# 2022秋-算法设计与分析

# 排序算法分析实验报告

|  |  |
| --- | --- |
| 姓名： |  |
| 学号： |  |

|  |
| --- |
| 2022年10月30日 |

**目录**

[2022秋-算法设计与分析 1](#_Toc117972348)

[排序算法分析实验报告 1](#_Toc117972349)

[一、实验要求 2](#_Toc117972350)

[二、实验报告 2](#_Toc117972351)

[2.1 十大排序算法的实现 2](#_Toc117972352)

[2.1.1 插入排序 2](#_Toc117972353)

[2.1.2 冒泡排序 2](#_Toc117972354)

[2.1.3 选择排序 3](#_Toc117972355)

[2.1.4 希尔排序 3](#_Toc117972356)

[2.1.5 归并排序 4](#_Toc117972357)

[2.1.6 快速排序 4](#_Toc117972358)

[2.1.7 堆排序 5](#_Toc117972359)

[2.1.8 计数排序 6](#_Toc117972360)

[2.1.9 桶排序 7](#_Toc117972361)

[2.1.10 基数排序 7](#_Toc117972362)

[2.2 不同排序算法的平均运行时间与关键操作次数比较 7](#_Toc117972363)

[2.3 数组规模对各排序算法的影响 14](#_Toc117972364)

[2.4 快速排序不同实现的性能比较及优化 16](#_Toc117972365)

[三、实验心得体会 18](#_Toc117972366)

# 一、实验要求

对几个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：

1. 以给出的插入排序算法为例，实现几种排序算法，至少要实现冒泡排序、快速排序、归并排序、shell排序算法；

2. 在排序算法中打桩，记录关键操作次数（例如比较次数、移动次数）；

3. 以待排序数组的大小n为输入规模，固定n，随机产生大量测试样本，统计不同排序算法的平均运行时间和关键操作次数，并进行记录；

4. 改变数组规模，对不同规模问题各算法的结果对比分析，通过统计画图，与理论值进行对照分析，并撰写实验报告；

5. 附加：对快速排序的几种不同的实现进行性能比较；对快速排序进行优化，对优化前后的性能进行分析。

# 二、实验报告

## 2.1 十大排序算法的实现

### 2.1.1 插入排序

插入排序已在样例中给出，不再进行实现。

### 2.1.2 冒泡排序

冒泡排序是一种简单直观的排序算法。它重复地遍历数组中未排序的部分，每次都针对相邻的两个元素根据大小进行交换，从而实现大元素往后移动或小元素往前移动的效果。随着遍历次数的增加，数组中已排序的部分逐渐增多，未排序的部分逐渐减少，数组整体趋于有序。

可以通过二重循环实现冒泡排序，第一重循环控制总共遍历的次数，第二重循环控制每次遍历到达的末尾边界，第二重循环需要根据第一重循环变量i控制第二重循环的末尾边界。

代码实现如下图所示：

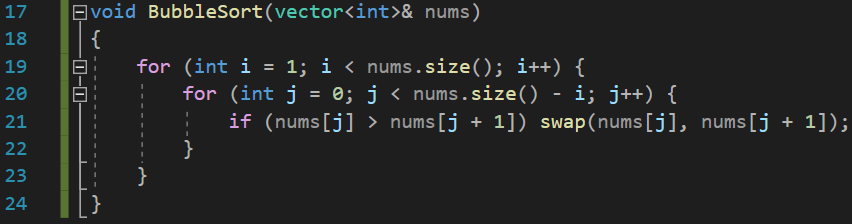


图1：冒泡排序的实现

### 2.1.3 选择排序

选择排序是一种简单直观的排序算法。它重复地遍历数组中未排序的部分，每次遍历都记录下当前未排序部分中最小元素或最大元素的位置，遍历结束后将该最小元素或最大元素与未排序部分最前端或最后端元素交换。随着遍历次数的增加，数组中已排序的部分逐渐增多，未排序的部分逐渐减少，数组整体趋于有序。

可以通过二重循环实现选择排序，第一重循环控制待交换的元素位置，即无序部分最前端或有序部分最后端，第二重循环在无序部分遍历寻找最小元素的位置并记录下来，第二重循环结束后进行交换。

代码实现如下图所示：

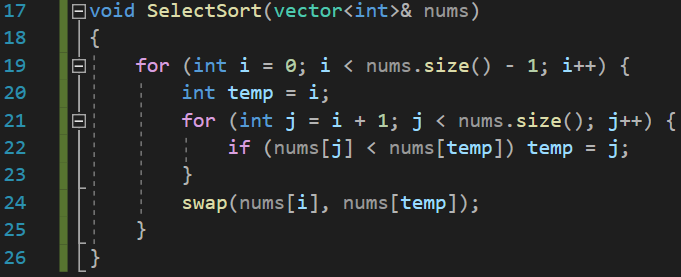


图2：选择排序的实现

### 2.1.4 希尔排序

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。希尔排序是基于插入排序的以下两点性质而提出改进方法的：

1. 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；

2. 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。

代码实现如下图所示：

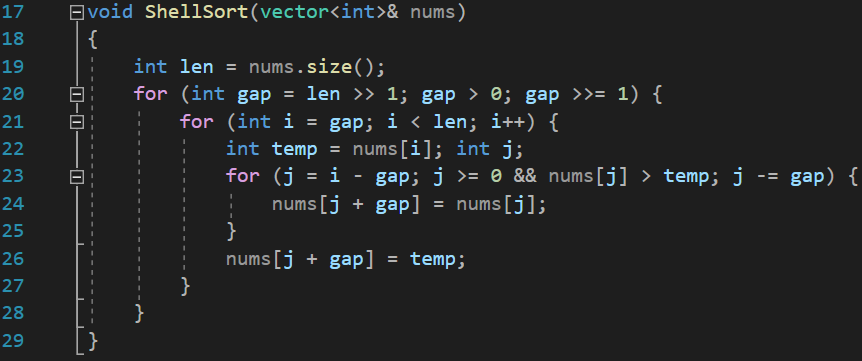


图3：希尔排序的实现

### 2.1.5 归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。作为一种典型的分而治之思想的算法应用，归并排序的实现有两种方法：

1. 自上而下递归；

2. 自下而上迭代。

归并排序需要申请额外的存储空间用来对两个排好序的部分进行合并排序，可以通过双指针的方法，快速地对两个排好序的部分进行归并。每个部分大小从1开始逐渐增加，直到大于等于数组总长度。

