# 编程练习和面试真题答案

第1章 排序算法

### 1.11 编程练习

（1）代码如下：

def bubble\_sort(alist):

count = 0

for j in range(0,len(alist) - 1):

for i in range(0,len(alist) - 1 - j):

if alist[i] > alist[i + 1]:

alist[i],alist[i + 1] = alist[i + 1],alist[i]

else:

continue

count += 1

print(count)

（2）代码如下：

字母：

def bubbleSort(arr):

n = len(arr)

# 遍历所有数组元素

for i in range(n):

# Last i elements are already in place

for j in range(0, n-i-1):

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

arr[j], arr[j+1] = arr[j+1], arr[j]

arr = ['a','c','b','e','b','a','c','b']

bubbleSort(arr)

print(arr)

对象（键值对为例）：

按键排序：

m = {'m2':2, 'm1':4, 'm4':1,'m3':3,}

for k in sorted(m):

print(k,m[k])

按值排序：

m = {'m2':2, 'm1':4, 'm4':1,'m3':3,}

for k in sorted(m,key=m.\_\_getitem\_\_):

print(k,m[k])

### 1.12 面试真题

（1）答：最好的排序算法是快速排序，平均性能最好，不管是从数据规模大小，还是从数列的不同有序度来看，平均性能最优，平均时间复杂度为O（nlogn）空间复杂度为O(nlog2n)。

（2）答：选择算法时优先考虑时间复杂度，因为程序最重要是准确性，要实现预期功能，在涉及相当大量的数据处理用不同算法，运行时间差别很大，因此相比于空间使用的差距，需要优先对时间复杂度进行分析。

第2章 查找算法

### 2.9 编程练习

（1）答：不一定，查找速度和数据规模和所处位置有关，例如2 3 4 5 6，查找2，线性查找只需一次，二分查找需要2次，代码略。

（2）代码如下：

class SingleNode(object):

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

self.item = item

self.next = None

class SingleLinkList(object):

def \_\_init\_\_(self, node=None):

self.\_head = node

def travel(self):

cur = self.\_head

m = 0

while cur != None:

m = max(cur.item,m)

cur = cur.next

print(m)

### 2.10 面试真题

（1）答：取决于目标值与mid指向的元素的大小关系，若大于则向右查找并减小2个斐波那契区间，若小于则向左查找并减小1个斐波那契区间。

（2）答：

前序：6次

中序：5次

后序：4次

层次：6次

第3章 字符串算法

### 3.8 编程练习

（1）代码如下：

def string\_match\_boyer\_moore(string,match,start=0):

string\_len = len(string)

match\_len = len(match)

end = match\_len - 1

if string\_len < match\_len:

print ('Not Found')

return start;

while string[end] == match[end]:

end -= 1

if end == 0:

print ('Success Step:' + str(start+1))

return

idx = contain\_char(match,string[end])

shift = match\_len

if idx > -1:

shift = end - idx

start += 1

string\_match\_boyer\_moore(string[shift:],match,start)

def contain\_char(s,c):

for i in range(len(s)):

if c == s[i]:

return i

return -1

string\_match\_boyer\_moore("Fred fed Ted bread, and Ted fed Fred mead.", 'mead')

（2）代码如下：

class Algorithm:

  def getNext(self, T):

    i = 0

    j = -1

    next = [-1] \* len(T)

    while i < len(T)-1:

      if j == -1 or T[i] == T[j]:

        i += 1

        j += 1

        if i < len(T) and T[i] != T[j]:

          next[i] = j

        else:

          next[i] = next[j]

      else:

        j = next[j]

    return next

  def kmp(self, S, T):

    i = 0

    j = 0

    count = 0

    next = self.getNext(T)

    while i < len(S) and j < len(T):

      count = count + 1

      if j == -1 or S[i] == T[j]:

        i += 1

        j += 1

      else:

        j = next[j]

    if j == len(T):

      return count

    else:

      return -1

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

  text = 'Fred fed Ted bread,and Ted fed Fred mead.'

  key = 'mead'

  s = Algorithm()

  print( s.kmp(text, key))

（3）代码如下：

class Trie:

  def \_\_init\_\_(self):

    self.root = {}

    self.word\_end = -1

  def insert(self, word):

    curNode = self.root

    for c in word:

      if not c in curNode:

        curNode[c] = {}

      curNode = curNode[c]

    curNode[self.word\_end] = True

trie = Trie()

trie.insert('she')

trie.insert('shed')

trie.insert('all')

trie.insert('shally')

trie.insert('ally')

### 3.9 面试真题

（1）答：BM算法效率高，KMP的时间复杂度是O(n+m)，但实际上如果无法匹配成功，这个复杂度一般都是能跑满的，一般在实际应用中BM算法效率比KMP高

（2）答：平滑处理，当待预测样本中出现了训练集中未出现的特征的时候（文本分类中非常常见），应用上述公式，不管是哪个类别，最后结果都是0，So为了增加算法的泛化能力，我们会统一给每个特征值α加上一个固定值不宜过大，当α=1的时候我们称之为拉普拉斯平滑。

（3）答：比较两个字符串的哈希值前需要先进行计算。若字符串q长度为n,则计算q的哈希值的时间为O(n)。

（4）BM伪代码：

def BoyerMooreHorspool(pattern, text):  
    m = len(pattern)  
    n = len(text)  
    if m > n: return -1  
    skip = []  
    for k in range(256): skip.append(m)  
    for k in range(m - 1): skip[ord(pattern[k])] = m - k - 1  
    skip = tuple(skip)  
    k = m - 1  
    while k < n:  
        j = m - 1; i = k  
        while j >= 0 and text[i] == pattern[j]:  
            j -= 1; i -= 1  
        if j == -1: return i + 1  
        k += skip[ord(text[k])]  
    return -1

第4章 字符串算法

### 4.6 编程练习

（1）代码如下：

class Stack:

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

self.theStack=[]

def top(self):

if self.isEmpty():

return "Empty Stack"

else:

return self.theStack[-1]

def isEmpty(self):

return len(self.theStack)==0

def push(self,item):

self.theStack.[append](https://www.cnpython.com/list/append)(item)

def pop(self):

if not self.isEmpty():

temp=self.theStack[-1]

del(self.theStack[-1])

return temp

else:

return "Empty Stack"

（2）栈结构：

class LNode(object):

