上海大学
Shanghai University

2023-2024 夏季学期

《计算机程序设计实训》课程小论文

COURSE PAPER

基于 C 语言对于快速排序、归并排序、堆排序算法的探索与优化

学　　院　计算机工程与科学学院

专　　业　网络空间安全

组　　长　范舒舰

组员姓名　王展鹏 陶宬洋 吴征鸿

摘　要　本文主要介绍了归并排序、堆排序、和快速排序四种排序算法，给出了算法的核心代码、时间复杂度、空间复杂度、排序算法的稳定性，以及排序算法对数据构型的“适用性”，并对其进行了优化与分析。同时，本文对不同存储方式的 C-字符串数组开展排序研究，分析了不同存储方式能够进行/不能进行的操作方法。最后给出了课程学习的心得体会。

关键词　C 语言；快速排序；归并排序；堆排序

Exploration and Optimization of Quick Sort, Merge Sort, and Heap Sort Algorithms Based on Combined Programming Language

Shujian Fan　Zhanpeng Wang　Chengyang Tao　Zhenghong Wu

(School of Computer Engineering and Science)

Abstract　This article mainly introduces four sorting algorithms: merge sort, heap sort, and quick sort. It provides the core code, time complexity, space complexity, stability of the sorting algorithms, as well as their "applicability" to different data structures. It also conducts optimization and analysis of these algorithms. Additionally, the article conducts a study on sorting C-string arrays with different storage methods, analyzing the operable and non-operable methods for different storage approaches. Finally, it presents the author's learning experience and insights from the course.

Key words Combined Programming Language; Quick Sort; Merge Sort; Heap Sort

1 引言

排序是数据处理中的常用基本操作，它通过元素间的比较、交换或移动（多次赋值）实现数据元素按某种顺序进行重新排列。采取不同的比较、交换或移动策略形成了不同的排序算法。不同的排序算法中对数据元素进行的比较次数、赋值次数可能有较大的差别，影响整个排序操作的效率。本小组在综合比较了八种常用排序：插入排序，share 排序，快速排序，归并排序，堆排序，基数排序，冒泡排序，简单选择排序后选择了效率较高的快速排序，堆排序，归并排序作为研究对象。在本文的排序算法实验研究中，通过统计排序算法函数的运行时间、数组元素间比较次数、数组元素间赋值次数，用图、表等方式来展现并分析相应算法的时间复杂度、空间复杂度、排序的稳定性并得出一定的结论。

2 排序算法概述及优化思路

2.1 快速排序

快速排序是一种高效的排序算法，采用分治法，其基本思想是选择一个基准值（pivot），然后重新排列数组，使得所有比基准值小的元素都在基准值的左边，所有比基准值大的元素都在基准值的右边。这个过程称为分区（partitioning）。之后，递归地对基准值左边和右边的子数组进行快速排序。三数取中法是通过取数组首项，位置中间项与末项进行比较，将值中等的一项作为切分项的方法。这增加了所找到切分点为中间值的概率。使用了 mid=size/2，在 a[high],a[mid],a[low]中选取中间值作为 pivot，通过大小比较与交换使得 a[mid]<a[low]=pivot<a[high]。

它的优点是效率极高，在大多数情况下，效率优于其他算法，且进行原地排序，空间效率高，在随机集上极少出现最坏情况，可预测性较高，且具有较高的适应性，在数组列表的情况下，均有较好的表现。

三数取中

核心思路就是，选取三个数中中间的那个数字，作为 key（基准），尽量从中间分割数组，从而减少递归次数。

以下是实现快速排序的代码（以 int 数组为例）：

```
void swap(int *a, int *b) {
```

```
    int temp = *a;
    *a = *b;
    *b = temp;
}
void F_Sort(int *a, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;          // 防止整数溢出
        int pivot_idx = low;
        if (a[low] > a[high]) swap(&a[low], &a[high]);
        if (a[mid] > a[high]) swap(&a[mid], &a[high]);
        if (a[low] > a[mid]) swap(&a[low], &a[mid]); // 三数取中，选择 pivot_idx
的位置
        int pivot = a[pivot_idx];                  // 将选择的 pivot_idx 位置的元素
作为 pivot
        int i = low, j = high;
        while (i <= j) {
            while (i <= j && a[i] < pivot) i++;
            while (i <= j && a[j] > pivot) j--;
            if (i <= j) {
                swap(&a[i], &a[j]);
                i++;
                j--;
            }
        }
        F_Sort(a, low, j);                         // 递归排序左部分
        F_Sort(a, i, high);                        // 递归排序右部分
    }
}
```

2.2 归并排序

归并排序算法有两个基本的操作，一个是分，也就是把原数组划分成两个子数组的过程。另一个是治，它将两个有序数组合并成一个更大的有序数组。将待排序的线性表不断地切分成若干个子表，直到每个子表只包含一个元素，这时，可以认为只包含一个元素的子表是有序表。将子表两两合并，每合并一次，就会产生一个新的且更长的有序表，重复这一步骤，直到最后只剩下一个子表，这个子表就是排好序的线性表。

由于可将数值相等的两个数据按照原有的顺序进行排序，归并排序具有较好的稳定性，且归并排序的时间复杂度始终为 O(n log n)，无论是在最好、平均还是最坏情况下，它都能提供稳定的性能，这使得他有较好的可预测性，同时，它的性能并不会随着数据量的增加而显著下降，这使得它在处理大量数据时有较好的表现。

以下是实现归并排序的代码（以 int 数组为例）：

```
void merge(int *a, int left, int mid, int right, int *temp)
{
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right)
    {
```

```
        if (a[i] <= a[j])
        {
            temp[k++] = a[i++];
        }
        else
        {
            temp[k++] = a[j++];
        }
    }
    while (i <= mid)
    {
        temp[k++] = a[i++];
    }
    while (j <= right)
    {
        temp[k++] = a[j++];
    }
    for (i = left, k = 0; i <= right; i++, k++)
    {
        a[i] = temp[k];
    }
}
```

2.3 堆排序

堆排序（Heap Sort）是一种基于堆数据结构的排序算法，其核心思想是将待排序的序列构建成一个最大堆（或最小堆），然后将堆顶元素与最后一个元素交换，再将剩余元素重新调整为最大堆（或最小堆），重复以上步骤直到所有元素都有序。堆是一个几乎完全的二叉树，其中每个父节点都比它的子节点具有更高的值（在最大堆中）或更低的值（在最小堆中）。堆排序通常使用最大堆来实现。

堆排序作为一种原地排序算法，它不需要额外的储存空间，这让他有了一个较低的空间复杂度，同时堆排序的性能不受数据顺序的影响，这使得他有很好的可预测性，且虽然构建推需要花费较多时间，但在处理大量数据时，构建堆所花费的时间与排序所花的时间相比，可以忽略不计，这使得他在处理大量数据时，尤其是当并不需要所有数据的顺序，而是取出前 100 个等情况时，它有极好的表现。

