

数据结构

数据结构

- 基本数据结构类型

 - 栈

 - 栈的操作

 - 栈的代码实现

 - 队列

 - 队列的操作

 - 队列的代码实现

 - 队列的应用

 - 热土豆游戏

 - 双端队列

 - 双端队列的操作

 - 双端队列的代码实现

 - 双端队列的应用

 - “回文词”判定

 - 无序列表列表(链表)

 - 无序列表列表的操作

 - 无序列表（链表）的实现

 - 节点

 - 无序列表（链表）的代码实现

 - 有序列表

 - 有序列表的操作

 - 关于数字大小有序列表的代码实现

- 递归

 - 递归三大定律

 - 实例

 - 字符串的几个常用切片函数

- 动态规划

 - 硬币找零问题

 - 背包问题

 - 单词最小编辑距离问题

- 搜索

 - 顺序搜索

 - 二分法搜索

 - 散列

 - 散列函数

- 排序

 - 冒泡排序

 - 选择排序

基本数据结构类型

栈，队列，双端队列，和列表，这四种数据集合的项的由添加或删除的方式整合在一起。当添加一个项目时，它就被放在这样一个位置：在之前存在的项与后来要加入的项之间。像这样的数据集合常被称为**线性数据**

栈

定义 一个栈（有时称“叠加栈”）是一个项的有序集合。添加项和移除项都发生在同一“端”。这一端通常被称为“顶”。另一端的顶部被称为“底”。**特征是**后进先出

栈的操作

Stack()	创建一个新的空栈。它不需要参数，并返回一个空栈。
Push(item)将新项添加到堆栈的顶部。它需要参数item 并且没有返回值。
pop()	从栈顶删除项目。它不需要参数,返回item。栈被修改。
peek()	返回栈顶的项，不删除它。它不需要参数。堆栈不被修改。
isEmpty()	测试看栈是否为空。它不需要参数， 返回一个布尔值。
size()	返回栈的项目数。它不需要参数， 返回一个整数。

栈的代码实现

```
class Stack:

    def __init__(self):
        self.item=[];

    def Push(self,val):
        self.item.append(val)

    def pop(self):
        try:
            return self.item.pop()
        except:
            return None

    def peek(self):
        try:
            return self.item[-1]
        except:
            return None

    def isEmpty(self):
        return len(self.item)==0

    def size(self):
        return len(self.item)
```

队列

定义 队列（Queue）是一系列有顺序的元素的集合，新元素的加入在队列的一端，这一端叫做“队尾”（rear），已有元素的移除发生在队列的另一端，叫做“队首”（front），**特点是先进先出**

队列的操作

Queue()	创建一个空队列对象，无需参数，返回空的队列；
enqueue(item)	将数据项添加到队尾，无返回值；
dequeue()	从队首移除数据项，无需参数，返回值为队首数据项；
isEmpty()	测试是否为空队列，无需参数，返回值为布尔值；
size()	返回队列中的数据项的个数，无需参数。

队列的代码实现

```
class Queue:
    def __init__(self):
        self.item = []

    def enqueue(self, val):
        self.item.insert(0, val)

    def dequeue(self):
        return self.pop()

    def isEmpty(self):
        return len(self.item) == 0

    def size(self):
        return len(self.item)
```

队列的应用

热土豆游戏

在这个游戏中小孩子们围成一个圆圈并以最快的速度接连传递物品，并在游戏的一个特定时刻停止传递，这时手中拿着物品的小孩就离开圆圈，游戏进行至只剩下一个小孩。

问题分析 这个游戏涉及以下几个问题：1.如何模拟圆圈；2.如何模拟时间走动

对于问题1 我们使用队列对圆圈进行模拟，圆圈并不是本质的，而是在时间走动时，可以出现一种循环,我们可以**通过不断的进站出站**来实现这个过程

```
from pythonds.basic.queue import Queue #引用pythonds中已经写好的队列类

from random import randint

def hotpotato(namelist): #热土豆游戏
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name) #所有的成员进入队列
```

```

m = 1
while simqueue.size()>1:#开始游戏

    number = randint(1, simqueue.size());#随机决定停止的位置
    for i in range(number):

        simqueue.enqueue(simqueue.dequeue())#从队首出，队尾进

    print('第%s轮淘汰的是%s'%(m, simqueue.dequeue()))#时间到，队首人员淘汰

    m = m+1

else:
    print('最后的胜利者是%s'%simqueue.dequeue())

namelist=('军哥','GP','老权','爸爸我','希特勒','蔡徐坤','李云龙','周杰伦')

hotpotato(namelist)

