云南大学数学与统计学院
上机实践报告

课程名称： 数据结构与算法实验	**年级：** 2015 级	**上机实践成绩：**
指导教师： 陆正福	**姓名：** 刘鹏	
上机实践名称： 算法分析实验	**学号：** 20151910042	**上机实践日期：** 2017-04-10
上机实践编号： No.03	**组号：**	**上机实践时间：** 上午 3、4 节

一、 实验目的

1. 熟悉算法分析的基本概念与方法

二、实验内容

1. 实验研究法

   任取一些典型算法, 进行 Python 实现，模仿 3.1 研究其运行时间，分析算法效率。

2. 渐近分析法

   查阅 Python 文献，绘制图 3.4，体会渐近分析法中经常遇到的几种函数的增长率。

三、实验平台

Windows 10 1703 Enterprise 中文版；
Python 3.6.0；
Wing IDE Professional 6.0.5-1 集成开发环境；
MATLAB R2017b。

四、实验记录与实验结果分析

1 题

对典型算法的运行时间研究，要求用 Python 进行实现。

Solution:

由于目前接触的算法还不算很多，所以这里仅仅考虑不同的排序算法。典型的排序算法有冒泡排序（Bubble Sort）、选择排序（Selection Sort）、快速排序（Quick Sort）。我们这里选择这三种排序方法进行 Python 实现，然后运行，分析各自的运行时间，进而分析算法效率。这里采取的是实验分析法。

冒泡排序（Bubble Sort）算法解释：

冒泡排序采用的是循环方法。通过比较、交换，不断对原数组进行变化，最终得到有序的数组。

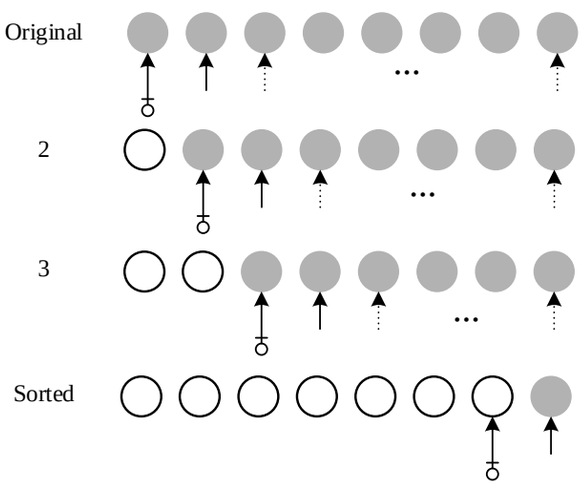


Figure 1

在 Figure 1 中，我们假设有 8 个无序的数字，现在我们需要用冒泡法进行由小到大排序。先定义一个指向，如带尾缀的箭头，它指向假定的最小值，然后建立另一个指向，它从该假想值之后的位置开始，遍历剩下的所有元素，如果遇到了比假想值小的元素，就互换假想值与该元素，然后认定该元素为新的假想值。如此一来，经过第一次循环，最左侧就是最小值。然后把带尾缀的箭头指向该表的第二个元素，将该算法运行于除最小值之外的所有元素上。最终尾缀箭头指向该数组的倒数第二个元素，这个时候再与最后一个元素相比，算法完成。

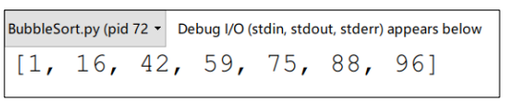
从上面的叙述中我们可以看到，当小箭头遍历完了剩余元素之后，才移动尾缀箭头，而且尾缀箭头的终点在倒数第二个元素位置上。而小箭头的起点都在对应的尾缀箭头的下一个位置上。由此写出 Python 代码。

程序代码

```
1   # filename: BubbleSort
2
3   def BubbleSort(L):
4       for i in range(0,len(L)-2):
5           for j in range(i,len(L)-1):
6               if L[i] > L[j]:
7                   L[i],L[j] = L[j],L[i]
8       return L
9
10  a = [1,96,88,75,42,16,59]
11  print(BubbleSort(a))
```