以下是实现堆排序的代码（以 int 数组为例）：

```
void HeapSort(int *arr, int size) {
    // 构建最大堆
    for (int i = size / 2 - 1; i >= 0; --i) {
        adjustHeap(arr, size, i);
    }

    // 堆排序
    for (int i = size - 1; i > 0; --i) {
        // 交换堆顶元素和当前未排序部分的最后一个元素
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
```

```
        // 重新调整堆结构
        adjustHeap(arr, i, 0);
    }
}

// 调整堆结构的函数
void adjustHeap(int *arr, int size, int i)
{
    int largest = i; // 初始化最大元素索引为根节点
    int left = 2 * i + 1; // 左子节点索引
    int right = 2 * i + 2; // 右子节点索引

    // 找出左、右子节点和根节点中的最大值
    if (left < size && arr[left] > arr[largest])
    {
        largest = left;
    }
    if (right < size && arr[right] > arr[largest])
    {
        largest = right;
    }

    // 如果最大元素不是根节点，则交换根节点和最大元素，并递归调整
    if (largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        adjustHeap(arr, size, largest);
    }
}
```

3 算法数据收集

3.1 算法测试结果及数据图表

我们通过 run.bat 批处理文件生成排序算法的时间、比较次数、赋值次数等数据。在此展示部分数据及表格。

表 1.正态分布整型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.000	0.000	0.008

16384	0.002	0.001	0.113
65536	0.006	0.005	栈溢出
262144	0.027	0.022	栈溢出

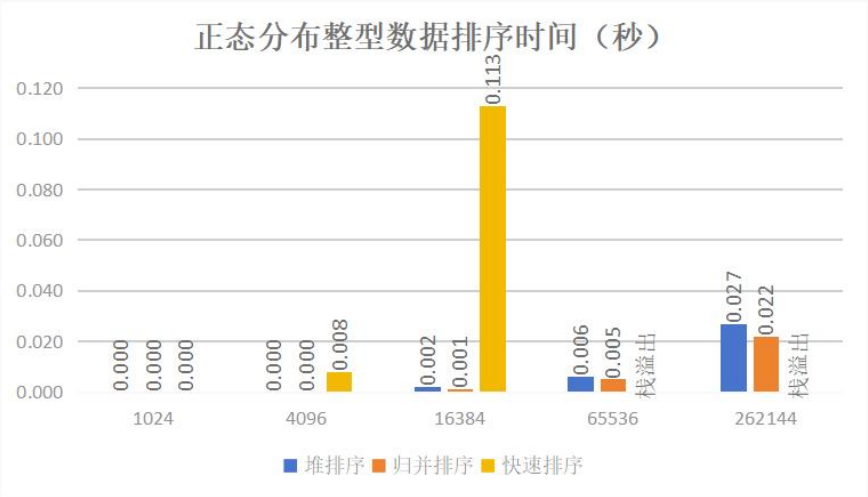


表 2.均匀分布整型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.001	0.001	0.003
16384	0.002	0.001	0.039
65536	0.005	0.006	栈溢出
262144	0.027	0.022	栈溢出

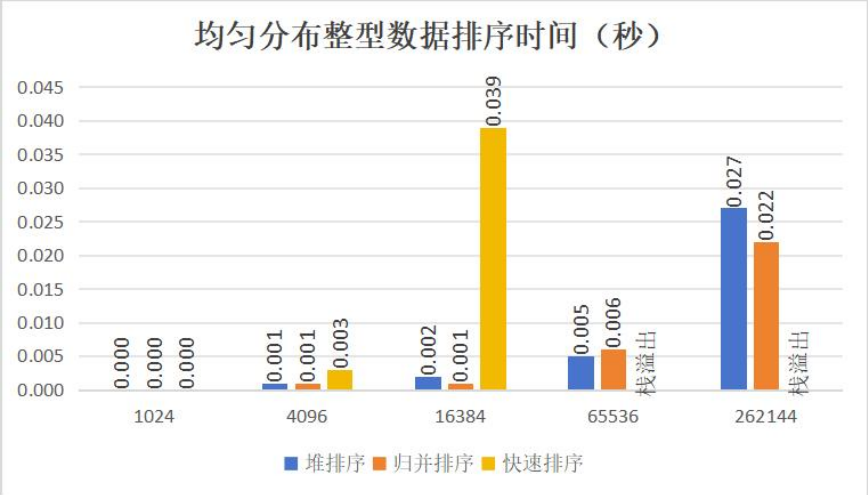


表 3.顺序分布整型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.000	0.000	0.000

16384	0.001	0.002	0.000
65536	0.008	0.007	0.004
262144	0.031	0.030	0.015

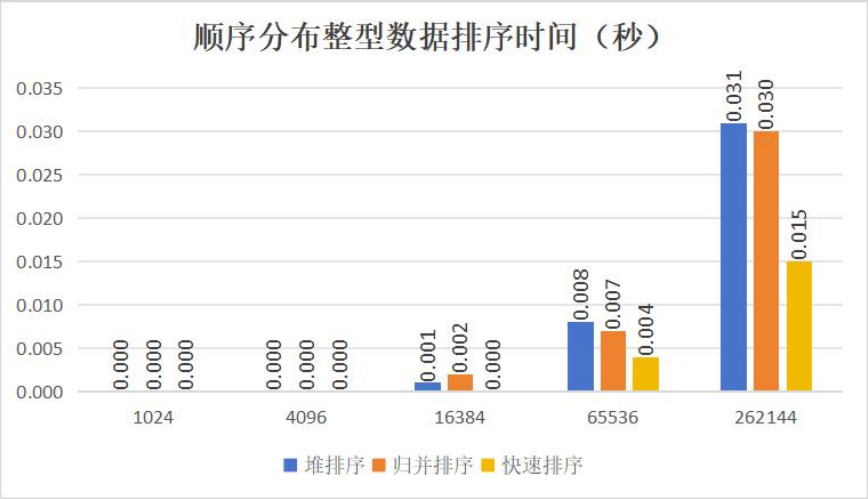


表 4.逆序分布整型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.001	0.001	0.000
16384	0.001	0.001	0.001
65536	0.007	0.007	0.004
262144	0.031	0.030	0.015

