**2024秋-《算法设计与分析》**

**排序算法分析实验报告**

|  |
| --- |
| 2024年11月1日 |

**目 录**

[一、 实验要求 1](#_Toc182099369)

[二、 实验报告 1](#_Toc182099370)

[1.1 实验设计 1](#_Toc182099371)

[1.1.1 实验原理 1](#_Toc182099372)

[1.1.2 实验步骤 4](#_Toc182099373)

[1.2 实验结果 4](#_Toc182099374)

[1.2.1 对几种排序算法的比较与分析 4](#_Toc182099375)

[1.2.2 对快速排序中枢点选取的优化 5](#_Toc182099376)

[1.2.3 不同快速排序实现的性能比较 5](#_Toc182099377)

[1.3 实验总结 6](#_Toc182099378)

[附录 7](#_Toc182099379)

[运行指导 7](#_Toc182099380)

[参考文献 8](#_Toc182099381)

# 实验要求

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：

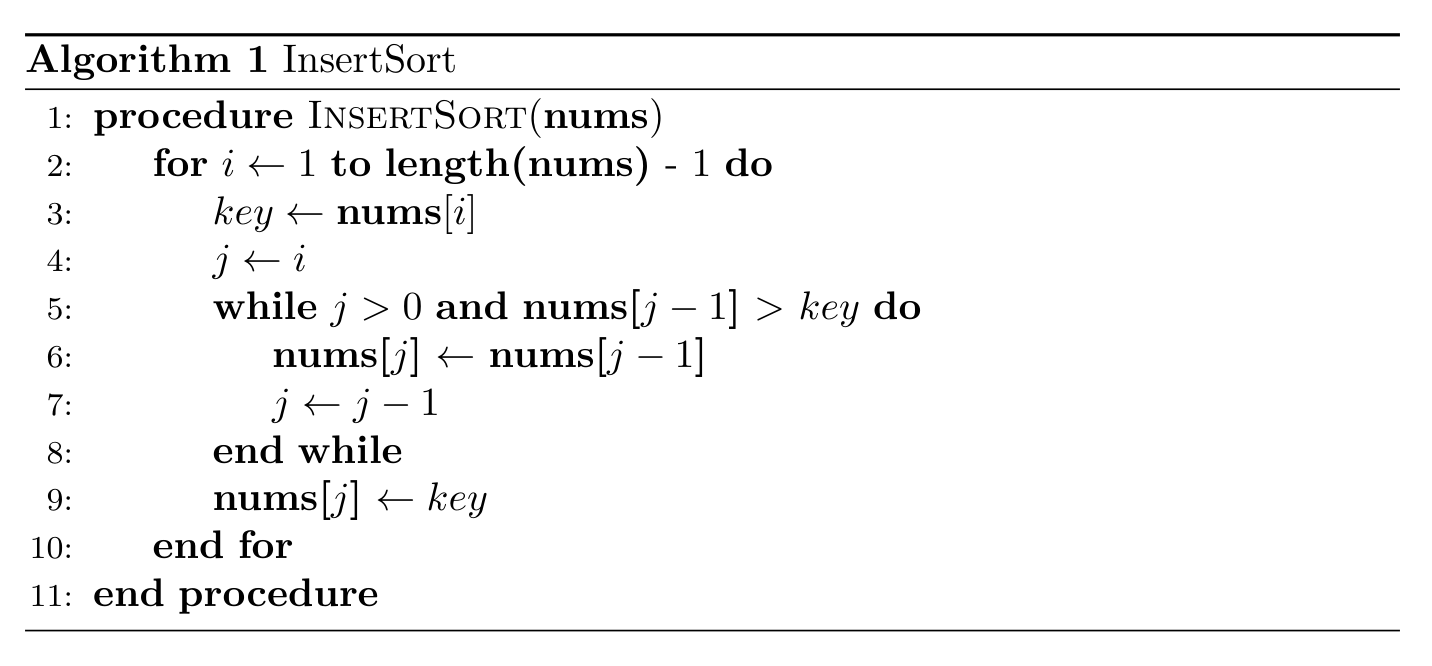
* 实现插入排序、冒泡排序、快速排序、归并排序、Shell排序算法；
* 在排序算法中插桩，记录关键操作次数（如比较次数、移动次数等）；
* 以待排序文件的行数n为输入规模，固定n随机产生多组测试样本，统计算法的平均运行时间和关键操作次数，改变n的规模重复多次实验，并对结果进行统计；
* 改变数组规模，对不同规模问题下各算法的结果进行统计并绘制图表，与理论值进行对照分析；
* 优化快速排序的中枢点选取，对优化前后的性能进行分析；
* 对快速排序的三种实现进行性能比较；
* 实现BlockQuickSort，就分支预测次数展开分析；
* 实现DualPivotQuickSort，就递归深度展开分析；
* 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。

