搜索与查找

Sequential search algorithm and analysis

顺序查找算法及分析

Sequential Search
顺序查找

If the data items are held in a collection such as a list, we call them linear or sequential.

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。

In Python List, these data items are stored at locations called subscripts (index), which are ordered integers.

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。

By subscripts, we can access and find data items in order, a technique called "sequential search"

通过下标 ，我们就可以按照顺序来访问和查找数据项 ，这种技术称为“顺序查找”

Sequential Search
顺序查找

To determine if there are any data items that we need to search for in the list

要确定列表中是否存在需要查找的数据项

Starting with the first data item of the list,

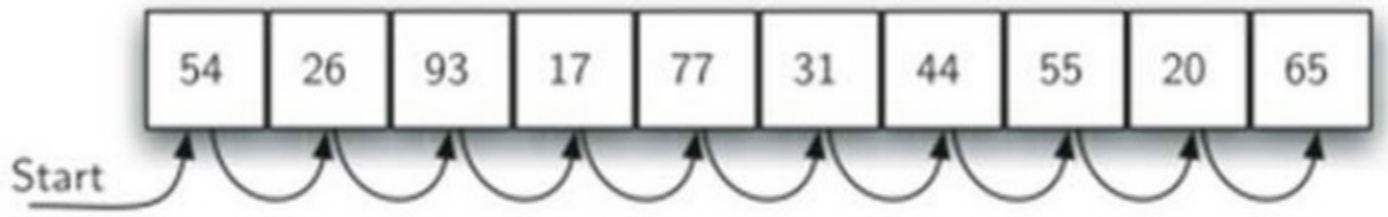
首先从列表的第1个数据项开始，

Compare data items in the order of subscript growth,

按照下标增长的顺序，逐个比对数据项，

If no item is found by the end of last item, the search fails.

如果到最后一个都未发现要查找的项，那么查找失败。



Sequential search: Unordered table search code

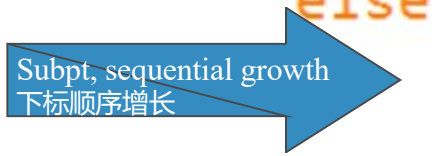
顺序查找：无序表查找代码

```
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))
```



Sequential search: algorithm analysis
顺序查找：算法分析

To analyze the search algorithm, the first step is to determine the basic calculation steps.

要对查找算法进行分析，首先要确定其中的基本计算步骤。

Reviewing the main points of algorithm analysis in Chapter 2, this basic computational step must be simple enough and performed repeatedly in the algorithm

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

In the search algorithm, this basic calculation step is to compare the data items

在查找算法中，这种基本计算步骤就是进行数据项的比对

The current data item is equal to the data item to be found or not, and the number of comparison determines the complexity of the algorithm

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

Sequential search: algorithm analysis
顺序查找：算法分析

In the sequential search algorithm, in order to guarantee the general situation to be discussed, it is assumed that the data items in the list are not arranged in value order, but are randomly placed in various positions in the list

在顺序查找算法中 ，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

In other words, the data item has the same probability of appearing throughout the list

换句话说，数据项在列表中各处出现的概率是相同的

Sequential search: algorithm analysis
顺序查找：算法分析

The number of times the comparison varies with whether the data item is in the list or not

数据项是否在列表中，比对次数是不一样的

If the data item is not in the list, all data items need to be compared, and the number of comparison is n

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

If data items are in the list, the number of times the comparison is complicated

如果数据项在列表中，要比对的次数，其情况就较为复杂

- Best case, the first time compare we get it

  最好的情况，第1次比对就找到

- In the worst case, we need to compare n times

  最坏的情况，要n次比对

Sequential search: algorithm analysis
顺序查找：算法分析

What is the general comparison of the data items in the list?

数据项在列表中， 比对的一般情形如何？

Because the data item has the same probability of location in the list; the average number of comparison is n / 2;

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；

Therefore, the algorithmic complexity of sequential searching is O (n)

所以 ，顺序查找的算法复杂度是O(n)

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

Here we assume that the data items in the list are unordered, so what will be the efficiency of the sequential search algorithm if the data items are ordered?

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 ，顺序查找算法的效率又如何呢？

Sequential search: algorithm analysis
顺序查找：算法分析

Actually, we introduced sequential search in the implementation of the ordered table Search method in Chapter 3

实际上 ，我们在第三章的有序表Search 方法实现中介绍过顺序查找

When the data term exists, the comparing procedure is identical to the unorder table

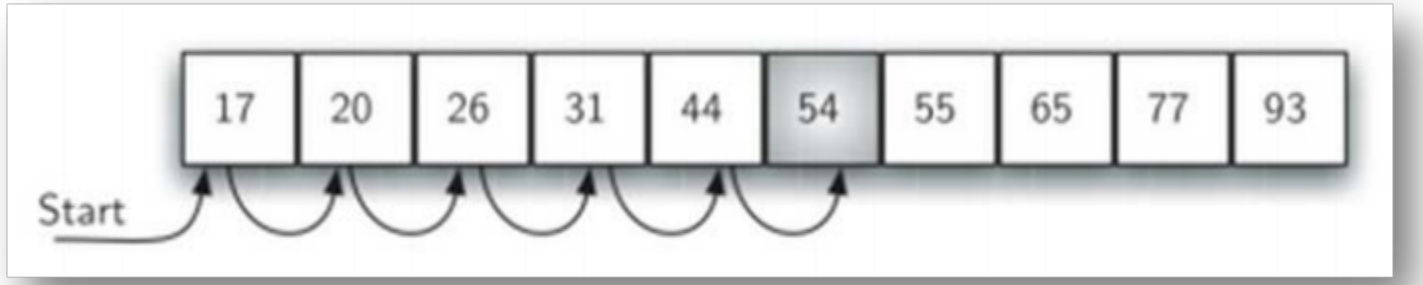
当数据项存在时，比对过程与无序表完全相同

The difference is that the comparison can end early if the data item does not exist

不同之处在于，如果数据项不存在，比对可以提 前结束

- Find the data item 50 in the figure below. When seeing 54, you can know that 50 cannot exist later, so you can quite the search in advance

如下图中查找数据项50，当看到54时，可知道后面 不可能存在50，可以提前退出查找



Sequential search: ordered table searching code

顺序查找：有序表查找代码

```
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))
```

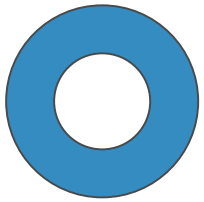
Sequential search: algorithm analysis
顺序查找：算法分析

Analysis of various cases of sequential search of ordered tables
顺序查找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

In fact, it is still O (n) in terms of algorithm complexity, but when data items do not exist, searching orderly tables can save some times of comparisons, but do not change the order of magnitude.

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，有序表的查找能节省一些比对次数，但并不改变其数量级。



Binary search algorithm and analysis

二分查找算法及分析

Binary Search
二分查找

So is there a better and faster search algorithm for ordered tables?
那么对于有序表，有没有更好更快的查找算法？

In the sequential searching, if the first data item does not match the search item, there will be at most n-1 data item to be matched
在顺序查找中，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 #

So is there a way to using the properties of ordered tables and quickly narrow down the range of data items to be compared?
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？

Binary Search
二分查找

Let's start comparing it from the middle of the list!
我们从列表中间开始比对！

If the item in the middle of the list matches the searching target, the search ends
如果列表中间的项匹配查找项，则查找结束

If there is no match, then there are two cases:
如果不匹配，那么就有两种情况：

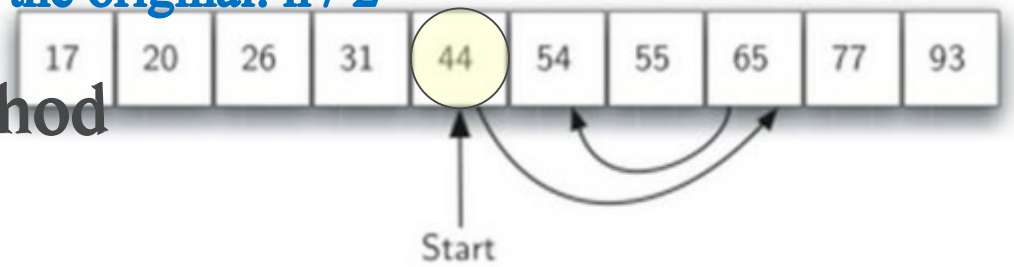
- The middle item of the list is larger than the lookup item, then the search item can only appear in the first half

