排序与搜索

5 排序与搜索

5.1 目标

- 了解和实现顺序搜索和二分搜索。
- 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。
- 了解搜索技术中的哈希算法。
- 了解抽象数据类型Map。
- 利用哈希实现抽象数据类型Map。

5.2 搜索

现在开始研究搜索和排序，它们是计算中的常见问题。本节将介绍搜索，排序在稍后的章节。搜索是在元素的容器中找到某个特定元素的算法过程。通常来说，搜索会返回True或者False来表示目标项是否存在。有时也可能返回目标项所在位置。考虑到本节的目标，此处仅研究是否存在的问题。

Python提供了一个非常简便的用于判断元素是否存在于容器中，即使用in操作符。

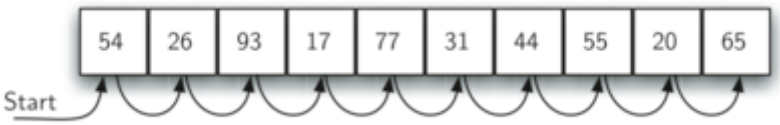
```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

虽然这个代码很容易写出来，但显然也需要执行某个底层程序。事实上，有许多方法实现搜索。本节关心的是这些算法是如何运行以及它们之间如何进行对比。

5.3 顺序搜索

当元素被存入容器中，比如说列表，这些元素便有了线性或者说顺序关系。每个数据项都被存储在与其它数据项相对的位置。以Python列表来说，所谓的相对位置即是各元素的索引值（index）。由于索引值是有序的，因此对其可以进行顺序访问。这一过程便产生了第一种搜索技术，顺序搜索。

顺序搜索如图1所示。从列表的第1项开始，依次逐项移动，按照顺序即可，直到遍历尽所有元素。当遍历了所有数据后，最终发现目标项并不在该列表中。



../_images/seqsearch.png

该算法的Python实现如代码片段1所示。该函数接受一个列表和一个目标项作为参数并返回布尔值表示是否存在。

```
| 1 | def sequentialSearch(alist, item): |
| 2 |     pos = 0 |
| 3 |     found = False |
| 4 |  |
| 5 |     while pos < len(alist) and not found: |
| 6 |         if alist[pos] == item: |
| 7 |             found = True |
| 8 |         else: |
| 9 |             pos = pos+1 |
| 10 |  |
| 11 |     return found |
| 12 |  |
| 13 | testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0] |
| 14 | print(sequentialSearch(testlist, 3)) |
| 15 | print(sequentialSearch(testlist, 13)) |
```

** 代码片段1:无序列表的顺序搜索 **

5.3.1 顺序搜索分析

为了分析搜索算法，有必要确定一个基本的计算单位。回想一下，一般来说，这个基本单位就是解决问题所需要的步骤数。就搜索来说，可以采用核对次数作为基本单位。此外，这里做了一个假设：列表是无序的，即元素在列表中的位置是随机的。换句话说，在任意位置找到目标项的概率都是一样的。

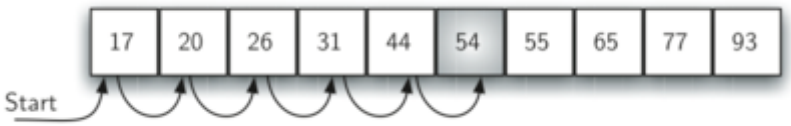
若目标项并不存在于列表中，确定此事实的唯一办法是将其与每一项进行对比。如果有n项，那么顺序搜索需要进行n次核对来确认目标项确实不存在。若目标项在列表中，分析起来就不是那么直接了。实际上有3种情况。最好的情况是目标项就在列表的首项，此时仅只需一次对比。最差的情况是，直到最后一项才找到目标项，此时进行了n次对比。

平均起来，应该是在列表的中部发现目标项，也就是说，需要进行$\frac{n}{2}$次对比。读者应该还记得，随着n的增大，不管系数为多少实际上都没意义了，因此顺序搜索的时间复杂度是O(n)，如表1所示。

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

之前假设了容器中的元素相互之间是没有位置关系的。那么假如这些元素有某种顺序的话，顺序搜索又是如何？能否在效率方面获得提升？

假定列表中的各项是以递增的关系构建的。如果目标项在列表中，在n个位置中的某一个找到该元素的概率跟之前是一样，依然要进行n次核对。然而，假如该项不存在于列表中，便可以获得一些提升。图2演示了搜索50的过程。注意，顺序对比进行到54时停止了，因为此时确定一些其它的东西：在54之后也不会有目标项存在了，因为该列表是有序的。在这种情况下，算法并不需要遍历完整个列表来确定目标项不存在，它可以立刻停止。代码片段2给出了上述的Python代码。



../_images/seqsearch2.png

代码片段2:有序列表的顺序搜索

```
| 1 | def orderedSequentialSearch(alist, item): |
| 2 |     pos = 0 |
| 3 |     found = False |
| 4 |     stop = False |
| 5 |     while pos < len(alist) and not found and not stop: |
| 6 |         if alist[pos] == item: |
| 7 |             found = True |
| 8 |         else: |
| 9 |             if alist[pos] > item: |
| 10 |                 stop = True |
| 11 |             else: |
| 12 |                 pos = pos+1 |
| 13 |  |
| 14 |     return found |
| 15 |  |
| 16 | testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,] |
```

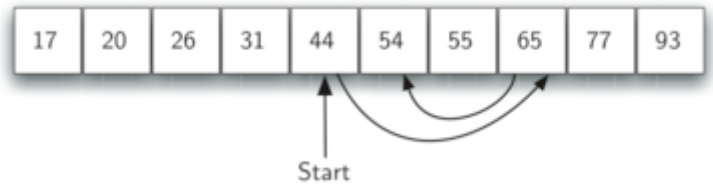
```
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

表2对这些结果进行了总结。注意，确认列表中不含目标项的最好情况是只看1项。平均来看，也需要遍历$\frac{n}{2}$项，因此其时间复杂度仍然是O(n)。总儿言之，顺序搜索仅在有序列表中不含目标项时能够有所提升。

5.4 二分搜索

采用更加灵巧的办法来进行比较的话，可以更好地利用有序列表的优势。在顺序搜索中，当与第一项进行对比时，如果第1项不是目标项的话，便最多会有n-1项需要遍历。二分法从中间项开始进行搜索，而不是从第一项。如果该项恰好就是目标项，那么便完成了。如果不是，便可以利用列表的有序性排除一半的剩余项。如果目标项比该项大，那么便可以确定列表的较小的那一半以及当前的中间项排除了。如果该项在列表中，那么一定是在更大的那一半中。

在较大的那一半中重复以上过程。从中间项开始并且与目标项进行对比，同样，要么找到目标项，要么再将搜索的列表分割一次，从而将搜索空间再次缩小一半。该算法可以快速地找到54，如图3所示，完整代码如代码片段3所示。



../_images/binsearch.png

代码片段3:有序列表的二分搜索

```
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
```

```
| 13 |             else: |
| 14 |                 first = midpoint+1 |
| 15 |  |
| 16 |     return found |
```

在进行分析之前，读者应当已注意到该算法是典型的**分而治之**策略。分而治之即将问题分为多个小规模的部分，以某种方式将各部分进行解决，最后重组为整个问题来获得解。对列表进行二分搜索时，首先检查的是中间项。如果目标项比中间项更小，便可以对原列表的左侧部分再进行二分搜索。反之，则对右侧进行二分搜索。不管是哪种，实质都是对二分搜索函数的递归调用，并且传入的是更小规模的列表。代码片段4给出了该递归算法。

** 代码片段4:二分搜索（递归版）**

```
| 1 | def binarySearch(alist, item): |
| 2 |     if len(alist) == 0: |
| 3 |         return False |
| 4 |     else: |
| 5 |         midpoint = len(alist)//2 |
| 6 |         if alist[midpoint]==item: |
| 7 |           return True |
| 8 |         else: |
| 9 |           if item<alist[midpoint]: |
| 10 |             return binarySearch(alist[:midpoint],item) |
| 11 |           else: |
| 12 |             return binarySearch(alist[midpoint+1:],item) |
| 13 |  |
| 14 | testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,] |
| 15 | print(binarySearch(testlist, 3)) |
| 16 | print(binarySearch(testlist, 13)) |
```

5.4.1 二分搜索的分析

要对二分搜索算法进行分析，一定要谨记每次对比后都将问题规模缩小了一半。那么对整个列表进行比较时，该算法所需最多的对比次数是多少？如果从n项开始，在第1次对比后将剩下$n/2$项；在第2次对比后，将剩下$n/4$项....那么到底可以分割多少次呢？表3可以给出该答案。

Comparisons	Approximate Number of Items Left
1	$n/2$
2	$n/4$

Comparisons	Approximate Number of Items Left
3	$n/8$
...	
i	$n/4^i$

如果拆分次数足够高，最后就只剩下了1项，该项要么是目标项要么不是。不管是哪种情况，算法都结束了。执行到这一步所需要的对比次数i满足$\frac{n}{2^i}=1$，解得$i=logn$。最大对比次数是列表元素个数的对数函数，因此二分搜索的时间复杂度是O(logn)。