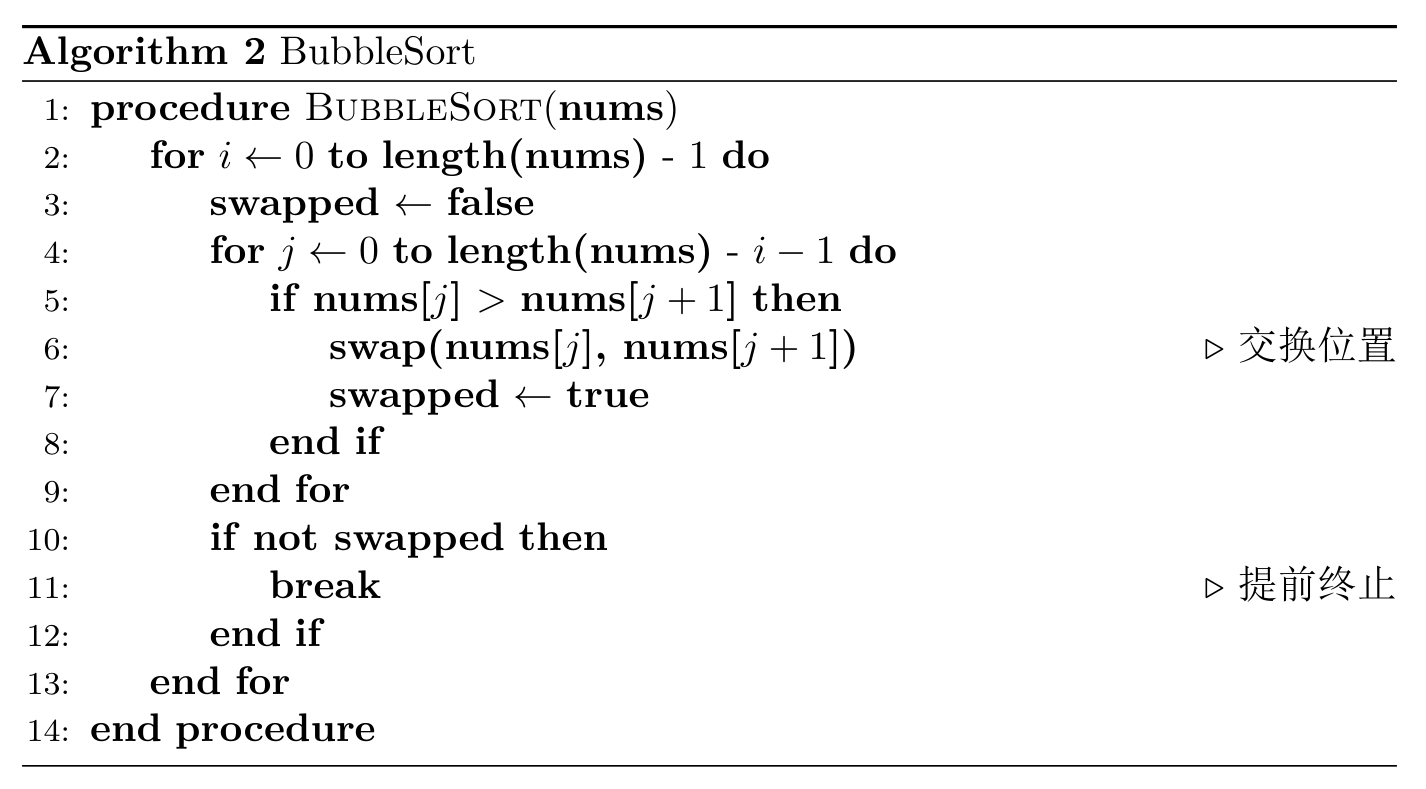
# 实验报告

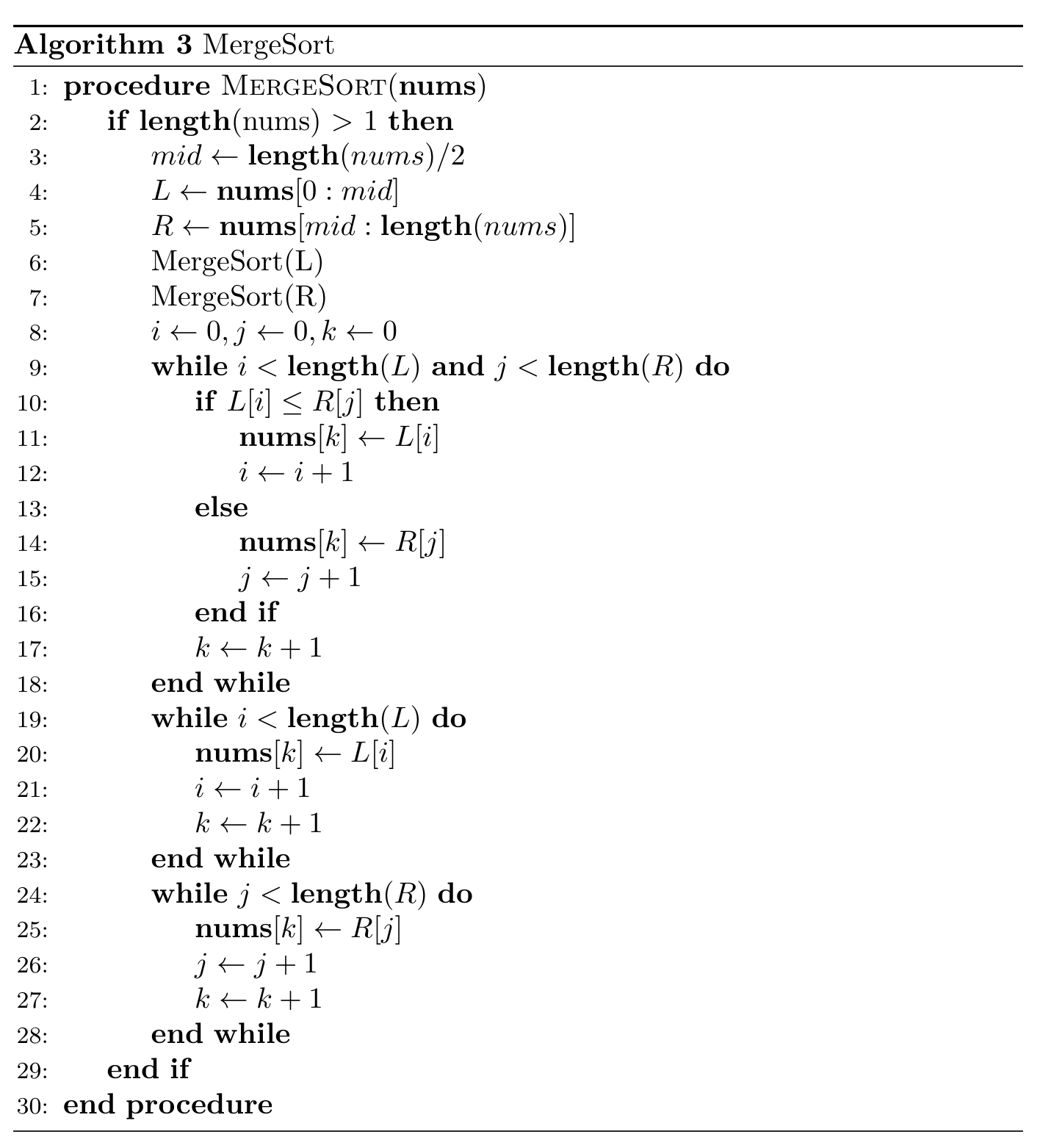