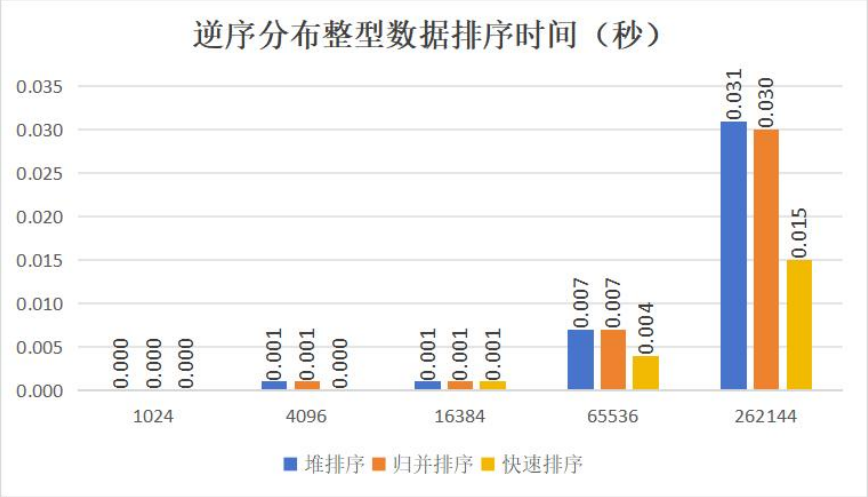
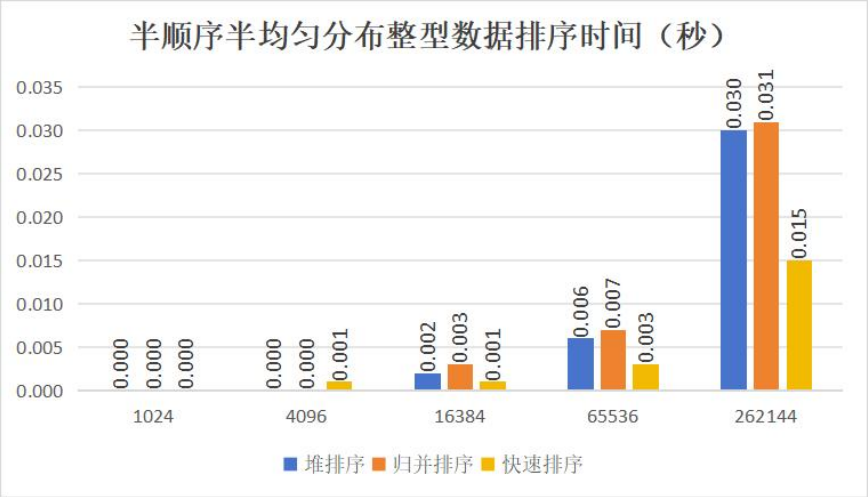


表 5.半顺序半均匀分布整型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000

4096	0.000	0.000	0.001
16384	0.002	0.003	0.001
65536	0.006	0.007	0.003
262144	0.030	0.031	0.015



3.2 复杂度概念

3.2.1 时间复杂度

衡量一个算法的快慢，一定要考虑数据规模的大小。所谓数据规模，一般指输入的数字个数、输入中给出的图的点数与边数等等。一般来说，数据规模越大，算法的用时就越长。而在算法竞赛中，衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即时间复杂度。

考虑用时随数据规模变化的趋势的主要原因有以下几点：

①现代计算机每秒可以处理数亿乃至更多次基本运算，因此处理的数据规模通常很大。在允许算法执行时间更久时，时间复杂度对可处理数据规模的影响就会更加明显，远大于同一数据规模下用时的影响

②采用基本操作数来表示算法的用时，而不同的基本操作实际用时是不同的，例如加减法的用时远小于除法的用时。计算时间复杂度而忽略不同基本操作之间的区别以及一次基本操作与十次基本操作之间的区别，可以消除基本操作间用时不同的影响。

当然，算法的运行用时并非完全由输入规模决定，而是也与输入的内容相关。所以，时间复杂度又分为几种，例如：

①最坏时间复杂度，即每个输入规模下用时最长的输入对应的时间复杂度。在算法竞赛中，由于输入可以在给定的数据范围内任意给定，为保证算法能够通过某个数据范围内的任何数据，一般考虑最坏时间复杂度。

②平均（期望）时间复杂度，即每个输入规模下所有可能输入对应用时的平均值的复杂度（随机输入下期望用时的复杂度）。

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 T(n)表示，若有某个辅助函数 f(n),使得当 n 趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称 f(n)是 T(n)的同数量级函数。记作 T(n)= O(f(n)),称 O(f(n))为算法的渐进时间复杂度。

3.2.2 空间复杂度

一个程序的空间复杂度是指运行完一个程序所需内存的大小。利用程序的空间复杂度，可以对程序的运行所需要的内存多少有个预先估计。一个程序执行时除了需要存储空间和存储本身所使用的指令、常数、变量和输入数据外，还需要一些对数据进行操作的工作单元和存储一些为现实计算所需信息的辅助空间。程序执行时所需存储空间包括以下两部分。

①固定部分。这部分空间的大小与输入/输出的数据的个数多少、数值无关。主要包括指令空间（即代码空间）、数据空间（常量、简单变量）等所占的空间。这部分属于静态空间。

②可变空间，这部分空间的主要包括动态分配的空间，以及递归栈所需的空间等。这部分的空间大小与算法有关。一个算法所需的存储空间用 f(n)表示。S(n)= O(f(n))其中 n 为问题的规模，S(n)表示空间复杂度。

排序方法	时间复杂度	空间复杂度
堆排序	$O(n\ logn)$	$O(1)$
归并排序	$O(n\ logn)$	$O(n)$
快速排序	$O(n\ logn)$	$O(n^2)$

3.3 测试结果分析

三种排序算法（堆排序、归并排序、快速排序）在不同数据类型和数据分布下展现了各自的特点和优劣。

堆排序和归并排序在大多数测试场景下性能相近，它们的时间复杂度都接近 O(nlogn)，且不受数据分布的影响，表现出较好的稳定性，能够适应 1024 到 262144 不同大小、类型数据集的排序。堆排序通过构建最大堆或最小堆来实现排序，其空间复杂度较低，适用于内存受限的场景。而归并排序则通过不断将数组分割成更小的子数组，并将它们合并成有序数组，其优点在于能够并行处理，但在合并过程中可能需要额外的存储空间。

快速排序在平均情况下性能优异，虽然时间复杂度也接近 O(nlogn)，但是在处理大量元素容易栈溢出。它通过选择一个基准元素，将数组划分为左右两个子数组，并对子数组进行递归排序。然而，快速排序在处理某些数据分布（如高斯分布和均匀分布）时，可能因递归过深导致栈溢出，这限制了它在大数据量下的应用。此外，快速排序的稳定性也较差，相同元素的相对位置在排序后可能会发生变化。

从比较和赋值次数来看：堆排序的比较次数随着数据量的增大而稳定增加，显示出较好的时间复杂度。其性能不受数据分布的影响，无论是顺序、逆序还是高斯分布，其性能都较为稳定。归并排序的比较次数随着数据量的增大而稳定增加，与堆排序类似，显示出较好的时间复杂度。赋值次数在所有测试中都保持不变，极大可能为代码出现错误。

快速排序在平均情况下的比较次数和赋值次数都较少，显示出优异的性能。然而，当数据分布呈现高斯分布时，快速排序的比较次数急剧增加，可能是递归出现了问题，导致比较次数呈 n^2 展示。赋值次数在大多数情况下也随数据量增大而增加。在某些排序下，由于递归层数过深，可能导致栈溢出问题，无法完成排序。