还有一个问题需要注意，在给出的递归算法中，递归调用binarySearch(alist[:midpoint],item)使用了切片操作符来生成下一次调用所需的列表左侧。在上述分析中是假定切片操作符的时间复杂度为常数即O(1)。然而在Python中，切片运算的时间复杂度实际上是O(k)。这意味着使用切片操作符的二分搜索并不会严格地符合对数时间复杂度。幸运的是，这可以通过传递列表的开始和结束索引值来解决。索引值可以通过代码3来解决，其实现就作为练习。

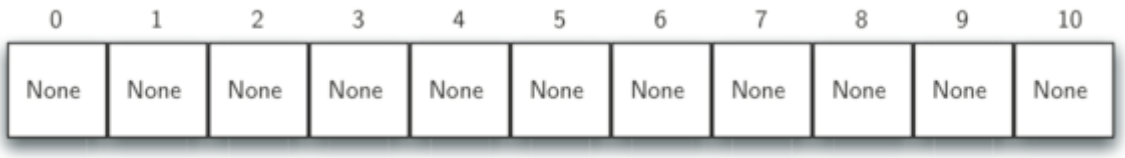
虽然二分搜索一般来说比顺序搜索更优，然而值得注意的是，当n比较小的时候，排序所附带的额外消耗或许是不划算的。实际上，需要考虑是否值得进行排序来获得二分搜索的优势。如果排序1次而进行多次搜索，那么排序的开销也就不那么突出了。然而，对于大列表来说，进行排序的开销可以是非常巨大的，此时进行顺序搜索有可能会是最佳选择了。

5.5 哈希

在之前的章节中，读者应该已经注意到了，可以利用容器中元素之间相对位置的信息来优化搜索算法。比如说，如果知道列表是有序的，那么便可以通过二分搜索实现对数复杂度。在本节中，将通过建立一种新的数据结构来实现O(1)的搜索算法时间复杂度。该概念被称为**哈希**（hash）。

为此，当在容器中进行搜索时，必须掌握更多各项可能位置的信息。如果每一项都在正确的位置上，搜索过程只需要进行一次便可以找到目标项。然而，通常情况都不会这么简单。

哈希表（hash table）是一种容器，其中的元素以一定的方式进行存储来使得搜索操作更加方便。哈希表的每一个位置，通常被称为槽（slot），可以容纳一个元素，并且以从0开始的整数对其命名。例如，第1个槽记为0，第2个槽记为1，第3个槽为2，以此类推。在初始化时，哈希表中是没有元素的，因此每个槽都为空。可以利用列表实现哈希表，其中的每个元素都被初始化为Python特殊值None。如图4所示，该哈希表长度为m=11。换言之，表中有m个槽，依次命名为0到10。

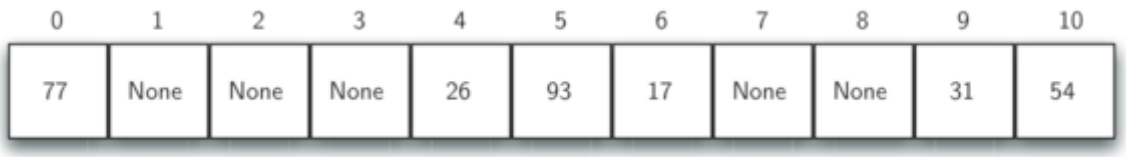


../_images/hashtable.png

某个元素与其在哈希表对应的槽之间的映射关系被成为**哈希函数**。哈希函数会将容器中的任意元素映射到槽命名区间之间的某个整数，即0到m-1。假设有一列整数54,26,93,17,77,31。先给出第1个哈希函数，有时被称为“求余法”，简单地将该元素与哈希表大小相除，得到的余数作为哈希值（h(item)=item%11），如表4所示。注意求余法（或者求模法)广泛地以某种形式出现在各种哈希函数中，因为其结果必须在槽命名空间内。

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

一旦哈希值计算完成，便可以将各项插入哈希表中的指定位置，如图5所示。可以发现，11个槽中仅有6个非空。这被称为**负载系数（load factor）**，一般表示为$\lambda = \frac{\text{元素个数}}{\text{哈希表长度}}$，比如本例为$\lambda = \frac{6}{11}$。



../_images/hashtable2.png

当进行搜索时，简单地使用哈希函数计算出槽的名字，然后在哈希表中检查是否存在即可。该搜索操作为O(1)，因为计算哈希值并且在哈希表中对该位置进行索引所消耗的时间为常数。若所有元素都在合适的位置，那么便获得了一种时间复杂度为常数的算法。

读者可能已经看出了该技术实现的关键在于必须保证每个元素映射到哈希表中的位置是唯一的。比如说，如果项44是容器中的下1项，那么其哈希值为-，然而77的哈希值也是0，那么问题来了。根据该哈希函数，有2个甚至可能更多的元素都在同一个槽里，这种情况称之为**冲突（collision/clash）**。显然，这种冲突导致哈希算法不可行了。稍后将对此进一步讨论。

5.5.1 哈希函数

给定一组元素，若某个哈希函数可以将其每个元素映射到唯一的槽中，那么该哈希函数便被称为**完美哈希函数（perfect hash function）**。若确定元素以及该容器是不可变的，那么就有可能构造出一个

完美哈希函数（参考练习以了解更多关于完美哈希函数）。不过，若给定的元素容器是不加限制的，并没有系统化的方法来生成完美哈希函数。然而幸运的是，使用不完美哈希函数同样也可以获得不错的性能提升。

获得完美哈希函数的方法之一是增加哈希表的长度，以此来保证所有可能出现的元素产生的哈希值都被哈希表所包含。这一方法便保证了每一个元素都有唯一的槽，它可能对元素数较少时比较使用，但是当可能出现的元素数较大时便有些尴尬了。比如说，如果元素是9位的社会安全码时，该方法就需要近10亿个槽了。若只需为25个学生保存该数据，那可以说是浪费了巨大的内存了。

那么目标便确定了：生成一个容易计算的哈希函数，它可以最小化冲突数并且将元素平衡地分配在哈希表中。有很多常用的方法可以将求余法进行扩展。这里介绍其中的几种。

折叠法（folding method）先将元素分为相同长度的小部分（最后一小部分可能长度不一样），接着将这些小部分相加以得到最后的哈希值。比如说，如果元素是电话号码436-555-4601，可以将这些数字拆分为2位数的组合（43,65,55,46,01）。经过加法即 43+65+55+46+01 可以得到210。若假设哈希表有11个槽，则将210除以11得到余数1，将其保存，那么电话号码436-555-4601的哈希值为1。有些折叠法会进一步地将每一个小部分逆序。比如上面这个例子，做法是43+56+55+64+01=219，然后 219 % 11=10。

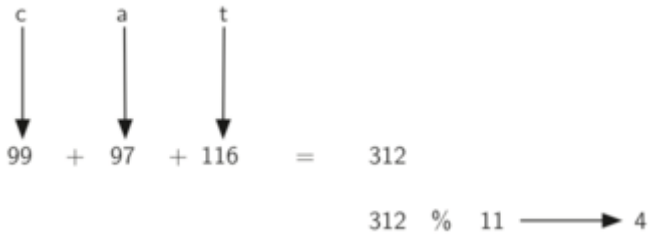
另一种构造哈希函数的数值方法是**平方取中法（mid-square method）**。首先将各元素，然后取结果的部分位数。比如，假如元素为44，首先计算平方值为$44^2 = 1936$。将中间两位提出来，93，对其作求余运算，得到5（93%11）。表5给出了在求余法和平法取中法的结果。读者应该确认自己明白如何得到这些值。

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

对基于字符的元素比如说字符串，也可以创建哈希函数。单词“cat”可以认为是一组普通数字的序列。

```
>>> ord('c')
99
>>> ord('a')
97
>>> ord('t')
116
```

将这3个数字加起来，使用求余法得到哈希值（如图6所示）。代码1给出了名为hash的函数，它接受1个字符串和1个哈希表长度作为参数，返回0到tablesize-1的哈希值。

