编号：_____

高级数据结构与算法设计

作业 1：排序算法的可视化

学　　院：计算机与通信技术学院

学　　号：G20188751

姓　　名：潘秋实

指 导 老 师：皇甫伟

得　　分：_____

2019 年 03 月 30 日

目录

摘要

为了学习几种基础的排序算法，本文使用 Python 语言实现了快速排序、希尔排序、归并排序和堆排序算法，并且实现了算法的运行过程和结果的可视化。本文实验观察到了四种排序算法，在不同长度的随机数列上的比较、移动次数和运行时间。综合来看，在足够长的序列上，堆排序是最快的，而希尔排序比其他三种算法要慢很多。

1 引言

排序算法一直是算法学习者最开始要学习的问题，而深入理解几种经典排序算法的原理，也非常有利于编程思维的养成。排序算法最优的时间复杂度是$O(n\log n)$，所以本文选取了几种在这个时间复杂度上，比较经典的排序算法来实现。除了希尔排序的时间复杂度是$O(n^{1.3\sim2})$，快速排序、归并排序和堆排序的时间复杂度都符合。

衡量排序算法的优劣，应在一定的局限范围内。由数组长度、元素排列等方面的差异，可以使这几种排序方式有不同的表现。所以本文的工作主要包含以下方面：

1）使用 Python 编程实现任意长度数组的快速排序、希尔排序、归并排序和堆排序算法；

2）使用 OpenCV 展示快速排序的过程，即指定任意一个数组，每个元素以唯一的颜色、固定长度的柱形显示，随着排序的过程变换位置，最后记录成 avi 格式的视频；

3）针对不同长度（N）的数组，分别进行多次试验，记录在四种排序的运行过程中平均的比较次数（C）和移动次数（M）。使用 Matplotlib 绘制 N-C、N-M 曲线，并且生成 Excel 表格记录下来。

4）进行多次试验，记录四种算法在不同长度（N）数组上的平均运行时间（T）。使用 Matplotlib 绘制 N-T 曲线，记录成 Excel 表格，并结合 N-C、N-M 曲线分析。

2 相关算法原理

四种排序算法的原理已经被世人所熟知，在本章中即集合本文实验中用到的核心思路，简单介绍一下四种算法的思想。

2.1 快速排序

快速排序对于每个区间来说，都是寻找一个基准（basic），将其他元素

分成大于等于基准和小于基准的两部分，然后再对这两部分分别实施快速排序，最后放在基准两边，形成一个排序好的数组。具体来说，就是将第一个元素作为基准，使用两个指针分别从前和后查找。当前方的指针找到一个大于基准的数，并且后方的指针找到一个小于基准的数，就将这两个数互换位置。

2.2 希尔排序

希尔排序实际上是对插入排序的一种改进，利用了插入排序在小数组和基本有序的数组上表现比较好的优点。希尔排序将整个数组分成 n 端，每隔 n 个元素被划分到同一小数组中，进行插入排序。将这 n 个小数组排序结束后分别放回原占用的位置，再分成更大一点的多个数组插入排序。比较常见的是每次把小数组的长度设置成上一次的 2 倍。

2.3 归并排序

归并排序实际上是一种先分再合的思想。首先将整个数组分成2^n个子数组，每个数组自行排序。再两两合并成一个数组，方法是将两个数组的最小值依次比较、放入新的数组中。层层合并，类似于二叉树的结构，从“叶子结点”开始，逐步产生“父节点”，一直合并到“根节点”，也就是最终排序得到的数组。

2.4 堆排序

堆排序是将数组序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余 n-1 个元素重新构造成一个堆，这样会得到 n 个元素的次小值。如此反复执行，便能得到一个有序序列了

