排序与查找

目标

- 理解并实现顺序查找和二分查找；
- 理解哈希表用于查找的技术；
- 介绍映射的抽象数据类型；
- 用哈希表实现映射的抽象数据类型；
- 理解并实现选择排序，冒泡排序，合并排序，快速排序，插入排序和shell排序。

查找

在python里，可使用in操作符查询某数据是否在列表里：

In [3]:
```
15 in [3, 5, 2, 4, 1]
```

Out[3]: False

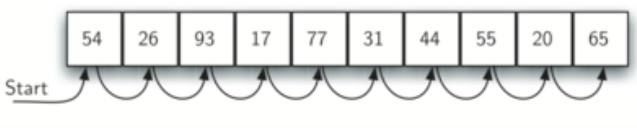
In [3]:
```
3 in [3,5,2,4,1]
```

Out[3]: True

虽然这样写很简单，不过这个算法后面的处理过程我们还是要学习，查找有很多种方法，我们感兴趣的是算法以及算法之间的比较。

顺序查找

从列表的第一个元素开始，逐个检查，要么找到，要么全部检查完还没找到。



In [5]:
```
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            return True
        else:
            pos += 1
    return found
```

In [8]:
```
testlist = [1, 2, 32, 8, 9, 12, 4]
print(sequentialSearch(testlist, 32))
print(sequentialSearch(testlist, 21))

```

```
True
False
```

顺序查找的性能分析

对长度为n的无序列表进行查找时，可能有三种情形：

- 最好情况：第一个元素就是我们要找的，只需1次比较；
- 最坏情况：全部检查完还没找到，需要n次比较；
- 平均情况：

  定义平均查找长度（Averaged Search Length, ASL）：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

  其中

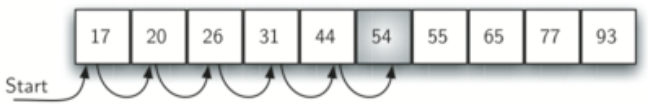
  - p_i表示查找表中第i个元素的概率
  - c_i表示找到第i个元素时已经比较过的次数

假设每个元素被查找的概率相等，则列表的平均查找长度为

$$ASL = \frac{1}{n}(1 + 2 + \cdots + n) = \frac{n+1}{2}$$

对有序列表（假设是升序排列）进行顺序查找时，

- 若元素在列表中，比较次数跟无序列表一样。
- 若元素不在列表中，性能上会稍有提高。



查找50

让50与列表中的元素逐个进行比较，当比较到54时你会发现，54以及其后面的元素都会大于50！

也就是说，程序此时可以停止遍历。

In [13]:
```
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 += 1
    return found
```

In [14]:
```
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

```
False
True
```

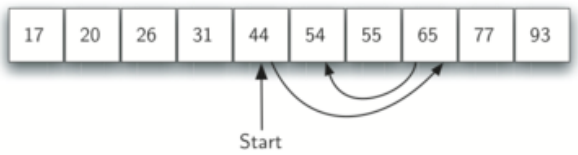
折半查找

- 处理对象：有序列表（如升序）
- 算法原理

  从中间元素开始比较，对于要查找的值，

  - 若等于中间元素，则查找成功；
  - 若小于中间元素，则在前半部分继续查找；
  - 若大于中间元素，则在后半部分继续查找。

  不断重复，知道查找成功或者查找失败为止。



折半查找54

In [21]:
```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

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

In [22]:
```
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42, 50]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

```
False
True
```

折半查找算法采用的是"分而治之（ divide and conquer ）"策略。

分而治之的基本思想是将一个规模为n的问题分解为k个规模较小的子问题，这些子问题互相独立且与原问题相同。

找出各部分的解，然后把各部分的解组合成整个问题的解。

在列表中查找时，先检查中间元素，如果小于中间元素，就在左半表中继续查找；类似地，如果大于中间元素，就在右边查找。

于是，折半查找可用递归来实现。

In [24]:
```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        mid = len(alist)b//2
        if alist[mid] == item:
            return True
        else:
            if item < alist[mid]:
                return binarySearch(alist[:mid], item)
            else:
                return binarySearch(alist[mid+1:], item)
```

In [26]:
```
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42, 50]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

```
False
True
```

哈希查找（Hash Search）

本节将创建一个数据结构（哈希表），使得查找性能提高到$O(1)$，称为哈希查找。

In [ ]:
```
要做到这样的性能，我们要知道元素的可能位置，如果每个元素就在他应该在的位置上，那么要查找的
```

哈希表 (Hash Table)