总的来说，堆排序和归并排序在各种排序中表现出较好的稳定性和可靠性，且时间复杂度接近。而快

速排序在某些情况下性能优异，但在特定数据分布下（如高斯分布）可能表现较差，且存在栈溢出风险。

4 C-字符串处理总结(思考题)

1.下面的测试函数中，

(1) 请先思考然后进行程序验证 sizeof(strA), sizeof(strB), sizeof(strC), sizeof(strD)各为多少字节，这些字节位于内存什么区域（代码区、全局数据区、栈区、堆区）？

(2) strA、strB、strC、strD 联系的 C-字符串数组的内容存储在内存的什么区域？它们的读/写属性（是否可读、可写）如何？

答：

(1)sizeof(strA)将会返回整个数组的大小，即 NUM * sizeof(char) * 数组中字符串的数量。在这个例子中，NUM 是 20，所以 sizeof(strA)是 20 * sizeof(char) * 15（假设每个字符串最多 20 个字符，数组中有 15 个字符串）。sizeof(strB)将会返回指针数组的大小，即 sizeof(char*) * 数组中字符串的数量。在这个例子中，它是 sizeof(char*) * 15。sizeof(strC)和 sizeof(strD)在声明时都是 char**类型，因此它们的大小都是 sizeof(char**)。

(2)strA 存储在栈区，strB 联系的字符串存储在常量区，strC、strD 存储在堆区。当字符数组和字符指针都是局部变量时，字符数组是申请在栈区，字符串的每一个字符存储在这个字符数组的每一个元素中。指针变量是声明在栈区的，字符串数据是以字符数组的形式存储在常量区的，指针变量中存储的是字符串在常量区的地址。

2.设计 BubbleA，BubbleB 两个函数之前，思考

(1) 如何比较两个字符串的内容？

(2) 存储在什么区域的字符串能交换其内容？

(3) 若不能交换字符串的内容，排序操作中交换什么？

答：

(1) 如何比较两个字符串的内容？

调用 strcmp 函数，按照 ASCII 码比较(字典序)，当 s1 和 s2 相等时返回 0，若 s1 大则返回 1，否则返回-1。

(2) 存储在什么区域的字符串能交换其内容？

只有存储在堆区或栈区的字符串可以交换其内容。全局数据区和代码区的字符串通常是只读的，不能交换。当我们以字符指针的形式要将字符串数据存储到常量区的时候，并不是直接将字符串存储到常量区，而是先检查常量区中是否有相同内容的字符串，如果有直接将这个字符串的地址拿过来返回，如果没有，才会将这个字符串数据存储在常量区中；当我们重新为字符指针初始化一个字符串的时候，并不是修改原来的字符串，而是重新的创建了一个字符串，把这个新的字符串的地址赋值给它。

(3) 若不能交换字符串的内容，排序操作中交换什么？

交换地址。如果不能交换字符串的内容，排序操作中交换的是指向字符串的指针。

GetStringsA 和 GetStringB 函数的第一个形式参数为什么需要用到三级指针，如果仅用二级指针会怎样？GetStringsA1 函数应该如何使用？

答： *dest = (char**)calloc(n, sizeof(char**))

由于 dest 指向的是一个指针（字符）数组中，其元素指向的对象的地址，这些函数需要修改调用者的指针变量，以指向新分配的内存。如果使用二级指针，函数只能修改传入的指针副本的值，而不能修改调用者的原始指针。使用三级指针允许函数修改调用者的二级指针，从而使其指向新分配的内存。因此只能用指针的指针的指针，即三级指针，才能改变指针的指向。如果不用三级指针，则指向的是此指针数组元素的地址，而不是元素指向的对象的地址。

要调用 GetStringsA1 函数，需要用一个字符指针指针变量来接收其返回值。

FreeStrings 函数的形式参数为什么需要用到三级指针？（或回答如下问题）

如果仅用二级指针（见 FreeStrings1 函数），能否释放所申请的堆内存资源？FreeStrings 函数与 FreeStrings1 函数的关键不同点是什么？

答：

FreeStrings 函数的形式参数使用三级指针（char ***strs）的原因是因为它不仅要释放每个字符串（即字符数组）所占据的堆内存，还要确保能够修改调用者传递的指向这些字符串指针的指针（即二级指针）本身。这是为了将二级指针设置为 NULL，以防止悬挂指针（dangling pointer）的产生，悬挂指针是指那些指向已经被释放的内存的指针。

FreeStrings1 函数则使用了二级指针（char **strs）作为参数。这个函数能够完成释放每个字符串（即字符数组）所占据的堆内存的任务，因为它遍历了 strs 数组，并对每个非空的字符串指针调用了 free 函数。然而，FreeStrings1 函数的一个关键缺陷是它无法修改调用者传递的 strs 指针本身。也就是说，即使在 FreeStrings1 函数内部将 strs 设置为 NULL，这也不会影响到调用者所持有的那个二级指针的值。这可能会导致悬挂指针的产生，因为调用者可能不知道 strs 所指向的内存已经被释放了。

因此，FreeStrings 函数和 FreeStrings1 函数的关键不同点在于它们如何处理传递给它们的二级指针。FreeStrings 函数能够确保调用者所持有的二级指针也被正确地设置为 NULL，而 FreeStrings1 函数则不能。这是通过 FreeStrings 函数使用三级指针作为参数来实现的。

```
void BubbleA(char(*str)[NUM], int size)        // 数组指针
{
    int flag = 1;
    int temp = 0;
    int i, j, h, t;
    char* space = (char*)calloc(NUM,sizeof(char));
    for (i = 1; i < size; i++) {
        for (j = 0; j < size - 1; j++) {
            for (h = 0; flag && h < NUM; h++) {
                if (*(*(str + j) + h) < *(*(str + j + 1) + h)) {
                    flag = 0;
                }

                if (*(*(str + j) + h) > * (*(str + j + 1) + h)) {
                    temp = 1;
                    flag = 0;
                }
                if (temp == 1) {
                    for (t = 0; t < NUM; t++) {
                        *(space + t) = *(*(str + j) + t);
                    }
                    for ( t = 0; t < NUM; t++) {
                        *(*(str + j) + t) = *(*(str + j + 1) + t);
                    }
                    for ( t = 0; t < NUM; t++) {
                        *(*(str + j + 1) + t) = *(space + t);
                    }
                }
                temp = 0;
```

```
            }
            flag = 1;
        }
    }
    printf("请完成函数 BubbleA 的定义，执行排序操作。\n");
}
```

```
void BubbleB(char* str[], int size)                // 指针数组
{
    int flag = 1;
    int temp = 0;
    int i, j, h;
    char* space = (char*)calloc(NUM,sizeof(char));
    for (i = 1; i < size; i++) {
        for (j = 0; j < size - 1; j++) {
            for (h = 0; flag && h < NUM; h++) {
                if (*(*(str + j) + h) < *(*(str + j + 1) + h)) {
                    flag = 0;
                }

                if (*(*(str + j) + h) > * (*(str + j + 1) + h)) {
                    temp = 1;
                    flag = 0;
                }
                if (temp == 1) {
                    space = *(str + j);
                    *(str + j) = *(str + j + 1);
                    *(str + j + 1) = space;
                }
                temp = 0;
            }
            flag = 1;
        }
    }
    printf("请完成函数 BubbleB 的定义，执行排序操作。\n");

}
```

