2020-2021 夏季学期
《计算机程序设计实训》课程论文

20121802 严昕宇 | 20122136 史诚鑫 | 20123272 刘桓菘 | 20123306 奚仲璞

(计算机工程与科学学院)

摘　要　本文主要介绍了冒泡排序、选择排序、插入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　程序设计实训；排序算法优化；字符串处理

Course Paper of “Programming Training” in 2020-2021 Summer Semester

20121802 Xinyu Yan | 20122136 Chengxin Shi | 20123272 Huansong Liu | 20123306 Zhongpu Xi

(School of Computer Engineering and Science)

Abstract　This article mainly introduces four sorting algorithms: Bubble Sorting, Selection Sorting, Insertion Sorting and Quick Sorting. The core code, time complexity, space complexity, stability of the sorting algorithm, the data configuration of the sorting algorithm and the respective optimization are also enclosed. Meanwhile, this article conducts a sorting study on C-string arrays of different storage methods, and analyzes the operation methods that can/cannot be performed in different storage methods. At last, this article gives the personal experience of course learning.

Key words　programming practical training; optimizing of sort algorithms; string manipulation

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。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。

C-字符串是 C/C++程序设计中常用的数据。然而理解和掌握 C-字符串的基本概念、处理方法却涉及到“容器”、数组、指针等概念。因此，C-字符串处理可以训练我们掌握数组与指针及其在函数中传递等程序设计能力。

2 论文内容

本文内容主要分为以下三个方面：

①四种未优化的排序算法的研究结论及代码说明；

②优化四种排序算法的研究结论及代码说明；

③C-字符串处理的研究结论明及代码说明。

随本文一并上交的文件如表 1 所示：

表 1. 上交的课程论文、源代码、程序文件和实验数据

序号	文件名	说明
1	2020-2021 夏季学期《计算机程序设计实训》课程论文.docx	本文件，作为课程小论文。
2	2020-2021 夏季学期《计算机程序设计实训》课程论文.pdf	导出为 PDF 格式的本文件，以方便查看此课程小论文，并防止排版错位。
3	Sorts-VS.zip	源代码。包含四种（冒泡、插入、选择、快速）未优化的排序算法； 多种构型测试数据（正序、逆序、均匀分布、正态分布、结构体）生成方法； 运行时间测试方法；C-字符串处理代码等。
4	Sorts-VS-Optimized.zip	源代码。包含四种（冒泡、选择、快速）优化后的排序算法；多种构型测试数据（正序、逆序、均匀分布、正态分布、结构体）生成方法；运行时间测试方法等。
5	Results.zip	实验数据文件，内有： ①给定的可执行批处理文件，用于统计排序算法运行时间； ②实验得到的部分原始数据； ③整理并处理过的数据 Results（未优化）.xlsx、Results（优化比较）.xlsx

3 排序算法的复杂度与稳定性

3.1 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，$\frac{T(n)}{f(n)}$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 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。

3.3 稳定性

假设某一序列的关键字是a_i(i = 1,2,3,⋯⋯,n)，且存在$a_s = a_t(1 \le s < t \le n)$ ，如果进过排序后，a_s依然在a_t左侧（前方），则称该排序方法是稳定的。反之，若a_s排列在a_t右侧（后方），则称该排序方法是不稳定的。

为了验证排序算法的稳定性，可在<Score.h>头文件中声明了一个稳定性检查函数，其定义如下

```
void SteadyCheck(const Score* a, int size)
{
    int i;
    int flag = 1;                                //用于表记元素相对位置是否有变化的标志
    for (i = 0; i < size - 1; i++)               // 共进行 size-1 轮
    {
        int Left_ID, Right_ID;
        sscanf(data[i].Id, "%d", &Left_ID);
        sscanf(data[i + 1].Id, "%d", &Right_ID);
        if (data[i].Total == data[i + 1].Total && Left_ID > Right_ID)
        {
            flag = 0;
            break;
        }
        if (flag)
            printf("稳定\n");
        else
            printf("不稳定\n");
    }
}
```