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

- The middle item of the list is smaller than the search item, then the search item can only appear in the second half

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

In any case, we will reduce the comparison to half of the original: n / 2
无论如何，我们都会将比对范围缩小到原来的一半：n/2

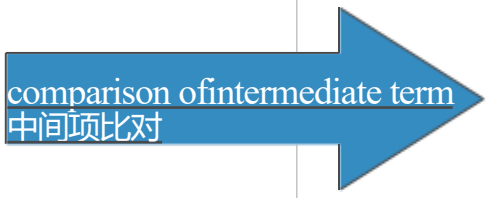


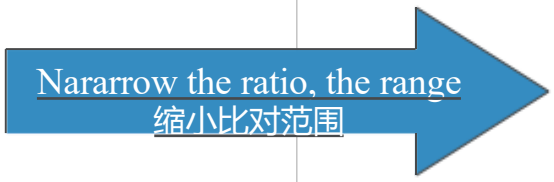
Continue searching using the above method
继续采用上面的方法查找

Each time, it reduces the comparison range by half
每次都会将比对范围缩小一半

Binary Search: Code

二分查找：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))
```

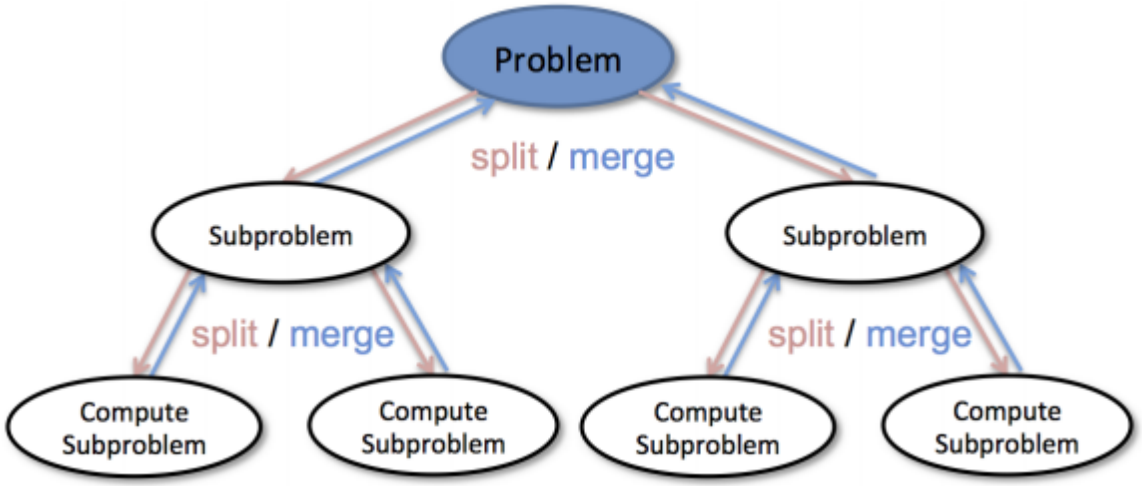
Binary search: divide and conquer
二分查找：分而治之

The binary search algorithm actually reflects the typical strategy to solve the problem: divide and conquer

二分查找算法实际上体现了解决问题的典型策略：分而治之

The problem is divided into several smaller-scale parts, by solving each small-scale part problem, and by summarizing the results to obtain the solution of the original problem

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



Binary search: divide and conquer
二分查找：分而治之

Obviously, the recursive algorithm is a typical divide-and-conquer strategy algorithm, and the binary-searching method is also suitable to implement with the recursive algorithm

显然，递归算法就是一种典型的分治策略算法，二分法也适合用递归算法来实现

```
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)
```

end condition
结束条件

Call yourself
调用自身

Reduce the scale
缩小规模

Binary search: algorithm analysis
二分查找：算法分析

Due to binary searches, each comparison reduces the next comparison's range by half

由于二分查找，每次比对都将下一步的比对范围缩小一半

The remaining data items after each comparison are shown in the following table:

每次比对后剩余数据项如下表所示：

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

Binary search: algorithm analysis

二分查找：算法分析

When enough alignments are made, only one data item is left in the alignment range

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

Whether or not this data item matches the search item, the comparison eventually ends, solving the following equation:

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：

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

Got from: $i=\log_2(n)$

得到：$i=\log_2(n)$

So the algorithm complexity of binary search is $O(\log_2(n))$

所以二分法查找的算法复杂度是$O(\log_2(n))$

Binary search: further consideration
二分查找：进一步的考虑

Although we derive the complexity of the binary finding O(logn)
虽然我们根据比对的次数 ，得出二分查找的复杂度O(logn)
However, in addition to the comparison, there is one more factor in this algorithm to note:
但本算法中除了比对 ，还有一个因素需要 注意到：

