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

001、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。
6. 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。

002、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）；

003、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；

004、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())
```

005、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

006、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
```

007、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())
```

008、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

009、台阶问题/斐波那契？（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 fib(n - 1) + fib(n - 2)
```

第二种方法

```
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
```

043、青蛙跳台阶问题？

一只青蛙要跳上 n 层高的台阶，一次能跳一级，也可以跳两级，请问这只青蛙有多少种跳上这个 n 层台阶的方法？
方法 1：递归
设青蛙跳上 n 级台阶有 f(n)种方法，把这 n 种方法分为两大类，第一种最后一次跳了一级台阶，这类共有 f(n-1)种，第二种最后一次跳了两级台阶，这种方法共有 f(n-2)种，则得出递推公式 f(n)=f(n-1) + f(n-2),显然 f(1)=1,f(2)=2，这种方法虽然代码简单，但效率低，会超出时间上限

```
class Solution:
    def climbStairs(self,n):
        if n ==1:
            return 1
        elif n==2:
            return 2
        else:
            return self.climbStairs(n-1) + self.climbStairs(n-2)
```

方法 2：用循环来代替递归

```
class Solution:
    def climbStairs(self,n):
        if n==1 or n==2:
            return n
        a,b,c = 1,2,3
        for i in range(3,n+1):
            c = a+b
            a = b
            b = c
        return c
```

011、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)
```

012、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
```

013、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
```

014、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
```

015、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}
```

016、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)
```

017、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'))
```

018、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/

021、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

022、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]

023、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

024、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

025、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)))
```

026、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]
```

027、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)
```

028、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)
```

029、求最大树深？

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

030、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
```

031、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
```

032、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

032、文件操作

1.有一个 jsonline 格式的文件 file.txt 大小约为 10K

```
def get_lines():
    with open('file.txt','rb') as f:
        return f.readlines()

if __name__ == '__main__':
    for e in get_lines():
        process(e) # 处理每一行数据
```

现在要处理一个大小为 10G 的文件，但是内存只有 4G，如果在只修改 get_lines 函数而其他代码保持不变的情况下，应该如何实现？需要考虑的问题都有那些？

```
def get_lines():
    with open('file.txt','rb') as f:
        for i in f:
            yield i
```

个人认为：还是设置下每次返回的行数较好，否则读取次数太多。

```
def get_lines():
    l = []
    with open('file.txt','rb') as f:
      data = f.readlines(60000)
    l.append(data)
    yield l
```

Pandaaaa906 提供的方法

```
from mmap import mmap
def get_lines(fp):
    with open(fp,"r+") as f:
        m = mmap(f.fileno(), 0)
        tmp = 0
        for i, char in enumerate(m):
            if char==b"\n":
                yield m[tmp:i+1].decode()
                tmp = i+1
```

```
if __name__=="__main__":
    for i in get_lines("fp_some_huge_file"):
        print(i)
```

要考虑的问题有：内存只有 4G 无法一次性读入 10G 文件，需要分批读入分批读入数据要记录每次读入数据的位置。分批每次读取数据的大小，太小会在读取操作花费过多时间。

https://stackoverflow.com/questions/30294146/python-fastest-way-to-process-large-file

2.补充缺失的代码

```
def print_directory_contents(sPath):
"""
这个函数接收文件夹的名称作为输入参数
返回该文件夹中文件的路径
以及其包含文件夹中文件的路径
"""
import os
for s_child in os.listdir(s_path):
    s_child_path = os.path.join(s_path, s_child)
    if os.path.isdir(s_child_path):
        print_directory_contents(s_child_path)
    else:
        print(s_child_path)
```

033、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]
```

034、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]

035、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

036、问题描述：判断输入的小括号、中括号、大括号，是否是合规？

一 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）

二 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 F a l s e
        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 F a l s e
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
```

三 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))
```

037、你知道的 python 排序算法？

01、冒泡排序（Bubble Sort）
02、选择排序（Selection Sort）
03、插入排序（Insertion Sort）
04、希尔排序（Shell Sort）
05、归并排序（Merge Sort）
06、快速排序（Quick Sort）
07、堆排序（Heap Sort）
08、计数排序（Counting Sort）
09、桶排序（Bucket Sort）
10、基数排序（Radix Sort）

038、介绍下你知道的缓存相关的算法?

039、介绍下你知道的负载均衡相关的算法?

1、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；
2、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
3、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；
4、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；
5、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；

040、在某系统中一个整数占用两个八位字节，使用 Python 按下面的要求编写完整程序。

接收从标准输入中依次输入的五个数字，将其组合成为一个整数，放入全局变量 n 中，随后在标准输出输出这个整数。（ord(char)获取字符 ASCII 值的函数）

041、斐波那契数列？

数列定义:
f 0 = f 1 = 1 f n = f (n-1) + f (n-2)
根据定义

速度很慢，另外(暴栈注意！⚠ ） O(fibonacci n)

```
def fibonacci(n):
     if n == 0 or n == 1:
          return 1
     return fibonacci(n - 1) + fibonacci(n - 2)
```

线性时间的
状态/循环

```
def fibonacci(n):
   a, b = 1, 1
   for _ in range(n):
        a, b = b, a + b
   return a
```

递归

```
def fibonacci(n):
     def fib(n_, s):
          if n_ == 0:
               return s[0]
          a, b = s
          return fib(n_ - 1, (b, a + b))
     return fib(n, (1, 1))
```

map(zipwith)

```
def fibs():
     yield 1
     fibs_ = fibs()
     yield next(fibs_)
     fibs__ = fibs()
     for fib in map(lambad a, b: a + b, fibs_, fibs__):
          yield fib

def fibonacci(n):
     fibs_ = fibs()
     for _ in range(n):
          next(fibs_)
```

```
    return next(fibs)
```
做缓存
```
def cache(fn):
    cached = {}
    def wrapper(*args):
        if args not in cached:
            cached[args] = fn(*args)
        return cached[args]
    wrapper.__name__ = fn.__name__
    return wrapper
@cache
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```
利用 funtools.lru_cache 做缓存
```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```
Logarithmic
矩阵
```
import numpy as np
def fibonacci(n):
    return (np.matrix([[0, 1], [1, 1]]) ** n)[1, 1]
```
不是矩阵
```
def fibonacci(n):
    def fib(n):
        if n == 0:
            return (1, 1)
        elif n == 1:
```

```
            return (1, 2)
        a, b = fib(n // 2 - 1)
        c = a + b
        if n % 2 == 0:
            return (a * a + b * b, c * c - a * a)
        return (c * c - a * a, b * b + c * c)
    return fib(n)[0]
```

034、平衡点问题

平衡点：比如 int[] numbers = {1,3,5,7,8,25,4,20}; 25 前面的总和为 24，25 后面的总和也是 24，25 这个点就是平衡点；假如一个数组中的元素，其前面的部分等于后面的部分，那么这个点的位序就是平衡点

要求：返回任何一个平衡点

使用 sum 函数累加所有的数。

使用一个变量 fore 来累加序列的前部。直到满足条件 fore<(total-number)/2;

python 代码如下:

```
numbers = [1,3,5,7,8,2,4,20]

#find total
total=sum(numbers)

#find num
fore=0
for number in numbers:
   if fore<(total-number)/2 :
      fore+=number
   else:
      break

#print answer
if fore == (total-number)/2 :
   print number
else :
   print r'not found'
```

算法简单，而且是 O(n)的，12 行代码搞定。参考 http://blog.renren.com/share/235087438/3004327956

035、支配点问题：

支配数：数组中某个元素出现的次数大于数组总数的一半时就成为支配数，其所在位序成为支配点；比如 int[] a = {3,3,1,2,3};3 为支配数，0，1，4 分别为支配点；

要求：返回任何一个支配点

```
li = [3,3,1,2,3]
def main():
    mid = len(li)/2
    for l in li:
        count = 0
        i = 0
        mark = 0
        while True:
            if l == li[i]:
                count += 1
                temp = i
            i += 1
            if count > mid:
                mark = temp
                return (mark,li[mark])
            if i > len(li) - 1:
                break

if __name__ == "__main__":
    print  main()
```

042、如何翻转一个单链表？

```
class Node:
    def __init__(self,data=None,next=None):
        self.data = data
        self.next = next
def rev(link):
    pre = link
    cur = link.next
    pre.next = None
    while cur:
```

```
            temp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
      return pre
if __name__ == '__main__':
      link = Node(1,Node(2,Node(3,Node(4,Node(5,Node(6,Node7,Node(8.Node(9)))))))
      root = rev(link)
      while root:
            print(roo.data)
            root = root.next
```

044、三种快排与四种优化？

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，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。

测试数据：

随机生成 1 百万个数字，进行排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms

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：

随机生成 1 百万个数字，进行排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms

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：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms

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：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms

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：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms

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：

三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms

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：

随机生成 1 百万个数字，进行升序排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	133ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	131ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms
STL 中的 Sort 函数	125ms	27ms	31ms	8ms

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

045、动态规划问题？

可参考：[动态规划(DP)的整理-Python 描述]

请先好好阅读如下内容–什么是动态规划?

摘录于《算法图解》

第 9 章

动态规划

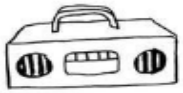
本章内容

- 学习动态规划，这是一种解决棘手问题的方法，它将问题分成小问题，并先着手解决这些小问题。
- 学习如何设计问题的动态规划解决方案。

9.1 背包问题

我们再来看看第8章的背包问题。假设你是个小偷，背着一个可装4磅东西的背包。

你可盗窃的商品有如下3件。



音响
3000美元
4磅

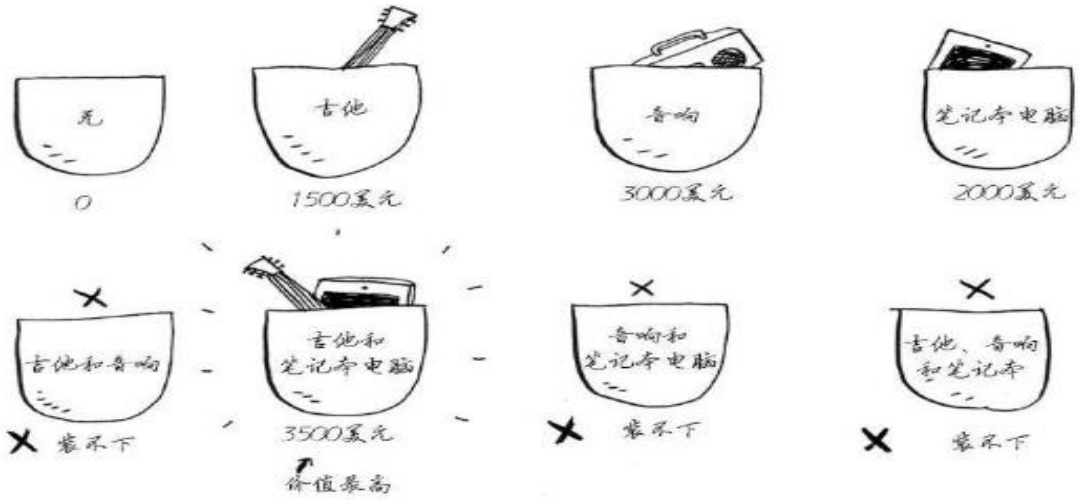
笔记本电脑
2000美元
3磅

吉他
1500美元
1磅

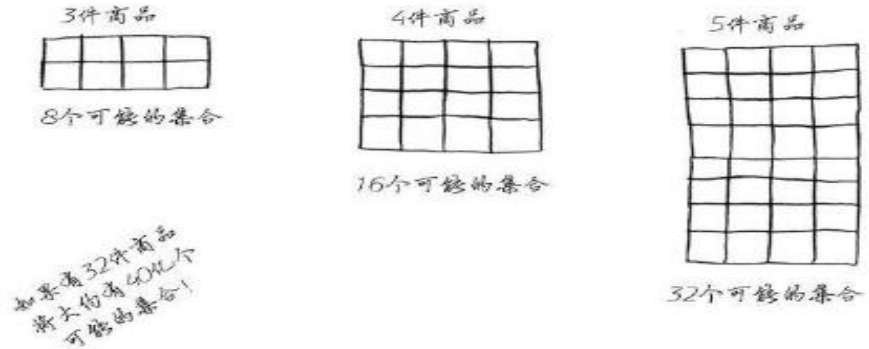
为了让盗窃的商品价值最高，你该选择哪些商品？

9.1.1 简单算法

最简单的算法如下：尝试各种可能的商品组合，并找出价值最高的组合。



这样可行，但速度非常慢。在有3件商品的情况下，你需要计算8个不同的集合；有4件商品时，你需要计算16个集合。每增加一件商品，需要计算的集合数都将翻倍！这种算法的运行时间为$O(2^n)$，真的是慢如蜗牛。



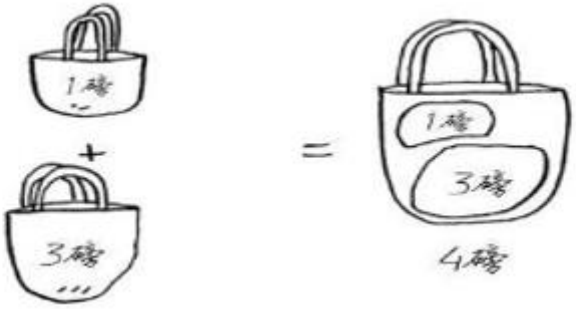
只要商品数量多到一定程度，这种算法就行不通。在第8章，你学习了如何找到近似解，这接近最优解，但可能不是最优解。

那么如何找到最优解呢？

9.1.2　动态规划

答案是使用动态规划！下面来看看动态规划算法的工作原理。动态规划先解决子问题，再逐步解决大问题。

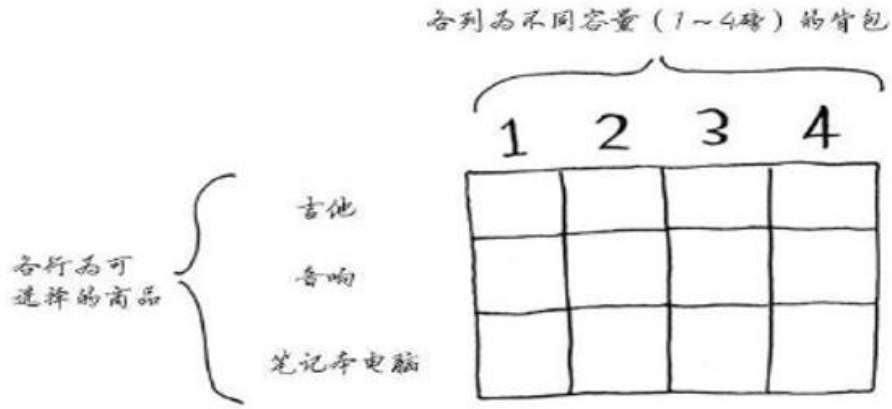
对于背包问题，你先解决小背包（子背包）问题，再逐步解决原来的问题。



动态规划是一个难以理解的概念，如果你没有立即搞懂，也不用担心，我们将研究很多示例。

先来演示这种算法的执行过程。看过执行过程后，你心里将有一大堆问题！我将竭尽所能解答这些问题。

每个动态规划算法都从一个网格开始，背包问题的网格如下。

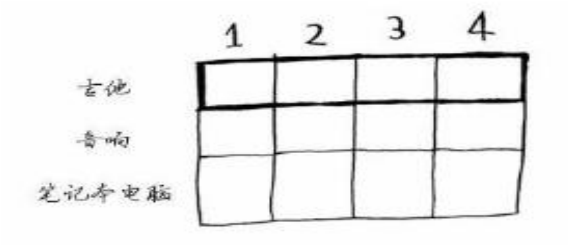


网格的各行为商品，各列为不同容量（1～4磅）的背包。所有这些列你都需要，因为它们将帮助你计算子背包的价值。

网格最初是空的。你将填充其中的每个单元格，网格填满后，就找到了问题的答案！你一定要跟着做。请你创建网格，我们一起来填满它。

1. 吉他行

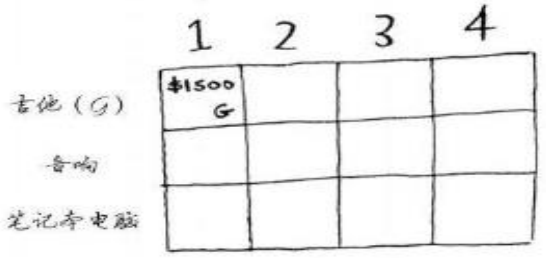
后面将列出计算这个网格中单元格值的公式。我们先来一步一步做。首先来看第一行。



这是吉他行，意味着你将尝试将吉他装入背包。在每个单元格，都需要做一个简单的决定：偷不偷吉他？别忘了，你要找出一个价值最高的商品集合。

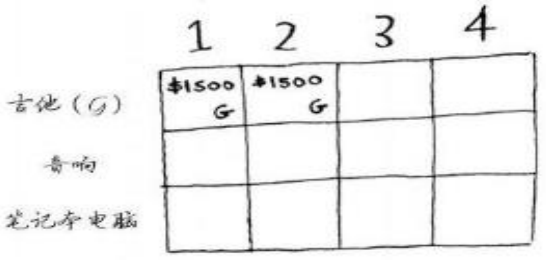
第一个单元格表示背包的容量为1磅。吉他的重量也是1磅，这意味着它能装入背包！因此这个单元格包含吉他，价值为1500美元。