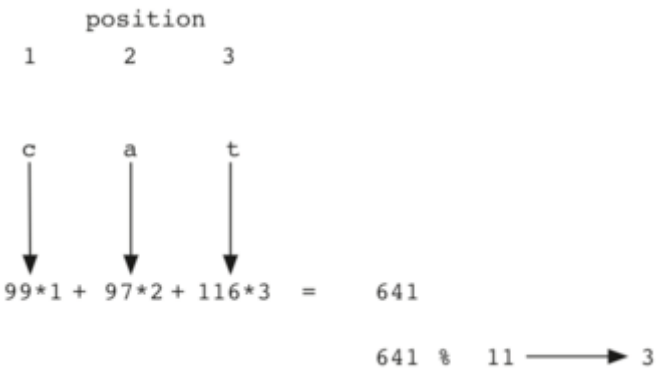


../_images/stringhash.png

代码1

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

    return sum%tablesize
```

有趣的是，使用该哈希函数时，回文词（正序、逆序都一样的英文单词）会得到相同的哈希值。为了纠正，可以将字符的位置作为权重，如图7给出的一种可行的办法。对该哈希函数的修改就作为练习了。



读者可以想出很多种其它方法来为容器中的元素计算哈希值，值得注意的是，一定要保证哈希函数足够高效，避免其消耗最主要的存储空间和搜索过程。若哈希函数过于复杂，相比于直接使用基本的顺序搜索或者二分搜索，反而会消耗更多资源来计算槽名。这就失去了使用哈希函数的意义。

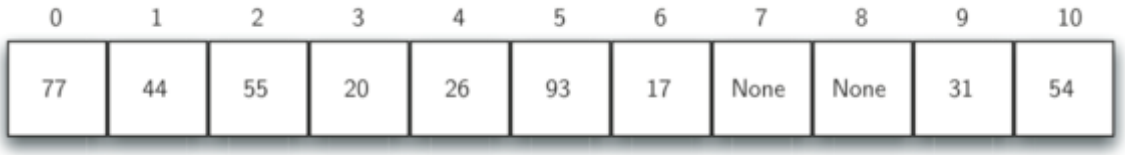
5.5.2 冲突处理

现在回过头来处理冲突的问题。当两个元素的哈希值一致时，必须有一种系统化的方法来将第2个元素放在哈希表中，该过程被称为**冲突处理（collision resolution）**。如前文所述，完美哈希函数是不会出现冲突的。然而，一般来说这是不大可能的，所以冲突处理是使用哈希算法的重要部分。

处理冲突的一种办法是在哈希表内部为哈希表内部为冲突项找到另一个空槽来放置。一种简单的实现办法是，从当前的哈希值所在位置出发，以顺序方式向下遍历槽，直到找到第1个空槽。注意，有可能需要回到第1个槽（从尾回到首的循环）来保证覆盖了整个哈希表。这种冲突处理过程被称为**空槽寻址（open addressing）**，它试图在哈希表中找到下一个空槽或者空地址。通过系统性地逐个遍历每个槽，这种开放寻址技术被称为**线性探测（linear probing）**。

图8给出了对1个扩大的整数集合(54,26,93,17,77,31,44,55,20)使用简单求余法后的结果。上面的表4给出了各原始项的哈希值，原始内容如图5所示。当试图将44放在槽0时，出现了冲突。使用线性探测，一个槽一个槽地检查，直到发现了一个空槽。在这个例子中，找到的是槽1。

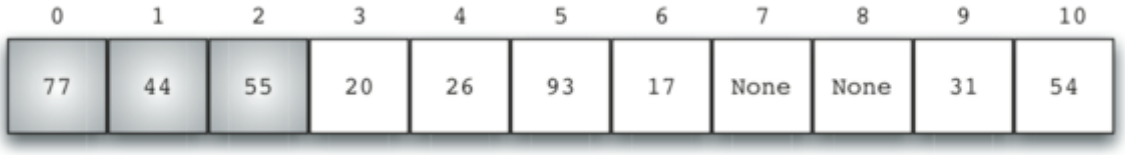
同样地，55本应放在槽0，但是不得不放在了槽2，因为槽2是最近的1个空槽。最后20的哈希值为9，但是槽9已被使用，只能进行线性探测。逐次访问10，0，1和2，最终找到的是槽3。



../_images/linearprobing1.png

若使用空槽寻址和线性探测技术来构建哈希表，那么也必须使用相同的方法来搜索元素。假如欲查找项为93，当计算出其哈希值时，得到的是5。查看槽5，发现其中就是93，因此返回结果为True。那如果查找的是20呢？其哈希值为9，但槽9中存放的是31，然而却并不能直接返回False，因为有存在冲突的可能性。现在不得不进行顺序搜索了，从槽10开始，直到找到元素20或者找到1个空槽。

线性探测的一个缺点就是它会导致**聚集（clustering）**趋势，元素在哈希表中会变得集中起来。这意味着，如果同一个哈希值出现了大量冲突，基于线性探测处理的话，其周边会有很多槽被填充。这会对其它正在插入的元素产生影响，就像之前尝试将20加入哈希表时那样子，必须跳过哈希值为0的那一簇元素才能找到空槽。这种集中（？cluster）如图9所示。



../_images/clustering.png

处理这种聚类的方法之一是将线性探测技术进行改进，使其跳跃式地进行空槽探测，而不是顺序遍历，这样便使得冲突项的分布更加平衡，从而潜在性地减少可能出现的集中现象。图10演示了使用“加3”探测法进行的冲突处理的结果，即冲突出现时每次跳过两个槽来进行遍历，直到找到空槽。

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

../_images/linearprobing2.png

这种为冲突项寻找空槽的过程被称为**再哈希（rehashing）**。使用简单的线性探测的话，再哈希函数为：

$newhashvalue = rehash(oldhashvalue)$ ，$rehash(pos) = (pos + 1)\%sizeoftable$

“加3”再哈希函数可以写作：

$rehash(pos) = (pos + 3)\%sizeoftable$

通用形式为：

$rehash(pos) = (pos + skip)\%sizeoftable$

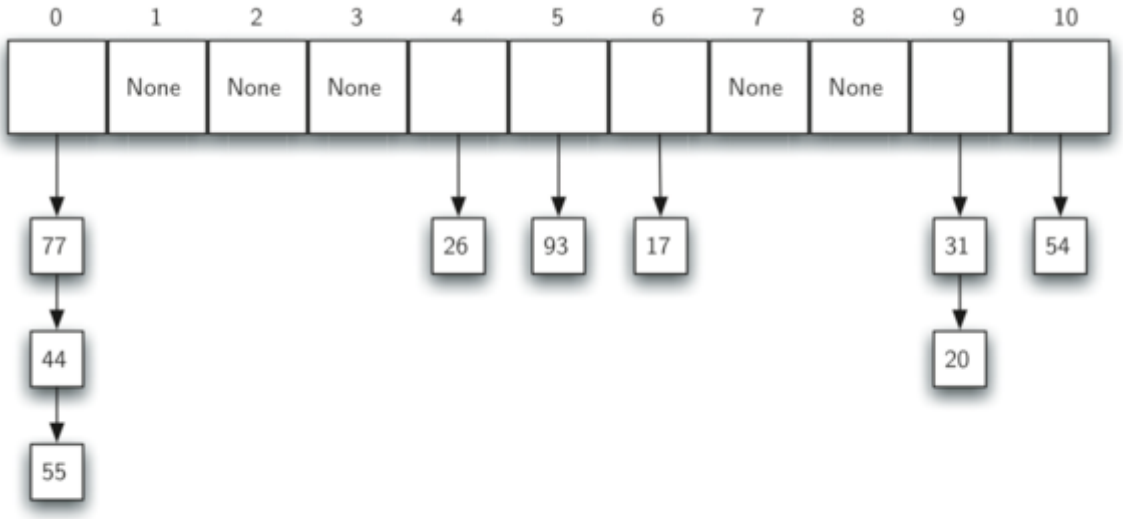
值得注意的是，skip的大小必须使得哈希表中的所有槽最终都可以被遍历。否则，哈希表的一部分槽将被闲置。为了保证这一点，通常建议将哈希表大小设置为质数。这就是在本例中使用11的原因。

线性探测法的一种变式是所谓的**二次探测法（quadratic probing）**。使用的再哈希函数每次增加哈希值依次为1，3，5，7，9等，而不是某个常数值。也就是说，如果第1个哈希值为h，则接下来值为h+1,h+4,h+9等。换言之，二次探测法的间隔是连续的完全平方数。图11是使用这种方法的一个例子。

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

../_images/quadratic.png

处理冲突问题的另一个可选方法是允许每个槽保存指向容器或者链的引用地址。**链（chaning）**使得在哈希表的同一个位置中可以同时存在许多元素。当出现冲突时，这些元素仍然放在哈希表中的正确的位置。随着越来越多的元素加入到同一个位置，在该容器中搜索目标元素的难度也就越来越大。图12对此进行了演示。



../_images/chaining.png

当搜索某个元素时，先用哈希函数求出该元素在哈希表中的槽位，由于每个槽都是个容器，可以使用搜索技术来确认该元素是否存在。这种技术的优势在于平均来看，每个槽位中的元素数应该比原始容器的总元素数少得多，因此搜索也许会更快一些。在本节末，将研究对哈希的分析。

5.5.3 实现抽象数据类型：Map

字典是Python最有用的数据类型之一。回忆一下，字典就是一种可以存储键-值对的关联性数据类型。键用于查找对应的数据值。这种思想常被称为**映射（map）**。

抽象数据类型Map定义如下：其结构是一种无序容器，用来存放键-值组合。map中的键都是唯一的以保证键-值之间是一一对应的。可用的操作如下：

- Map()生成新的，空map。返回空map容器。
- put(key,val)将一组新的键-值对放入map中，如果键已在map中，则将旧值替换为新值。
- get(key)给定键，返回map中存储的值或者None。
- del 以del map[key]语句的形式删除键-值对。
- len() 返回map中存储的键-值对数量。
- in 以key in map语句的形式返回True或者False。

字典的好处之一在于，给定键便可以极快速地找到对应数据值。为了获得这种快速查找功能，必须给以合适的实现。使用采用顺序搜索或者二分搜索的列表是可以的，然而更优的选择是使用哈希表，因为它的搜索操作可以实现O(1)的复杂度。

在代码2中，使用了两个列表来创建HashTable类用于实现Map抽象数据类型。其中的1个列表slots用来保存键，而另一个列表data用来保存数据值。当查找键的时候，数据表中的对应位置即保存了关联的数据。将键表使用前文的方法处理为哈希表。可以看到该哈希表的初始大小被设定为了11。虽然这个值可以是任意的，但值得注意的是，使用质数的话可以尽可能地提高冲突处理算法的效率。

代码2

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

这里的hashfunction实现的是简单的求余法。冲突处理采用的结合了“加1”再哈希函数的线性探测法。put函数（代码3）假定最终必能找到某个空槽除非该键已经存在于self.slots中了。据此，它计算出当前元素的哈希值，若该槽不为空，则迭代调用rehash函数直到直到空槽。如果该键已存在于某非空槽中，那么其中的数据值被替换为新的数据值。处理没有空槽的情况就作为练习了。

代码3

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

  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,len(self.slots))
      while self.slots[nextslot] != None and \
                      self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot,len(self.slots))

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

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

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

类似地，get函数（如代码4所示）首先计算初始哈希值。若该值不在该初始值对应的槽，便使用rehash函数来定位下一个可能的位置。注意行15通过检测是否回到了起始槽来保证搜索操作最终是会结束的。如果回到了初始位置，那说明已经遍历尽了所有可能的槽位，而该项必然不存在于该表中。

HashTable类的最后一种方法提供了额外的字典功能。重写__getitem__和__setitem__方法使得其可以使用"[]"来访问数据。这意味着若生成了HashTable，便可以使用熟悉的索引操作符了。剩下的方法就作为练习了。

代码4

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

  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,len(self.slots))
       if position == startslot:
           stop = True
  return data

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

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

下面的内容展示了HashTable类的使用。首先实例化一个哈希表并存入一些数据，键为整数，而值为字符串。

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

接下来对哈希表中的项作访问和修改操作。可以看到，键为20的对应数据值被替换了。

