数据结构与算法（Python）-05：排序与搜索

陈斌 北京大学地球与空间科学学院 gischen@pku.edu.cn

目录

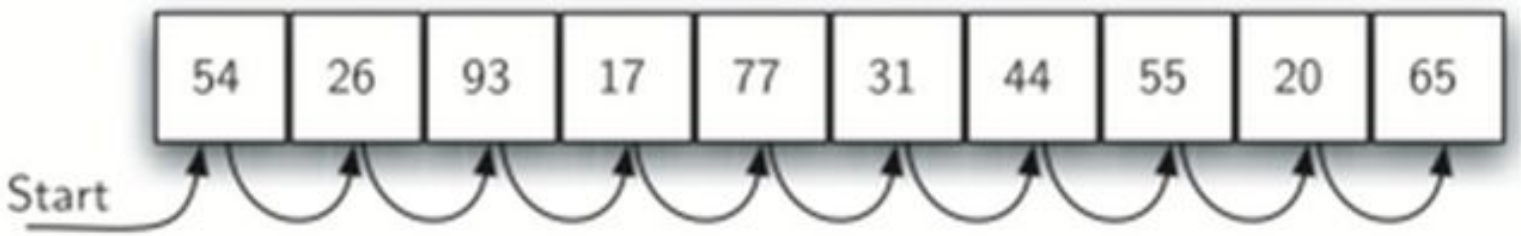
- 本章目标
- 顺序搜索
- 二分搜索
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序

本章目标

- 了解和实现顺序搜索和二分法搜索；
- 了解和实现选择排序、冒泡排序、归并排序、快速排序、插入排序和希尔排序；
- 了解用散列Hashing实现搜索的技术；
- 了解抽象数据类型：映射Map；
- 采用散列实现抽象数据类型Map。

顺序搜索Sequential Search

- 如果数据项保存在如列表这样的集合中，我们会称这些数据项具有线性或者顺序关系，在Python List中，这些数据项的存储位置称为下标（index），这些下标都是有序的整数，通过下标，我们就可以按照顺序来访问和查找数据项，这种技术就称为“顺序搜索”
- 要确定列表中是否存在需要查找的数据项，首先从列表的第1个数据项开始，按照下标增长的顺序，逐个比对数据项，如果到最后一个都未发现要查找的项，那么搜索失败。



顺序搜索：无序表搜索代码


```
def sequentialSearch(alist, item):
    pos = 0
    found = False

    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))
```

下标顺序增长

顺序搜索：算法分析

- **要对搜索算法进行分析，首先要确定其中的基本计算步骤。回顾第二章算法分析的要点，这种基本计算步骤必须要足够简单，并且在算法中反复执行**
- **在搜索算法中，这种基本计算步骤就是进行数据项的比对**

  当前数据项等于还是不等于要查找的数据项，比对的次数决定了算法复杂度
- **在顺序搜索算法中，为了保证是讨论的一般情形，需要假定列表中的数据项并没有按值排列顺序，而是随机放置在列表中的各个位置**

  换句话说，数据项在列表中各处出现的概率是相同的
- **数据项是否在列表中，比对次数是不一样的**

  如果数据项不在列表中，需要比对所有数据项才能得知，比对次数是n

  如果数据项在列表中，要比对的次数，其情况就较为复杂
  - 最好的情况，第1次比对就找到；最坏的情况，要n次比对

顺序搜索：算法分析

- **数据项在列表中，比对的一般情形如何？**

  因为数据项在列表中各个位置出现的概率是相同的；

  所以平均状况下，比对的次数是n/2；

- **所以，顺序搜索的算法复杂度是O(n)**

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item is not present	n	n	n

- **这里我们假定列表中的数据项是无序的，那么如果数据项排了序，顺序搜索算法的效率又如何呢？**

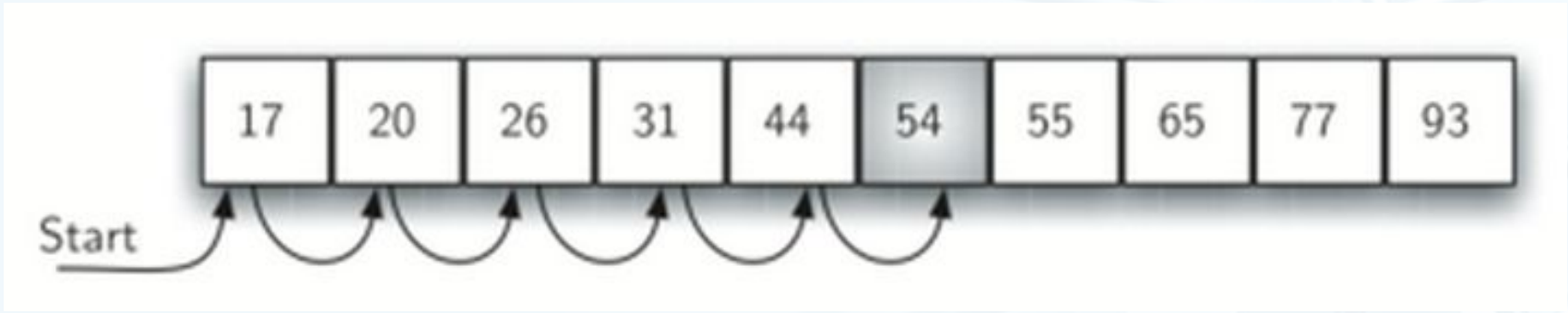
顺序搜索：算法分析

- **实际上，我们在第三章的有序表Search方法实现中介绍过顺序搜索**

  当数据项存在时，比对过程与无序表完全相同

  不同之处在于，如果数据项不存在，比对可以提前结束

  - 如下图中搜索数据项50，当看到54时，可知道后面不可能存在50，可以提前退出搜索



顺序搜索：有序表搜索代码


```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

提前退出

顺序搜索：算法分析

- 顺序搜索有序表的各种情况分析

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item is not present	1	n	n/2

- 实际上，就算法复杂度而言，仍然是O(n)
- 只是在数据项不存在的时候，有序表的搜索能节省一些比对次数，但并不改变其数量级。

二分搜索

- **那么对于有序表，有没有更好的搜索算法？**
- **在顺序搜索中，如果第1个数据项不匹配查找项的话，那最多还有n-1个待比对的数据项，那么有没有方法能利用有序表的特性，迅速缩小待比对数据项的范围呢？**
- **我们从列表中间开始比对！**

  如果列表中间的项匹配查找项，则搜索结束

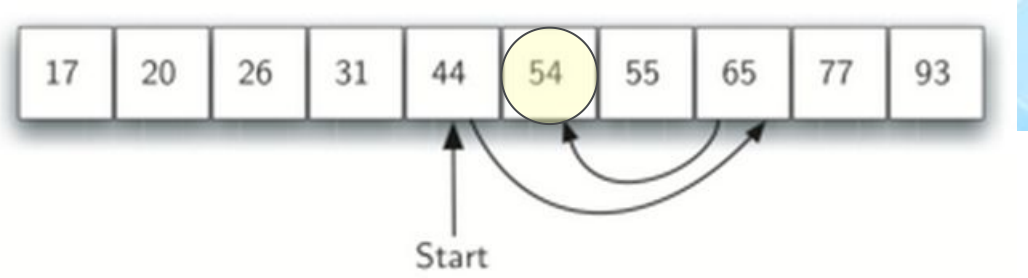
  如果不匹配，那么就有两种情况：

  - 列表中间项比查找项大，那么查找项只可能出现在前半部分
  - 列表中间项比查找项小，那么查找项只可能出现在后半部分

  无论如何，我们都会将比对范围缩小到原来的一半：n/2

- **继续采用上面的方法搜索**

  每次都会将比对范围缩小一半



二分搜索：代码


```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

中间项比对

缩小比对范围

二分搜索：分而治之（Divide & Conquer）

- **二分搜索算法实际上体现了解决问题的另一个典型策略：分而治之**

  将问题分为若干更小规模的部分

  通过解决每一个小规模部分问题，并将结果汇总得到原问题的解

- **显然，递归算法就是一种典型的分而治之策略，二分法也适合用递归算法来实现**

基本结束条件

缩小规模

调用自身

```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binarySearch(alist[:midpoint],item)
          else:
            return binarySearch(alist[midpoint+1:],item)
```

二分搜索：算法分析

- 由于二分搜索每次比对都将下一步的比对范围缩小一半，每次比对如下表所示：

Comparisons	Approximate Number of Items Left
1	n/2
2	n/4
3	n/8
…	
i	n/2^i

- 当比对次数足够多以后，比对范围内就会仅剩余1个数据项，无论这个数据项是否匹配查找项，比对最终都会结束，解下列方程：

  得到：$i=\log_2(n)$

$$\frac{n}{2^i} = 1$$

- 所以二分法搜索的算法复杂度是O(log n)

二分搜索：进一步的考虑

- **虽然我们根据比对的次数，得出二分搜索的复杂度O(log n)，但要注意到本算法中除了比对，还有一个因素需要注意到：**

  binarySearch(alist[:midpoint],item)

  这个递归调用使用了列表切片，而切片操作的复杂度是O(k)，这样会使整个算法的时间复杂度稍有增加；

  当然，我们采用切片是为了程序可读性更好，实际上也可以不切片，而只是传入起始和结束的索引值即可，这样就不会有切片的时间开销了。

- **另外，虽然二分搜索在时间复杂度上优于顺序搜索，但也要考虑到对数据项进行排序的开销**

  如果一次排序后可以进行多次搜索，那么排序的开销就可以摊薄

  但如果数据集经常变动，搜索次数相对较少，那么可能还是直接用无序表加上顺序搜索来得经济

- **所以，在算法选择的问题上，光看时间复杂度的优劣是不够的，还需要考虑到实际应用的情况。**

散列：Hashing

- **前面我们利用数据集中关于数据项之间排列关系的知识，来将搜索算法进行了提升。**

  如果数据项之间是按照大小排好序的话，就可以利用二分搜索来降低算法复杂度。

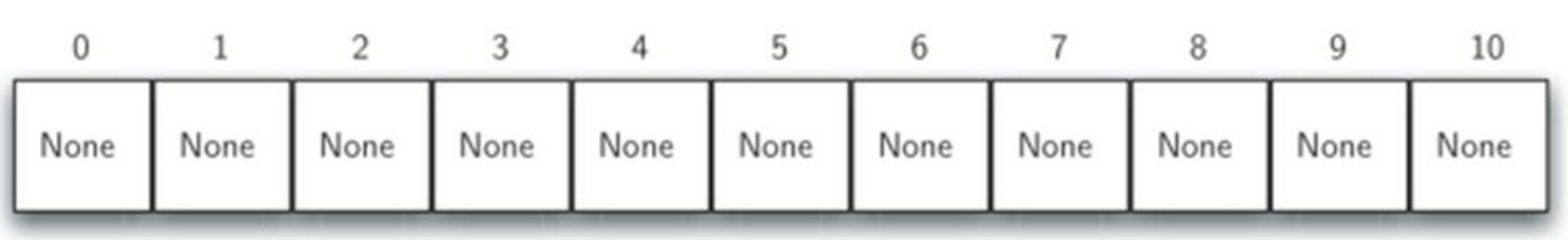
- **现在我们进一步来构造一个新的数据结构，能使得搜索算法的复杂度降到O(1)，这种概念称为“散列Hashing”**