下面来开始填充网格。

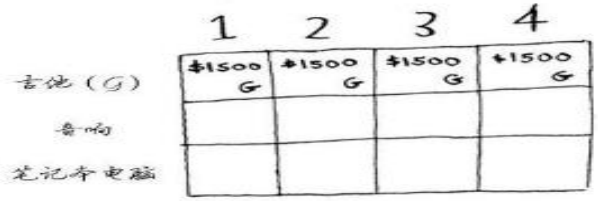


与这个单元格一样，每个单元格都将包含当前可装入背包的所有商品。

来看下一个单元格。这个单元格表示背包的容量为2磅，完全能够装下吉他！

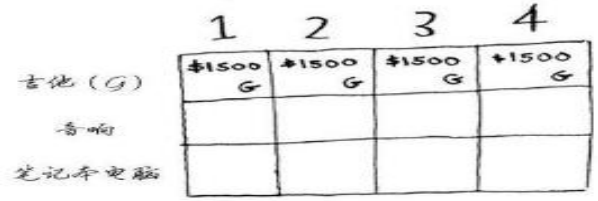


这行的其他单元格也一样。别忘了，这是第一行，只有吉他可供你选择。换言之，你假装现在还没法盗窃其他两件商品。

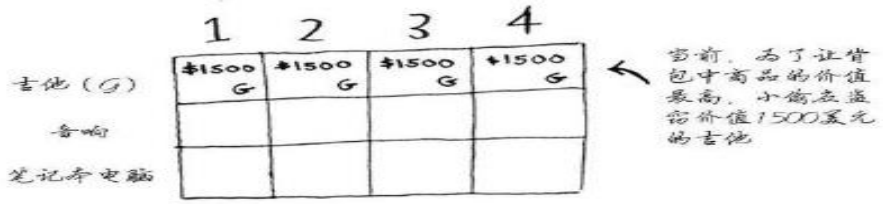


此时你很可能心存疑惑：原来的问题说的是4磅的背包，我们为何要考虑容量为1磅、2磅等的背包呢？前面说过，动态规划从小问题着手，逐步解决大问题。这里解决的子问题将帮助你解决大问题。请接着往下读，稍后你就会明白的。

此时网格应类似于下面这样。



别忘了，你要做的是让背包中商品的价值最大。这行表示的是当前的最大价值。它指出，如果你有一个容量4磅的背包，可在其中装入的商品的最大价值为1500美元。



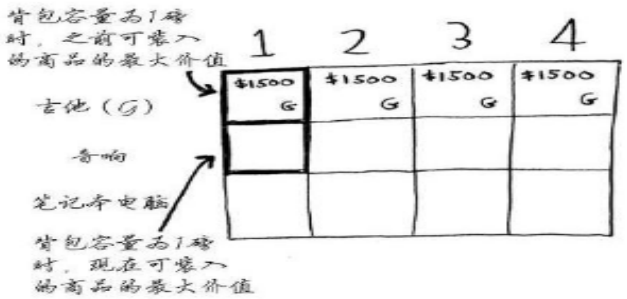
你知道这不是最终的解。随着算法往下执行，你将逐步修改最大价值。

2. 音响行

我们来填充下一行——音响行。你现在出于第二行，可偷的商品有吉他和音响。在每一行，

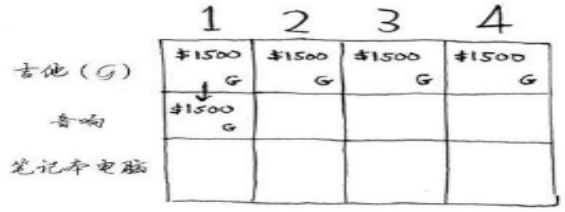
2. 音响行

我们来填充下一行——音响行。你现在出于第二行，可偷的商品有吉他和音响。在每一行，可偷的商品都为当前行的商品以及之前各行的商品。因此，当前你还不能偷笔记本电脑，而只能偷音响和吉他。我们先来看第一个单元格，它表示容量为1磅的背包。在此之前，可装入1磅背包的商品的最大价值为1500美元。

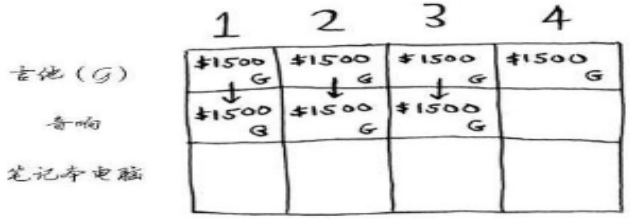


该不该偷音响呢？

背包的容量为1磅，能装下音响吗？音响太重了，装不下！由于容量1磅的背包装不下音响，因此最大价值依然是1500美元。

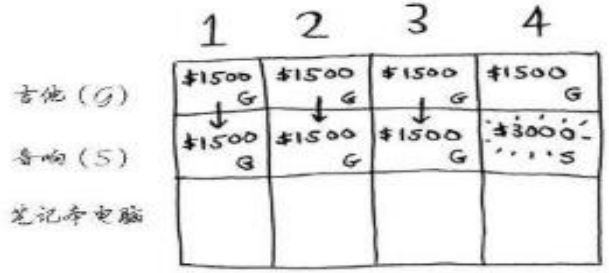


接下来的两个单元格的情况与此相同。在这些单元格中，背包的容量分别为2磅和3磅，而以前的最大价值为1500美元。

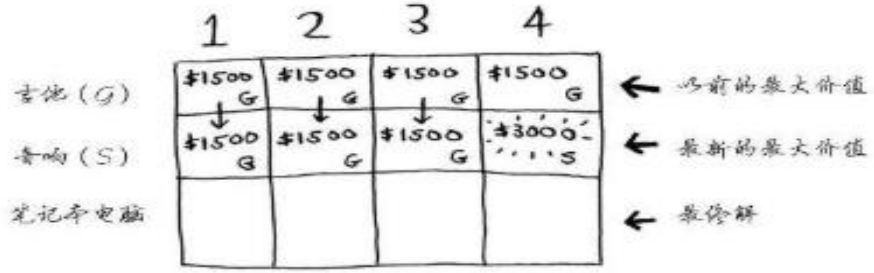


由于这些背包装不下音响，因此最大价值保持不变。

背包容量为4磅呢？终于能够装下音响了！原来的最大价值为1500美元，但如果在背包中装入音响而不是吉他，价值将为3000美元！因此还是偷音响吧。

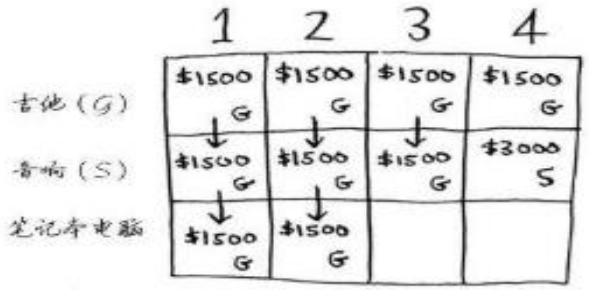


你更新了最大价值！如果背包的容量为4磅，就能装入价值至少3000美元的商品。在这个网格中，你逐步地更新最大价值。

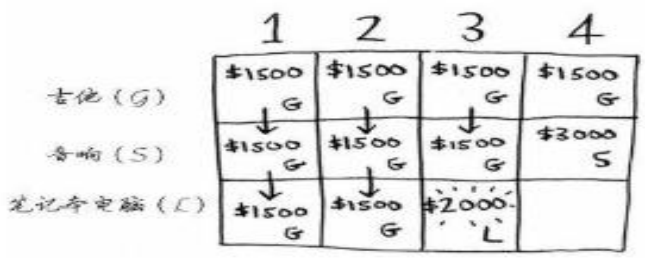


3. 笔记本电脑行

下面以同样的方式处理笔记本电脑。笔记本电脑重3磅，没法将其装入容量为1磅或2磅的背包，因此前两个单元格的最大价值还是1500美元。



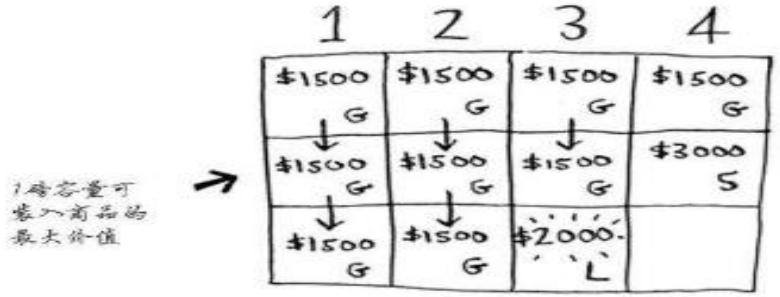
对于容量为3磅的背包，原来的最大价值为1500美元，但现在你可选择盗窃价值2000美元的笔记本电脑而不是吉他，这样新的最大价值将为2000美元！



对于容量为4磅的背包，情况很有趣。这是非常重要的部分。当前的最大价值为3000美元，你可不偷音响，而偷笔记本电脑，但它只值2000美元。

价值没有原来高。但等一等，笔记本电脑的重量只有3磅，背包还有1磅的容量没用！

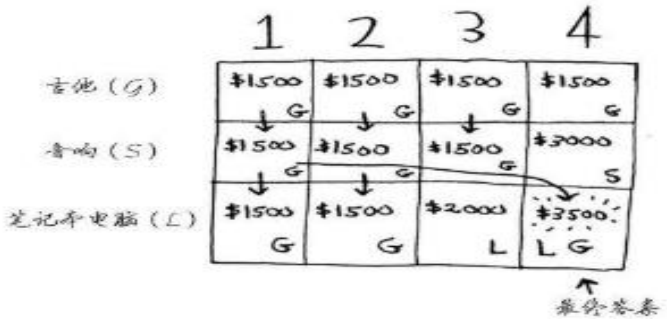
在1磅的容量中，可装入的商品的最大价值是多少呢？你之前计算过。



根据之前计算的最大价值可知，在1磅的容量中可装入吉他，价值1500美元。因此，你需要做如下比较。

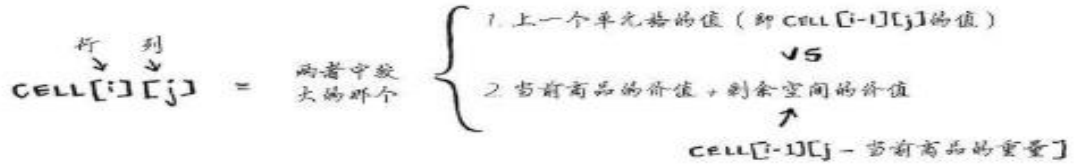
你可能始终心存疑惑：为何计算小背包可装入的商品的最大价值呢？但愿你现在明白了其中的原因！余下了空间时，你可根据这些子问题的答案来确定余下的空间可装入哪些商品。笔记本电脑和吉他的总价值为3500美元，因此偷它们是更好的选择。

最终的网格类似于下面这样。

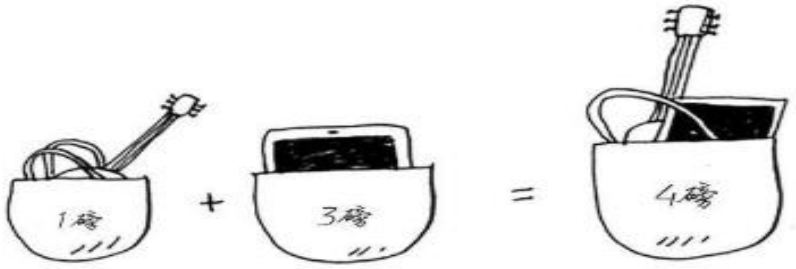


答案如下：将吉他和笔记本电脑装入背包时价值最高，为3500美元。

你可能认为，计算最后一个单元格的价值时，我使用了不同的公式。那是因为填充之前的单元格时，我故意避开了一些复杂的因素。其实，计算每个单元格的价值时，使用的公式都相同。这个公式如下。



你可以使用这个公式来计算每个单元格的价值，最终的网格将与前一个网格相同。现在你明白了为何要求解子问题吧？你可以合并两个子问题的解来得到更大问题的解。



9.2 背包问题 FAQ

你可能还是觉得这像是变魔术。本节将回答一些常见的问题。

9.2.1 再增加一件商品将如何呢

假设你发现还有第四件商品可偷——一个iPhone！

此时需要重新执行前面所做的计算吗？不需要。别忘了，动态规划逐步计算最大价值。到目前为止，计算出的最大价值如下。

	1	2	3	4
吉他（G）	$1500 G	$1500 G	$1500 G	$1500 G
音响（S）	$1500 G	$1500 G	$1500 G	$3000 S
笔记本电脑（L）	$1500 G	$1500 G	$2000 L	$3500 LG

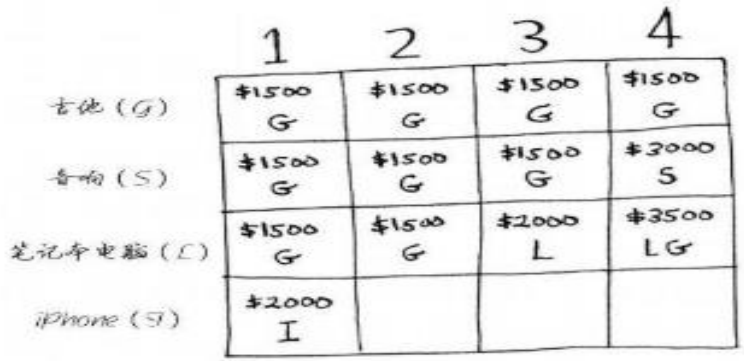
这意味着背包容量为4磅时，你最多可偷价值3500美元的商品。但这是以前的情况，下面再添加表示iPhone的行。

	1	2	3	4
吉他（G）	$1500 G	$1500 G	$1500 G	$1500 G
音响（S）	$1500 G	$1500 G	$1500 G	$3000 S
笔记本电脑（L）	$1500 G	$1500 G	$2000 L	$3500 LG
iPhone				

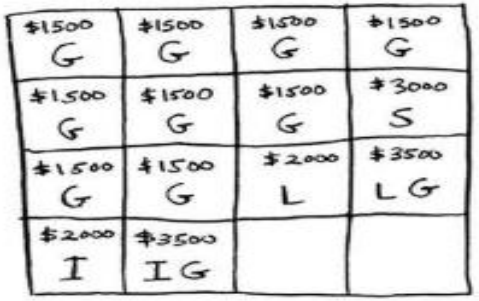
新的答案

最大价值可能发生变化！请尝试填充这个新增的行，再接着往下读。

我们从第一个单元格开始。iPhone可装入容量为1磅的背包。之前的最大价值为1500美元，但iPhone价值2000美元，因此该偷iPhone而不是吉他。



在下一个单元格中，你可装入iPhone和吉他。



对于第三个单元格，也没有比装入iPhone和吉他更好的选择了。

对于最后一个单元格，情况比较有趣。当前的最大价值为3500美元，但你可偷iPhone，这将余下3磅的容量。

3磅容量的最大价值为2000美元！再加上iPhone价值2000美元，总价值为4000美元。新的最大价值诞生了！

最终的网格如下。

以上的都建议自己手推一下，然后知道怎么回事，核心的部分是 142 页核心公式，待会代码会重现这个过程，推荐没有算法基础的小伙伴看这本书《算法图解》很有意思的书，讲的很清晰，入门足够

更深入的请阅读 python 算法-动态规划写的不错，可以参考

为什么要使用动态规划?