3 实现过程

本文的所有代码都是自行编写完成，尽管浏览过一些网上的简易实现代码，本文的实现过程还是严格按照定义来做的。本章即是介绍实现过程中的一些关键问题。

3.1 排序算法实现

网上的快速排序都是确定一个基准，直接从数组的一边开始查找，把比基准大的数放到一个新的数组中，比基准小的数组放到另一个数组中，而不是采用同一数组中互换的形式。

我的实现代码，在确定基准数值 basic 以后，从目标数组的两端交替查找，直到将将数组分成比 basic 大（或等于）和 basic 小的两部分，将 basic 插到中心再对这两部分分别排序。这一部分知识在本科的课程中曾经学过，而各类工具书中也反复介绍，所以我对快速排序算法的印象比较深刻，实

现起来很轻松。遇到的唯一问题是：我习惯以排序区间的第一个值为 basic，排序的过程中，小于 basic 的元素移到左边，大于等于 basic 的元素移到右边;当排序中的 i 和 j 相交的时候，将交点与 basic 的值交换，从而出现了问题。后来发现，将交点的值与 basic 交换，在 basic 的值等于交点的时候，会使左侧出现不小于 basic 的值，从而导致出错。所以我在最终实现的时候，都是将 basic 的值与交点的前一个值交换。

希尔排序的实现比较轻松，我用 log 计算了一下数组长度，由此做划分，实现了层层的插入排序。在做归并排序的时候，我本来打算做类似于二叉树的指针结构，后来发现这对存储是一个巨大的考验，所以还是采用了递归调用的方式。将数组不断划分，直到长度剩一个节点为止，借助“完全二叉树”的思想，就顺利做了出来。而堆排序则主要在二叉树位置变换上，我参考了一下本科的 PPT，使用了压缩存储的方式，将二叉树存到了一个数组中，可以使用位置确定一个节点的父节点和子节点，从而成功实现。

3.2 图形化显示过程

本来 python 有好多模块是专门画图用的。但是这次要求不同颜色和相应长度的柱形图，并且记录成视频，所以我专门使用 OpenCV 封装了一个方法：显示的柱形可以根据元素的数量调整宽度，根据最大元素的绝对值调整所有柱形的高度，并且每个元素都绑定了一个随机生成的颜色。在一个固定大小的画面上，随着元素位置的改变，相应柱形的高度和颜色是不变的。其次，在排序中参与排序的区间会用红框圈出，排序中的参考量 basic、i 和 j 也会被绘制在相应柱形的下方，充分反映排序的过程。OpenCV 绘制完毕图像后，可以将图像展示出来，并且记录成 avi 格式的视频；所以每次运行程序，即使是同一个数组，每个元素对应的颜色也是不同的，并且都会生成一个新的结果视频文件。

3.3 记录测试数据

针对四种排序算法，我编写了两个测试脚本：一个是测试程序运行过程中的比较次数 C 和移动次数 M;一个是测试不同算法针对同一数组的运行时间。因为在算法执行过程中记录操作次数，加法操作回应性算法速度，所以这两个脚本不能合二为一。

针对测量操作次数的脚本，程序在运行的时候，每执行一次操作，相应的参数会自增 1，并且动态传递。测试的每一轮，程序都会生成一个随机数组，使用四种排序算法对其排序，记录操作次数。对每种长度的数组，都会进行多轮测试，取多轮的平均操作数作为结果，绘制到最终的图表中。而测量时间的脚本，则是使用了一个不计数版本的函数库，直接使用 python 的

time 模块在算法执行前和执行后计数。每轮中，四种算法针对同一随机数组排序，取多轮的平均时间值作为该种算法的消耗时间，最终记录下来。

当执行完所有长度的数组后，程序会调用 Matplotlib 模块绘制 N-C、N-M、N-T 折线图，并保存成 png 格式的图片。调用 pettytable 把结果在命令行中打印成表格，使用 xlwt 输出成 xls 的文件。

4 测试

我在 Ubuntu 16.04 系统中完成了测试实验。测试平台搭载的处理器是 Intel i7-8700k，主频在 4.4GHz 左右，该算法运行的结果，应根据其他平台处理器性能作为参考。