- **能够使得搜索的次数降低到常数级别，我们对数据项所处的位置就必须有更多的先验知识。如果我们事先能知道要找的数据项应该出现在数据集中的什么位置，就可以直接到那个位置看看数据项是否存在即可。**

- **由数据项的值来确定其存放位置，如何能做到这一点呢？**

散列：基本概念

- 散列表（hash table，又称哈希表）是一种数据集，其中数据项的存储方式尤其有利于将来快速的搜索定位。散列表中的每一个存储位置，称为槽（slot），可以用来保存数据项，槽有一个名称。
- 例如：一个包含11个槽的散列表，槽的名称分别为0～10
- 在插入数据项之前，每个槽的值都是None，表示空槽
- 实现从数据项到存储槽的转换的，称为**散列函数**（hash function）
- 下面例子中，散列函数接受数据项作为参数，返回整数值0～10，表示数据项存储的槽号



散列：示例

- **为了将数据项保存到散列表中，我们设计第一个散列函数**

  数据项：54，26，93，17，77，31

- **有一种常用的散列方法是“求余数”，将数据项除以散列表的大小，得到的余数作为槽号。**

  实际上“求余数”方法会以不同形式出现在所有散列函数里，因为散列函数返回的槽号必须在散列表大小范围之内，所以一般会对散列表大小求余

- **本例中我们的散列函数是最简单的求余h(item)= item% 11**

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

散列：示例

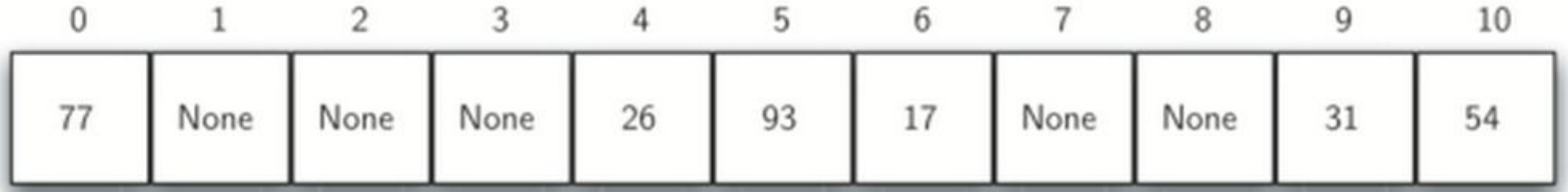
- **按照散列函数h(item)，为每个数据项计算出存放的位置之后，就可以将数据项存入相应的槽中，例子中的6个数据项插入后，占据了散列表11个槽中的6个。**

  槽被数据项占据的比例称为散列表的“负载因子”，这里负载因子为6/11

- **数据项都保存到散列表后，搜索就变得无比简单，要查找某个数据项是否存在于表中，我们只需要使用同一个散列函数，对查找项进行计算，测试下返回的槽号所对应的槽中是否有数据项即可**

  实现了O(1)时间复杂度的搜索算法。

- **不过，你可能也看出这个方案的问题所在，这组数据相当凑巧，各自占据了不同的槽，假如还要保存44，h(44)=0，它跟77被分配到同一个0#槽中，这种情况称为“冲突collision”，我们后面会讨论到这个问题的解决方案。**



完美散列函数：Perfect Hash Function

- **给定一组数据项，如果一个散列函数能把每个数据项映射到不同的槽中，那么这个散列函数就可以称为“完美散列函数”**
  对于固定的一组数据，总是能想办法设计出完美散列函数
- **但如果数据项经常性的变动，很难有一个系统性的方法来设计对应的完美散列函数。**
  当然，冲突也不是致命性的错误，我们会有办法处理的。
- **获得完美散列函数的一种方法是扩大散列表的容量，大到所有可能出现的数据项都能够占据不同的槽，但这种方法对于可能数据项范围过大的情况并不实用**
  假如我们要保存手机号（11位数字），完美散列函数得要求散列表具有百亿个槽！对于仅需要保存班级100名同学的手机号来说，浪费了太多空间
- **退而求其次的话，好的散列函数需要具备3个特性：冲突最少（近似完美）、计算难度低（额外开销小）、充分分散数据项（节约空间）**

完美散列函数的更多用途

- **除了用于在散列表中安排数据项的存储位置，散列技术还用在信息处理的很多领域。**
- **由于完美散列函数能够对任何不同的数据生成不同的散列值，如果把散列值当作数据的“指纹”或者“摘要”，这种特性被广泛应用在数据的一致性校验上**

  由任意长度的数据生成长度固定的“指纹”，还要求具备唯一性，这在数学上是无法做到的，但设计巧妙的“准完美”散列函数却能在实用范围内做到这一点。

- **作为一致性校验的数据“指纹”函数需要具备如下的特性**

  压缩性：任意长度的数据，得到的“指纹”长度是固定的；

  易计算性：从原数据计算“指纹”很容易；（从指纹计算原数据是不可能的）；

  抗修改性：对原数据的微小变动，都会引起“指纹”的大改变；

  抗冲突性：已知原数据和“指纹”，要找到相同指纹的数据（伪造）是非常困难的；同样，要找到两组不同的数据，使它们具有相同的指纹，也是非常困难的。

散列函数MD5/SHA

- **最著名的近似完美散列函数是MD5和SHA系列函数：MD5（Message Digest）将任何长度的数据变换为固定长为128位（16字节）的“摘要”**

  128位二进制已经是一个极为巨大的数字空间：据说是地球沙粒的数量

- **SHA（Secure Hash Algorithm）是另一组散列函数，SHA-0/SHA-1输出散列值160位（20字节），SHA-256/SHA-224分别输出256位、224位，SHA-512/SHA-384分别输出512位和384位**

  160位二进制相当于10的48次方，地球上水分子数量估计是47次方

  256位二进制相当于10的77方，已知宇宙所有基本粒子大约是72～87次方

- **虽然近年发现MD5/SHA-0/SHA-1三种散列函数能够以极特殊的情况来构造个别碰撞（散列冲突），但在实用中从未有实际的威胁。**

- **关于数量级的知识：**

  http://zh.wikipedia.org/wiki/%E6%95%B0%E9%87%8F%E7%BA%A7_(%E6%95%B0)

Python的散列函数库hashlib

- **Python自带MD5和SHA系列的散列函数库：hashlib**

  包括了md5/sha1/sha224/sha256/sha384/sha512等6种散列函数

```
>>> import hashlib
>>> hashlib.md5("hello world!").hexdigest()
'fc3ff98e8c6a0d3087d515c0473f8677'
>>> hashlib.sha1("hello world!").hexdigest()
'430ce34d020724ed75a196dfc2ad67c77772d169'
```

- **除了对单个字符串进行散列计算之外，还可以用update方法来对任意长的数据分部分来计算，这样不管多大的数据都不会有内存不足的问题。**

```
>>> import hashlib
>>> m= hashlib.md5()
>>> m.update("hello world!")
>>> m.update("this is part #2")
>>> m.update("this is part #3")
>>> m.hexdigest()
'a12edc8332947a3e02e5668c6484b93a'
>>>
```

散列函数MD5/SHA系列用于数据一致性校验

- **数据文件一致性判断**

  为每个文件计算其散列值，仅对比其散列值即可得知是否文件内容相同；

  用于网络文件下载完整性校验；

  用于文件分享系统：网盘中相同的文件（尤其是电影）可以无需存储多次。

- **加密形式保存密码**

  仅保存密码的散列值，用户输入密码后，计算散列值并比对；

  无需保存密码的明文即可判断用户是否输入了正确的密码。

- **防止文件篡改：原理同数据文件一致性判断**

  当然还有更多密码学机制来保护数据文件，防篡改，防抵赖，是电子商务的信息技术基础。

- **网络赌场应用**

  赌徒下注前，赌场将投骰的结果散列值公布，然后赌徒投注，开盘后，赌徒可以通过公布的结果和散列值对比，验证赌场是否作弊。

散列函数设计：折叠法Folding Method

- 折叠法设计散列函数的基本步骤是，将数据项按照位数分为若干段，再将几段数字相加，最后对散列表大小求余，得到散列值
- 例如，对电话号码62767255，可以两位两位分为4段（62、76、72、55），相加（62+76+72+55=265），散列表包括11个槽，那么就是265%11=1，所以h(62767255)=1
- 有时候折叠法还会包括一个**隔数反转**的步骤，比如（62、76、72、55）隔数反转为（62、67、72、55），再累加（62+67+72+55=256），对11求余（256%11=3），所以h'(62767255)=3
- 虽然隔数反转从理论上看来毫无必要，但这个步骤确实为折叠法得到散列函数提供了一种微调手段，以便更好符合上述的3个特性。

散列函数设计：平方取中法Mid-Square Method

- 平方取中法，首先将数据项做平方运算，然后取平方数的中间两位，再对散列表的大小求余
- 例如，对44进行散列，首先44*44=1936，然后取中间的93，对散列表大小11求余，93%11=5
- 下表是两种散列函数的对比