3.4 各种排序方法的性能比较

表 2. 各种排序方法的性能比较

排序方法	最好时间	平均时间	最坏时间	稳定性
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	稳定
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	不稳定
快速排序	$O(n\,logn)$	$O(n\,logn)$	$O(n^2)$	不稳定

4 未优化的排序算法及其分析

排序算法，即通过特定的算法因式将一组或多组数据按照既定模式进行重新排序。这种新序列遵循着一定的规则，体现出一定的规律，因此，经处理后的数据便于筛选和计算，大大提高了计算效率。

但随着数据量的大幅提升,常规的排序方式在处理大数据的时候效率并不尽如人意。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。进而需要对这些排序算法进行优化，以达到更加高效的数据处理。

4.1 冒泡排序

冒泡排序是最基础的排序方式之一，其基本思想是通过比较相邻两个元素的大小，将逆序的元素进行交换来完成排序。冒泡排序采用多趟比较来完成，如果相邻两个元素的大小为逆序关系，则予以交换，在此过程中，大的元素“下沉”，小的元素“上浮”，直到最大的元素“下沉”到最末尾的位置，这样便完成了第一趟“冒泡”，之后继续第二趟“冒泡”，将第二大的元素“下沉”到倒数第二个位置……继续此过程直到排序完毕。整个过程类似于水泡上升，因此形象地称之为冒泡排序。

冒泡排序的时间复杂度为$O(n^2)$，空间复杂度为$O(1)$，属于稳定排序。

冒泡排序的核心代码如下

```
void BubbleSort(int *a, int size)            // 冒泡排序
{
    int temp;                          // 定义一个局部变量，数据类型与形式数据类型相同
    int i, j;
    for (i = 1; i < size; i++)                // 共进行 size-1 轮比较和交换
    {
        for (j = 0; j < size - i; j++)
        {
            if (a[j] > a[j + 1])              // 相邻元素之间比较，必要时
            {
                temp = a[j];                  // 交换 a[j] 与 a[j+1]
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

4.2 选择排序

选择排序（selection sort），其算法思想为：首先在未排序序列中找到最小元素，将其交换到排序序列的起始位置，再从剩余未排序元素中继续找出最小元素，将其交换到已排序序列的末尾，重复此过程，直到所有元素均排序完毕。值得注意，选择排序和冒泡排序的算法思想看起来类似，但实质上是不同的。

选择排序的时间复杂度为$O(n^2)$，空间复杂度为$O(1)$，如果使用原地交换的方法，属于不稳定排序，如果借助额外的存储空间可使算法成为稳定的。

选择排序的核心代码如下

```
void SelectSort(int* a, int size)        // 选择排序
{
    int temp;
    int i, j, k = 0;
    for (i = 1; i < size; i++)                // 循环 size-1 次
    {
        for (j = i; j < size; j++)
            if (a[j] < a[k])
                k = j;                        // 找出当前范围内"最小"元素的下标
        if (k != i - 1)                       // 若"最小"元素不是 a[i-1]，则交换之
        {
            temp = a[k]; a[k] = a[i - 1]; a[i - 1] = temp;
        }
        k = i;
    }
}
```

4.3 快速排序

快速排序（quicksort）是对冒泡排序的一种改进，由查尔斯·安东尼·理查德·霍尔在 1962 年提出的一种划分交换排序发展而来，它采用了一种分治的策略，通常称其为分治法（divide-and-conquer method）。

算法基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的数据比另外一部分的数据都要小，然后再按此方法对这两部分数据分别进行快速排序。整个排序过程可以递归进行，以此达到整个数据有序的目的。

排序的第一步是要确定一个基准值（pivot），为了简便，一般选择位于区间中心的元素作为基准值，然后以此基准值将区间划分为两部分进行排序，之后递归调用。但编写一个正确的快速排序并非想象中的那么容易，需要考虑许多边界情形。

快速排序的时间复杂度为$O(nlog_n)$，空间复杂度为$O(log_n)$，属于不稳定排序。

快速排序的核心代码如下

```
void QuickSort(int* a, int size)             // 快速排序
{
    int pivot, temp;
    int left = 0, right = size - 1;              // 下标（整数）
    if (size <= 1) return;
    pivot = a[right];                            // 选择最后一个值为分界值
    do
    {
        while (left < right && a[left] <= pivot)  left++;
        // 此处 "<=" 是让与分界值相等的元素暂时留在原地
    //接下页代码
```