```
>>> H[20]
'chicken'
>>> H[17]
'tiger'
>>> H[20]='duck'
>>> H[20]
'duck'
>>> H.data
['bird', 'goat', 'pig', 'duck', 'dog', 'lion',
 'tiger', None, None, 'cow', 'cat']
>> print(H[99])
None
```

完整的代码如可执行代码1所示。

可执行代码1:完整的HashTable示例

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

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

      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,len(self.slots))
          while self.slots[nextslot] != None and \
                          self.slots[nextslot] != key:
            nextslot = self.rehash(nextslot,len(self.slots))

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

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

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

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

      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,len(self.slots))
           if position == startslot:
               stop = True
      return data

    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])
```

5.5.4 哈希法的分析

之前说过，在最好的情况下，哈希法的时间复杂度为O(1)。然而，由于冲突的出现，比较操作的次数并没有那么简单了。虽然对哈希法的完整分析超出了本章的范围，但这里会给出一些著名的估测搜索操作中所进行的对比次数的方法。

在对哈希表进行分析时，最重要的信息就是其负载因子λ。从概念来看，若λ小，那么冲突的概率便较低，说明各项更有可能位于正确的位置；若λ大，则说明哈希表近于被填满，发生冲突的可能性更高，此时冲突处理会变得更加复杂，需要更多的比较操作来找到空槽。若使用数据链技术，冲突数上升意味着每条链上存放的项增加。

和先前一样，成功和不成功的搜索操作其计算结果是不一样的。对于使用线性探测实现空槽寻址且成功完成的搜索，平均比较次数大约是$\frac{1}{2}(1+\frac{1}{1-\lambda})$，而不成功的则是$\frac{1}{2}(1+(\frac{1}{1-\lambda})^2)$。如果使用数据链，成功搜索是$1+\frac{1}{\lambda}$，不成功时则为λ。

5.6 排序

排序是将容器中的元素按照某种顺序放置的过程。比如说，单词列表可以是按字母排序也可以是按长度排序；城市列表可以按人口，面积或者邮编来排序。在前文中，读者已经看到有很多算法可以从排序后的列表中获得增益。

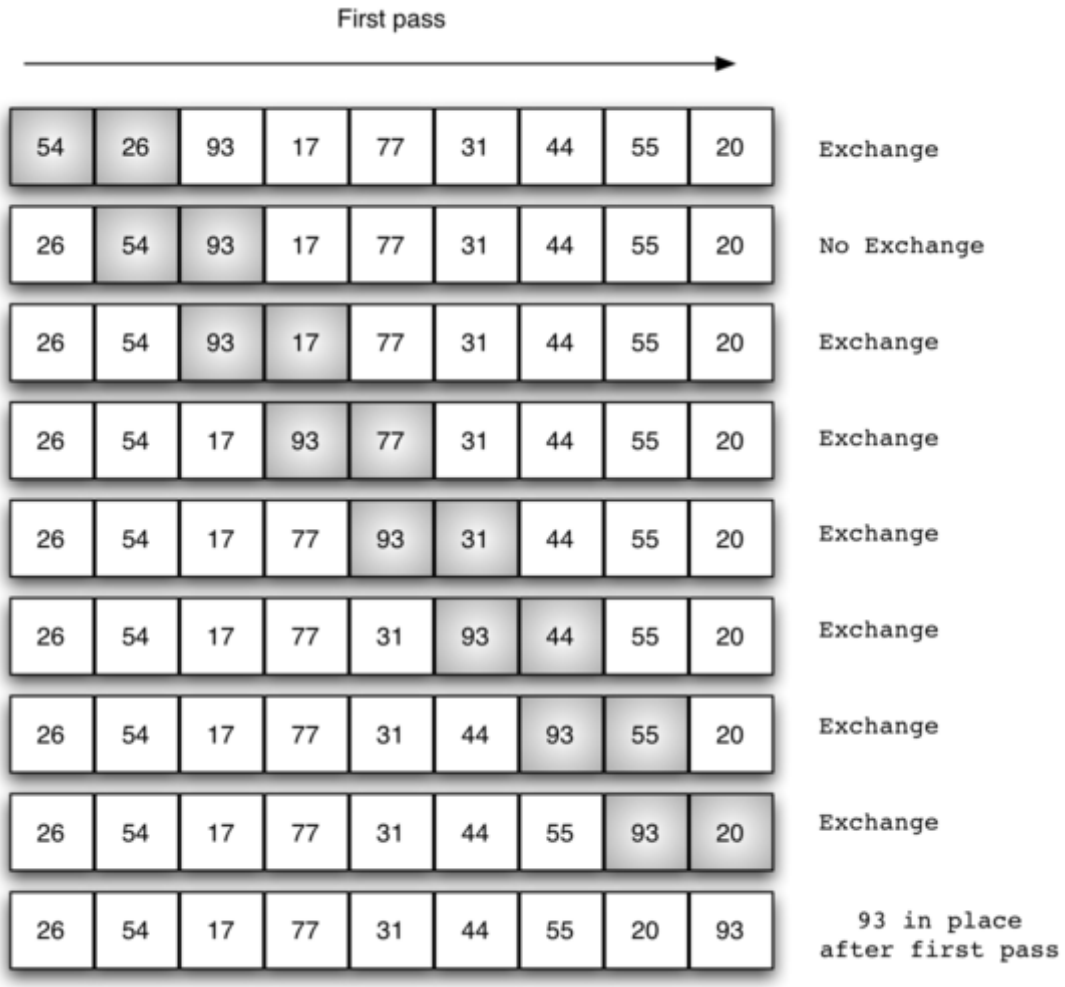
现在已经发展出了很多排序算法，并对它们做了分析，这表面排序实际上是计算机科学的重要研究领域。在海量元素中进行排序需要大量计算资源。跟搜索一样， 排序算法的效率跟元素规模有直接关系。对于小容器，使用复杂的排序算法可能并不划算，其造成的负载可能太大了。然而，对于大容器，应当尽可能地使用各种优化手段。本节将研究几种排序技术并且对它们的运行时间做一个对比。

在研究具体的算法之前，读者应当先思考一下用于分析排序过程的运算逻辑。首先，它应当是对数值的比较，判断哪个更大或者更小。为了对容器进行排序，必须要有系统性的方法来对这种值进行比较，检测是否符合排序规则。比较的总数应是排序过程性能的判定标准。第二，当该值的位置并不在正确的位置上时，需要对其进行调整。这种交换是一个开销很大的操作，交换的总数对于评价算法的性能也应是重要指标。

5.7 冒泡算法

冒泡排序对列表进行多次遍历。它将相邻项进行对比，并将顺序错误的进行位置交换。每对列表进行一次遍历，比如第n次遍历，则将第n大的值放在正确的位置上。事实上，每一项都是像冒泡一样跑到了正确的位置上。