5 小组体会

5.1 贡献

姓名	分工	贡献度
范舒舰	大部分源文件编写，部分注释，函数封装，论文结构，部分论文主体	45%
王展鹏	部分源文件编写，部分注释，部分论文主体	30%
陶宬洋	小部分源文件编写	5%
吴征鸿	大部分论文主体，部分源文件编写	20%

5.2 心得体会

姓名	感想与建议
范舒舰	在本次实训中，我学到了如何调用函数，深刻地了解到指针的定义以及函数元素类型的定义。本次函数主体部分和所调用的函数大部分由我编写，在函数的实际编写中，由于已经数月未用 C 语言写过代码略显生疏，但是提供的 Sort.c 函数以及 Test 中的代码给了我不小启示，让我学会了代码的封装，函数的调用，头文件的书写等等。编写排序代码的途中遇到的各种问题，如变量类型出错，编译一直报错等问题在不懈攻克下还是迎刃而解。学校到的论文排版、批处理文件的知识相信能在今后的学习生活中不多发挥作用。成功地完成了项目要求的大部分内容，夏季实训帮我重拾了写代码的记忆。希望之后的夏季实训课程能够注重讲解，一知半解地去写代码遇到的困难确实会令人无从下手。
王展鹏	这次的体验让我对从未接触过的大作业有了全新的感受，作业中，一些关键时刻让我抓耳挠腮，不停的查找资料的挫折更让我意识到了基础的重要性，小组作业中，一些合作时，大家没想到过的障碍，也更让我明白了团队配合的重要性，虽然这次作业已经完成了，但我对批处理文件仍然只是一知半解，希望之后的课程中能对这些从未学习到的知识有更多的引导。
陶宬洋	该同学未填写
吴征鸿	纸上得来终觉浅，本次实训让我重新拾起 C 语言。从联赛中我体悟到编程对解决复杂实际问题的帮助。本次主要学习了排序算法与指针，负责一些修改编辑工作，体会到自身知识储备不足与缺乏实践能力。这激发了我更加努力地学习，以便更好地应对未来的挑战。感谢组内各位的付出，共同学习进步。

5.3 反思

首先，在之前的学习中，我们并没有做过这种大作业，也没有尝试过组队编写代码，没有尝试过多文件的程序编写方式，也没有尝试过批处理，也没有尝试过深入研究某一种算法，即使老师在课上已有简单讲解，但当我们在实际操作时，却仍然一头雾水，这一过程成为了我们前行路上的大阻碍，好在经过了资料的查找以及组内的互帮互助后，我们初步掌握了代码的编写以及逻辑的串联。

其次，由于函数是分工编写的缘故，导致许多函数名和变量名的格式不统一，大家在阅读他人写的代码时比较费劲，在总体整理时也经常会出现自己认为的函数与实际函数不符而导致各种报错，这次的经历让我们知道了，在之后写代码的时候，最好组内统一规范格式，并在自己写的代码旁边做一些简单的注释，变量名也不适宜随意取，而是能让人一眼就看出它的实际意思。在实际的工作过程中，我们编写读取双浮点数组的排序算法时，就遇到了变量名错乱的情况，导致主函数经过了数个小时的调试才成功运行小容量数据，中间又出现快速排序栈溢出的故障，导致我们不得不跳过一些数组以生成时间数据。

文件的传递也给我们带来了不小的困扰，在小组配合时，我们经常出现一个人的代码，写好了在自己的电脑上能跑，但将它发到群里，由其他组员下载时却发现代码跑不了的情况，造成这个问题的原因有很多，编译环境不同，又或是传递时产生了格式错误，还有修改完忘了保存等各种情况，这次的经历也为我们敲响了警钟，让我们在之后的大作业中更加注意。

同时，由于很久不再运用 C 语言，导致我们对 C 语言的一些运用给予法规的产生了一部分的遗忘，这次的大作业不仅很好的为我们提供了一个复习的机会，也警醒我们抓紧利用暑假时间夯实基础，为开学后学习新的知识做好充足的准备。

6 结语

在本论文中，我们对排序算法的优化及处理的内容进行了展示。通过该研究，不仅仅让我们 对排序算法与 C 语言有了更深入的了解，同时也锻炼了我们进行研究、查阅文献、文字编辑排版与 团队合作的能力，为我们日后的计算机研究学习奠定了良好的基础。

致谢 在本次论文设计过程中，感谢上海大学上海大学计算机工程与科学学院给了我们学习研究的机会。在学习中，老师们给予了细致的指导，提出了很多宝贵的意见与推荐，对老师们表示衷心的感谢。

感谢本研究小组中的各位同学的积极参与。在大家的共同努力下，顺利地完成了此次实训任务。 谨以此致谢。

最后，要向百忙之中抽时间对本文进行审阅的各位老师，表示衷心的感谢。

参 考 文 献

[1] CSDN技术社区 https://blog.csdn.net/qq_43179428/article/details/89481937

[2] CSDN技术社区 https://blog.csdn.net/qq_58325487/article/details/124068253

[3] CSDN技术社区 https://blog.csdn.net/2301_80636143/article/details/138220994

附录：算法测试结果及数据图表

表 6.正态分布双浮点型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.001
4096	0.001	0.000	0.008
16384	0.002	0.001	0.124
65536	0.008	0.005	栈溢出
262144	0.029	0.020	栈溢出

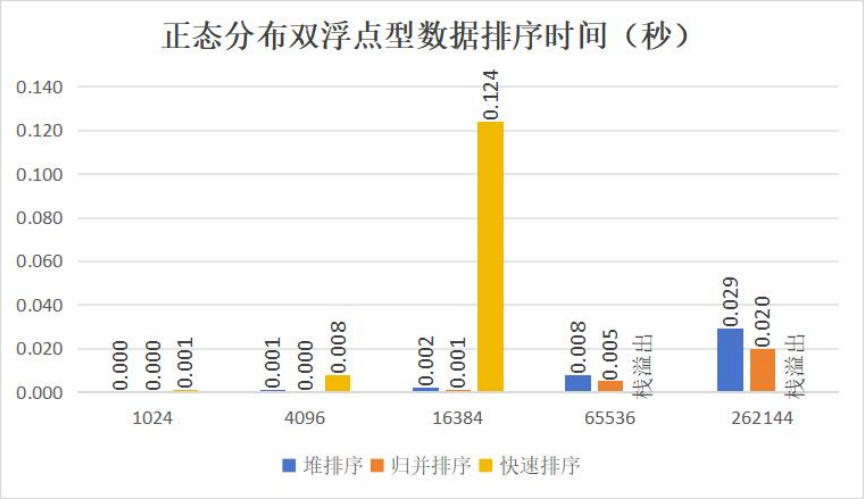


表 7.均匀分布双浮点型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.001	0.000	0.000
4096	0.001	0.000	0.000
16384	0.002	0.001	0.000
65536	0.007	0.004	栈溢出
262144	0.031	0.019	栈溢出