```

运行得到：

```

第1轮淘汰的是GP
第2轮淘汰的是周杰伦
第3轮淘汰的是老权
第4轮淘汰的是李云龙
第5轮淘汰的是爸爸我
第6轮淘汰的是蔡徐坤
第7轮淘汰的是希特勒
最后的胜利者是军哥

```

双端队列

双端队列（deque 或 double-ended queue）与队列类似，也是一系列元素的有序组合。其两端称为队首（front）和队尾（rear）**特点是**元素可以从两端插入，也可以从两端删，**拥有栈和队列各自拥有的所有功能**

双端队列的操作

Deque()	创建一个空双端队列，无参数，返回值为 Deque 对象。
addFront(item)	在队首插入一个元素，参数为待插入元素，无返回值。
addRear(item)	在队尾插入一个元素，参数为待插入元素，无返回值
removeFront()	在队首移除一个元素，无参数，返回值为该元素。双端队列
removeRear()	在队尾移除一个元素，无参数，返回值为该元素。双端队列会
isEmpty()	判断双端队列是否为空，无参数，返回布尔值。
size()	返回双端队列中数据项的个数，无参数，返回值为整型数值。

双端队列的代码实现

```
class Deque:
    def __init__(self):
        self.item=[]

    def addRear(self,item):
        self.item.insert(0,item)

    def addFront(self,item):
        self.item.append(item)

    def removeFront(self):
        if self.item:
            return self.item.pop()
        else:
            return print('The deque is empty')

    def removeRear(self):
        if self.item:
            return self.item.pop(0)
        else:
            return print('The deque is empty')

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

    def size(self):
        return len(self.item)
```

双端队列的应用

“回文词”判定

回文词指的是正读和反读都一样的词，如：radar、toot 和madam。我们想要编写一个算法来检查放入的字符串是否为回文词。

思路分析：我们在栈的介绍时，提到过栈可以用来反转顺序，而双端队列，具有‘顺序’，‘逆序’两种性质，所以我们可以通过队列来实现回文词的判断

代码实现

```
def palchecker(aString):
    d=Deque()

    for i in aString:#使词进队
        d.addRear(i)

    while d.size()>1:#进行判断
        if d.removeFront()==d.removeRear():
            pass
        else:
            return False
    else:
        return True
```

评：上述代码虽然可以实现，但是最好时在判断的过程中加入标记，使得代码有更好的维护性

```
def palchecker(aString):
    d=Deque()

    for i in aString:#使词进队
        d.addRear(i)

    stillEqual = True

    while d.size()>1:#进行判断
        if d.removeFront()==d.removeRear():
            pass
        else:
            stillEqual = False

    return stillEqual
```

无序列表列表(链表)

无序列表结构是一个由各个元素组成的集合，在其中的每个元素拥有一个不同于其它元素的相对位置，**无序列表，其实仍有序，只不过是成为了相对序**，举例来说，电影院里坐着一排人，你可以说，“喂，快看1排 6号的迪丽热巴！”，但是如果走在街上的话，比较合适的描述是“喂，快看前面黄毛古惑仔，旁边的如花！”

无序列表列表的操作

list()	创建一个新的空列表。它不需要参数，而返回一个空列表。
add(item)	将新项添加到列表，没有返回值。假设元素不在列表中。
remove(item)	从列表中删除元素。需要一个参数，并会修改列表。此处假设元素在列表中。
search(item)	搜索列表中的元素。需要一个参数，并返回一个布尔值。
isEmpty()	判断列表是否为空。不需要参数，并返回一个布尔值。
size()	返回列表的元素数。不需要参数，并返回一个整数。
append(item)	在列表末端添加一个新的元素。它需要一个参数，没有返回值
index(item)	此处假设该元素原本在列表中，返回元素在列表中的位置。它需要一个参数，并返回位置索引值。
insert(pos,item)	在指定的位置添加一个新元素。它需要两个参数，没有返回值
pop()	从列表末端移除一个元素并返回它。它不需要参数，返回一个元素。
pop(pos)	从指定的位置移除列表元素并返回它。它需要一个位置参数，并返回值

无序列表（链表）的实现

为了实现无序列表，我们将构建一个链表，每个项目的相对位置就可以通过以下简单的链接从一个项目到下一个来确定，链表实现的基本模块是**节点**

节点

节点必须**包含列表元素本身**。我们将这称为该节点的“数据区”（data field）。此外，每个节点必须保持到**下一个节点的引用**

节点代码实现

```
class Node:
    def __init__(self,initdate):
        self.data=initdate
        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
```