```
binarySearch(alist[ :midpoint],item)
```

This recursive call uses list slices, and the complexity of the slice operation is O(k), which slightly increases the time complexity of the whole algorithm; of course, we use slicing to be more readable, and actually without slicing, but only with the index value of the start and end, so there is no more time for slicing.
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。

Binary search: further consideration
二分查找：进一步的考虑

In addition, although binary search outperform sequential search in time complexity, the cost of sorting the data items is also taken into account

另外，虽然二分查找在时间复杂度上优于顺序查找，但也要考虑到对数据项进行排序的开销

If you can be searched multiple times after a single sorting, then the cost of sorting can be diluted

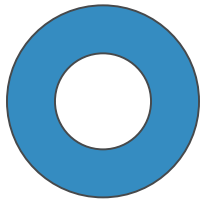
如果一次排序后可以进行多次查找，那么排序的开销就可以摊薄

But if the data set changes frequently and searches relatively little, it may be more economical to use unordered tables plus sequential search

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

Therefore, in the problem of algorithm selection, it is not enough to just look at the merits of the time complexity, but also need to take into account the practical application situation.

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。



Bubble sorting and selection sorting algorithm and analysis

冒泡排序和选择排序算法及分析

Sort: the Bubble Sort

排序：冒泡排序BubbleSort

The algorithmic idea of bubble sorting is to compare unordered tables multiple times,

冒泡排序的算法思路在于对无序表进行多趟比较交换，

Each time includes multiple pairwise comparisons, and swap the inverted data items, finally putting the maximum item in place in this time

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

After n-1 times of comparison and exchange, the whole table ranking is realized

经过n-1趟比较交换，实现整表排序

Each trip is similar to the process of a "bubble" constantly rising to the surface of the water

每趟的过程类似于“气泡”在水中不断上浮到水面的经过

Sort: the Bubble Sort

排序：冒泡排序BubbleSort

The first comparison exchange showed a total of n-1 comparing adjacent data

第1趟比较交换，共有n-1对相邻数据进行比较

Once the maximum item is passed, the maximum item is exchanged all the way to the last item

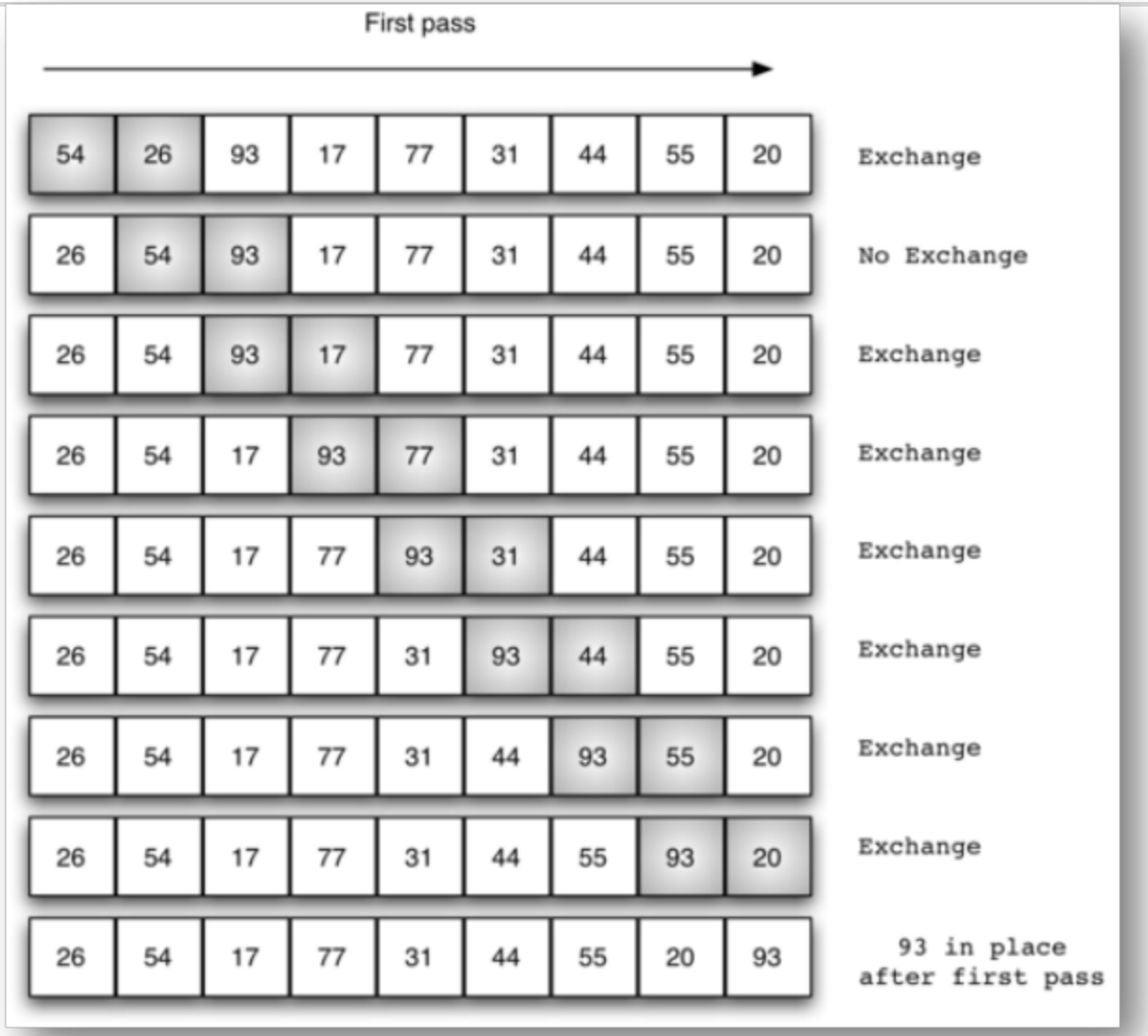
一旦经过最大项，则最大项会一路交换到达最后一项

When the second comparison exchange is made, the maximum item is already in place, and the data to be sorted is reduced to n-1 with n-2 compareing adjacent data. After trip n-1, the minimum item must be the first in the list, so there is no need to be processed.

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，小项一定在列表首位，就无需再处理了。

Bubble sorting: trip 1

冒泡排序：第1趟



Bubble sort: Code

冒泡排序：代码

```
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)
```

n-1 trip
n-1趟

Order wrong, exchange
序错，交换

The Python supports direct switching

Python支持直接交换

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

https://zh.visualgo.net/sorting

Bubble sorting: algorithm analysis
冒泡排序：算法分析

The arrangement status of the initial data items of the unordered table had no effect on the bubble ordering

无序表初始数据项的排列状况对冒泡排序没有影响

The algorithm process always requires n-1 trips, and the number of comparisons gradually decreases from n-1 to 1, and includes the possible exchange of data items.

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。

The number of comparisons is 1~n-1 accumulation: the time complexity of the contrast is O (n^2)

比对次数是1~n-1的累加：对比的时间复杂度是O(n^2)

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

Bubble sorting: algorithm analysis
冒泡排序：算法分析

Regarding the number of exchanges, the time complexity is also $O(n^2)$, usually each exchange consists of 3 assignments

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

At best, the list is ordered before sorting, and the number of exchanges is 0

最好的情况是列表在排序前已经有序，交换次数为0

The worst case is every data item needs to be exchanged, and the number of exchanges is equal to the number of comparisons

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

The average case is the half of the worst case

平均情况则是最差情况的一半

Bubble sorting: algorithm analysis
冒泡排序：算法分析

Bubble sorting is usually used as a poor-time-efficient sorting algorithm, as a benchmark for other algorithms.

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。

Its efficiency is poor mainly because before each data item finds its final position,

其效率主要差在每个数据项在找到其最终位置之前，

must undergo multiple comparison and exchange, and most of the operations are invalid.

必须要经过多次比对和交换，其中大部分的操作是无效的。

But one advantage is that no need to cost any additional storage.

但有一点优势，就是无需任何额外的存储空间开销。

Bubble sorting: Performance improvement
冒泡排序：性能改进

In addition, by monitoring whether each trip has been exchanged or not, you can determine whether the sorting has been completed in advance
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

This is also something that most other sorting algorithms cannot do
这也是其它多数排序算法无法做到的

If there is no exchange in a comparison, the list is in order and can end the algorithm in advance
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

Bubble sorting: Performance improvement
冒泡排序：性能改进

