**大家准备面试从以下方面入手**

**Python：列表，字典，三大器，元类（引入Django ORM）**

**MySQL：隔离级别，脏读幻读，优化，索引**

**Redis：数据类型及应用场景，增删改查方法，击穿，穿透，雪崩及解决方法（布隆过滤器）**

**Django：流程，中间件**

**DRF：可以准备下源码，基本没问过**

**Celery：Broker，Backend，分布式系统下怎么布置**

**Docker：Dockerfile中ADD，COPY，RUN，CMD等，镜像过大解决方法，docker-compose**

**Linux常用指令：查看进程，磁盘，端口等**

**Git：基本没遇到问的，可以准备一写基本的**

**算法：快排，重复数据多的优化方法**

**Python**

**009、栈和堆的区别是什么？**

1. 申请方式的不同。栈由系统自动分配，而堆是人为申请开辟；
2. 申请大小的不同。栈获得的空间较小，而堆获得的空间较大；
3. 申请效率的不同。栈速度较快，堆速度比较慢;
4. 底层不同。栈是连续的空间，堆是不连续的空间，是一棵完全二叉树。
5. 存储内容的不同。

栈在函数调用时，第一个进栈的是主函数中的下一条指令的地址，然后是函数的各个参数，== 在大多数C编译器中，参数是由右向左入栈的，然后是函数中的局部变量，注意静态变量是不入栈的，静态变量存储在静态存储区。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行；堆一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

**03.堆、栈、队列之间的区别？**

1. 堆是在程序运行时，而不是在程序编译时，申请某个大小的内存空间。即动态分配内存，对其访问和对一般内存的访问没有区别。
2. 栈就是一个桶，后放进去的先拿出来，它下面本来有的东西要等它出来之后才能出来。（后进先出）；
3. 队列只能在队头做删除操作,在队尾做插入操作.而栈只能在栈顶做插入和删除操作。（先进先出）；

**04.简述数组、链表、队列、堆栈的区别？**

数组和链表是存储方式的概念，数组在连续的空间中存储数据，链表在非连续的空间中存储数据；

队列和堆栈是描述数据存取方法的概念，队列是先进先出，而堆栈是后进后出，队列和堆栈可以用链表来实现，也可以用数组来实现；

**131、手写一个栈？**

#给一个点，我们能够根据这个点知道一些内容  
class Node(object):  
   def \_\_init\_\_(self,val): #定位的点的值和一个指向  
       self.val=val #指向元素的值,原队列第二元素  
       self.next=None #指向的指针  
         
class stack(object):  
   def \_\_init\_\_(self):  
       self.top=None #初始化最开始的位置  
         
   def push(self,n):#添加到栈中  
       n=Node(n) #实例化节点  
       n.next=self.top #顶端元素传值给一个指针  
       self.top=n  
       return n.val  
         
   def pop(self): #退出栈  
       if self.top == None:  
           return None  
       else:  
           tmp=self.top.val  
           self.top=self.top.next #下移一位，进行  
           return tmp  
             
if \_\_name\_\_=="\_\_main\_\_":  
   s=stack()  
   print(s.pop())  
   s.push(1)  
   print(s.pop())  
   s.push(2)  
   s.push(3)  
   print(s.pop())  
   s.push(3)  
   s.push(3)  
   s.push(3)  
   print(s.pop())  
   print(s.pop())  
   print(s.pop())  
   print(s.pop())

**132、使用两个队列实现一个栈？**

class Stack(object):  
   def \_\_init\_\_(self):  
       self.queueA=[]  
       self.queueB=[]  
   def push(self,node):  
       self.queueA.append(node)  
   def pop(self):  
       if len(self.queueA)==0:  
           return None  
       while len(self.queueA)!=1:  
           self.queueB.append(self.queueA.pop(0))  
       self.queueA,self.queueB=self.queueB,self.queueA  
       return self.queueB.pop()  
         
st=Stack()  
print(st.pop())  
st.push(1)  
print(st.pop())  
st.push(1)  
st.push(1)  
st.push(1)  
print(st.pop())  
print(st.pop())  
print(st.pop())

* 注意上面两个栈的实现方法，第一种的效率高，队列的这种方法效率低

**133、有如下链表类，请实现单链表逆置？**

class ListNode:

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

self.val=val

self.next=None

class Solution:

def reverseList(self,pHead):

if not pHead or not pHead.next:

return pHead

last=None

while pHead:

tmp=pHead.next

pHead.next=last

last=pHead

pHead=tmp

return last

**135、手写一个队列？**

class Queue(object):