From:动态规划是什么，意义在哪里？！！！！
首先我们要知道为什么要使用(Dynamic programming)dp，我们在选择 dp 算法的时候，往往是在决策问题上，而且是在如果不使用 dp，直接暴力效率会很低的情况下选择使用 dp.
那么问题来了，什么时候会选择使用 dp 呢，一般情况下，我们能将问题抽象出来，并且问题满足无后效性，满足最优子结构，并且能明确的找出状态转移方程的话，dp 无疑是很好的选择。
无后效性通俗的说就是只要我们得出了当前状态，而不用管这个状态怎么来的，也就是说之前的状态已经用不着了，如果我们抽象出的状态有后效性，很简单，我们只用把这个值加入到状态的表示中。
最优子结构(自下而上)：在决策问题中，如果，当前问题可以拆分为多个子问题，并且依赖于这些子问题，那么我们称为此问题符合子结构，而若当前状态可以由某个阶段的某个或某些状态直接得到，那么就符合最优子结构
重叠子问题(自上而下)：动态规划算法总是充分利用重叠子问题，通过每个子问题只解一次，把解保存在一个需要时就可以查看的表中，每次查表的时间为常数，如备忘录的递归方法。斐波那契数列的递归就是个很好的例子
状态转移：这个概念比较简单，在抽象出上述两点的的状态表示后，每种状态之间转移时值或者参数的变化。

小结

动态规划： 动态规划表面上很难，其实存在很简单的套路：当求解的问题满足以下两个条件时， 就应该使用动态规划：
1、主问题的答案 包含了 可分解的子问题答案 （也就是说，问题可以被递归的思想求解）
2、递归求解时， 很多子问题的答案会被多次重复利用
动态规划的本质思想就是递归， 但如果直接应用递归方法， 子问题的答案会被重复计算产生浪费， 同时递归更加耗费栈内存， 所以通常用一个二维矩阵（表格）来表示不同子问题的答案， 以实现更加高效的求解。

Talk is cheap ,Show me the code

翻阅很多资料，貌似 python 描述的比较少，这里总结一下，用前面的图解中的伪代码重构下

背包问题

多谢 rubik_wong–0/1 背包问题，代码参考如下

```
# 这里使用了图解中的吉他，音箱，电脑，手机做的测试，数据保持一致
w = [0, 1, 4, 3, 1]   #n 个物体的重量(w[0]无用)
```

```
p = [0, 1500, 3000, 2000, 2000]   #n 个物体的价值(p[0]无用)
n = len(w) - 1   #计算 n 的个数
m = 4   #背包的载重量

x = []   #装入背包的物体，元素为 True 时，对应物体被装入(x[0]无用)
v = 0
#optp[i][j]表示在前 i 个物体中，能够装入载重量为 j 的背包中的物体的最大价值
optp = [[0 for col in range(m + 1)] for raw in range(n + 1)]
#optp 相当于做了一个 n*m 的全零矩阵的赶脚，n 行为物件，m 列为自背包载重量

def knapsack_dynamic(w, p, n, m, x):
    #计算 optp[i][j]
    for i in range(1, n + 1):       # 物品一件件来
        for j in range(1, m + 1):   # j 为子背包的载重量，寻找能够承载物品的子背包
            if (j >= w[i]):         # 当物品的重量小于背包能够承受的载重量的时候，才考虑能不能放进去
                optp[i][j] = max(optp[i - 1][j], optp[i - 1][j - w[i]] + p[i])    # optp[i - 1][j]是上一个单元的值， optp[i - 1][j
 - w[i]]为剩余空间的价值
            else:
                optp[i][j] = optp[i - 1][j]

    #递推装入背包的物体,寻找跳变的地方，从最后结果开始逆推
    j = m
    for i in range(n, 0, -1):
        if optp[i][j] > optp[i - 1][j]:
            x.append(i)
            j = j - w[i]

    #返回最大价值，即表格中最后一行最后一列的值
    v = optp[n][m]
    return v

print '最大值为：' + str(knapsack_dynamic(w, p, n, m, x))
print '物品的索引：',x
```

```
#最大值为：4000
#物品的索引： [4, 3]
```

优化背包问题的递归方法

参考自：麻省理工的 背包算法 python

```
def MaxVal2(memo , w, v, index, last):
    """
    得到最大价值
    w 为 widght
    v 为 value
    index 为索引
    last 为剩余重量
    """

    global numCount
    numCount = numCount + 1

    try:
        #以往是否计算过分支，如果计算过，直接返回分支的结果
        return memo[(index , last)]
    except:
        #最底部
        if index == 0:
            #是否可以装入
            if w[index] <= last:
                return v[index]
            else:
                return 0

        #寻找可以装入的分支
        without_1 = MaxVal2(memo , w, v, index - 1, last)

        #如果当前的分支大于约束
        #返回历史查找的最大值
        if w[index] > last:
```

```
            return without_1
        else:
            #当前分支加入背包，剪掉背包剩余重量，继续寻找
            with_1 = v[index] + MaxVal2(memo , w, v , index - 1, last - w[index])

        #比较最大值
        maxvalue = max(with_1 , without_1)
        #存储
        memo[(index , last)] = maxvalue
        return maxvalue

w = [0, 1, 4, 3, 1]   # 东西的重量
v = [0, 1500, 3000, 2000, 2000]       # 东西的价值

numCount = 0
memo = {}
n = len(w) - 1
m = 4
print MaxVal2(memo , w, v, n, m) , "caculate count : ", numCount

# 4000 caculate count :  20
```

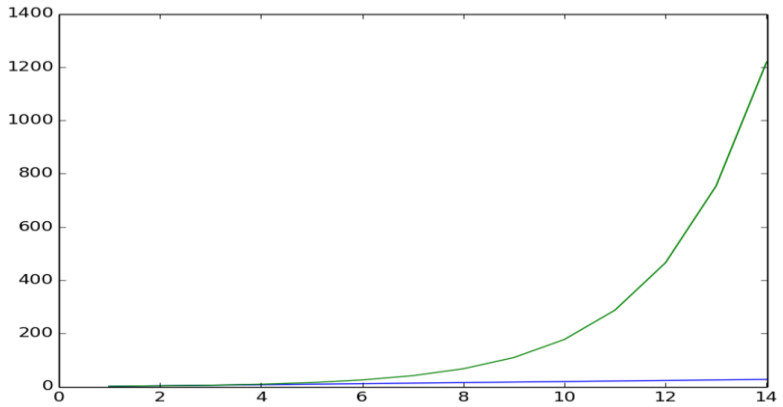
优化斐波那契数列的递归方法

多谢 Python 科学实验----动态规划,也就是对应上面的重叠子问题的方法，备忘录的递归方法

```
#Dynamic Method Experiment
import matplotlib.pyplot as plt
count=0;
#blank
def f(n):
    global count
    count=count+1
    if n==1:
        return 1
    elif n==0:
        return 1
```

```
    else:
        return f(n-1)+f(n-2)


# function calls count
def calc_f(n):
    global count
    count=0
    f(n)
    return count

#using memorization
mem={}

def mem_f(n):
    global count,mem
    count=count+1
    if n in mem:
        return mem[n]
    else:
        if n==1:
            result=1
        elif n==0:
            result=1
        else:
            result=mem_f(n-1)+mem_f(n-2)
        mem[n]=result
        return result


def mem_calc_f(n):
    global count
    global mem
    mem={}
```

```
    count=0
    mem_f(n)
    return count


x=range(1,15)
y=[]
y2=[]
for i in x:

    c=mem_calc_f(i)
    y.append(c)

    c2=calc_f(i)
    y2.append(c2)
    print "规模为%d 时计算了%d 次 i=%d 时, val=%d"%(i,c,i,mem_f(i))
    print "规模为%d 时计算了%d 次 i=%d 时, val=%d"%(i,c2,i,f(i))
plt.plot(x,y)
plt.plot(x,y2)
plt.show()
```



它的基本思想就是记录已经计算过的值，避免重复计算。

如果使用装饰器的写法，则会优雅很多

```
from functools import wraps
```

```
def memo(func):
    cache={}
    @wraps(func)
    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)

fib(2)
```

换个角度考虑，如果用生成器，那么将会更加优雅

```
def fib(n=None):
        l1 = 0
        l2 = 1
        result = 1
        i = 0
        forever = n is None
        while forever or i < n:
                yield result
                result = l1 + l2
                l1 = l2
                l2 = result
                i +=1

c = fib(5)
for _ in c:
        print (_)
```

```
1
1
2
3
5
```

n 如果不传递，那么将不断的可以生成斐波那契数列

一些利用 DP 的笔试题

CPU 双核问题

网易笔试—动态规划: 题目的大概意思：一种双核 CPU 的两个核能够同时的处理任务，现在有 n 个已知数据量的任务需要交给 CPU 处理，假设已知 CPU 的每个核 1 秒可以处理 1kb，每个核同时只能处理一项任务。n 个任务可以按照任意顺序放入 CPU 进行处理，现在需要设计一个方案让 CPU 处理完这批任务所需的时间最少，求这个最小的时间。

输入包括两行：

第一行为整数 n(1 ≤ n ≤ 50)

第二行为 n 个整数 length[i](1024 ≤ length[i] ≤ 4194304)，表示每个任务的长度为 length[i]kb，每个数均为 1024 的倍数。

输出一个整数，表示最少需要处理的时间。

问题实质是动态规划问题，把数组分成两部分，使得两部分的和相差最小。

如何将数组分成两部分使得两部分的和的差最小？参考博客 http://www.tuicool.com/articles/ZF73Af

思路：

差值最小就是说两部分的和最接近，而且各部分的和与总和的一半也是最接近的。假设用 sum1 表示第一部分的和，sum2 表示第二部分的和，SUM 表示所有数的和，那么 sum1+sum2=SUM。假设 sum1<sum2 那么 SUM/2-sum1 = sum2-SUM/2;

所以我们就有目标了，使得 sum1<=SUM/2 的条件下尽可能的大。也就是说从 n 个数中选出某些数，使得这些数的和尽可能的接近或者等于所有数的和的一般。这其实就是简单的背包问题了：

背包容量是 SUM/2. 每个物体的体积是数的大小，然后尽可能的装满背包。

```
w = [0, 3072, 3072, 7168, 3072, 1024]  # 假设进入处理的的任务大小
w = map(lambda x:x/1024,w)  # 转化下
p = w  # 这题的价值和任务重量一致
n = sum(w)/2 +1 # 背包承重为总任务的一半
optp = [[0 for j in range(n+1)] for i in range(len(w))]
for i in range(1,len(p)):
    for j in range(1,n+1):
        if j >= p[i]:
            optp[i][j] = max(optp[i-1][j],p[i]+optp[i-1][j-w[i]])
```

```
        else:
            optp[i][j] = optp[i-1][j]
print optp[-1][-1]
print optp
# 背包矩阵入下所示，第一列和第一行无效占位符
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 3, 3, 3, 3, 3, 3, 3],
 [0, 0, 0, 3, 3, 3, 6, 6, 6, 6],
 [0, 0, 0, 3, 3, 3, 6, 7, 7, 7],
 [0, 0, 0, 3, 3, 3, 6, 7, 7, 9],
 [0, 1, 1, 3, 4, 4, 6, 7, 8, 9]]
```

LIS 问题

```
longest increasing subsequence 问题，
写的很棒，不再赘述 DP 动态规划（Python 实现）
# 讲 DP 基本都会讲到的一个问题 LIS：longest increasing subsequence
# http://www.deeplearn.me/216.html
lis = [2 ,1, 5, 3, 6 ,4 ,8 ,9, 7]

d = [1]*len(lis)
res = 1
for i in range(len(lis)):
    for j in range(i):
        if lis[j] <= lis[i] and d[i] < d[j]+1:
            d[i] = d[j]+1
        if d[j] >  res:
            res = d[j]
print res
```

LCS 问题

```
一个非常好的图解教程：动态规划 最长公共子序列 过程图解
# 根据图解教程写的伪代码，其实最后评论里面的代码就是我添加上去的

s1 = [1,3,4,5,6,7,7,8]
s2 = [3,5,7,4,8,6,7,8,2]
```

```
d = [[0]*(len(s2)+1) for i in range(len(s1)+1) ]

for i in range(1,len(s1)+1):
    for j in range(1,len(s2)+1):
        if s1[i-1] == s2[j-1]:
            d[i][j] = d[i-1][j-1]+1
        else:
            d[i][j] = max(d[i-1][j],d[i][j-1])



print "max LCS number:",d[-1][-1]
```

给定一个有 n 个正整数的数组 A 和一个整数 sum

给定一个有 n 个正整数的数组 A 和一个整数 sum,求选择数组 A 中部分数字和为 sum 的方案数。
当两种选取方案有一个数字的下标不一样,我们就认为是不同的组成方案。

输入描述:

输入为两行:
第一行为两个正整数 n(1 ≤ n ≤ 1000), sum(1 ≤ sum ≤ 1000)
第二行为 n 个正整数 A[i](32 位整数), 以空格隔开。

输出描述:

输出所求的方案数

示例 1

输入

```
5 15
5 5 10 2 3
```

输出

```
4
#动态规划算法。dp[i][j]代表用前 i 个数字凑到 j 最多有多少种方案。
#dp[i][j]=dp[i-1][j];   //不用第 i 个数字能凑到 j 的最多情况
#dp[i][j]+=dp[i-1][j-value[i]];用了 i 时, 只需要看原来凑到 j-value[i]的最多情况即可。并累加
num_ = 5
sum_ = 10
line = [5 ,5 ,10 ,2 ,3]
optp = [[1]+[0]*sum_ for i in range(num_+1)]  # 第一列为 1 的原因是和为 0 的时候只有一种取法, 就是什么都不取
```

```
for i in range(1,num_+1):
    for j in range(1,sum_+1):
        if j - line[i-1] >=0:
            optp[i][j] = optp[i-1][j] + optp[i-1][j-line[i-1]]
        else:
            optp[i][j] = optp[i-1][j]
print optp
       0   1  2  3  4  5  6  7  8  9 10
0  [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
5   [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
5   [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1],
10  [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 2],
2   [1, 0, 1, 0, 0, 2, 0, 2, 0, 0, 2],
3   [1, 0, 1, 1, 0, 3, 0, 2, 2, 0, 4]]
# 转化为背包问题后，开始推，（注意第一行和第一列是预至位）比如说第一个数字是 5，那么从构成和为 1，怎么取？当然没得取，直到构成和为 5 的时候，开始执行，如果用这个 5，那么还剩下 5-5=0 的和，0 的和取法只有 1，而如果不用 5，取法只有 0，所以为 1，之后重复推，再说第二个 5，直到和为 5 之前，都是 0 取法，到了 5 之后，两种取法，一种是要不要这个新的 5，如果要这个新的 5，那么剩下的和即 5-5=0，一种取法，如果这个新的 5 不取，那以前能取到和为 5 就上次循环中的一种取法，所以合起来两种取法
```

一个数组有 N 个元素，求连续子数组的最大和

一个数组有 N 个元素，求连续子数组的最大和。 例如：[-1,2,1]，和最大的连续子数组为[2,1]，其和为 3

输入描述:

输入为两行。
第一行一个整数 n(1 <= n <= 100000)，表示一共有 n 个元素
第二行为 n 个数，即每个元素,每个整数都在 32 位 int 范围内。以空格分隔。

输出描述:

所有连续子数组中和最大的值。

示例 1

输入

```
3
-1 2 1
```

输出

```
3
# 采用动态规划的方法
```

```
# 设 dp[i]表示以第 i 个元素为结尾的连续子数组的最大和，则递推方程式为 dp[i]=max{dp[i-1]+a[i], a[i]};
num = raw_input("")
line = raw_input("")
line = map(lambda x:int(x),line.split(" "))
num = int(num)

d =[0]*(num-1)
d.insert(0,line[0])

for i in range(1,num):

    d[i] = max(d[i-1]+line[i],line[i])

print max(d)
```

X*Y 的网格迷宫

有一个 X*Y 的网格，小团要在此网格上从左上角到右下角，只能走格点且只能向右或向下走。请设计一个算法，计算小团有多少种走法。给定两个正整数 int x,int y，请返回小团的走法数目。

输入描述:

输入包括一行，逗号隔开的两个正整数 x 和 y，取值范围[1,10]。

输出描述:

输出包括一行，为走法的数目。

示例 1

输入

```
3 2
```

输出

```
10
# 动态规划，使用递推方程 d[i][j] = d[i-1][j] + d[i][j-1]
# 因为可能从两个方向走到同一个点，所以从上到下为一种走法，从左到右是另一种走法
# 注意题目给的是 x*y 方格，所以是(x+1)*(y+1)个点



line = map(int, raw_input("").split(" "))
x = line[0]
```

```
y = line[1]

d = [[0]*(y+2) for i in range(x+2)]  # 比较冗余，需要额外开辟扩展的行和列

for i in range(1,x+2):
    for j in range(1,y+2):
        if i==j and i==1:
            d[i][j] = 1
        else:
                d[i][j] = d[i-1][j] + d[i][j-1]

print d[-1][-1]

# 解法 2，不需要扩展行列
def findMatrix(m,n):
        d = [[0]*(n+1) for _ in range(m+1)]
        for _ in range(n+1):
                d[0][_] = 1
        for _  in range(m + 1):
                d[_][0] = 1
        for i in range(1,m+1):
                for j in range(1,n+1):
                        d[i][j] = d[i-1][j] + d[i][j-1]
        return d
d = findMatrix(5,7)

[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 2, 3, 4, 5, 6, 7, 8],
 [1, 3, 6, 10, 15, 21, 28, 36],
 [1, 4, 10, 20, 35, 56, 84, 120],
 [1, 5, 15, 35, 70, 126, 210, 330],
 [1, 6, 21, 56, 126, 252, 462, 792]]
d[-1][-1]即为解
```

