排序算法实验报告

姓名：覃果　学号： 2020012379　班级：软件02

2022 年 3 月 26 日

摘要

实现了插入排序，希尔排序，快速排序，归并排序，基数排序以及 timsort共 6 个排序算法。

并比较了它们对在 $[0\,2^{32}-1]$ 范围内随机生成的规模分别为 $10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 2*10^8$ 的整数的排序效果。

1　实验环境

CPU: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz

OS: ubuntu

Python interpreter: Python 3.8.0

2　算法分析

2.1　插入排序

2.1.1　算法过程

对每个元素，将其插入到前面有序序列中的合适位置，从而使有序的范围越来越长，直到整个数组有序。

2.1.2　算法分析

时间复杂度： $O(n^2)$

空间复杂度： $O(1)$

2.1.3 算法评价

其复杂度相较其他算法较慢，但是在数据规模小的时候，由于它算法的简单性，它的速度是最快的。

2.2 希尔排序

2.2.1 算法过程

对插入排序进行了优化，每隔step个分为一组，组内进行插入排序，之后逐渐降低step至1。此时数组完全有序。

2.2.2 算法分析

时间复杂度：选择不同的step序列，复杂度为 $O(n)$ 到 $O(n^2)$ 之间
空间复杂度：$O(1)$

2.2.3 算法评价

突破了2次方的算法，在希尔排序出现之后，许多快速的算法相继出现

2.3 快速排序

2.3.1 算法过程

选择某个元素作为轴，将数组分为两个部分，之后递归处理两个子问题

2.3.2 算法分析

时间复杂度：期望复杂度$O(nlgn)$
空间复杂度：空间复杂度主要由函数栈导致，最优$O(lgn)$，最坏$O(n)$

2.3.3 算法评价

平均性能最好的排序算法，被广泛地使用

2.4 归并排序

2.4.1 算法过程

将数组划分为长度相等的两个部分，先递归地排好子问题，之后对两个子问题进行归并，保持其有序性

2.4.2 算法分析

时间复杂度：$O(nlgn)$

空间复杂度：最多使用的辅助空间就是最后一个归并的时候，故为$O(n)$

2.4.3 算法评价

达到了基于比较的排序算法的渐进下界，有许多应用，比如计算逆序对的个数

2.5 基数排序

2.5.1 算法过程

依次对每一位进行计数排序，从最低位到最高位，排完最高位后数组整体有序

2.5.2 算法分析

时间复杂度：若数据最高为 d 位二进制数，每次选择 r 位数进行计数排序，复杂度为 $O(\frac{b}{r}(n+2^r))$

空间复杂度：计数排序需要用的桶的大小，为$O(2^r)$

2.5.3 算法评价

取 $r=lgn$ 时，算法的复杂度可达到 $O(n)$，排序效果十分理想

2.6 timsort

2.6.1 算法过程

当数据个数比较少的时候，采用二分查找的插入排序。

当数据比较多的时候，会计算出一个最佳的 runLen。先寻找数组中有序或者逆序的块，如果块的大小小于 runLen，采用插入排序补全至runLen。

之后将数据块的大小以及长度压入栈中，每次压入的时候使栈保持栈不等式 $A>B+C$fi$B>C$，其中 A, B, C 分别为栈第三高，第二高和栈顶，代表的是相应数据块的长度

每当栈不满足上述不等式的时候，进行合并块。

合并的时候采用优化的归并方法。具体过程比较复杂

2.6.2 算法分析

时间复杂度：最坏$O(nlgn)$，最好 $O(n)$，平均$O(nlgn)$

空间复杂度：$O(n)$

2.6.3 算法评价

算法设计复杂，利用了数据本身可能具有的有序性质。是插入排序和归并排序的结合。

我使用 Python 实现的时候，由于 Python 的基本操作比较高层，加之算法过于复杂，导致其性能较低。若采用 C 语言实现，实际性能应该会快上不少。

3 实验设计思路

对于每个算法，分别测试其在数据规模为$10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 2*10^8$时所用的时间。