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

self.data = x

self.next = None

class MyStack(object):

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

#pHead = LNode

self.data = None

self.next = None

def is\_empty(self):

"""判断是否为空"""

if self.next == None:

return True

return False

def size(self):

"""返回栈的大小"""

size=0

p = self.next

while p != None:

# while p is not None:

p = p.next

size += 1

return size

def push(self, element):

"""压栈(加入元素)"""

p = LNode(element)

p.data = element

p.next = self.next

self.next = p

def pop(self):

"""弹栈(弹出元素)"""

tmp = self.next

if tmp != None:

self.next = tmp.next

print("栈已经为空")

return None

def top(self):

"""返回栈顶元素"""

if self.next != None:

return self.next.data

print("栈已经为空")

return None

队列结构：

class LNode(object):

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

self.data = x

self.next = None

class MyQueue(object):

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

"""分配头结点"""

self.pHead = None

self.pEnd = None

def is\_empty(self):

"""判断是否为空"""

if self.pHead == None:

return True

return False

def size(self):

"""获取队列的大小"""

size=0

p = self.pHead

while p != None:

# while p is not None:

p = p.next

size += 1

return size

def enQueue(self, element):

"""入队列，从队尾加"""

p = LNode(element)

p.data = element

p.next = None

if self.pHead == None:

self.pHead = self.pEnd=p

else:

self.pEnd.next = p

self.pEnd = p

def deQueue(self):

"""出队列，删除首元素"""

if self.pHead == None:

print("出队列失败，队列已经为空")

self.pHead = self.pHead.next

if self.pHead == None:

self.pEnd = None

def getFront(self):

"""返回队列首元素"""

if self.pHead == None:

print("获取队列首元素失败，队列已经为空")

return None

return self.pHead.data

def getBack(self):

"""返回队列尾元素"""

if self.pEnd == None:

print("获取队列尾元素失败，队列已经为空")

return None

return self.pEnd.data

### 4.7 面试真题

（1）答：31,16

代码如下：

def josephus(n,k):

List = list(range(1,n+1))

index = 0

while List:

temp = List.pop(0)

index += 1

if index == k:

index = 0

continue

List.append(temp)

if len(List)==2:

print(List)

break

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

josephus(41,3)

（2）答：栈和队列数据结构的特点是：栈特点就是一个先进后出的结构。队列特点就是一个先进先出的结构。

栈和队列的区别是：数据结构不同队列先进先出，栈先进后出。

对插入和删除操作的"限定"。 栈是限定只能在表的一端进行插入和删除操作的线性表。队列是限定只能在表的一端进行插入和在另一端进行删除操作的线性表。

遍历数据速度不同。栈只能从头部取数据 也就最先放入的需要遍历整个栈最后才能取出来，而且在遍历数据的时候还得为数据开辟临时空间，保持数据在遍历前的一致性队列怎不同，他基于地址指针进行遍历，而且可以从头或尾部开始遍历，但不能同时遍历，无需开辟临时空间。

应用：

栈：栈先进先出的特点，可以很好的控制访问控制，栈的数据访问是有很严格的，只能访问最后加入的数据，这对数据访问控制严格的应用很有好处。现实中，字符串倒序输出，使用栈的原理就可以很好的实现。

队列：队列可以模拟很多现实的生产环境，例如排队，队列是先进先出，不允许有任何元素插队，这对于解决现实生产问题有很大帮助。

第5章 树结构

### 5.9 编程练习

（1）B树代码如下:

class Node(object):

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

self.key1=key

self.key2=None

self.left=None

self.middle=None

self.right=None

def isLeaf(self):

return self.left is None and self.middle is None and self.right is None

def isFull(self):

return self.key2 is not None

def hasKey(self,key):

if (self.key1==key) or (self.key2 is not None and self.key2==key):

return True

else:

return False

def getChild(self,key):

if key<self.key1:

return self.left

elif self.key2 is None:

return self.middle

elif key<self.key2:

return self.middle

else:

return self.right

（2）B+树

from random import randint,choice

from bisect import bisect\_right,bisect\_left

from collections import deque

class InitError(Exception):

pass

class ParaError(Exception):

pass

class KeyValue(object):

\_\_slots\_\_=('key','value')

def \_\_init\_\_(self,key,value):

self.key=key

self.value=value

def \_\_str\_\_(self):

return str((self.key,self.value))

def \_\_cmp\_\_(self,key):

if self.key>key:

return 1

elif self.key==key:

return 0

else:

return -1

class Bptree\_InterNode(object):

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

if not isinstance(M,int):

raise InitError,'M must be int'

if M<=3:

raise InitError,'M must be greater then 3'

else:

self.\_\_M=M

self.clist=[]

self.ilist=[]

self.par=None

def isleaf(self):

return False

def isfull(self):

return len(self.ilist)>=self.M-1

def isempty(self):

return len(self.ilist)<=(self.M+1)/2-1

@property

def M(self):

return self.\_\_M

class Bptree\_Leaf(object):

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

if not isinstance(L,int):

raise InitError,'L must be int'

else:

self.\_\_L=L

self.vlist=[]

self.bro=None

self.par=None

def isleaf(self):

return True

def isfull(self):

return len(self.vlist)>self.L

def isempty(self):

return len(self.vlist)<=(self.L+1)/2

@property

def L(self):

return self.\_\_L

class Bptree(object):

def \_\_init\_\_(self,M,L):

if L>M:

raise InitError,'L must be less or equal then M'

else:

self.\_\_M=M

self.\_\_L=L

self.\_\_root=Bptree\_Leaf(L)

self.\_\_leaf=self.\_\_root

@property

def M(self):

return self.\_\_M

@property

def L(self):

return self.\_\_L

def insert(self,key\_value):

node=self.\_\_root

def split\_node(n1):

mid=self.M/2

newnode=Bptree\_InterNode(self.M)

newnode.ilist=n1.ilist[mid:]

newnode.clist=n1.clist[mid:]

newnode.par=n1.par

for c in newnode.clist:

c.par=newnode

if n1.par is None:

newroot=Bptree\_InterNode(self.M)

newroot.ilist=[n1.ilist[mid-1]]

newroot.clist=[n1,newnode]

n1.par=newnode.par=newroot

self.\_\_root=newroot

else:

i=n1.par.clist.index(n1)