暗黑字符串

一个只包含’A’、‘B’和’C’的字符串，如果存在某一段长度为 3 的连续子串中恰好’A’、’B’和’C’各有一个，那么这个字符串就是纯净的，否则这个字符串就是暗黑的。例如：
BAACAACCBAAA 连续子串"CBA"中包含了’A’，‘B’,’C’各一个，所以是纯净的字符串
AABBCCAABB 不存在一个长度为 3 的连续子串包含’A’，‘B’，‘C’,所以是暗黑的字符串
你的任务就是计算出长度为 n 的字符串(只包含’A’、‘B’和’C’)，有多少个是暗黑的字符串。

输入描述:

输入一个整数 n，表示字符串长度(1 ≤ n ≤ 30)

输出描述:

输出一个整数表示有多少个暗黑字符串

示例 1

输入

3

输出

21

思路解析

以下是一个思考过程，动态规划问题的常用套路，自己手打，就发个图片好了。
想用动态规划解决这种问题，就需要推导出公式：

$$f(n) = \sum_{i=1}^{M} K_i * f(n-i)$$

这个是最常见的一种情况，M表示和过去相关的状态数，Ki表示对应状态对应的系数。

在本题中，如果要求长度为n的暗黑串数量，先假设只和前一个n-1的暗黑串相关，那么从前一个往后扩展一位，就需要加ABC其中 一个字母，此时要保证还是暗黑串，就需要考察n-1状态时末尾两个字母的状态，末尾两个字母的状态只有相同和不相同两种。

0. f(n-1) = S(n-1) + D(n-1)
1. 如果相同（假设有S(n-1)个），那么新增加的字母ABC都可以，有3*S(n-1)种
2. 如果不相同（假设有D(n-1)个），那么新增加的字母只能是那两个字母中的一个，比如AB知只能扩展为ABA和ABB，有2*D(n-1)种

那么可以得出 f(n) = 3*S(n-1) + 2*D(n-1) = 2*f(n-1) + S(n-1)。
此时发现多了一个S(n-1)项无法消除，因为 S(n-1) 是 f (n-1) 的一个子集，我们自然需要去更前面的状态，比如 f(n-2) 中去寻找 S(n-1) 和 f(n-2) 的关系。

此时再观察上文的两个情况分析，

1. 如果相同（假设有S(n-1)个），那么新增加的字母ABC都可以，有3*S(n-1)种
   ├---这其中，扩展之后的结果中，有1/3的结果是相同的，个数就是S(n-1)，比如AA分别变为AAA,AAB,AAC，而不相同的有2*S(n-1)
2. 如果不相同（假设有D(n-1)个），那么新增加的字母只能是那两个字母中的一个，比如AB知只能扩展为ABA和ABB，有2*D(n-1)种
   ----这其中，扩展之后的结果中，有1/2的结果是相同的，个数就是D(n-1)，比如AB分别变为ABA,ABB，不相同的也是D(n-1)

0. f(n-1) = S(n-1) + D(n-1)

----对这个式子有 f(n) = S(n) + D(n)，结合1、2中的分析可知，
S(n) = S(n-1) + D(n-1)； D(n) = 2*S(n-1) + D(n-1) 则可以得到 S(n) = f(n-1) 。
代入上文得出的阶段性结论 f(n) = 3*S(n-1) + 2*D(n-1) = 2*f(n-1) + S(n-1)
可得 f(n) = 2*f(n-1) + f(n-2)

接下来代码就是随便写写的事了。

```
#方式二，这么 low 的方式是我根据上面的解析写的。递归所以速度慢
num = int(raw_input(""))

def dark(num):
```

```
    if num == 1:
        return 3
    elif num==2:
        return 9
    else:
        return 2*dark(num-1) + dark(num-2)

print dark(num)
# 方式一：别人家的代码
n = int(raw_input())
dp = [0]*31
dp[0] = 3
dp[1] = 9
for i in xrange(2, n):
    dp[i] = 2*dp[i-1]+dp[i-2]

print dp[n-1]
```

数据结构

222、数组中出现次数超过一半的数字-Python 版

1.数组中出现次数超过一半的数字
数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如
输入一个长度为 9 的数组

{1,2,3,2,2,2,5,4,2}。

由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此

输出 2。

如果不存在则

输出 0。

---

python-dict
python–collections–counter
1.利用 dict.setdefault(key, default=None)

使用 dict 时，如果引用的 Key 不存在，就会抛出 KeyError。如果希望 key 不存在时，返回一个默认值，如果存在则返回该 value 值,就可以用 defaultdict:

dict.setdefault(key, 0)#返回的是 对应 key 值，如果不存在则返回的 value 值为 0，如果存在则返回该 value 值

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1 存在
'abc'
>>> dd['key2'] # key2 不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建 defaultdict 对象时传入。
除了在 Key 不存在时返回默认值，defaultdict 的其他行为跟 dict 是完全一样的。

```
# -*- coding:utf-8 -*-
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        length = len(numbers)
        d = {}
        d[1] = 2
        for num in numbers:
            d[num] = d.setdefault(num, 0) + 1
        for key in d.keys():  #python 中 for 循环近似于 range base for
             if d[key]*2 > length:
                return int(key)
        return 0
```

2.利用 python 的 collections 这个库中的计数器 Counter

```
Counter
```

`Counter` 是一个简单的计数器，例如，统计字符出现的个数，这里是统计一个 `list` 中不同元素的个数，且返回的为 `dict()`类

```
>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})
```

代码：

```
# -*- coding:utf-8 -*-
```

```
from collections import Counter #导入 collections 库中的 Counter 类
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        length = len(numbers)
        c = collections.Counter(numbers)
        for k,v in c.items():
            if v*2>length: #大于一半长度
                return k
        return 0
```

同理

```
# -*- coding:utf-8 -*-
from collections import Counter
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        if not numbers: return 0
        count = Counter(numbers).most_common()
        if count[0][1] > len(numbers) / 2: #换汤不换药
            return count[0][0]
        return 0
```

- collections 是 Python 内建的一个集合模块，提供了许多有用的集合类。
- python–collections–counter
- 实际上 Counter 也是 dict 的一个子类，上面的结果可以看出，字符’ g’ 、’ m’ 、’ r’ 各出现了两次，其他字符各出现了一次。

3.利用 list 的特点，使用一个 times 保存当前数字出现了多少次。这样，如果出现了 0 次，把新数字加入 res；如果当前数字和 res 相等，那么 times+=1；如果不等 times-=1。

最后遍历完成之后 res 保存的数字，应该是出现超过一半的数字或者没有出现超过一半的数字。因此，最后判断再进行一个判断；

```
# -*- coding:utf-8 -*-
from collections import Counter
class Solution:
    def MoreThanHalfNum_Solution(self, numbers):
        if not numbers: return 0
        res = 0
        times = 0
```

```
        for i, num in enumerate(numbers):
            if times == 0:
                res = num
                times = 1
            elif num == res:
                times += 1
            else:
                times -= 1
        return res if numbers.count(res) > len(numbers) / 2 else 0
```

223、求 100 以内的质数

题目： 获取 100 以内的质数。

程序分析：质数（prime number）又称素数，有无限个。质数定义为在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的数称为质数，如：2、3、5、7、11、13、17、19。

方法一：

```
#!/usr/bin/python # -*- coding: UTF-8 -*- num=[]; i=2 for i in range(2,100): j=2 for j in range(2,i): if(i%j==0): break else: num.append(i)
print(num)
```

方法二：

```
import math def func_get_prime(n): return filter(lambda x: not [x%i for i in range(2, int(math.sqrt(x))+1) if x%i ==0], range(2,n+1)) print
func_get_prime(100)
```

输出结果为：

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

224、无重复字符的最长子串-Python 实现

方法 01、leetcode 无重复字符的最长子串 python 实现

无重复字符的最长子串

给定一个字符串，找出不含有重复字符的**最长子串**的长度。

示例 1:

```
输入: "abcabcbb"
输出: 3
解释: 无重复字符的最长子串是 "abc"，其长度为 3。
```

示例 2:

```
输入: "bbbbb"
输出: 1
解释: 无重复字符的最长子串是 "b"，其长度为 1。
```

示例 3:

```
输入: "pwwkew"
输出: 3
解释: 无重复字符的最长子串是 "wke"，其长度为 3。
     请注意，答案必须是一个子串，"pwke" 是一个 子序列 而不是子串。
```

这道题需要借助哈希查找 key 的 O(n) 时间复杂度， 否则就会超时

初始化一个 哈希表\字典 dic

头指针 start 初始为 0

当前指针 cur 初始为 0

最大长度变量 l 初始为 0

用 cur 变量从给定字符串 str 的开头开始 一位一位的向右查看字符，直到整个字符串遍历完， 对每一位字符进行如下：

当前位置的字符为 c = str[cur]

查询当前字符 c 是否 在哈希表 dic 的键 当中,表示 当前字符 c 是否之前遍历到过

如果 当前字符还没出现过，就 在 dic 中记录一个键值对 (当前字符 c，当前位置 cur)

cur 后移一位

如果 当前字符出现过， 获取 当前字符串 c 上次出现的位置 pre = dic[c]

如果 pre 在 start 后面即 pre>start， 则把 start 移动到 pre 的下一位， start = pre + 1， 这样保证 cur 继续向后遍历中 从 start 到 cur 没有重复元素

否则 start 不动，start 移动到某一个位置，说明在这个位置之前有重复的元素了，所以 start 只往后移动不往回移动

这时候在衡量一下 如果 cur - start + 1 (衡量当前没重复子串开头到结尾的长度) 比 长度变量 l 大， 那就替换 l 为 cur - start + 1

```python
class Solution:
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        l = 0
        start = 0
        dic = {}
        for i in range(len(s)):
            cur = s[i]
            if cur not in dic.keys():
                dic[cur] = i
            else:
                if dic[cur] + 1 > start:
                    start = dic[cur] + 1
                dic[cur] = i
            if i - start + 1 > l:
                l = i - start + 1

        return l


if __name__ == '__main__':
    s = Solution()
    # print(s.lengthOfLongestSubstring("abcabcbb"))
    # print(s.lengthOfLongestSubstring("abba"))
    print(s.lengthOfLongestSubstring("aabaab!bb"))
    # print(s.lengthOfLongestSubstring("bbbbb"))
```

方法 02：
给定一个字符串，找出不含有重复字符的最长子串的长度。
示例：
给定 “abcabcbb” ，没有重复字符的最长子串是 “abc” ，那么长度就是 3。
给定 “bbbbb” ，最长的子串就是 “b” ，长度是 1。
给定 “pwwkew” ，最长子串是 “wke” ，长度是 3。请注意答案必须是一个子串，” pwke” 是 子序列 而不是子串。

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        # 存储历史循环中最长的子串长度
        max_len = 0
        # 判断传入的字符串是否为空
        if s is None or len(s) == 0:
            return max_len
        # 定义一个字典，存储不重复的字符和字符所在的下标
        str_dict = {}
        # 存储每次循环中最长的子串长度
        one_max = 0
        # 记录最近重复字符所在的位置+1
        start = 0
        for i in range(len(s)):
            # 判断当前字符是否在字典中和当前字符的下标是否大于等于最近重复字符的所在位置
            if s[i] in str_dict and str_dict[s[i]] >= start:
                # 记录当前字符的值+1
                start = str_dict[s[i]] + 1
            # 在此次循环中，最大的不重复子串的长度
            one_max = i - start + 1
            # 把当前位置覆盖字典中的位置
            str_dict[s[i]] = i
            # 比较此次循环的最大不重复子串长度和历史循环最大不重复子串长度
```

```
            max_len = max(max_len, one_max)
        return max_len

if __name__ == '__main__':
    sol = Solution()
    print(sol.lengthOfLongestSubstring("bbbbb"))
    print(sol.lengthOfLongestSubstring("eeydgwdykpv"))
    print(sol.lengthOfLongestSubstring("pwwkew"))
    print(sol.lengthOfLongestSubstring("abcabcbb"))
```

225、通过 2 个 5/6 升得水壶从池塘得到 3 升水

假设有一个池塘，里面有无穷多的水。现有 2 个空水壶，容积分别为 5 升和 6 升。问题是如何只用这 2 个水壶从池塘里取得 3 升的水。

答案：由满 6 向空 5 倒，剩 1 升，把这 1 升倒 5 里，然后 6 剩满，倒 5 里面，由于 5 里面有 1 升水，因此 6 只能向 5 倒 4 升水，然后将 6 剩余的 2 升，倒入空的 5 里面，再灌满 6 向 5 里倒 3 升，剩余 3 升。

226、什么是 MD5 加密，有什么特点?

1、md5 的加密提点

（1）md5 加密不可逆，所以它的安全度比较高

（2）不管多大的字符串，它都能生成 32 位字符串

2、用法

```
import hashlib
def getMd5(value):
    md5 = hashlib.md5()
    md5.update(value,encoding='utf-8')
    result = md5.hexdigest()
```

227、什么是对称加密和非对称加密

1.对称加密

对称加密是最快速、最简单的一种加密方式，加密（encryption）与解密（decryption）用的是同样的密钥（secret key）.对称加密有很多种算法，由于它效率很高，所以被广泛使用在很多加密协议的核心当中.

对称加密的一大缺点是密钥的管理与分配，换句话说，如何把密钥发送到需要解密你的消息的人的手里是一个问题。在发送密钥的过程中，密钥有很大的风险会被黑客们拦截。现实中通常的做法是将对称加密的密钥进行非对称加密，然后传送给需要它的人.

1.1 常见的对称加密有 DES、三 重 DES、AES 等

2.非对称加密

非对称加密为数据的加密与解密提供了一个非常安全的方法，它使用了一对密钥，公钥（public key）和私钥（private key）.私钥只能由一方安全保管，不能外泄，而公钥则可以发给任何请求它的人.非对称加密使用这对密钥中的一个进行加密，而解密则需要另一个密钥.
比如，你向银行请求公钥，银行将公钥发给你，你使用公钥对消息加密，那么只有私钥的持有人--银行才能对你的消息解密。与对称加密不同的是，银行不需要将私钥通过网络发送出去，因此安全性大大提高.
例如,我之前公司是做数字签名的,网络请求时,使用非对称加密,是服务器端保留公钥,私钥发给特定的客户,客户在客户端使用私钥加密,服务器用公钥解密.
非对称加密:

- 只能私钥加密公钥解密,或者公钥加密私钥解密;
- 一方保留密钥,一方公开密钥;不能同时公开,具体公开哪方需要根据实际情况来决定

3.非对称加密算法常见有:RSA 算法(3 个人名简写)和 diffie - hellman 算法(迪菲•赫尔曼算法)

230、如何判断单向链表中是否有环?

用 python 判断一个单链表是否有环.

用 python 判断一个单链表是否有环.

思路 1:

```
判断一个单链表是否有环,
可以用 set 存放每一个 节点, 这样每次 访问后把节点丢到这个集合里面.
其实 可以遍历这个单链表, 访问过后,
如果这个节点 不在 set 里面, 把这个节点放入到 set 集合里面.
如果这个节点在 set 里面 , 说明曾经访问过, 所以这个链表有重新 走到了这个节点, 因此一定有环
如果链表都走完了, 把所有的节点都放完了. 还是没有重复的节点, 那说明没有环.
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@Time    : 2019/1/12 00:59
@File    : has_circle.py
@Author  : frank.chang@shoufuyou.com
https://leetcode.com/problems/linked-list-cycle/
141. Linked List Cycle
Easy
1231
```

93
Favorite
Share
Given a linked list, determine if it has a cycle in it.
To represent a cycle in the given linked list,
we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to.
If pos is -1, then there is no cycle in the linked list.
Example 1:
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where tail connects to the second node.
Example 2:
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where tail connects to the first node.
Example 3:
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
Follow up:
Can you solve it using O(1) (i.e. constant) memory?
Accepted
340,579
Submissions
971,443
"""

```
class LinkNode:
    def __init__(self, value):
        self.value = value
```