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

self.queue=[]

self.size=size

def is\_empty(self):

return not bool(len(self.queue))

def is\_full(self):

return len(self.queue)==self.size

def enqueue(self,val):

if not self.is\_full():

self.queue.insert(0,val)

return True

return False

def dequeue(self):

if not self.is\_empty():

return self.queue.pop()

return None

s=Queue(2)

print(s.is\_empty)

s.enqueue(1)

s.enqueue(2)

print(s.is\_full())

print(s.dequeue())

print(s.dequeue())

print(s.is\_empty())

**000、红黑树？**

红黑树与AVL的比较：

AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；

红黑是用非严格的平衡来换取增删节点时候旋转次数的降低；

所以简单说，如果你的应用中，搜索的次数远远大于插入和删除，那么选择AVL，如果搜索，插入删除次数几乎差不多，应该选择RB。

红黑树详解: <https://xieguanglei.github.io/blog/post/red-black-tree.html>

教你透彻了解红黑树: <https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md>

**沉默是金** [**https://blog.markhoo.com**](https://blog.markhoo.com)

**001台阶问题/斐波那契？**

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

fib = lambda n: n if n <= 2 else fib(n - 1) + fib(n - 2)

第二种记忆方法

def memo(func):

cache = {}

def wrap(\*args):

if args not in cache:

cache[args] = func(\*args)

return cache[args]

return wrap

@memo

def fib(i):

if i < 2:

return 1

return fib(i-1) + fib(i-2)

第三种方法

def fib(n):

a, b = 0, 1

for \_ in xrange(n):

a, b = b, a + b

return b

**002、变态台阶问题？**

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

fib = lambda n: n if n < 2 else 2 \* fib(n - 1)

**003、矩形覆盖？**

我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

第2\*n个矩形的覆盖方法等于第2\*(n-1)加上第2\*(n-2)的方法。

f = lambda n: 1 if n < 2 else f(n - 1) + f(n - 2)

**004、杨氏矩阵查找？**

在一个m行n列二维数组中，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。

使用Step-wise线性搜索。

def get\_value(l, r, c):

return l[r][c]

def find(l, x):

m = len(l) - 1

n = len(l[0]) - 1

r = 0

c = n

while c >= 0 and r <= m:

value = get\_value(l, r, c)

if value == x:

return True

elif value > x:

c = c - 1

elif value < x:

r = r + 1

return False

**005、去除列表中的重复元素？**

用集合

list(set(l))

用字典

l1 = ['b','c','d','b','c','a','a']

l2 = {}.fromkeys(l1).keys()

print l2

用字典并保持顺序

l1 = ['b','c','d','b','c','a','a']

l2 = list(set(l1))

l2.sort(key=l1.index)

print l2

列表推导式

l1 = ['b','c','d','b','c','a','a']

l2 = []

[l2.append(i) for i in l1 if not i in l2]

sorted排序并且用列表推导式.

l = ['b','c','d','b','c','a','a']  
[single.append(i) for i in sorted(l) if i not in single]  
print single

**006、链表成对调换？**

1->2->3->4转换成2->1->4->3.

class ListNode:

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

self.val = x

self.next = None

class Solution:

# @param a ListNode

# @return a ListNode

def swapPairs(self, head):

if head != None and head.next != None:

next = head.next

head.next = self.swapPairs(next.next)

next.next = head

return next

return head

**007、创建字典的方法？**

**1 直接创建：**

dict = {'name':'earth', 'port':'80'}

**2 工厂方法：**

items=[('name','earth'),('port','80')]

dict2=dict(items)

dict1=dict((['name','earth'],['port','80']))

**3 fromkeys()方法：**

dict1={}.fromkeys(('x','y'),-1)

dict={'x':-1,'y':-1}

dict2={}.fromkeys(('x','y'))

dict2={'x':None, 'y':None}

**008、合并两个有序列表？**

知乎远程面试要求编程

尾递归

def \_recursion\_merge\_sort2(l1, l2, tmp):

if len(l1) == 0 or len(l2) == 0:

tmp.extend(l1)

tmp.extend(l2)

return tmp

else:

if l1[0] < l2[0]:

tmp.append(l1[0])

del l1[0]

else:

tmp.append(l2[0])

del l2[0]

return \_recursion\_merge\_sort2(l1, l2, tmp)

def recursion\_merge\_sort2(l1, l2):

return \_recursion\_merge\_sort2(l1, l2, [])