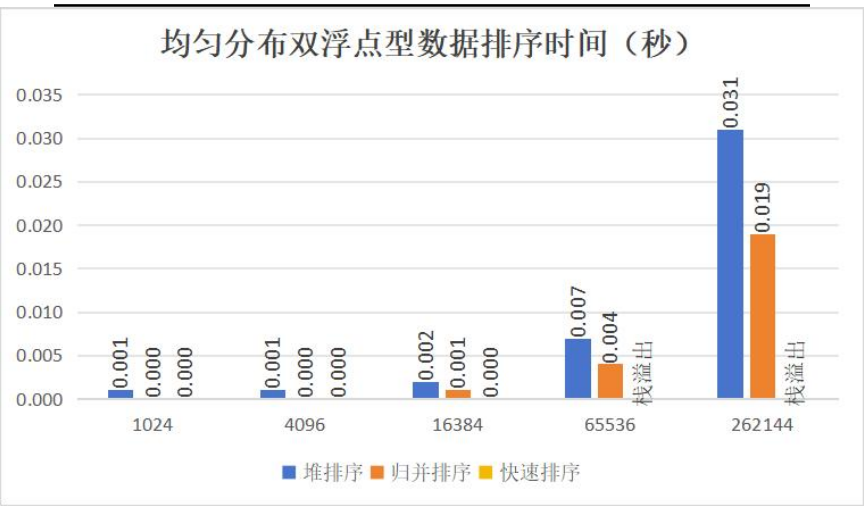


表 8.顺序分布双浮点型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.001	0.001	0.001
16384	0.003	0.002	0.001
65536	0.011	0.004	栈溢出
262144	0.051	0.019	栈溢出

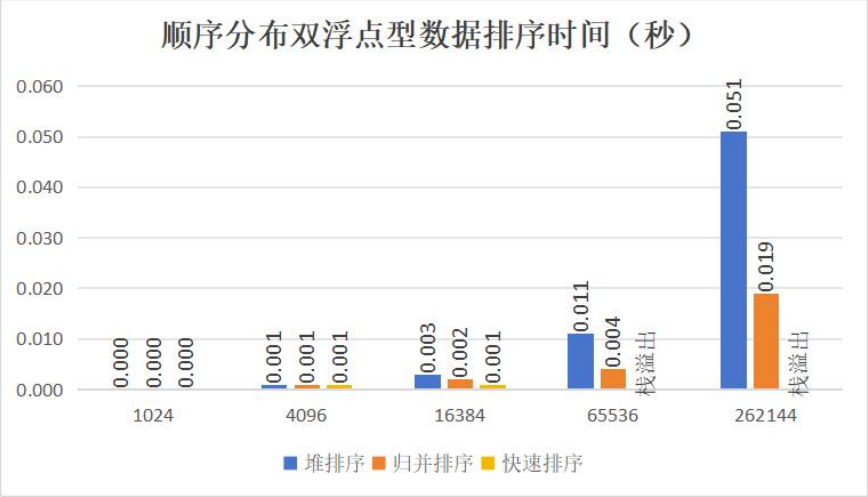


表 9.逆序分布双浮点型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	栈溢出
4096	0.000	0.001	栈溢出
16384	0.003	0.001	栈溢出
65536	0.011	0.004	栈溢出
262144	0.050	0.020	栈溢出

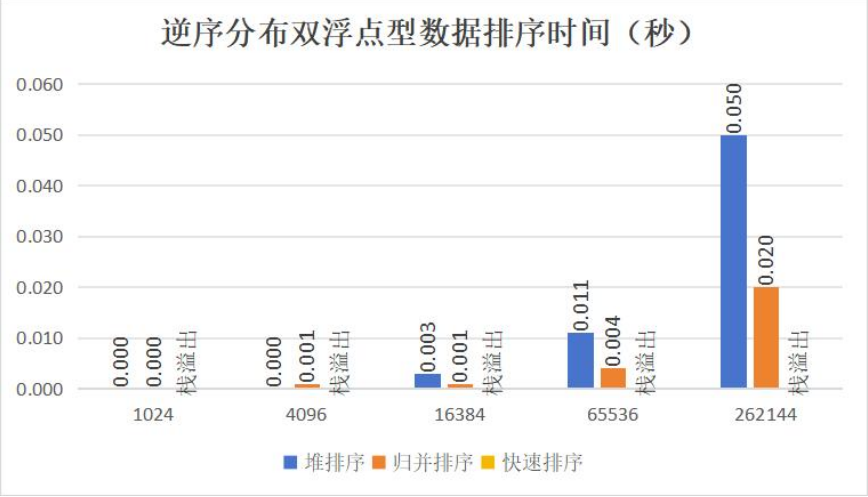


表 10.半顺序半均匀分布双浮点型数据排序时间（秒）

数据规模	堆排序	归并排序	快速排序
1024	0.000	0.000	0.000
4096	0.001	0.000	0.001
16384	0.004	0.001	0.001
65536	0.011	0.006	栈溢出
262144	0.048	0.021	栈溢出

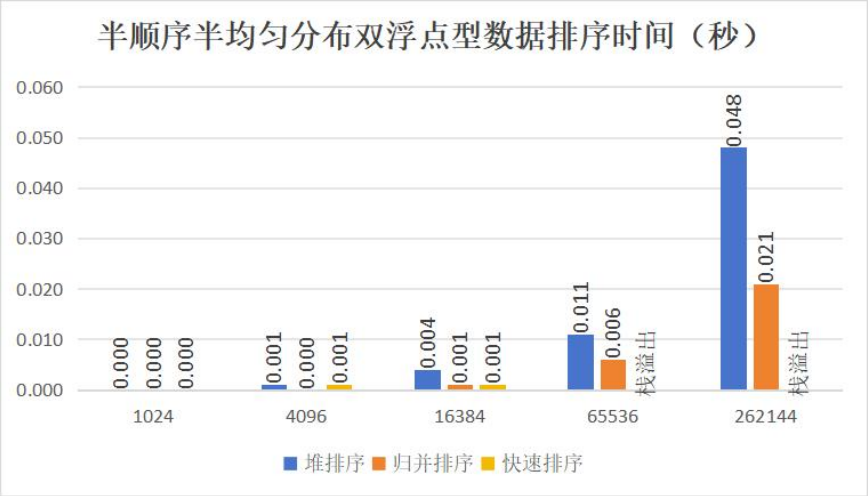


表 11.正态分布整型数据赋值次数（次）

数据规模	堆排序	归并排序	快速排序
1024	74713	20480	3069
4096	362737	98304	12285
16384	1695440	458752	49149
65536	7758555	2097152	栈溢出
262144	34976249	9437184	栈溢出