## 实验设计

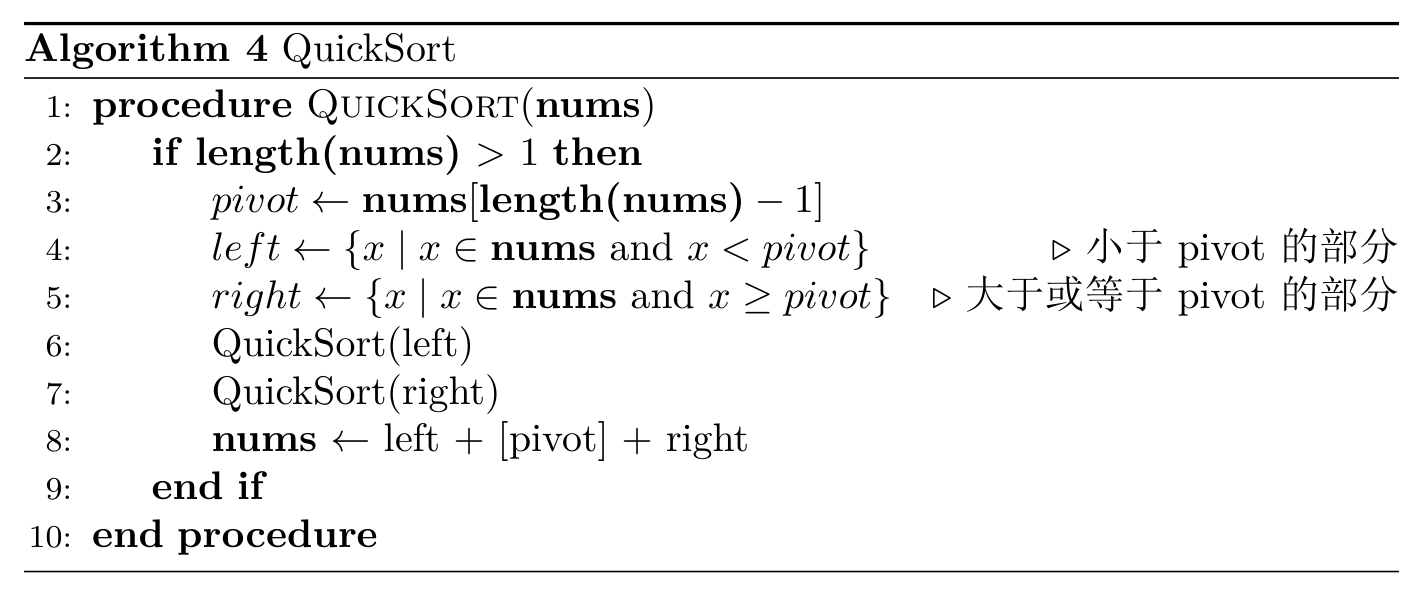
