**2024秋-《算法设计与分析》**

**实验一：分治法 实验报告**

|  |
| --- |
| 2024年10月26日 |

**目 录**

[一、 实验要求 1](#_Toc180851151)

[二、 实验报告 1](#_Toc180851152)

[1.1 实验设计 1](#_Toc180851153)

[1.1.1实验内容1 1](#_Toc180851154)

[1.1.2实验内容2，3，4 1](#_Toc180851155)

[1.1.3实验内容5 2](#_Toc180851156)

[1.1.4实验内容6 2](#_Toc180851157)

[1.1.5附加内容 2](#_Toc180851158)

[1.2实验结果分析 3](#_Toc180851159)

[1.2.1实验内容1结果分析 3](#_Toc180851160)

[1.2.2实验内容2,3,4结果分析 3](#_Toc180851161)

[1.2.3实验内容5结果分析 3](#_Toc180851162)

[1.2.4实验内容6结果分析 3](#_Toc180851163)

[1.2.5附加内容结果分析 4](#_Toc180851164)

# 实验要求

一、实验内容

对几种经典的排序算法进行分析，理解算法在不同输入时的表现，深入剖析算法优缺点及其根源。具体要求如下：

实现常见排序算法，至少要实现插入排序、冒泡排序、快速排序、归并排序、shell排序算法；

在排序算法中插桩，记录关键操作次数（如比较次数、移动次数等）；

以待排序文件的行数n为输入规模，固定n随机产生多组测试样本，统计算法的平均运行时间和关键操作次数，改变n的规模重复多次实验，并对结果进行统计；

改变数组规模，对不同规模问题下各算法的结果进行统计并绘制图表，与理论值进行对照分析；

优化快速排序的中枢点选取，对优化前后的性能进行分析；

对快速排序的三种实现进行性能比较。

附加：

实现BlockQuickSort，就分支预测次数展开分析；

实现DualPivotQuickSort，就递归深度展开分析；

在超大规模数据上（如1亿个整数），对比以上快排实现的性能。

# 实验报告

## 实验设计

### 1.1.1实验内容1

源代码见sort\_algorithms\_show.cpp，所有要求算法已经实现。测试设置了五个数组，包括完全重复的数组，空数组，正常数组，倒序数组，随机数组。编译执行即可

### 1.1.2实验内容2，3，4

源代码见algorithms\_main.cpp。先编译algorithms\_main.cpp，然后会生成一个exe，执行得到sorting\_results.txt。执行draw\_sorting\_results.py，得到相应的Figure\_X.png，X代表一轮测试中样本大小的最大值。algorithms\_main.cpp包含了对几个排序算法的插桩和对比，生成的txt文件记录了每一种排序算法在每一个输入规模下的运行时间，比较次数，移动次数。num\_trials变量代表了每一轮中样例的数量，最后会运行num\_trials次循环，然后取均值。sizes存放了size的一系列大小，作为数据规模的输入。每一个size都对应同样的num\_trials。draw\_sorting\_results.py是用来根据sorting\_results.txt的内容来画统计图的。每个统计图都是三个子图，从左到右分别是运行时间，比较次数，移动次数的统计记录。Figure\_1000，Figure\_10000, Figure\_50000对应着样本为1000,10000,50000时的情况。如果想更改输入规模，请改写cpp中的sizes数组

### 1.1.3实验内容5

源代码见better\_quick\_sort.cpp首先编译better\_quick\_sort.cpp，然后会生成一个exe，执行得到Better\_Quick\_Sorting.txt。执行draw\_better\_quick\_quick\_sorting\_results.py，得到相应的Figure\_1.png。better\_quick\_sort.cpp包含了两种排序算法（经典快排，优化基元选择的快排），生成的txt文件记录了两种排序算法在每一个输入规模下的运行时间，比较次数，移动次数。num\_trials变量代表了每一轮中样例的数量，最后会运行num\_trials次循环，然后取均值。sizes存放了size的一系列大小，作为数据规模的输入。每一个size都对应同样的num\_trials。优化基元的方法采取了三数取中法。它选择数组的第一个元素、中间元素和最后一个元素中的中位数作为枢轴，这样可以减少最坏情况发生的可能性。draw\_better\_quick\_quick\_sorting\_results.py是用来根据Better\_Quick\_Sorting.txt的内容来画统计图的。每个统计图都是三个子图，从左到右分别是运行时间，比较次数，移动次数的统计记录。

### 1.1.4实验内容6

源代码见ThreeWaysToQuickSort.cpp。执行顺序：

首先编译ThreeWaysToQuickSort.cpp，然后会生成一个exe，执行得到three\_ways\_sorting\_result.txt。执行draw\_threeways\_quick\_sorting\_results.py，得到相应的Figure\_1.png。

文件说明：

ThreeWaysToQuickSort.cpp包含了三种排序算法（经典快排，随机选择基元的快排，三路快排），生成的txt文件记录了三种排序算法在每一个输入规模下的运行时间，比较次数，移动次数。num\_trials变量代表了每一轮中样例的数量，最后会运行num\_trials次循环，然后取均值。sizes存放了size的一系列大小，作为数据规模的输入。每一个size都对应同样的num\_trials。随机选择基元的快排采取了随机数选择基元。三路快排使用了三路进行递归

draw\_threeways\_quick\_sorting\_results.py是用来根据three\_ways\_sorting\_result.txt的内容来画统计图的。每个统计图都是三个子图，从左到右分别是运行时间，比较次数，移动次数的统计记录。