无序列表（链表）的代码实现

无序列表将由一个节点集合组

只要我们知道第一个节点的位置（包含第一项），在这之后的每个元素都可以通过以下链接找到下一个节点。为实现这个想法，UnorderedList 类**必须保持一个对第一节点的引用**

节点就像一个机器人有两个机械臂，左臂小，右臂大，大到和机器人的身体一样大，链表就相当于一系列机器人，这些机器人左手拿着val,右手要么空着，要么抓着另一个机器人

 微信截图_20190516185622

```
class UnorderedList:

    def __init__(self):
        self.head = None

    def add(self, item):
        temp=Node(item)
        temp.setNext(self.head)
        self.head=temp

    def append(self, item):
        last=self.head
        while last.getNext() != None:
            last=last.getNext()
        temp=Node(item)
        last.setNext(temp)

    def isEmpty(self):
        return self.head==None

    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 search_pos(self, item):
    current=self.head
    previous=None
    found=False
    while current!=None and not found:
        if current.getData()==item:
            found=True

        else:
            previous=current
            current=current.getNext()

    return current

def remove(self, item):
    current=self.head
    previous=None
    found=False
    while not found:
        if current.getData()==item:
            found = True

        else:

            previous=current
            current=current.getNext()

    if previous==None: #如果找到的节点在第一项
        self.head=current.getNext()
    else:
        previous.setNext(current.getNext())

def index(self, item):

    previous=None

    current=self.head

    while current!=None and current.getData()!=item:
        previous=current
        current=current.getNext()

    print("The last item of %s is %s"%(str(item), str(previous.getData())))

def insert(self, pos, item):

    temp_0=self.search_pos(pos)
    temp_1=Node(item)
    temp_1.setNext(temp_0.getNext())
    temp_0.setNext(temp_1)

def pop(self, pos=None):

```

```

current=self.head
previous=None
if pos==None:
    while current.getNext()!=None:
        previous=current
        current=current.getNext()
    if previous==None:
        self.head=None
    else:
        previous.setNext(None)
    return current.getData()
else:
    self.remove(pos)
    return pos

```

有序列表

有序列表的结构是一个数据的集合体，在集合体中，每个元素相对其他元素**有一个基于元素的某些基本性质的位置**，举例来说，一个班级排队，一开始大家都是随便排，只产生一种相对位置，比如说小花后面是狗蛋，此时队伍就是一个无序列表；如果此时老师要求按身高进行排队，就产生了一种关于属性的位置，此时我们就说队伍是一个有序列表

有序列表的操作

OrderedList()	创建一个新的空有序列表。它返回一个空有序列表并且不需要传递任何参数。
add(item)	在保持原有顺序的情况下向列表中添加一个新的元素，新的元素作为参数传递进函数无返回值
remove(item)	从列表中删除某个元素。欲删除的元素作为参数，并且会修改原列表
search(item)	在列表中搜索某个元素，被搜索元素作为参数，返回一个布尔值。
isEmpty()	测试列表是否为空，不需要输入参数并且其返回一个布尔值。
size():	返回列表中元素的数量。不需要参数，返回一个整数。
index(item)	返回元素在列表中的位置。需要被搜索的元素作为参数输入，返回此元素的索引
pop()	删除并返回列表中的最后一项。不需要参数，返回删除的元素。假设列表中至少有一个
pop(pos)	删除并返回索引pos 指定项。需要被删除元素的索引值作为参数，并且返回这个元素

关于数字大小有序列表的代码实现

在有序列表中除了add，search外，其他所有函数的实现都和无序列表相同

```

class OrderedList:

    def __init__(self):
        self.head = None

    def add(self,item):

```

```

current=self.head

previous=None

stop=False

while current!=None and not stop:
    if current.getData()>item:
        stop=True
    else:
        previous=current
        current=current.getNext()
temp=Node(item)
if previous==None:
    temp.setNext(self.head)
    self.head=temp
else:
    temp.setNext(current)
    previous.setNext(temp)

def search(self,item):
    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 print(self):

    current=self.head

    s=[]

    if current==None:
        return None
    else:
        while current!=None:
            s.append(current.getData())

            current=current.getNext()