循环算法

思路：

1、定义一个新的空列表；

2、比较两个列表的首个元素；

3、小的就插入到新列表里；

4、把已经插入新列表的元素从旧列表删除；

5、直到两个旧列表有一个为空；

6、再把旧列表加到新列表后面；

def loop\_merge\_sort(l1, l2):

tmp = []

while len(l1) > 0 and len(l2) > 0:

if l1[0] < l2[0]:

tmp.append(l1[0])

del l1[0]

else:

tmp.append(l2[0])

del l2[0]

tmp.extend(l1)

tmp.extend(l2)

return tmp

pop弹出

a = [1,2,3,7]

b = [3,4,5]

def merge\_sortedlist(a,b):

c = []

while a and b:

if a[0] >= b[0]:

c.append(b.pop(0))

else:

c.append(a.pop(0))

while a:

c.append(a.pop(0))

while b:

c.append(b.pop(0))

return c

print merge\_sortedlist(a,b)

**022、两个字符串是否是变位词？**

class Anagram:

"""

@:param s1: The first string

@:param s2: The second string

@:return true or false

"""

def Solution1(s1,s2):

alist = list(s2)

pos1 = 0

stillOK = True

while pos1 < len(s1) and stillOK:

pos2 = 0

found = False

while pos2 < len(alist) and not found:

if s1[pos1] == alist[pos2]:

found = True

else:

pos2 = pos2 + 1

if found:

alist[pos2] = None

else:

stillOK = False

pos1 = pos1 + 1

return stillOK

print(Solution1('abcd','dcba'))

def Solution2(s1,s2):

alist1 = list(s1)

alist2 = list(s2)

alist1.sort()

alist2.sort()

pos = 0

matches = True

while pos < len(s1) and matches:

if alist1[pos] == alist2[pos]:

pos = pos + 1

else:

matches = False

return matches

print(Solution2('abcde','edcbg'))

def Solution3(s1,s2):

c1 = [0]\*26

c2 = [0]\*26

for i in range(len(s1)):

pos = ord(s1[i])-ord('a')

c1[pos] = c1[pos] + 1

for i in range(len(s2)):

pos = ord(s2[i])-ord('a')

c2[pos] = c2[pos] + 1

j = 0

stillOK = True

while j<26 and stillOK:

if c1[j] == c2[j]:

j = j + 1

else:

stillOK = False

return stillOK

print(Solution3('apple','pleap'))

**009、交叉链表求交点？**

其实思想可以按照从尾开始比较两个链表，如果相交，则从尾开始必然一致，只要从尾开始比较，直至不一致的地方即为交叉点，如图所示略：

# 使用a,b两个list来模拟链表，可以看出交叉点是 7这个节点

a = [1,2,3,7,9,1,5]

b = [4,5,7,9,1,5]

for i in range(1,min(len(a),len(b))):

if i==1 and (a[-1] != b[-1]):

print "No"

break

else:

if a[-i] != b[-i]:

print "交叉节点：",a[-i+1]

break

else:

pass

另外一种比较正规的方法，构造链表类

class ListNode:

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

self.val = x

self.next = None

def node(l1, l2):

length1, lenth2 = 0, 0

# 求两个链表长度

while l1.next:

l1 = l1.next

length1 += 1

while l2.next:

l2 = l2.next

length2 += 1

# 长的链表先走

if length1 > lenth2:

for \_ in range(length1 - length2):

l1 = l1.next

else:

for \_ in range(length2 - length1):

l2 = l2.next

while l1 and l2:

if l1.next == l2.next:

return l1.next

else:

l1 = l1.next

l2 = l2.next

修改了一下:

#coding:utf-8

class ListNode:

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

self.val = x

self.next = None

def node(l1, l2):

length1, length2 = 0, 0

# 求两个链表长度

while l1.next:

l1 = l1.next#尾节点

length1 += 1

while l2.next:

l2 = l2.next#尾节点

length2 += 1

#如果相交

if l1.next == l2.next:

# 长的链表先走

if length1 > length2:

for \_ in range(length1 - length2):

l1 = l1.next

return l1#返回交点

else:

for \_ in range(length2 - length1):

l2 = l2.next

return l2#返回交点

# 如果不相交

else:

return

思路: <http://humaoli.blog.163.com/blog/static/13346651820141125102125995/>

**010、二分查找？**

#coding:utf-8

def binary\_search(list,item):

low = 0

high = len(list)-1

while low<=high:

mid = (low+high)/2

guess = list[mid]

if guess>item:

high = mid-1

elif guess<item:

low = mid+1

else:

return mid

return None

mylist = [1,3,5,7,9]

print binary\_search(mylist,3)

参考: <http://blog.csdn.net/u013205877/article/details/76411718>

**011、快排？**

#coding:utf-8

def quicksort(list):

if len(list)<2:

return list

else:

midpivot = list[0]

lessbeforemidpivot = [i for i in list[1:] if i<=midpivot]

biggerafterpivot = [i for i in list[1:] if i > midpivot]

finallylist = quicksort(lessbeforemidpivot)+[midpivot]+quicksort(biggerafterpivot)

return finallylist

print quicksort([2,4,6,7,1,2,5])