4.1 验证快速排序算法

在我的工程中，指定一个待排序数组，运行 demo_quicksort.py 脚本，可以通过 OpenCV 展示数组排序的过程，如图 4-1 所示。

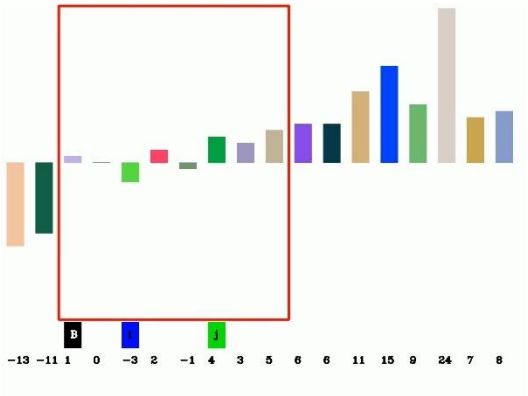


图 4-1 快速排序过程截图

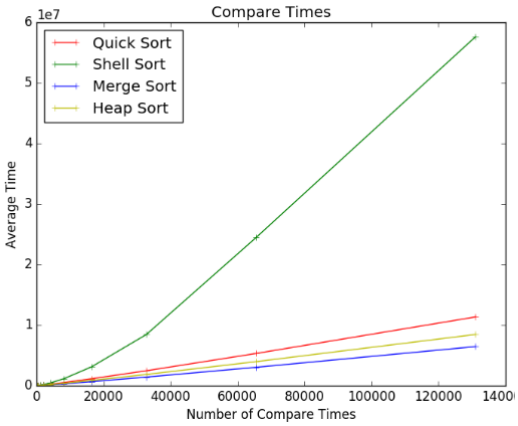
每个元素和其对应的柱形，会随着排序的过程移动。同时在图上用红框圈出当前参与排序的区间，并且标注 basic、i 和 j 三个指针的位置。排序结束后，会在 output 文件夹中生成 QuickSort.avi 视频文件来记录过程。

4.2 测试操作次数

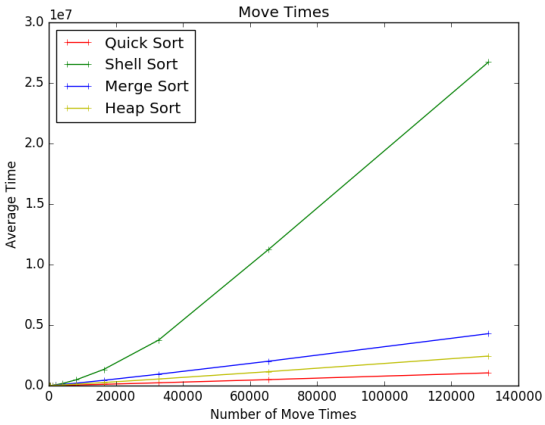
我将四种排序算法封装到了 lib 文件夹的四个库文件中，给这四个方法传递一个待排数组作为参数，会得到排序好的序列、比较次数和移动次数。

demo_test_operate_times.py 是操作次数的测试脚本。我设定从长度为 4 的序列开始，每次长度是上一次的两倍，一直测试到 131072。对于每一种长度，测试 4 种算法，在相同 100 个随机数列上的平均操作次数。程序运行结束后，会使用 Matplotlib 工具绘制三个 N-M、N-C 和 N-Total 三个结果图，保存到 output 文件夹中，并且会在命令行中将具体的数值结果打印出来，并保存到 TestResult.xls 文件来。