代码实现如下图所示：

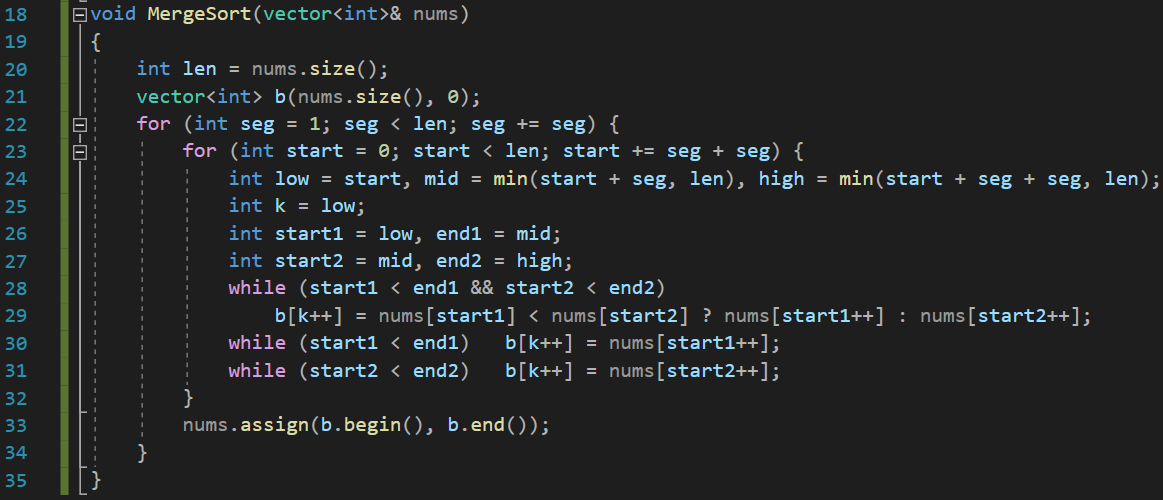


图4：归并排序的实现

### 2.1.6 快速排序

快速排序是一种非常经典的排序算法。正如其名，快速排序可以以较快速度对数组进行排序。该算法进行多次数组遍历并交换元素，每次都选定一个标准值，遍历后，比标准值小的元素在标准值左边，比标准值大的元素在标准值右边，然后分别对左边和右边的子数组分别进行快速排序。

快速排序有多种实现方法，此处先使用基础的“两头凑”方法，采用递归思想进行实现。

代码实现如下图所示：

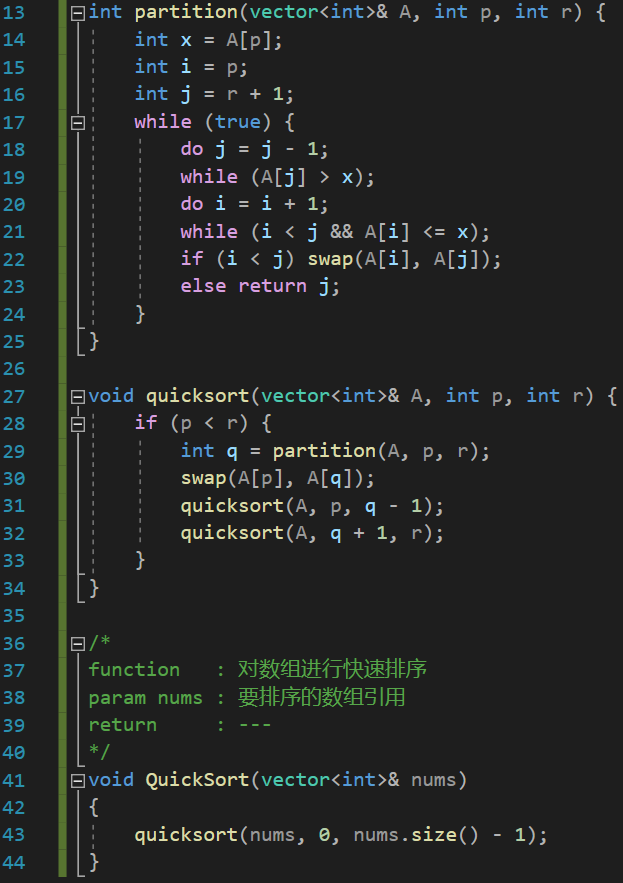


图5：快速排序的实现

### 2.1.7 堆排序

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。堆排序可以说是一种利用堆的概念来排序的选择排序。分为两种方法：

1. 大顶堆：每个节点的值都大于或等于其子节点的值，在堆排序算法中用于升序排列；

2. 小顶堆：每个节点的值都小于或等于其子节点的值，在堆排序算法中用于降序排列。

堆排序实现较为复杂：首先将数组元素堆化，即把最大值或最小值通过交换置于堆顶；然后，将堆顶元素和堆尾元素进行交换，此时堆尾已经稳定，以后的操作中不需要考虑堆尾；再进行堆化和交换，直到实现数组完全有序。