更多排序问题可见：[数据结构与算法-排序篇-Python描述](http://blog.csdn.net/mrlevo520/article/details/77829204)

**012、找零问题？**

# coding:utf-8

# values是硬币的面值values = [ 25, 21, 10, 5, 1]

# valuesCounts 钱币对应的种类数

# money 找出来的总钱数

# coinsUsed 对应于目前钱币总数i所使用的硬币数目

def coinChange(values,valuesCounts,money,coinsUsed):

#遍历出从1到money所有的钱数可能

for cents in range(1,money+1):

minCoins = cents

#把所有的硬币面值遍历出来和钱数做对比

for kind in range(0,valuesCounts):

if (values[kind] <= cents):

temp = coinsUsed[cents - values[kind]] +1

if (temp < minCoins):

minCoins = temp

coinsUsed[cents] = minCoins

print ('面值:{0}的最少硬币使用数为:{1}'.format(cents, coinsUsed[cents]))

思路: <http://blog.csdn.net/wdxin1322/article/details/9501163>

方法: <http://www.cnblogs.com/ChenxofHit/archive/2011/03/18/1988431.html>

**013、广度遍历和深度遍历二叉树？**

给定一个数组，构建二叉树，并且按层次打印这个二叉树

**014、二叉树节点？**

class Node(object):

def \_\_init\_\_(self, data, left=None, right=None):

self.data = data

self.left = left

self.right = right

tree = Node(1, Node(3, Node(7, Node(0)), Node(6)), Node(2, Node(5), Node(4)))

**015、层次遍历？**

def lookup(root):

row = [root]

while row:

print(row)

row = [kid for item in row for kid in (item.left, item.right) if kid]

**016、深度遍历？**

def deep(root):

if not root:

return

print root.data

deep(root.left)

deep(root.right)

if \_\_name\_\_ == '\_\_main\_\_':

lookup(tree)

deep(tree)

**017、前中后序遍历？**

深度遍历改变顺序就OK了

# coding:utf-8

# 二叉树的遍历

# 简单的二叉树节点类

class Node(object):

def \_\_init\_\_(self,value,left,right):

self.value = value

self.left = left

self.right = right

#中序遍历:遍历左子树,访问当前节点,遍历右子树

def mid\_travelsal(root):

if root.left is None:

mid\_travelsal(root.left)

#访问当前节点

print(root.value)

if root.right is not None:

mid\_travelsal(root.right)

#前序遍历:访问当前节点,遍历左子树,遍历右子树

def pre\_travelsal(root):

print (root.value)

if root.left is not None:

pre\_travelsal(root.left)

if root.right is not None:

pre\_travelsal(root.right)

#后续遍历:遍历左子树,遍历右子树,访问当前节点

def post\_trvelsal(root):

if root.left is not None:

post\_trvelsal(root.left)

if root.right is not None:

post\_trvelsal(root.right)

print (root.value)

**018、求最大树深？**

def maxDepth(root):

if not root:

return 0

return max(maxDepth(root.left), maxDepth(root.right)) + 1

**019、求两棵树是否相同？**

def isSameTree(p, q):

if p == None and q == None:

return True

elif p and q :

return p.val == q.val and isSameTree(p.left,q.left) and isSameTree(p.right,q.right)

else :

return False

**020、前序中序求后序？**

推荐: <http://blog.csdn.net/hinyunsin/article/details/6315502>

def rebuild(pre, center):

if not pre:

return

cur = Node(pre[0])

index = center.index(pre[0])

cur.left = rebuild(pre[1:index + 1], center[:index])

cur.right = rebuild(pre[index + 1:], center[index + 1:])

return cur

def deep(root):

if not root:

return

deep(root.left)

deep(root.right)

print root.data

**021、单链表逆置？**

class Node(object):

def \_\_init\_\_(self, data=None, next=None):

self.data = data

self.next = next

link = Node(1, Node(2, Node(3, Node(4, Node(5, Node(6, Node(7, Node(8, Node(9)))))))))

def rev(link):

pre = link

cur = link.next

pre.next = None

while cur:

tmp = cur.next

cur.next = pre

pre = cur

cur = tmp

return pre

root = rev(link)

while root:

print root.data

root = root.next

思路: <http://blog.csdn.net/feliciafay/article/details/6841115>

方法: <http://www.xuebuyuan.com/2066385.html?mobile=1>

**022、一个保存整数（int）的数组，除了一个元素出现过1次外，其他元素都出现过两次，请找出这个元素？答：item = [i for i in list1 if list1.count(i) == 1]**

**023、个外观相同的篮球，其中1个的重要和其他11个的重量不同（有可能轻有可能重），现在有一个天平可以使用，怎样才能通过最少的称重次数找出这颗与众不同的球？**

将球编号，分为3组，每组4个（一）ABCD （二）EFGH （三）IJKL

情况一

1）ABCD与EFGH比，若ABCD=EFGH，说明ABCDEFGH全正常

2）IJ与LA(1已证明A正常)，K放一边，若IJ=LA，说明K不正常

3）将K与A比即可知K是轻是重

情况二

1）同情况一

2）若IJ小于LA，说明K正常，IJ有一轻或L重

3）I与J比，I=J则L不正常，重

[TOC]