```
    //接上页代码
        while (left < right && a[right] >= pivot) right--;
        // 此处 ">=" 是让与分界值相等的元素暂时留在原地
        if (left < right)
        {
            temp = a[left]; a[left] = a[right]; a[right] = temp;
        }
    } while (left < right);
    a[size - 1] = a[left]; a[left] = pivot;     // 找到分界点 left
    QuickSort(a, left);                         // 递归调用(左侧部分)
    QuickSort(a + left + 1, size - left - 1);   // 递归调用(右侧部分)
}
```

同时，在C++中，也可以使用Algorithm头文件中的qsort来调用快速排序的功能。qsort的声明为：**void** qsort (**void*** base, size_t num, size_t size, **int** (*compar)(**const void***,**const void***));

其中第一个参数为待排序数组起始地址，第二个参数为数组中待排序元素数量，第三个参数为单个数组元素占用空间的大小，第四个参数为指向比较函数的指针。比较函数以数组中的两个元素a和b作为参数，当函数返回大于 0 的值时，指示 qsort 在排序中将a放在b之后，即$a>b$；如果返回值小于 0，则将a放在b之前，即$a<b$；返回 0 值表示a和b相等。

4.4 插入排序

插入排序（insertion sort）的算法思想为：通过持续构建有序序列来达到整个序列有序的目的，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上通常采用本地排序，因此在从后向前扫描过程中，需要反复地把已排序元素逐步向后挪位，为最新元素提供插入空间。

其时间复杂度为$O(n^2)$，空间复杂度为$O(1)$，属于稳定排序。

插入排序的核心代码如下