哈希表（Hash table，也叫散列表），是根据关键码值(key-value)而直接进行访问的数据结构。

也就是说，它通过把关键码值映射到表中一个位置("槽位")来访问记录，以加快查找的速度。

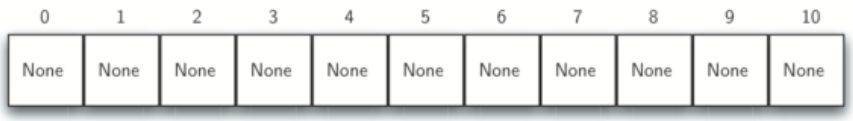
这个映射函数叫做散列函数，存放记录的数组叫做散列表。

哈希表是一种根据关键码值(key-value)而直接进行访问的数据结构。

也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。

- 哈希表中的每一个位置称为“槽位”，每个槽位都能保存一个数据元素并以一个整数命名(从0开始)。这样我们就有0号槽位，1号槽位等等。

  构建哈希表时，可以把槽位值都初始化为None。



一个大小为11的哈希表

哈希函数（Hash Function）

元素和其槽位之间的映射关系，称为哈希函数。

哈希函数接受一个元素作为参数，返回一个整数作为槽位名。

设有一个整数集合

$$54，26，93，17，77，31$$

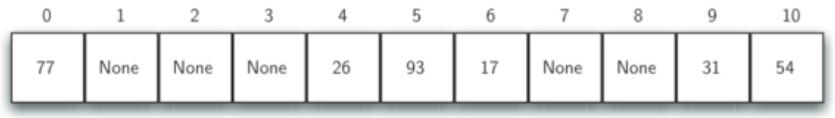
下面介绍哈希函数的几种构造方法。

1、余数法

将元素除以表的大小（11），所得余数作为哈希值。

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

一旦哈希值计算出来，就要把元素插入到哈希表中指定的位置。



11个槽位中，6个已经填满

满载因子

$$\lambda = \frac{\text{number of items}}{\text{table size}}$$

该例中，

$$\lambda = \frac{6}{11}.$$

如何查找？

1. 先用哈希函数计算出槽位值；
2. 然后到表中检查是否存在。

其时间复杂度是$O(1)$。

对于集合

$$54，26，93，17，77，31$$

若增加一个44，则其哈希值是0，跟77的哈希值一样。

这就出现了2个值对应同一个槽位的情况，称为“冲突(Collision)”。

很明显，冲突给哈希技术造成了困难。

对给定的数据集，哈希函数将每个元素映射为单个的槽位，称为“完美哈希函数”。

如果我们知道元素和集合固定不变，那么构造一个完美哈希函数也许是可能的。

坏消息是，对一个任意数据集合，没有一个系统的方法来构造完美哈希函数；

好消息是，哈希函数不完美也能提供不错的性能。

如果一定要完美的哈希函数，一种方法是做大哈希表，以保证每个元素都有自己的索引。虽然在数据不多的情况下可行，但是如果数据很大就不可行。

比如，如果数据项是8位数，这就需要十亿个槽位，如果仅仅用来保存25个学生的学号，就太浪费了。

我们的目标是：冲突最少，计算简单，分布均匀。

2、折叠法

把元素分成相等的几块（最后一块可能不相等），然后再把这些块求哈希值。

假设哈希表有11个槽位，数据项是一个电话号码 436 555 4601。折叠法的处理过程如下：

1. 按2个一组分块，然后加起来，即 $43 + 65 + 55 + 46 + 01$，得到 210。
2. 用11除210来得到槽位，即$210\%11 = 1$，故 436 555 4601 的哈希值是 1。

3、平方取中法

先计算元素的平方值，再从中提取几位数字。

例如，对元素44，先计算$44^2 = 1936$，提取中间两位 93，然后再取余数法，得到5 $(93\%11 = 5)$。

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

字符类元素的哈希函数

对于字符类元素也能创建哈希函数，如单词cat可以看成一个数字串。

In [7]:
```
1 ord('c')
```

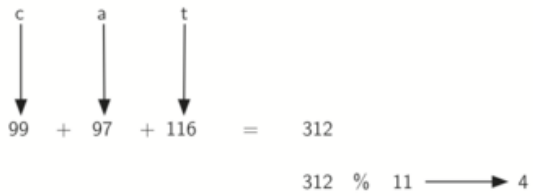
Out[7]: 99

In [11]:
```
1 ord('a')
```

Out[11]: 97

In [10]:
```
1 ord('t')
```

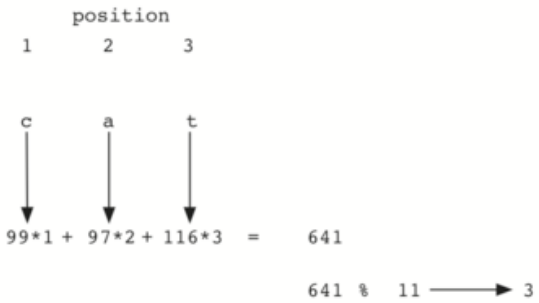
Out[10]: 116



将字母的ASCII码相加，再用余数法来计算哈希值。

In [2]:
```
1 def hash(astring, tablesize):
2     sum = 0
3     for pos in range(len(astring)):
4         sum = sum + ord(astring[pos])
5
6     return sum%tablesize
```

但对于字母相同而顺序不同的单词，用该函数算得的哈希值相等，解决办法是加上字母的位置作为权重。



原则

哈希函数必须要简单高效，不能成为计算的主要负担。

如果哈希函数太复杂，计算槽位名的时间超过了简单的顺序查找或二分查找的时间，那么哈希函数还有什么意义呢？

冲突解决

当两个元素的哈希值指向同一个槽位，就应该有办法把第二个元素放进表中，这个过程叫做“冲突解决”。

我们前面说过的，如果哈希函数是完美的，不会发生冲突。但完美无缺的事很少，所以冲突解决就成为哈希算法中的重要部分。

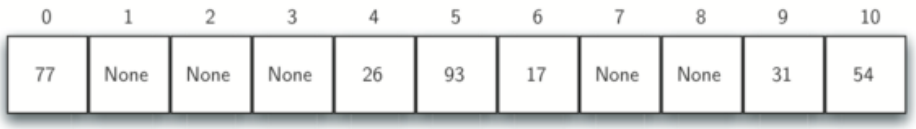
冲突解决之方法一：开放地址+线性探测

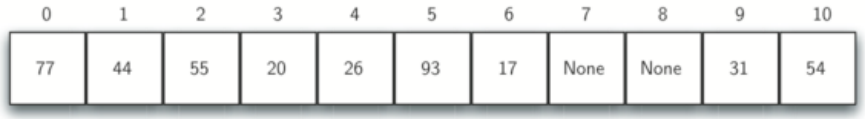
从冲突的槽位开始，顺序（循环）向前查找，直到遇到一个空闲的槽位为止。

这种方法称为开放地址法。

通过系统地访问每个槽位，我们的开放地址技术叫做线性探测。

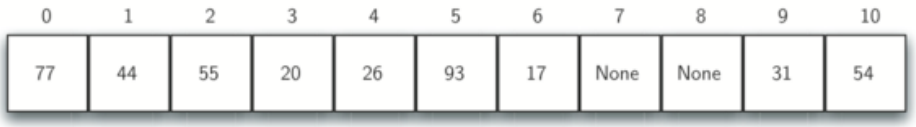
Item	Hash value
54	10
26	4
93	5
17	6
77	0
31	9





依次加入44、55、20

如何查找？



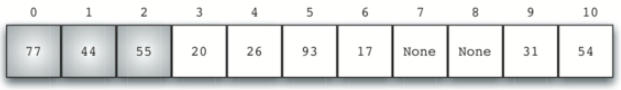
- 若查找**93**
  - 计算哈希值5
  - 查找5号槽
  - 返回True

- 若查找**20**
  - 计算哈希值9
  - 查找9号槽，因可能存在冲突，不能直接返回False。
  - 从10号槽开始顺序查找，直到找到**20**或发现空槽位。

线性探测的缺点是可能引起聚类现象，元素在表里成群出现。

因为如果在同一槽位上发生很多次冲突，附近的空槽位就会被线性探测找到并被填满，这就会影响到其他插入的元素。

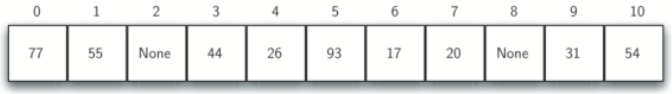
就象前面我们看到插入20的情况，20不得不跨过0后面好几个元素才找到开放地址，如下图所示。



聚类现象：插入20时，不得不跨过0号槽位后好几个槽位才找到开放地址

线性+k探测

避免聚类现象的一种方法是延伸线性探测，改变就近查找空槽的方法，而是跳过几个槽位，这样冲突的元素可以分散开来，从而潜在减少了聚类现象的可能性。



线性+3探测：冲突发生时，每隔3个位置探测一下是否空槽

再哈希（Rehash）

冲突发生后查找另一个空槽的方法称为“再哈希”。

再哈希函数的一般定义为

$$NewHashValue = rehash(OldHashValue)$$

- 用线性探测法时，

$$rehash(pos) = (pos + 1) \ \% \ sizeof table$$

- 用+3探测法时，

$$rehash(pos) = (pos + 3) \ \% \ sizeof table$$

- 一般情况下：

$$rehash(pos) = (pos + skip) \ \% \ sizeof table$$

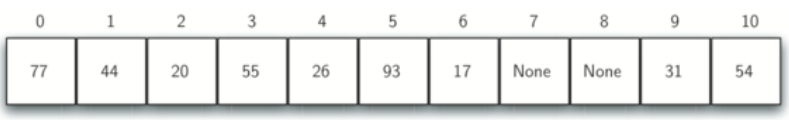
变量$skip$的选择必须保证表中所有的槽位都能访问到，否则，表中有些位置就会一直空闲。