冒泡排序的第1次遍历如图1所示，阴影表示正在进行比较以确定顺序是否正确的项。如果列表中有n项，那么在第1次遍历时便需要对(n-1)个元素组合进行比较。值得注意的是，一旦出现列表中的最大值，它将会一直移动，直到本次遍历结束。



../_images/bubblepass.png

在第2次遍历的开始，最大值位于正确的位置，因此还有n-1项需要排序，即(n-2)对。由于第n次遍历会将第n大的元素放在正确的位置，总的遍历次数应当是n-1。在完成n-1次遍历后，最小项必然会在正确的位置而不需要再作处理了。可执行代码1给出了完整的bubbleSort函数。它接收列表作为参数，并通过交换元素来作必要的调整。

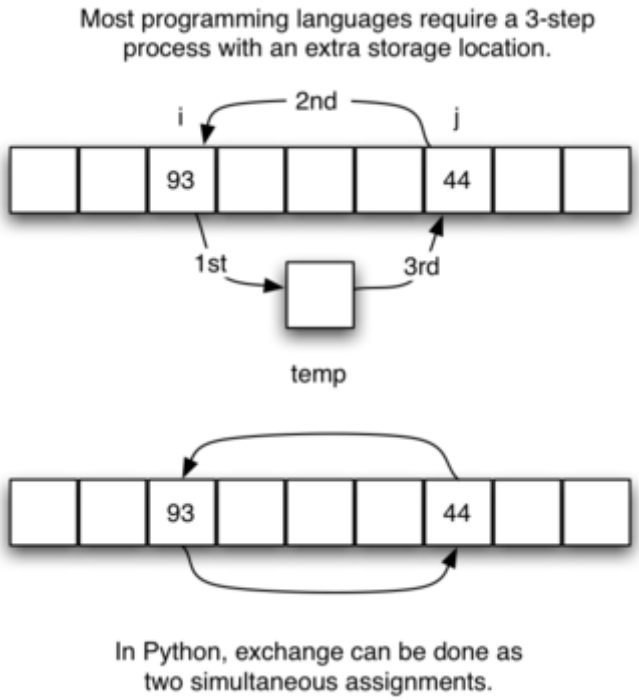
数据的互换操作，有时被称为swap（交换），在python中跟其它大多数语言不太一样。一般来说，对列表中的两个元素进行交换需要1个暂存的空间（额外的内存空间）。比如这样的代码：

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

会将列表的第i项和第j项进行交换。如果没有暂存的话，其中的一个值会被覆盖掉。

在Python中，同时进行复制是有可能的。语句a, b = b, a会在同时完成两个赋值语句，如图2所示。

可执行代码1的行5-7将i和i+1项使用的是3步法，实际上也可以用同时赋值语句来完成。



../_images/swap.png

下面的可执行代码以上文的列表为例演示了其运行。

可执行代码1：泡沫排序

```
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)
```

为了分析泡沫排序，必须意识到无论初始列表中各元素是怎样放置的，对于长度为n的列表，都需要进行n-1次遍历。表1给出了每次遍历的对比数，总的对比数是前n-1个整数之和。前n个整数之和为$\frac{1}{2}n^2+\frac{1}{2}n$，则前n-1个整数之和为$\frac{1}{2}n^2-\frac{1}{2}n$，比较操作的时间复杂度仍然为$O(n^2)$。在最好情况下，若列表已经排好序，那么便不需要进行交换，然而，在最差的情况下，每次对比都必须进行交换。平均来说，交换次数是比较次数的一半。

Pass	Comparisons

1	n–1
2	n–2
3	n–3
...	...
n–1	1

冒泡排序算法通常被认为是效率最低的排序算法，因为在确定最终位置前必须不断进行元素交换，这些不必要的交换开销是非常巨大的。然而，因为排序算法遍历了列表整个乱序部分，因此它可以实现很多其它算法无法办到的事情。特别是，对于没有发生交换的遍历中，便可以确定列表已经排好序了。排序算法可以进行改进，在确定列表排好序过后即停止。这意味着对于只需要几次遍历的列表，排序算法也许会很快识别出其已经排好序，并且停止。可执行代码2给出了改进版，常常被称为**紧凑冒泡（short bubble）**。

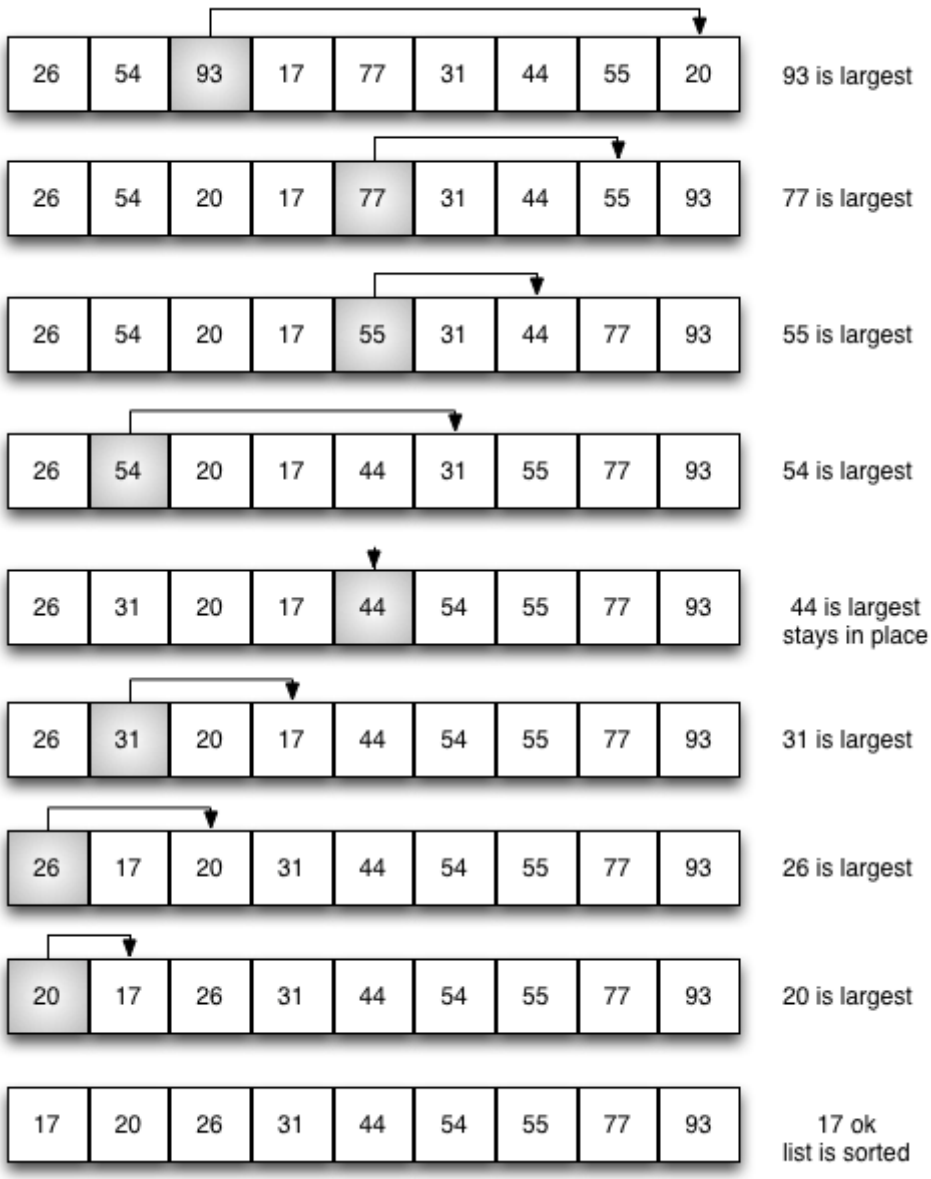
可执行代码2:紧凑冒泡法

```
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)
```

5.8 选择排序

选择排序通过在每次遍历时仅作1次交换来提升了冒泡排序的效率。为了实现此，选择排序在每次遍历时都搜索最大值，并且在完成遍历后，将其放在合适的位置。和冒泡排序一样，在第1此遍历后，全局最大项被放在了正确的位置。在第2次遍历后，第2大的项也在正确位置了。重复进行该过程，一共需要n-1次遍历来对n个元素排序。

图3给出了整个排序过程。在每次遍历时，未排序的剩余项中的最大项都被选中，然后放在正确的位置。第1次遍历排好了93，第2次77，第3次55，依次类推。该函数如可执行代码1所示。