数据类型的堆栈存储？

　　　堆栈是一个后进先出的数据结构，其工作方式就像一堆汽车排队进去一个死胡同里面，最先进去的一定是最后出来。

　　　队列是一种先进先出的数据类型，它的跟踪原理类似于在超市收银处排队，队列里的的第一个人首先接受服务，新的元素通过入队的方式添加到队列的末尾，而出队就是将队列的头元素删除。

　　　栈：是一种容器，可存入数据元素、访问元素、删除元素

　　　特点：只能从顶部插入（入栈）数据和删除（出栈）数据

　　　原理：LIFO(Last In First Out)后进先出  栈可以使用顺序表实现也可使用链表实现 使用python列表实现代码：  class Stack(object):   """   栈   使用python列表实现   """

  　　　　def \_\_init\_\_(self):    self.items = list()

 　　　　 def is\_empty(self):    """判空"""    return self.items == []

  　　　　def size(self):    """获取栈元素个数"""    return len(self.items)

  　　　　def push(self, item):    """入栈"""    self.items.append(item)

  　　　　def pop(self):    """出栈"""    self.items.pop()

 　　 　　def peek(self):    """获取栈顶元素"""    if self.is\_empty():     raise IndexError("stack is empty")    return self.items[-1]

**判断括号是否合法，类似于leetcode上的，[](){}，每个括号都要成对**

**一 问题描述：判断输入的小括号、中括号、大括号，是否是合规？**

**二 解答思路：数据结构+算法**

**直接上来考虑是使用列表、元组还是散列表不是太容易考虑，所以可以先考查算法**

**1 算法：怎么样才算合规？（1）符号的左端先出现，若直接出现右端部分，立马判断不合规；（2）如果符号的左端出现了（例如一个 '(' ），那么继续往里填充下一个符号，下一个符号如果是右端符号，则立马判断是不是能够跟第一个符号进行配对，如果配对成功，则可以删除这一对符号，如果配对不成功，则说明不是一对符号，立马判断不合规；但如果下一个符号仍然是左端符号，那么需要继续填充，直到最终填充结束，最后判断一次存储是否为空，空就是合规，不空就是不合规。（3）同时，第二步中的配对成功之后，如果继续填充至末尾，那么也需要进行存储是否为空的判断。**