因此，为保证$skip$取值合适，一般建议哈希表的大小是个质数，这也是为什么前面的例子中我们使用了11。

二次探测

线性探测思想的一个变种，它不再使用一个常数的“跳步”，而是用再哈希函数来逐次提高哈希值。

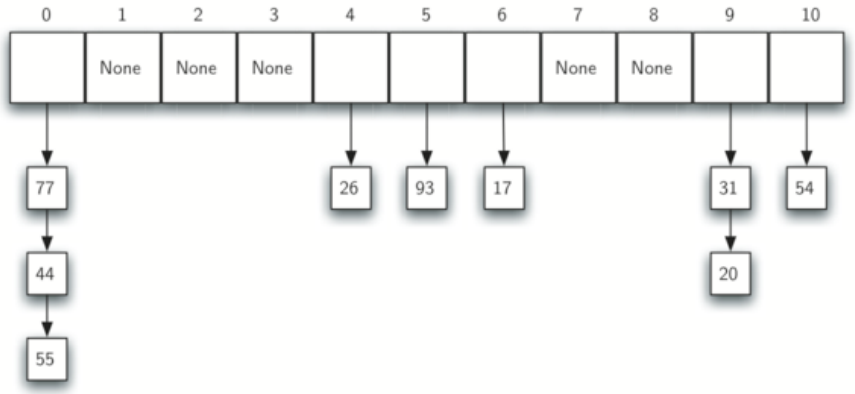
这就是说，如果第一次计算得到的哈希值是h,那么后续的值就是$h+1, h+4, h+9, h+16$等。



二次探测：使用完全平方数作为跳步

冲突解决之方法二：链表方法

让槽位上保存一个指向链地址的引用，这个方法可以在一个槽位上存在很多元素。这样当发生冲突时，元素仍然可以放在这个槽位上。但是如果越来越多的元素放置在同一槽位上，查找的难度也要增加。



使用链表方法解决冲突

当查找一个元素时，使用哈希函数计算出一个元素“应该”在的槽位值，既然每个槽位挂了一个集合，也要使用查找技术来判断是否存在。这样的好处是每个槽位上的元素数量要少很多，所以查找的效率更高。

实现map抽象数据类型

字曲是python里最有用的数据集合之一，字典是 key-value 的组合，key-value 用来查找相应的数据，我们把这种思想称为“map”。

map的抽象数据类型

map是一个无序的键值对的集合，键总是唯一的，以便建立与数据值的一一对应关系。

map的操作方法如下：

- Map( ): 创建一个空 map，返回一个空集合。
- put(key, val): 在 map 中增加新的键值对，如果键已经存在，则用新值代替旧值。
- get(key): 根据给定的键，返回对应的值，找不到时返回 None。
- del: 从 map 中删除键值对，其形式为 del map[key]。
- len(): 返回 map 中键值对的数量。
- in: 如果键在 map 中，则语句key in map返回 True，否则返回 False。

字典的好处之一，是给一个键可以很快返回相关的数据。为了提供这样快速查询的能力，我们需要引入一个高效的查找功能。

可以对列表使用顺序或二分查找，但最好是用哈希表，因为它能提供接近$O(1)$的性能。

以下代码中我们使用两个列表来创建 HashTable 类，以实现 map 的抽象数据类型。

- 一个名为 slots 的列表，用来保存键；
- 一个名为 data 的列表，用来保存数据。

查询键时，data列表相应的位置上就保存有数据。

我们将slots处理成哈希表。注意哈希表的初始大小为是11，虽然这个大小是随意的，但是选择为质数特别重要，因为这样一来后面处理冲突的效率就比较高。

In [34]:
```
class HashTable():
    def __init__(self):
        self.size  = 11
        self.slots = [None] * self.size
        self.data  = [None] * self.size
```

Hashfunction()：哈希函数

简单地采用余数法。

In [36]:
```
def hashfunction(self, key, size):
     return key%size
```

rehash()：再哈希函数，用于解决冲突

采用+1线性探测。

In [37]:
```
def rehash(self, oldhash, size):
    return (oldhash + 1)%size
```

put 函数

除非 self.slots 中包括键 key，否则这个槽位就认为是空的。它计算出的哈希值如果非空，就迭代rehash函数，直到找到一个空槽位。如果一个非空的槽位上已经有键值，就用新数据代替原数据。

In [49]:
```
def put(self, key, data):
    hashvalue = self.hashfunction(key, self.size)

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

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

get()

从计算哈希值开始，如果这个值不是一个起始的槽位，rehash就去查找另一个可能的位置。

注意第15行检查有没有返回最早的槽位，如果是，查找将停止，因为那表明已经找过所有的槽位，元素不存在。

In [50]:
```
1  def get(self, key):
2      startslot = self.hashfunction(key, self.size)
3      data = None
4      stop = False
5      found = False
6      position = startslot
7      while self.slots[position] != None and  \
8                          not found and not stop:
9          if self.slots[position]== key:
10             found = True
11             data = self.data[position]
12         else:
13            position = self.rehash(position,self.size)
14            if position == startslot:
15                stop = True
16     return data
```

附加的字典函数。

我们重载了__getitem__和 __setitem__方法来实现“[ ]”符号的使用。这也意味着，一旦HashTable对象创建，熟悉的索引方法就可用了。

In [40]:
```
1  def __getitem__(self, key):
2      return self.get(key)
3
4  def __setitem__(self, key, data):
5      self.put(key, data)
```

完整代码

In [44]:

```
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, self.size)

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

          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, self.size)