```
void InsertionSort(int data[], int n)   // 插入排序
{
    for (int i = 1; i < n; i++)
    {
        // 查找插入位置。若未找到，则将有序元素向后移动一个位置。
        int temp = data[i], j = i - 1;
        while (j >= 0 && data[j] > temp)
        {
            data[j + 1] = data[j];
            j--;
        }
        // 将元素写入找到的位置。
        data[j + 1] = temp;
    }
}
```

4.5 未优化算法的效率分析

4.5.1 排序算法执行时间比较——以整型数据为例

对于不同规模均匀分布的整形数据，不同算法的运行时间如下图

图 1. 各排序算法在面对不同规模正态分布整形数据时的运行时间

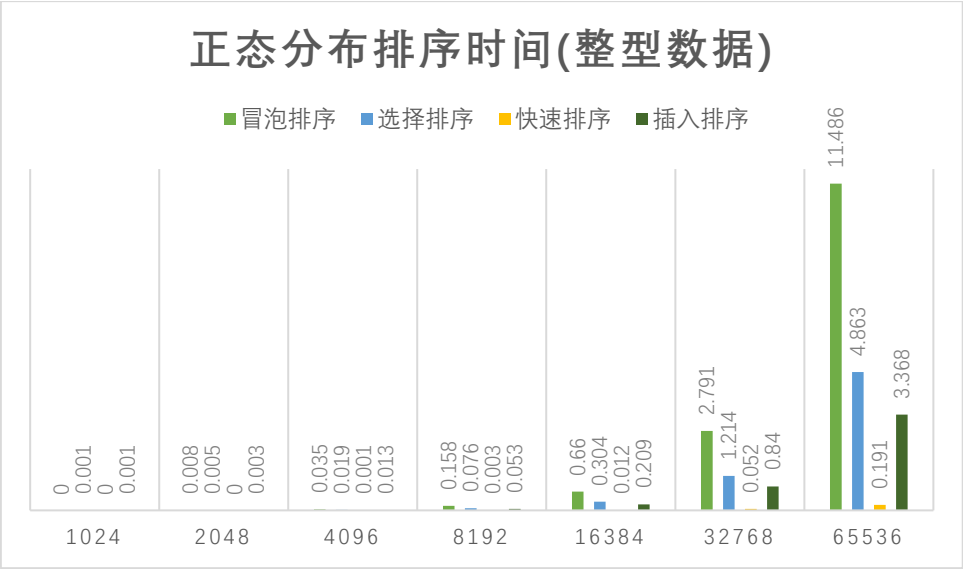


图 2. 各排序算法在面对不同规模均匀分布整型数据时的运行时间

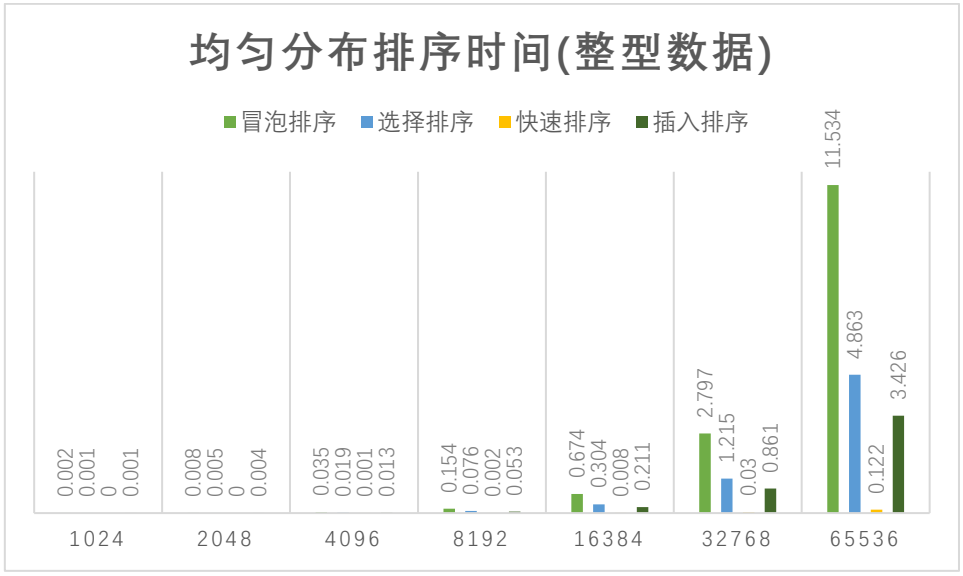


图 3. 各排序算法在面对不同规模完全顺序整型数据时的运行时间

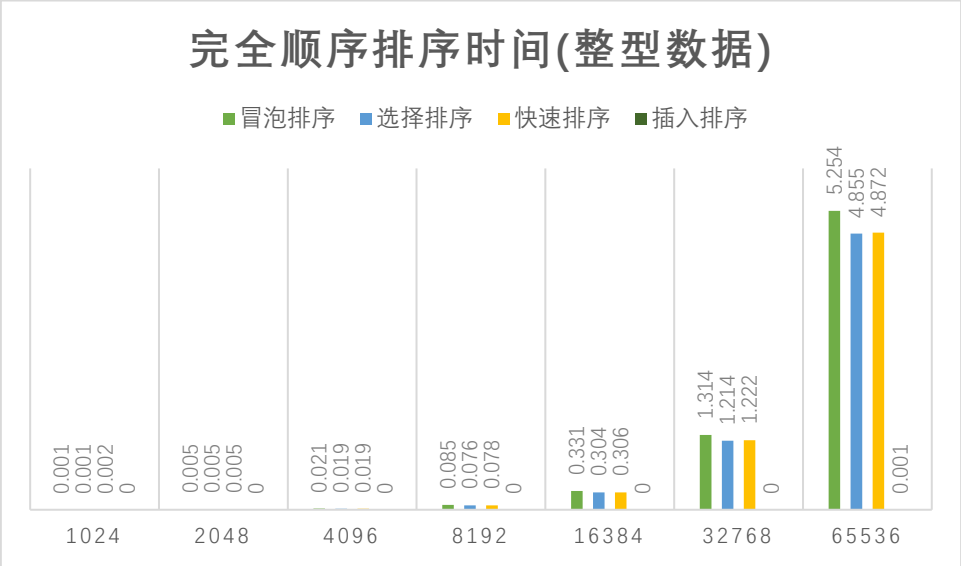
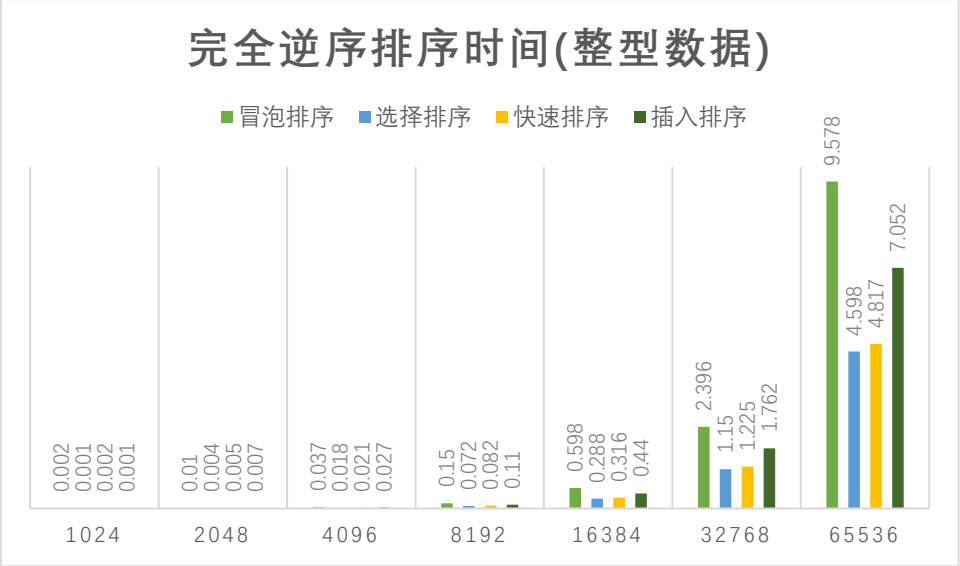


图 4. 各排序算法在面对不同规模完全逆序整型数据时的运行时间



首先对于以上四种不同排列的数据，各种算法的耗时都随着数据规模的增大而增大。

（分析数据可知：数据规模增大两倍，耗时近似增加四倍）

在面对不同规模**正态分布**整形数据时所耗时间：快速排序<插入排序<选择排序<冒泡排序。

在面对不同规模**均匀分布**整形数据时所耗时间：快速排序<插入排序<选择排序<冒泡排序。