```
        self.next = None
class Solution1:
    """
    思路分析:
    判断一个单链表是否有环,
    可以用 set 存放每一个 节点, 这样每次 访问后把节点丢到这个集合里面.
    其实 可以遍历这个单链表, 访问过后,
    如果这个节点 不在 set  里面, 把这个节点放入到 set 集合里面.
    如果这个节点在  set 里面 , 说明曾经访问过, 所以这个链表有重新 走到了这个节点, 因此一定有环.
    如果链表都走完了, 把所有的节点都放完了. 还是没有重复的节点, 那说明没有环.
    """
    def hasCycle(self, head):
        mapping = set()
        flag = False
        p = head
        while p:
            if p not in mapping:
                mapping.add(p)
            else:
                flag = True
                break
            p = p.next
        return flag
```

还有一个解决方案:

```
定义 两个指针, 一个快指针 fast, 一个慢指针 slow, 快指针一次都两步,慢指针一次走一步.
如果 两个指针相遇了, 则说明链表是有环的.
如果 fast 都走到 null 了, 还没有相遇则说明没有环.
```

为什么是这样呢? 简单分析一下?

用图形来分析一下,这样可以清晰一点.

图形分析

因为快指针 先走 所以快指针先进入环,之后慢指针后进入环, 无论如何,

最后 要么 慢指针进入环的时候, 快指针可能已经走了 很多遍环, 也有可能没有走完环. 但无论如何 当慢指针 进入环的时候,

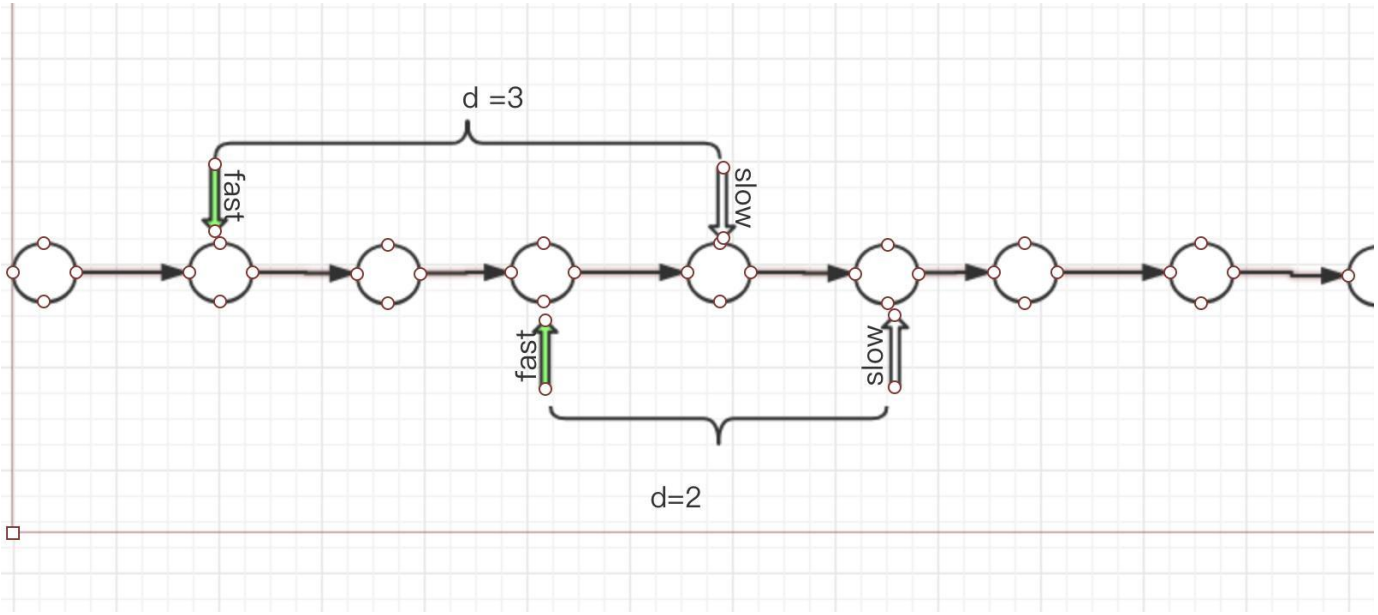
fast 有可能在 慢指针的后面, 或者前面, 无论如何 快指针 是必慢指针走的快的 , 所以 只要有环 一定可以 和慢指针来一次相遇.

你可能想 会不会错过呢?

答案 是不会的. 你想 快指针一次 走两步, 慢指针一次都一步.

假设 这是一条无穷尽的单链表. 他们 每走一步, 两者之间的距离就减 1, 所以 只要链表足够长, 是不是一定会相遇.

看下图:




```
class Solution:
    """
    定义 两个指针, 一个快指针 fast, 一个慢指针 slow,  快指针一次都两步,慢指针一次走一步.
    如果 两个指针相遇了, 则说明链表是有环的.
    如果 fast 都走到 null 了, 还没有相遇则说明没有环.
    """
    def hasCycle(self, head):
```

```
        flag = False
        if head is None or head.next is None or head.next.next is None:
            return flag
        fast = head.next.next
        slow = head.next
        while fast is not slow:
            if fast.next is None or fast.next.next is None:
                # no circle
                return flag
            fast = fast.next.next
            slow = slow.next
        # 相遇肯定有环
        if fast is slow:
            # hasCircle
            flag = True
        return flag
if __name__ == '__main__':
    pass
```

235、两数之和 Two Sum

题目描述：

中文：

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

英文：

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

方法 01：

示例：

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
```

所以返回 [0, 1]

代码

```
class Solution:
   def twoSum(self, nums, target):
      """
      :type nums: List[int]
      :type target: int
      :rtype: List[int]
      """
      d = dict()
      for k, i in enumerate(nums):
         p = target - i
         if p in d:
            return [d[p], k]
         else:
```

方法 02：

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        dict= {}
        for i in range(0,len(nums)):
            if nums[i] in dict :
                return dict[nums[i]] ,i
            else :
                dict[target - nums[i]] = i
```

236、搜索旋转排序数组 Search in Rotated Sorted Array

python 实现 Search in Rotated Sorted Array 搜索旋转排序数组

中文：假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 O(log n) 级别。

英文：Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.
Your algorithm's runtime complexity must be in the order of O(log n).

```python
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        start = 0
        end = len(nums)-1
        while start<=end:
            mid = (start + end)/2   #对于整数会自动省去小数部分
            if nums[mid] == target:
                return mid
            if nums[mid]>=nums[start]:
                if target >= nums[start] and target<=nums[mid]:
                    end = mid-1
                else:
                    start = mid + 1

            if nums[mid]<nums[end]:
                if target > nums[mid] and target<=nums[end]:
                    start = mid+1

                else:
                    end = mid -1
```

```
26          return -1
```

237、Python 实现一个 Stack 的数据结构

方法 01：

队、栈和链表一样，在数据结构中非常基础一种数据结构，同样他们也有各种各样、五花八门的变形和实现方式。但不管他们形式上怎么变，队和栈都有其不变的最基本的特征，我们今天就从最基本，最简单的实现来看看队列和堆栈。

不管什么形式的队列，它总有的一个共同的特点就是"先进先出"。怎么理解呢?就像是超市排队结账，先排队的人排在队的前面，先结账出队。这是队列的特征。

而堆栈则和队列相反，它是"先进后出"，怎么理解呢?基本所有的编辑器都有一个撤销功能，就是按 Ctrl+Z。当你写了一段文字，第一次按 Ctrl+Z，消失的是你最后写的文字，第二次按 Ctrl+Z，同样消失的是当前编辑器内最后写的文字。这就是一个堆栈结构的应用例子。

好，介绍完概念我们来看一下代码如何实现这两种数据结构，这篇文章我们采用最简单方式——通过 Python 原生的数据类型列表来实现。上篇文章，我们介绍了链表，通过链表我们同样可以实现堆栈和队列，感兴趣的朋友不妨尝试一下。

队列

首先，我们来定义一个队列类：

```
1
2
3
class Queue():
  def __init__(self):
  self.__list = list()
```

接下来，我们给队列类添加一些方法：

•判断队列是否为空

```
1
2
def isEmpty(self):
return self.__list == []
```

入队

```
1
2
def push(self, data):
  self.__list.append(data)
```

•出队

```
def pop(self):
if self.isEmpty():
  return False
return self.__list.pop(0)

```

•定义 len()函数和 print()操作类方法

```

def __len__(self):
return len(self.__list)
def __str__(self):
if self.isEmpty():
  return''
return' '.join([str(x) for x in self.__list])

```

OK，到这里，一个最简单的队列就实现啦，自己实例化一个队列测试一下吧

下面我们来看堆栈

堆栈

堆栈的实现和队列类似，同样有入栈和出栈操作，我们直接上代码：

```
class Stack():
  def __init__(self):
  self.__list =list()

  def isEmpty(self):
  return self.__list ==[]

  def push(self, data):
  self.__list.append(data)
```

```
8
9       def pop(self):
10      if self.isEmpty():
11        return False
        return self.__list.pop()
12
13      def __len__(self):
        return len(self.__list)
14
15      def __str__(self):
16      if self.isEmpty():
          return''
        return''.join([str(x) for x in self.__list])
```

方法 02：

1.栈(stack)定义

栈，又叫堆栈，是一种容器，可存入数据元素、访问元素、删除元素，它的特点在于**只能允许在容器的一端**（栈顶 top 或者栈底）进行**加入数据（push）和输出数据（pop）**的运算。

由于栈数据结构只允许在一端进行操作，因而按照**后进先出**==（LIFO, Last In First Out）==的原理运作。

2.python 代码实现

栈可以用顺序表实现，也可以用链表实现。如果采用顺序表,从栈底进行操作.采用链表,从栈顶操作.以上选择从时间复杂度角度分析.

分类	链表	顺序表
栈底-添加/删除	append()/O(1)	append()/O(n)
栈顶-添加/删除	add()/O(n)	add()/O(1)

```
"""栈  先进后出,后进先出"""
class Stack(object):
    """创建一个新栈"""
    def __init__(self):
        """初始化栈"""
        self.data = []
```

```
    def push(self,item):
        """添加一个新的元素 item 到栈顶"""
        return self.data.append(item)
    def pop(self):
        """弹出栈顶元素"""
        return self.data.pop()
    def peek(self):
        """返回栈顶元素"""
        return self.data[-1]
        # return self.data[len(self.data)-1]
    def is_empty(self):
        """判断栈是否为空"""
        return self.data == []
    def size(self):
        """返回栈的元素个数"""
        return len(self.data)

if __name__ == '__main__':
    stack = Stack()
    print(stack.is_empty())
    stack.push("1")
    stack.push("2")
    stack.push("3")
    print(stack.size())
    print(stack.is_empty())
    print(stack.peek()) # 到栈顶元素
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
```

以上内容仅是代表个人总结 若有错误之处,还请批评指正,欢迎大家一起学习!

238、Python 实现各种数据结构

python 基本数据结构栈 stack 和队列 queue

1，栈，后进先出，多用于反转

Python 里面实现栈，就是把 list 包装成一个类，再添加一些方法作为栈的基本操作。

栈的实现：

```
class Stack(object):
    #初始化栈为空列表
    def __init__(self):
        self.items = []    #self.__items = []可以把 items 变成私有属性
    #判断栈是不是为空
    def isEmpty(self):
        return len(self.items) ==0
    #返回栈顶的元素
    def peek(self):
        return self.items[-1]
    #返回栈的大小
    def size(self):
        return len(self.items)
    #给栈加入新元素
    def push(self,a):
        self.items.append(a)
    #删除栈顶的元素
    def pop(self):
        return self.items.pop()
```

栈应用实例：十进制转化为二进制

```
def divideBy2(decNumber):
    remstack = Stack()    #实例化一个栈，因为需要栈结构来存储数据，也是为了用到栈方法
    while decNumber > 0:
        rem = decNumber%2    #除二倒取余法，最后反转拼接所有的余数
        remstack.push(rem)    #余数依次放到栈中储存起来
        decNumber = decNumber // 2
    binstring = ''
    while not remstack.is_empty():
        binstring = binstring + str(remstack.pop())    #反序获取栈中的元素
    return binstring
```

```
print divideBy2(10)
```

2 队列 queue

队列实际上就是一个包装了的列表，从 list[0]添加新元素，用 pop()来获取，符合先进先出的规则。

```
class Queue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self,item):    #添加一个新元素 item 到队列的开头，这叫队尾
        self.items.insert(0,item)
    def dequeue(self):    #减去一个最后一个元素，这叫队首
        return self.items.pop()
    def size(self):
        return len(self.items)
    def show(self):    #自主添加的，好跟踪查看队列里有啥
        return self.items
```

队列应用实例：热土豆

```
#就是一队人围着圈报数，从 0 开始，报到 m 就出局，看谁活最久。
from pythonds.basic.queue import Queue
def HotPotato(namelist,num):
    simqueue =  Queue()
    for name in namelist:
        simqueue.enqueue(name)    #把先 namelist 添加到队列中去，ps：从下标是 0 的位置开始添加，整个插入完成以后序列就反过来了
    while simqueue.size()>1:
        for i in range(num):    #
            simqueue.enqueue(simqueue.dequeue())
            #从列表的末尾减去什么就把什么添加到列表开头
        simqueue.dequeue()    #哪个排最后哪个就是热土豆，直接出局
    return simqueue.dequeue()
print  HotPotato(['lili','jiajia','dahu','wangba','daqing','tamato','potato','hehe'],3)
```

3 双端队列有点类似于列表，不多赘述

4，链表

基本链表的实现：　　#链表是环环相扣形成的序列结构，每一环首先定义 self 变量，其次要标记下一个变量。所以访问的时候只能按照顺序来。

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None
    def getData(self):
        return self.data
    def getNext(self):
        return self.next
    def setData(self,newdata):
        self.data = newdata
    def setNext(self,newnext):
        self.next = newnext
```

无序链表：

```
class UnorderedList:
    def __init__(self):
        self.head = None    #表示链表的头部不引用任何内容
    def isEmpty(self):
        return self.head == None
    def add(self,item):    #链表中有两个属性，一个是本身，一个是 next，
        temp = Node(item)    #这个设定了链表本身，也就是 data,无序链表也是由 node 构成的
        temp.setNext(self.head)    #这个设定了 next
        self.head = temp    #把链表的 data 参数设定为无序列表的头----head
    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count +1
            current = current.getNext()
        return count
    def search(self,item):
```

```
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()
            return found
    def remove(self,item):
        current = self.head
        previous=None
        found = False
        while not found:    #找到要删除的 item 以后会跳出循环，此时 current.getData()是要删除的项目
            if current.getData()==item:
                found=True
            else:
                previous=current
                current=current.getNext()
        if previous ==None:    #只有一种情况下，previous 会是 None，那就是要删除的是第一个，也就是想删除 self.head
            self.head=current.getNext()
        else:
            previous.setNext(current.getNext())    # 本来的指向是 previous.getData()--item(也就是 previous.getNext(),还是
current.getData())--current.getNext()
                                                   #要想删除 item，那就把 previous 的指向改成 current.getNext()，这样 item 就不能在原
来的链表中瞎掺和了
```

有序链表：

```
class OrderedList:
    def __init__(self):
        self.head = None
    def isEmpty(self):    #同无序列表
        return self.head == None
    def show(self):
        current = self.head
        while current != None:
```

```
            print current.getData()
            current = current.getNext()
    def __iter__(self):
        current = self.head
        while current != None:
            yield current.getData()
            current = current.getNext()
    def size(self):    #同无序列表
        current = self.head
        count = 0
        while current != None:
            count +=1
            current =current.getNext()
        return count
    def search(self,item):    #默认从小到大排列的链表
        current = self.head
        found = False
        stop = False
        while current != None and not found and not stop:
            if current.getData() == item:
                found = True
            else:
                if current.getData() > item:
                    stop = True
                else:
                    current = current.getNext()
        return found
    def add(self,item):
        current = self.head
        previous = None
        stop = False
        while current != None and not stop:    #有一个以上元素的情况
            if current.getData() > item:
                stop = True
```

```
            else:
                previous = current
                current = current.getNext()     #不用担心元素添加到最后的情况，因为链表中自带 None 封住了两头
        temp = Node(item)
        if previous == None:     #添加到链表头的情况
            temp.setNext(self.head)
            self.head=temp
        else:
            temp.setNext(current)
            previous.setNext(temp)
    def remove(self, item):
        current = self.head
        previous = None
        found = False
        while not found:
            # 迭代每一项，得到要删除的那个，并且通过赋值前一个执行删除
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if previous == None:
            # 如果删除的是第一项，那么把第二项变成第一项，否则给 previous 赋值
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

239、in 时间复杂度是多少，为什么?

python in 的时间复杂度:
list: O(n)
dic/set: O(1)

Python 中的成员资格(membership)检查运算“in”，在列表(list)中遍历成员，时间复杂度为 O(N)；在字典(dict)中， 时间复杂度为 O(N)，测试结果如下：