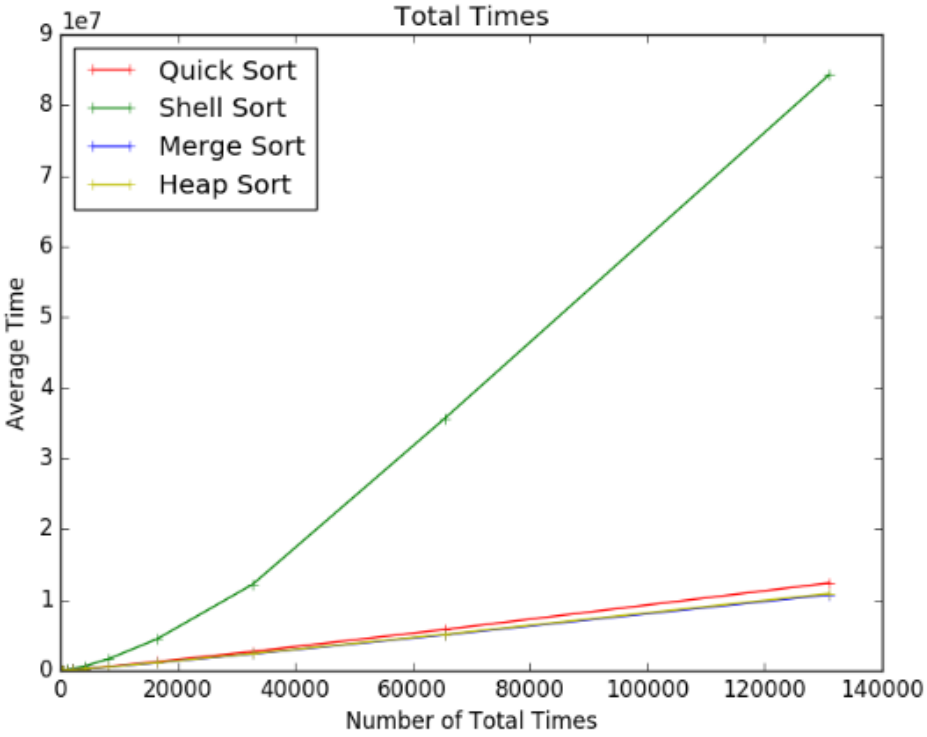
通过实验，得到的比较次数 C 如图 4-2(a)所示，数值结果记录在附录的表 1 中；移动次数 M 如图 4-2(b)所示，数值结果记录在附录的表 2 中；而总操作次数如图 4-2(c)所示，数值为 C 与 M 的和。



(a) 比较次数



(b) 移动次数



(c) 总操作次数

图 4-2 四种算法的操作次数的测试平均值

由此可见，除了希尔排序，其余三种算法的操作次数都是差不多的。在长度比较小的情况下，四条曲线会出现交错的情况；而当数组长度达到一定长度的时候，归并排序的比较次数最少，快速排序的移动次数最少。而通过附录 1-3 的表可以看出来，快速排序、归并排序和堆排序的操作次数，随着数组长度的增加，复杂度是呈$O(n\log n)$增加的，希尔排序则是呈指数型增长。但是操作总次数并不代表时间长度。因为比较和移动操作所占用的时间是不一样的，我们在下一节

中会具体分析。

4.3 比较时间复杂度

在工程中，用于测试时间复杂度的是 demo_test_perform_time.py 脚本，它调用了 lib 中的 onlysort 库。onlysort 库也是我自己编写的，其中的接口名称虽然和 4.2 中用到的排序接口一样，但是在执行的过程中不计算操作次数，最后只返回排序好的数组。

在测试中，我依然使用了长度从 4 到 131072、每种长度分别随机生成的 50 个数组，通过计算这些数组使用 4 种排序算法处理的平均时间，来得出 4 中排序算法的时间复杂度。时间复杂度的结果依然通过 Matplotlib 打印出来，如图 4-3 所示，数值结果记录在 output 文件夹的 CostTime.xls 中，如表 4-1 所示。