```

```
return s
```

实例

```
w=OrderedList()

w.add(1)
w.add(19)
w.add(20)
w.add(17)
w.print()
```

运行得到：

```
[1, 17, 19, 20]
```

递归

定义 递归是一种解决问题的方法，它把一个问题分解为越来越小的子问题，直到问题的规模小到 可以被很简单直接解决

通常为了达到分解问题的效果，递归过程中要**引入一个调用自身的函数**

递归三大定律

- 递归算法必须有个基本**结束条件**
- 递归算法必须改变自己的状态并**向基本结束条件演进**
- 递归算法必须递归地**调用自身**

实例

对列表中的元素做加法

```
def list_sum(num_list):
    if len(num_list) == 1:
        return num_list[0] #1.结束条件
    else:
        return num_list[0] + list_sum(num_list[1:]) #3.调用自身并2.向基本结束条件演进
print(list_sum([1,3,5,7,9]))
```

计算一个整数的阶乘

```
def fact(n:int):
    if n>0:#结束条件
        return n*fact(n-1) #调用自身，并向基本结束条件演进
    else:
        return 1
```

整数转化成任意进制表示的字符串形式

```
def to_str(n,base=17):
    convert_string="0123456789ABCDEFG"

    if n<base:#结束条件
        return convert_string[n]
    else:
        return to_str(n//base,base)+convert_string[n%base] #调用自身，并向基本结束条件演进
```

反转字符串

```
def reverse(Str):

    if len(Str)>1:#结束条件

        return Str[-1]+reverse(Str[:-1]) #调用自身，并向基本结束条件演进
    else:
        return Str[0]
```

判断回文词

这一题目我们曾在双端队列中做过

```
def judge_rev(Str):

    judge=False

    if len(Str)>1:#结束条件
        if Str[0]==Str[-1]:

            return judge_rev(Str[1:-1])#调用自身，并向基本结束条件演进

        else:
            return judge

    else:
        judge=True

    return judge
```

注意：在调用函数本身后，最外层的数据一定要对调用结果有引用，否则容易出错

判断句子是否是构成回文词

一些优秀的回文也可以是短语,但是你需要在验证前删除空格和标点符号。例如:madam i'm adam 是个回文

```
def clearingstr(Str):
    Str=Str.lower()#将字母都换为小写
    if len(Str)>1:#结束条件

        if not Str[0].isalpha() and not Str[0].isdigit():
            Str=Str[1:]
            return clearingstr(Str) #调用自身,并向基本结束条件演进
        else:
            return Str[0]+clearingstr(Str[1:])#调用自身,并向基本结束条件演进
    else:
        if not Str[0].isalpha() and not Str[0].isdigit():
            return ''
        else:
            return Str
```

图形递归

```
import turtle
myTurtle = turtle.Turtle()

def drawSpiral(myTurtle, lineLen):

    if lineLen > 0:#结束条件
        myTurtle.forward(lineLen)#向前走lineLen个距离

        myTurtle.right(90)#向右旋转90度
        drawSpiral(myTurtle,lineLen-5)#调用自身,并向基本结束条件演进

drawSpiral(myTurtle,100)
```

画一棵分形树

```
import turtle
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)#向前走branchLen步
        t.right(20)#向右旋转20度
        tree(branchLen-15,t)#调用函数自身
        t.left(40)#向左旋转40度
        tree(branchLen-15,t)#调用函数自身
        t.right(20)#向右旋转20度
        t.backward(branchLen)#回退branchLen长度

def main():
    t = turtle.Turtle()
    t.left(90)
    t.up()#移动时不绘制图形
```

```
t.backward(100)
t.down()
t.color("green")
tree(75,t)

main()
```

评：通过上面画一个分形树我们知道，函数一旦调用自身的话，就直接调用到结束条件触发为止，当我们对tree的参数给的很大时，小海龟的行为变的很复杂，我们无法知道非常的通过程序的运行，了解其中的机制，此时需要我们化简模型，将模型化简到只递归一次或者两次的情况，来帮助我们理解。