```
In [1]: list1 = list(range(100000))

In [2]: dict1 = dict(zip(list1,list1))

In [3]: %timeit 49879 in list1
513 µs ± 15.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [4]: %timeit 49879 in dict1
54.7 ns ± 2.61 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [5]:
```

10000倍的性能差异!

10000倍的性能差异

10000 倍的性能差异
把下面的程序中的"in"操作的列表实现

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        length = len(nums)
        for i in range(length-1):
            if target - nums[i] in nums[i+1:]:
                return i, nums[i+1:].index(target - nums[i]) + i + 1
```

改为“in”操作的字典实现：

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        length = len(nums)
        target_dict = {}
        for i in range(length):
            num = target - nums[i]
            if num in target_dict:
                return target_dict[num],i
            target_dict[nums[i]] = i
```

执行用时，从 900ms，提升为 60ms

240、列表中有 n 个正整数范围在[0，1000]，进行排序;

排序，是许多编程语言中经常出现的问题。同样的，在 Python 中，列表如何是实现排序呢？（以下排序都是基于列表来实现）

一、使用 Python 内置函数进行排序

Python 中拥有内置函数实现排序，可以直接调用它们实现排序功能

Python 列表有一个内置的 list.sort() 方法可以直接修改列表。还有一个 sorted() 内置函数，它会从一个可迭代对象构建一个新的排序列表。

1.sort()函数：

```
list.sort(cmp=None, key=None, reverse=False)
```

其中参数的含义是：

cmp -- 可选参数，如果指定了该参数会使用该参数的方法进行排序。

key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。

reverse -- 排序规则，reverse = True 降序， reverse = False 升序（默认）。

默认输入列表就可以排序，例如：

```
list=[1,2,4,5,3]
list.sort()
print(list)
>>>[1,2,3,4,5]
```

2.sorted()函数：

```
sorted(iterable, cmp=None, key=None, reverse=False)
```

其中：

iterable -- 可迭代对象。

cmp -- 比较的函数，这个具有两个参数，参数的值都是从可迭代对象中取出，此函数必须遵守的规则为，大于则返回 1，小于则返回-1，等于则返回 0。

key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。

reverse -- 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。

同样的，使用 sorted()函数可以对列表进行排序，例如：

```
list=[1,2,4,5,3]
print(sorted(list))
>>>[1,2,3,4,5]
```

sort()和 sorted()虽然相似，都可以实现排序功能，但是它们有很大的不同：

sort ()与 sorted()区别：

sort() 是应用在 list 上的方法，sorted() 可以对所有可迭代的对象进行排序操作。

list 的 sort() 方法返回的是对已经存在的列表进行操作，无返回值，而内建函数 sorted() 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

列表的翻转（reverse）、升序（sort）、降序（sorted），按长度排列的用法

```
list4 = [10,10,50,20,30,60,51,20,10,10]
print(list4)
list4.reverse()                                #翻转
print(list4)

list4.sort()
```

```
print(list4)                          #升序排列，直接对表进行操作

list4.sort(reverse=True)
print(list4)                          #降序排列

list41 = [10,10,50,20,30,60,51,20,10,10]
print(sorted(list41))                  #升序排列，生成一个新表
print(list41)

print(sorted(list41,reverse=True)) #降序排列，从之前的列表中挑选出元素组成新的表
print(list41)

list43 = ["fddg","gfdggfg","f"]    #按照长度进行排序，生成新的列表
print(sorted(list43,key=len))
```

二、使用常用的排序算法进行排序

同其他高级函数一样，Python 也可以使用算法，利用一般语句进行排序。

1.冒泡排序

冒泡排序是最常见到的排序算法，也是很基础的一种排序算法。它的实现思想是：相邻的两个元素进行比较，然后把较大的元素放到后面（正向排序），在一轮比较完后最大的元素就放在了最后一个位置，像鱼儿在水中吐的气泡在上升的过程中不断变大，

```
def bubble_sort(list):
    count = len(list)
    for i in range(count):
        for j in range(i + 1, count):
            if list[i] > list[j]:
                list[i], list[j] = list[j], list[i]
    return list
```

2.选择排序

选择排序的思路是：第一轮的时候，所有的元素都和第一个元素进行比较，如果比第一个元素大，就和第一个元素进行交换，在这轮比较完后，就找到了最小的元素；第二轮的时候所有的元素都和第二个元素进行比较找出第二个位置的元素，以此类推。

```
def selection_sort(list):
    length = len(list)
    for i in range(length - 1, 0, -1):
        for j in range(i):
            if list[j] > list[i]:
                list[j], list[i] = list[i], list[j]
        return list
```

3.插入排序

插入排序的思想是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为 O(n^2)。 是稳定的排序方法。插入算法把要排序的数组分成两部分：第一部分包含了这个数组的所有元素，但将最后一个元素除外（让数组多一个空间才有插入的位置）， 而第二部分就只包含这一个元素（即待插入元素）。在第一部分排序完成后，再将这个最后元素插入到已排好序的第一部分中

```

def insert_sort(list):
    count = len(list)
    for i in range(1, count):
        key = list[i]
        j = i - 1
        while j >= 0:
            if list[j] > key:
                list[j + 1] = list[j]
                list[j] = key
            j -= 1
    return list

```

4.快速排序

快速排序的思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小， 然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

```
def quick_sort(list, left, right):
    if left >= right:
        return list
    key = lists[left]
    low = left
    high = right
    while left < right:
        while left < right and list[right] >= key:
            right -= 1
        lists[left] = lists[right]
        while left < right and list[left] <= key:
            left += 1
        list[right] = list[left]
    list[right] = key
    quick_sort(list, low, left - 1)
    quick_sort(list, left + 1, high)
    return list

lst1 = raw_input().split()    #调用函数
lst = [int(i) for i in lst1]
#lst = input()
quick_sort(lst,0,len(lst)-1)
for i in range(len(lst)):
    print lst[i],
```

5.希尔排序

希尔排序是插入排序的一种。也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。 该方法因 DL．Shell 于 1959 年提出而得名。 希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少， 每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止。

```
def shell_sort(list):
    count = len(list)
```

```
    step = 2
    group = count / step
    while group > 0:
        for i in range(group):
            j = i + group
            while j < count:
                k = j - group
                key = list[j]
                while k >= 0:
                    if list[k] > key:
                        list[k + group] = list[k]
                        list[k] = key
                    k -= group
                j += group
        group /= step
    return list
```

以上就是本文的全部内容，希望对大家的学习有所帮助，也希望大家多多支持脚本之家。

241、面向对象编程中有组合和继承的方法实现新的类

Python（面向对象编程——2 继承、派生、组合、抽象类）

继承与派生

```
'''
继承：属于
组合：包含
一、
在 OOP 程序设计中，当我们定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）。
继承有什么好处？最大的好处是子类获得了父类的全部功能。
继承：是基于抽象的结果，通过编程语言去实现它，肯定是先经历抽象这个过程，才能通过继承的方式去表达出抽象的结构。
二、
组合指的是，在一个类中以另外一个类的对象作为数据属性，称为类的组合
三、
接口的特征:
```

```
* 1)是一组功能的集合,而不是一个功能
* 2)接口的功能用于交互,所有的功能都是 public,即别的对象可操作
* 3)接口只定义函数,但不涉及函数实现
* 4)这些功能是相关的,都是某个类相关的功能。
接口提取了一群类共同的函数，可以把接口当做一个函数的集合。然后让子类去实现接口中的函数。
这么做的意义在于归一化，什么叫归一化，就是只要是基于同一个接口实现的类，那么所有的这些类产生的对象在使用时，从用法上来说都一样。
归一化，让使用者无需关心对象的类是什么，只需要的知道这些对象都具备某些功能就可以了，这极大地降低了使用者的使用难度。
四、
抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化
如果说类是从一堆对象中抽取相同的内容而来的，那么抽象类就是从一堆类中抽取相同的内容而来的，内容包括数据属性和函数属性。
'''
class Animal:                           #人属于动物
    def __init__(self,name,age):
        self.name=name
        self.age=age
        print('Animal.----init----')
        print(self.__dict__)
    def walk(self):
        print(self.name,'wlaking')

class Date_b:                           #人有生日
    def __init__(self,year,mon,day):
        self.year=year
        self.mon=mon
        self.day=day
    def p_birth(self):
        print('出生于%s 年，%s 月,%s 日' %(self.year,self.mon,self.day))

class People(Animal):         #继承
    def __init__(self,name,age,sex,hobbies,*args) :       #这里也可以把 年月日列出来传，那样写的时候 pycharm 也有提示，会比较好，这
里偷懒了
        Animal.__init__(self,name,age)
        self.sex=sex
```

```
        self.hobbies=hobbies
        self.birth=Date_b(*args)        #组合
        print('People--init--')
        # print(self.__dict__)
    def walk(self):             #道理上来说，这里可以不用类的原名字，可以任意命名，如何解决————抽象类（起到接口作用）（必须定义某几个函数的功能）
        Animal.walk(self)
        print('两脚兽 walking：名字：%s，年龄：%s，性别：%s，爱好：%s' %(self.name,self.age,self.sex,self.hobbies))   #派生其实就是子类
        print('年月：<%s><%s><%s>' %(self.birth.year,self.birth.mon,self.birth.day))     #直接继承的内容，和组合上的内容，调用方式不大一样

class Student(People):
    pass

alex=People('alex',10,'mail','anyway','1666','06','66')
alex.walk()
print('-----------------------------')
alex.birth.p_birth()

#-----------------抽象类----------------------

import abc
class File_personal(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def read(self):
        pass

class Txt(File_personal):
    def read(self):
        print('reading')
    def lll(self):
        print('lalala')
```

```
a=Txt()
a.read()
a.lll()

'''
另一种组合
'''
class Animal:                          #人属于动物
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def walk(self):
        print(self.name,'wlaking')

class Date_b:                          #人有生日
    def __init__(self,year,mon,day):
        self.year=year
        self.mon=mon
        self.day=day
    def p_birth(self):
        print('出生于%s 年，%s 月,%s 日' %(self.year,self.mon,self.day))

class People(Animal):          #继承
    def __init__(self,name,age,sex,hobbies) :
        Animal.__init__(self,name,age)
        self.sex=sex
        self.hobbies=hobbies
    def walk(self):                #道理上来说，这里可以不用类的原名字，可以任意命名，如何解决————抽象类（起到接口作用）（必须定义某几个函数的功能）
        Animal.walk(self)
        print('两脚兽 walking：名字：%s，年龄：%s，性别：%s，爱好：%s' %(self.name,self.age,self.sex,self.hobbies))    #派生
```

```
class Student(People):
    pass
alex=People('alex',10,'mail','anyway')
alex.walk()
alex.birth=Date_b('1666','06','66')          #之后再定制属性
alex.birth.p_birth()
```

1 什么是继承

继承是一种创建新类的方式，在 python 中，新建的类可以继承一个或多个父类，父类又可称为基类或超类，新建的类称为派生类或子类

python 中类的继承分为：单继承和多继承

```
class ParentClass1: #定义父类
    pass

class ParentClass2: #定义父类
    pass

class SubClass1(ParentClass1): #单继承，基类是 ParentClass1，派生类是 SubClass
    pass

class SubClass2(ParentClass1,ParentClass2): #python 支持多继承，用逗号分隔开多个继承的类
    pass
```

查看继承

```
>>> SubClass1.__bases__ #__base__只查看从左到右继承的第一个子类，__bases__则是查看所有继承的父类
(<class '__main__.ParentClass1'>,)
>>> SubClass2.__bases__
(<class '__main__.ParentClass1'>, <class '__main__.ParentClass2'>)
```

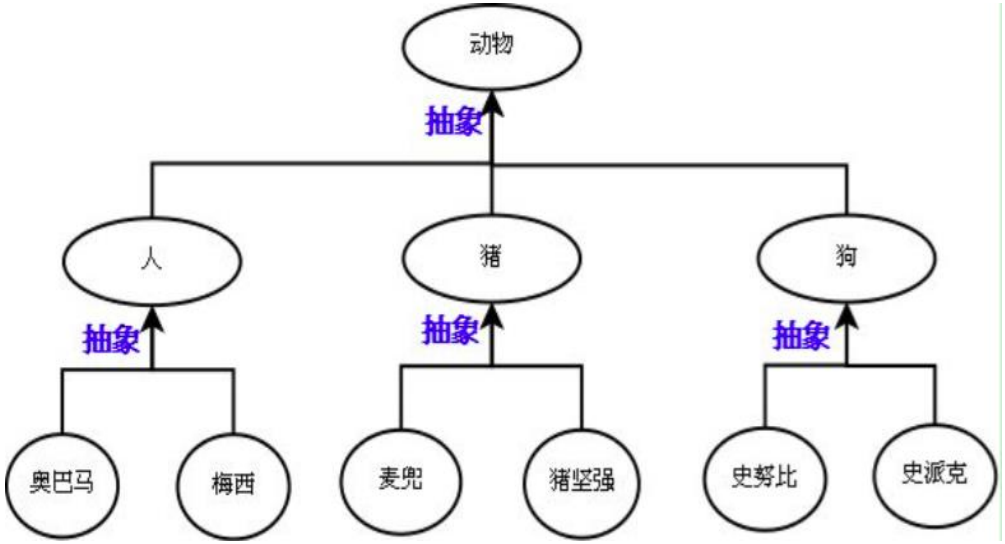
提示：如果没有指定基类，python 的类会默认继承 object 类，object 是所有 python 类的基类，它提供了一些常见方法（如__str__）的实现。

```
>>> ParentClass1.__bases__
(<class 'object'>,)
>>> ParentClass2.__bases__
(<class 'object'>,)
```

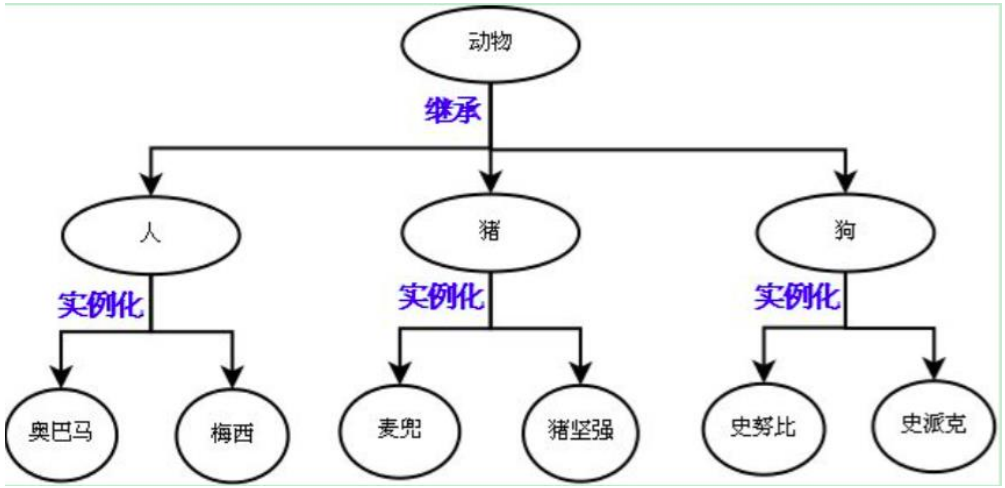
2 继承与抽象（先抽象再继承）

抽象即抽取类似或者说比较像的部分。

抽象分成两个层次：
1.将奥巴马和梅西这俩对象比较像的部分抽取成类；
2.将人，猪，狗这三个类比较像的部分抽取成父类。
抽象最主要的作用是划分类别（可以隔离关注点，降低复杂度）



继承：是基于抽象的结果，通过编程语言去实现它，肯定是先经历抽象这个过程，才能通过继承的方式去表达出抽象的结构。
抽象只是分析和设计的过程中，一个动作或者说一种技巧，通过抽象可以得到类



3 继承与重用性

在开发程序的过程中，如果我们定义了一个类 A，然后又想新建立另外一个类 B，但是类 B 的大部分内容与类 A 的相同时
我们不可能从头开始写一个类 B，这就用到了类的继承的概念。
通过继承的方式新建类 B，让 B 继承 A，B 会'遗传'A 的所有属性(数据属性和函数属性)，实现代码重用

```
class Hero:
    def __init__(self,nickname,aggressivity,life_value):
        self.nickname=nickname
        self.aggressivity=aggressivity
        self.life_value=life_value

    def move_forward(self):
        print('%s move forward' %self.nickname)

    def move_backward(self):
        print('%s move backward' %self.nickname)

    def move_left(self):
        print('%s move forward' %self.nickname)

    def move_right(self):
        print('%s move forward' %self.nickname)

    def attack(self,enemy):
        enemy.life_value-=self.aggressivity
class Garen(Hero):
    pass

class Riven(Hero):
    pass

g1=Garen('草丛伦',100,300)
r1=Riven('锐雯雯',57,200)
```

```
print(g1.life_value)
r1.attack(g1)
print(g1.life_value)