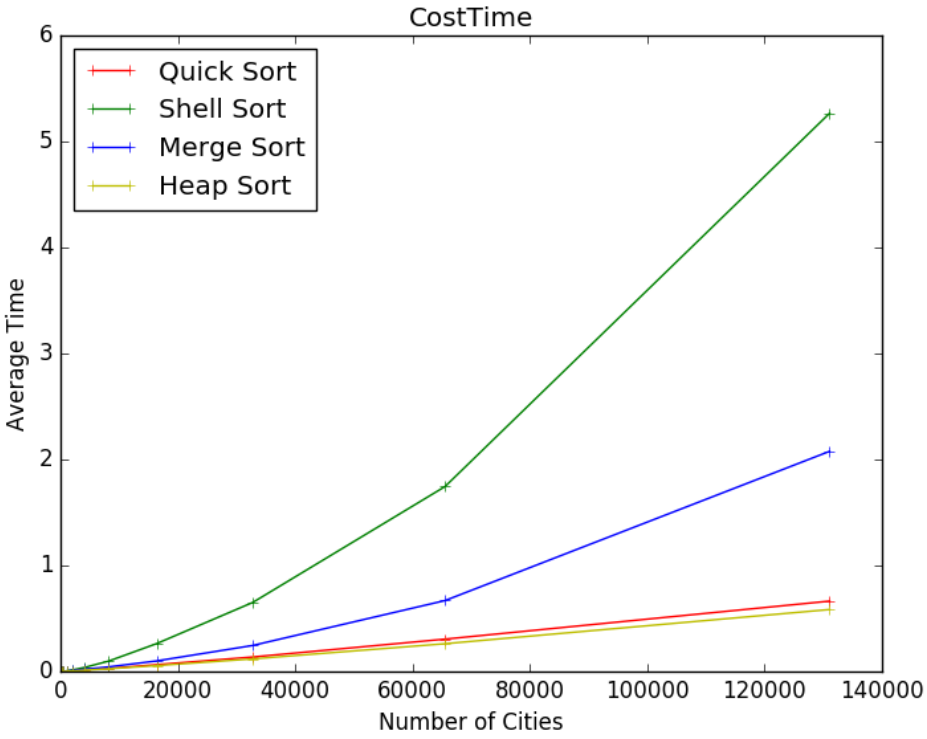


图 4-3 4 种排序算法针对不同长度随机数组的平均处理时间

其中，我们可以直观看出，当数组长度超过一定数值以后，4 种排序算法的时间复杂度是比较稳定的。在 4.2 节中我们得出结论，希尔排序的操作数量和其他三种算法不在同一数量级，所以用希尔排序和其他三种作对比的结果是显而易见的，并且意义不大。

而对于这三种比较快的算法而言，归并排序的比较次数最少，移动次数比快速排序多几倍，因为移动操作需要比比较操作花费更多的时间，所以归并排序花费的时间是最多的。但即

便堆排序的移动次数比快速排序要多一些，因为整体操作次数少的缘故，依然比快速排序要快一些。

表 4-1 4 种算法的平均处理时间

Length	Quick	Shell	Merge	Heap
4	0	0	0	0
8	0	0	0	0
16	0	9.75E-05	0.000103	0
32	0.0001	0.0001	0	0
64	9.97E-05	0.0001	0.0002	0.0001
128	0.0001	0.000301	0.000602	0.000201
256	0.000501	0.000805	0.000796	0.000501
512	0.000997	0.002119	0.001799	0.001204
1024	0.002801	0.005926	0.004109	0.002501
2048	0.006014	0.015345	0.009123	0.005208
4096	0.017749	0.038099	0.020755	0.012534
8192	0.029084	0.097756	0.043616	0.025671
16384	0.064366	0.262898	0.099566	0.055655
32768	0.13687	0.651529	0.245656	0.118714
65536	0.305421	1.744844	0.669272	0.2622
131072	0.664085	5.266601	2.077106	0.584663

值得注意的是，在实验中，我是通过 Python 的 time 模块来对排序算法计时。即在排序算法开始之前和之后读取系统时间。因为数组短的时候排序太快了，所以前几行打印出来的结果都是 0。但我们依然可以通过附录中的操作次数来判断时间复杂度。