**2 数据结构：上述算法要求，先入后出，逐渐配对合并消除，这是栈的思想，因此可以选择使用栈来作为数据结构。（示意见下图）**

**三 伪代码实现：**

**S = "{({[])"**

**satck = []**

**pa = {'}':'{',']':'[',')':'('}**

**for ret in S:**

**if ret is not in pa:**

**append ret into stack**

**elif stack is not empty or pa[ret] != stack.pop() # stack最后一位与下一个ret的值匹配不上，那么就输出Ｆａｌｓｅ**

**return False**

**# 如果最终匹配成对结束或者程序结束之后，仍然stack非空，那么返回Ｆａｌｓｅ**

**return not stack # stack 如果是空，那么返回的值就是True,如果非空，返回为False**

**四 python 实现：**

**第一种代码：**

**#!/usr/bin/env python3**

**# -\*- coding: utf-8 -\*-**

**"""**

**Created on Thu Jun 27 08:39:58 2019**

**@author: cui**

**"""**

**class legal(object):**

**def isValid(self,strings):**

**stack =[]**

**pa = {']':'[','}':'{',')':'('}**

**for i in strings:**

**if i not in pa:**

**stack.append(i)**

**elif not stack or pa[i] != stack.pop():**

**return False**

**return not stack**

**SS = legal()**

**print(SS.isValid('(((({{{{{[[[[]]]]]'))**

**第二种代码**

**@greg 2017-08-23**

**判断字符串中的括号是否为成对的 –> () {} [] {([])} 均为有效的**

**Given a string containing just the characters ‘(‘, ‘)’, ‘{‘, ‘}’, ‘[’ and ‘]’,**

**determine if the input string is valid.**

**The brackets must close in the correct order, “()” and “()[]{}” are all valid but “(]” and “([)]” are not.**

**class Solution():**

**'''**

**首先生成一个空列表和一个字典，字典中的键和值分别为各种括号的右边和左边的样式，**

**首先判断字符是否属于字典中的值，**

**如果属于字典中的值，则加入列表中，**

**如果不是判断是否属于字典的键，此时再进行判断：**

**如果列表为空，因为该字符是括号的右边样式，如果列表为空，则说明没有与其配对的左边样式，所以为False**

**如果列表为中的最后一个元素与该字符在字典中对应的键不相等（即该字符不能和列表的最后一个键配对），则返回False**

**如果字符不属于字典中的值，则为False**

**注意，每调用一次stack.pop()，stack中的最后一个字符就会弹出一次，也就是说如果所有的都成对，则最后列表是空的**

**'''**

**def isValid(self, s):**

**stack = []**

**dict = {'}' : '{', ']' : '[', ')' : '('}**

**for char in s:**

**if char in dict.values():**

**stack.append(char)**

**elif char in dict.keys():**

**if stack == [] or dict[char] != stack.pop():**

**return False**

**else:**

**return False**

**return stack == []**

**def isValid1(self, s):**

**stack = []**

**dict = {'{' : '}', '(' : ')', '[' : ']'}**

**for char in s:**

**if char in dict.keys():**

**stack.append(dict[char])**

**print(1, char, stack)**

**elif stack == [] or char != stack.pop():**

**print(2, char, stack)**

**return False**

**return stack == []**

**if \_\_name\_\_ == '\_\_main\_\_':**

**s = Solution()**

**str = '}{'**

**print(s.isValid1(str))**

**常用算法模块**

## 01.你知道的python排序算法？

1. 冒泡排序（Bubble Sort）
2. 选择排序（Selection Sort）
3. 插入排序（Insertion Sort）
4. 希尔排序（Shell Sort）
5. 归并排序（Merge Sort）
6. 快速排序（Quick Sort）
7. 堆排序（Heap Sort）
8. 计数排序（Counting Sort）
9. 桶排序（Bucket Sort）
10. 基数排序（Radix Sort）

## 02.你了解的高级语言中的垃圾回收机制有哪些？Python中用的是什么?

**03.介绍下你知道的缓存相关的算法?**

**04.介绍下你知道的负载均衡相关的算法?**

* **轮询** 将所有请求，依次分发到每台服务器上，适合服务器硬件相同的场景。 优点：服务器请求数目相同； 缺点：服务器压力不一样，不适合服务器配置不同的情况；
* **随机** 请求随机分配到各台服务器上。 优点：使用简单； 缺点：不适合机器配置不同的场景
* **最少链接** 将请求分配到连接数最少的服务器上（目前处理请求最少的服务器）。 优点：根据服务器当前的请求处理情况，动态分配； 缺点：算法实现相对复杂，需要监控服务器请求连接数；
* **Hash（源地址散列）** 根据IP地址进行Hash计算，得到IP地址。 优点：将来自同一IP地址的请求，同一会话期内，转发到相同的服务器；实现会话粘滞。 缺点：目标服务器宕机后，会话会丢失；
* **加权** 在轮询，随机，最少链接，Hash等算法的基础上，通过加权的方式，进行负载服务器分配。 优点：根据权重，调节转发服务器的请求数目； 缺点：使用相对复杂；

[**三种快排与四种优化**](https://www.cnblogs.com/2390624885a/p/7067669.html)

**1、快速排序的基本思想：**   
快速排序使用分治的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，以达到整个序列有序的目的。

**2、快速排序的三个步骤：**   
(1)选择基准：在待排序列中，按照某种方式挑出一个元素，作为 “基准”（pivot）   
(2)分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大   
(3)递归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

**3、选择基准的方式：**   
对于分治[算法](http://lib.csdn.net/base/datastructure)，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。

最理想的方法是，选择的基准恰好能把待排序序列分成两个等长的子序列

**我们介绍三种选择基准的方法：(3种)**

**方法(1)：固定位置**

思想：取序列的第一个或最后一个元素作为基准

基本的快速排序

1 int SelectPivot(int arr[],int low,int high)

2 {

3 return arr[low];//选择选取序列的第一个元素作为基准

4 }

注意：基本的快速排序选取第一个或最后一个元素作为基准。但是，这是一直很不好的处理方法。

[测试](http://lib.csdn.net/base/softwaretest)数据：   


测试数据分析：如果输入序列是随机的，处理时间可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。因为每次划分只能使待排序序列减一，此时为最坏情况，快速排序沦为起泡排序，时间复杂度为Θ(n^2)。而且，输入的数据是有序或部分有序的情况是相当常见的。因此，使用第一个元素作为枢纽元是非常糟糕的，为了避免这个情况，就引入了下面两个获取基准的方法。