      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)
```

Analysis of Hashing

We stated earlier that in the best case hashing would provide a $O(1)$, constant time search technique. However, due to collisions, the number of comparisons is typically not so simple. Even though a complete analysis of hashing is beyond the scope of this text, we can state some well-known results that approximate the number of comparisons necessary to search for an item.

The most important piece of information we need to analyze the use of a hash table is the load factor, λλ. Conceptually, if λλ is small, then there is a lower chance of collisions, meaning that items are more likely to be in the slots where they belong. If λλ is large, meaning that the table is filling up, then there are more and more collisions. This means that collision resolution is more difficult, requiring more comparisons to find an empty slot. With chaining, increased collisions means an increased number of items on each chain.

As before, we will have a result for both a successful and an unsuccessful search. For a successful search using open addressing with linear probing, the average number of comparisons is approximately $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$ and an unsuccessful search gives $\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$ If we are using chaining, the average number of comparisons is $1+\frac{\lambda}{2}$ for the successful case, and simply λ comparisons if the search is unsuccessful.

排序

排序是将集合中的元素以某种规律放置的过程。例如，

- 一个单词的列表，可以按字母顺序排列；
- 一个城市的列表，可以按人口、面积、邮政编码来排序。

关于有序列表的好处，在折半查找中有过体现。

很多的排序算法被开发和分析，这也说明排序在计算机科学中的重要性。

大数据量的排序要占用海量的计算资源。同查找一样，排序的算法效率与元素的数量有关。

- 对小的数据集来说，复杂的排序算法没有必要，其代价太高。
- 对大的数据集来说，应尽可能地采用高效率的算法。

本节中，我们将讨论几种排序算法，并比较它们的运行时间。

在讲具体的算法之前，先考虑一下在算法过程中要用到的操作。

1. 比较

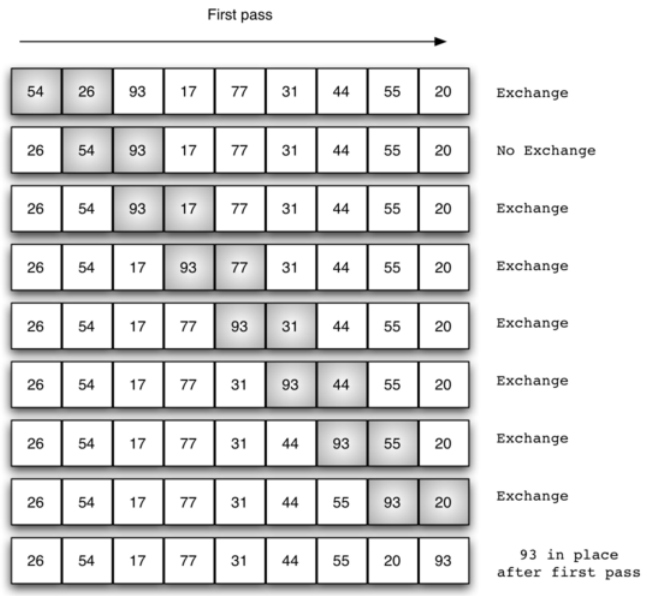
   为了排序，需要一个系统化的方法比较元素的大小以判断他们是否在正确的位置上。总的比较次数是评估排序过程的通用方法。
2. 交换

   如果数值不在正确的位置上，交换次数也是评估算法效率的重要方法。

冒泡排序（Bubble Sort）

冒泡排序是一种最简单的排序算法。它重复地访问要排序的元素，依次比较相邻两个元素，如果它们的顺序错误就进行调换，直到没有元素再需要交换，排序完成。

因为越小(或越大)的元素会经由交换慢慢“浮”到数列的顶端，这就是“冒泡”的由来。



冒泡排序的第一次遍历

若列表中有 n 个元素，则第一次遍历时需要进行 $n-1$ 次比较。

第二次遍历开始时，因最大元素已经就位，只需对剩余的 $n-1$ 个元进行要排序，这需要 $n-2$ 次比较。

经过 $n-1$ 次遍历后，最小元素也一定就位，不再需要交换。

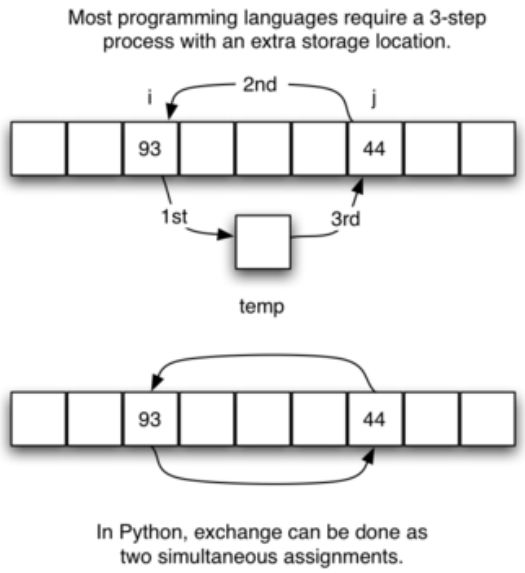
交换两个元素时，需要一个临时变量：

In [ ]:
```
1 temp = alist[i]
2 alist[i] = alist[j]
3 alist[j] = temp
```

如果不用临时变量，其中一个值会被覆盖。

在 python 中，允许进行同时赋值，即

```
a, b = b, a
```



In [4]:
```
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)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

冒泡排序算法的分析

无论原列表的元素如何排列，冒泡算法都必须进行 $n-1$ 次遍历。

Pass	Comparisons
1	n-1
2	n-2
3	n-3
...	...
n-1	1

总的比较次数是

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2).$$

关于交换：

- 最好情况：若列表有序，不要做任何交换。
- 最坏情况：每次比较都引起了一次交换；
- 平均情况：有一半的交换。

冒泡排序是最简单的、但同时也是效率最差的排序算法，因为它在元素最终确定位置之前需要反复交换，这非常浪费资源。

但冒泡排序也能做到一些其他排序算法做不到的事情。

如果在做某一次遍历时没有发生任何交换，则可马上确定此时的列表有序，从而冒泡排序可以提前停止。利用了该特点的冒泡算法通常称为"短冒泡算法"。

In [5]:
```
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)
```

```
[20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
```