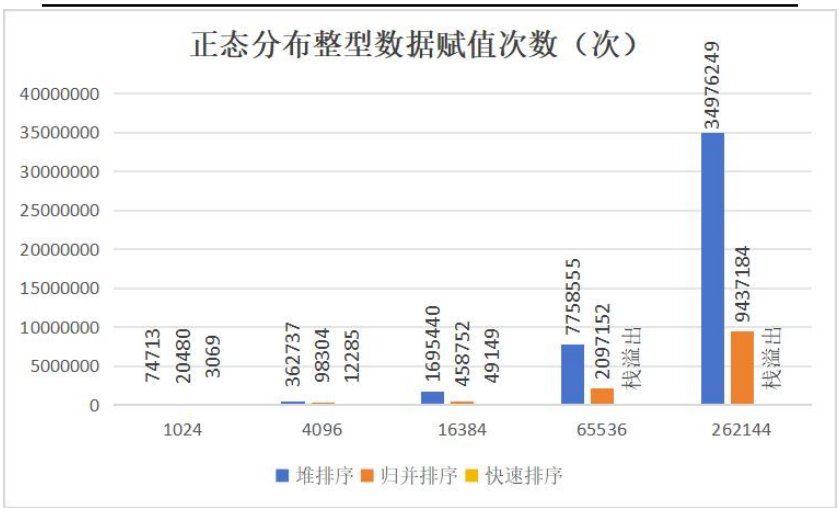


表 12.均匀分布整型数据赋值次数（次）

数据规模	堆排序	归并排序	快速排序
1024	63459	20480	4629
4096	316430	98304	18459
16384	1512167	458752	73761
65536	7014995	2097152	栈溢出
262144	31959587	9437184	栈溢出

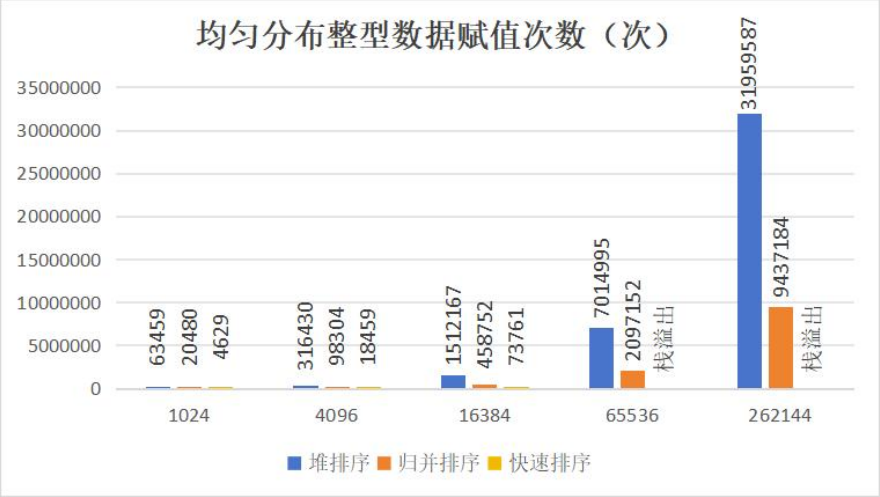


表 13.正序分布整型数据赋值次数（次）

数据规模	堆排序	归并排序	快速排序
1024	68096	20480	11043
4096	335863	98304	54816
16384	1586250	458752	267639
65536	7319664	2097152	1271172
262144	33210595	9437184	5858688

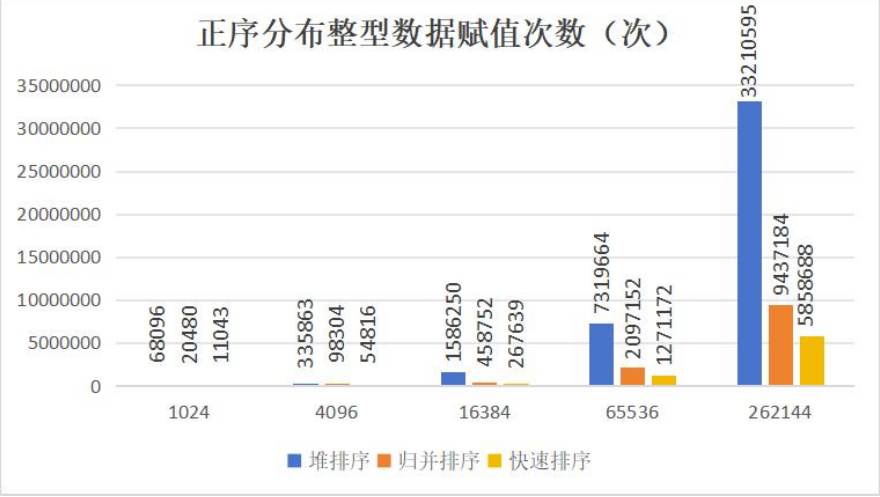


表 14.逆序分布整型数据赋值次数（次）

数据规模	堆排序	归并排序	快速排序
1024	68342	20480	10506
4096	332547	98304	53370
16384	1569911	458752	261738
65536	7241875	2097152	1243860
262144	32776205	9437184	5712171

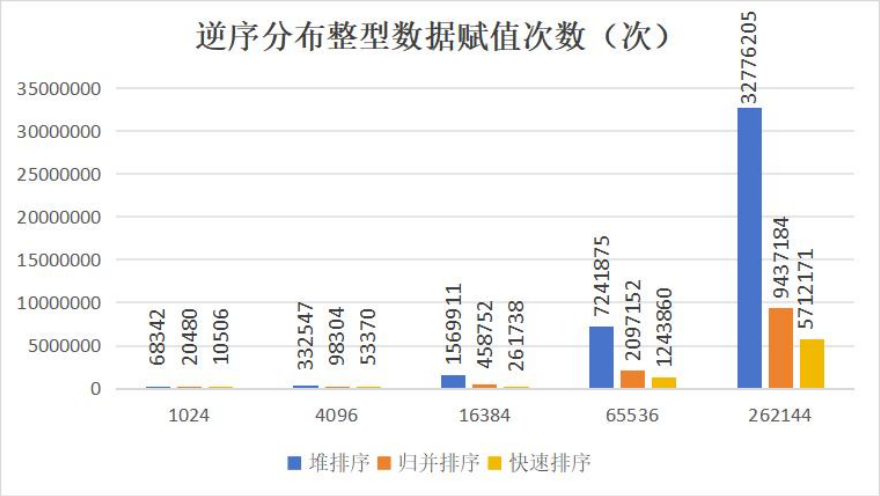


表 15.半顺序半均匀分布整型数据赋值次数（次）

数据规模	堆排序	归并排序	快速排序
1024	68647	20480	10785
4096	333967	98304	54996
16384	1587602	458752	265419
65536	7322350	2097152	1259076
262144	33223200	9437184	5871870