n1.par.ilist.insert(i,n1.ilist[mid-1])

n1.par.clist.insert(i+1,newnode)

n1.ilist=n1.ilist[:mid-1]

n1.clist=n1.clist[:mid]

return n1.par

def split\_leaf(n2):

mid=(self.L+1)/2

newleaf=Bptree\_Leaf(self.L)

newleaf.vlist=n2.vlist[mid:]

if n2.par==None:

newroot=Bptree\_InterNode(self.M)

newroot.ilist=[n2.vlist[mid].key]

newroot.clist=[n2,newleaf]

n2.par=newleaf.par=newroot

self.\_\_root=newroot

else:

i=n2.par.clist.index(n2)

n2.par.ilist.insert(i,n2.vlist[mid].key)

n2.par.clist.insert(i+1,newleaf)

newleaf.par=n2.par

n2.vlist=n2.vlist[:mid]

n2.bro=newleaf

def insert\_node(n):

if not n.isleaf():

if n.isfull():

insert\_node(split\_node(n))

else:

p=bisect\_right(n.ilist,key\_value)

insert\_node(n.clist[p])

else:

p=bisect\_right(n.vlist,key\_value)

n.vlist.insert(p,key\_value)

if n.isfull():

split\_leaf(n)

else:

return

insert\_node(node)

def search(self,mi=None,ma=None):

result=[]

node=self.\_\_root

leaf=self.\_\_leaf

if mi is None and ma is None:

raise ParaError,'you need to setup searching range'

elif mi is not None and ma is not None and mi>ma:

raise ParaError,'upper bound must be greater or equal than lower bound'

def search\_key(n,k):

if n.isleaf():

p=bisect\_left(n.vlist,k)

return (p,n)

else:

p=bisect\_right(n.ilist,k)

return search\_key(n.clist[p],k)

if mi is None:

while True:

for kv in leaf.vlist:

if kv<=ma:

result.append(kv)

else:

return result

if leaf.bro==None:

return result

else:

leaf=leaf.bro

elif ma is None:

index,leaf=search\_key(node,mi)

result.extend(leaf.vlist[index:])

while True:

if leaf.bro==None:

return result

else:

leaf=leaf.bro

result.extend(leaf.vlist)

else:

if mi==ma:

i,l=search\_key(node,mi)

try:

if l.vlist[i]==mi:

result.append(l.vlist[i])

return result

else:

return result

except IndexError:

return result

else:

i1,l1=search\_key(node,mi)

i2,l2=search\_key(node,ma)

if l1 is l2:

if i1==i2:

return result

else:

result.extend(l.vlist[i1:i2])

return result

else:

result.extend(l1.vlist[i1:])

l=l1

while True:

if l.bro==l2:

result.extend(l2.vlist[:i2+1])

return result

else:

result.extend(l.bro.vlist)

l=l.bro

def traversal(self):

result=[]

l=self.\_\_leaf

while True:

result.extend(l.vlist)

if l.bro==None:

return result

else:

l=l.bro

def show(self):

print 'this b+tree is:\n'

q=deque()

h=0

q.append([self.\_\_root,h])

while True:

try:

w,hei=q.popleft()

except IndexError:

return

else:

if not w.isleaf():

print w.ilist,'the height is',hei

if hei==h:

h+=1

q.extend([[i,h] for i in w.clist])

else:

print [v.key for v in w.vlist],'the leaf is,',hei

def delete(self,key\_value):

def merge(n,i):

if n.clist[i].isleaf():

n.clist[i].vlist=n.clist[i].vlist+n.clist[i+1].vlist

n.clist[i].bro=n.clist[i+1].bro

else:

n.clist[i].ilist=n.clist[i].ilist+[n.ilist[i]]+n.clist[i+1].ilist

n.clist[i].clist=n.clist[i].clist+n.clist[i+1].clist

n.clist.remove(n.clist[i+1])

n.ilist.remove(n.ilist[i])

if n.ilist==[]:

n.clist[0].par=None

self.\_\_root=n.clist[0]

del n

return self.\_\_root

else:

return n

def tran\_l2r(n,i):

if not n.clist[i].isleaf():

n.clist[i+1].clist.insert(0,n.clist[i].clist[-1])

n.clist[i].clist[-1].par=n.clist[i+1]

n.clist[i+1].ilist.insert(0,n.ilist[i])

n.ilist[i]=n.clist[i].ilist[-1]

n.clist[i].clist.pop()

n.clist[i].ilist.pop()

else:

n.clist[i+1].vlist.insert(0,n.clist[i].vlist[-1])

n.clist[i].vlist.pop()

n.ilist[i]=n.clist[i+1].vlist[0].key

def tran\_r2l(n,i):

if not n.clist[i].isleaf():

n.clist[i].clist.append(n.clist[i+1].clist[0])

n.clist[i+1].clist[0].par=n.clist[i]

n.clist[i].ilist.append(n.ilist[i])

n.ilist[i]=n.clist[i+1].ilist[0]

n.clist[i+1].clist.remove(n.clist[i+1].clist[0])

n.clist[i+1].ilist.remove(n.clist[i+1].ilist[0])

else:

n.clist[i].vlist.append(n.clist[i+1].vlist[0])

n.clist[i+1].vlist.remove(n.clist[i+1].vlist[0])

n.ilist[i]=n.clist[i+1].vlist[0].key

def del\_node(n,kv):

if not n.isleaf():

p=bisect\_right(n.ilist,kv)

if p==len(n.ilist):

if not n.clist[p].isempty():

return del\_node(n.clist[p],kv)

elif not n.clist[p-1].isempty():

tran\_l2r(n,p-1)

return del\_node(n.clist[p],kv)

else:

return del\_node(merge(n,p),kv)

else:

if not n.clist[p].isempty():

return del\_node(n.clist[p],kv)

elif not n.clist[p+1].isempty():

tran\_r2l(n,p)

return del\_node(n.clist[p],kv)

else:

return del\_node(merge(n,p),kv)

else:

p=bisect\_left(n.vlist,kv)

try:

pp=n.vlist[p]

except IndexError:

return -1

else:

if pp!=kv:

return -1

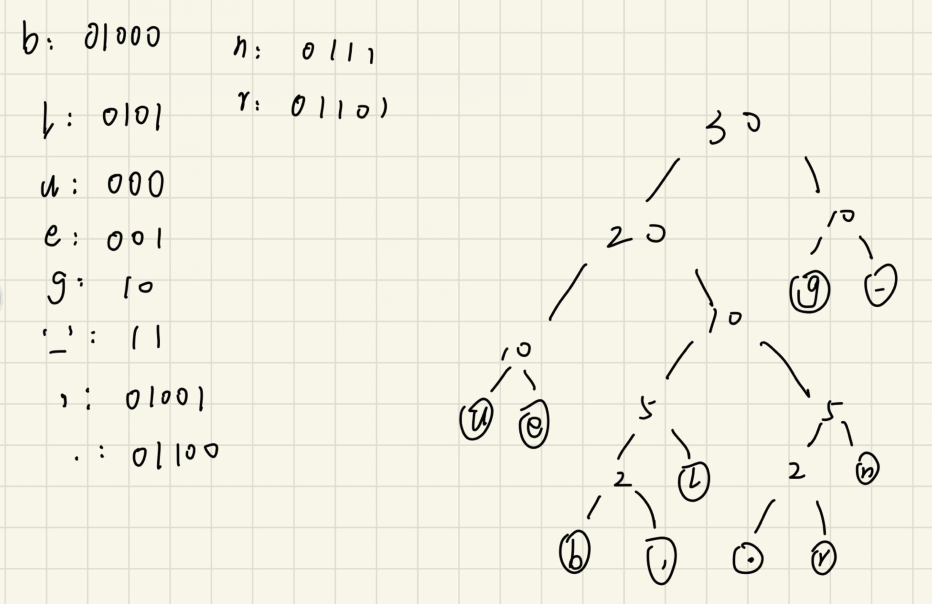
else:

n.vlist.remove(kv)

return 0

del\_node(self.\_\_root,key\_value)

（3）答：



# 树节点类构建

class TreeNode(object):

  def \_\_init\_\_(self, data):

    self.val = data[0]

    self.priority = data[1]

    self.leftChild = None

    self.rightChild = None

    self.code = ""

# 创建树节点队列函数

def creatnodeQ(codes):

  q = []

  for code in codes:

    q.append(TreeNode(code))

  return q

# 为队列添加节点元素，并保证优先度从大到小排列

def addQ(queue, nodeNew):

  if len(queue) == 0:

    return [nodeNew]

  for i in range(len(queue)):

    if queue[i].priority >= nodeNew.priority:

      return queue[:i] + [nodeNew] + queue[i:]

  return queue + [nodeNew]

# 节点队列类定义

class nodeQeuen(object):

  def \_\_init\_\_(self, code):

    self.que = creatnodeQ(code)

    self.size = len(self.que)

  def addNode(self,node):

    self.que = addQ(self.que, node)

    self.size += 1

  def popNode(self):

    self.size -= 1

    return self.que.pop(0)

# 各个字符在字符串中出现的次数，即计算优先度

def freChar(string):

  d ={}

  for c in string:

    if not c in d:

      d[c] = 1

    else:

      d[c] += 1

  return sorted(d.items(),key=lambda x:x[1])

# 创建哈夫曼树

def creatHuffmanTree(nodeQ):

  while nodeQ.size != 1:

    node1 = nodeQ.popNode()

    node2 = nodeQ.popNode()

    r = TreeNode([None, node1.priority+node2.priority])

    r.leftChild = node1

    r.rightChild = node2

    nodeQ.addNode(r)

  return nodeQ.popNode()

codeDic1 = {}

codeDic2 = {}

# 由哈夫曼树得到哈夫曼编码表

def HuffmanCodeDic(head, x):

  global codeDic, codeList

  if head:

    HuffmanCodeDic(head.leftChild, x+'0')

    head.code += x

    if head.val:

      codeDic2[head.code] = head.val

      codeDic1[head.val] = head.code

    HuffmanCodeDic(head.rightChild, x+'1')

# 字符串编码

def TransEncode(string):

  global codeDic1

  transcode = ""

  for c in string:

    transcode += codeDic1[c]

  return transcode

# 字符串解码

def TransDecode(StringCode):

  global codeDic2

  code = ""

  ans = ""

  for ch in StringCode:

    code += ch

    if code in codeDic2:

      ans += codeDic2[code]

      code = ""

  return ans

### 5.10 面试真题

（1）答：前缀编码是指对字符集进行编码时，要求字符集中任一字符的编码都不是其它字符的编码的前缀，例如：设有abcd需要编码表示（其中，a=0、b=10、c=110、d=11,则110的前缀表示的可以是c或者是d跟a，出现这种情况是因为d的前缀11与c的前缀110有重合部分，这个是关键。）

（2）答：在压缩文件时，从源文件每读取一个字节(字符)，得到它新编码的二进制形式的字符串(很可能不够8位)。当凑够8位后，才作为一个字节写入到目标文件中。那么，很有可能，在向目标文件写入的最后一个字节的二进制形式的字符串不够8位!如果直接写入，那么就是默认高位用0填充。这里对最后一个字节的处理，也会影响到后面解压时对最后个字节的处理!进行左移(<<)操作。不足8位，我就将低位尽量往高位移，低位用0填充。

（3）答：红黑树在插入和删除上优于AVL树，AVL树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于AVL树为了维持平衡的开销要小得多

（4）答：B树的优势除了树高小，还有对访问局部性原理的利用。所谓局部性原理，是指当一个数据被使用时，其附近的数据有较大概率在短时间内被使用。B树将键相近的数据存储在同一个节点，当访问其中某个数据时，数据库会将该整个节点读到缓存中；当它临近的数据紧接着被访问时，可以直接在缓存中读取，无需进行磁盘IO；换句话说，B树的缓存命中率更高。

第6章 堆结构

### **6.9 编程练习**

（1）代码如下：

from Queue import PriorityQueue

import time

import random

import threading

class CompareAble:

def \_\_init\_\_(self,priority,jobname):

self.priority = priority

self.jobname = jobname

def \_\_cmp\_\_(self, other):

if self.priority < other.priority:

return -1

elif self.priority == other.priority:

return 0

else:

return 1

tasks = [(i, "admin %s"%i) for i in range(10,100,5)]

# 生产者

def produce(pq,lock):

while True:

lock.acquire()

task = tasks[random.randint(0,len(tasks)-1)]

print(' %s %s publish task'%(task[0],task[1]))

pq.put(CompareAble(task[0],task[1]))

time.sleep(1)

lock.release()