### 实验原理

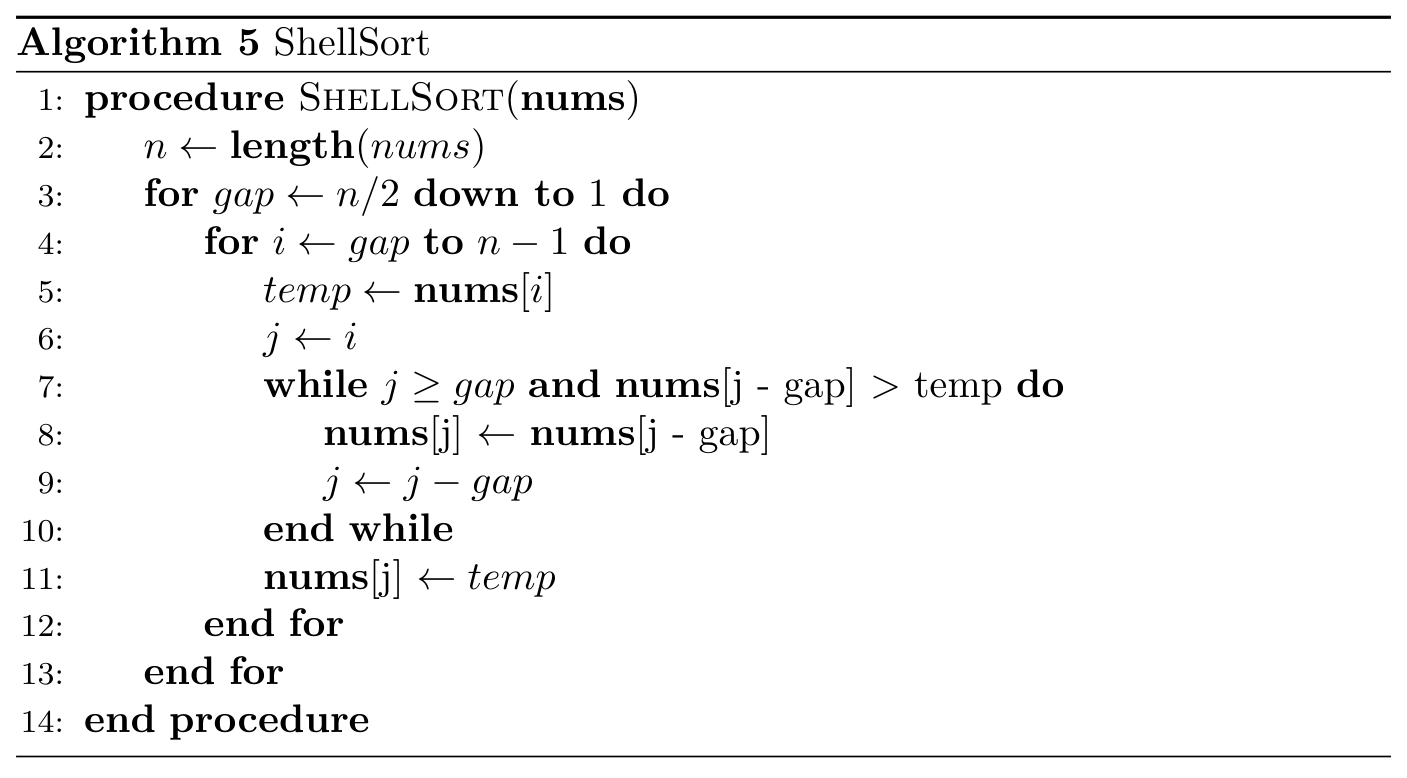
以下是各种排序算法的原理：