../_images/selectionsortnew.png

可执行代码1:选择排序

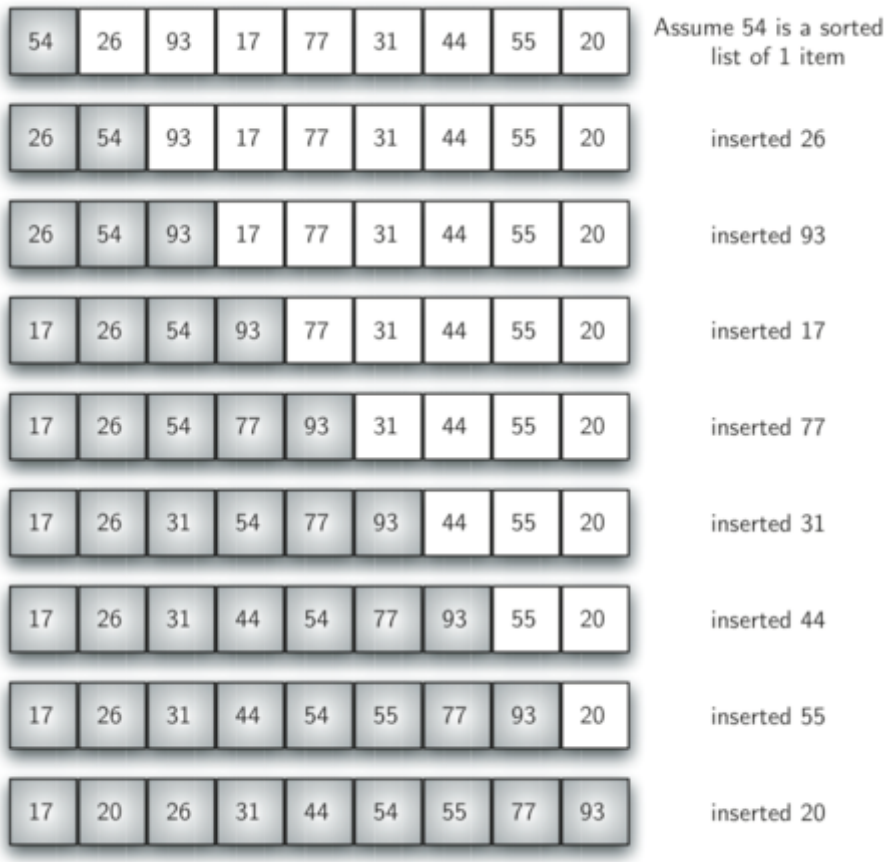
```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

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

```
        alist[positionOfMax] = temp

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

可以看出，选择算法进行的对比次数跟排序算法一样，因此复杂度为$O(n^2)$。然而，由于交换次数的下降，选择算法运行速度要高很多。实际上，对上面的例子，排序算法要作20次交换，而选择排序仅需要8次。

5.9 插入排序

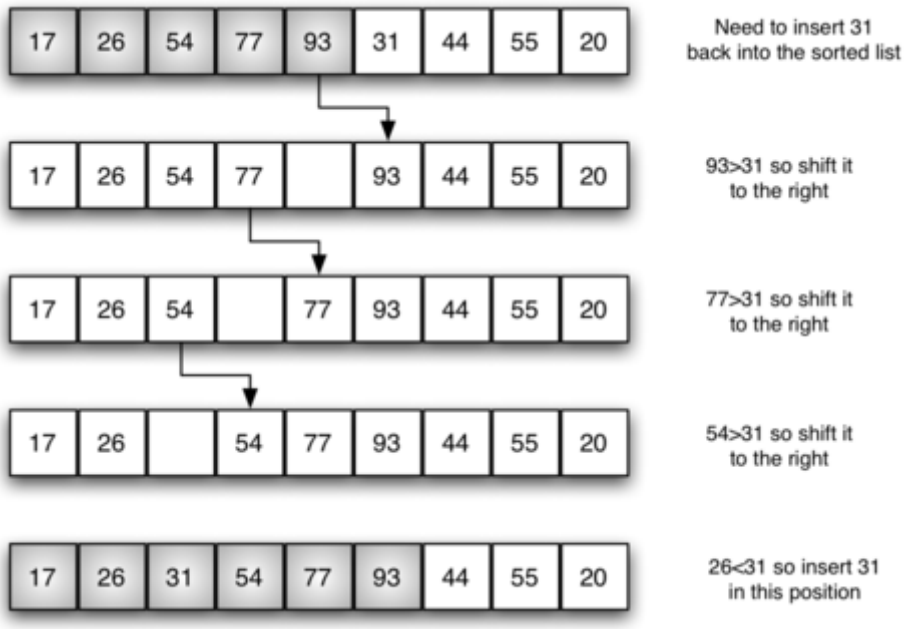
插入排序，虽然复杂度也是$O(n^2)$，运行方式有些不一样。它在列表靠前的位置维护一个排好序的子列表。每个新的元素都插入到前部的子序列，这样子序列每次长度都增加1。插入排序的过程如图4所示。阴影表示每次遍历时的子列表。



../_images/insertionsort.png

在程序开始的时候，假设子列表是由索引为0的元素构成的排好序的列表。每次从第1项到n-1项的遍历时，当前项都将于已经位于排好序的子序列中的各项进行比较，将对比出来较大的元素向右移，若遇到较小的元素或者到达了子列表的尽头，则可以直接将当前项插入。

第5步的详细过程如图5所示。在该算法的这一步，已经有一个由17，26，54，77，93构成的排好序的子列表了。现在想要将31插入已经排好序的子序列中。首先跟93进行比较，由于93更大，因此将其向右移动1位，77和54也是同理。当遇到26时，便不用再右移了，将31放在空位上即可。现在便得到了6元素的子列表。



../_images/insertionpass.png

从insertionSort的实现（可执行代码1）可以看出，它也需要n-1次遍历来对n项进行排序。迭代从位置1开始，并且遍历剩下的n-1个位置，因为要将数据放入已经排好序的子列表中。行8进行了换位操作，即将列表中的值后移1个位置，在其前部留出空位给将要插入的数据使用。需要注意的是，这并不是前面算法中的完整交换。

插入排序的最大比较次数为前n-1个整数之和。同样地，复杂度也是$O(n^2)$。然而，在最好情况下，每次遍历只需要1次赋值操作。

关于“移位”和“交换”还有一点很重要。总的来说，移位操作消耗的计算资源大概是交换操作的1/3，因为其中值进行了1次赋值。在性能评分测试中，插入排序往往表现非常好。

可执行代码1:插入排序

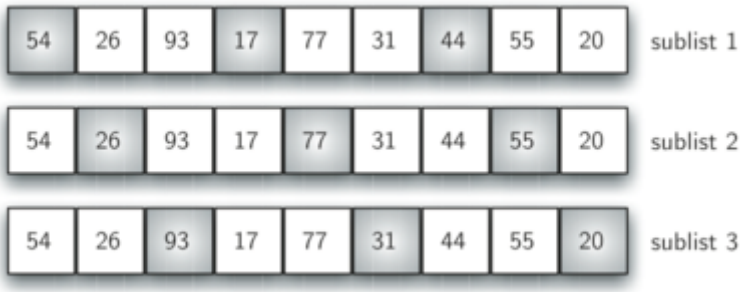
```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

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

```
        alist[position]=currentvalue

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

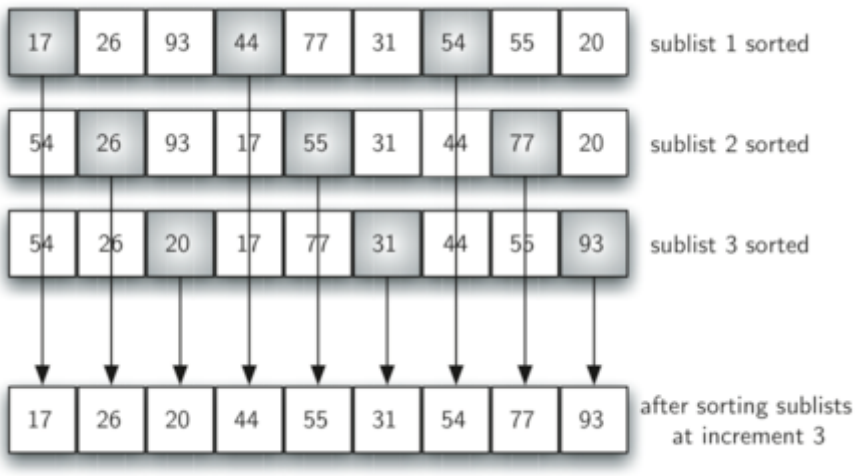
5.10 希尔排序

希尔排序（shell sort），有时也被称为**减小增量排序**，对插入排序实现了改进，即将原列表切分为许多小子列表，对其中的每个小列表都使用插入排序。划分子列表的唯一性方法是希尔排序的关键。希尔算法并不是将列表拆分为连续的元素构成的子序列，而是使用了一个增量i，有时也被称为**间隔（gap）**，将间隔为i的项用来构建子列表。

如图6所示，该列表有9项。如果以3为间隔来选取元素进行划分，则有3个子列表，其中的每一个都用插入排序处理。在排序完成后，得到如图7所示的列表，虽然改列表还没有被完全排序，但有趣的是，将各子列表排序后，各元素都更加靠近其正确位置了。