代码实现如下图所示：

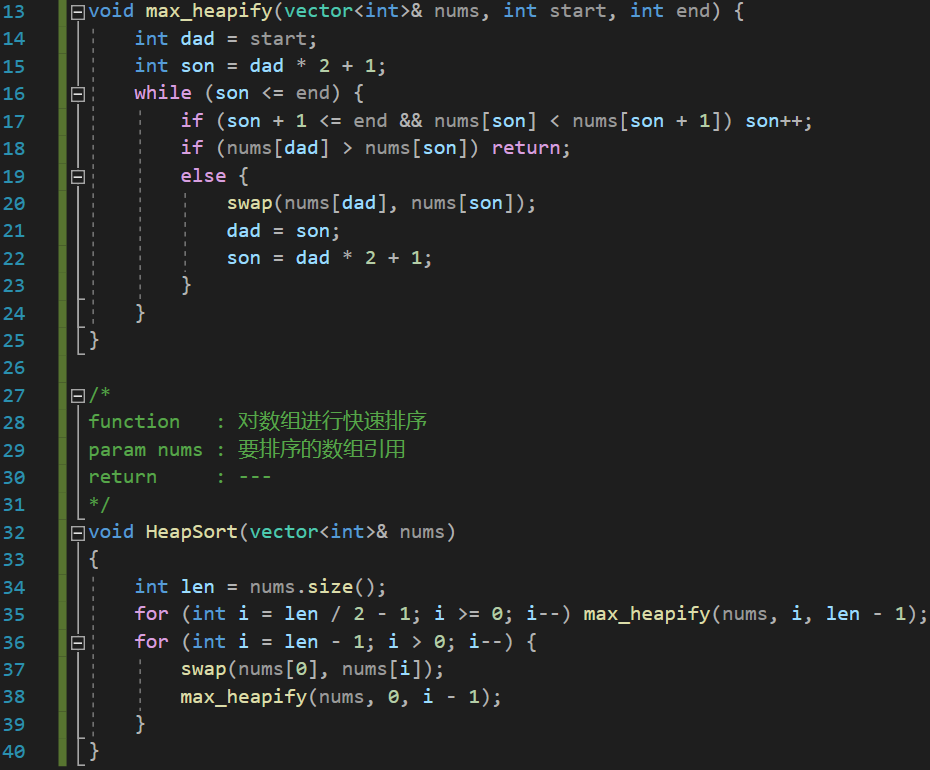


图6：堆排序的实现

### 2.1.8 计数排序

计数排序是一种容易理解的排序算法，它通过统计对每个元素建立索引，求得每个元素出现的次数，然后还原出正确顺序。计数排序时间复杂度可以达到O(n)，但是需要额外的空间来统计每个元素出现的次数，是一种空间换时间的思想。

代码实现如下图所示：

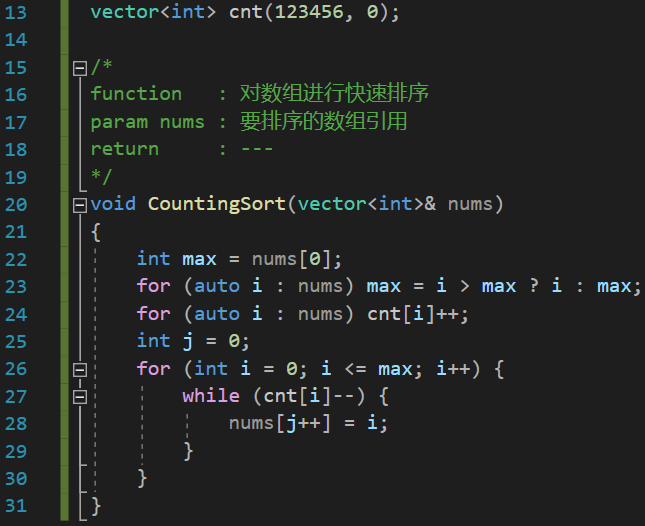


图7：技术排序的实现

### 2.1.9 桶排序

桶排序是计数排序的升级版，它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。为了使桶排序更加高效，需要做到以下两点：

1. 在额外空间充足的情况下，尽量增大桶的数量；

2. 使用的映射函数能够将输入的n个数据均匀地分配到k个桶中。

代码实现略。

### 2.1.10 基数排序

数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。

代码实现略。

## 2.2 不同排序算法的平均运行时间与关键操作次数比较

固定待排序数组的大小为2000，针对每个排序算法，循环执行100次。每次执行都随机生成元素数量为2000的数组，然后调用排序算法进行排序，统计出总共执行时间，从而获得每个排序算法的平均执行时间。

对各个排序算法进行插桩，记录下各排序算法执行过程中的比较次数和移动次数，输出于控制台中并做记录。接下来以插入排序为例，进行插桩操作的示意：

首先，为了记录插入排序算法执行过程中的平均比较次数和移动次数，在main()函数中需要首先声明两个变量sum\_cop和sum\_mov来记录插入排序算法执行100次的总比较次数和移动次数，最后通过除以执行次数100获得平均比较次数和移动次数。期间，为了修改sum\_cop和sum\_mov，需要修改InsertSort()函数，增加两个形式参数，以传引用的方式传入函数内部，在函数内部进行修改：