程序代码 1

运行结果



运行结果 1

代码分析：

对冒泡排序的复杂度的分析。如果我们将数组按照递增顺序排列，那么显然，最坏的情况就是，输入的数组是按照递减顺序输入的，即，每次选中一个假定数值，在对该假定数值之后的所有数据进行遍历尝试的时候，由于剩余的数组部分都是去除上一次循环所剔除的那个数值之后，剩余数组元素的相对原顺序，所以剩余数组保留了递减顺序。这就导致每一次的遍历尝试都是完整遍历。而完整遍历的结果就是每一个判断语句都是为 True，每一次都需要互换。这就导致了复杂度很高乃至是最高。一次判断、三次交换（虽然在 Python 中是一个语句，不过实际上还是如此进行，尤其是当元素不是数字，而是其他对象的时候），然后缩小规模（减 1）继续操作。故假设输入规模为n，那么第一次需要常量级操作$4(n-1)$次，之后第二次需要常量级操作$4(n-2)$次，……，一直到最终 4 次，那么总的常量级操作就是

$$4\sum_{i=1}^{n-1} i = 2(n^2-n),$$

总得来看，这是一个平方量级的最坏复杂度。

那么最好的情形，就是输入一个递增顺序的数组。这个时候，每一个判断都是假 False，很显然，这种清醒只是避免了三次交换，所以就是上式去掉乘数 4 而已，也就是$\frac{n^2-n}{2}$，也是一个$O(n^2)$量级的操作。所以，冒泡排序的复杂度，总是平方量级。即

$$\frac{n^2-n}{2} \leqslant \text{Complexity} \leqslant 2(n^2-n)$$

值得指出的是，就算是所有的数组元素都一样大，那么这些判断也是不可以省略的，仍然是最小量级

快速排序（Quick Sort）：

对于快速排序而言，它的实现基础是分治策略，首先判断一次，然后进行数组分段，对分得的两段进行递归。当然，只要遇到了递归，就需要多加注意，很有可能因为递归使得内存占用很大，最终导致系统崩溃。在 Python 中，系统把递归深度限制在了 900 层左右，防止因为内存铺张影响总体性能。

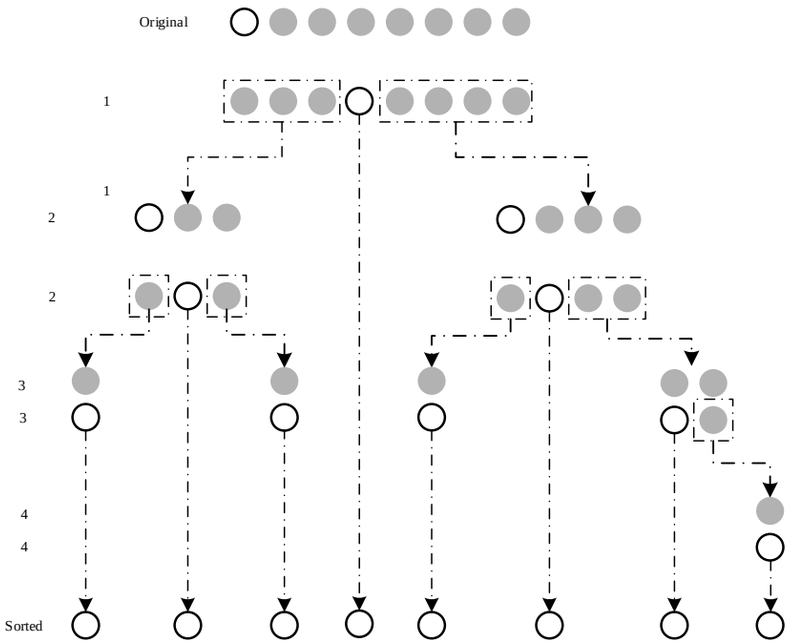


Figure 2

在 Figure 2 中我们可以看到快速排序的行为方式。白圈就是我们选定的一个基，在它的基础上，我们进行左右分类，分为比它小的，比它大的，与它相等的（包括它自己），然后把这个算法递归性地应用的两类不等的数组中。当数组元素为 1，结束划分。最后把这些合并起来，就是最后的有序数组。这个过程有很明显的递归性。