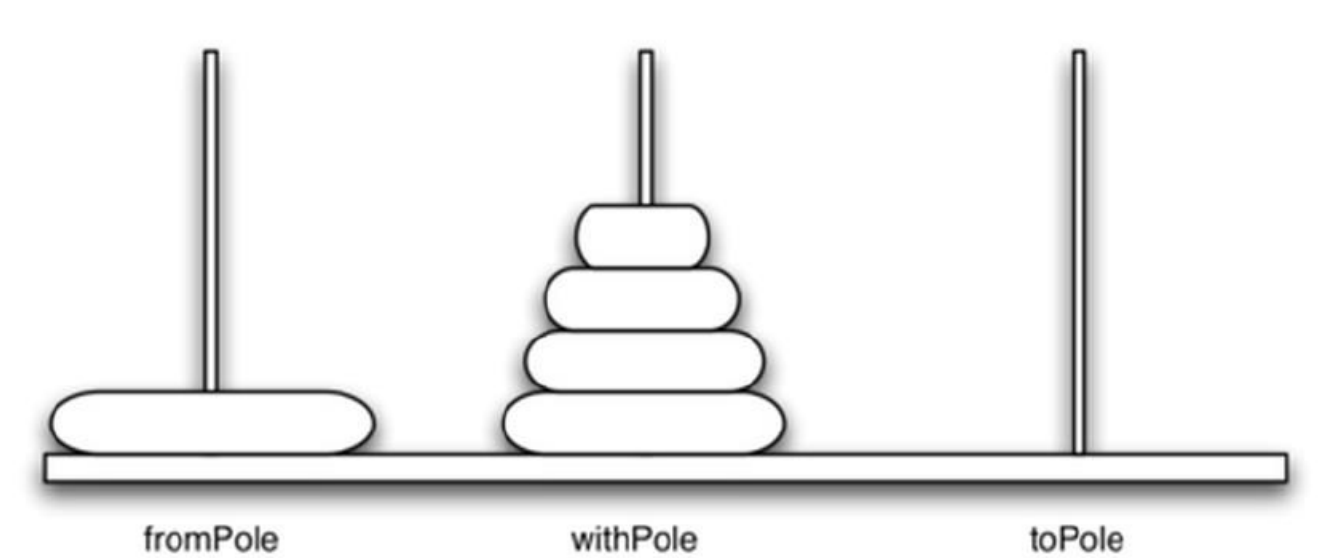
字符串的几个常用切片函数

函数名	语法	返回值
strip()	str.strip([chars]);	返回移除字符串头尾指定的字符生成的新字符串。
replace()	str.replace(old, new[, max])#old -- 将被替换的子字符串; new -- 新字符串，用于替换old子字符串; max -- 可选字符串, 替换不超过 max 次	替换后的新字符串
split()	str.split(str="", num=string.count(str))#str--分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。num--分割次数。默认为 -1, 即分隔所有。	返回分割后的字符串列表

河内塔问题(难)

河内塔问题是法国数学家爱德华·卢卡斯于1883年发现的。他受到一个关于印度教寺庙的传说 的启发，故事中这一问题交由年轻僧侣们解决。最开始，僧侣们得到**三根杆子**，64个**金圆盘堆叠在其中一根上**，每个圆盘比其下的小一点。僧侣们的任务是将64个圆盘**从一根杆上转移到另一根杆上**，但有两项重要的限制，一是他们**一次只能移动一个圆盘**，**是不能将大圆盘放在小圆盘之上**。

我们接下来用代码来实现整个过程



分析：开始时，所有的盘子都在fromPole上，我们要将所有的盘子移动到toPole上；不妨假设只有三个我们来观察整个过程，记三个盘子分别为1、2、3，柱子分别记为A,B,C

假设我们知道如何 $1,2: A \rightarrow B$ ，那么我们只需要 $3: A \rightarrow C$ ，然后利用 $1,2: A \rightarrow B$ 的方法，进行 $1,2: B \rightarrow C$

即可，但是！如果你不知道如何 $1,2: A \rightarrow B$ ，那么你只需要知道如何 $1: A \rightarrow C$ ，利用同样的方法就可以使得

$2: A \rightarrow B$ ，然后 $1: C \rightarrow B$ 即可（ps:这种思想才是递归法的精华所在，我们不是简单的找规律，而是不断的减小问题的规模）

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)

def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

动态规划

动态规划算法的基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，**列出各种可能的局部解**，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。**依次解决各子问题**，最后一个子问题就是初始问题的解。

硬币找零问题

对于任意给出的金额，要求用最少的硬币来找零。

我们已经学过递归算法，下面我们先递归算法来解决

分析：假设一个顾客投了1美元来购买37美分的物品。你用来找零的硬币的最小数量是多少？答案是六枚硬币：两个25美分，一个10美分，三个1美分。我们是怎么得到六个硬币这个答案的呢？首先我们要使用面值中**最大的硬币**（25美分），并且**尽可能多的使用它**，接着我们再使用下一个可使用的最大面值的硬币，也是尽可能多的使用。这种方法被称为贪心算法，因为我们试图尽可能快的解。（这里提一个小问题：为什么最新版人民币的币值，只有那几种？）

```
def recMC(coinlist,money):
    coins=0
    if money in coinlist:#结束条件
        coins=coins+1
        return coins
    else:
        s=[c for c in coinlist if c<=money]
        m=max(s)
        last=recMC(coinlist,money-m)#调用自身，并向基本结束条件演进
        coins=1+last
        return coins
```