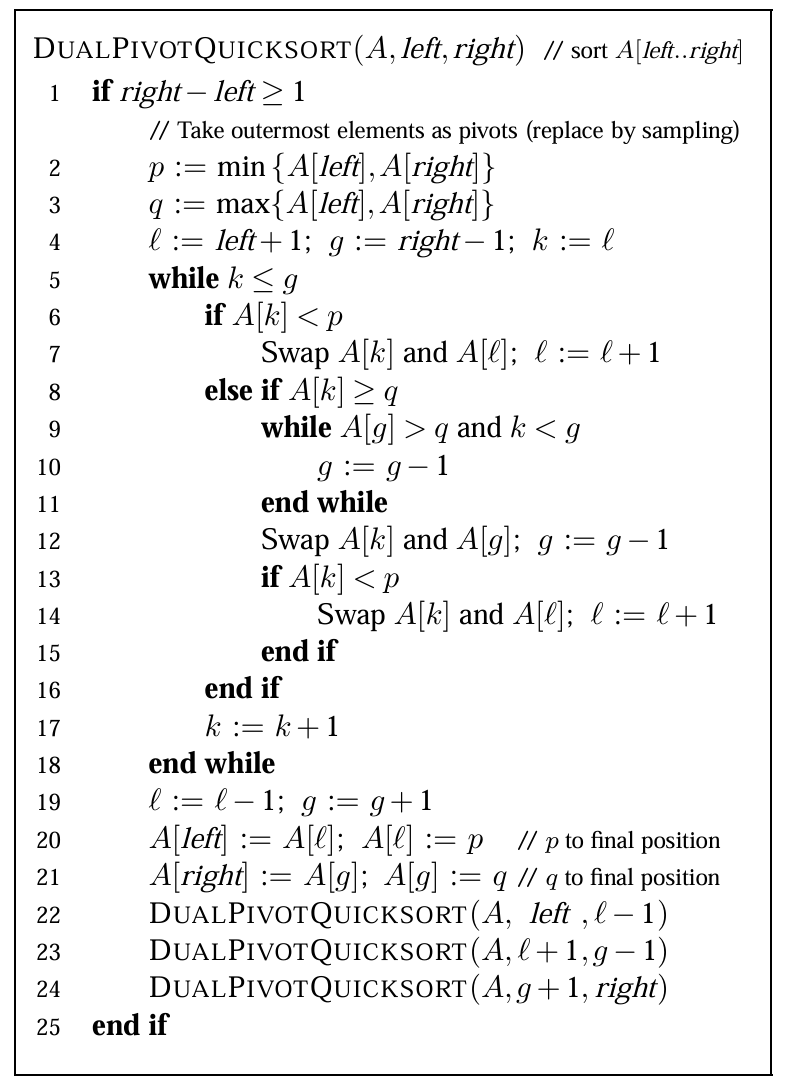
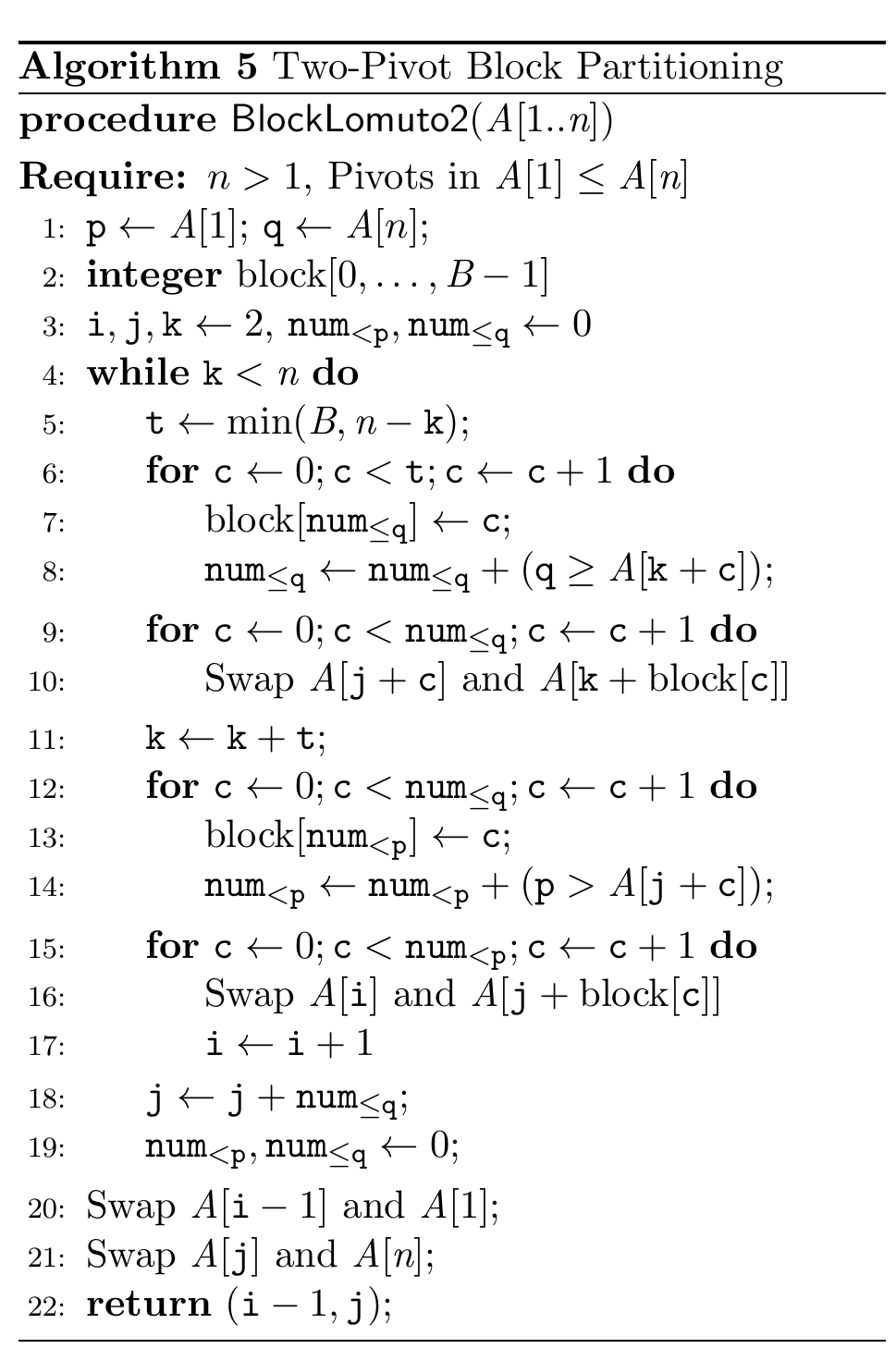