5. 总结

在本次作业中，本文顺利进行了快速排序、希尔排序、归并排序和堆排序的实现、计算操作数和验证时间复杂度等工作，并且将工作成果使用图形化展示。因为整个工程的代码高达一千多行，所以就使用了建库与调用的方式，在文中就不便于粘贴代码，所以在这里只简单介绍一下工程目录，如图 5-1 所示。

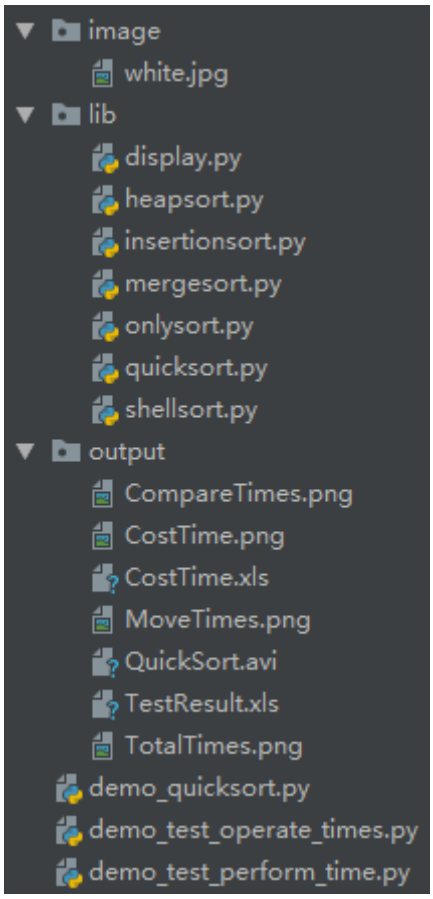


图 5-1 工程目录

根目录中三个脚本，分别对应第四章中三节的实现；lib 文件夹中，有测试所需的各种库文件，直接被调用即可；output 文件夹则是所有测试结果的输出文件夹，保存程序运行的所有可视化结果。

附录

表 1 4 种算法在不同长度数组上的平均比较次数

Length	Quick Sort	Shell Sort	Merge Sort	Heap Sort
4	27.9	28	20.4	28.9
8	90.3	87	60.2	81.3
16	240	255.4	165.9	214.3
32	670.5	684	423.5	539.7
64	1658.1	1750.6	1042.2	1329.4
128	3949.2	4598.8	2462.3	3159.1
256	9016.5	11415.6	5679.6	7306.7
512	20658.3	28429.8	12896	16654.8
1024	48184.2	75131.2	28900.5	37353.9
2048	106497.9	178118.2	63913.4	82960.9
4096	238008	449994.8	140119.2	182209.5
8192	527408.4	1160735.4	304745.9	397086.6
16384	1138885.2	3191833.8	658730.7	859807.6
32768	2463067.2	8030214	1415860.4	1850449.6
65536	5272321.8	22751111.6	3028282.2	3963364.1
131072	11235920.7	57412661.2	6449227.7	8450966

表 2 4 种算法在不同长度数组上的平均移动次数

Length	Quick Sort	Shell Sort	Merge Sort	Heap Sort
4	5.6	7.5	12.8	17.1
8	13.4	24.5	39.4	40.3
16	32.8	76.2	109.3	93.6
32	80.6	210	280.5	213.4
64	193.8	550.8	689.4	490.8
128	434.6	1526.4	1632.1	1110.8
256	997	3910.3	3771.2	2462.5
512	2215.2	10112.9	8566	5439.4
1024	4933	28343.1	19191.5	11888.9
2048	10748.8	68572.1	42467.8	25832.6
4096	23418.6	179933.9	93128.4	55744.7
8192	50419.4	482055.7	202617.3	119648.2
16384	108479.6	1382916	438030.9	255727.5
32768	231972.6	3556346	941628.8	544128.8
65536	494325.8	10392506	2014313	1153974
131072	1053295	26609169	4290586	2438900