但是贪心算法并不是总能给出最优解，如何coinlist=[1,5,10,21,25]；money=63；贪心算法会直接选择25作为第一次的找零对象

为此我们需要改进自己的算法

类似是在河内塔问题的分析,我们做如下的分析:

假设我们的coinlist=[1,5,10,21,25], money=N,假设我们已经知道了N-1,N-5,N-10,N-21,N-25的最少找零方法,那么我们只需要选择上述N-i(i in coinlist)最少的一个就好,但是如果不知道N-1的找零方法,但是我们知道N-1-1,N-1-5,N-1-10,N-1-21,N-1-25的方法的话,同样只要需从中选择最小的一个然后加一就好.....

```
def recMC(coinlist,money):
    coins=money
    if money in coinlist:#结束条件
        coins=coins+1
        return coins
    else:
        last=money
        for i in [c for c in coinlist if c<=money]:
            last=1+recMC(coinlist,money-i)#调用自身,并向结束条件演进
            if last<coins:#进行比较选择最小的方案
                coins=last
        return coins
```

上述算法虽然可以实现找零,但是效率太低了,距离来说coinlist=[1,5,10,21,25], money=12,上述算法会分析出11,7,2的找零方法,而为了得到7的找零方法,我们又需要得治2,6的找零方法,可是我们不是已经知道2的找零方法了吗?? 我们曾经在画一个**分形树**的例子中提到过,递归一旦开始,只能到一口气到结束条件,也就使得我们在分析7的找零方法时,2的找零方法还是要分析一次

减少我们的工作量的关键在于记住一些出现过的结果,这样就能避免重复计算我们已经知道的结果

```
def recDC(coinValueList,change,knownResults):
    minCoins = change
    if change in coinValueList:#结束条件
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:#结束条件
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recDC(coinValueList, change-i, knownResults)#调用自身并向结束条件演进
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins#更新记录
        return minCoins
print(recDC([1,5,10,25],63,[0]*64))
```

注:一定程度上,递归法的思想十分类似于归纳法的想法,比如上述硬币找零的思想就很类似于第二归纳法.

事实上,我们目前所采用的方法还不是动态规划,我们只是使用了一种叫做**“函数值缓存”**,或者一般称为**“缓存”**的方法改善了程序的性能.

动态规划的解决方法是从为1分钱找零的最优解开始,逐步递加上去,直到我们需要的找零钱数

这就保证了在算法的每一步过程中,我们已经知道了兑换任何更小数值的零钱时所需的硬币数量的最小值

```

def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
    for cents in range(change+1):
        coinCount = cents
        newCoin = 1
        for j in [c for c in coinValueList if c <= cents]:#计算从1到change逐一增加的找零钱数
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j]+1
                newCoin = j#记录使用的钱币
        minCoins[cents] = coinCount
        coinsUsed[cents] = newCoin
    return minCoins[change]

def printCoins(coinsUsed,change):
    coin = change
    while coin > 0:
        thisCoin = coinsUsed[coin]
        print(thisCoin)
        coin = coin - thisCoin

def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)
    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)
main()

```

评：提一个简单的问题：为什么这里我们没有使用递归？如果使用的递归从1到change 逐一的寻找最优解，我们将每次都把小于cents的最佳解，重新求解一遍，而这正是动态规划的优势所在。

背包问题

假设你是一个计算机科学家或是一个艺术小偷，闯入了一个艺术画廊。你身上只有一个背 包可以用来偷出宝物，这个背包只能装W英镑的艺术品，但你知道每一件艺术品的价值和它的重 量。运用动态规划写一个函数，来帮助你获得最多价值的宝物。你可以利用下面的例子来编写程 序：假设你的背包可以容纳的总重量为20，你有如下5件宝物：

item	weight	value
1	2	3
2	3	4
3	4	8
4	5	8
5	9	10

贪心算法：每次只拿最有价值的宝物，一直到背包放不下为止

```
def Artsteal(artlist,bag):
    s=[c for c in artlist if c[1]<=bag]
    value=0
    if s !=[]:
        w=(0,0,0)
        for i in s:
            if w[2]<=i[2]:
                w=i
        value=w[2]
        print(w)
        value=value+Artsteal(artlist,bag-w[1])
        return value
    else:
        return value

artlist=[(1,2,3),(2,3,4),(3,4,8),(4,5,8),(5,9,10)]
print(Artsteal(artlist,20))
```