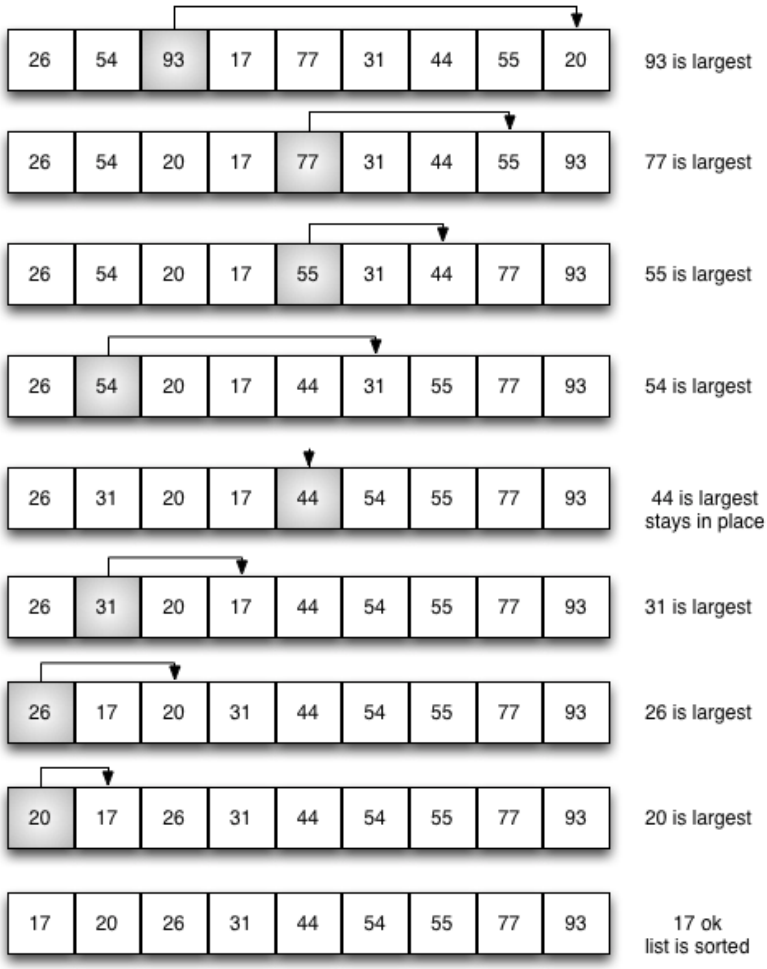
选择排序（Selection Sort）

选择排序是冒泡排序的改进。

- 第一次遍历时，找到其中最大的元素，将其与最后一个元素交换位置。
- 第二次遍历时，因最后一个元素最大，只对前$n-1$个元素重复上述操作。
- 第三次遍历时，因最后两个元素的位置已经确定，只对前$n-2$个元素重复上述操作。
- ...
- 至第$n-1$次遍历完成后，排序完成。

注意

与冒泡算法不同，选择排序每遍历一次只进行一次交换。



完整的选择排序过程

In [52]:
```
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)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

选择排序算法的分析

- 比较次数

  和冒泡排序一样多，依然是 $O(n^2)$。

- 交换次数

  比冒泡排序要少，所以选择排序一般运行得比冒泡要快。

插入排序（**Insertion Sort**）

插入排序的工作模式与冒泡、选择排序稍有不同。

它总在列表左端保持一个有序的子列表，后面的元素被逐个“插入”到前面的有序子表中。

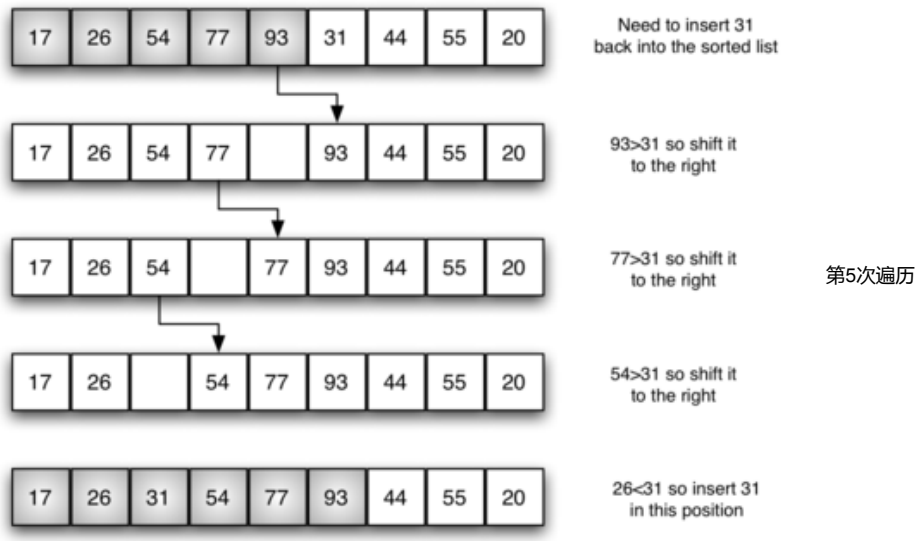
这样，有序的子表会逐渐变大，直到跟原列表一样长，排序结束。

开始时，只有包含一个元素（在 0 位上）的列表，并且它是有序的。

在进行第 i 次遍历时，将位于第 i 位的元素（”当前元素“）与有序子表进行比较（从右至左）。

- 若子表中的元素比当前元素大，往前移动；
- 当遇到一个比它小的元素、或到达子表的最左端时，当前元素就插在这个位置上。

这时已经生成一个有序的子表，包含17，26，54，77和93五个元素。下面要把31插入到这个子表中。先与93比较，93向右移动一个位置，然后是77和54也移动了，当遇到26时，停止移动，31放在空白位置上，这时就有序子表有6个元素了。



In [56]:
```
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)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

插入排序算法的分析

- 比较次数
  - 最坏情况为$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$次。
  - 最好情况，即有序列表的情形，只需一次遍历。
- 关于交换