**方法(2)：随机选取基准**

引入的原因：在待排序列是部分有序时，固定选取枢轴使快排效率底下，要缓解这种情况，就引入了随机选取枢轴

思想：取待排序列中任意一个元素作为基准

随机化算法

/\*随机选择枢轴的位置，区间在low和high之间\*/

int SelectPivotRandom(int arr[],int low,int high)

{

//产生枢轴的位置

srand((unsigned)time(NULL));

int pivotPos = rand()%(high - low) + low;

//把枢轴位置的元素和low位置元素互换，此时可以和普通的快排一样调用划分函数

swap(arr[pivotPos],arr[low]);

return arr[low];

}

测试数据：   


测试数据分析：:这是一种相对安全的策略。由于枢轴的位置是随机的，那么产生的分割也不会总是会出现劣质的分割。在整个数组数字全相等时，仍然是最坏情况，时间复杂度是O(n^2）。实际上，随机化快速排序得到理论最坏情况的可能性仅为1/(2^n）。所以随机化快速排序可以对于绝大多数输入数据达到O(nlogn）的期望时间复杂度。一位前辈做出了一个精辟的总结：“随机化快速排序可以满足一个人一辈子的人品需求。”

**方法(3)：三数取中（median-of-three）**

**引入的原因：虽然随机选取枢轴时，减少出现不好分割的几率，但是还是最坏情况下还是O(n^2），要缓解这种情况，就引入了三数取中选取枢轴**

分析：最佳的划分是将待排序的序列分成等长的子序列，最佳的状态我们可以使用序列的中间的值，也就是第N/2个数。可是，这很难算出来，并且会明显减慢快速排序的速度。这样的中值的估计可以通过随机选取三个元素并用它们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。显然使用三数中值分割法消除了预排序输入的不好情形，并且减少快排大约14%的比较次数

**举例：待排序序列为：8 1 4 9 6 3 5 2 7 0**   
**左边为：8，右边为0，中间为6.**   
**我们这里取三个数排序后，中间那个数作为枢轴，则枢轴为6**

注意：在选取中轴值时，可以从由左中右三个中选取扩大到五个元素中或者更多元素中选取，一般的，会有（2t＋1）平均分区法（median-of-(2t+1)，三平均分区法英文为median-of-three）。

具体思想：对待排序序列中low、mid、high三个位置上数据进行排序，取他们中间的那个数据作为枢轴，并用0下标元素存储枢轴。

即：采用三数取中，并用0下标元素存储枢轴。

/\*函数作用：取待排序序列中low、mid、high三个位置上数据，选取他们中间的那个数据作为枢轴\*/

int SelectPivotMedianOfThree(int arr[],int low,int high)

{

int mid = low + ((high - low) >> 1);//计算数组中间的元素的下标

//使用三数取中法选择枢轴

if (arr[mid] > arr[high])//目标: arr[mid] <= arr[high]

{

swap(arr[mid],arr[high]);

}

if (arr[low] > arr[high])//目标: arr[low] <= arr[high]

{

swap(arr[low],arr[high]);

}

if (arr[mid] > arr[low]) //目标: arr[low] >= arr[mid]

{

swap(arr[mid],arr[low]);

}

//此时，arr[mid] <= arr[low] <= arr[high]

return arr[low];

//low的位置上保存这三个位置中间的值

//分割时可以直接使用low位置的元素作为枢轴，而不用改变分割函数了

}

测试数据：



测试数据分析：使用**三数取中**选择枢轴优势还是很明显的，但是还是处理不了重复数组

**4、四种优化方式：**

**优化1：当待排序序列的长度分割到一定大小后，使用插入排序**

原因：对于很小和部分有序的数组，快排不如插排好。当待排序序列的长度分割到一定大小后，继续分割的效率比插入排序要差，此时可以使用插排而不是快排

截止范围：待排序序列长度N = 10，虽然在5~20之间任一截止范围都有可能产生类似的结果，这种做法也避免了一些有害的退化情形。摘自《[数据结构](http://lib.csdn.net/base/datastructure)与算法分析》Mark Allen Weiness 著

if (high - low + 1 < 10)

{

InsertSort(arr,low,high);

return;

}//else时，正常执行快排

测试数据：



测试数据分析：针对随机数组，使用三数取中选择枢轴+插排，效率还是可以提高一点，真是针对已排序的数组，是没有任何用处的。因为待排序序列是已经有序的，那么每次划分只能使待排序序列减一。此时，插排是发挥不了作用的。所以这里看不到时间的减少。另外，三数取中选择枢轴+插排还是不能处理重复数组