对于插入排序和希尔排序，部分数据规模没有测试。

4 结果分析

实验结果见下表： 根据数据大小可见，数据规模每扩大10倍，insertsort用时变为

数据规模	insertsort/s	shellsort/s	timsort/s	quicksort/s	mergesort/s	radixsort/s
10	1.05E-05	1.26E-05	2.65E-05	1.14E-05	3.10E-05	0.000262
100	0.00046	0.000165	0.000303	0.000129	0.000351	0.000699
1_000	0.050333	0.003494	0.004866	0.001925	0.0049	0.005486
10_000	4.104266	0.065777	0.067027	0.026444	0.061472	0.053807
100_000	443.1358	0.860708	0.641207	0.266237	0.707138	0.515868
1_000_000	\	15.08475	8.059071	3.262861	7.454342	5.326147
10_000_000	\	254.5274	105.1256	42.23226	95.80675	57.50836
100_000_000	\	4301.217	1254.371	569.3573	1161.695	621.4876
200_000_000	\	\	2909.95	1259.87	2613.792	1367.09

原来的100倍左右。shellsort变为原来的15到20倍左右，其他算法变为原来的10倍左右。符合对应算法的复杂度

对数据进行可视化分析如下：

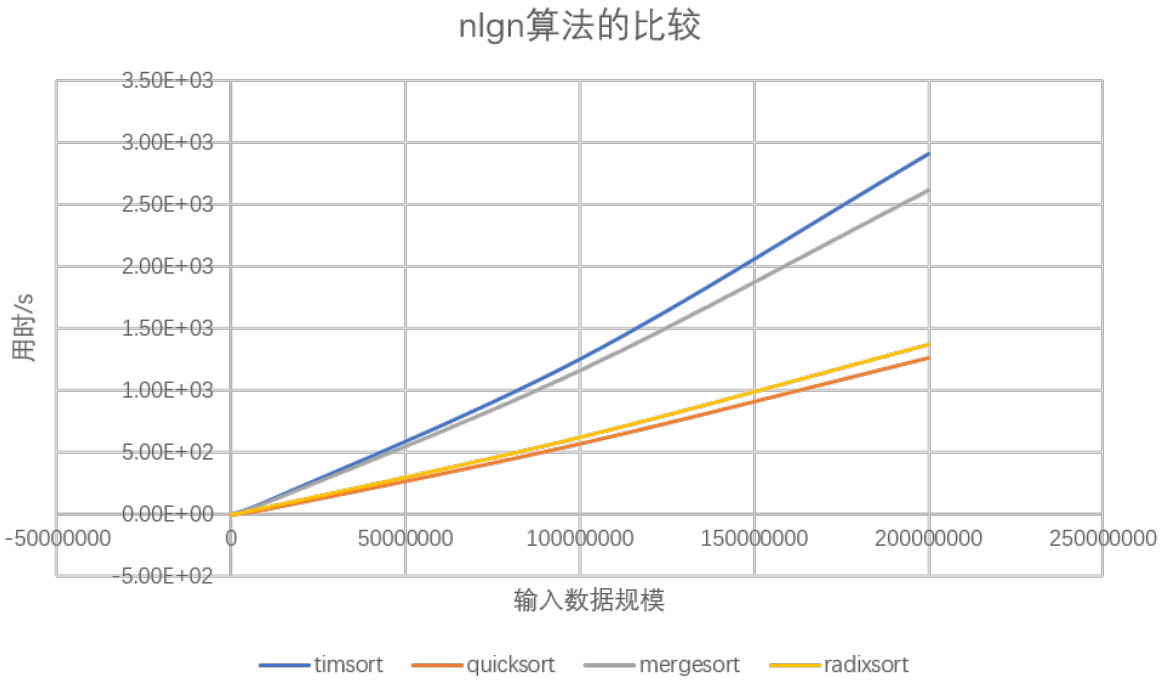

图 1: nlgn算法性能比较

nlgn 的算法中，quicksort 表现最好；radixsort 只比 quicksort 稍微差一点。mergesort 和 timsort 要差上一些。timsort 速度主要受我实现的影响，用Python实现的时候使用Python自带的一些操作没有进行优化，而算法过于复杂导致其实际运行效果不够好。