  不涉及交换，但需要移动元素的位置。 一般情况下，移动操作大概需要交换操作三分之一资源，因为移动只需要一次赋值操作。在性能测试中，插入排序显示了很好的性能。

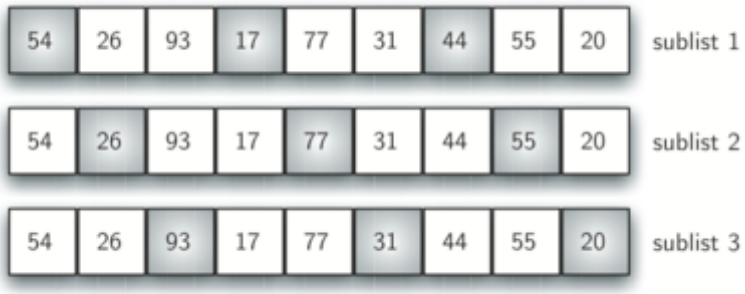
希尔排序（Shell Sort）

希尔排序的实质就是分组插入排序，也称"缩小增量排序"，因D.L. Shell于1959年提出而得名。

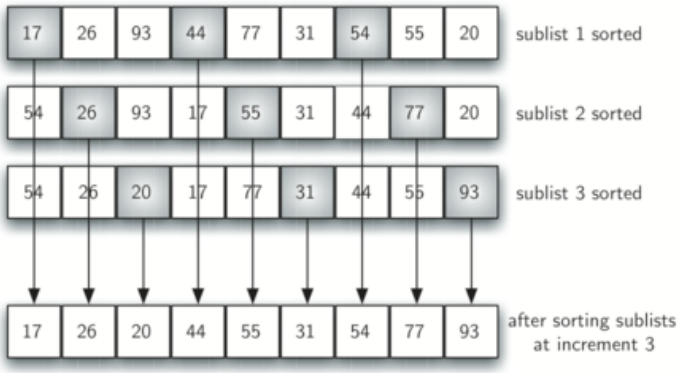
其思想是将列表拆分称若干个较小的子表，然后对每个子表使用插入排序。

第一步

希尔排序的关键是选出子表，它需要用到一个增量值 i（也称"间隙"），然后每隔一个间隙选中一个元素来组成子表。



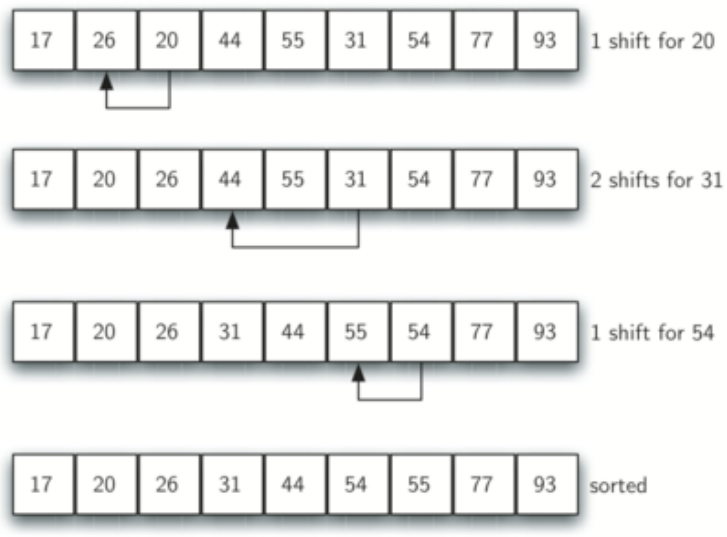
选出子表
间隙为3，得到3个子表



对子表进行插入排序
注意该表没有完全排序，但已经非常接近。

第二步

对所得的列表进行标准的插入排序，此时需要的移动操作要少很多。

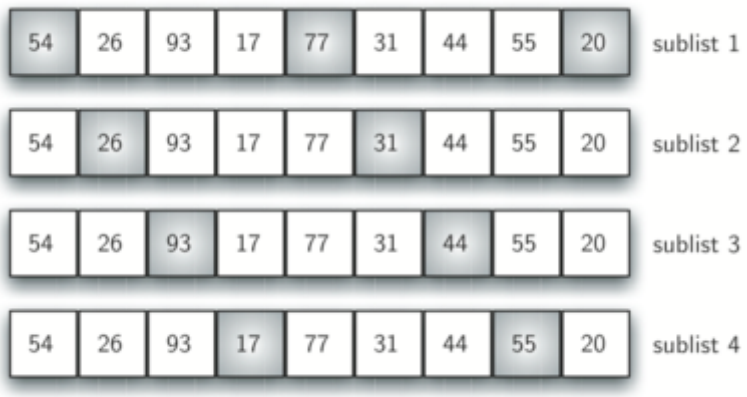


标准插入排序
该例只需要移动4次便可完成排序。

希尔排序的独特性就是间隙的选取。

以下函数使用了一个不同增量的集合，从 n/2 个子表开始，下一步就是 n/4 个子表要排序，最终是 1 个子表进行插入排序。

下图是这种增量的第一批4个子表。



In [27]:
```
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
```

In [28]:
```
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
```

In [29]:
```
alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
shellSort(alist)
print(alist)
```

```
After increments of size 4 The list is [20, 26, 44, 17, 54, 31, 93, 55, 77]
After increments of size 2 The list is [20, 17, 44, 26, 54, 31, 77, 55, 93]
After increments of size 1 The list is [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

乍一看，希尔排序不见得比插入排序更好，因为最后一步就完全是一个插入排序。

但是，最后一步的插入排序，不需要很多次的比较和移动，因为通过前面的增量插入排序，列表已经做了“预排序”。

也就是说，这个列表已经比普通列表“更有序”，所以在效率上有很大的不同。

希尔排序的时间复杂度介于$O(n)$与$O(n^2)$之间。

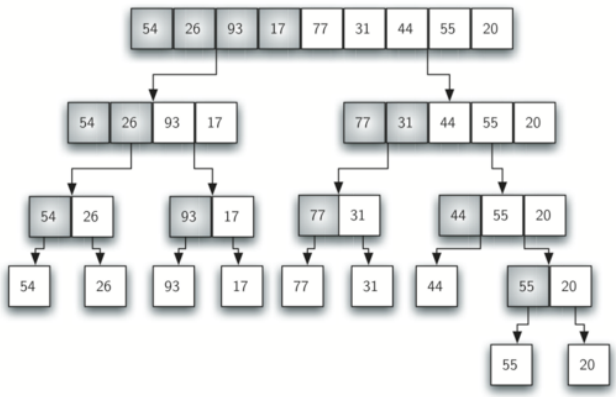
对listing5中的增量，性能是O(n2)，变更增量，例如使用 2k−1 (1, 3, 7,15, 31, 等)，性能可达到O(n3/2).