对冒泡排序的复杂度的分析。首先抛开复杂度分析，单纯对递归的过程进行内存占用分析。

假设我们输入了十万个递减排列的整数，要将它排列成递增顺序。首先调用 sys 模块的 getsizeof()方法，可以看到一个不算很大的整数（100000 以内）的内存占用是 28 个字节，即 28Bytes，那么十万个整数就是 2734KB，即 2.67MB。这在一般的计算机上还是很容易得到的，毕竟是大内存机器时代。然而，在我们的分组中，只有首元素被分配到 Middle 数组中，剩下的元素都比首元素小，所以都在 Left 数组中（注意，这是一个新的数组，用 append 操作一个个扩充进去的，几乎是复制了一遍内存中的原数组），而 Right 数组为空。接下来对 Left 数组进行快速排序的递归调用。整个第一步，仅仅是容量减 1，变成了 99999 个数值。就这样一直进行下去，直到数组容量变为 1。很遗憾，这个减少量在这种情形下实在太过于微不足道，所以，即使在进行了 1000 次递归之后，数组还有 99000 个需要排列的，而这个时候，内存已经用了接近 2.6GB，简直恐怖。难以想象以后会怎么样。

这里引出了我之前的操作与之后操作的不同。之前我随机输了 10 个数字进去，然后在编辑器里面进行全选、拷贝、粘贴，最后扩充到了十万，这样一来数据就比较均匀，Left 与 Right 是相似的大小。再次分割，也几乎是对半分，这样的结果直接导致数组大小不停除以二，很快就变到 1，从下面的这个程序中我们可以看到，对于一种最坏情形，这个快速排序几乎是非常愚蠢的行为，仅仅 3500 个整数，就耗时一秒钟。

程序代码

```
1   # Quick Sort for a list
2
3   from time import time
4   import sys
```

```
5   sys.setrecursionlimit(1000000)
6
7   def QuickSort(L):
8       L_Left = []
9       L_Right = []
10      L_Middle = []
11      if len(L) <= 1:
12          return L
13      else:
14          for i in L:
15              if i > L[0]:
16                  L_Left.append(i)
17              elif i < L[0]:
18                  L_Right.append(i)
19              else:
20                  L_Middle.append(i)
21      L_Left = QuickSort(L_Left)
22      L_Right = QuickSort(L_Right)
23      return L_Left + L_Middle + L_Right
24
25  A = list(range(3500))
26  print("Length of list a is: ",len(A))
27  begin = time()
28  QuickSort(A)
29  end = time()
30  print("Time: ",end - begin)
```

程序代码 2

运行结果

```
QuickSort.py (pid 130 ▾   Debug I/O (stdin, stdout, stderr) appears below
Length of list a is:  3500
Time:  1.086442470550537
```

运行结果 2

在实验体会里，我对代码进行了修改，针对枢纽元的选择进行了重新的审视，因为要考虑的情形很多都不是随机均匀的。之前的十万个数字，实在是一个错误的例子。

总结

基于上面的两种算法，我们用十万个数字对他们进行排序速度测试。由于把数据写入文件再调用，会造成硬盘读写的干扰，为了最大化降低干扰，我们直接把数据写在代码里，运行的时候相当于直接调用内存中的文件，速度会快很多，干扰也会小很多。

在导入了 time 模块之后，Python 也可以像 C 语言编译一样，查看运行时间。然而，遗憾的是，Python 在进行排序的时候，会返回不了。当输入量是 10, 000 左右的时候，快速排序会很好地执行

2 题

查阅 Python 文献，绘制图 3.4，体会渐近分析法中经常遇到的几种函数的增长率。

Solution:

To sum up, Table 3.1 shows, in order, each of the seven commo functions used in algorithm analysis.

constant	logarithm	Linear	n-log-n	quadratic	cubic	exponen-tial
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 3.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or *n*-log-*n* time. Algorithm with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 3.4.

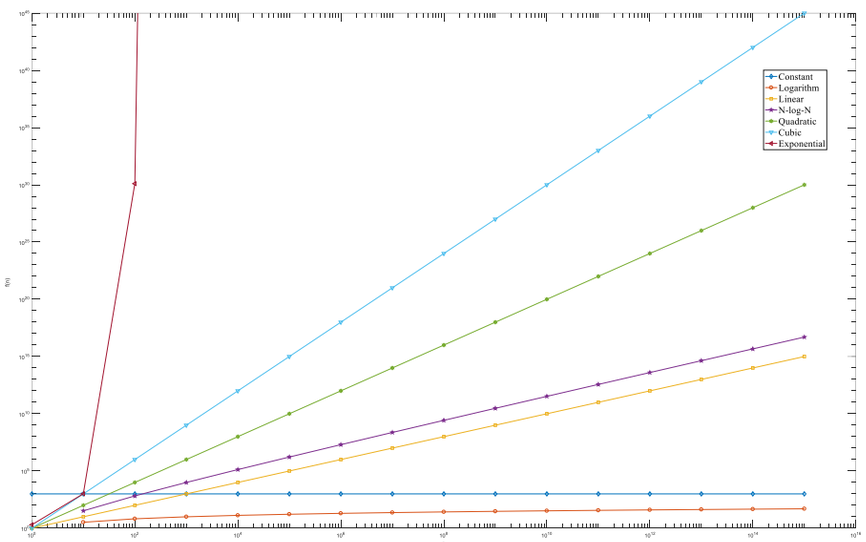


Figure 3.4: Growth rates for the seven fundamental functions used in algorithm used in algorithm analysis. We use base a = 2 for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