运行得到：

```
(5, 9, 10)
(5, 9, 10)
(1, 2, 3)
23
```

动态规划：从背包容纳重量为1开始求最优解，一直找到bag重

```
def Artsteal(artlist,maxres,steallist,bag):
    for w in range(bag+1):
        s=[c for c in artlist if c[1]<=w]
        value=0
        use=0
        if s !=[]:
            for j in s:
                if maxres[w-j[1]]+j[2]>value:
                    value=maxres[w-j[1]]+j[2]
                    use=j
            maxres[w]=value
```

```

        steallist[w]=use
    return maxres[bag]

def stealed(steallist,bag):
    s=bag
    while s>1:
        stealed=steallist[s]
        print(stealed)
        s=s-stealed[1]

def main():
    bag=20
    artlist=[(1,2,3),(2,3,4),(3,4,8),(4,5,8),(5,9,10)]
    maxres=[0]*(bag+1)
    steallist=[(0,0,0)]*(bag+1)
    print(Artsteal(artlist,maxres,steallist,bag))
    stealed(steallist,bag)
main()

```

运行得到：

```

40
(3, 4, 8)
(3, 4, 8)
(3, 4, 8)
(3, 4, 8)
(3, 4, 8)

```

单词最小编辑距离问题

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

```

插入一个字符
删除一个字符
替换一个字符

```

```

def minDistance( word1, word2):
    n = len(word1)
    m = len(word2)

    # if one of the strings is empty
    if n * m == 0:
        return n + m

    # array to store the conversion history
    d = [ [0] * (m + 1) for _ in range(n + 1)]

    # init boundaries
    for i in range(n + 1):

```

```

    d[i][0] = i
    for j in range(m + 1):
        d[0][j] = j

    # DP compute
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            left = d[i - 1][j] + 1
            down = d[i][j - 1] + 1
            left_down = d[i - 1][j - 1]
            if word1[i - 1] != word2[j - 1]:
                left_down += 1
            d[i][j] = min(left, down, left_down)

    return d[n][m]

```

评：这次动态规划，我们从1维到1维，改为了2维的规划，通过观察注意到当我们获得 $d[i-1][j]$ ， $d[i][j-1]$ 和 $d[i-1][j-1]$ 的值之后就可以计算出 $d[i][j]$ 。

如果两个子串的最后一个字母相同， $word1[i] = word2[i]$ 的情况下：

$$\begin{aligned}
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1] - 1) \\
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1] - 1) \\
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1] - 1)
 \end{aligned}$$

否则， $word1[i] \neq word2[i]$ 我们将考虑替换最后一个字符使得他们相同：

$$\begin{aligned}
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1]) \\
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1]) \\
 D[i][j] &= 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1])
 \end{aligned}$$

否则， $word1[i] \neq word2[i]$ 我们将考虑替换最后一个字符使得他们相同：

总结：动态规划和递归不同，递归是先解决n的情况，再解决n-1的情况，而动态规划刚好相反，先解决n-1的情况，再解决n的情况。

搜索

搜索的算法过程就是在一些项**集合中找到一个特定的项**。搜索过程通常会根据特定项是否存在来给出回答**True或者False**。

顺序搜索

在Python列表，这些相对位置所对应的是单个项的索引值。由于这些索引值是有一定次序的，可以依次访问它们。

```
def sequentialSearch(alist, item):
    pos = 0#位置从0开始
    found = False#记录是否找到
    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos + 1
    return found
testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

我们寻找的目标项可能在任意位置，对列表的每一个位置，我们找到它的**概率是相等的**，从而顺序搜索的**复杂度是 $O(n)$** 。

二分法搜索

假设在一组有序的元素中进行搜索，二分搜索将从**中间项开始检测**