归并排序（**Merge Sort**）

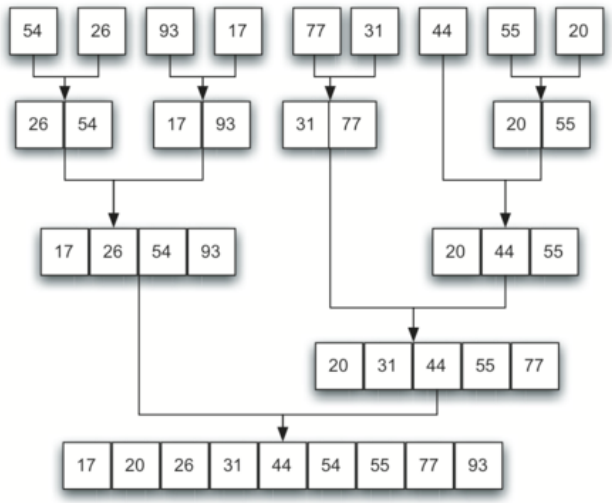
归并排序是建立在归并操作上的一种有效的排序算法，该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

将已有序的子列表合并，得到完全有序的列表。亦即，先使每个子列表有序，再使子列表段间有序。

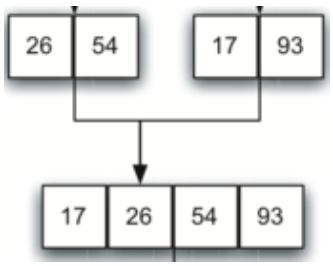
将两个有序表合并成一个有序表，称为二路归并。



切分过程



归并过程



归并过程

In [4]:
```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist) > 1:
        mid = len(alist) // 2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)
```

In [ ]:
```
        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
```

In [ ]:
```
        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)
```

In [32]:
```
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)
```

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

归并排序的性能分析

考虑其中的两个过程。

- 切分过程

  对于长为n的列表，需要$\log n$的时间来切分。

- 归并过程

  列表中每个元素都要被处理并放在有序表中，所有归并操作n个元素的列表，需要n步操作。

因此，总共需要$\log n$次切分，每次切分包括n次归并，总的时间复杂度为$O(n \log n)$。

回想列表切片的操作，如果切片大小是k，那么这个操作的性能是O(k)，为了保证mergeSoft的性能是O(nlogn) ，我们需要清除切片操作。这样，我们需要简单地将开始和结束的索引值连同列表一起传递给递归过程。这个修改留作练习。要注意的是，mergeSort函数需要额外的内存来保存切出的两个子表，当数据集很大时，问题会变得很严重。

快速排序（Quick Sort）

快速排序也使用了分而治之的策略来提高性能，而且不需要额外的内存，但是这么做的代价就是，列表不是对半切分的，因而，性能上就有所下降。

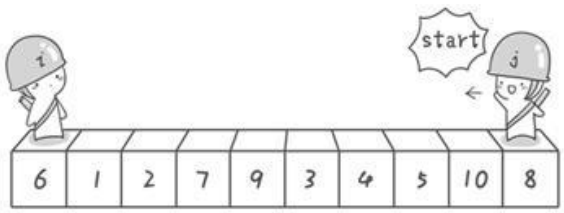
假设我们现在对 `6 1 2 7 9 3 4 5 10 8` 这个10个数进行排序。

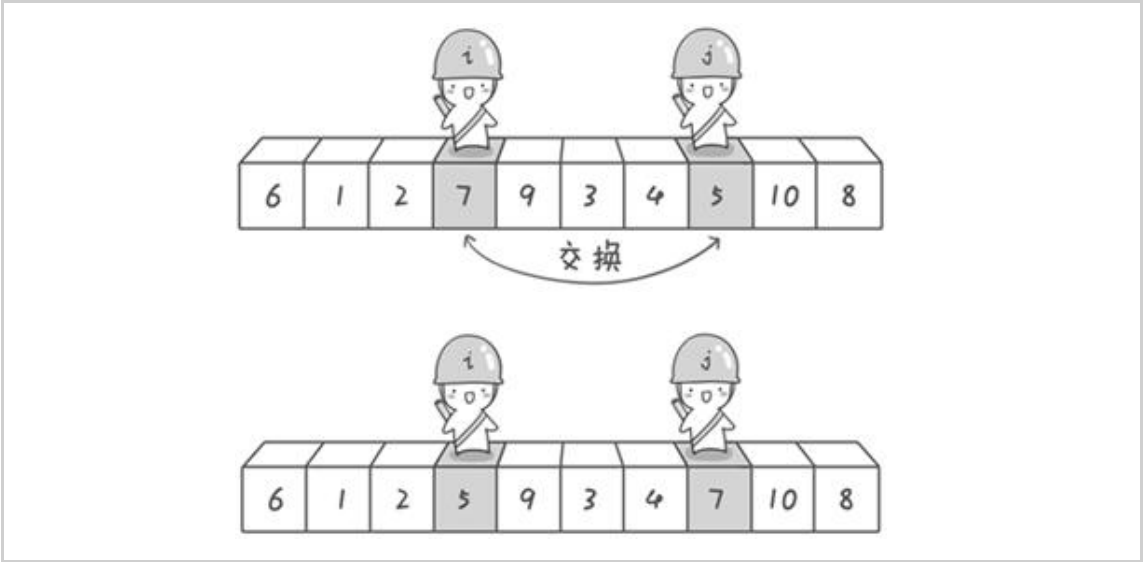
1. 首先在这个序列中随便找一个数作为基准数。为了方便，就选第一个数 `6` 作为基准数。