'''
运行结果
300
243
'''
```

提示：用已经有的类建立一个新的类，这样就重用了已经有的软件中的一部分设置大部分，大大生了编程工作量，这就是常说的软件重用，不仅可以重用自己的类，也可以继承别人的，比如标准库，来定制新的数据类型，这样就是大大缩短了软件开发周期，对大型软件开发来说，意义重大.

注意：像 g1.life_value 之类的属性引用，会先从实例中找 life_value 然后去类中找，然后再去父类中找...直到最顶级的父类。

当然子类也可以添加自己新的属性或者在自己这里重新定义这些属性（不会影响到父类），需要注意的是，一旦重新定义了自己的属性且与父类重名，那么调用新增的属性时，就以自己为准了。

```
class Riven(Hero):
    camp='Noxus'
    def attack(self,enemy): #在自己这里定义新的 attack,不再使用父类的 attack,且不会影响父类
        print('from riven')
    def fly(self): #在自己这里定义新的
        print('%s is flying' %self.nickname)
```

在子类中，新建的重名的函数属性，在编辑函数内功能的时候，有可能需要重用父类中重名的那个函数功能，应该是用调用普通函数的方式，即：类名.func()，此时就与调用普通函数无异了，因此即便是 self 参数也要为其传值

```
class Riven(Hero):
    camp='Noxus'
    def __init__(self,nickname,aggressivity,life_value,skin):
        Hero.__init__(self,nickname,aggressivity,life_value) #调用父类功能
        self.skin=skin #新属性
    def attack(self,enemy): #在自己这里定义新的 attack,不再使用父类的 attack,且不会影响父类
        Hero.attack(self,enemy) #调用功能
        print('from riven')
    def fly(self): #在自己这里定义新的
        print('%s is flying' %self.nickname)
```

```
r1=Riven('锐雯雯',57,200,'比基尼')
r1.fly()
print(r1.skin)
'''
运行结果
锐雯雯 is flying
比基尼
'''
```

4 组合与重用性

软件重用的重要方式除了继承之外还有另外一种方式，即：组合

组合指的是，在一个类中以另外一个类的对象作为数据属性，称为类的组合

其实早在 3.5 小节中我们就体会了组合的用法，比如一个英雄有一个装备

```
>>> class Equip: #武器装备类
...     def fire(self):
...         print('release Fire skill')
...
>>> class Riven: #英雄 Riven 的类,一个英雄需要有装备,因而需要组合 Equip 类
...     camp='Noxus'
...     def __init__(self,nickname):
...         self.nickname=nickname
...         self.equip=Equip() #用 Equip 类产生一个装备,赋值给实例的 equip 属性
...
>>> r1=Riven('锐雯雯')
>>> r1.equip.fire() #可以使用组合的类产生的对象所持有的方法
release Fire skill
```

组合与继承都是有效地利用已有类的资源的重要方式。但是二者的概念和使用场景皆不同，

1.继承的方式

通过继承建立了派生类与基类之间的关系，它是一种'是'的关系，比如白马是马，人是动物。

当类之间有很多相同的功能，提取这些共同的功能做成基类，用继承比较好，比如教授是老师

```
>>> class Teacher:
...     def __init__(self,name,gender):
...         self.name=name
...         self.gender=gender
```

```
...     def teach(self):
...         print('teaching')
...
>>>
>>> class Professor(Teacher):
...     pass
...
>>> p1=Professor('egon','male')
>>> p1.teach()
teaching
```

2.组合的方式

用组合的方式建立了类与组合的类之间的关系，它是一种'有'的关系,比如教授有生日，教授教 python 课程

```
class BirthDate:
    def __init__(self,year,month,day):
        self.year=year
        self.month=month
        self.day=day

class Couse:
    def __init__(self,name,price,period):
        self.name=name
        self.price=price
        self.period=period

class Teacher:
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender
    def teach(self):
        print('teaching')
class Professor(Teacher):
    def __init__(self,name,gender,birth,course):
        Teacher.__init__(self,name,gender)
```

```
        self.birth=birth
        self.course=course

p1=Professor('egon','male',
             BirthDate('1995','1','27'),
             Couse('python','28000','4 months'))

print(p1.birth.year,p1.birth.month,p1.birth.day)
print(p1.course.name,p1.course.price,p1.course.period)
'''
运行结果:
1995 1 27
python 28000 4 months
'''
```

当类之间有显著不同，并且较小的类是较大的类所需要的组件时，用组合比较好

5 接口与归一化设计

1.什么是接口

继承有两种用途：

一：继承基类的方法，并且做出自己的改变或者扩展（代码重用）

二：声明某个子类兼容于某基类，定义一个接口类 Interface，接口类中定义了一些接口名（就是函数名）且并未实现接口的功能，子类继承接口类，并且实现接口中的功能

```
class Interface:#定义接口 Interface 类来模仿接口的概念，python 中压根就没有 interface 关键字来定义一个接口。
    def read(self): #定接口函数 read
        pass

    def write(self): #定义接口函数 write
        pass



class Txt(Interface): #文本，具体实现 read 和 write
    def read(self):
        print('文本数据的读取方法')
```

```
    def write(self):
        print('文本数据的读取方法')

class Sata(Interface): #磁盘，具体实现 read 和 write
    def read(self):
        print('硬盘数据的读取方法')

    def write(self):
        print('硬盘数据的读取方法')

class Process(Interface):
    def read(self):
        print('进程数据的读取方法')

    def write(self):
        print('进程数据的读取方法')
```

实践中，继承的第一种含义意义并不很大，甚至常常是有害的。因为它使得子类与基类出现强耦合。
继承的第二种含义非常重要。它又叫"接口继承"。
接口继承实质上是要求"做出一个良好的抽象，这个抽象规定了一个兼容接口，使得外部调用者无需关心具体细节，可一视同仁的处理实现了特定接口的所有对象"——这在程序设计上，叫做归一化。
归一化使得高层的外部使用者可以不加区分的处理所有接口兼容的对象集合——就好象 linux 的泛文件概念一样，所有东西都可以当文件处理，不必关心它是内存、磁盘、网络还是屏幕（当然，对底层设计者，当然也可以区分出"字符设备"和"块设备"，然后做出针对性的设计：细致到什么程度，视需求而定）。

在 python 中根本就没有一个叫做 interface 的关键字，上面的代码只是看起来像接口，其实并没有起到接口的作用，子类完全可以不用去实现接口 ，如果非要去模仿接口的概念，可以借助第三方模块：
http://pypi.python.org/pypi/zope.interface
twisted 的 twisted\internet\interface.py 里使用 zope.interface
文档 https://zopeinterface.readthedocs.io/en/latest/
设计模式：https://github.com/faif/python-patterns
2. 为何要用接口
接口提取了一群类共同的函数，可以把接口当做一个函数的集合。
然后让子类去实现接口中的函数。
这么做的意义在于归一化，什么叫归一化，就是只要是基于同一个接口实现的类，那么所有的这些类产生的对象在使用时，从用法上来说都一样。

归一化，让使用者无需关心对象的类是什么，只需要的知道这些对象都具备某些功能就可以了，这极大地降低了使用者的使用难度。
比如：我们定义一个动物接口，接口里定义了有跑、吃、呼吸等接口函数，这样老鼠的类去实现了该接口，松鼠的类也去实现了该接口，由二者分别产生一只老鼠和一只松鼠送到你面前，即便是你分别不到底哪只是什么鼠你肯定知道他俩都会跑，都会吃，都能呼吸。
再比如：我们有一个汽车接口，里面定义了汽车所有的功能，然后由本田汽车的类，奥迪汽车的类，大众汽车的类，他们都实现了汽车接口，这样就好办了，大家只需要学会了怎么开汽车，那么无论是本田，还是奥迪，还是大众我们都会开了，开的时候根本无需关心我开的是哪一类车，操作手法（函数调用）都一样

6 抽象类

1 什么是抽象类

与 java 一样，python 也有抽象类的概念但是同样需要借助模块实现，**抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化**

2 为什么要有抽象类

如果说**类是从**一堆**对象**中抽取相同的内容而来的，那么**抽象类**就**是从**一堆**类**中抽取相同的内容而来的，内容包括数据属性和函数属性。

比如我们有香蕉的类，有苹果的类，有桃子的类，从这些类抽取相同的内容就是水果这个抽象的类，你吃水果时，要么是吃一个具体的香蕉，要么是吃一个具体的桃子。。。。。。你永远无法吃到一个叫做水果的东西。

从设计角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的。

从实现角度来看，抽象类与普通类的不同之处在于：抽象类中只能有抽象方法（没有实现功能），该类不能被实例化，只能被继承，且子类必须实现抽象方法。这一点与接口有点类似，但其实是不同的，即将揭晓答案

3. 在 python 中实现抽象类

```
#_*_coding:utf-8_*_
__author__ = 'Linhaifeng'
#一切皆文件
import abc #利用 abc 模块实现抽象类

class All_file(metaclass=abc.ABCMeta):
    all_type='file'
    @abc.abstractmethod #定义抽象方法，无需实现功能
    def read(self):
        '子类必须定义读功能'
        pass

    @abc.abstractmethod #定义抽象方法，无需实现功能
    def write(self):
        '子类必须定义写功能'
        pass
```

```
# class Txt(All_file):
#     pass
#
# t1=Txt() #报错,子类没有定义抽象方法

class Txt(All_file): #子类继承抽象类，但是必须定义 read 和 write 方法
    def read(self):
        print('文本数据的读取方法')

    def write(self):
        print('文本数据的读取方法')

class Sata(All_file): #子类继承抽象类，但是必须定义 read 和 write 方法
    def read(self):
        print('硬盘数据的读取方法')

    def write(self):
        print('硬盘数据的读取方法')

class Process(All_file): #子类继承抽象类，但是必须定义 read 和 write 方法
    def read(self):
        print('进程数据的读取方法')

    def write(self):
        print('进程数据的读取方法')

wenbenwenjian=Txt()

yingpanwenjian=Sata()

jinchengwenjian=Process()

#这样大家都是被归一化了,也就是一切皆文件的思想
```

```
wenbenwenjian.read()
yingpanwenjian.write()
jinchengwenjian.read()