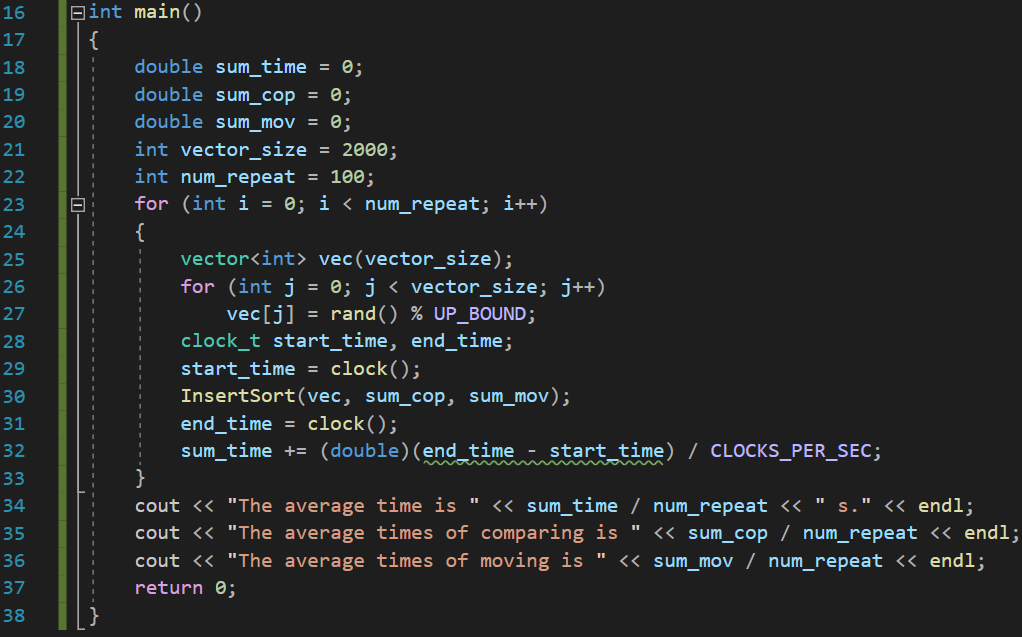


图8：修改后的main()函数

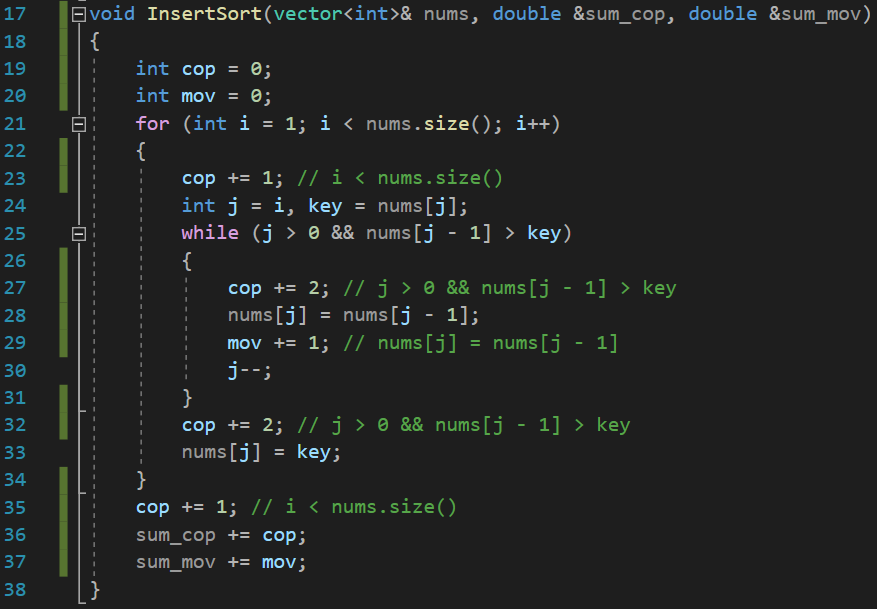


图9：修改后的InsertSort()函数

执行程序，得到控制台输出结果如下图所示：

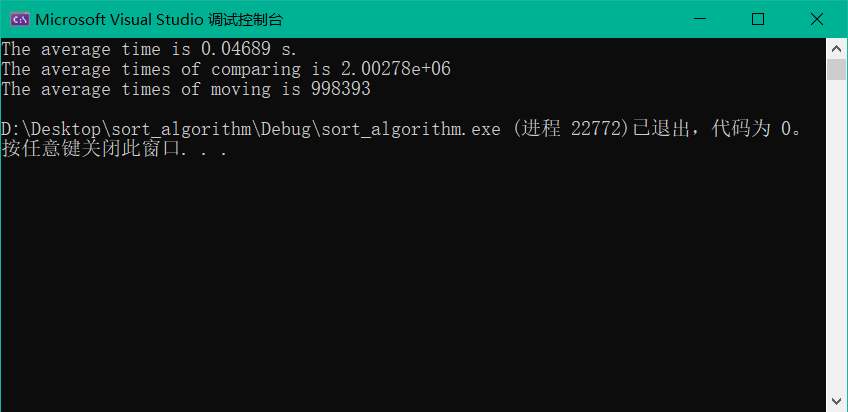


图10：执行修改后的程序

使用上述方法修改冒泡排序如下图所示：

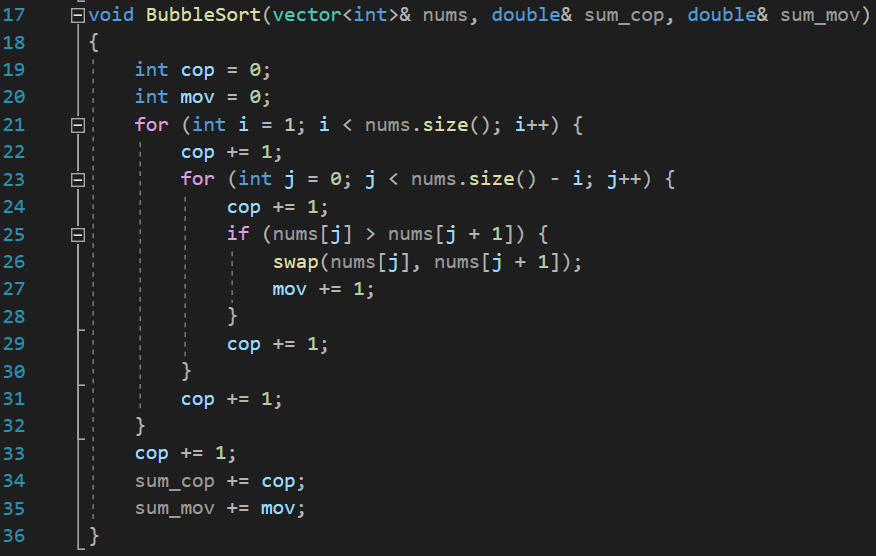


图11：修改后的BubbleSort()函数

使用上述方法修改选择排序如下图所示：

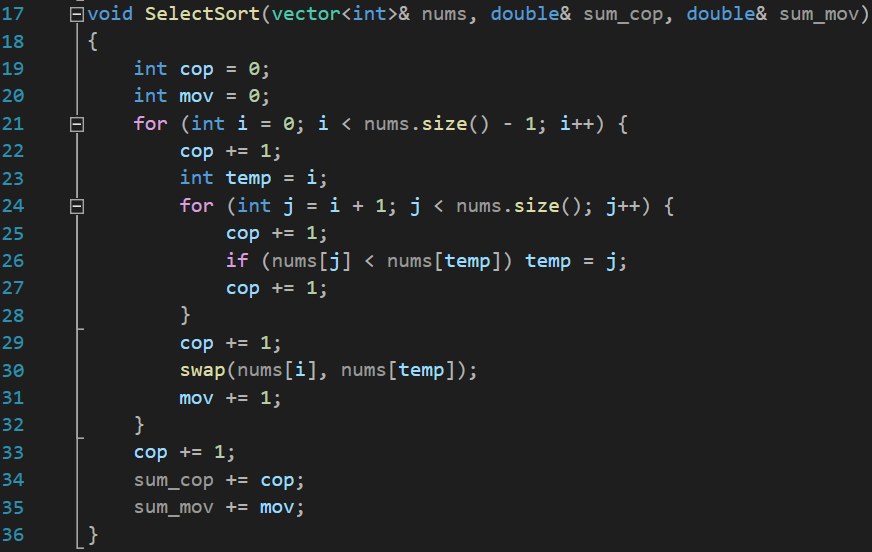


图12：修改后的SelectSort()函数

使用上述方法修改希尔排序如下图所示：

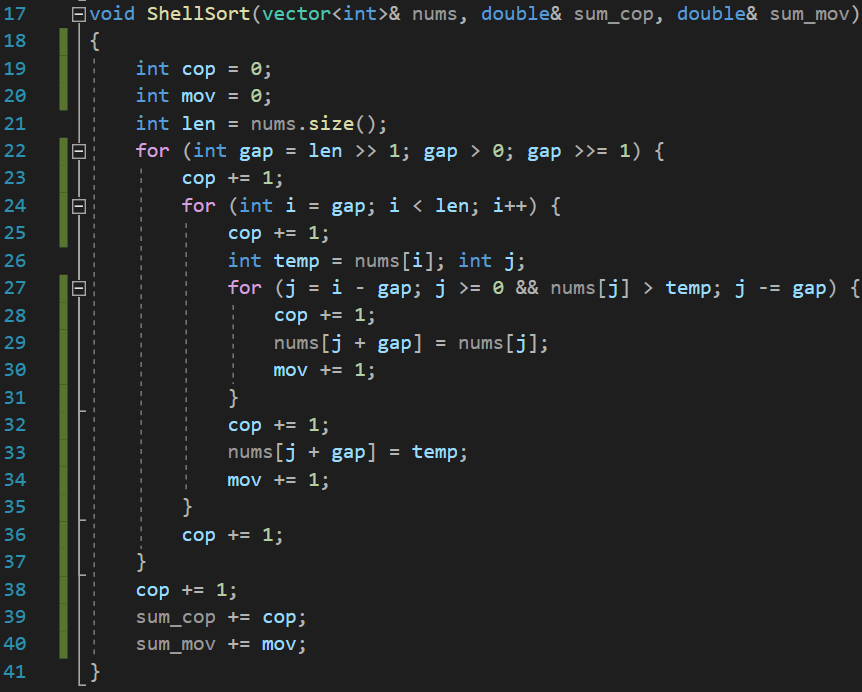


图13：修改后的ShellSort()函数

使用上述方法修改归并排序如下图所示：

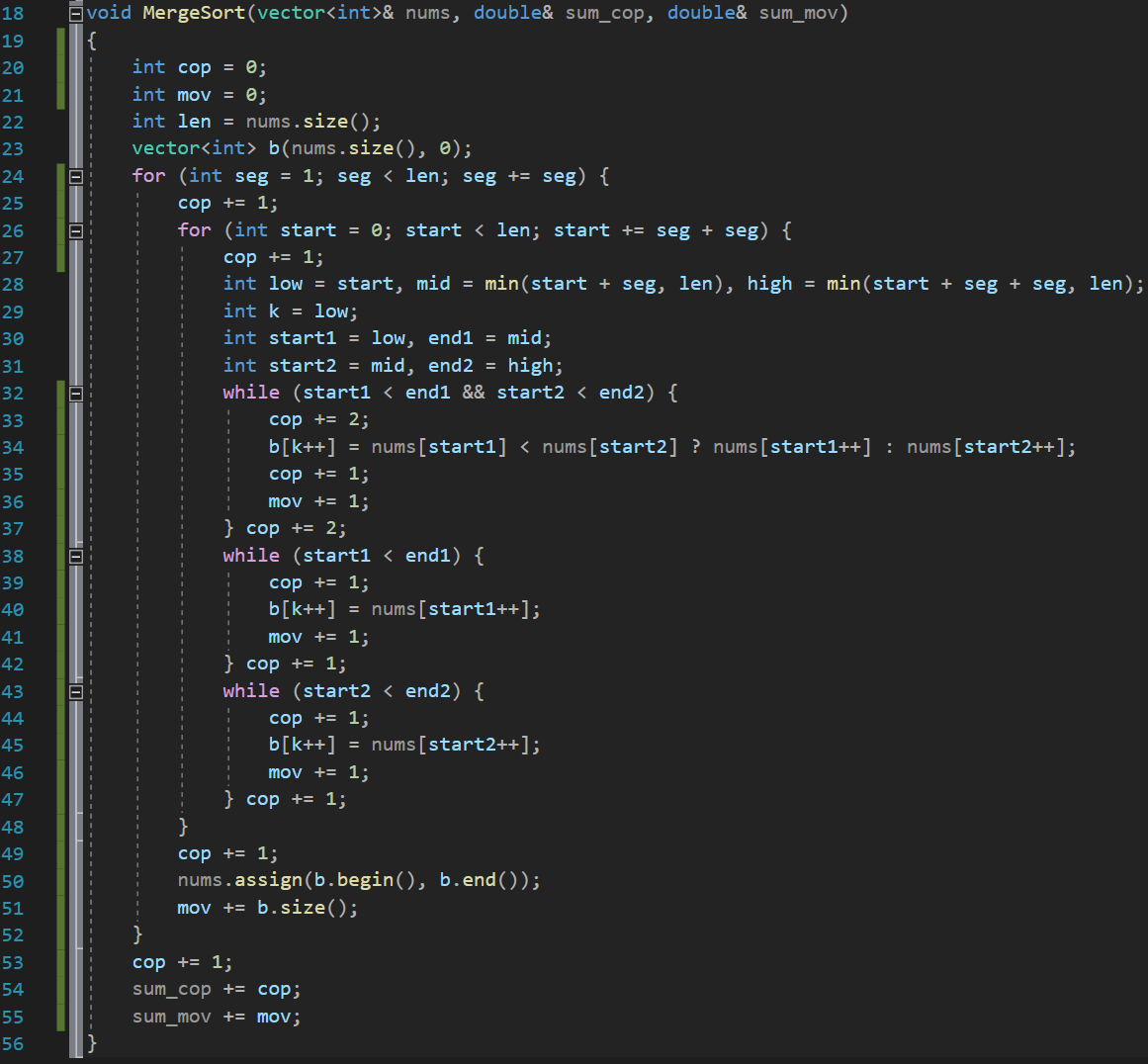


图14：修改后的MergeSort()函数

使用上述方法修改快速排序如下图所示：