2. 接下来，需要将这个序列中所有比基准数大的数放在 `6` 的右边，比基准数小的数放在 `6` 的左边，类似下面这种排列：
   `3 1 2 5 4 6 9 7 10 8`

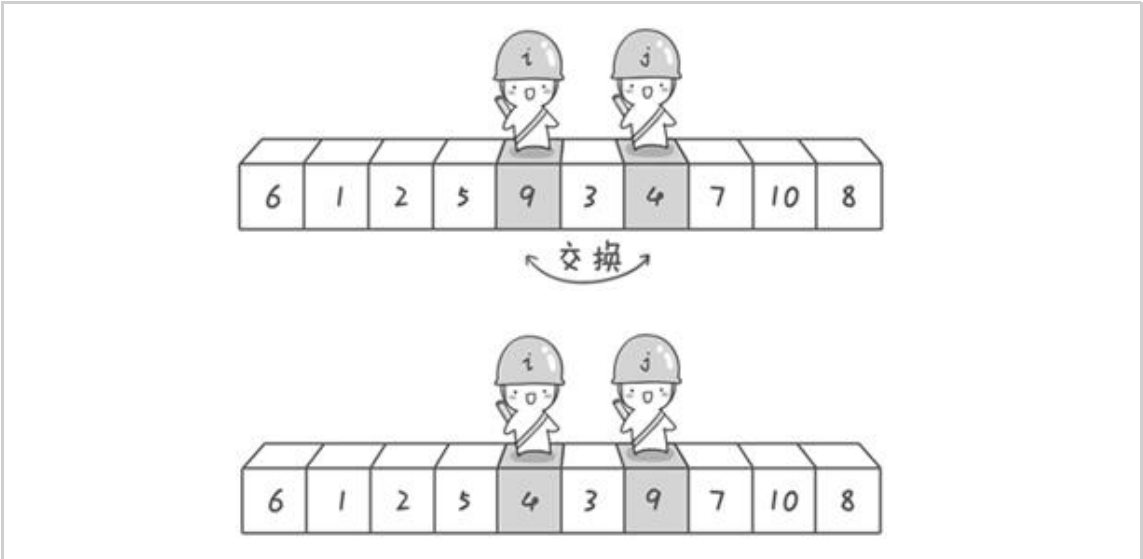
初始状态下，数字 `6` 在序列的第 `0` 位。

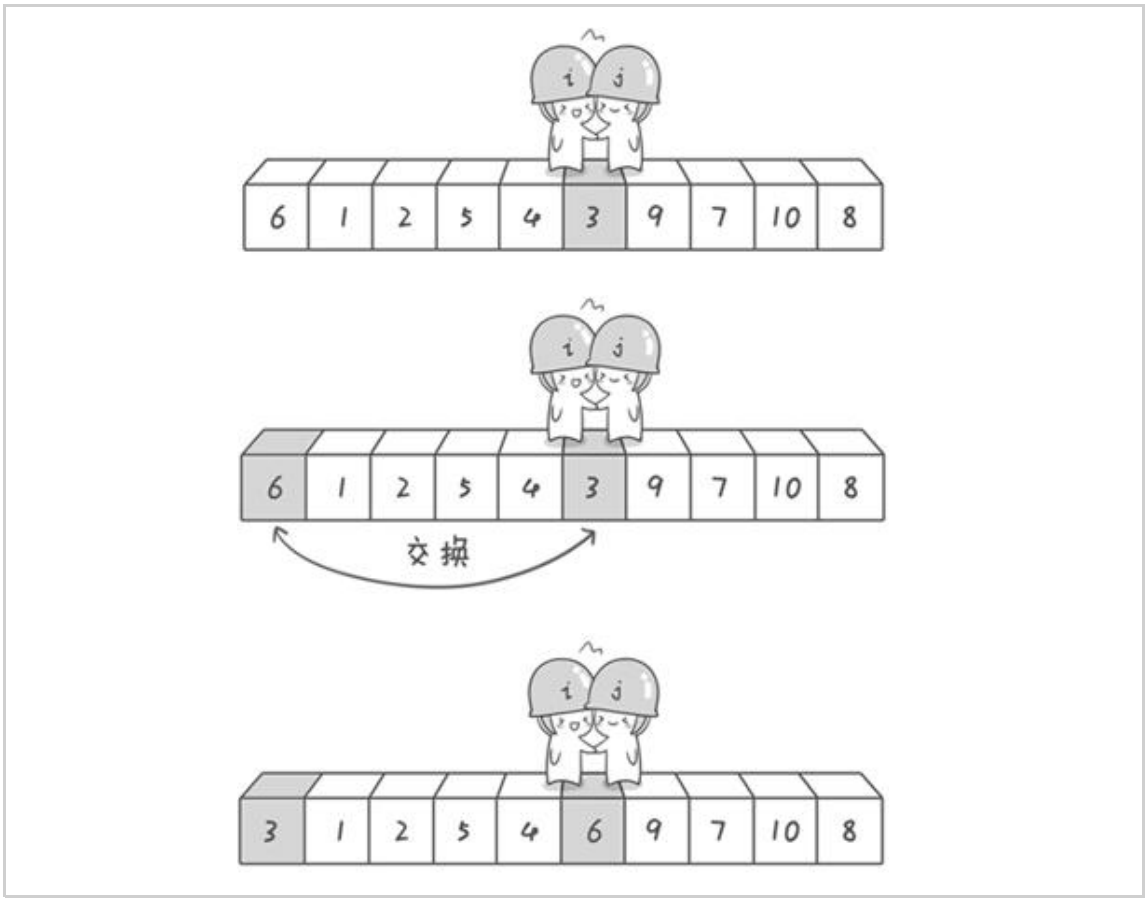
我们的目标是将 `6` 挪到序列中间的某个位置，假设这个位置是 `k` 。

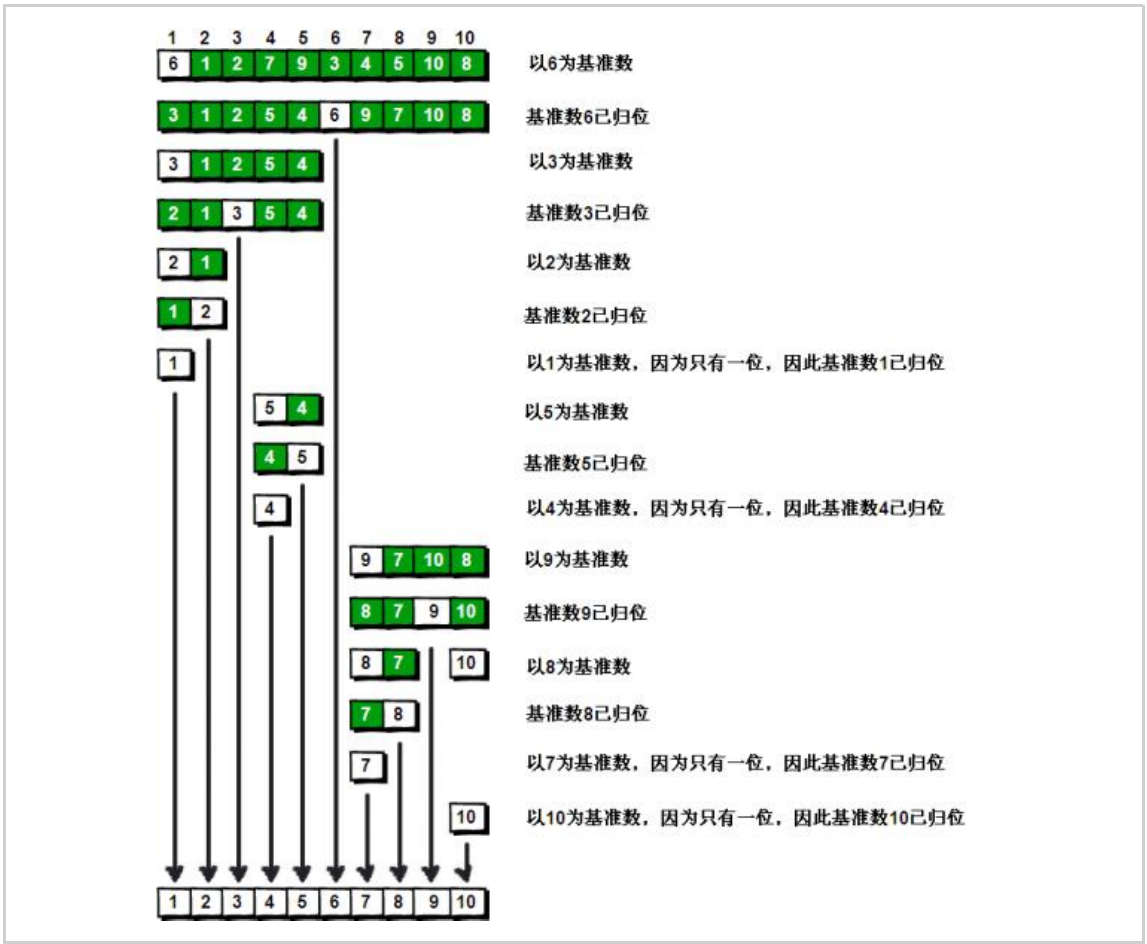
现在就需要寻找这个 `k` ，并且以第 `k` 位为分界点，左边的数都小于等于 `6` ，右边的数都大于等于 `6` 。




i
j
6 1 2 7 9 3 4 5 10 8
交换
i
j
6 1 2 5 9 3 4 7 10 8


i
j
6 1 2 5 9 3 4 7 10 8
交换
i
j
6 1 2 5 4 3 9 7 10 8


i
j
6 1 2 5 4 3 9 7 10 8
交换
3 1 2 5 4 6 9 7 10 8


1 2 3 4 5 6 7 8 9 10
6 1 2 7 9 3 4 5 10 8
以6为基准数
3 1 2 5 4 6 9 7 10 8
基准数6已归位
3 1 2 5 4
以3为基准数
2 1 3 5 4
基准数3已归位
2 1
以2为基准数
1 2
基准数2已归位
1
以1为基准数，因为只有一位，因此基准数1已归位
5 4
以5为基准数
4 5
基准数5已归位
4
以4为基准数，因为只有一位，因此基准数4已归位
9 7 10 8
以9为基准数
8 7 9 10
基准数9已归位
8 7
10
以8为基准数
7 8
基准数8已归位
7
以7为基准数，因为只有一位，因此基准数7已归位
10
以10为基准数，因为只有一位，因此基准数10已归位
1 2 3 4 5 6 7 8 9 10

In [ ]:
```
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

```

In [ ]:
```
        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
    print(alist)
    return rightmark
```

In [ ]:
```
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("Finally", alist)
```

快速排序的性能分析

注意到一个长度为n的列表，若分界点总在列表的中间，则要做$\log n$次切分。为了找到分界点，n个元素都要与关键值作一次比较，故总的时间复杂度为$n \log n$。

另外，与归并排序不同，快速排序不需要消耗额外的内存。

遗憾的是，分界点一般不在列表的中间，而是偏左或者偏右，从而生成很不均匀的子表。

最坏的情况是，生成的两个子表中，其中一个为空表，另一个包含$n-1$个元素；继续切分，又生成两个子表，其中一个仍为空表，另一个包含$n-2$个元素，等等。此时的时间复杂度就是$O(n^2)$。