print(wenbenwenjian.all_type)
print(yingpanwenjian.all_type)
print(jinchengwenjian.all_type)
```

4. 抽象类与接口

抽象类的本质还是类，指的是一组类的相似性，包括数据属性（如 all_type）和函数属性（如 read、write），而接口只强调函数属性的相似性。

抽象类是一个介于类和接口直接的一个概念，同时具备类和接口的部分特性，可以用来实现归一化设计

大数据

242、找出 1G 的文件中高频词

1.题目描述

有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，**内存限制大小是 1M**，要求返回频数最高的 100 个词

2.思考过程

（1）参见我的其他大数据面试题博文。此处 1G 文件远远大于 1M 内存，**分治法**，先 hash 映射把大文件分成很多个小文件，具体操作如下：读文件中，对于每个词 x，取 hash(x)%5000，然后按照该值存到 5000 个小文件(记为 f0,f1,...,f4999)中，这样每个文件大概是 200k 左右**（每个相同的词一定被映射到了同一文件中）**

（2）对于每个文件 fi，都用 hash_map 做词和出现频率的统计，取出频率大的前 100 个词（怎么取？topK 问题，建立一个 100 个节点的最小堆），把这 100 个词和出现频率再单独存入一个文件

（3）根据上述处理，我们又得到了 5000 个文件，**归并文件取出 top100（Top K 问题，比较最大的前 100 个频数）**

堆排序找 Top K

借助堆这个数据结构，找出 Top K，时间复杂度为 N ‘logK。即借助堆结构，我们可以在 log 量级的时间内查找和调整/移动。因此，维护一个 K(该题目中是 10)大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比。所以，我们最终的时间复杂度是：O（N） + N' * O（logK），（N 为 1000 万，N’ 为 300 万）。

思路总结:

要解决该问题首先要进行分类，把重复出现的 IP 都放到一个文件里面，一共分成 100 份，这可以通过把 IP 对 100 取模得到，具体方法如把 IP 中的点转化为整型 long 型变量，这样取模为 0,1,2...99 的 IP 都分到一个文件了，但是要考虑一个问题，如果某个文件的 IP 取余之后还是特别多无法放入内存中，可以再对这一类 IP 做一次取模，直到每个小文件足够载入内存为止。这个分类很关键，如果是随便分成 100 份，相同的 IP 被分在了不同的文件中，接下来再对每个文件统计次数并做归并，这个思路就没有意义了，起不到“大而化小，各个击破，缩小规模，逐个解决”的效果了。

接下来把每个小文件载入内存，建立哈希表将每个 IP 作为关键字映射为出现次数，这个哈希表建好之后也得先写入硬盘，因为内存就那么多，一共要统计 100 个文件。

在统计完 100 个文件之后，我再建立一个小顶堆，大小为 100，把建立好并存在硬盘哈希表载入内存，逐个对出现次数排序，挑出出现次数最多的 100 个，由于**次数直接和 IP 是对应的**，找出最多的次数也就找出了相应的 IP。

方法总结

01、分而治之，进行哈希取余；
02、使用 HashMap 统计频数；
03、求解**最大**的 TopN 个，用**小顶堆**；求解**最小**的 TopN 个，用**大顶堆。**

243、一个大约有一万行的文本文件统计高频词

1、建立 Trie 树，记录每颗树的出现次数，O(n*le); le:平均查找长度
2、维护一个 10 的小顶堆，O(n*lg10);
3、总复杂度： O(n*le) + O(n*lg10);

244、怎么在海量数据中找出重复次数最多的一个?

简答：方案 1：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

问题一：

怎么在海量数据中找出重复次数最多的一个

算法思想：

方案 1：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。
然后找出上一步求出的数据中重复次数最多的一个就是所求（如下）。

问题二：

网站日志中记录了用户的 IP，找出访问次数最多的 IP。

算法思想：

IP 地址最多有 2^32=4G 种取值可能，所以不能完全加载到内存中。
可以考虑分而治之的策略；

map

按照 IP 地址的 hash(IP)%1024 值，将海量日志存储到 1024 个小文件中，每个小文件最多包含 4M 个 IP 地址。

reduce

对于每个小文件，可以构建一个 IP 作为 key，出现次数作为 value 的 hash_map，并记录当前出现次数最多的 1 个 IP 地址。

有了 1024 个小文件中的出现次数最多的 IP，我们就可以轻松得到总体上出现次数最多的 IP。

同样的问题：

假设有 1kw 个身份证号，以及他们对应的数据。身份证号可能重复，要求找出出现次数最多的身份证号。

补充问题：

如果是要找出前 k 个最大的呢？

类似问题：

有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M。返回频数最高的 100 个词。

算法思想：

方案：顺序读文件中，对于每个词 x，取 hash(x)%5000，然后按照该值存到 5000 个小文件（记为 x0,x1,…x4999）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过 1M。

(第一步结束后，相同内容的词在同一个文件中，且文件比较小)

对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用 trie 树/hash_map 等），并取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆），并把 100 个词及相应的频率存入文件，这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并（类似与归并排序）的过程了。

类似问题：

有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

算法思想：

还是典型的 TOP K 算法，解决方案如下：

方案 1： 顺序读取 10 个文件，按照 hash(query)%10 的结果将 query 写入到另外 10 个文件（记为）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。 找一台内存在 2G 左右的机器，依次对用 hash_map(query, query_count)来统计每个 query 出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的 query 和对应的 query_cout 输出到文件中。这样得到了 10 个排好序的文件（记为 b0,b1,b2,...,b9）。对这 10 个文件进行归并排序（内排序与外排序相结合）。

方案 2： 一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案 3： 与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

245、判断数据是否在大量数据中

题目描述：

在 2.5 亿个整数中判断一个数是否存在，注意，内存不足以容纳 2.5 亿个整数。

分析解答：

方法一：分治法

对于大数据相关的算法题，分治法是一个非常好的方法。针对这一题来说，主要思路为：可以根据实际可用内存的情况，确定一个 Hash 函数，比如：hash(value)%1000，通过这个 Hash 函数可以把这 2.5 亿个数字划分到 1000 个文件中去（a1，a2……，a1000），然后再对待查找的数字使用同

样的 Hash 函数求出 Hash 值，假设计算出的 Hash 值为 i，如果这个数存在，那么它一定在文件 ai 中。通过这种方法就可以把题目转化为文件 ai 中是否存在这个数。那么接下来的求解过程中可以选用的思路计较多，有：
（1）由于划分后的文件比较小了，就可以直接装载到内存中去，可以把文件中所有的数字都保存到 hash_set 中，然后判断待查找的数字是否存在。
（2）如果这个文件中的数字占用的空间还是太大，那么可以用 1 相同的方法把这个文件继续划分为更小的文件，然后确定待查找的数字可能存在的文件，然后在相应的文件中继续查找。
方法二：位图法
对于这一类判断数字是否存在、判断数字是否重复的问题，位图法是一种非常高效的方法。以 32 位整型为例，它可以表示数字的个数为 2^32.可以申请一个位图，让每个整数对应的位图中的一个 bit，这样 2^32 个数需要的位图的大小为 512MB。具体实现的思路为：申请一个 512MB 的位图，并把所有的位都初始化为 0；接着遍历所有的整数，对遍历到的数字，把相应的位置上的 bit 设置为 1.最后判断待查找的数对应的位图上的值是多少，如果是 0，那么表示这个数字不存在，如果是 1，那么表示这个数字存在。

246、4G 内存怎么读取一个 5G 的数据?

方法一
可以通过生成器，分多次读取，每次读取数量相对少的数据（比如 500MB）进行处理，处理结束后
再读取后面的 500MB 的数据。
方法二
可以通过 linux 命令 split 切割成小文件，然后再对数据进行处理，此方法效率比较高。可以按照行
数切割，可以按照文件大小切割。

247、海量日志数据，提取出某日访问百度次数最多的那个 IP。

首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。注意到 IP 是 32 位的，最多有个 2^32 个 IP。同样可以采用映射的方法，比如模 1000，把整个大文件映射为 1000 个小文件，再找出每个小文中出现频率最大的 IP（可以采用 hash_map 进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。

248、搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。
典型的 Top K 算法，还是在这篇文章里头有所阐述。 文中，给出的最终算法是：第一步、先对这批海量数据预处理，在 O（N）的时间内用 Hash 表完成排序；然后，第二步、借助堆这个数据结构，找出 Top K，时间复杂度为 N‘logK。 即，借助堆结构，我们可以在 log 量级的时间内查找和调整/移动。因此，维护一个 K(该题目中是 10)大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比所以，我们最终的时间复杂度是：O（N）+ N’*O（logK），（N 为 1000 万，N’为 300 万）。ok，更多，详情，请参考原文。
或者：采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的最小推来对出现频率进行排序。

251、 给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url?

方案 1：可以估计每个文件安的大小为 5G×64=320G，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。
遍历文件 a，对每个 url 求取 hash(url)%1000，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为 a0,a1,…,a999）中。这样每个小文件的大约为 300M。

遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 小文件（记为 b0,b1,…,b999）。这样处理后，所有可能相同的 url 都在对应的小文件（a0vsb0,a1vsb1,…,a999vsb999）中，不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。
求每对小文件中相同的 url 时，可以把其中一个小文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。
方案 2：如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。
Bloom filter 日后会在本 BLOG 内详细阐述。

252、在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数。

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。所描完事后，查看 bitmap，把对应位是 01 的整数输出即可。
方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

253、腾讯面试题：给 40 亿个不重复的 unsigned int 的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 40 亿个数当中?

与上第 6 题类似，我的第一反应时快速排序+二分查找。以下是其它更好的方法： 方案 1：oo，申请 512M 的内存，一个 bit 位代表一个 unsigned int 值。读入 40 亿个数，设置相应的 bit 位，读入要查询的数，查看相应 bit 位是否为 1，为 1 表示存在，为 0 表示不存在。
dizengrong： 方案 2：这个问题在《编程珠玑》里有很好的描述，大家可以参考下面的思路，探讨一下：又因为 2^32 为 40 亿多，所以给定一个数可能在，也可能不在其中；这里我们把 40 亿个数中的每一个用 32 位的二进制来表示假设这 40 亿个数开始放在一个文件中。
然后将这 40 亿个数分成两类: 1.最高位为 0 2.最高位为 1 并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20 亿，而另一个>=20 亿（这相当于折半了）；与要查找的数的最高位比较并接着进入相应的文件再查找
再然后把这个文件为又分成两类: 1.次最高位为 0 2.次最高位为 1
并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10 亿，而另一个>=10 亿（这相当于折半了）； 与要查找的数的次最高位比较并接着进入相应的文件再查找。 … 以此类推，就可以找到了,而且时间复杂度为 O(logn)，方案 2 完。
附：这里，再简单介绍下，位图方法： 使用位图法判断整形数组是否存在重复 判断集合中存在重复是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。
位图法比较适合于这种情况，它的做法是按照集合中最大元素 max 创建一个长度为 max+1 的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上 1，如遇到 5 就给新数组的第六个元素置 1，这样下次再遇到 5 想置位时发现新数组的第六个元素已经是 1 了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时置零其后置一的做法类似于位图的处理方法故称位图法。它的运算次数最坏的情况为 2N。如果已知数组的最大值即能事先给新数组定长的话效率还能提高一倍。

255、上千万或上亿数据（有重复），统计其中出现次数最多的钱 N 个数据。

方案 1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用 hash_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前 N 个出现次数最多的数据了，可以用第 2 题提到的堆机制完成。

256、一个文

计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析。

方案 1：这题是考虑时间效率。用 trie 树统计每个词出现的次数，时间复杂度是 O(n*le)（le 表示单词的平准长度）。然后是找出出现最频繁的前 10 个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 O(n*lg10)。所以总的时间复杂度，是 O(n*le)与 O(n*lg10)中较大的哪一个。

附、100w 个数中找出最大的 100 个数。

方案 1：在前面的题中，我们已经提到了，用一个含 100 个元素的最小堆完成。复杂度为 O(100w*lg100)。

方案 2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比 100 多的时候，采用传统排序算法排序，取前 100 个。复杂度为 O(100w*100)。

方案 3：采用局部淘汰法。选取前 100 个元素，并排序，记为序列 L。然后一次扫描剩余的元素 x，与排好序的 100 个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把 x 利用插入排序的思想，插入到序列 L 中。依次循环，知道扫描了所有的元素。复杂度为 O(100w*100)。

第二部分、十个海量数据处理方法大总结

ok，看了上面这么多的面试题，是否有点头晕。是的，需要一个总结。接下来，本文将简单总结下一些处理海量数据问题的常见方法。

257、本文将简单总结下一些处理海量数据问题的常见方法。

下面的方法全部来自 http://hi.baidu.com/yanxionglu/blog/博客，对海量数据的处理方法进行了一个一般性的总结，当然这些方法可能并不能完全覆盖所有的问题，但是这样的一些方法也基本可以处理绝大多数遇到的问题。下面的一些问题基本直接来源于公司的面试笔试题目，方法不一定最优，如果你有更好的处理方法，欢迎讨论。

一、Bloom filter

适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集

基本原理及要点：

对于原理来说很简单，位数组+k 个独立 hash 函数。将 hash 函数对应的值的位数组置 1，查找时如果发现所有 hash 函数对应位都是 1 说明存在，很明显这个过程并不保证查找的结果是 100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是 counting Bloom filter，用一个 counter 数组代替位数组，就可以支持删除了。

还有一个比较重要的问题，如何根据输入元素个数 n，确定位数组 m 的大小及 hash 函数个数。当 hash 函数个数 k=(ln2)*(m/n)时错误率最小。在错误率不大于 E 的情况下，m 至少要等于 n*lg(1/E)才能表示任意 n 个元素的集合。但 m 还应该更大些，因为还要保证 bit 数组里至少一半为 0，则 m 应该>=nlg(1/E)*lge 大概就是 nlg(1/E)1.44 倍(lg 表示以 2 为底的对数)。

举个例子我们假设错误率为 0.01，则此时 m 应大概是 n 的 13 倍。这样 k 大概是 8 个。

注意这里 m 与 n 的单位不同，m 是 bit 为单位，而 n 则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多 bit 的。所以使用 bloom filter 内存上通常都是节省的。

扩展：

Bloom filter 将集合中的元素映射到位数组中，用 k（k 为哈希函数个数）个映射位是否全 1 表示元素在不在这个集合中。Counting bloom filter（CBF）将位数组中的每一位扩展为一个 counter，从而支持了元素的删除操作。Spectral Bloom Filter（SBF）将其与集合元素的出现次数关联。SBF 采用 counter 中的最小值来近似表示元素的出现频率。

问题实例：给你 A,B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4G，让你找出 A,B 文件共同的 URL。如果是三个乃至 n 个文件呢？
根据这个问题我们来计算下内存的占用，4G=2^32 大概是 40 亿*8 大概是 340 亿，n=50 亿，如果按出错率 0.01 算需要的大概是 650 亿个 bit。现在可用的是 340 亿，相差并不多，这样可能会使出错率上升些。另外如果这些 urlip 是一一对应的，就可以转换成 ip，则大大简单了。
二、Hashing
适用范围：快速查找，删除的基本数据结构，通常需要总数据量可以放入内存
基本原理及要点：
hash 函数选择，针对字符串，整数，排列，具体相应的 hash 方法。
碰撞处理，一种是 open hashing，也称为拉链法；另一种就是 closed hashing，也称开地址法，opened addressing。
扩展：
d-left hashing 中的 d 是多个的意思，我们先简化这个问题，看一看 2-left hashing。2-left hashing 指的是将一个哈希表分成长度相等的两半，分别叫做 T1 和 T2，给 T1 和 T2 分别配备一个哈希函数，h1 和 h2。在存储一个新的 key 时，同时用两个哈希函数进行计算，得出两个地址 h1[key]和 h2[key]。这时需要检查 T1 中的 h1[key]位置和 T2 中的 h2[key]位置，哪一个位置已经存储的（有碰撞的）key 比较多，然后将新 key 存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个 key，就把新 key 存储在左边的 T1 子表中，2-left 也由此而来。在查找一个 key 时，必须进行两次 hash，同时查找两个位置。
问题实例：
1).海量日志数据，提取出某日访问百度次数最多的那个 IP。
IP 的数目还是有限的，最多 2^32 个，所以可以考虑使用 hash 将 ip 直接存入内存，然后进行统计。
三、bit-map
适用范围：可进行数据的快速查找，判重，删除，一般来说数据范围是 int 的 10 倍以下
基本原理及要点：使用 bit 数组来表示某些元素是否存在，比如 8 位电话号码
扩展：bloom filter 可以看做是对 bit-map 的扩展
问题实例：
1)已知某个文件内包含一些电话号码，每个号码为 8 位数字，统计不同号码的个数。
8 位最多 99 999 999，大概需要 99m 个 bit，大概 10 几 m 字节的内存即可。
2)2.5 亿个整数中找出不重复的整数的个数，内存空间不足以容纳这 2.5 亿个整数。
将 bit-map 扩展一下，用 2bit 表示一个数即可，0 表示未出现，1 表示出现一次，2 表示出现 2 次及以上。或者我们不用 2bit 来进行表示，我们用两个 bit-map 即可模拟实现这个 2bit-map。
四、堆
适用范围：海量数据前 n 大，并且 n 比较小，堆可以放入内存
基本原理及要点：最大堆求前 n 小，最小堆求前 n 大。方法，比如求前 n 小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元素。这样最后得到的 n 个元素就是最小的 n 个。适合大数据量，求前 n 小，n 的大小比较小的情况，这样可以扫描一遍即可得到所有的前 n 元素，效率很高。
扩展：双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。

问题实例：
1)100w 个数中找最大的前 100 个数。
用一个 100 个元素大小的最小堆即可。
五、双层桶划分----其实本质上就是【分而治之】的思想，重在分的技巧上！
适用范围：第 k 大，中位数，不重复或重复的数字
基本原理及要点：因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子。
扩展：
问题实例：
1).2.5 亿个整数中找出不重复的整数的个数，内存空间不足以容纳这 2.5 亿个整数。
有点像鸽巢原理，整数个数为 232,也就是，我们可以将这 232 个数，划分为 2^8 个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用 bitmap 就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。
2).5 亿个 int 找它们的中位数。
这个例子比上面那个更明显。首先我们将 int 划分为 2^16 个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。
实际上，如果不是 int 是 int64，我们可以经过 3 次这样的划分即可降低到可以接受的程度。即可以先将 int64 分成 224 个区域，然后确定区域的第几大数，在将该区域分成 220 个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有 2^20，就可以直接利用 direct addr table 进行统计了。
六、数据库索引
适用范围：大数据量的增删改查
基本原理及要点：利用数据的设计实现方法，对海量数据的增删改查进行处理。
七、倒排索引(Inverted index)
适用范围：搜索引擎，关键字查询
基本原理及要点：为何叫倒排索引？一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。
以英文为例，下面是要被索引的文本： T0 = “it is what it is” T1 = “what is it” T2 = “it is a banana”
我们就能得到下面的反向文件索引：
“a”: {2} “banana”: {2} “is”: {0, 1, 2} “it”: {0, 1, 2} “what”: {0, 1}
检索的条件"what","is"和"it"将对应集合的交集。
正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。
扩展：
问题实例：文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索。
八、外排序

适用范围：大数据的排序，去重
基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树
扩展：
问题实例：
1).有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 个字节，内存限制大小是 1M。返回频数最高的 100 个词。
这个数据具有很明显的特点，词的大小为 16 个字节，但是内存只有 1m 做 hash 有些不够，所以可以用来排序。内存可以当输入缓冲区使用。
九、trie 树
适用范围：数据量大，重复多，但是数据种类小可以放入内存
基本原理及要点：实现方式，节点孩子的表示方式
扩展：压缩实现。
问题实例：
1).有 10 个文件，每个文件 1G，每个文件的每一行都存放的是用户的 query，每个文件的 query 都可能重复。要你按照 query 的频度排序。
2).1000 万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现？
3).寻找热门查询：查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个，每个不超过 255 字节。
十、分布式处理 mapreduce
适用范围：数据量大，但是数据种类小可以放入内存
基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。
扩展：
问题实例：
1).The canonical example application of MapReduce is a process to count the appearances ofeach different word in a set of documents:
2).海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。
3).一共有 N 个机器，每个机器上有 N 个数。每个机器最多存 O(N)个数并对它们操作。如何找到 N^2 个数的中数(median)？
经典问题分析
上千万 or 亿数据（有重复），统计其中出现次数最多的前 N 个数据,分两种情况：可一次读入内存，不可一次读入。
可用思路：trie 树+堆，数据库索引，划分子集分别统计，hash，分布式计算，近似统计，外排序
所谓的是否能一次读入内存，实际上应该指去除重复后的数据量。如果去重后数据可以放入内存，我们可以为数据建立字典，比如通过 map，hashmap，trie，然后直接进行统计即可。当然在更新每条数据的出现次数的时候，我们可以利用一个堆来维护出现次数最多的前 N 个数据，当然这样导致维护次数增加，不如完全统计后在求前 N 大效率高。
如果数据无法放入内存。一方面我们可以考虑上面的字典方法能否被改进以适应这种情形，可以做的改变就是将字典存放到硬盘上，而不是内存，这可以参考数据库的存储方法。
当然还有更好的方法，就是可以采用分布式计算，基本上就是 map-reduce 过程，首先可以根据数据值或者把数据 hash(md5)后的值，将数据按照范围划分到不同的机子，最好可以让数据划分后可以一次读入内存，这样不同的机子负责处理各种的数值范围，实际上就是 map。得到结果后，各个机子只需拿出各自的出现次数最多的前 N 个数据，然后汇总，选出所有的数据中出现次数最多的前 N 个数据，这实际上就是 reduce 过程。

实际上可能想直接将数据均分到不同的机子上进行处理，这样是无法得到正确的解的。因为一个数据可能被均分到不同的机子上，而另一个则可能完全聚集到一个机子上，同时还可能存在具有相同数目的数据。比如我们要找出现次数最多的前 100 个，我们将 1000 万的数据分布到 10 台机器上，找到每台出现次数最多的前 100 个，归并之后这样不能保证找到真正的第 100 个，因为比如出现次数最多的第 100 个可能有 1 万个，但是它被分到了 10 台机子，这样在每台上只有 1 千个，假设这些机子排名在 1000 个之前的那些都是单独分布在一台机子上的，比如有 1001 个，这样本来具有 1 万个的这个就会被淘汰，即使我们让每台机子选出出现次数最多的 1000 个再归并，仍然会出错，因为可能存在大量个数为 1001 个的发生聚集。因此不能将数据随便均分到不同机子上，而是要根据 hash 后的值将它们映射到不同的机子上处理，让不同的机器处理一个数值范围。
而外排序的方法会消耗大量的 IO，效率不会很高。而上面的分布式方法，也可以用于单机版本，也就是将总的数据根据值的范围，划分成多个不同的子文件，然后逐个处理。处理完毕之后再对这些单词的及其出现频率进行一个归并。实际上就可以利用一个外排序的归并过程。
另外还可以考虑近似计算，也就是我们可以通过结合自然语言属性，只将那些真正实际中出现最多的那些词作为一个字典，使得这个规模可以放入内存。