上面的图标是用 MATLAB 做出的。MATLAB 代码如下：

程序代码

```
%% 输入自变量 n 的数量级
n = logspace(0,15,16);

%% 定义函数
f_constant = n - n + 10e2;
f_logarithm = log2(n);
f_linear = n;
f_nlogn = n .* log2(n);
f_quadratic = n.^2;
f_cubic = n.^3;
f_exponentional = 2.^n;

%% 作图
loglog(n,f_constant,'-d');
hold on;
```

```
16  loglog(n,f_logarithm,'-o');
17  loglog(n,f_linear,'-s');
18  loglog(n,f_nlogn,'-p');
19  loglog(n,f_quadratic,'-h');
20  loglog(n,f_cubic,'-v');
21  loglog(n,f_exponentional,'-<');
22  axis([1 10e15 1 10e44]);
23  xlabel('n');
24  ylabel('f(n)');
25  hold off
```

程序代码 3

五、教材翻译

Translation

Chapter 3 Algorithm Analysis

＊第三章 算法分析

In a classic story, the famous mathematician Archimedes was asked to deter- mine if a golden crown commissioned by the king was indeed pure gold, and not part silver, as an informant had claimed. Archimedes discovered a way to perform this analysis while stepping into a bath. He noted that water spilled out of the bath in proportion to the amount of him that went in. Realizing the implications of this fact, he immediately got out of the bath and ran naked through the city shouting, “Eureka, eureka!” for he had discovered an analysis tool (displacement), which, when combined with a simple scale, could determine if the king’s new crown was good or not. That is, Archimedes could dip the crown and an equal-weight amount of gold into a bowl of water to see if they both displaced the same amount. This discovery was unfortunate for the goldsmith, however, for when Archimedes did his analysis, the crown displaced more water than an equal-weight lump of pure gold, indicating that the crown was not, in fact, pure gold.

＊在一个经典故事中，著名的数学家阿基米德被要求确定国王金冠是否是由纯金打造，而不是掺杂了部分白银。阿基米德在泡澡时发现了一种分析的方法。他指出，他的身体浸入澡盆水面越多，溢出的水量越大，而且两者严格相等。意识到这个事实，他马上脱身洗澡，赤身裸体穿过这个城市大喊：“尤里卡，尤里卡！”因为他发现了一个分析方法（排水量），当与排水容量进行简单结合时，就可以确定王冠是否是纯金的。也就是说，阿基米德可以将王冠和等量的纯金分别浸入水中，看看排水量是否相等。这个发现对于金匠来说是不幸的，当阿基米德做了他的分析时，皇冠比同等块纯金溢出了更多的水，这表明皇冠实际上并不是纯金。

In this book, we are interested in the design of “good” data structures and algorithms. Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some task in a finite amount of time. These concepts are central to computing, but to be able to classify some data structures and algorithms as “good,” we must have precise ways of analyzing them.

＊在本书中，我们关心的是“好的”数据结构和算法。简单地说，数据结构是一种组织和访问数据的系统方式，算法是在有限的时间内执行一些任务的一步一步的过程。这些概念是计算的核心，但为了能够将一些数据结构和算法定义为“好”，我们必须有精确的分析方法。

The primary analysis tool we will use in this book involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of “goodness,” since time is a precious resource—computer solutions should run as fast as possible. In general, the running time of an algorithm or data structure operation increases with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by the hardware environment (e.g., the processor, clock rate, memory, disk) and software environment (e.g., the operating system, programming language) in which the algorithm is implemented and executed. All other factors being equal, the running time of the same algorithm on the same input data will be smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation. We begin this chapter by discussing tools for performing experimental studies, yet also limitations to the use of experiments as a primary means for evaluating algorithm efficiency.