```
def binarySearch(alist,item):
    pos=0;
    found=False
    last=len(alist)-1#元素的位置从0开始计算
    while pos<=last and not found:
        mid=(pos+last)//2#找到中间位置
        if alist[mid]==item:#比较中间位置与item
            found=True
            return found
        else:
            if item<alist[mid]:
                last=mid-1
            else:
                pos=mid+1
    return found
```

在二分法搜索中我们最多分割 $\frac{n}{2^i}$ 次，其中 $i: 2^i \leq n < 2^{i+1}$ ，也就是说我们差不多**最多对比 i 次**，解出来后 $i = \log(n)$ ，从而**二分搜索的复杂度是 $O(\log(n))$** 。

在顺序搜索和二分搜索中我们用来搜索的对象都是具有相对位置的集合，也就是有序列表，但是面对毫无顺序的“散列”，我们又该如何搜索呢？或者说我们又应该如何使其变得有序呢？

散列

散列表也叫哈希表是一种**数据的集合**，其中的每个数据都通过**某种特定的方式进行存储以方便日后的查找**（也就是说对于本来无序的数据，通过某种映射使其变得有序）。散列表的**每一个位置叫做槽**，槽能够存放一个数据项，并以从0开始递增的整数命名。例如，第一个槽记为0，下一个记为1，再下一个记为2，并以此类推。

某个**数据项**与在散列表中存储它的**槽之间的映射叫做 散列函数**。

散列函数

如果一个散列函数可以将每一个数据项都**映射到不同的槽中**，那么这样的散列函数叫做 **完美散列函数**。

我们的**目标**是创建一个能够将冲突的**数量降到最小**，**计算方便**，并且最终将**数据项分配到散列表**中的散列函数
常用的方法有：

散列函数	详细说明
直接定址法	直接定址法是以关键字K本身或关键字加上某个数值常量C作为散列地址的方法。对应的散列函数h(K)为： $h(K)=K+C$ 若C为0，则散列地址就是关键字本身。这种方法计算最简单，并且没有冲突发生，若有冲突发生，则表明是关键字错误。它适应于关键字的分布基本连续的情况，若关键字分布不连续，空号较多，将造成存储空间的浪费。
除留余数法	除留余数法是用关键字K除以散列表长度m所得余数作为散列地址的方法。对应的散列函数h(K)为： $h(K)=k \% m$
数字分析法	数字分析法是取关键字中某些取值较分散的数字位作为散列地址的方法。它适合于所有关键字已知，并对关键字中每一位的取值分布情况作出了分析。例如，有一组关键字为(92317602，92326875，92739628，92343634，92706816，92774638，92381262，92394220)，通过分析可知，每个关键字从左到右的第1，2，3位和第6位取值较集中，不宜作散列地址。剩余的第4，5，7和8位取值较分散，可根据实际需要取其中的若干位作为散列地址。若取最后两位作为散列地址，则散列地址的集合为(2，75，28，34，16，38，62，20)。
平方取中法	平方取中法是取关键字平方的中间几位作为散列地址的方法，具体取多少位视实际要求而定。一个数平方后的中间几位和数的每一位都有关。从而可知，由平方取中法得到的散列地址同关键字的每一位都有关，使得散列地址具有较好的分散性。平方取中法适应于关键字中的每一位取值都不够分散或者较分散的位数小于散列地址所需要的位数的情况。
折叠法	折叠法是首先将关键字分割成位数相同的几段(最后一段的位数若不足应补0)，段的位数取决于散列地址的位数，由实际需要而定，然后将它们的叠加和(舍去最高位进位)作为散列地址的方法。例如一个关键字K=68242324，散列地址为3位，则将此关键字从左到右每三位一段进行划分，得到的三段为682，423和240，叠加和为682+423+240=345，此值就是存储关键字为68242324元素的散列地址。折叠法适应于关键字的位数较多，而所需的散列地址的位数又较少，同时关键字中每一位的取值又较集中的情况。

散列函数在运作时，常常并不是完美的，也就是会出现冲突的情况，对于冲突的情况，这里不在做详解，有兴趣的小伙伴可以自行搜索。

排序

冒泡排序

冒泡排序要对一个列表多次重复遍历项，并且交换顺序排错的项。每对列表实行一次遍历，就有一个最大项排在了正确的位置。（毫无疑问，通过每次的两两对比，每一次遍历都将会把最大值找到正确位置）

由于每一次遍历都会使下一个最大项归位，所需要遍历的总次数就是 $n-1$

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                alist[i],alist[i+1] = alist[i+1],alist[i]
alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

评：一般的编程语言，在交换两项的顺序时，都需要第三项来做临时的存储器，但是Python可以进行同时赋值。

容易得知，冒泡排序的复杂度是 $O(n^2)$

选择排序

选择排序提高了冒泡排序的性能，它每遍历一次列表只交换一次数据，即进行一次遍历时找到最大的项

```
def Selectsort(alist):
    l=len(alist)
    for i in range(l):#遍历次数
        pos=l-i-1
        maxpos=0
        for j in range(pos+1):#找出最大项
            if alist[maxpos]<=alist[j]:
                maxpos=j
        alist[pos],alist[maxpos]=alist[maxpos],alist[pos]#把最大项放到最后
    return print(alist)
```

显然，选择排序和冒泡排序有相同的复杂度是 $O(n^2)$