Figure\_X对应了sizes中的sizes向右位移的次数。例如figure\_11对应于size>>11。这是为了观察重复元素较多时的情况

### 1.1.5附加内容

源代码见BlockQuickSort\_VS\_DualPivotQuickSort.cpp。编译BlockQuickSort\_VS\_DualPivotQuickSort.cpp观察结果即可。觉得太慢可以去主程序里把那1亿和1亿5000万的测试删掉。运行结果看截图

## 1.2实验结果分析

### 1.2.1实验内容1结果分析

排序结果符合预期

### 1.2.2实验内容2,3,4结果分析

观察图片，可以看到冒泡排序的运行时间远大于其他几个排序的运行时间，与其他几个排序的运行时间之差随样本大小的增大而增大。符合N^2级别的时间复杂度。同样的，插入排序也是O(N^2)级别的时间复杂度，随时间复杂度增长速度和其他几个排序不是一个级别，但是明显比冒泡排序快一些。这是因为其比较次数要小于冒泡排序，且移动次数远小于冒泡排序（因为冒泡排序可能会产生气泡向后移动的情况，这种情况移动次数非常多，插入排序不会）

快速排序，归并排序，希尔排序远快于前两者。快速排序和归并排序是一个数量级的，可以看到他们几乎重合。仔细观察Figure\_10000中的的紫色线（Figure\_50000中的坐标轴步长太大不好看，Figure\_1000里由于n^a次方的增长特性，见./nlongn和n的a次方增长对比，n较小时可能比nlogn的排序更快，具体取决于系数），可以看到希尔排序在n=10000时略微比其他两个排序慢，符合n^a和nlogn的增长规律

### 1.2.3实验内容5结果分析

理论分析：

提升不大，都是同级别的，估计系数都差不多。提升的点主要在于快排里某些轮的排序可能基元选的不好，排的次数更多（见最右边对比移动次数的子图，很明显坏轮数的时候移动次数优化快排明显小于原始快排，比较次数都拉不开差距），三数取中优化了基元的选择，减少了排序的次数。但是不是每一轮，基元的选择都不是很合适。对于最差情况，快排是O(N^2)，但是数据毕竟是随机生成的，不可能每一轮都是最坏的，只能说有某些轮是最坏的，或者比较坏的，正是这些轮拉开了二者的差距

结果分析：

和理论一致，随n增大，优化快排比原始快排要快一点点，且差距逐渐增大，这是由于坏轮数随着n的增大也更多，符合随机规律。可以预测在n为10亿以上时可能会拉开1秒以上的差距

### 1.2.4实验内容6结果分析

理论分析：

三路快排肯定最快，三路快排的时间复杂度单次递归对应的是log(3/2)n，数量级更大时，肯定比普通的更快。重复元素更多时，三路快排也应该更快，因为它能分出重复的基元。随机快排可能比经典快排快点，因为它能规避最坏基元情况

结果分析：

可以看到每一张图，三路快排都比其他两者快。在figure0里也是这样，说明三路快排本身就比经典快排快。在其他的figure中可以看到三路快排显著快于正常快排，位移数越大，size的随机范围越小，重复元素越多，三路快排的效果就越快（实际上快的多，主要是那俩太拉）

随机快排比经典稍微慢点，但是从图上能看出来不是数量级的差距。可能是因为调用随机数产生开销（猜想）且基元并不是很坏

请注意，要得到figure10和11，预计要运行300秒，请耐心等

### 1.2.5附加内容结果分析

分区预测次数：

在 BlockQuickSort 中，partitionBlock 函数会对每个待排序的分区进行处理，选择一个枢轴 (pivot) 并将分区内的元素分为较小部分和较大部分。对于每个元素来说，有一个比较操作 arr[j] < pivot，这是一种分支预测行为。现代处理器通过分支预测来提高处理性能，但如果数据不均匀或不符合分支预测的期望，将导致性能下降。特别是对于大数组和重复许多次的元素，这种情况会更为明显。因此，BlockQuickSort 在元素不均匀分布时可能遭遇更多的错误预测。

块优化：

BlockQuickSort 通过在超过某个 blockSize 时递归调用自身，而在小于这个尺寸时使用插入排序，来优化小数组的排序性能，在极端的情况下（例如选了坏基元）肯定会更好一点，但是由于数据随机，所以效果有限

递归深度分析：

这种算法中，二元划分导致每次生成三个子数组，这会减少划分的平均长度（与经典快速排序中生成两个子数组相比）。理论上，DualPivotQuickSort 的递归深度趋近于 log3/2(n)，比经典的 log2(n) 要浅，因为它的划分更多而且更快收敛。这里要注意枢轴选取。如果枢轴选的不好，会拖后腿，但是由于它本身是三路快排，所以也不会太差。如果枢轴选的好（只要最差基元的情况不是很多），应该也能拉开数量级的差距

Classic QuickSort 应该会受到 pivot 选择不当时性能下降的影响，比如在接近已排序的数组上。

BlockQuickSort 将受益于减少递归调用数和对于小块的优化（尤其是插入排序在小规模数据上表现通常较好）。

DualPivotQuickSort 能够有效地处理多数情况下的分区问题，降低递归深度和提高数据处理速度。