# 消费者

def consumer(pq,lock):

while True:

lock.acquire()

task = pq.get\_nowait()

if task:

print(task.priority, task.jobname)

else:

time.sleep(1)

lock.release()

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

# 创建队列

task\_queue = PriorityQueue()

# 模拟生产 消费

task\_lock = threading.Lock()

for i in range(3):

t = threading.Thread(target=produce,args=(task\_queue,task\_lock))

t.setDaemon(False)

t.start()

for i in range(2):

t = threading.Thread(target=consumer,args=(task\_queue,task\_lock))

t.setDaemon(False)

t.start()

（2）代码如下：

class Binomial\_Heap():

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

self.head = None

self.min = None

def merge(self,H2):

if not self.min:

self.min = H2.min

elif not H2.min:

return

else:

if self.min.data > H2.min.data:

self.min = H2.min

h1 = self.head

h2 = H2.head

unique\_order\_heap = Binomial\_Heap()

c = unique\_order\_heap.head

while h1 and h2:

if h1.order == h2.order:

if h1.data <= h2.data:

h1.order+=1

h2.pre = None

d = h2

h2 = h2.sibling

d.sibling = h1.Lchild

d.parent = h1

h1.Lchild = d

if unique\_order\_heap.head == None:

unique\_order\_heap.head = h1

c = h1

h1.pre = None

h1 = h1.sibling

else:

h1.pre = c

c.sibling = h1

h1 = h1.sibling

c = c.sibling

else:

h2.order += 1

h1.pre = None

d = h1

h1 = h1.sibling

d.sibling = h2.Lchild

d.parent = h2

h2.Lchild = d

if unique\_order\_heap.head == None:

unique\_order\_heap.head = h2

c = h2

h2.pre = None

h2 = h2.sibling

else:

h2.pre = c

c.sibling = h2

h2 = h2.sibling

c = c.sibling

elif h1.order < h2.order:

if unique\_order\_heap.head == None:

unique\_order\_heap.head = h1

c = unique\_order\_heap.head

h1.pre = None

h1 = h1.sibling

else:

if h1.order == c.order:

if c.data <= h1.data:

c.order += 1

d = h1

h1.pre = None

h1 = h1.sibling

d.sibling = c.Lchild

c.Lchild = d

d.parent = c

else:

if c.pre == None: # single B

unique\_order\_heap.head = h1

h1.pre = None

else:

c.pre.sibling = h1

h1.pre = c.pre

h1.order += 1

c.sibling = h1.Lchild

h1.Lchild = c

c.parent = h1

c = h1

h1 = h1.sibling

else:

c.sibling = h1

h1.pre = c

h1 = h1.sibling

c = c.sibling

else:

if unique\_order\_heap.head == None:

unique\_order\_heap.head = h2

c = unique\_order\_heap.head

h2.pre = None

h2 = h2.sibling

else:

if h2.order == c.order:

if c.data <= h2.data:

d = h2

c.order += 1

h2 = h2.sibling

d.sibling = c.Lchild

c.Lchild = d

d.parent = c

else:

if c.pre == None: # single B

unique\_order\_heap.head = h2

h2.pre = None

else:

c.pre.sibling = h2

h2.pre = c.pre

h2.order += 1

c.sibling = h2.Lchild

h2.Lchild = c

c.parent = h2

c = h2

h2 = h2.sibling

else:

c.sibling = h2

h2.pre = c

h2 = h2.sibling

c = c.sibling

if c:

c.sibling = None

while h1:

if c:

if h1.order == c.order:

if c.data <= h1.data:

c.order += 1

d = h1

h1.pre = None

h1 = h1.sibling

d.sibling = c.Lchild

c.Lchild = d

d.parent = c

else:

if c.pre == None: # single B

unique\_order\_heap.head = h1

h1.pre = None

else:

c.pre.sibling = h1

h1.pre = c.pre

h1.order += 1

c.sibling = h1.Lchild

h1.Lchild = c

c.parent = h1

c = h1

h1 = h1.sibling

else:

c.sibling = h1

h1.pre = c

break

else:

unique\_order\_heap.head = h1

h1.pre = None

break

while h2:

if c:

if h2.order == c.order:

if c.data <= h2.data:

d = h2

c.order += 1

h2 = h2.sibling

d.sibling = c.Lchild

c.Lchild = d

d.parent = c

else:

if c.pre == None: # single B

unique\_order\_heap.head = h2

h2.pre = None

else:

c.pre.sibling = h2

h2.pre = c.pre

h2.order += 1

c.sibling = h2.Lchild

h2.Lchild = c

c.parent = h2

c = h2

h2 = h2.sibling

else:

c.sibling = h2

h2.pre = c

break

else:

unique\_order\_heap.head = h2

h2.pre = None

break

self.head = unique\_order\_heap.head

def find\_Min(self):

return self.min

def insert(self,item):

node = Binomial\_Trees(item)

H2 = Binomial\_Heap()

H2.head = node

H2.min = node

self.merge(H2)

def extractMin(self):

m = self.min

if m.pre:

m.pre.sibling = m.sibling

else:

self.head = m.sibling

if m.sibling:

m.sibling.pre = m.pre

new\_heap = Binomial\_Heap()

to\_transform = m.Lchild

minimum = to\_transform

while to\_transform:

if minimum.data > to\_transform.data:

minimum = to\_transform

to\_transform.parent = None

if not new\_heap.head:

new\_heap.head = to\_transform

to\_transform = to\_transform.sibling

new\_heap.head.sibling = None

else:

d = to\_transform

to\_transform = to\_transform.sibling

d.sibling = new\_heap.head

new\_heap.head.pre = d

new\_heap.head = d

new\_heap.min = minimum

cur = self.head

minimum = self.head

while cur:

if minimum.data < cur.data:

minimum = cur

cur = cur.sibling

self.min = minimum

self.merge(new\_heap)

return m.data

class Binomial\_Trees():

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

self.order = 0

self.parent = None

self.sibling = None

self.Lchild = None

self.data = item

self.pre = None

### 6.10 面试真题

（1）答：二项堆是符合最小堆性质的二项树链接成的森林斐波那契堆类似二项堆，不同的是链表中的每个树不一定是二项树，思路是把复杂操作延后到min操作上，add和merge操作都非常简单。