**优化2：在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割**

举例：

待排序序列 1 4 6 7 6 6 7 6 8 6

三数取中选取枢轴：下标为4的数6

转换后，待分割序列：6 4 6 7 1 6 7 6 8 6

枢轴key：6

本次划分后，未对与key元素相等处理的结果：1 4 6 6 7 6 7 6 8 6

下次的两个子序列为：1 4 6 和 7 6 7 6 8 6

本次划分后，对与key元素相等处理的结果：1 4 6 6 6 6 6 7 8 7

下次的两个子序列为：1 4 和 7 8 7

经过对比，我们可以看出，在一次划分后，把与key相等的元素聚在一起，能减少迭代次数，效率会提高不少

**具体过程：在处理过程中，会有两个步骤**   
**第一步，在划分过程中，把与key相等元素放入数组的两端**   
**第二步，划分结束后，把与key相等的元素移到枢轴周围**

举例：

待排序序列 1 4 6 7 6 6 7 6 8 6

三数取中选取枢轴：下标为4的数6

转换后，待分割序列：6 4 6 7 1 6 7 6 8 6

枢轴key：6

第一步，在划分过程中，把与key相等元素放入数组的两端   
结果为：6 4 1 6(枢轴) 7 8 7 6 6 6

此时，与6相等的元素全放入在两端了

第二步，划分结束后，把与key相等的元素移到枢轴周围

结果为：1 4 66(枢轴) 6 6 6 7 8 7

此时，与6相等的元素全移到枢轴周围了

之后，在1 4 和 7 8 7两个子序列进行快排

void QSort(int arr[],int low,int high)

{

int first = low;

int last = high;

int left = low;

int right = high;

int leftLen = 0;

int rightLen = 0;

if (high - low + 1 < 10)

{

InsertSort(arr,low,high);

return;

}

//一次分割

int key = SelectPivotMedianOfThree(arr,low,high);//使用三数取中法选择枢轴

while(low < high)

{

while(high > low && arr[high] >= key)

{

if (arr[high] == key)//处理相等元素

{

swap(arr[right],arr[high]);

right--;

rightLen++;

}

high--;

}

arr[low] = arr[high];

while(high > low && arr[low] <= key)

{

if (arr[low] == key)

{

swap(arr[left],arr[low]);

left++;

leftLen++;

}

low++;

}

arr[high] = arr[low];

}

arr[low] = key;

//一次快排结束

//把与枢轴key相同的元素移到枢轴最终位置周围

int i = low - 1;

int j = first;

while(j < left && arr[i] != key)

{

swap(arr[i],arr[j]);

i--;

j++;

}

i = low + 1;

j = last;

while(j > right && arr[i] != key)

{

swap(arr[i],arr[j]);

i++;

j--;

}

QSort(arr,first,low - 1 - leftLen);

QSort(arr,low + 1 + rightLen,last);

}

测试数据：



测试数据分析：**三数取中选择枢轴+插排+聚集相等元素**的组合，效果竟然好的出奇。

原因：在数组中，如果有相等的元素，那么就可以减少不少冗余的划分。这点在重复数组中体现特别明显啊。

其实这里，插排的作用还是不怎么大的。

**优化3：优化递归操作**

快排函数在函数尾部有两次递归操作，我们可以对其使用尾递归优化

优点：如果待排序的序列划分极端不平衡，递归的深度将趋近于n，而栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归耗费的空间也越多。优化后，可以缩减堆栈深度，由原来的O(n)缩减为O(logn)，将会提高性能。

void QSort(int arr[],int low,int high)

{

int pivotPos = -1;

if (high - low + 1 < 10)

{

InsertSort(arr,low,high);

return;

}

while(low < high)

{

pivotPos = Partition(arr,low,high);

QSort(arr,low,pivot-1);

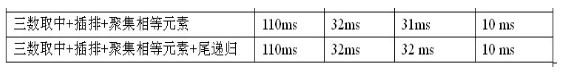
low = pivot + 1;

}

}

**注意：在第一次递归后，low就没用了，此时第二次递归可以使用循环代替**

测试数据：



测试数据分析：其实这种优化编译器会自己优化，相比不使用优化的方法，时间几乎没有减少

**优化4：使用并行或多线程处理子序列（略）**

所有的数据测试：



**概括：这里效率最好的快排组合 是：三数取中+插排+聚集相等元素,它和STL中的Sort函数效率差不多**

**注意：由于测试数据不稳定，数据也仅仅反应大概的情况。如果时间上没有成倍的增加或减少，仅仅有小额变化的话，我们可以看成时间差不多。**

**转载自： http://blog.csdn.net/hacker00011000/article/details/52176100**

**023、动态规划问题？**

可参考：[动态规划(DP)的整理-Python描述](http://blog.csdn.net/mrlevo520/article/details/75676160)