两个都是完美散列函数

分散度都很好

平方取中法计算量稍大

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

散列函数设计：非数项

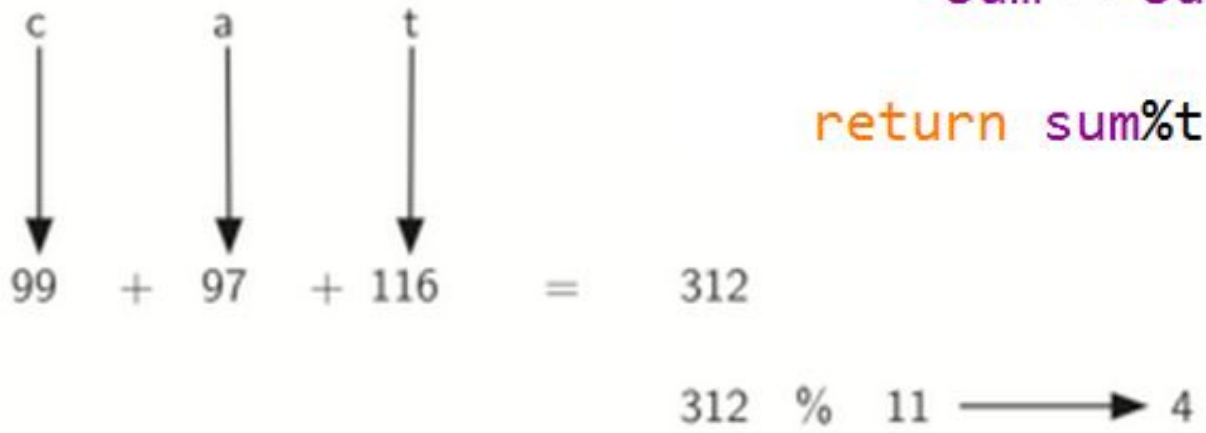
- 我们也可以对非数字的数据项（如字符串）进行散列，把字符串中的每个字符看作ASCII码整数即可

  如cat, ord('c')==99, ord('a')==96, ord('t')==116

- 再将这些整数累加，对散列表大小求余

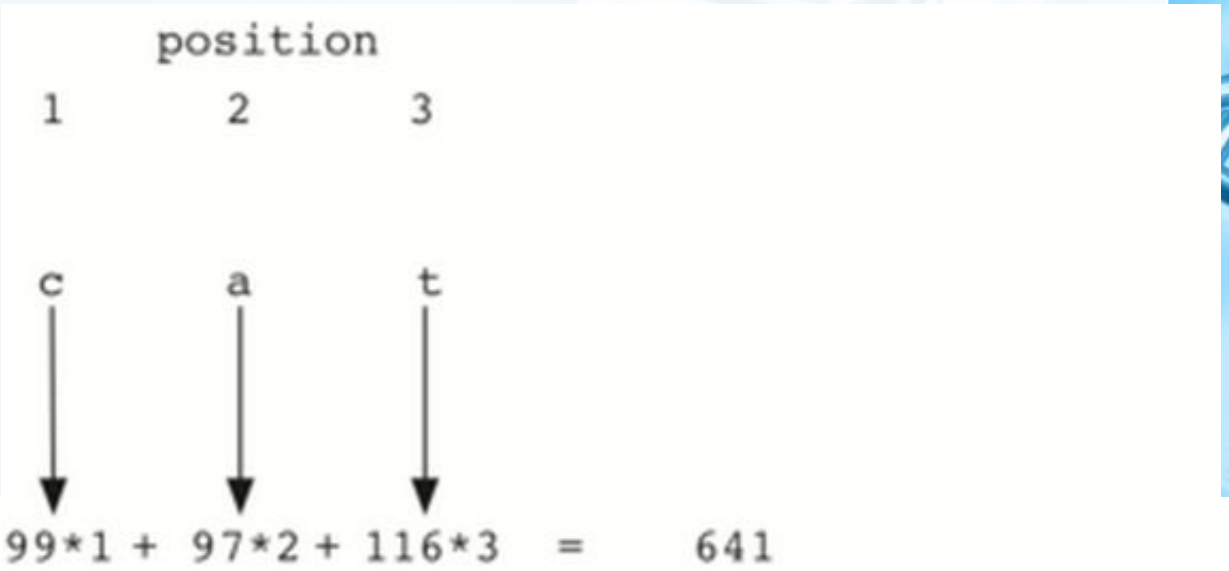
```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```



散列函数设计

- 当然，这样的散列函数对所有的变位词都返回相同的散列值，为了防止这一点，可以将字符串所在的位置作为权重因子，乘以ord值
- 我们还可以设计出更多的散列函数方法，但要坚持的一个基本出发点是，散列函数不能称为存储过程和搜索过程的计算负担，如果散列函数设计太过复杂，去花费大量的计算资源计算槽号，可能还不如简单地进行顺序搜索或者二分搜索，失去了散列本身的意义



冲突解决方案

- 如果两个数据项被散列映射到同一个槽，需要有一个系统化的方法在散列表中保存第二个数据项，这个过程称为“解决冲突”。
- 前面提到，如果说散列函数是完美的，那就不会有散列冲突，但完美散列函数常常是不现实的，所以解决散列冲突成为散列方法中很重要的一部分。
- 解决散列的一种方法就是为冲突的数据项再找一个开放的空槽来保存，其中最简单的就是从冲突的槽开始往后扫描，直到碰到一个空槽。（如果到散列表的尾部还未找到，则要从首部接着扫描）
- 这种寻找空槽的技术称为“开放定址open addressing”，向后逐个槽寻找的方法则是开放定址技术中的“线性探测linear probing”

冲突解决方案：线性探测Linear Probing

- **接前例，我们把44、55、20逐个插入到散列表中。**

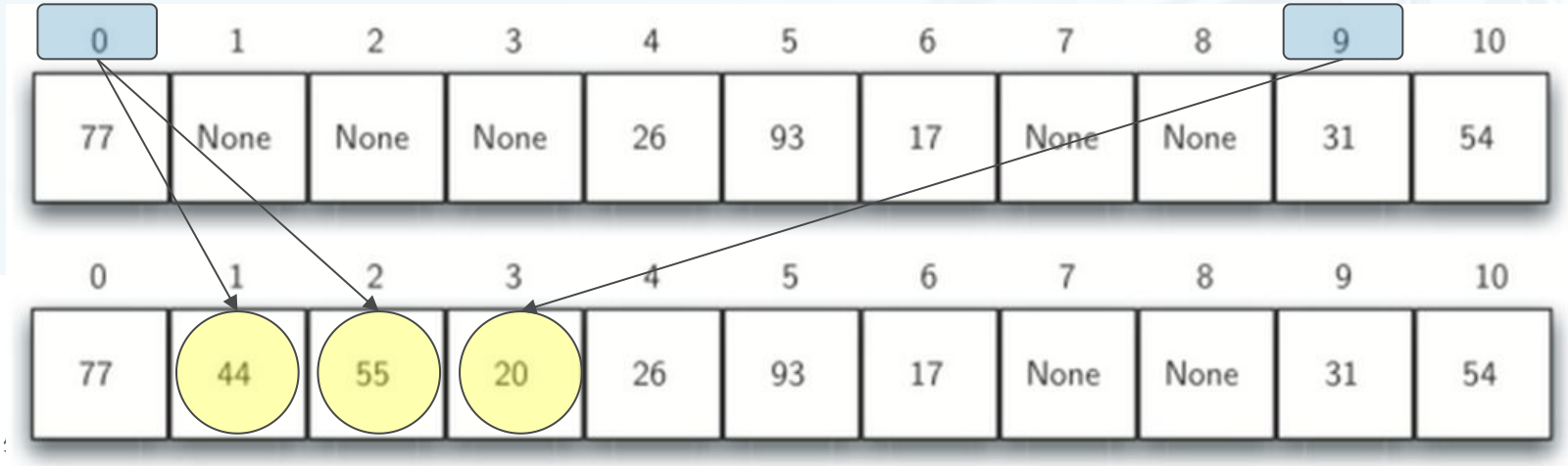
  h(44)== 0，发现0#槽已经被77占据了，向后找到第一个空槽1#，保存

  h(55)== 0，同样0#槽已经被占据，向后找到第一个空槽2#，保存

  h(20)== 9，发现9#槽已经被31占据了，向后，再从头开始找到3#槽保存

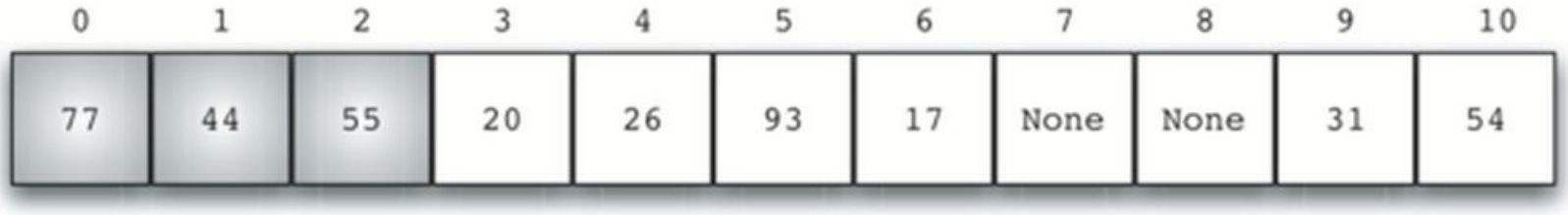
- **采用线性探测方法来解决散列冲突的话，则散列表的查找也遵循同样的规则**

  如果在散列位置没有找到查找项的话，就必须向后做顺序搜索，直到找到查找项，或者碰到空槽（搜索失败）。



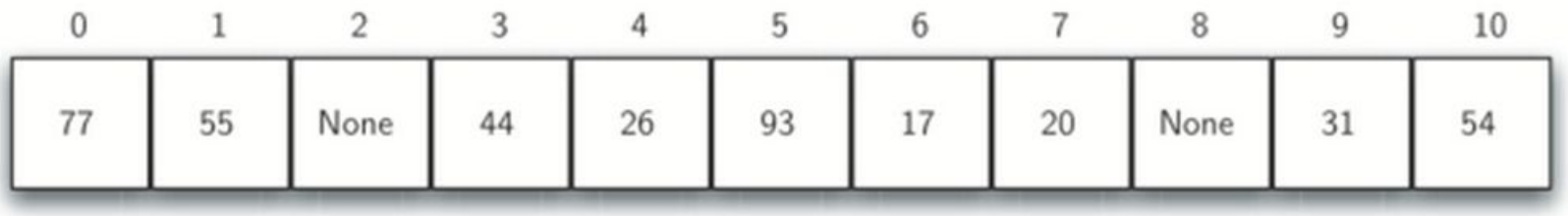
冲突解决方案：线性探测的改进

- 线性探测法的一个缺点是有聚集（clustering）的趋势，即如果同一个槽冲突的数据项较多的话，这些数据项就会在槽附近聚集起来，从而连锁式影响其它数据项的插入。



- 避免聚集的一种方法就是将线性探测扩展，从逐个探测改为跳跃式探测

下图是“+3”探测插入44、55、20



冲突解决方案：再散列rehashing

- **重新寻找空槽的过程可以用一个更为通用的“再散列rehashing”来概括，也就是说newhashvalue= rehash(oldhashvalue)**

  对于线性探测来说，rehash(pos)= (pos+ 1)% sizeoftable

  “+3”的跳跃式探测则是：rehash(pos)= (pos+ 3)% sizeoftable

  跳跃式探测的再散列通式是：rehash(pos)= (pos+skip)% sizeoftable

- **跳跃式探测中，需要注意的是skip的取值，不能被散列表大小整除，否则会产生周期，造成很多空槽永远无法探测到**

  一个技巧是，把散列表的大小设为素数，如例子的11

- **还可以将线性探测变为“二次探测quadratic probing”，不再固定skip的值，而是逐步增加skip值，如1、3、5、7、9，这样槽号就会是原散列值以平方数增加：h，h+1，h+4，h+9，h+16...**

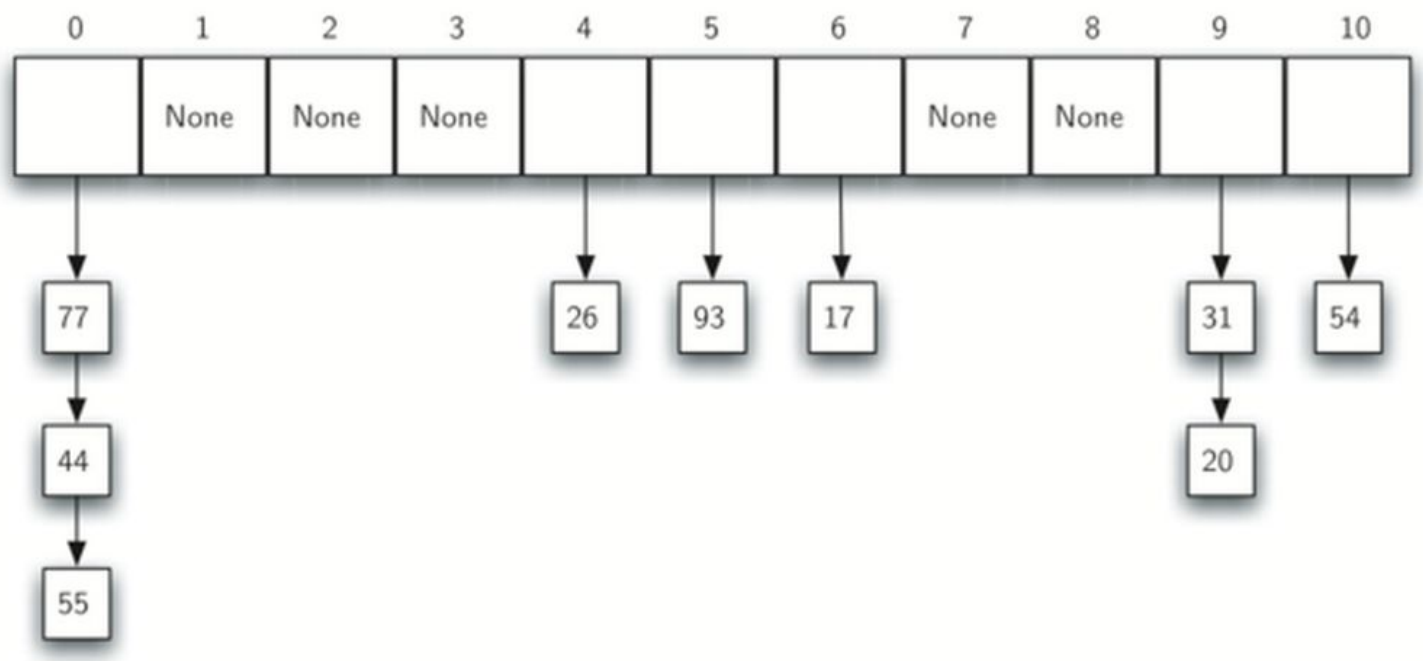
0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

冲突解决方案：数据项链Chaining

- **除了寻找空槽的开放定址技术之外，另一种解决散列冲突的方案是将容纳单个数据项的槽扩展为容纳数据项集合（或者对数据项链表的引用）**

  这样，散列表中的每个槽就可以容纳多个数据项，如果有散列冲突发生，只需要简单地将数据项添加到数据项集合中。

  查找数据项时则需要搜索同一个槽中的整个集合，当然，随着散列冲突的增加，对数据项的查找时间也会相应增加。



抽象数据类型“映射”：ADT Map

- **Python最有用的内置数据类型之一是“字典Dictionary”，字典是一种可以保存key-data键值对的数据类型，关键码可用于查询关联的数据值，这种键值关联的方法通常称为“映射map”**
- **ADT Map的结构是键-值关联的无序集合，关键码具有唯一性，通过关键码可以唯一确定一个数据值，ADT Map定义的操作如下：**

Map(): 创建一个空映射，返回空映射对象；

put(key, val): 将key-val关联对加入映射中，如果key已经存在，则将val替换原来的旧关联值；

get(key): 给定key，返回关联的数据值，如不存在，则返回None；

del: 通过del map[key]的语句形式删除key-val关联；

len(): 返回映射中key-val关联的数目；

in: 通过key in map的语句形式，返回key是否存在于关联中，布尔值

实现ADT Map

- **使用字典的优势在于，给定关键码，能够很快得到关联的数据值，为了达到快速查找的目标，需要一个支持高效搜索的ADT实现**

  可以采用列表数据结构加顺序搜索或者二分搜索。当然，更为合适的是使用前述的散列表来实现，这样查找可以达到最快O(1)的性能

- **下面，我们用一个HashTable类来实现ADT Map，该类包含了两个列表作为成员**

  其中一个slot列表用于保存key，另一个平行的data列表用于保存数据项

  在查找到一个key以后，在data列表对应相同位置的数据项即为关联数据

  保存key的列表就作为散列表来处理，这样可以迅速搜索到指定的key

  注意散列表的大小，虽然可以是任意数，但考虑到要让冲突解决算法能有效工作，应该选择为素数。

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

实现ADT Map：put方法代码

- hashfunction方法采用了简单求余方法来实现散列函数，冲突解决则采用线性探测“加1”再散列函数。

key不存在，未冲突

key已存在，替换val

散列冲突，再散列，直到找到空槽或者key