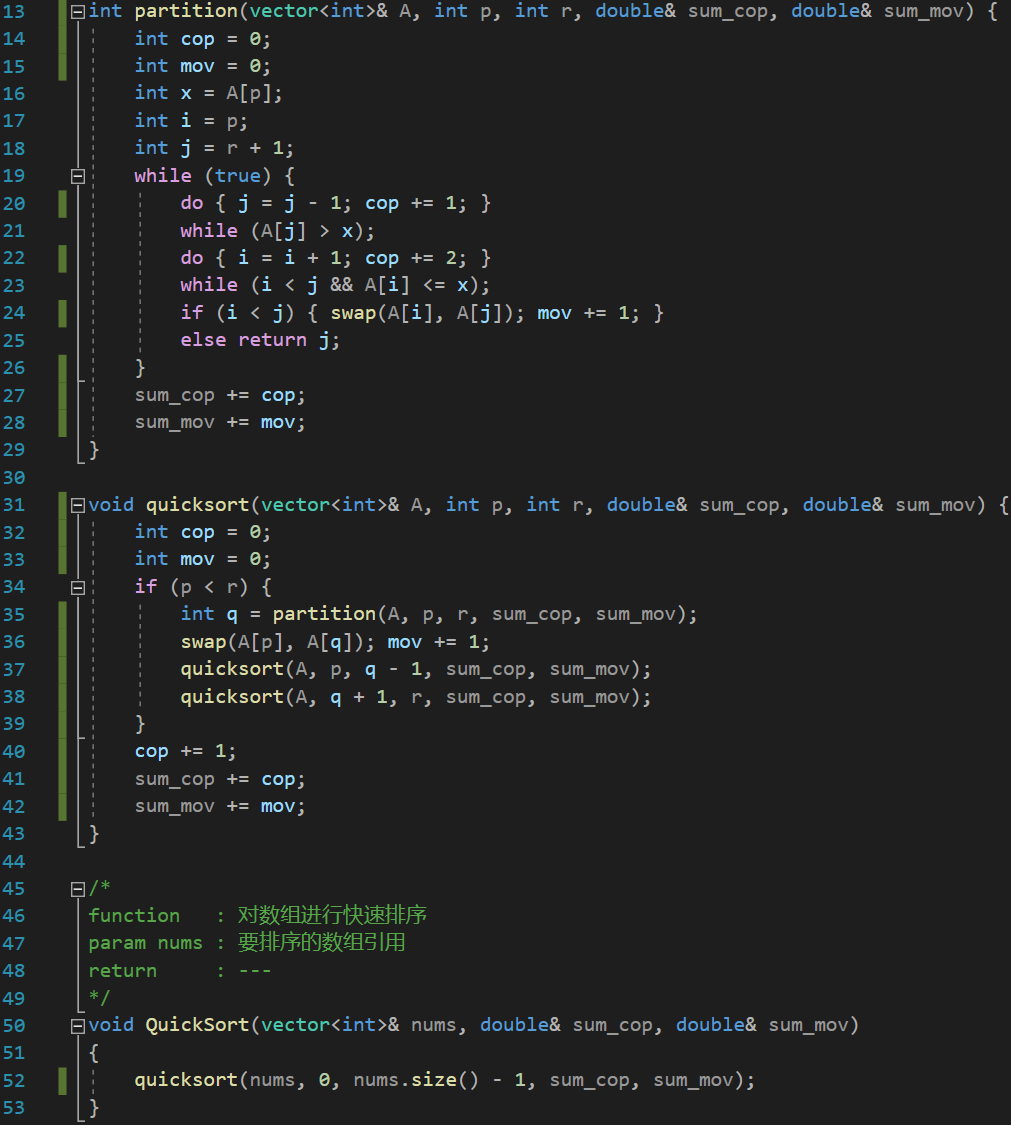


图15：修改后的QuickSort()函数

使用上述方法修改堆排序如下图所示：

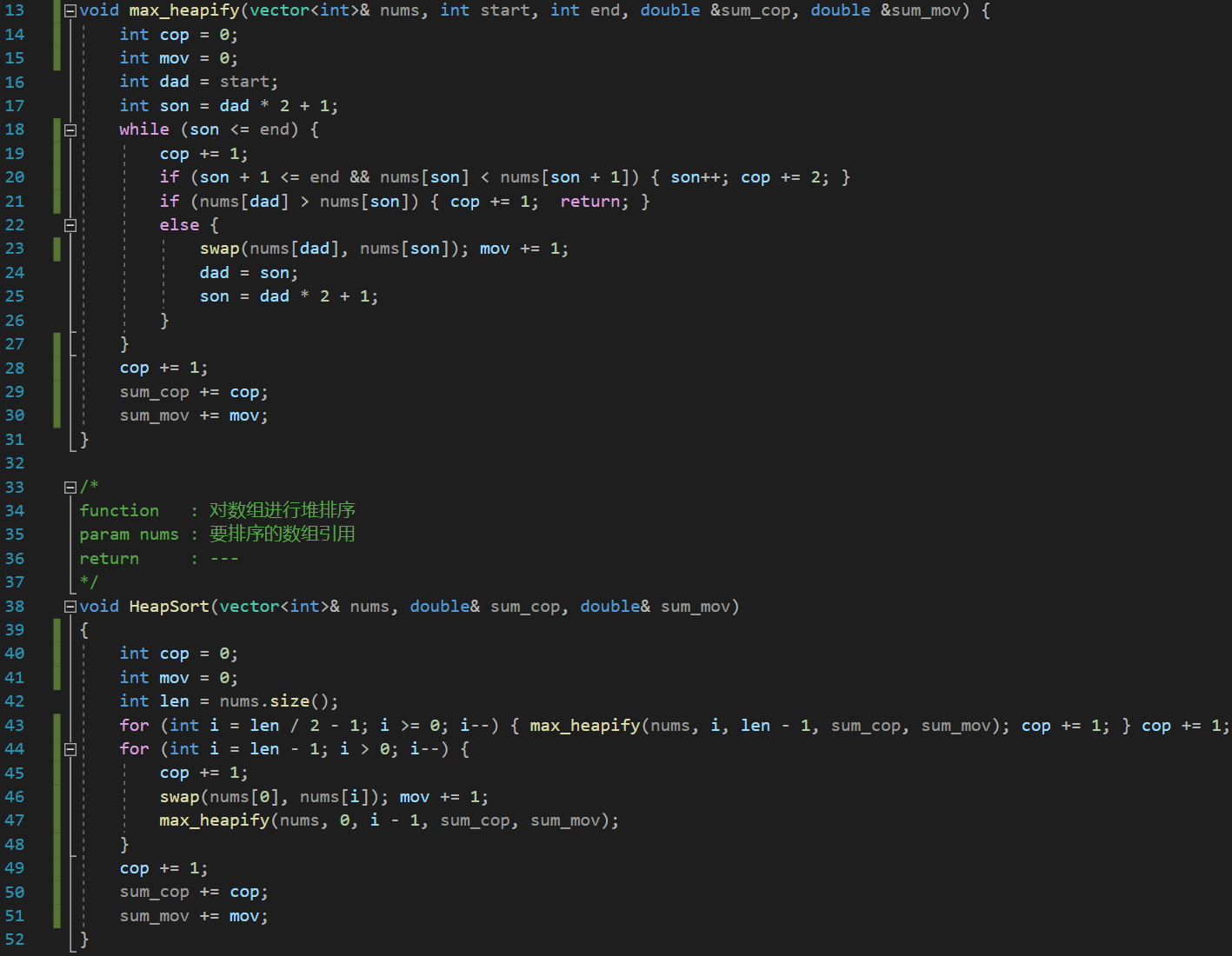


图16：修改后的HeapSort()函数

使用上述方法修改计数排序如下图所示：

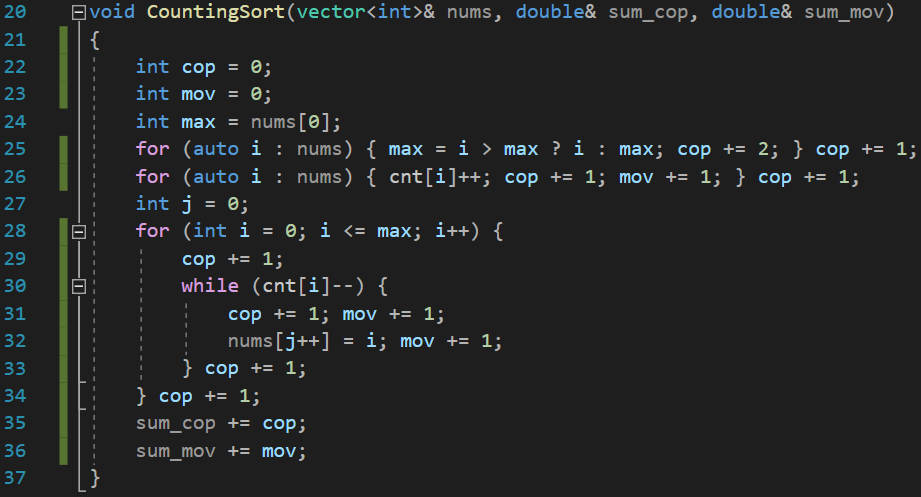


图17：修改后的CountingSort()函数

各个排序算法插桩完成后，分别运行程序，记录下数组元素大小为2000，循环执行100次，每次循环数组元素随机生成情况下的各排序算法的平均执行时间、比较次数和移动次数，得到结果如下表所示：

表1：不同排序算法运行表现统计

|  |  |  |  |
| --- | --- | --- | --- |
| 排序算法 | 平均运行时间 | 平均比较次数 | 平均移动次数 |
| 插入排序 | 0.04671 s | 2002780 | 998393 |
| 冒泡排序 | 0.19005 s | 4002000 | 998393 |
| 选择排序 | 0.09164 s | 4002000 | 1999 |
| 希尔排序 | 0.00206 s | 54950 | 36923 |