（2）略。

第7章 散列结构

### 7.7 编程练习

（1）答：可采用直接定址法，把每个字符的ascii码直接加一起或者乘以31再相加作为key的值，在对其进行相应计算。

（2）代码如下：

**class Node:**

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

**self.data = data**

**self.next = None**

**class Hashtable:**

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

**self.data = [Node()] \* size**

**self.size = size**

**def hash\_function(self, key, size):**

**return key % size**

**def put(self, key):**

**hash\_value = self.hash\_function(key, self.size)**

**if self.data[hash\_value].data == None:**

**self.data[hash\_value].data = key**

**else:**

**temp = Node(key)**

**p = self.data[hash\_value]**

**while p.next != None:**

**p = p.next**

**p.next = temp**

**def get(self, key):**

**hash\_value = self.hash\_function(key, self.size)**

**if self.data[hash\_value].data == key:**

**return True**

**else:**

**p = self.data[hash\_value]**

**while p != None and p.data != key:**

**p = p.next**

**if p != None and p.data == key:**

**return True**

**return False**

**def delete(self, key):**

**if not self.get(key):**

**return 'Delete Error'**

**hash\_value = self.hash\_function(key, self.size)**

**if self.data[hash\_value].data == key:**

**self.data[hash\_value].data = None**

**else:**

**p = self.data[hash\_value]**

**pre = None**

**while p != None and p.data != key:**

**pre = p**

**p = p.next**

**if p == None:**

**return 'Delete Error'**

**else:**

**pre.next = p.next**

**n = int(input())**

**h = Hashtable(n)**

**lst = list(set(input().split()))**

**slst = [int(i) for i in lst]**

**for i in slst:**

**h.put(i)**

**n = int(input())**

**for i in range(n):**

**tmp = int(input())**

**h.put(tmp)**

**n = int(input())**

**for i in range(n):**

**tmp = int(input())**

**tmp1 = h.delete(tmp)**

**if tmp1 == 'Delete Error':**

**print('Delete Error')**

**n = int(input())**

**for i in range(n):**

**tmp = int(input())**

**print(h.get(tmp))**

### 7.8 面试真题

（1）答：开放定址法、链地址法、再哈希法、建立公共溢出区。

（2）略。

（3）略。

（4）答：删除慢，如果不知道关键则存取很慢，对存储空间使用不充分

第8章 图结构

### 8.7 编程练习

（1）答：邻接表。代码如下：

class Vertex:

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

self.id = key

self.connectedTo = {}

def addNeighbor(self,nbr,weight = 0):

self.connectedTo[nbr] = weight

def \_\_str\_\_(self):

return str(self.id) + 'connectedTo' + str([x.id for x in self.connectedTo])

def getConnections(self):

return self.connectedTo.keys()

def getId(self):

return self.id

def getweight(self,nbr):

return self.connectedTo[nbr]

class Graph:

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

self.vertList = {}

self.numVertices = 0

def addVertex(self,key):

self.numVertices = self.numVertices + 1

newVertex = Vertex(key)

self.vertList[key] = newVertex

return newVertex

def getVertex(self,n):

if n in self.vertList:

return self.vertList[n]

else:

return None

def \_\_contains\_\_(self, n):

return n in self.vertList

def addEdge(self,f,t,const = 0):

if f not in self.vertList:

nv = self.addVertex(f)

if t not in self.vertList:

nv = self.addVertex(t)

self.vertList[f].addNeighbor(self.vertList[t],const)

def getVertices(self):

return self.vertList.keys()

def \_\_iter\_\_(self):

return iter(self.vertList.values())

（2）答：

DFS：ABCDFEG（不唯一）

def DFS(graph,s):#图 s指的是开始结点

#需要一个队列

stack=[]

stack.append(s)

seen=set()#看是否访问过

seen.add(s)

while (len(stack)>0):

#拿出邻接点

vertex=stack.pop()#这里pop参数没有0了，最后一个元素

nodes=graph[vertex]

for w in nodes:

if w not in seen:#如何判断是否访问过，使用一个数组

stack.append(w)

seen.add(w)

print(vertex)

BFS：ABCFGDE（不唯一）

def BFS(graph,s):#graph图 s指的是开始结点

#需要一个队列

queue=[]

queue.append(s)

seen=set()#看是否访问过该结点

seen.add(s)

while (len(queue)>0):

vertex=queue.pop(0)#保存第一结点，并弹出，方便把他下面的子节点接入

nodes=graph[vertex]#子节点的数组

for w in nodes:

if w not in seen:#判断是否访问过，使用一个数组

queue.append(w)

seen.add(w)

print(vertex)

### 8.8 面试真题

（1）答：对一个有向无环图G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边∈E(G)，则u在线性序列中出现在v之前。简单的说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序

应用于深度学习，有拓扑结构的任务/数据调度各种包管理器，构建工具，配置管理工具，数据流工具等。

（2）答：例如地图导航中设计的最短路径的算法。

（3）代码如下：

def prim(vertexs, edges,start='D'):

adjacent\_dict = defaultdict(list)

for weight,v1, v2 in edges:

adjacent\_dict[v1].append((weight, v1, v2))

adjacent\_dict[v2].append((weight, v2, v1))

minu\_tree = []

visited = [start]

adjacent\_vertexs\_edges = adjacent\_dict[start]

heapify(adjacent\_vertexs\_edges)

while adjacent\_vertexs\_edges:

weight, v1, v2 = heappop(adjacent\_vertexs\_edges)

if v2 not in visited:

visited.append(v2)

minu\_tree.append((weight, v1, v2))

for next\_edge in adjacent\_dict[v2]:

if next\_edge[2] not in visited:

heappush(adjacent\_vertexs\_edges, next\_edge)

return minu\_tree

第9章 递归算法

### 9.4 编程练习

（1）代码如下：

def func(n):

if n == 1 or n == 2:

return 1

else:

return func(n-1) + func(n-2)

res = func(x)

print(res)

（2）答：避免重复计算的方法就是，每计算得到一个值，便用一个数组保存这个值，下次计算该值的时候直接引用就可以了，代码略。

### 9.5 面试真题

（1）答：相同点包括：① 递归和循环的本质都是代码复用。② 递归和循环在理论上具有相同的计算能力（在可计算性理论中，这两种计算模型所计算的函数类是相同的）。③递归是一种特殊的循环。