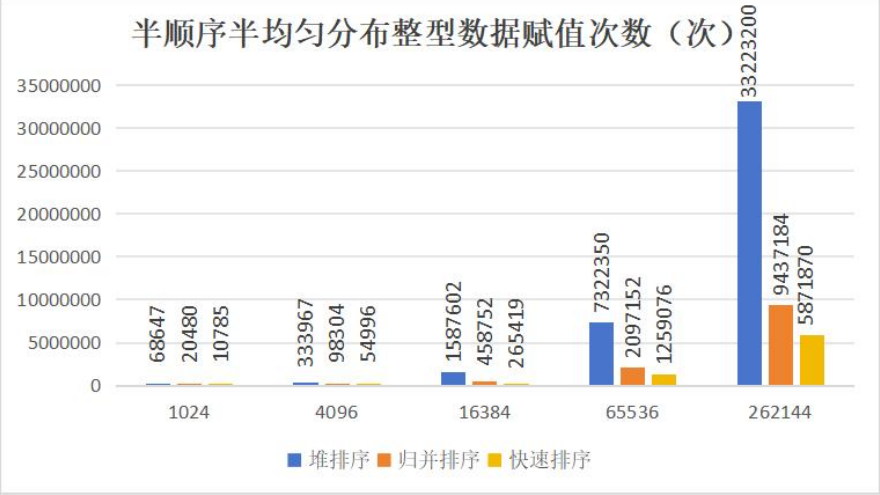


表 16.正态分布整型数据比较次数（次）

数据规模	堆排序	归并排序	快速排序
1024	35683	30720	1051644
4096	177131	147456	16789500
16384	838707	688128	268484604
65536	3875555	3145728	栈溢出
262144	17593191	14155776	栈溢出

表 17.均匀分布整型数据比较次数（次）

数据规模	堆排序	归并排序	快速排序
1024	28861	30720	355663
4096	148647	147456	5616973
16384	725447	688128	89576779
65536	3414427	3145728	栈溢出
262144	15713115	14155776	栈溢出

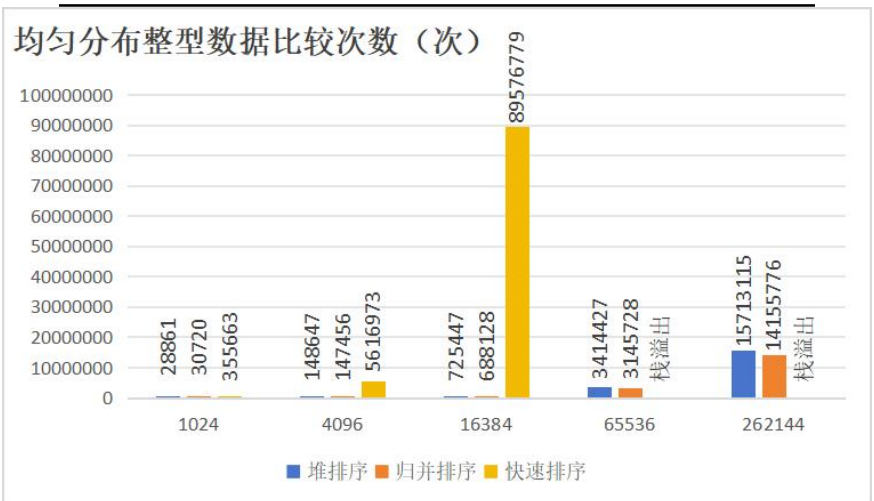


表 18.正序分布整型数据比较次数（次）

数据规模	堆排序	归并排序	快速排序
1024	31304	38380	14390
4096	159067	186078	78440
16384	765010	875892	272686
65536	3578232	4028094	1200659
262144	16405180	18209814	4799368

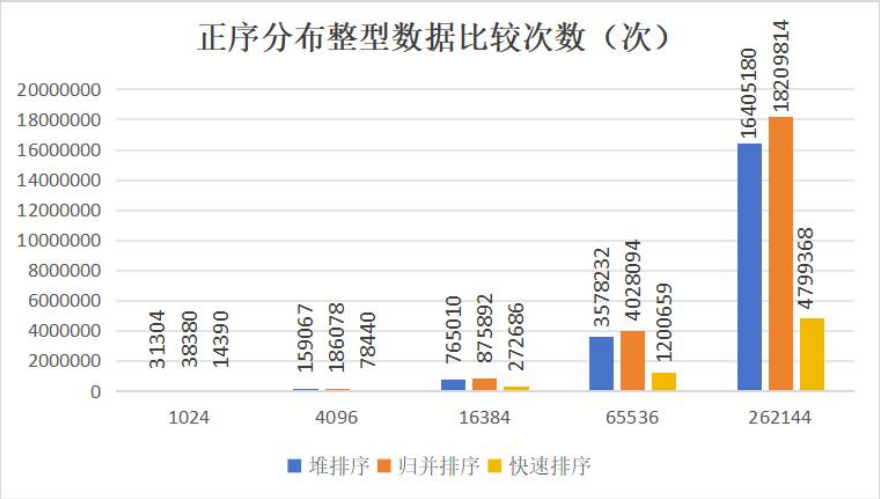


表 19.逆序分布整型数据比较次数（次）

数据规模	堆排序	归并排序	快速排序
1024	31455	38318	16833
4096	157231	186018	71373
16384	755630	874856	276526
65536	3533050	4020710	1160160
262144	16150486	18172934	5749447

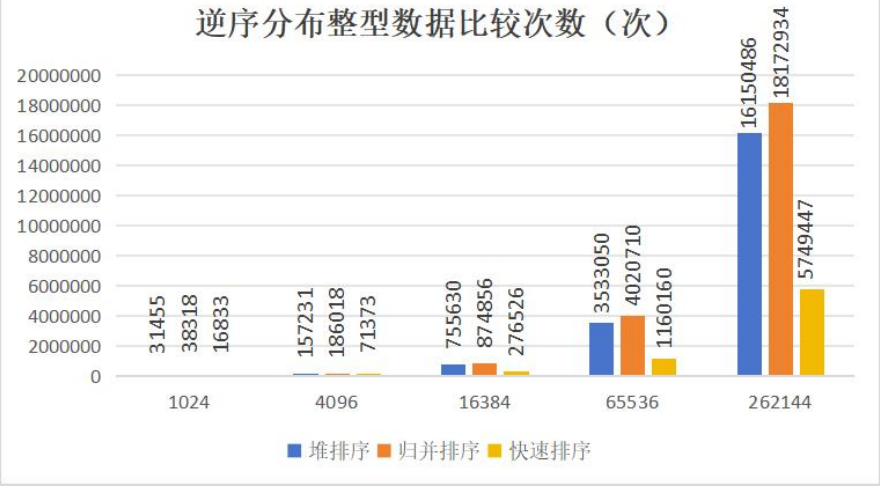


表 20.半顺序半均匀分布整型数据比较次数（次）

数据规模	堆排序	归并排序	快速排序
1024	31614	38400	16689
4096	157893	186170	64135
16384	765690	875916	292655
65536	3579268	4028044	1057660
262144	16407728	18209000	4986859