| 归并排序 | 0.00182 s | 70901 | 44000 |
| 快速排序 | 0.00115 s | 2666 | 1332 |
| 堆排序 | 0.00322 s | 35977 | 17550.1 |
| 计数排序 | 0.00205 s | 250914 | 6000 |

## 2.3 数组规模对各排序算法的影响

数组规模由main()函数中的变量vector\_size来决定，因此改变数组规模即改变vector\_size的大小。针对各排序算法，固定重复执行次数num\_repeat的大小，更改vector\_size的大小，分别设置为500、1000、1500、2000、2500、3000、3500，执行程序，记录下插入排序、冒泡排序、选择排序算法的平均执行时间，如下图所示：

图18：插入排序、冒泡排序、选择排序执行时间

**显然，插入排序、冒泡排序和选择排序的时间复杂度均符合预期O(n2)，并且，虽然三者时间复杂度的表示相同，但是从执行时间上看，插入排序快于选择排序快于冒泡排序。**

**我认为，冒泡排序每次遍历都需要在未排序的部分中，不断比较相邻元素并进行交换，所以消耗时间最长；选择排序虽然减少了交换次数，但是每次仍需遍历所有未排序部分；而插入排序，每次遍历时将一个未排序的元素与已排序的部分进行比较，所以平均情况下，只需要比较到已排序部分的一半即可停止，所以消耗时间最短。**

使用同样的方法记录下希尔排序、归并排序、快速排序、堆排序的平均执行时间，如下图所示：

图19：希尔排序、归并排序、快速排序、堆排序执行时间

**显然，希尔排序、归并排序、快速排序、堆排序均符合O(nlogn)的时间复杂度，并且，从执行算法所需时间上来看，快速排序最好，堆排序最差，归并排序优于希尔排序。**

使用同样的方法记录下计数排序的平均执行时间，如下图所示：

图20：计数排序执行时间

**显然，计数排序符号预期O(n)的时间复杂度，算法执行时间随着规模增大有所增加，但增量很小。**

**同时，我也探究了这些排序算法中表现最好的快速排序与C++ STL中的sort ()函数执行所需时间的对比，结果如下图：**

图21：快速排序、STL sort()函数执行时间

**出乎预料的是，最基本的“两头凑”快速排序竟然比STL sort()函数执行所需时间更短！**

## 2.4 快速排序不同实现的性能比较及优化

使用三种方法实现快速排序，分别是“两头凑”、“空穴法”和“追赶法”。其中，“两头凑”已经在QuickSort.cpp中实现，下面将在QuickSort1.cpp中实现“空穴法”，在QuickSort2.cpp中实现“追赶法”。