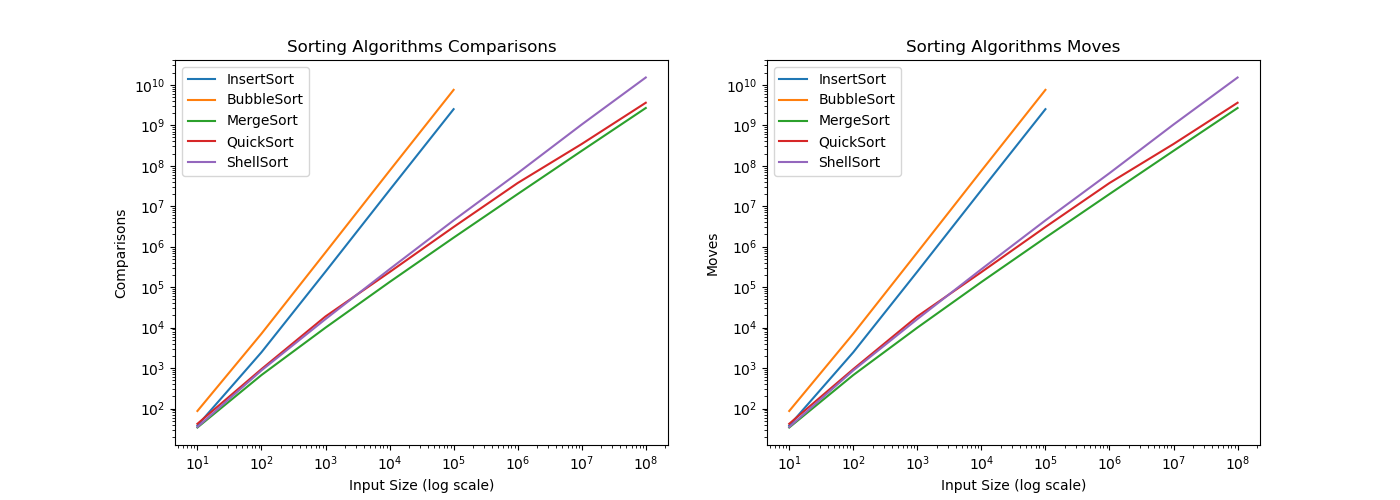
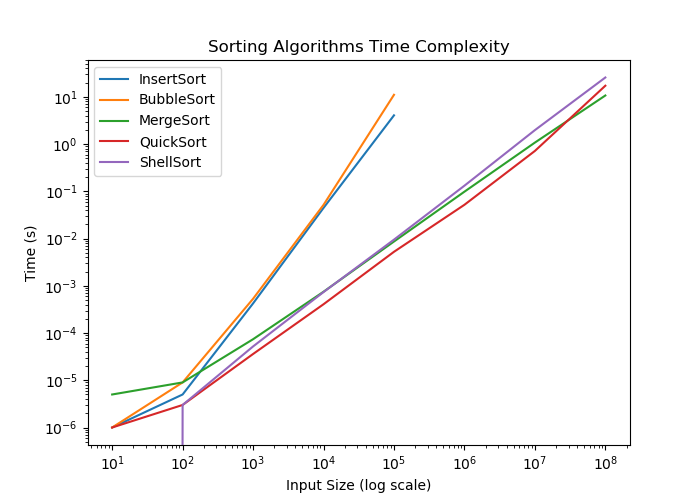
### 实验步骤