```
def put(self,key,data):
  hashvalue = self.hashfunction(key)

  if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
    else:
      nextslot = self.rehash(hashvalue)
      while self.slots[nextslot] != None and \
                      self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot)

      if self.slots[nextslot] == None:
        self.slots[nextslot]=key
        self.data[nextslot]=data
      else:
        self.data[nextslot] = data #replace
```

```
def hashfunction(self, key):
     return key% self.size

def rehash(self,oldhash):
    return (oldhash+ 1)% self.size
```

实现ADT Map：get方法

标记散列值为查找起点

找key，直到空槽或回到起点

未找到key，再散列继续找

回到起点，停

```
def get(self,key):
    startslot = self.hashfunction(key)

    data = None
    stop = False
    found = False
    position = startslot
    while self.slots[position] != None and  \
                         not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position=self.rehash(position)
            if position == startslot:
                stop = True
    return data
```

实现ADT Map：附加代码

- 通过特殊方法实现[]访问

```
def __getitem__(self,key):
    return self.get(key)

def __setitem__(self,key,data):
    self.put(key,data)
```

```
H=HashTable()
H[54]="cat"
H[26]="dog"
H[93]="lion"
H[17]="tiger"
H[77]="bird"
H[31]="cow"
H[44]="goat"
H[55]="pig"
H[20]="chicken"
print(H.slots)
print(H.data)

print(H[20])