shellsort 和 insertsort 与 quicksort 的比较可以看出 insertsort 在数据规模增大到一定程度后性能迅速降低。

shellsort相比之下比insertsort表现更好，但与nlgn的算法相比还是差上一些

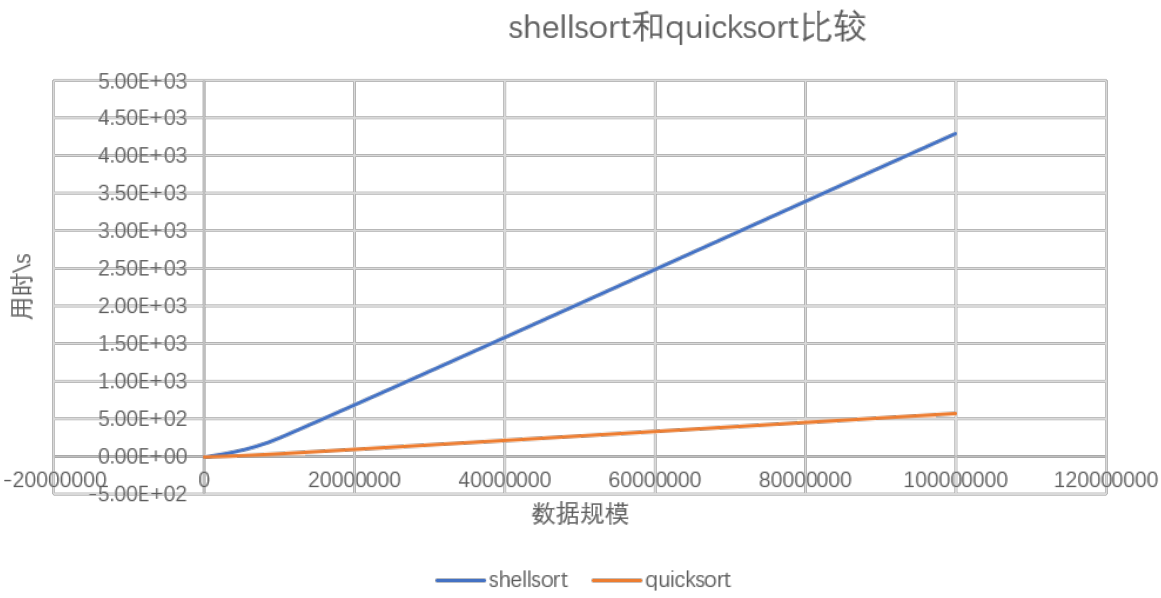

图 2: shellsort 和 quicksort比较

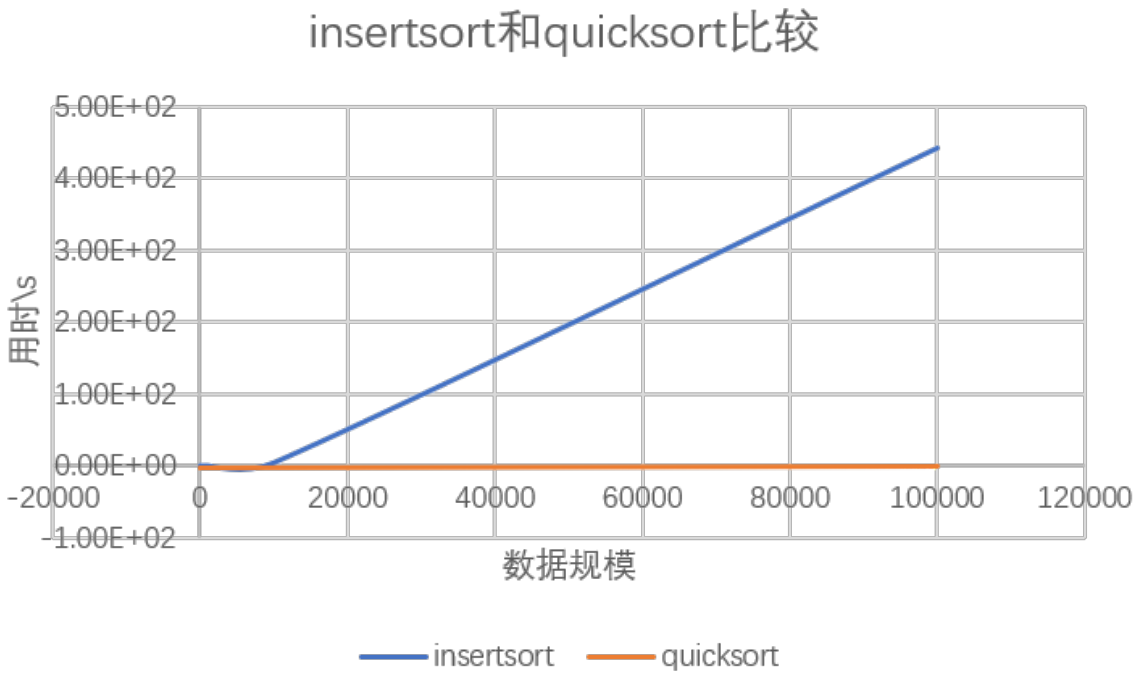

图 3: insertsort 和 quicksort比较