* 编写代码，实现各种排序算法。
* 编写测试脚本进行测试。
* 分析结果，撰写实验报告。

## 实验结果

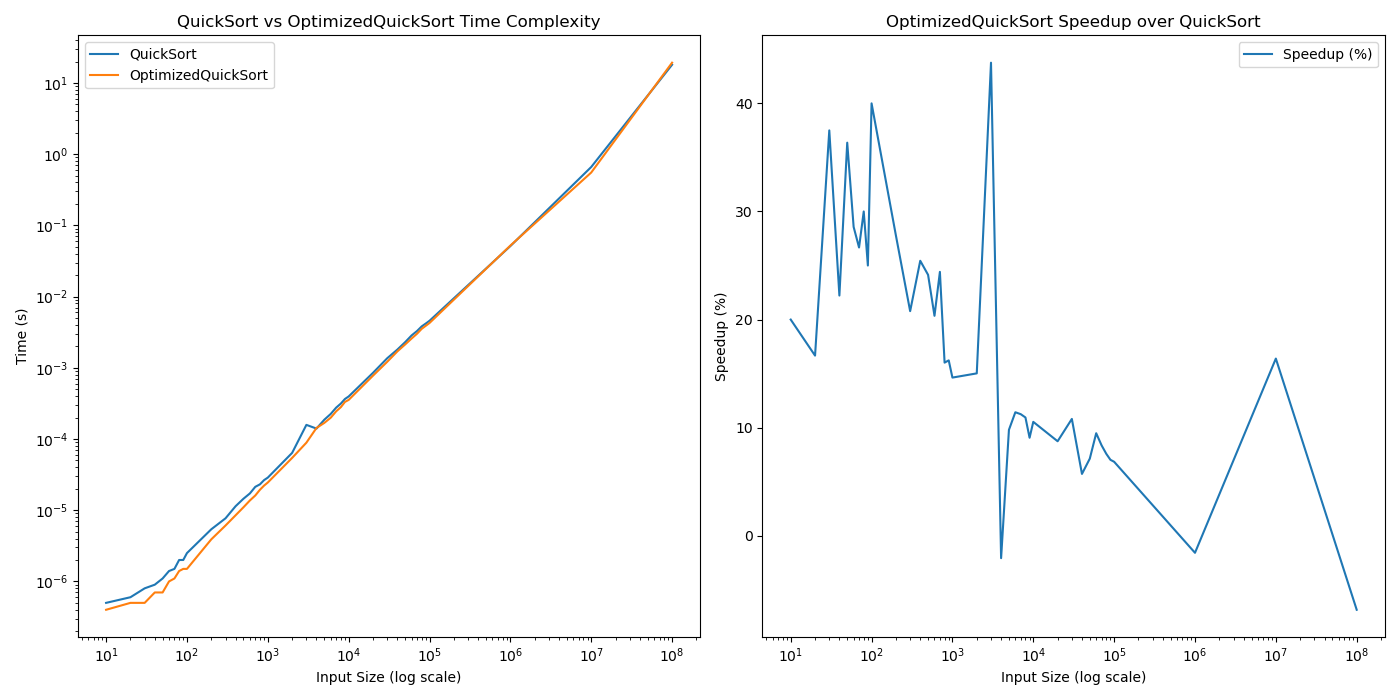
### 对几种排序算法的比较与分析

从n=10开始，逐渐增大规模至108量级，在每个规模下，每种算法重复运行10次，统计算法运行的平均时间，得到以下折线图：



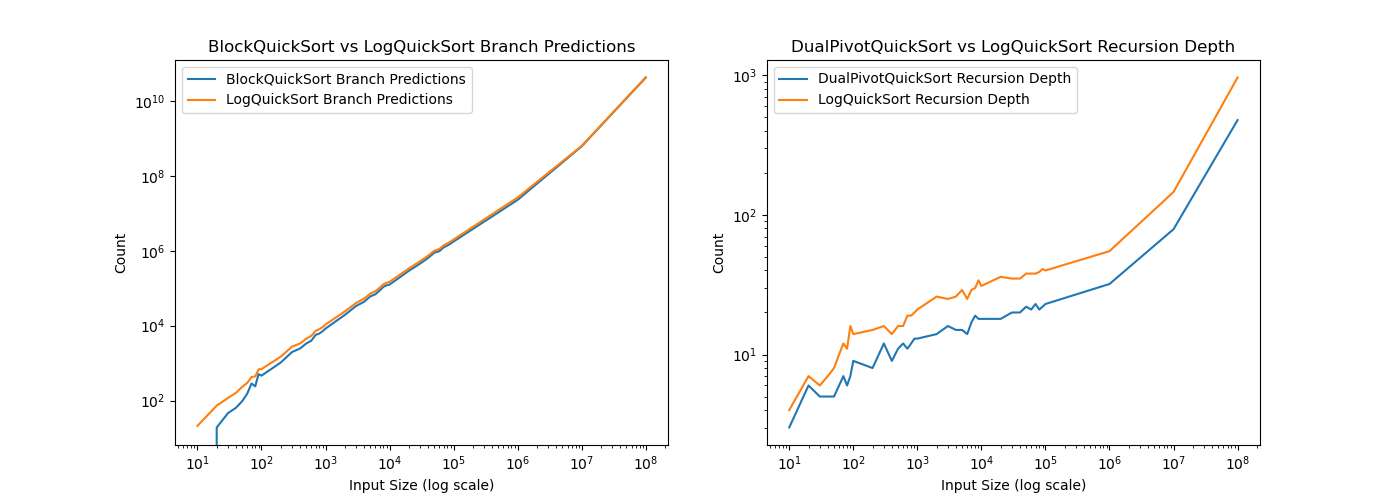
分析折线图，可以发现插入排序、冒泡排序对应折线斜率为2，表明它们的时间复杂度为。归并排序是复杂度为的算法。从图中可以看出，当规模较大（大于103时）归并排序的斜率非常稳定地保持在1。此外，由于待排序数组是均匀随机的，所以快速排序的平均时间复杂度也为。从图中可以看出，快速排序与归并排序类似，斜率较为稳定地保持为1。最后，希尔排序的平均时间复杂度为左右，而从图中可以看出希尔排序的曲线与归并排序相交，且在交点两边保持稳定的斜率差，这与理论是相符的。

### 对快速排序中枢点选取的优化

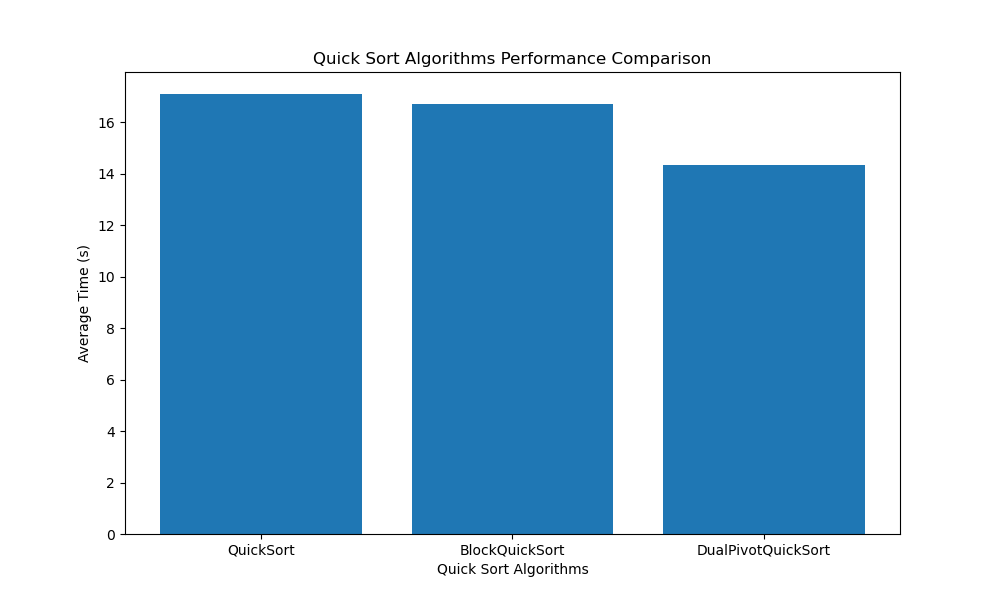
采用三数取中法选取中枢点。下面是针对快速排序优化中枢点选取的实验结果：  


其中，左图是运行时间比较，右图是加速比。据图分析可知，首先，经过优化之后的算法在各种规模测试下性能均有所提升；其次，可以看到在规模很小时，加速比较大，而当规模逐渐增大后，优化效果变差。这是由于当规模增大后，数组的随机性使得待排序数据分布越来越平均，出现快速排序最坏情况的概率越来越小。