```
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)
```

Selection Sorting

选择排序Selection

Selection sorting improves the bubbling sorting, retaining its basic multiple-trip comparison idea, with each trip putting the current largest item in place.

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。

However, the selection sorting reduces the exchange. Compared with the bubble sorting, there is only 1 exchange per trip, recording the location of the largest item, and finally exchanging with the last item of this trip. The time complexity of the selection sorting is slightly better than the bubble sorting

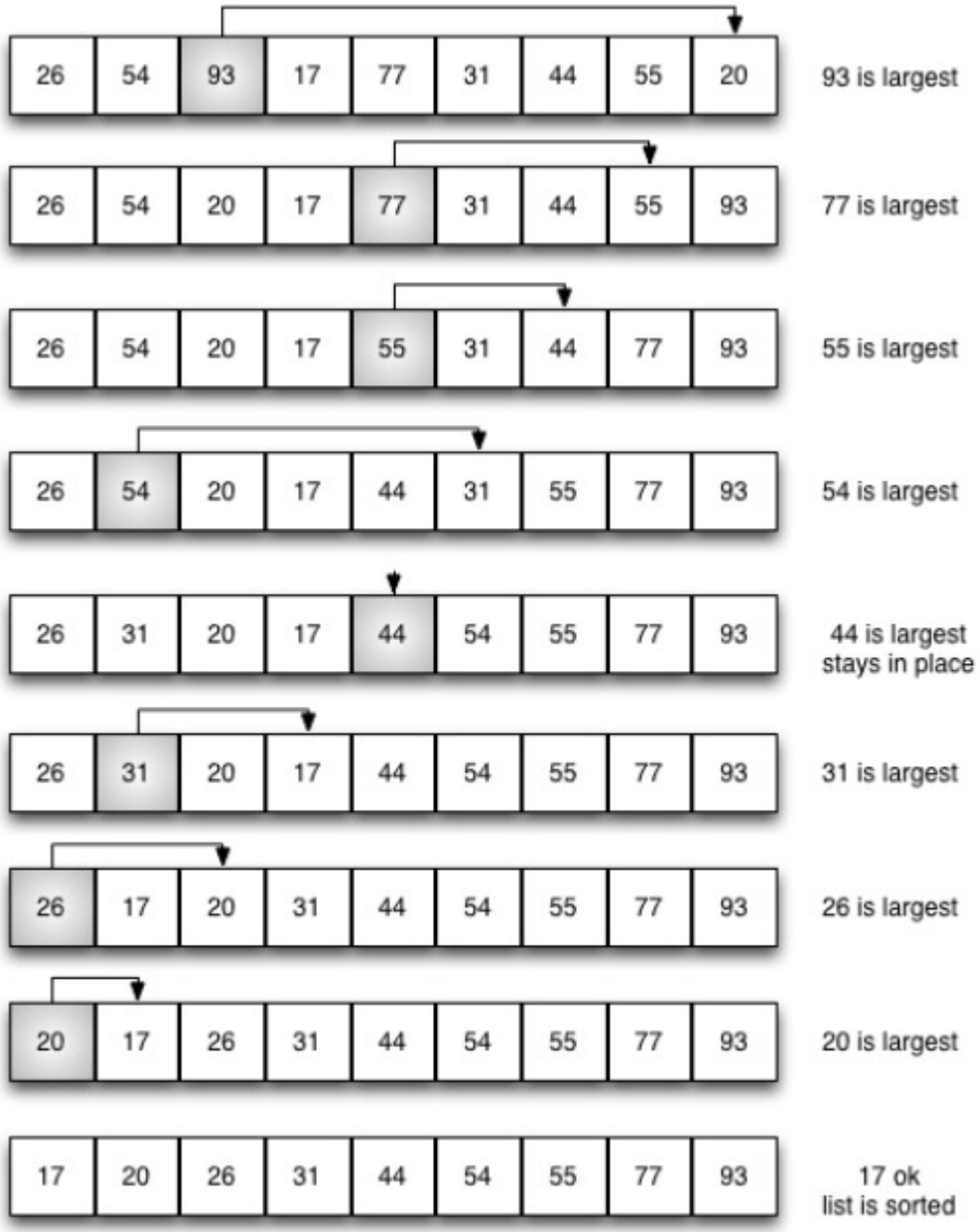
但选择排序对交换进行了削减，相比起冒泡排序进行多次交换，每趟仅进行1次交换，记录最大项的所在位置，最后再跟本趟最后一项交换，选择排序的时间复杂度比冒泡排序稍优

The number of comparisons remains unchanged, or O (n^2)

比对次数不变，还是O(n^2)

Switching number is reduced to O(n)

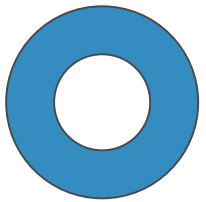
交换次数则减少为O(n)


26 54 93 17 77 31 44 55 20
93 is largest
26 54 20 17 77 31 44 55 93
77 is largest
26 54 20 17 55 31 44 77 93
55 is largest
26 54 20 17 44 31 55 77 93
54 is largest
26 31 20 17 44 54 55 77 93
44 is largest
stays in place
26 31 20 17 44 54 55 77 93
31 is largest
26 17 20 31 44 54 55 77 93
26 is largest
20 17 26 31 44 54 55 77 93
20 is largest
17 20 26 31 44 54 55 77 93
17 ok
list is sorted

Selection Sort: Code
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
```



Insertion sort algorithm and the analysis
插入排序算法及分析

Insertion Sort
插入排序

Time complexity of insertion sort is still O (n^2), But the algorithm idea is different from the bubble sorting and choice sorting

插入排序时间复杂度仍然是O(n^2)，但算法思路与冒泡排序、选择排序不同

Insertion sort maintains an ordered sublist, always at the front of the list, and then expands this sublist to the full table

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

Insertion Sort
插入排序

On the first trip, the sublist contains only the first data item, and the second data item is inserted in the sublist as a new item item, so that the sorted sublist contains 2 data items

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

On the second trip, continue to compare the previous two data items with the third data item, and move the data items larger than itself to add to the sublist. After n-1 times of comparisons and insertions, the sublist is extended to the full table, and the sorting is completed

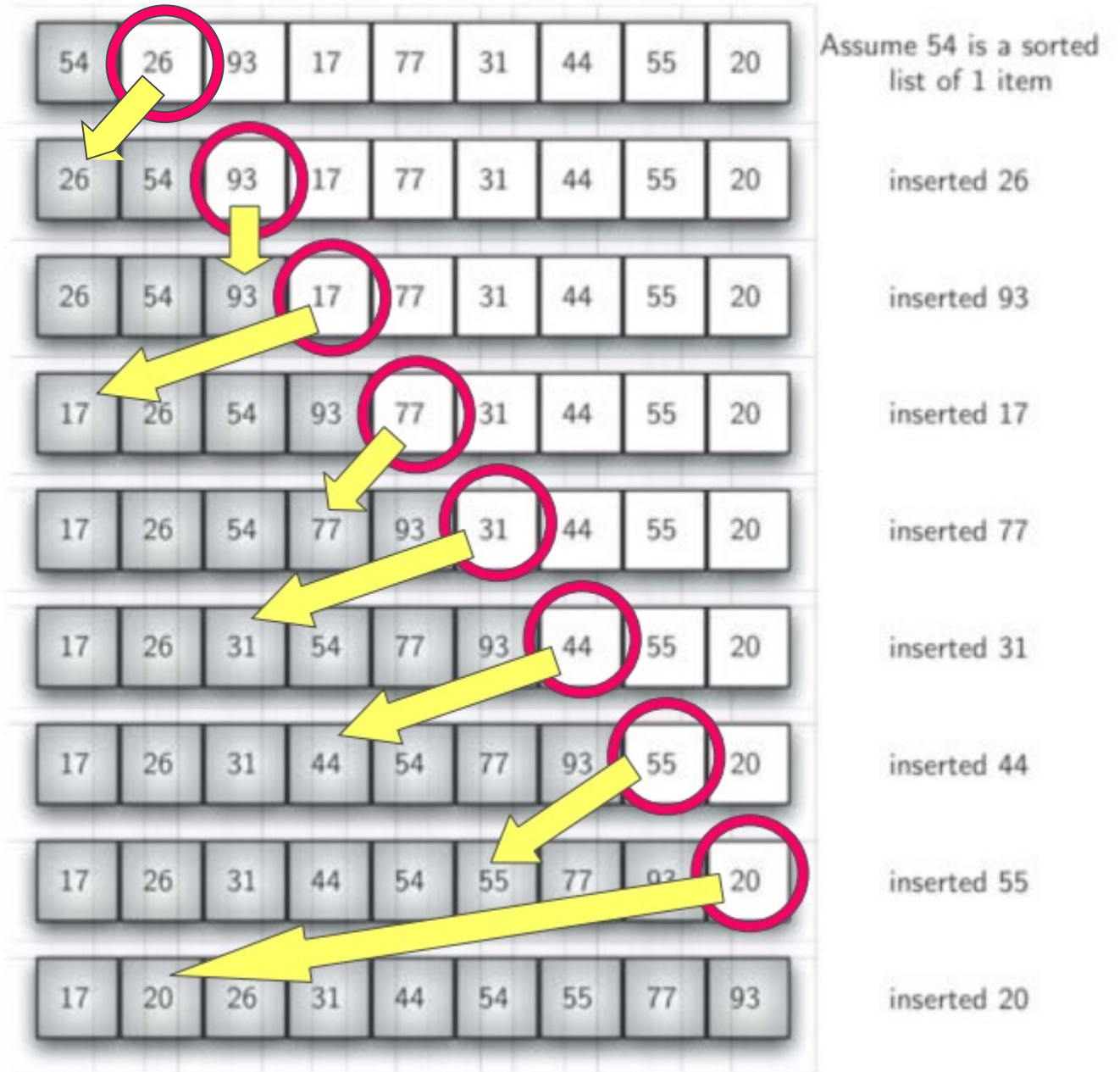
第2趟，再继续将第3个数据项跟前2个数据项比对，并移动比自身大的数据项，空出位置来，以便加入到子列表中，经过n-1趟比对和插入，子列表扩展到全表，排序完成

Insertion Sort
插入排序

The comparison of insertion sort are mainly used to find the insertion position of the "new item"
插入排序的比对主要用来寻找“新项”的插入位置