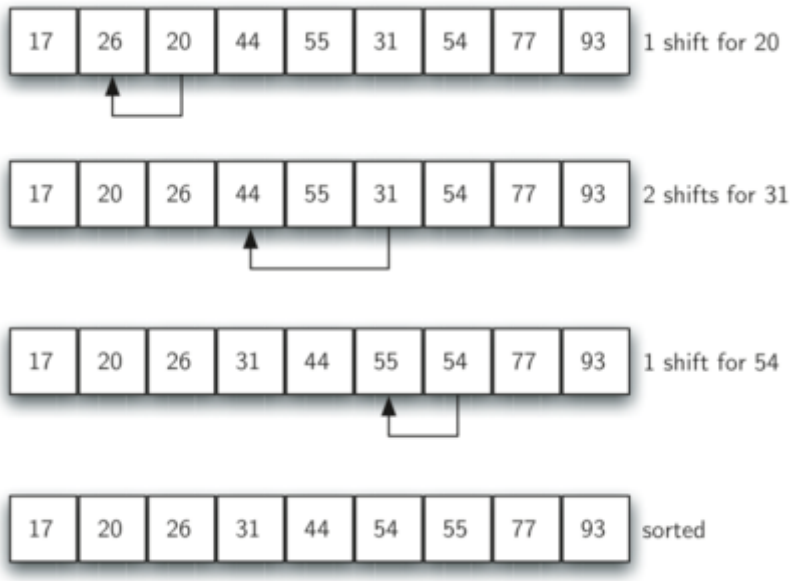


../_images/shellsortA.png



../_images/shellsortB.png

图8是使用间隔1的插入排序，换句话说，也就是标准的插入排序。注意，通过对子序列进行排序，减少了最终排好序所需要的位移操作次数。对于本例，仅需要4次位移就可以完成该过程。



../_images/shellsortC.png

之前说过，间隔的选择方式是希尔算法的关键。函数代码如可执行代码1所示，它使用了1系列间隔数。在本例中，开始的使用$\frac{n}{2}$的子列表，下一步则对$\frac{n}{4}$的子列表排序。最终使用基本的插入排序来完成单元素列表的排序。使用间隔时，该例子的第1次形成的子列表如图9所示（有4个）。

../_images/shellsortD.png

下面对shellSort函数的调用给出了每个间隔值下完成了部分排序的列表，最后的排序间隔为1。

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:
        for startposition in range(sublistcount):
```

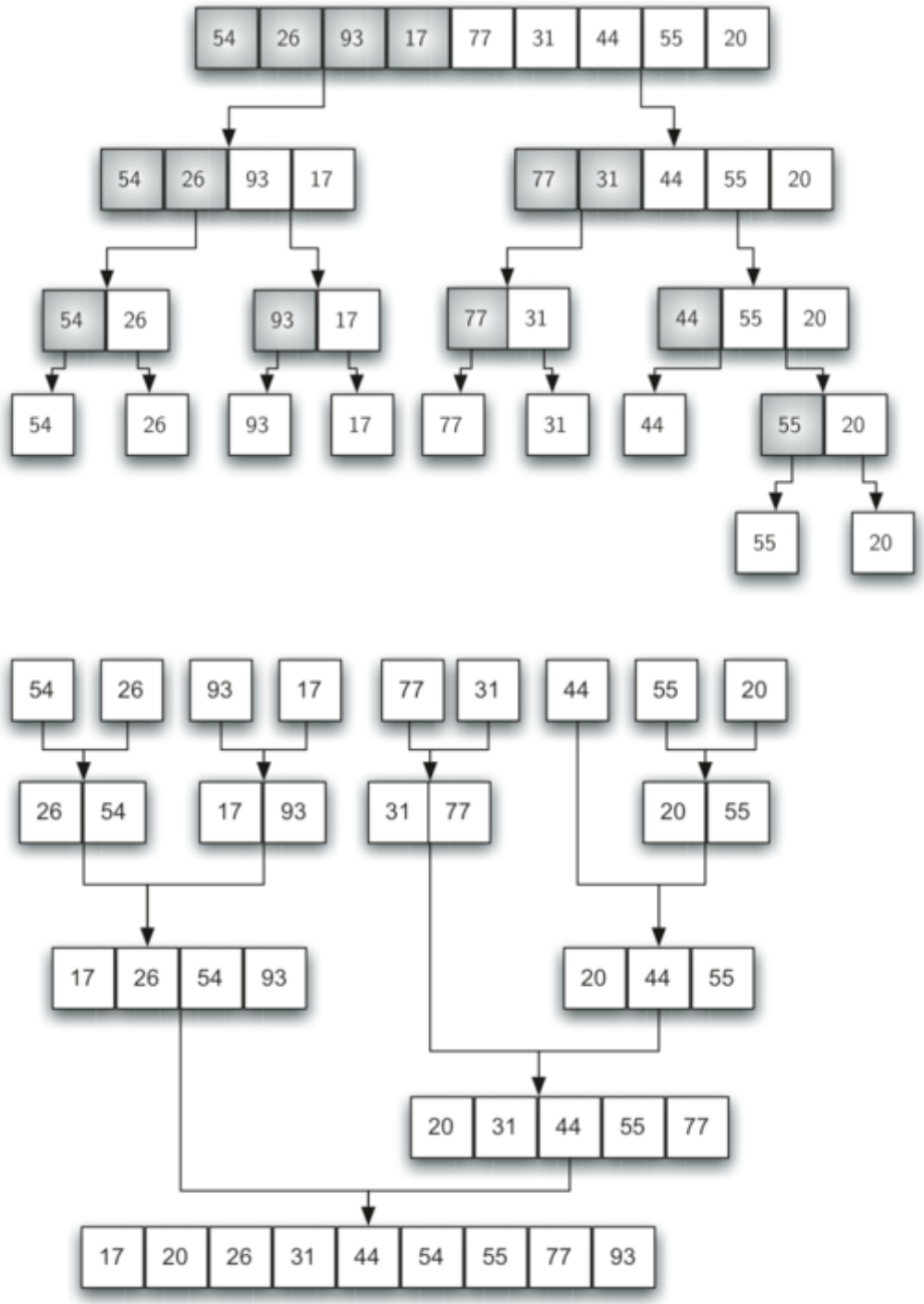
```
gapInsertionSort(alist,startposition,sublistcount)
        print("After increments of size",sublistcount,"The list
is",alist)

      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

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

乍一看，读者可能会认为这并不比插入排序好，因为它在最后也必须要进行一次插入排序。实际上，最后一次插入排序并不需要做很多次比较或者位移操作。因为该列表已经在子列表中实现了部分排序。换句话说，每一步都比前一次更有序。这可以保证最后一步的速度是非常快的。

对希尔排序的综合分析超出了本书范围，但根据前文所述的步骤，可以判断其应该大致位于$O(n)$和$O(n^2)$间。使用某些间隔值，时间复杂度会是$O(n^2)$，而改变间隔值，比如说使用2^k-1(1,3,7,15,31...），可以达到$O(n^{\frac{3}{2}})$。

5.11 归并排序

现在开始研究使用分而治之策略来提示排序算法的性能。第一个算法是**归并排序（merge sort）**，它是一种递归算法，不断地将列表切半。若列表为空或者仅有1个元素，它便实现了排序（约束条件）。若列表有超过2个元素，则将其拆分，并对两部分递归地调用归并排序。一旦两侧都完成了，便执行被称为**归并（merge）**的关键操作。归并即是将两个较小的列表合并为1个有序的新列表的过程。图10以常用的例子演示了mergeSort的切割过程。图11给出了合并过程。