不同点包括：① 程序独立性。递归由程序和系统共同完成。递归需要系统维护一个系统工作栈。循环由程序单独完成。但是，循环需要程序设定好循环条件。② 程序规模。递归的规模很小，受到系统工作栈大小的限制。循环的规模很大，几乎不会受到限制。在VS2012中计算1+2+3+······+n使用递归n只能达到4710.使用循环n可以达到20 0000 0000。③ 递复用单位。递归的复用单位是函数，循环的复用单位是语句（for循环语句或while循环语句）。④ 解题方向。递归往往是自顶向下（1 <—— n），将问题的规模逐步缩小，直到缩小至递归结束条件成立（n == 1）。循环既可以是自顶向下（1 <—— n），也可以是自底向上（1 ——> n），但是一般还是自底向上（1 ——> n）的比较多。⑤ 优缺点。递归的优点：代码清晰简洁，易于理解，可读性强。递归的缺点：运行效率低（函数调用需要参数入 栈和出栈）。

（2）答：运用场景：

① 数据的定义是按递归定义的。（Fibonacci[函数](http://baike.baidu.com/view/15061.htm)）

② 问题解法按[递归算法](http://baike.baidu.com/view/1733593.htm)实现。（回溯）

③ 数据的结构形式是按递归定义的。（[树的遍历](http://baike.baidu.com/view/6313985.htm)，图的搜索）

第10章 分类算法

### 10.7 编程练习

见范例文件。

### 10.8 面试真题

（1）答：TP = 45 FN = 5

FP = 10 TN = 40

精度：(TP+TN) / (P+N) = 0.85

准确率：TP / (TP+FP) = 0.818

召回率: TP / (TP+FN) = 0.9

（2）答：误分类,因为惩罚太小了，分类器会最大化大多数点之间的间隔，少数点会误分类。

（3）答：出现欠拟合的原因是模型尚未学习到数据的真实结构。因此，模拟在训练集和验证集上的性能都很差。最好的选择是创造更多特征带入模型训练。其次，增加模型复杂度。如果模型太简单，不能够应对复杂的任务。可以使用更复杂的模型，减小正则化系数。具体来说可以使用核函数，集成学习方法，深度学习等。

第11章 回归算法

### 11.6 编程练习

见范例文件。

### 11.7 面试真题

（1）答：这两个概念都是监督机器学习技术的一个重要方面。通过分类，输出被分类为用于进行预测的不同类别。而回归模型通常用于找出预测和变量之间的关系。分类和回归之间的关键区别在于，在前者中，输出变量是离散的，而在后者中是连续的。

（2）答：过拟合就是在机器学习中，模型训练效果好，但对新的测试数据预测很差。

产生过拟合原因：数据有噪声；训练数据不足，有限的训练数据训练模型过度导致模型非常复杂。

避免过拟合方法：early stopping，在发生拟合之前提前结束训练。数据集扩增，最大的满足全样本。正则化，引入范数概念，增强模型泛化能力。droput，每次训练时丢弃一些节点，增强泛化能力；batch normalization；减小模型复杂度。

第12章 聚类算法

### 12.7 编程练习

见范例文件。

### 12.8 面试真题

（1）答：聚类算法和分类算法的区别如下表hi

|  |  |  |
| --- | --- | --- |
|  | **聚类** | **分类** |
| 核心 | 将数据分成多个组 | 从已经分组的数据中学习 |
| 学习类型 | 无监督，不需要标签进行训练 | 有监督，需要标签进行训练 |
| 典型算法 | K-means,DBSCAN等 | 决策树、支持向量机、贝叶斯等 |
| 算法输出 | 聚类结果是不确定的  不一定能够反映数据的真实分类 | 分类结果是确定的 |

（2）答：执行过程如下。

① 数据预处理，如归一化、离散点处理即可。

② 从数据集D中随机选择k个样本作为初始簇的质心。

③ 计算每个数据点与K个质心之间的距离，将其分配到最近的质心所代表的簇中，形成K个簇。

④ 重新计算K个簇的质心。

⑤ 重复③和④，直至K个簇的质心都不再发生变化。

⑥ 输出簇的划分。

（3）答：可以采用以下3种方法。

① 数据归一化和离群点处理。K均值聚类本质上是一种基于欧氏距离度量的数据划分方法，均值和方差大的维度将对数据的聚类结果产生决定性的影响，所以未做归一化处理和统一单位的数据是无法直接参与运算和比较的。同时，离群点或者少量噪声数据就会对均值产生较大的影响，导致中心偏移，因此使用K均值聚类算法之前通常需要对数据做预处理。

② 合理选择K值。K值的选择是K均值聚类最大的问题之一，这也是K均值聚类算法的主要缺点。K值的选择一般基于经验或者多次实验结果。

③ 采用核函数。采用核函数是另一种可以尝试的改进方向。传统的欧氏距离度量方式，使得K均值算法本质上假设了各个数据簇的数据具有一样的先验嫌疑，并呈现球形或者高维球形分布，这种分布在实际生活中并不常见。面对非凸的数据分布数据时，可能需要引入核函数来优化，这时算法又是通过一个非线性映射，将输入空间中的数据点映射到高位的特征空间中，并在新的特征空间中进行聚类。非线性映射增加了数据点线性可分的概率，从而在经典的聚类算法失效的情况下，通过引入核函数可以达到更为准确的聚类结果。

第13章 降维算法

### 13.6 编程练习

见范例文件。

### 13.7 面试真题

（1）答：协方差矩阵度量的是维度与维度之间的关系，能同时表现不同维度间的相关性以及各个维度上的方差。协方差矩阵的主对角线上的元素是各个维度上的方差，其他元素是两两维度间的协方差（即相关性）。

降维的目的就是“降噪”和“去冗余”。“降噪”的目的就是让保留下的不同维度间的相关性尽可能小，也就是说让协方差矩阵中非对角线元素都基本为零。达到这个目的的方式——矩阵对角化。“去冗余”的目的就是对角化后的协方差矩阵，对角线上较小的新方差对应的就是那些该去掉的维度。我们只取那些含有较大能量（特征值）的维度，其余的就舍掉即可。