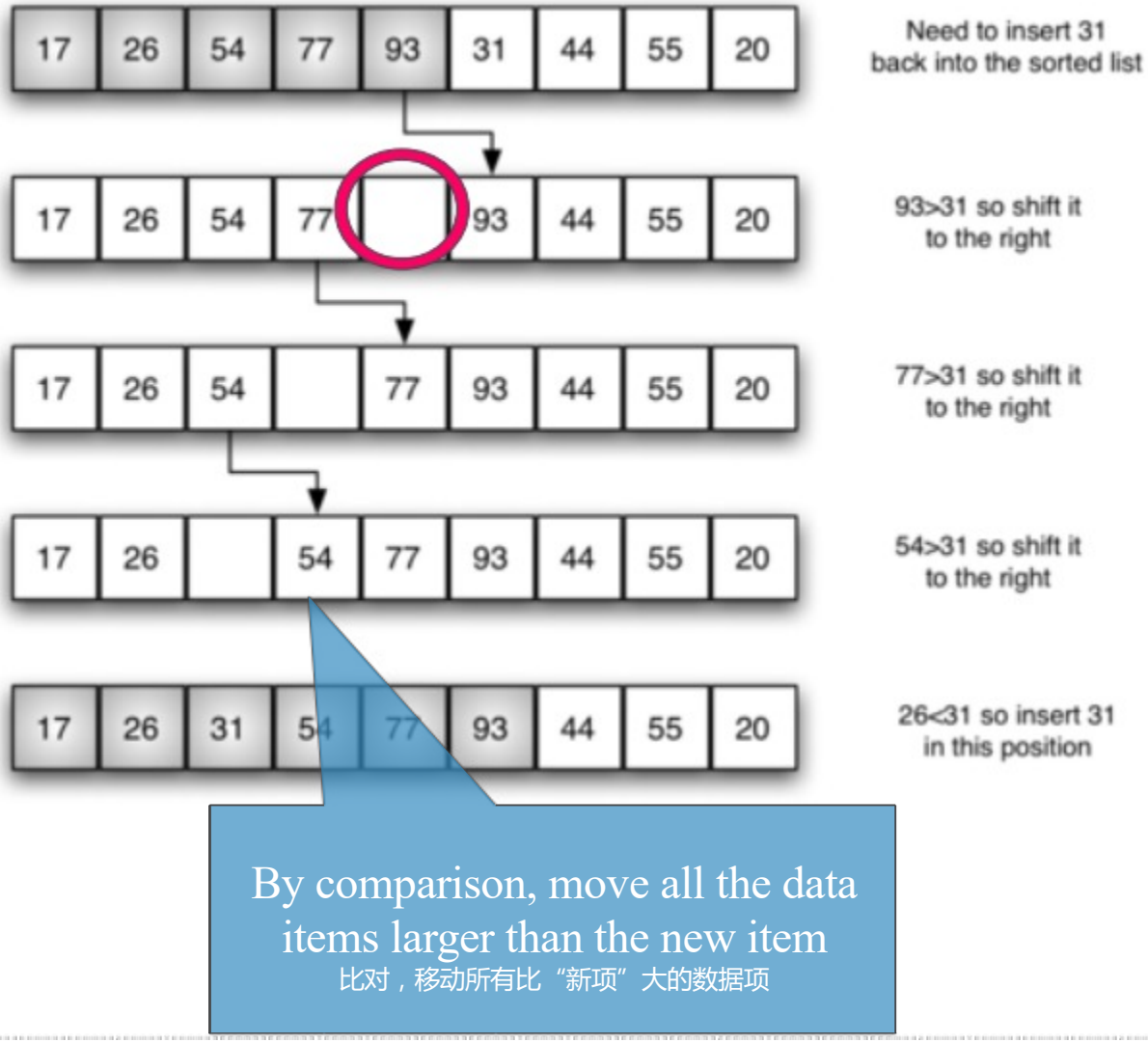
In the worst case, each trip compares all the items in the sublist, the total comparisons are the same as the bubble sort, and the order of magnitude remains O (n^2)
最差情况是每趟都与子列表中所有项进行比对，总比对次数与冒泡排序相同，数量级仍是$O(n^2)$

At best, when the list is ordered, only one comparison per trip, and the total number is O (n)
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)