＊在本书中使用的主要分析工具包括：算法运行时间、数据结构的操作方法、内存的占用。运行时间是判别是否“良好”的自然尺度，因为时间是宝贵的资源，计算机解决问题应尽可能快。通常，算法或数据结构操作的运行时间随着输入大小而增加，尽管对于相同大小的不同输入，运行时间也会不同。此外，运行时间受实施和执行算法的硬件环境（例如，处理器，时钟速率，存储器，磁盘）和软件环境（例如，操作系统，编程语言）的影响。所有其他因素相同，如果计算机具有比较快的处理器，或者如果是被编译为机器码、而不是在解释性程序中完成了实现，则相同输入数据上相同算法的运行时间将更短。我们从实验研究开始本章，但也将尽量避免使用实验方法作为评估算法效率的主要手段。

Focusing on running time as a primary measure of goodness requires that we be able to use a few mathematical tools. In spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input. We are interested in characterizing an algorithm’s running time as a function of the input size. But what is the proper way of measuring it? In this chapter, we “roll up our sleeves” and develop a mathematical way of analyzing algorithms.

＊为了将运行时间作为主要衡量标准的要求，我们需要使用几种数学工具。忽略来自不同环境因素的变化，我们重点关注算法的运行时间与其输入的大小之间的关系。我们把算法的运行时间视为输入量的函数。但是什么是衡量它的正确方法？在本章中，我们“撸起袖子”，开发一套分析算法的数学方法。

3.1 Experimental Studies

＊实验研究

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the time spent during each execution. A simple approach for doing this in Python is by using the time function of the time module. This function reports the number of seconds, or fractions thereof, that

have elapsed since a benchmark time known as the epoch. The choice of the epoch is not significant to our goal, as we can determine the *elapsed* time by recording the time just before the algorithm and the time just after the algorithm, and computing their difference, as follows:

＊如果一个算法已被实现，我们可以通过各种输入测试来记录其运行时间。在 Python 中执行此操作的简单方法是使用 time 模块的 time()函数。此函数此时时刻的秒数或者分数，数据来源是系统时间。时刻的选择对我们的目标并不重要，因为我们可以通过记录算法运行之前的时间和算法结束之后的时间来确定经过的时间，并计算它们的差，如下所示：

```
from time import time
start time = time( )                    # record the starting time
run algorithm
end time = time( )                      # record the ending time
elapsed = end time − start time         # compute the elapsed time
```

We will demonstrate use of this approach, in Chapter 5, to gather experimental data on the efficiency of Python's list class. An elapsed time measured in this fashion is a decent reflection of the algorithm efficiency, but it is by no means perfect. The time function measures relative to what is known as the "wall clock." Because many processes share use of a computer's ***central processing unit*** (or ***CPU***), the elapsed time will depend on what other processes are running on the computer when the test is performed. A fairer metric is the number of CPU cycles that are used by the algorithm. This can be determined using the clock function of the time module, but even this measure might not be consistent if repeating the identical algorithm on the identical input, and its granularity will depend upon the computer system. Python includes a more advanced module, named timeit, to help automate such evaluations with repetition to account for such variance among trials.

＊我们将在第 5 章中演示使用这种方法来收集 Python 列表类的效率数据。以这种方式测量的经过时间是算法效率的一个有效的反映，但并不完美。时间功能依靠"系统时间"来测量。由于许多进程共享计算机的中央处理单元（CPU），所以经过的时间将取决于计算机上运行的其他进程。更公平的度量是算法使用的 CPU 周期数。这可以使用 time 模块的 clock 函数来确定，但即使在相同的输入上重复相同的算法，即使这种措施可能不一致，其差异将取决于计算机系统。Python 涵盖一个更高级的模块，名为 timeit，以自动化地重复这样的评估，以考虑试验之间的这种差异。

Because we are interested in the general dependence of running time on the size and structure of the input, we should perform independent experiments on many different test inputs of various sizes. We can then visualize the results by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size, n, and y-coordinate equal to the running time, *t*. Figure 3.1 displays such hypothetical data. This visualization may provide some intuition regarding the relationship between problem size and execution time for the algorithm. This may lead to a statistical analysis that seeks to fit the best function of the input size to the experimental data. To be meaningful, this analysis requires that we choose good sample inputs and test enough of them to be able to make sound statistical claims about the algorithm's running time.