print(H[17])
H[20]='duck'
print(H[20])
print(H[99])
```

```
>>>
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', None,
None, 'cow', 'cat']
chicken
tiger
duck
None
```

散列算法分析

- **散列在最好的情况下，可以提供O(1)常数级时间复杂度的搜索性能，但由于散列冲突的存在，搜索比较次数就没有这么简单。**
- **评估散列冲突的最重要信息就是负载因子λ，一般来说：**

  如果λ较小，散列冲突的几率就小，数据项通常会保存在其所属的散列槽中

  如果λ较大，意味着散列表填充较满，冲突会越来越多，冲突解决也越复杂，也就需要更多的比较来找到空槽；如果采用数据链的话，意味着每条链上的数据项增多

- **如果采用线性探测的开放定址法来解决冲突（λ在0~1之间）**

  成功的搜索，平均需要比对次数为：$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$

  不成功的搜索，平均比对次数为：$\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$

- **如果采用数据链来解决冲突（λ可以大于1）**

  成功的搜索，平均需要比对次数为：1+λ/2

  不成功的搜索，平均比对次数为：λ

随堂作业5-1

› http://pkuchenbin.mikecrm.com/lrtd3j

对于大小为13的散列表，27、130的散列值分别是多少？ *

- 1 , 10
- 13 , 0
- 1 , 0
- 2 , 3

对于大小为11的散列表，如果插入如下数据项，并采用线性探测解决散列冲突，最后的布局是什么？ *

113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99

- 100, __, __, 113, 114, 105, 116, 117, 97, 108, 99
- 99, 100, __, 113, 114, __, 116, 117, 105, 97, 108
- 100, 113, 117, 97, 14, 108, 116, 105, 99, __, __
- 117, 114, 108, 116, 105, 99, __, __, 97, 100, 113

排序：冒泡排序Bubble Sort

- **冒泡排序的算法思路在于对无序表进行多趟比较交换，每趟包括了多次两两相邻比较，并将逆序的数据项互换位置，最终能将本趟的最大项就位，经过n-1趟比较交换，实现整表排序**

  每趟的过程类似于“气泡”在水中不断上浮到水面的经过

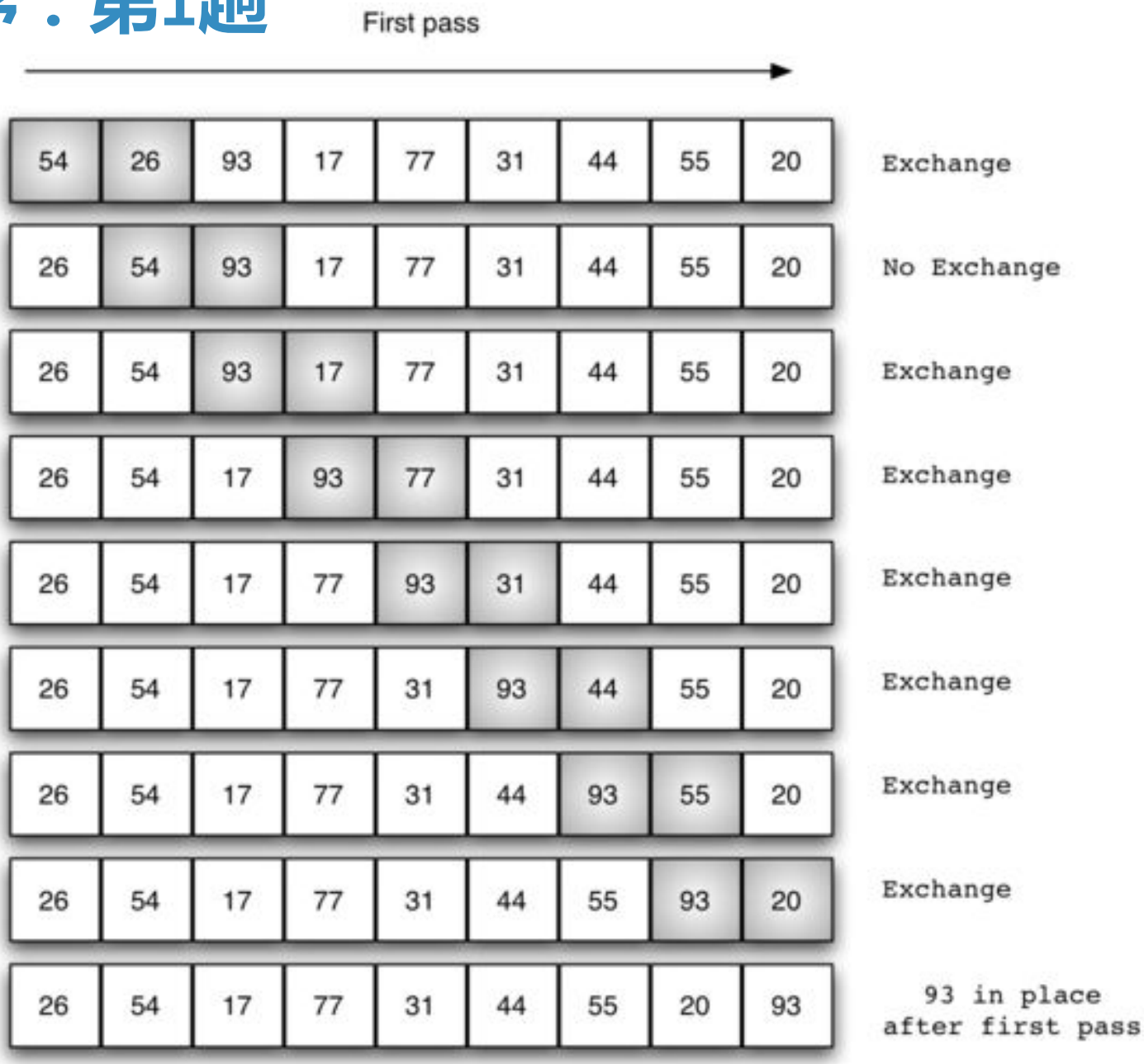
- **第1趟比较交换，共有n-1对相邻数据进行比较**

  一旦经过最大项，则最大项会一路交换到达最后一项

- **第2趟比较交换时，最大项已经就位，需要排序的数据减少为n-1，共有n-2对相邻数据进行比较**

- **直到第n-1趟完成后，最小项一定在列表首位，就无需再处理了。**

冒泡排序：第1趟


First pass
54 26 93 17 77 31 44 55 20
Exchange
26 54 93 17 77 31 44 55 20
No Exchange
26 54 93 17 77 31 44 55 20
Exchange
26 54 17 93 77 31 44 55 20
Exchange
26 54 17 77 93 31 44 55 20
Exchange
26 54 17 77 31 93 44 55 20
Exchange
26 54 17 77 31 44 93 55 20
Exchange
26 54 17 77 31 44 55 93 20
Exchange
26 54 17 77 31 44 55 20 93
93 in place after first pass

冒泡排序：代码

n-1趟

序错，交换

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Python支持直接交换

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

冒泡排序：算法分析

- **无序表初始数据项的排列状况对冒泡排序没有影响，算法过程总需要n-1趟，随着趟数的增加，比对次数逐步从n-1减少到1，并包括可能发生的数据项交换。**

- **比对次数是1～n-1的累加：** $\frac{1}{2}n^2 - \frac{1}{2}n$

  比对的时间复杂度是O(n^2)

- **关于交换次数，时间复杂度也是O(n^2)，通常每次交换包括3次赋值**

  最好的情况是列表在排序前已经有序，交换次数为0

  最差的情况是每次比对都要进行交换，交换次数等于比对次数

  平均情况则是最差情况的一半

- **冒泡排序通常作为时间效率较差的排序算法来作为其它算法的对比基准。**

  其效率主要差在每个数据项在找到其最终位置之前，必须要经过多次比对和交换，其中大部分的操作是无效的。

  但有一点优势，就是无需任何额外的存储空间开销。

冒泡排序：性能改进

- **另外，通过监测每趟比对是否发生过交换，可以提前确定排序是否完成，这也是其它多数排序算法无法做到的**

  如果某趟比对没有发生任何交换，说明列表已经排好序，可以提前结束算法

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
        exchanges = False
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                exchanges = True
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
        passnum = passnum-1

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

选择排序Selection Sort

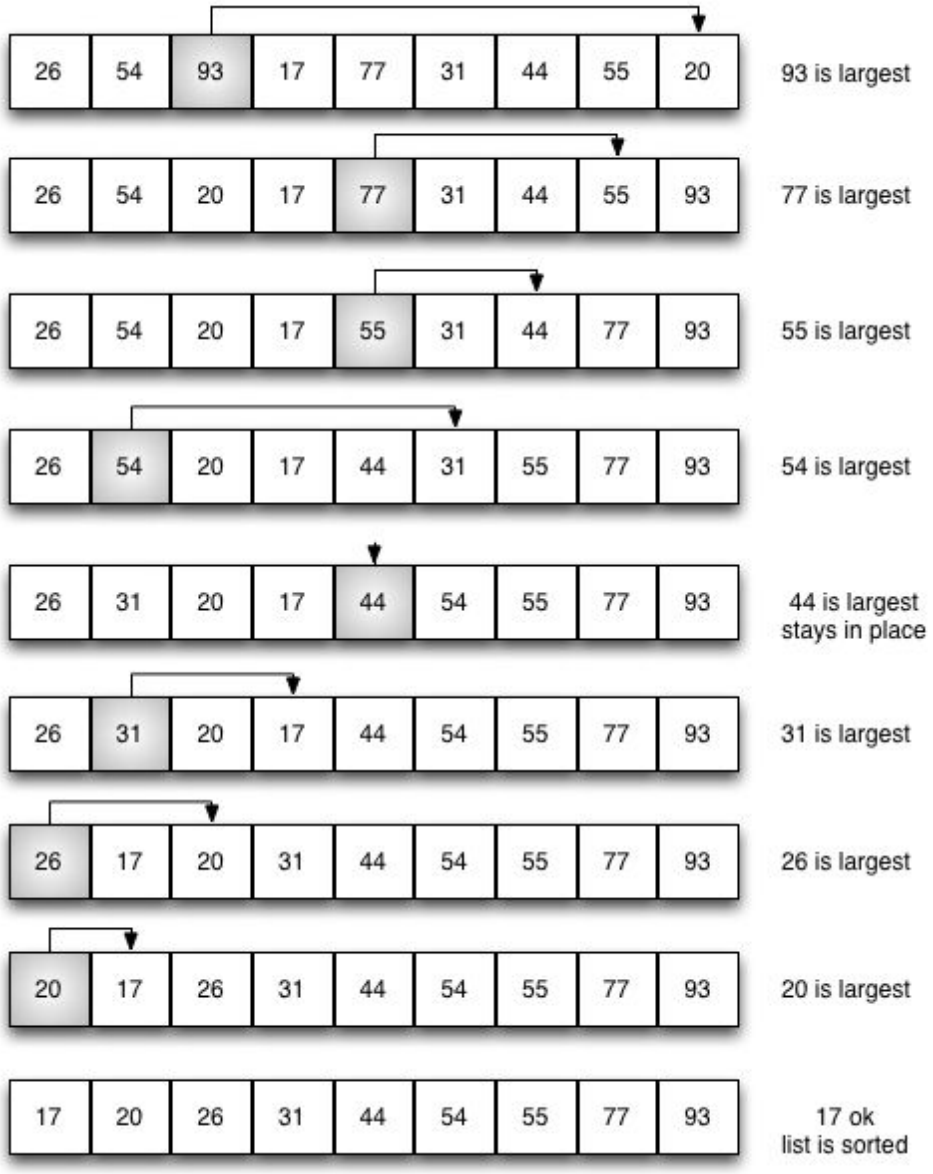
- **选择排序对冒泡排序进行了改进，保留了其基本的多趟比对思路，每趟都使当前最大项就位。**
- **但选择排序对交换进行了削减，相比起冒泡排序进行多次交换，每趟仅进行1次交换**

  记录最大项的所在位置，最后再跟本趟最后一项交换
- **选择排序的时间复杂度比冒泡排序稍优**

  比对次数不变，还是O(n^2)

  交换次数则减少为O(n)

选择排序：代码