（2）答：相同点包括：① 两者均可用于数据降维；② 两者在降维时均使用了矩阵特征分解的思想；③ 两者都假设被降维的数据服从高斯分布。不同点包括：① LDA是有监督的降维方法，PCA是无监督降维方法；② 当共有K个类别时，LDA最多降到K-1维，而PCA没有这个限制；③ LDA除了用于降维，还可以用于分类；④ PCA投影的坐标系都是正交的，而LDA根据类别的标注，关注分类能力，因此不保证投影到的坐标系是正交的（一般都不正交）。

第14章 集成学习算法

### 14.7 编程练习

见范例文件。

### 14.8 面试真题

（1）答：XGBoost使用了一阶和二阶偏导，二阶导数有利于梯度下降的更快更准。使用泰勒展开取得函数做自变量的二阶导数形式，可以在不选定损失函数具体形式的情况下，仅仅依靠输入数据的值就可以进行叶子分裂优化计算，本质上也就把损失函数的选取和模型算法优化/参数选择分开了。这种去耦合增加了XGBoost的适用性。使得它按需选取损失函数，可以用于分类，也可以用于回归。

精准性：相对于GBDT的一阶泰勒展开，XGBoost采用了二阶泰勒展开，可以更为精准的逼近真实的损失函数。

可扩展性：损失函数支持自定义，只需要新的损失函数二阶可导。

（2）答：XGBoost在设计时，为了防止过拟合做了很多优化。具体如下。

① 目标函数添加正则项，。

② 列采样：训练的时候只用一部分特征。

③ 子采样：每轮可以不选用全部样本而选取部分样本进行训练，增加了树的多样性。

④ shrinkage：调小学习率增加树的数量，为后面的树留下提升空间。

⑤ early stopping：如果经过固定的迭代次数后，并没有在验证集上改善性能，停止训练过程。

第15章 基于价值的强化学习(Value-Based RL)算法

### 15.6 编程练习

见范例文件。

### 15.7 面试真题

（1）答：强化学习由环境、动作和奖励组成，强化学习的目标是使得作出的一系列决策得到的总的奖励的期望最大化。

（2）答：监督学习一般有标签信息，而且是单步决策问题，比如分类问题。监督学习的样本一般是独立同分布的；

无监督学习没有任何标签信息，一般对应的是聚类问题；

强化学习介于监督和无监督学习之间，每一步决策之后会有一个标量的反馈信号，即回报。通过最大化回报以获得一个最优策略。因此强化学习一般是多步决策，并且样本之间有强的相关性。

强化学习的损失函数（loss function）是什么？和深度学习的损失函数有何关系？

（3）答：强化学习适合于模型输出的动作必须要能够改变环境的状态，并且模型能够获得环境的反馈，同时状态应该是可重复到达的。

第16章 基于策略的强化学习(Policy-Based RL)算法

### 16.5 编程练习

见范例文件。

### 16.6 面试真题

（1）答：在线策略（on-policy）用于学习和用于采样的是同一个策略，离线策略（off-policy）中行动策略用来采样，目标策略是用来学习的。在线策略不学习最优动作而是学习一个接近最优动作同时又能继续探索的动作，离线策略直接学习最优动作。

（2）答：value-based 的典型算法是DQN，policy-based是policy gradient，结合这两种具体算法可能会更好的理解。

处理的action space不同：value-based适合处理的action space低维离散的，policy-based适合处理连续的action space。

针对action的价值输出不同：value-based计算出每个action的价值，policy-based一般情况下只给出较价值较高的actions。

更新频率不同：value-based每个action执行都可以更新，policy-based 每个episode完成之后才能更新一次。

第17章 神经网络模型算法

### 17.6 编程练习

见范例文件。

### 17.7 面试真题

（1）答：计算出输出与标签间的损失函数值，然后计算其相对于每个神经元的梯度，根据梯度方向更新权值。

① 将训练集数据输入到ANN的输入层，经过隐藏层，最后达到输出层并输出结果，这是ANN的前向传播过程；

② 由于ANN的输出结果与实际结果有误差，则计算估计值与实际值之间的误差，并将该误差从输出层向隐藏层反向传播，直至传播到输入层；

③ 在反向传播的过程中，根据误差调整各种参数的值；不断迭代上述过程，直至收敛。

（2）答：① sigmoid：将输出实值压缩到0-1之间。 缺点：（输入非常大或非常小的时候）容易梯度消失；sigmoid函数是非0均值的，下一层的神经元将从上一层神经元得到的非0 均值的信号作为输入，再结合w计算梯度，始终都是正的。（可根据batch调节）

② Tanh：是0均值的。

③ Relu（修正线性单元）：好处：收敛快，求梯度简单。具有稀疏特性。

第18章 循环神经网络算法

### 18.8 编程练习

见范例文件。

### 18.9 面试真题

（1）答：RNN在计算梯度的时候，会有激活函数导数的累乘，如果取tanh或sigmoid函数作为激活函数的话，那么必然是一堆小数在做乘法，结果就是越乘越小。随着时间序列的不断深入，小数的累乘就会导致梯度越来越小直到接近于0，这就是“梯度消失“现象。

解决方法：选取更好的激活函数，如Relu激活函数。ReLU函数的左侧导数为0，右侧导数恒为1，这就避免了“梯度消失“的发生。但恒为1的导数容易导致“梯度爆炸“，但设定合适的阈值可以解决这个问题。

加入BN层，其优点包括可加速收敛、控制过拟合，可以少用或不用Dropout和正则、降低网络对初始化权重不敏感，且能允许使用较大的学习率等。

改变传播结构，LSTM结构可以有效解决这个问题。

（2）答：LSTM是RNN最知名和成功的扩展，由于RNN梯度消失和梯度爆炸的原因，很难学习到序列中长距离的依赖关系。LSTM通过门控机制可以对有价值的信息长时间记忆，从而减小循环神经和网络的学习难度。

LSTM记忆时间长的原因是因为LSTM增加了门控和细胞状态，细胞状态贯穿始终，类似传送带，信息在上面流动，通过遗忘门去除不必要的信息，通过更新门来增加信息。而RNN前面的信息则会随着时间步的推移，慢慢减少。

第19章 卷积神经网络算法

### 19.7 编程练习

见范例文件。

### 19.8 面试真题

（1）答：答：GradientDescentOptimizer, AdagradOptimizer, Optimize 和优化最小代价函数。

（2）答： relu，sigmoid,tanh