＊我们致力于研究运行时间关于数据规模和数据结构的关系，所以我们应该对许多不同规模的不同输入进行独立的实验。然后，我们可以画一个二维坐标图，横轴 x 坐标表示输入大小，纵轴 y 坐标表示运行时间。图 3.1 显示了这样的关系。这种可视化操作可以提供关于算法的输入和执行时间之间的关系的一些直觉的形成。可以通过统计分析，确定与输入规模所匹配的最佳函数。为了有意义，这种分析要求我们选择良好的样本输入，并对此进行足够的测试，以便对算法的运行时间进行有效统计。

3.1.1 Moving Beyond Experimental Analysis

＊

END

六、实验体会

根据书中给出的分析方式，可以初步分析一些带有循环的算法的时间复杂度。但是有些还是很难自己证明。这有待进一步的学习。

在快速排序中，我曾经写过一段代码，直接导致了很错误的情况。数据结构与算法分析：C 语言描述（原书第二版）的 Page179，直接指出了这种愚蠢的做法。

> **一种错误的方法**
>
> 通常的、没有经过充分考虑的选择是将第一个元素用作枢纽元。如果输入是随机的，那么这是可以接受的，但是如果输入是预排序的或是反序的，那么这样的枢纽元就产生一个劣质的分割，因为所有的元素不是都被划入S_1就是都被划入S_2。更有甚者，这种情况可能发生在所有的递归调用中。实际上，如果第一个元素用作枢纽元而且输入是预先排序的，那么快速排序花费的时间将是二次的，可是实际上却根本没干什么事，这是相当尴尬的。然而，预排序的输入（或具有一大段预排序的输入）是相当常见的，因此，使用第一个元素作为枢纽元是绝对糟糕的主意，应该立即放弃这种想法。另一种想法是选取前两个互异的关键字中的较大者作为枢纽元，不过这和只选取第一个元素作为枢纽元具有相同的害处。不要使用这两种选取枢纽元的策略。
>
> **一种安全的做法**
>
> 一种安全的方针是随机选取枢纽元。一般来说这种策略非常安全，除非随机数生成器有问题（它不像你可能想象的那么罕见），因为随机的枢纽元不可能总在接连不断地产生劣质的分割。另一方面，随机数的生成一般是昂贵的，根本减少不了算法其余部分的平均运行时间。
>
> **三数中值分割法**
>
> 一组 N 个数的中值是第[N/2]个最大的数。枢纽元的最好的选择是数组的中值。不幸的是，这很难算出，且明显减慢快速排序的速度。这样的中值的估计量可以通过随机选取三个元素并使用他们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。。例如，输入为 1，1，4，9，6，3，5，，7，0，它的左边元素是 8，右边元素是 0，中心位置（$\lfloor (Left + Right)/2 \rfloor$）上的元素是 6.于是枢纽元则是$v = 6$。显然使用三数中值分割法消除了预排序输入的坏情况（在这种情况下，这些分割都是一样的），并且减少了快速排序大约 5%的运行时间。

Python 的强大便捷，使得最后的三数中值分割变得很简单。对代码进行稍微的改动就可以了。

```
# Quick Sort for a list

from time import time
import sys
sys.setrecursionlimit(1000000)

def QuickSort(L):
    L_Left = []
    L_Right = []
    L_Middle = []
    if len(L) <= 1:
        return L
    else:
        for i in L:
            pivot = (L[0] + L[-1] + L[len(L)//2])/3
            if i > pivot:
                L_Left.append(i)
            elif i < pivot:
                L_Right.append(i)
```

```
            else:
                L_Middle.append(i)
    L_Left = QuickSort(L_Left)
    L_Right = QuickSort(L_Right)
    return L_Left + L_Middle + L_Right

A = list(range(3500))
print(A)
print("Length of list a is: ",len(A))
begin = time()
A = QuickSort(A)
end = time()
print("Time: ",end - begin)
print(A)
```

程序代码 4

```
QuickSort.py (pid 103 ▾   Debug I/O (stdin, stdout, stderr) appears below   Options ▾
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17
Length of list a is:  3500
Time:  0.019082307815551758
[3499, 3498, 3497, 3496, 3495, 3494, 3493, 3492, 3491, 3490,
```

运行结果 3

可以看到，这个程序在改动 pivot 的意义之后，速度变得明显快了很多。之前需要一秒多的运行时间，这里只需要 0.02 秒，提升了 50 倍！

七、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python
[2] 数据结构与算法分析：C 语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社[i]
[3] 算法导论（原书第三版），（美）科尔曼（Cormen，T.H.）等；殷建平等译. 北京：机械工业出版社