```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

插入排序Insertion Sort

- **插入排序时间复杂度仍然是O(n^2)，但算法思路与前两个不同**
- **插入排序维持一个已排好序的子列表，其位置始终在列表的前部，然后逐步扩大这个子列表直到全表**

  第1趟，子列表仅包含第1个数据项，将第2个数据项作为“新项”插入到子列表的合适位置中，这样已排序的子列表就包含了2个数据项；

  第2趟再继续将第3个数据项跟前2个数据项比对，并移动比自身大的数据项，空出位置来，以便加入到子列表中

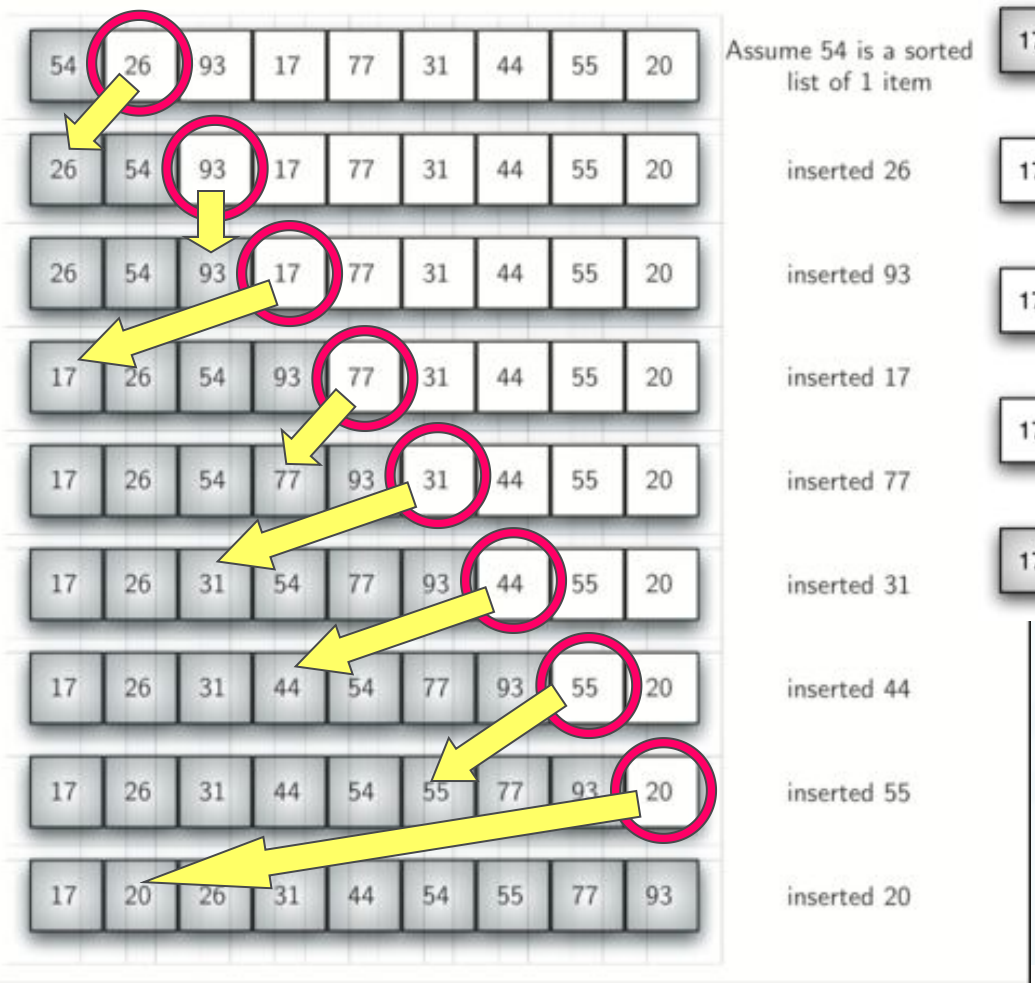
  经过n-1趟比对和插入，子列表扩展到全表，排序完成

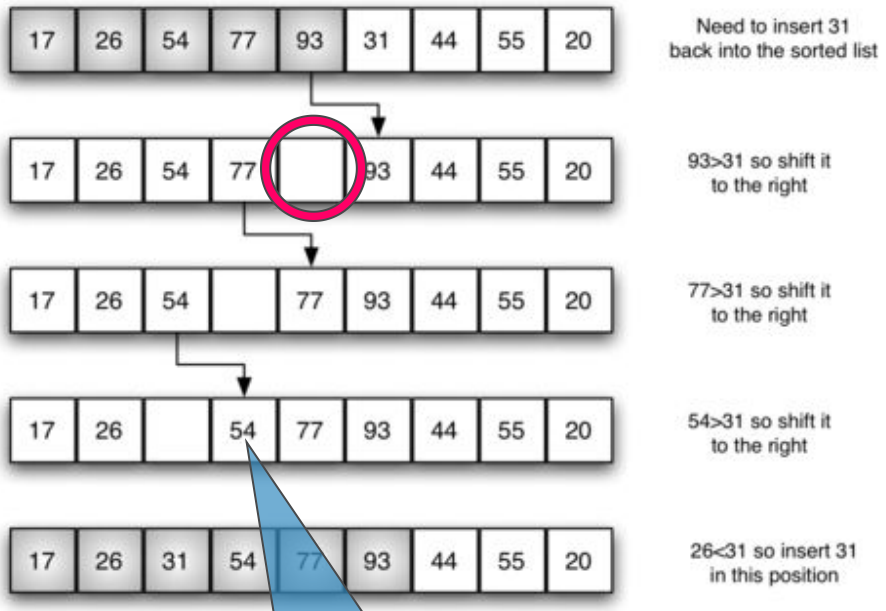
- **插入排序的比对主要用来寻找“新项”的插入位置**

  最差情况是每趟都与子列表中所有项进行比对，总比对次数与冒泡排序相同，数量级仍是O(n^2)

  最好情况，列表已经排好序的时候，每趟仅需1次比对，总次数是O(n)

插入排序：思路


54 26 93 17 77 31 44 55 20
Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20
inserted 26
26 54 93 17 77 31 44 55 20
inserted 93
17 26 54 93 77 31 44 55 20
inserted 17
17 26 54 77 93 31 44 55 20
inserted 77
17 26 31 54 77 93 44 55 20
inserted 31
17 26 31 44 54 77 93 55 20
inserted 44
17 26 31 44 54 55 77 93 20
inserted 55
17 20 26 31 44 54 55 77 93
inserted 20


17 26 54 77 93 31 44 55 20
Need to insert 31 back into the sorted list
17 26 54 77 93 44 55 20
93>31 so shift it to the right
17 26 54 77 93 44 55 20
77>31 so shift it to the right
17 26 54 77 93 44 55 20
54>31 so shift it to the right
17 26 31 54 77 93 44 55 20
26<31 so insert 31 in this position
比对，移动所有
比“新项”大的数据项

插入排序：代码


```
def insertionSort(alist):
    for index in range(1,len(alist)):

        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1

        alist[position]=currentvalue
```

新项/插入项

比对、移动

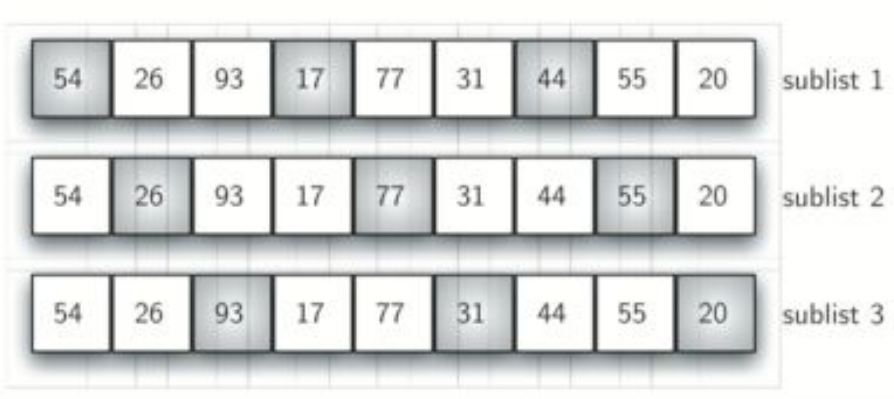
插入新项

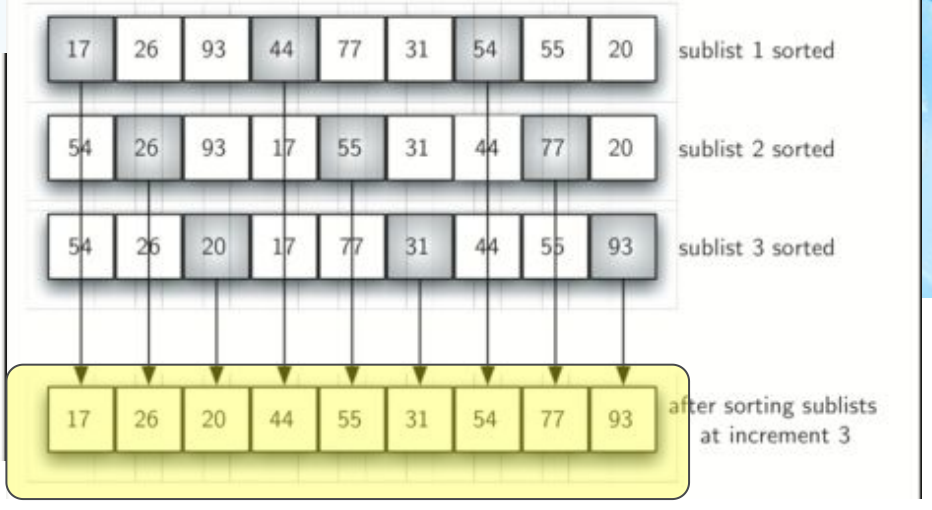
由于移动操作仅包含1次赋值，是交换操作的1/3，所以插入排序性能会较好一些。

谢尔排序Shell Sort

- 我们注意到插入排序的比对次数，在最好的情况下是O(n)，这种情况发生在列表已是有序的情况下，实际上，列表越接近有序，插入排序的比对次数就越少
- 从这个情况入手，谢尔排序以插入排序作为基础，对无序表进行“间隔”划分子列表，每个子列表都执行插入排序