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

Insert sort: idea

插入排序：思路



Insertion sort: Code

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
```

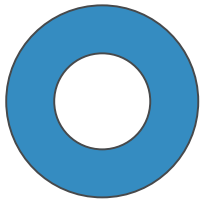
New item / insert
新项/插入项

Comparison and movement
比对、移动

Insert a new item
插入新项

Since the move operation contains only one assignment and is 1 / 3 of the swap operation, the insertion sorting performance is better.

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。



Shell Sort Algorithm and Analysis
谢尔排序算法及分析

Shell sort

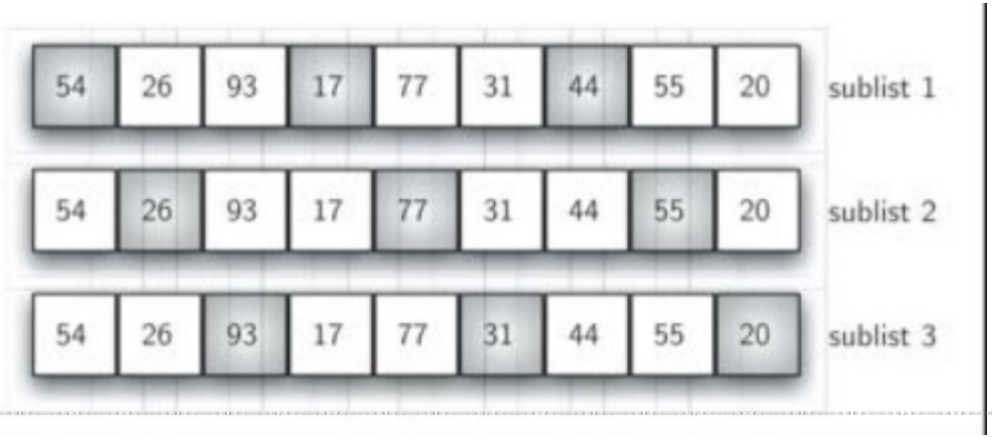
谢尔排序

We note that the number of comparisons, at best, O(n), which occurs when the list is already ordered. Actually, the more ordered the list is , the fewer comparisons insertion sort has to take

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**

From this case, Shell sorting is based on insertion sort, which "interval" the sublist, and each sublist performs insertion sorting

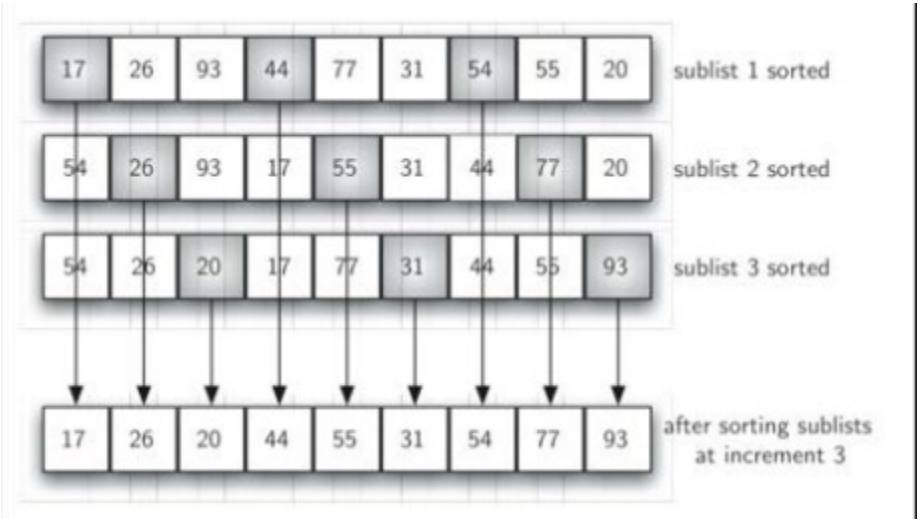
从这个情况入手，谢尔排序以插入排序作为基础，对无序表进行“间隔”划分子列表，每个子列表都执行插入排序



Shell sort
谢尔排序

As the number of sublists decreases, the unordered table gets closer to order, thus reducing the number of comparisons for the overall list. Sublists with an interval of 3, the overall situation is closer to the order after the sublists are inserted and sorted

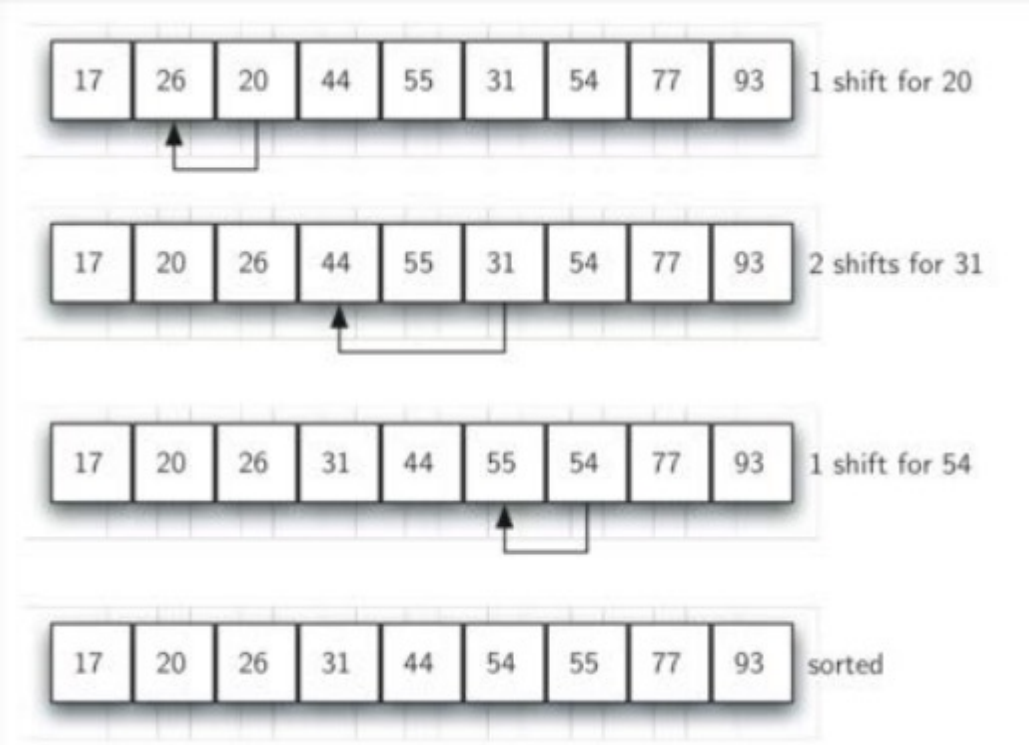
随着子列表的数量越来越少，无序表的整体越来越接近有序，从而减少整体排序的比对次数，间隔为3的子列表，子列表分别插入排序后的整体状况更接近有序



Shell sort: idea
谢尔排序：思路

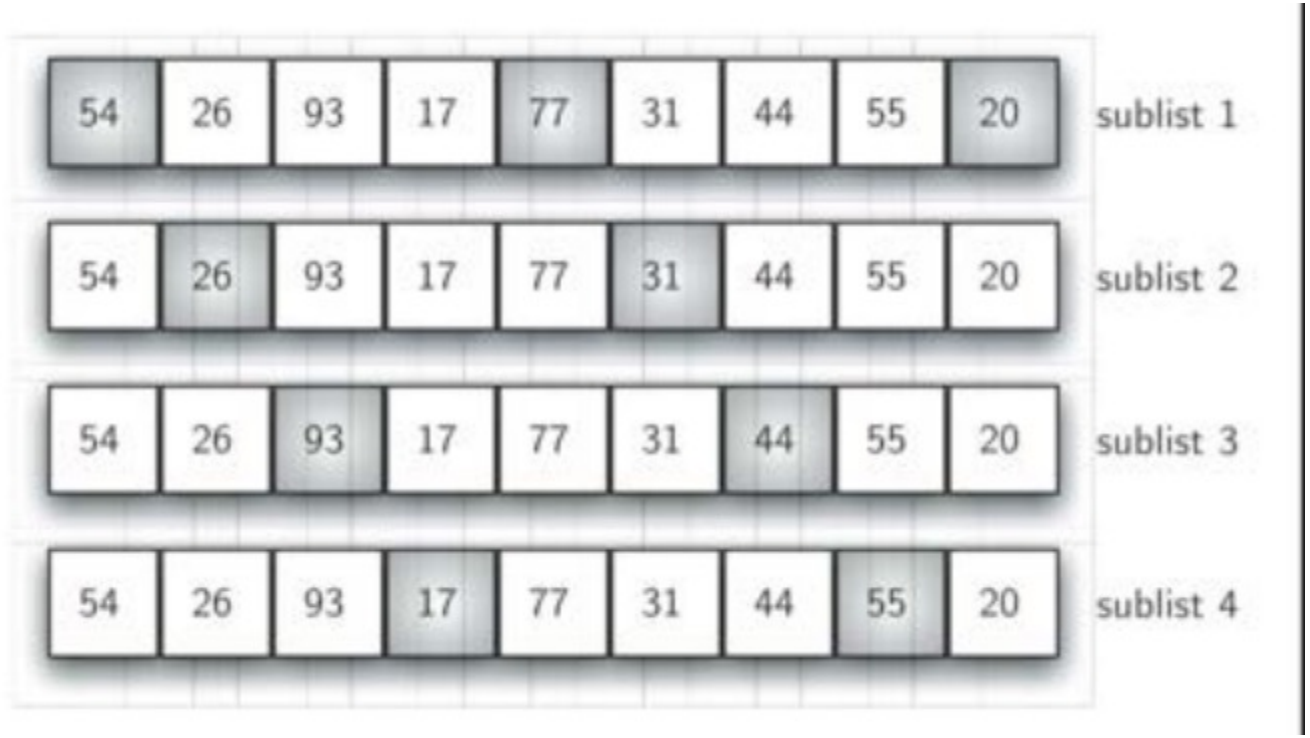
The last trip is a standard insertion sort, but since the previous trips had already processed the list to be very close to order, this time only took a few moves to complete

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



Schell sort: idea
谢尔排序：思路

The sublist interval usually starts at n / 2, doubling for each trip: n / 4, n / 8 …Until 1
子列表的间隔一般从n/2开始，每趟倍增：n/4,n/8……直到1



Shell sort:Code
谢尔排序：代码