实现“空穴法”代码如下图所示：

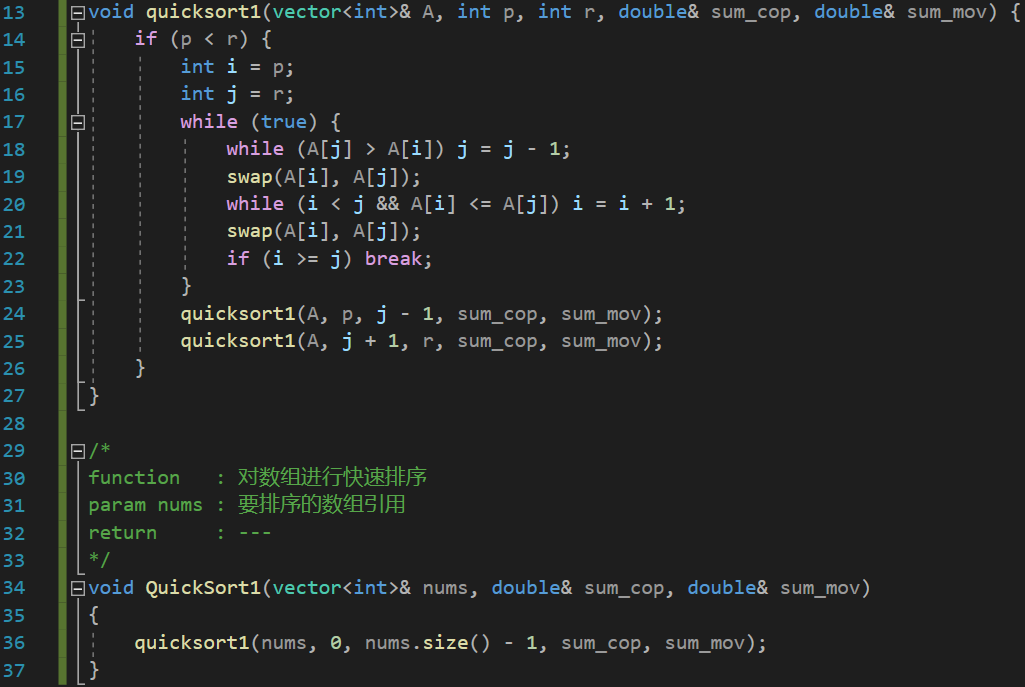


图22：“空穴法”快速排序

实现“追赶法”代码如下图所示：

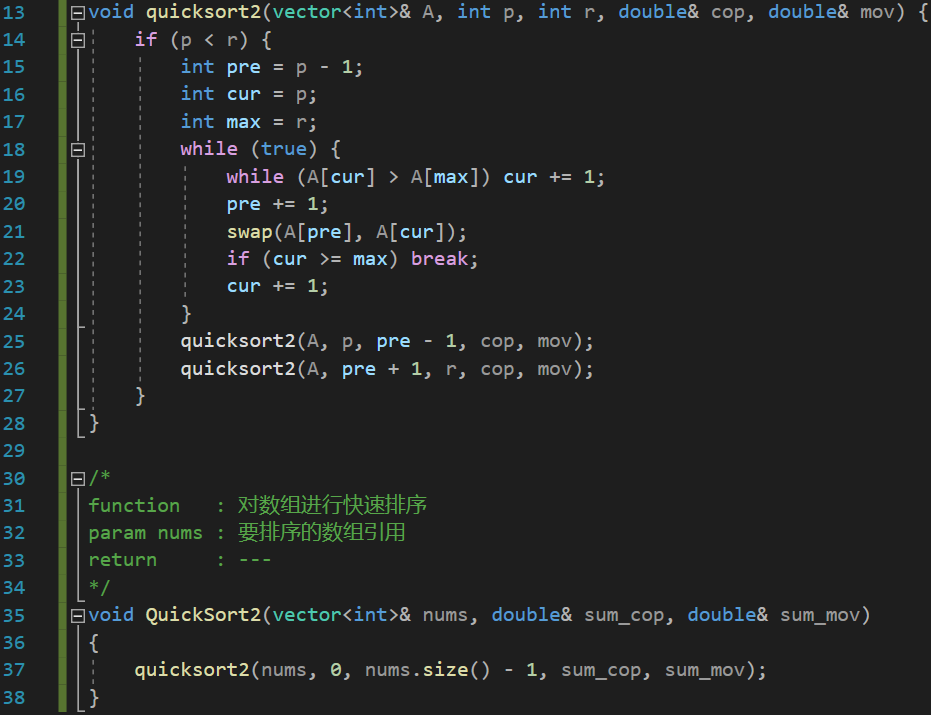


图23：“追赶法”快速排序

使用同样的方法，分析三种快速排序算法执行所需时间，得到结果如下图所示：

**显然，三种排序算法都是相对高效快速的。其中，“空穴法”和“追赶法”运行时间相差不大，而“两头凑”法运行时间明显要更短，运行速度也更快，是更高效的快速排序算法。**

# 三、实验心得体会

1. 通过此实验，我对十大排序的时间复杂度、稳定性、具体实现方法有了更加深刻的体会和更好的掌握，对各种算法的计算思维有了更深入的了解；

2. 通过对不同排序算法进行分析，我直观地感受到不同排序算法执行所需时间的差异，感受到理论分析与实际执行的一致性和差异性，同时，对各种时间复杂的算法的执行时间随着规模的变化有更加深刻的体会；

3. 通过对比快速排序与C++ STL sort()函数的执行效率，我惊讶于经过对各种排序算法综合优化的sort()函数的执行效率竟低于快速排序，仍值得深入研究。