### 不同快速排序实现的性能比较

下面是对团快排(BlockQuickSort)、双枢三分快排(DualPivotQuickSort)的性能分析。  


图中，LogQuickSort即普通快排算法（Log即“记录”，是快速排序的插桩版本）。团快排引入了分支消除技术，可以看到左图中团快排的分支预测次数比快排少，有利于性能的提升；而双枢三分快排引入两个中枢点将序列分成三段，因此三分递归的深度只有二分的63%左右，表现为右图（采用对数坐标轴）中两条曲线稳定的y轴差距上。

下面是在一亿规模下运行三种快速排序的时间比较：  


从图中可以看出，两种不同的快排实现均在性能上实现了一定的优化。实践经验表明，团快排的性能优化需要在较为苛刻的条件下才能见效，因为其中使用的分支消除技术的具体实现和CPU指令集、编译器都有紧密关系。笔者将相同的程序放到AMD平台上运行，团快排中的优化变成了负优化，这与Intel、AMD CPU的微体系结构不同有较大关系。此外，双枢三分快排通过减少递归层数来增大Cache利用度，提升Cache命中率以减少访存开销，最终实现性能优化。从图中可以看到，在Intel i7平台（实验所用平台）下，双枢三分快排的性能是最高的。此平台的CPU拥有共24MB的三级缓存，如果缓存更小，双枢三分快排与其他两种快排的性能差距可能还会拉大。

## 实验总结

本次实验通过对多种排序算法的实现和分析，深入理解了不同算法在时间复杂度、空间复杂度上的差异，并验证了不同优化方法对算法性能的影响。实验结果表明：

插入排序和冒泡排序在小规模数据上具有较好的性能，但在大规模数据上其时间复杂度较高，限制了适用范围。归并排序和快速排序在大规模数据上表现出更优的性能，特别是归并排序的时间复杂度始终保持在。

通过使用三数取中法选择中枢点，可以有效减少最坏情况的发生概率，优化了排序性能，尤其在小规模数据上提升显著。然而，随着数据规模增大，优化效果趋于稳定，这是由于随机数据分布减少了最坏情况的概率。

实验对比了普通快速排序、团块快速排序（BlockQuickSort）和双枢快速排序（DualPivotQuickSort）。结果显示，双枢快速排序在Intel平台上具有更高的性能，尤其在大规模数据处理上显著优于其他实现。值得注意的是，性能优化与硬件平台和CPU结构紧密相关，团块快速排序在不同平台上表现出差异。

通过本次实验，我不仅掌握了经典排序算法的实现和优化方法，也更深入理解了算法性能与硬件结构的关系。这为后续学习过程中针对排序算法的选择和优化提供了实践依据。

# 附录

## 运行指导

***需要在 Linux 环境下运行。***

cd **/path/to/script.py**

make

python script.py

运行make后，输出应当为

>make

g++ -O3 -o main.o main.cpp Test.cpp algorithm.cpp

并且，工作目录下会多出一个mian.o文件。

运行python script.py后，输出将会类似于：

>python script.py

...

Running InsertSort with size 10...

InsertSort with size 10 completed.

Running BubbleSort with size 10...

BubbleSort with size 10 completed.

Running MergeSort with size 10...

MergeSort with size 10 completed.

Running QuickSort with size 10...

QuickSort with size 10 completed.

Running ShellSort with size 10...

ShellSort with size 10 completed.

Running BlockQuickSort with size 10...

BlockQuickSort with size 10 completed.

Running DualPivotQuickSort with size 10...

DualPivotQuickSort with size 10 completed.

...

此外，你将在./pictures文件夹中看到程序画出的各种图片。

根据电脑性能差距，Python 脚本的整个运行过程需要 1~2 小时。如果你觉得这太久了，请修改script.py中的第74行，将"--repeat=10"改为"--repeat=1"。这样，运行时间会控制在 6~12 分钟。

此外，你可以单独运行main.o来测试某种算法。下面是main.o的帮助信息：

>./main.o

usage: ./main <FUNCTION> [OPTION]

DESCRIPTION:

FUNCTION:

--function=FUNC

MUST be given. if not, turn to --help

choose function FUNC to sort the vector

FUNC list: InsertSort, BubbleSort, MergeSort, QuickSort, ShellSort, BlockQuickSort, DualPivotQuickSort,

OptimizedQuickSort, LogQuickSort

(LogQuickSort is the same with QuickSort but it will print branch predictions and recursion depth instead of

comparsions and moves)

OPTION:

-h, --help

print this help info and quit

-t, --test

run the function Test() to check the FUNC

this parameter will disable --size and --repeat

--size=N

to determine vector\_size in main

the default value is 10000 if not specially given

--repeat=N

to determine num\_repeat in main

the default value is 100 if not specially given

下面是运行示例：

>./main.o --function=QuickSort --size=100000 --repeat=10

The average time is 0.0052576 s.

Comparisons: 2029938, Moves: 3260251

>./main.o --function=QuickSort -t

The code is correct.

很遗憾的是，本程序不支持自动补全，所以需要准确输入算法的名称，推荐先打印帮助信息，然后复制粘贴（查看上面的帮助信息，在 FUNC list 中的就是所有可使用的排序算法的名称）。

## 参考文献

[1]Martin Aumüller;Nikolaj Hass.Simple and Fast BlockQuicksort using Lomuto's Partitioning Scheme[J].2018,

[2]Sebastian Wild. Why Is Dual-Pivot Quicksort Fast? [J]. 2016.