```
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
```

Interval setting
间隔设定

子列表排序

The interval is reduce
间隔缩小

Shell sorting: Algorithm analysis

谢尔排序：算法分析

In rough terms, Shell sorting is based on insertion sorting, and may not be better than insertion sorting

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

But as each trip makes the list closer to order, the process reduces the number of "invalid" comparisons originally needed

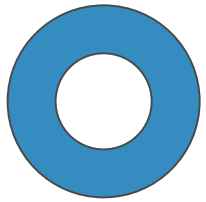
但由于每趟都使得列表更加接近有序，这过程会减少很多原先需要的“无效”比对

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

If keeping the interval at 2^k-1 (1,3,5,7,15,31, etc.), the time complexity of Shell sorting is approx O ($n^{3/2}$)

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($n^{3/2}$)

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



Algorithm of merge sort and analysis
归并排序算法及分析

Merge Sort
归并排序

Let's take a look at the application of divide-and-conquer strategies in sorting

下面我们来看看分治策略在排序中的应用

Merge sorting is a recursive algorithm. The idea is to continuously divide the data table into two halves and sort the two halves separately

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

- The basic end condition for recursion: data table has only 1 data item. Naturally, it is in good order;

  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；

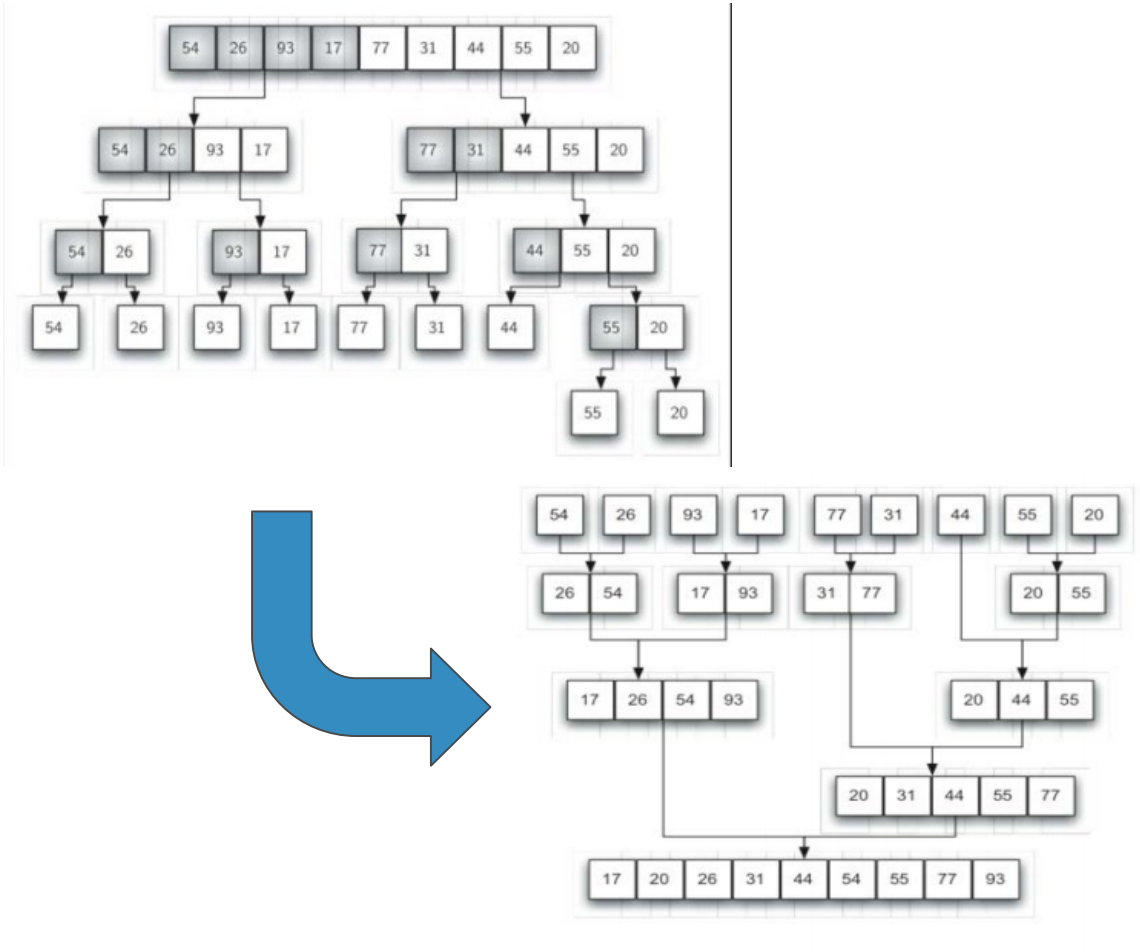
- Size reduction: split the data sheet into two equal halves, and reduce to half the size;

  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；

- Call itself: Call the two halves to sort themselves, and then merge the sorted halves to get the sorted data table

  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

Merge Sort
归并排序



Merge sort: Code
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)
```

Basic End Conditions
recursive
基本结束条件
递归调用

Zippered stagger merges the left and right halves into the result list from small to large
拉链式交错把左右半部从小到大归并到结果列表中

Merge the remaining item of the left half
归并左半部剩余项

Merge the remaining item of the right half
归并右半部剩余项

Another merge sort's code (more Pythonic)
另一个归并排序代码(更Pythonic)

```python
# merge sort
# 归并排序


def merge_sort(lst):
    # 递归结束条件
    if len(lst) <= 1:
        return lst

    # 分解问题，并递归调用
    middle = len(lst) // 2
    left = merge_sort(lst[:middle])  # 左半部排好序
    right = merge_sort(lst[middle:])  # 右半部排好序

    # 合并左右半部，完成排序
    merged = []
    while left and right:
        if left[0] <= right[0]:
            merged.append(left.pop(0))
        else:
            merged.append(right.pop(0))

    merged.extend(right if right else left)
    return merged
```

Merge sort: algorithm analysis
归并排序：算法分析

The sort is sorting into two processes: splitting and merging
将归并排序分为两个过程来分析：分裂和归并
The process of splitting, drawing from the analysis results in binary search, is logarithmic complexity and the time complexity is $O(\log_2 n)$
分裂的过程，借鉴二分查找中的分析结果，是对数复杂度，时间复杂度为$O(\log_2 n)$
Here all its data items are compared and placed once respective of each part of the split, so it is of linear complexity, and its time complexity is O(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)

Overall, O(n), and the overall time complexity is $O(n\log_2 n)$
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$O(n\log_2 n)$

Merge sort: algorithm analysis
归并排序：算法分析

Finally, we still notice two slice operations

最后，我们还是注意到两个切片操作

For the accuracy of time complexity analysis,

为了时间复杂度分析精确起见，

You can pass the start and end points of the two split parts by canceling the slicing operation, and there is no problem.

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，

It's just that algorithm readability is sacrificed a little bit

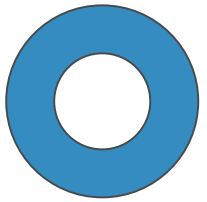
只是算法可读性稍微牺牲一点点。

We noticed that the merge sort algorithm uses an extra storage space of one time for the merge

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

This feature needs to be taken into account when sorting very large datasets

这个特性在对特大数据集进行排序的时候要考虑进去



Quick sort algorithm and analysis
快速排序算法及分析

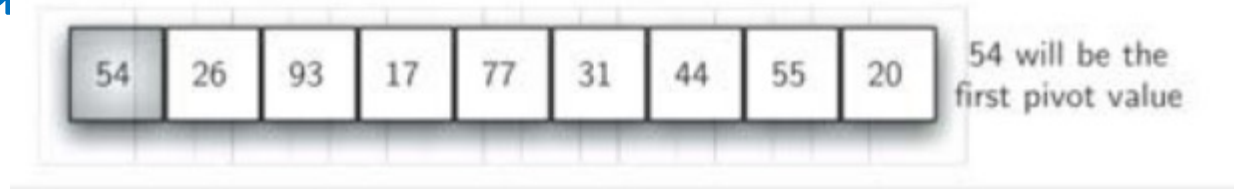
Quick sort
快速排序

The idea of quick sorting is to divide the data table into two parts based on a "median" data item: less than half of the median and more than half of the median, and then do quickly sort to each part (recursion)

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(递归)

If you want these two halves to have an equal number of data items, you should find the "median" of the data table, but finding the median requires computational cost! For no cost, you have to find a number as the "median", such as the first number.

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，第一个数



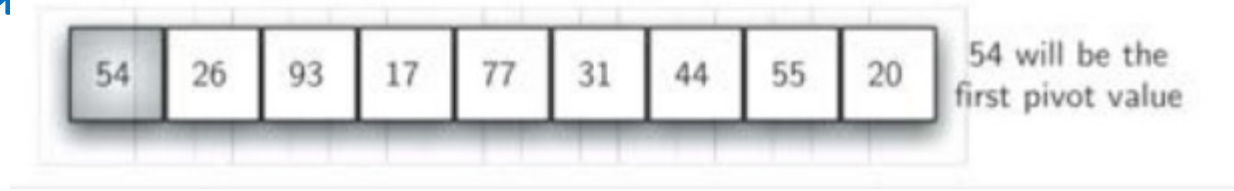
Quick sort
快速排序

The idea of quick sorting is to divide the data table into two parts based on a "median" data item: less than half of the median and more than half of the median, and then do quickly sort to each part (recursion)

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(递归)

If you want these two halves to have an equal number of data items, you should find the "median" of the data table, but finding the median requires computational cost! For no cost, you have to find a number as the "median", such as the first number.

