k]) \* float(kernel[kk][k])

return res

print my\_conv(Input, kernel)

**平方根** 二分法

def mySqrt(x):

if x <= 1:

return x

l, r = 0, x

while True:

mid = round((r+l)/2)

if mid\*\*2 <= x < (mid+1)\*\*2:

break

elif mid\*\*2 < x:

l=mid

else:

r=mid

return mid

**判断是否素数**

def fun(n):

if n % 2 == 0:

return False

for i in range(3, int(np.sqrt(n)), 2): # **不能被2整除,那接着看3～√n 且每次+2**

if n % i == 0:

return False

return True

删除链表的重复元素 1->2->2->3->4->5->5 --> 1->3>4

class Solution:

def deleteDuplicates(self, pHead):

if not pHead or not pHead.next:

return pHead

**new\_head = ListNode(-1)**  # 在头节点前再加一个节点

new\_head.next = pHead

pre = new\_head

p = pHead # pre->p->nex

nex = None

while p and p.next:

nex = p.next

if p.val == nex.val:

**while nex and nex.val == p.val:**

nex = nex.next

**pre.next = nex**  # pre的next直接连接nex 中间重复的都删掉了

p = nex

else:

pre = p

p = p.next

return new\_head.next

**leetcode92 ac**

**链表指定index反转** 1->2->3->4->5 ---> 1->4->3->2->5 m=2 n=4

class Solution(object):

# reverse()辅助函数

def reverse(self, head):

pre = None

while head:

nxt = head.next

head.next = pre # 反指针操作

pre = head

head = nxt

return pre

# 找第k个节点 因为链表不连续 所以只能一直next的找 不像数组可以直接index得到

def findkth(self, head, k):

for i in range(k):

if head:

head = head.next

else:

return None

return head

def reverseBetween(self, head, m, n):

dummy = ListNode(-1)

dummy.next = head # dummy是新init的一个head之前的节点

mth\_pre = self.findkth(dummy, m-1)

mth = mth\_pre.next

nth = self.findkth(dummy, n)

nth\_next = nth.next

nth.next = None

self.reverse(mth)

# 下面两句是精髓 画个链表图会清晰很多

mth\_pre.next = nth

mth.next = nth\_next

return dummy.next

**链表局部排序 leetcode86**

class Solution(object):

def partition(self, head, x):

# **init两个虚拟指针 p1负责<x的移动 p2负责>=x的移动**

p1 = ListNode(-1)

p2 = ListNode(-1)

tmp\_p1 = p1

tmp\_p2 = p2

while head:

if head.val < x:

p1.next = ListNode(head.val) # 注意是p1.next

p1 = p1.next

else:

p2.next = ListNode(head.val)

p2 = p2.next

head = head.next

p1.next = tmp\_p2.next # 移好的p1接到p2前去

return tmp\_p1.next

**重排链表** 1->2->3->4->5, 重新排列为 1->5->2->4->3 pop(0) and pop()完成

class Solution(object):

def reorderList(self, head):

if not head or not head.next:

return head

cur = head

stack = []

while cur:

stack.append(cur)

cur = cur.next

cur = stack.pop(0) # 弹出第一个

while stack:

**cur.next** = stack.pop() # next的赋值

cur = cur.next

if stack:

**cur.next** = stack.pop(0)

cur=cur.next

cur.next = None # 走到最末了

**链表排序 快排and归并**

# 归并排序

class Solution:

def sortList(self, head):

if not head or not head.next:

return head

mid = self.get\_mid(head)

l = head

r = mid.next

mid.next = None

return self.merge(self.sortList(l), self.sortList(r))

def merge(self, p, q):

tmp = ListNode(0)

h = tmp

# p q 分别是两段的头节点

while p and q:

if p.val < q.val:

h.next = p

p = p.next

else:

h.next = q

q = q.next

h = h.next

if p:

h.next = p

if q:

h.next = q

return tmp.next

def get\_mid(self, node):

if not node:

return node

fast = slow = node

while fast.next and fast.next.next:

slow = slow.next

fast = fast.next.next

return slow

# 快排

class Solution:

def sortList(self,phead):

if not phead:

return None

else:

self.quicksort(phead,None) # head and end

return phead

# p1 p2 节点交换

def swap(self,node1,node2):

tem = node1.val

node1.val = node2.val

node2.val = tem

def quicksort(self,head,end):

if head != end:

key = head.val

p = head

q = head.next # p q 两指针

while q != end: # q 遍历除参考值外的所有节点

if q.val < key: # 出现节点的值小于参考值

p = p.next # 先把p前移一位，再给这个位置赋予刚刚q的值

self.swap(p,q)# 将q的值给p 使得p遍历的节点都小于key

q = q.next

self.swap(head,p) # 这一步别漏了，把key\_ind和之前的head互换 然后分两段使两段均有序

self.quicksort(head,p)

self.quicksort(p.next,end)

**快慢指针找环入口**

首先快慢指针，fast=2\*slow fast和slow相遇，则fast比slow多走了一个环才会相遇的

所以slow此时是走到了一个环长度的位置

那现在把fast放到head处，slow fast同步走，下一次相遇是slow比fast多走一个环长度的位置

即正好是环的入口啊...

class Solution:

def detectCycle(self, pHead):

if not pHead or not pHead.next or not pHead.next.next:

return None # 链表长度为012，都直接None

# init 快慢指针 和找mid不同，找mid的话，init: fast=slow=head

**fast = pHead.next.next**

**slow = pHead.next**

# 判断有无环

while fast!=slow:

if fast.next and fast.next.next:

fast=fast.next.next

slow=slow.next

else:

return -1 # 无环

# 跳出循环说明是有环的

fast = pHead # 现在的slow在fast前环长度

while fast!=slow:

fast=fast.next

slow=slow.next

return slow # 在环入口相遇

**有序链表转二叉搜索树**

class Solution(object):

def sortedListToBST(self, head):

if not head:

return None

fast, slow, **pre** = head, head, **None**

# 快慢指针找到中点，作为二叉搜索树的根节点 然后左右边递归

while fast and fast.next:

pre = slow

fast = fast.next.next

slow = slow.next

# 出循转是因为fast到尾了 此时slow正好是mid

root = TreeNode(slow.val)

if pre:

**pre.next = None**

root.left = self.sortedListToBST(**head**) # 注意这里是head不是root！！

root.right = self.sortedListToBST(slow.next)

return root

**两数组交集 用dict存储数组1元素及对应的个数 然后再遍历数组2**

class Solution(object):

def intersect(self, nums1, nums2):

res = []

if not nums1 or not nums2:

return res

dict1={}.fromkeys(nums1,0) # nums1的值作为key，value init为0

for num in nums1:

dict1[num] += 1

for num in nums2:

if num in dict1.keys() **and dict1[num]**: # and dict1[num] 为了确保dict1中的value还>0

res.append(num)

dict1[num] -= 1 # 这里dict1自减很重要

return res

**数组峰值 二分 logn时复**

def fengIndex(nums):

l,r = 0, len(nums)-1

while l<=r:

mid = (l+r)/2

if nums[mid]<nums[mid-1]: # mid选大了，数组已经在递减了

r = mid

elif nums[mid]<nums[mid+1]: # mid选小了，数组还在递增

l = mid

else:

return mid