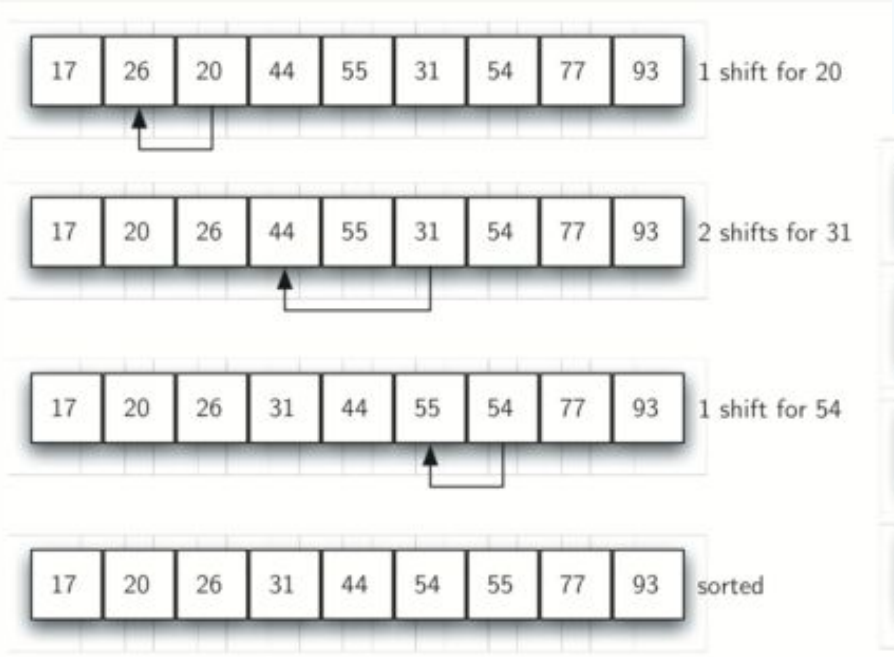
  随着子列表的数量越来越少，无序表的整体越来越接近有序，从而减少整体排序的比对次数
- 间隔为3的子列表，子列表分别插入排序后的整体状况更接近有序

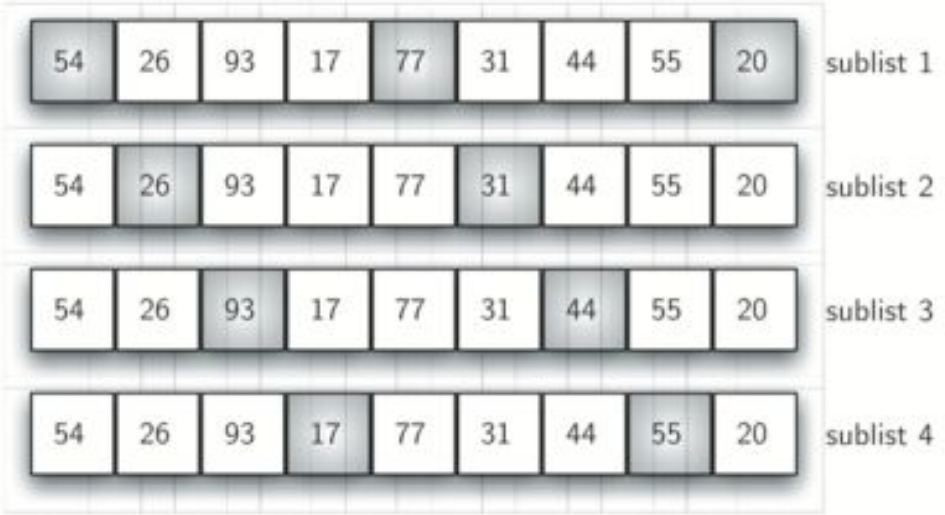




谢尔排序：思路

- 最后一趟是标准的插入排序，但由于前面几趟已经将列表处理到接近有序，这一趟仅需少数几次移动即可完成
- 子列表的间隔一般从n/2开始，每趟倍增：n/4，n/8……直到1





谢尔排序：代码

间隔设定

子列表排序

间隔缩小

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                       "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue
```

谢尔排序：算法分析

- 粗看上去，谢尔排序以插入排序为基础，可能并不会比插入排序好
- 但由于每趟都使得列表更加接近有序，这个过程会减少很多原先需要做的“无效”比对
- 对谢尔排序的详尽分析比较复杂，大致说是介于O(n)和O(n^2)之间
- 如果将间隔保持在2^k-1(1、3、5、7、15、31等等），谢尔排序的时间复杂度约为O（$n^{3/2}$）

$$O(n^{\frac{3}{2}})$$

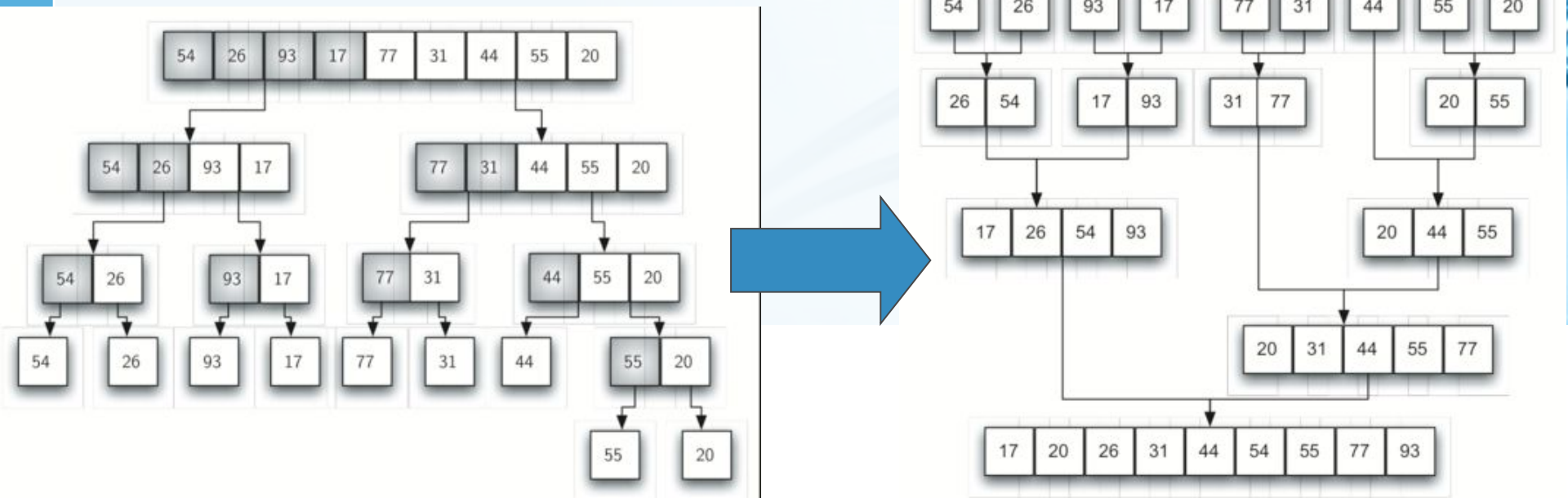
归并排序Merge Sort

- **下面我们来看看分而治之的策略在排序中如何应用**
- **归并排序是递归算法，思路是将数据表持续分裂为两半，对两半分别进行归并排序**

递归的基本结束条件是：数据表仅有1个数据项，自然是排好序的；

缩小规模：将数据表分裂为相等的两半，规模减为原来的二分之一；

调用自身：将两半分别调用自身排序，然后将分别排好序的两半进行归并，得到排好序的数据表。



归并排序：代码