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”，比如第一个数



Quick sort
快速排序

The fast-sorted recursive algorithm "recursive three elements" is as follows

快速排序的递归算法“递归三要素”如下

Basic end condition: the data sheet has only 1 data item, which is naturally in good order

基本结束条件：数据表仅有1个数据项，自然是排好序的

Size reduction: divide the data sheet into two halves according to the "median", preferably of equal size

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

Call itself: call two halves to sort itself separately (sort basic operations during splitting)

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)

Quick sort: Figure
快速排序：图示

Target to split the data table: Find the location of the median
分裂数据表的目标：找到“中值”的位置

The means of splitting the data sheet
分裂数据表的手段

Set left / rightmark
设置左右标(`left/rightmark`)

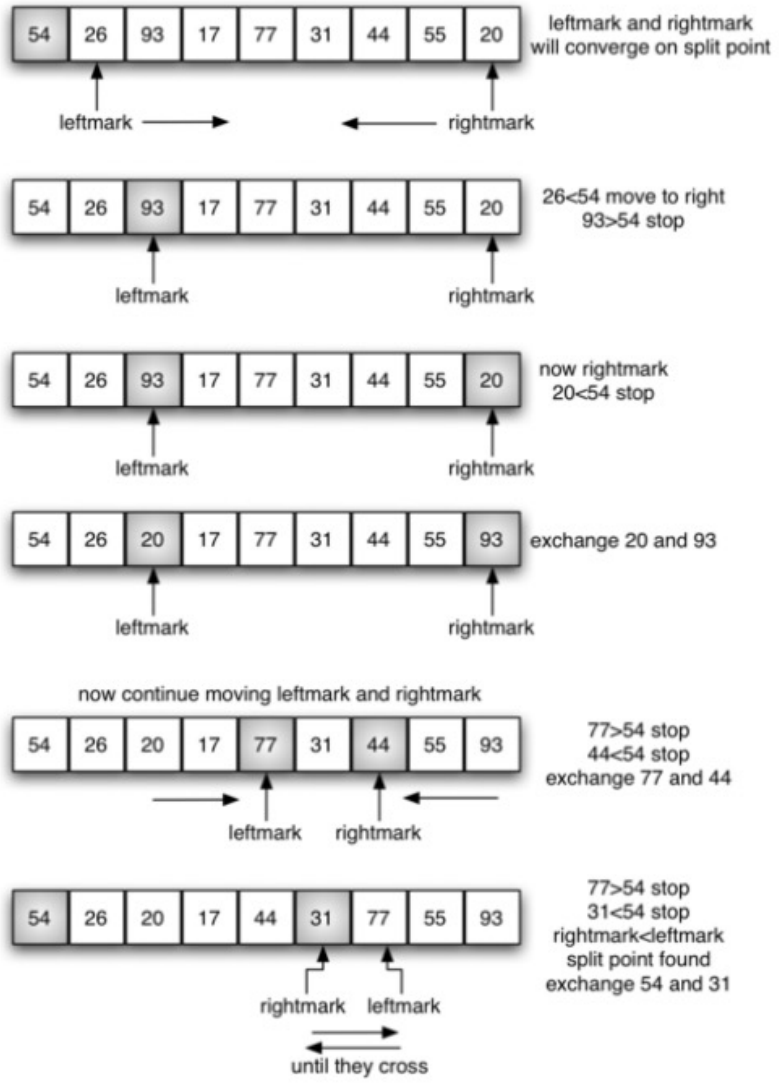
Move left label to right and right to left
左标向右移动，右标向左移动

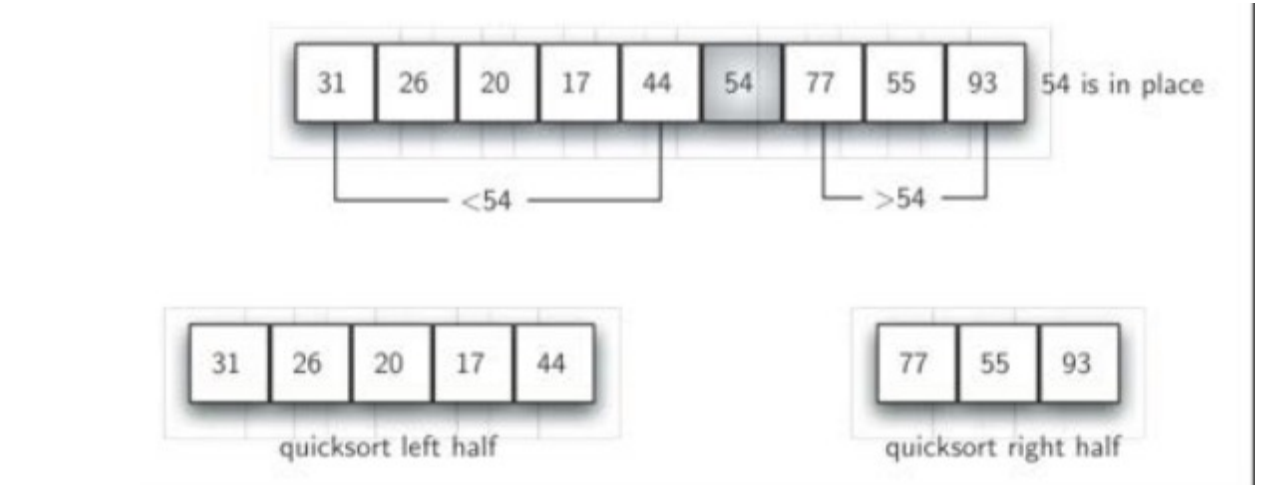
- The left mark keeps moving to the right and stops when it is larger than the median
  左标一直向右移动，碰到比中值大的就停止
- The right mark moves to the left and stops when it is smaller than the median
  右标一直向左移动，碰到比中值小的就停止
- Then exchange the left and right labeled data items
  然后把左右标所指的数据项**交换**

Continue to move until the left mark moves to the right side of the right mark, stop the movement, then the right mark is the position of the "median", finish the median and this position's exchange, the left half is smaller than the median, and the right half is larger than the median
继续移动，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完成，左半部比中值小，右半部比中值大

Quick sort: Fig

快速排序：图示





Quick sort: Code

快速排序：代码

```
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)
```

Basic End Conditions
基本结束条件

divisive
分裂

recursive
invocation
递归调用

Quick sort: Code

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
```

elect the Median Value
选定“中值”

eft and right standard initial valu
左右标初值

Move the left marker to the right
向右移动左标

Move to the left and to the right
向左移动右标

End the move
两标相错就结束移动

Left and right undermark value exchange
左右标的值交换

The median is in place
中值就位

Median point, which is also the split point
中值点，也是分裂点

Quick sort: Algorithm analysis
快速排序：算法分析

The quick sorting process is divided into two parts: split and move
快速排序过程分为两部分：分裂和移动

If the split always divides the data sheet into two equal parts, then it is the complexity of O (logn);
如果分裂总能把数据表分为相等的两部分，那么就是O(logn)的复杂度；

Move needs to compare each item to the median, still O(n)
而移动需要将每项都与中值进行比对，还是O(n)

Overall, it is O(nlogn);
综合起来就是O(nlogn)；

Also, no additional storage space is required during the running of the algorithm.
而且，算法运行过程中不需要额外的存储空间。

Quick sort: Algorithm analysis
快速排序：算法分析

However, if not so lucky, the split point where the median is is too far from the middle, causing an imbalance between the left and right parts

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

In extreme cases, some cases always have no data, so that the time complexity degrades to O (n^2)

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)

Plus the cost of recursive calls (even worse than bubble sorting)

还要加上递归调用的开销(比冒泡排序还糟糕)

Quick sort: Algorithm analysis
快速排序：算法分析

The median selection method can be appropriately improved, to make the median more representative

可以适当改进下中值的选取方法，让中值更具有代表性

For example, the "three-point sampling", selecting the median value from the head, end, and middle of the data table, will produce additional computational overhead, and extreme cases still cannot be ruled out

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

What other samples are representative of them?

还有什么采样具有代表性？