mergeSort函数如可执行代码1所示，首先给出约束条件。如果列表小于等于0，则列表已经有序，不再需要作进一步处理了。反之，如果大雨1，则使用Python的slice操作来提取左侧和右侧两部分。值得注意的是，列表元素数可能并不是偶数，但那并不影响，最大的长度差也不会超过1。

```
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=0
        j=0
        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)

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

mergeSort函数在左半侧和右半侧调用时，会假定这两部分都已排好序了。剩下的函数（11-31行）用于将较小的有序列表合并为较大的有序列表。注意，归并操作是将有序列表中的最小元素一个一个地放入到原始列表中（alist）。

mergeSort函数中增加了一个print语句（行2）用来在每次调用时显示正在进行排序的列表的内容。在32行也有个print语句，它是用来展示归并过程的。

为了分析mergeSort函数，需要先考虑下实现它的两个不同的步骤。首先，将列表分半。在二分搜索中提到过，对列表进行分半操作总共需要logn的时间复杂度，其中n是列表的长度。第二个步骤是归并。列表中的每一项最后都将被处理然后放进排好序的列表中，因此归并操作需要进行n次操作。结果是，拆分总共需要进行logn次，每1次又需要进行n次操作，那么总共时间复杂度就是nlogn了。

回想一下，切片操作的复杂度是O(k)其中k为切片长度。为了保证mergeSort是O(nlogn）的，必须要去掉切片操作符。同样的，传入起始和结束索引值即可，这就作为练习了。

值得注意的是，mergeSort函数需要额外的空间来保存切片操作获取的两半列表。这种额外消耗的空间对于大列表来说开销很大，并且使得这种排序方法在大规模数据集上不可用。

5.12 快速排序

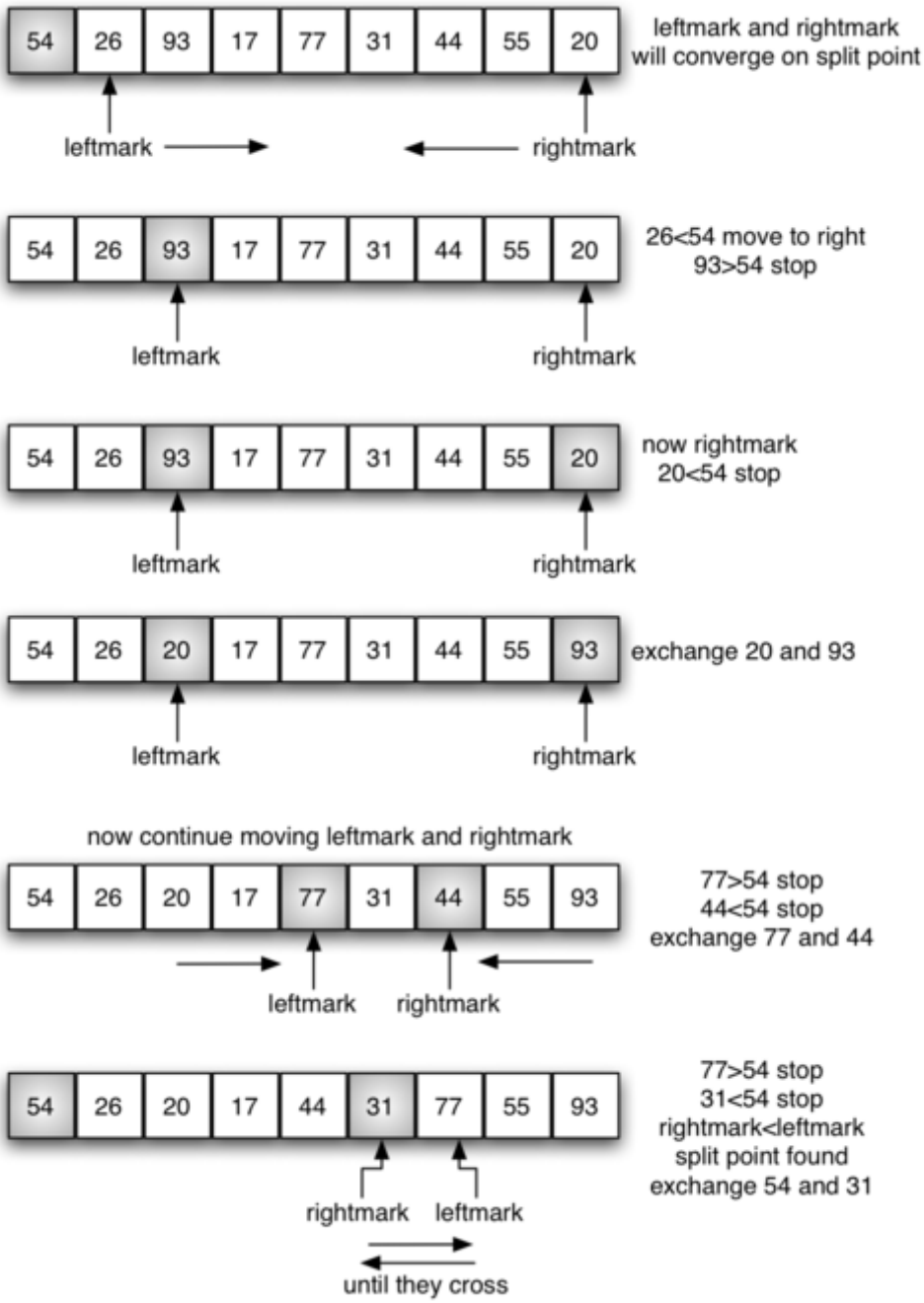
快速排序使用分而治之策略，因而也获得了归并排序的优势，但却并不消耗额外的空间。然而作为牺牲，列表也许不能以对半的形式进行划分了，出现这种情况时，会导致性能的下降。

快速排序算法先选择一个值，称之为**基准值（pivot value）**，虽然有很多种选择方法，这里就直接使用列表中的第1个好了。基准值的意义在于对列表进行划分。在最终排好序的列表中，该基准值所在的位置通常被称为**分割点（split point）**，用于把列表分为两个部分，然后再分别调用快速排序函数。

如图12所示，54被作为基准值，读者可能已经对这个例子很熟悉了，1眼就可以看出54的正确位置是在31上。接下来进行**分区**过程，找到其对应的基准点，并且同时将列表中其它项放在正确的一侧，即比该基准值大的一侧或者比其小的一侧。

../_images/firstsplit.png

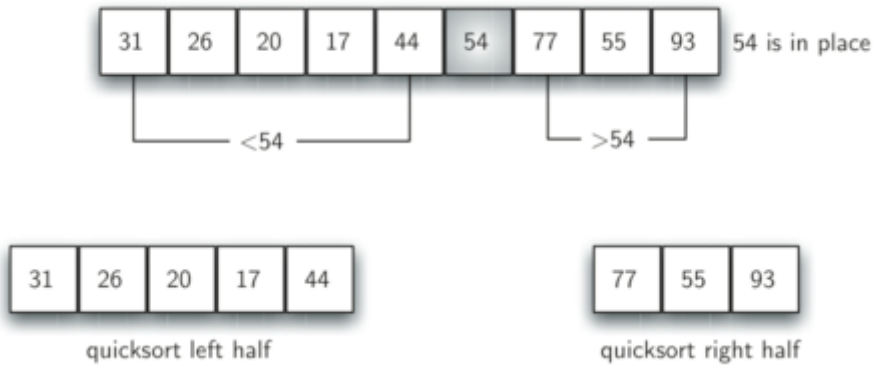
首先设置两个位置标识，1个为leftmark，另1个为rightmark，分别位于列表中剩余项的开始和结束位置（也就是图13中的位置1和位置8）。分区的目标是将位于分割点错误一侧的值，并且找到分割点。图13给出了定位54的位置的过程。



../_images/partitionA.png

先将leftmark右移，直到发现比基准值大的数值。然后将rightmark左移，直到发现比基准值小的值。此时便发现了两个错放于最终分割点两侧的值，本例中就是93和20。现在将这两项进行交换，并重复以上步骤。

当出现rightmark比leftmark小的时候，停止，现在rightmark的位置就是分割点了。基准值与当前分割点上的值进行交换，现在基准值便位于正确的位置了（图14）。此外，分割点左侧的值均比基准值小，而右侧的都比基准值大。现在列表可以在分割点进行画个，然后分别对两部分递归调用快速排序法。



../_images/partitionB.png

可执行代码1中的quickSort函数递归调用了函数quickSortHelper。quickSortHelper的约束条件同归并排序一样：若列表长度小于等于1，则该列表已完成排序。若大于1，则将其分而治之。partition函数实现了前文所述的过程。

```
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)


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

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

现在来分析quickSort函数，注意对于长度为n的函数，如果分割点总是出现在列表的重点，那么会进行logn次划分。为了找到分割点，n项中的每一个都要和基准点进行对比。因此最终结果是nlogn。此外，在合并过程中，也不用额外的内存空间了。

不幸的是，在最坏的情况下，分割点并不在中间而是位于极偏左或偏右，产生极不均匀的分割。在这种情况下，将n个元素的列表分为0项和n-1项的的列表，然后又将n-1项的那个列表分为0项或n-2项的列表，以此类推。最终所有的递归调用都确实计算了，导致复杂度是$O(n^2)$。

在前文中提到，有许多办法来选择基准值。尤其是一种名叫**三点取中（median of three）**的技术，通过它可以降低出现不平衡分割的可能性。在选择临界值时，先对列表的第1个，中间以及最后1个元素进行研究。以本例来说，即是54，77，20。选择选用其中间的值，54，将其作为基准值（恰好就是原本使用的那个）。这种方法主要是用来避免列表的第1项不偏向于列表中间值的情况，这时选用这三者中选择一个更加偏中的值。当列表本身已经有一定程度的排序时，这种方法尤其有用。它的实现就作为练习了。

5.13 总结

- 顺序搜索对于有序和无序列表的复杂度都是O(n)。
- 对有序列表的二分搜索在最坏情况下是O(logn）。
- 哈希表可以提供O(1)的时间复杂度。
- 冒泡排序，选择排序，以及插入排序都是$O(n^2)$的算法。
- 希尔法利用间隔子列表对插入排序实现了改进，其复杂度是O(n)与O(n^2)之间。
- 归并排序的复杂度是O(nlogn），但是在合并过程中需要额外的空间。

- 快速排序的时间复杂度是O(nlogn），但如果分割点不在列表中间的话，可能会导致$O(n^2)$的复杂度。它并不需要额外的空间。