```
def mergeSort(alist):
##    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i= j= k= 0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
##    print("Merging ",alist)
```

基本结束条件

递归调用

归并，i左半部，j右半部，k结果

拉链式交错把左右半部
从小到大归并到结果列表中

归并左半部剩余项

归并右半部剩余项

归并排序：算法分析

- **将归并排序分为两个过程来分析：分裂和归并**

  分裂的过程，借鉴二分搜索中的分析结果，是对数复杂度，其时间复杂度为O(log n)

  归并的过程，相对于分裂的每个部分，其所有数据项都会被比较和放置一次，所以是线性复杂度，其时间复杂度是O(n)

  综合考虑，每次分裂的部分都进行一次O(n)的数据项归并，总的时间复杂度是O(nlog n)

- **最后，我们还是注意到两个切片操作，为了时间复杂度分析精确起见，可以通过取消切片操作，改为传递两个分裂部分的起始点和终止点，也是没问题的，只是算法可读性稍微牺牲一点点。**

- **我们注意到归并排序算法使用了额外1倍的存储空间用于归并，这个特性在对特大数据集进行排序的时候要考虑进去**

随堂作业5-2

- **给定排序列表 [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]，在归并排序的第3次递归调用时，排序的是哪个子表？**

  a) [16, 49, 39, 27, 43, 34, 46, 40]

  b) [21,1]

  c) [21, 1, 26, 45]

  d) [21]

- **排序数据同上，归并排序中，哪两个子表是最先归并的？**

  a) [21, 1] and [26, 45]

  b) [1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]

  c) [21] and [1]

  d) [9] and [16]

快速排序Quick Sort

- **另一个采用分而治之策略的排序算法是快速排序，其优势是不需要额外的存储空间，这一点比归并排序强**

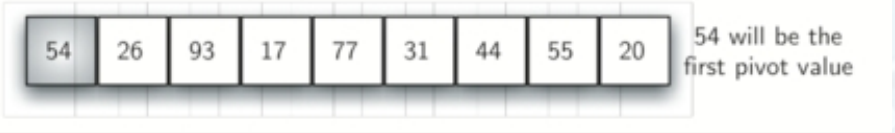
  但是！快速排序不总能把数据表正好分成相同长度的两半，这样在性能上就会打折扣了

- **快速排序的思路是依据一个“中值”数据项来把数据表分为两半：小于中值的一半和大于中值的一半，然后每部分分别进行快速排序**

  如果希望这两半拥有相等数量的数据项，则应该找到数据表的“中位数”

  但找中位数需要计算开销！要想没有开销，只能随意找一个数来充当“中值”

  比如，第1个数。



- **快速排序的递归算法“递归三要素”如下：**

  基本结束条件：数据表仅有1个数据项，自然是排好序的；

  缩小规模：根据“中值”，将数据表分为两半，最好情况是相等规模的两半

  调用自身：将两半分别调用自身进行排序（排序基本操作在分裂过程中）

快速排序：图示

- **分裂数据表的目标**
  找到“中值”应处的位置

- **分裂数据表的手段**
  设置左右标（left/rightmark）

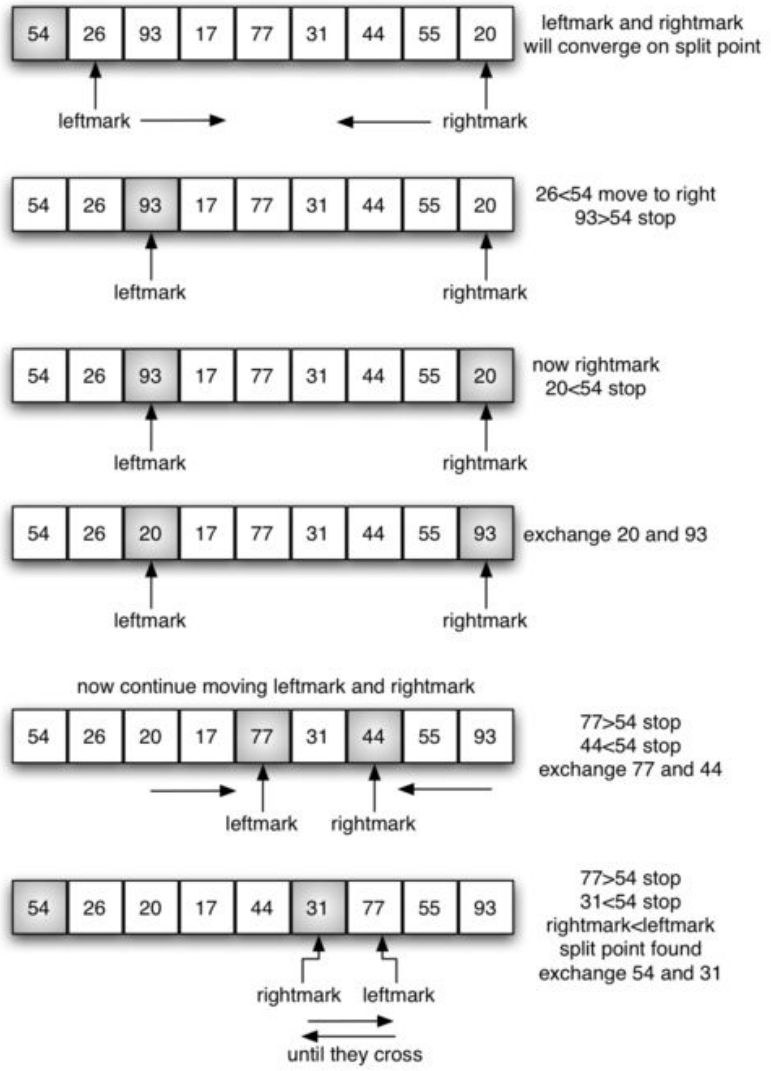
  左标向右移动，右标向左移动

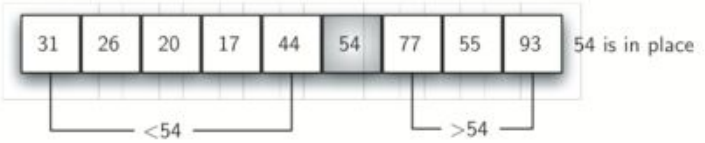
  - 左标一直向右移动，碰到比中值大的就停止
  - 右标一直向左移动，碰到比中值小的就停止
  - 然后把左右标所指的数据项交换

  继续前述的移动过程，直到左标移到右标的右侧，停止移动

  这时右标所指位置就是“中值”应处的位置，将中值和这个位置交换

  分裂完成，左半部全比中值小，右半部都比中值大





快速排序：代码

基本结束条件

分裂为两部分

递归调用

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)

        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
```

```
alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

快速排序：代码


```
def partition(alist,first,last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and \
                alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and \
                rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp


    return rightmark
```

选定“中值”

左右标初值

向右移动左标

向左移动右标

两标相错就结束移动

左右标指向的值交换，继续移动

中值就位

中值点，也是分裂点

快速排序：算法分析

- **快速排序基本的过程也分为两部分：分裂和移动**

  如果分裂总能把数据表分为相等的两部分，那么就是O(log n)的复杂度；

  而移动需要将每项都与中值进行比对，还是O(n)；

  综合起来就是O(nlog n)；

  而且，算法运行过程中不需要额外的存储空间。

- **但是，如果不那么幸运的话，中值所在的分裂点过于偏离中部，造成左右两部分数量不平衡**

  极端情况，有一部分始终没有数据，这样时间复杂度就退化到O(n^2)，还要加上递归调用的开销（比冒泡排序还糟糕）

- **可以适当改进下中值的选取方法，让中值更具有代表性**

  比如“三点取样”，从数据表的头、尾、中间选出中值

  会产生额外计算开销，仍然不能排除极端情况

随堂作业5-2

- **给定排序列表[14, 17, 13, 15, 19, 10, 3, 16, 9, 12]，快速排序在第2次分裂后，列表内容是：**

  a) [9, 3, 10, 13, 12]

  b) [9, 3, 10, 13, 12, 14]

  c) [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]

  d) [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

- **给定排序列表 [1, 20, 11, 5, 2, 9, 16, 14, 13, 19]，快速排序如果采用“三点取样”法，得到的第1个“中值”是：**

  a) 1 b) 9 c) 16 d) 19

- **下面哪些算法，即使在最坏情况下，复杂度还保证是O(nlog n)**

  a)谢尔排序 b)快速排序 c)归并排序 d)插入排序

算法的选择：一封同学来信

- **我们在冒泡排序算法中提到一个对其改进的算法ShortBubbleSort，可以利用“如果一趟比对未产生任何交换，则无需后续计算，已排好序”的“短路”特性，来消除无效比对，理应更快！**

- **但有细心的同学自己动手做了试验，结果不太乐观：**

  我试了一下排序算法，发现shortBubbleSort比BubbleSort慢很多啊（每组数据运行10次取平均），算法就是照着幻灯片上写的，为什么呢？

```
1  def BubbleSort(alist):
2      passnum = len(alist) - 1
3      while passnum > 0:
4          for i in range(passnum):
5              if alist[i] > alist[i + 1]:
6                  alist[i],alist[i + 1] = alist[i + 1],alist[i]
7          passnum -= 1
8      return alist
```

```
10 def ShortBubbleSort(alist):
11     exchanges = True
12     passnum = len(alist) - 1
13     while passnum > 0 and exchanges:
14         exchanges = False
15         for i in range(passnum):
16             if alist[i] > alist[i + 1]:
17                 exchanges = True
18                 alist[i],alist[i + 1] = alist[i + 1],alist[i]
19         passnum -= 1
20     return alist
```

算法的选择：一封同学来信

- **第一次数据：**

  alist = [i for i in range(10) for j in range(10)]（再用shuffle打乱）

  {'Sort': 0.0, 'shuffle': 0.0, 'SelectionSort': 0.0, 'InsertionSort': 0.0, 'BubbleSort': 0.0, 'ShortBubbleSort': 0.0016000032424926757, 'ShellSort': 0.0}

- **第二次数据：**

  alist = [i for i in range(100) for j in range(100)]（再用shuffle打乱）

  {'Sort': 0.0, 'shuffle': 0.0, 'SelectionSort': 2.8600001335144043, 'InsertionSort': 3.8279998302459717, 'BubbleSort': 7.0, 'ShortBubbleSort': 7.921999931335449, 'ShellSort': 0.04700016975402832}

- **第三次数据：**

  alist = [i for i in range(1000) for j in range(100)]（再用shuffle打乱）

  {'Sort': 0.031000137329101562, 'shuffle': 0.031000137329101562, 'SelectionSort': 297.72199988365173, 'InsertionSort': 397.25999999046326, 'BubbleSort': 765.3139998912811, 'ShortBubbleSort': 847.1009998321533, 'ShellSort': 0.6410000324249268}

  （运行了一天）

算法的选择：一封同学来信

- **为什么理论和实际相差甚远？**
  到底还能不能愉快地选择一个好算法了？

- **ShortBubbleSort的“短路”优势高度依赖于数据的初始布局**
  如果数据布局的随机度很高，造成每趟比对都会发生交换的话，ShortBubbleSort就完全没有优势，还要额外付出一个exchanges变量和相应赋值语句的代价，反倒比原始的冒泡排序要慢。

- **同学的测试代码没有问题，问题在于排序对象，其数据是经过random.shuffle来乱序的**
  首先，alist是用range嵌套生成，生成的数据从小到大排列，非常整齐；
  而这个shuffle会尽量把数据打乱到最混乱的程度，造成数据布局随机度很高；
  这样，ShortBubbleSort的短路特性就完全失效，还要付出exchanges变量的判断、赋值代价，实测比原始冒泡排序算法要慢不少。

算法的选择：一封同学来信

- **所以排序算法有时候并不能绝对的优劣，尤其是时间复杂度相同的算法们**
- **要在特定的应用场合取得最高排序性能的话，还需要对数据本身进行分析，针对数据的特性来选择相应排序算法**
- **另外，除了时间复杂度，有时候空间复杂度也是需要考虑的关键因素**

  归并排序时间复杂度O(nlog n)，但需要额外一倍的存储空间

  快速排序时间复杂度最好的情况是O(nlog n)，而且不需要额外存储空间，但“中值”的选择又成为性能的关键，选择不好的话，极端情况性能甚至低于冒泡排序

- **算法选择不是一个绝对的过程，需要综合考虑各方面的因素**

  包括运行环境要求、处理数据对象的特性

本章小结

- 在无序表或者有序表上的顺序搜索，其时间复杂度为O(n)
- 在有序表上进行二分搜索，其最差复杂度为O(log n)
- 散列表可以实现常数级时间的搜索
- 冒泡排序、选择排序和插入排序是O(n^2)的算法
- 谢尔排序在插入排序的基础上进行了改进，采用对递增子表排序的方法，其时间复杂度可以在O(n)和O(n^2)之间
- 归并排序的时间复杂度是O(nlog n)，但归并的过程需要额外存储空间
- 快速排序最好的时间复杂度是O(nlog n)，也不需要额外的存储空间，但如果分裂点偏离列表中心的话，最坏情况下会退化到O(n^2)