在面对不同规模**完全顺序**整形数据时所耗时间：插入排序<选择排序<快速排序<冒泡排序。

在面对不同规模**完全逆序**整形数据时所耗时间：选择排序<快速排序<插入排序<冒泡排序。

综合来看，快速排序的速度较快，而冒泡排序速度最慢

4.5.2 排序算法操作次数比较

详细数据请见**附页 A 未优化排序算法的操作次数**

由于整型与浮点型测试中规律类似，以下只分析浮点型数据：

对于附页 A 中四种不同排列的数据，各种算法的比较次数和赋值次数都随着数据规模的增大而增大（除非有次数为 0 的情况）。

对于处理正态分布的浮点型数据时的操作次数：

①比较次数：冒泡=插入>快速排序>选择排序。②赋值次数：冒泡>快速>>选择>插入

对于处理均匀分步的浮点型数据时的操作次数：

①比较次数：冒泡=插入>快速排序>选择排序。②赋值次数：冒泡>快速>>选择>插入

对于处理完全顺序的浮点型数据时的操作次数：

①比较次数：冒泡=插入=选择>>快速排序。　②赋值次数：选择>快速>>冒泡=插入（为 0 次）

对于处理完全逆序的浮点型数据时的操作次数：

①比较次数：快速排序>冒泡=插入=选择。　②赋值次数：冒泡>快速>>选择>插入

5 排序算法的优化及其分析

5.1 双向冒泡排序

双向冒泡排序（bidirectional bubble sort）在冒泡排序的基础上进行了少量优化，其基本思想并未改变，只不过在每次“冒泡”进行到最后时，不是从头开始“冒泡”，而是从后往前将最小的元素“冒泡”到其正确位置，这样可以最大程度减少循环比较的次数，得到常数项的优化，但是总的时间复杂度仍然是$O(n^2)$。

双向冒泡排序的核心代码如下

```
void bidirectionalBubbleSort(int *a, int n)
{
    int left = 0, right = n - 1, shift, temp;
    while(left < right)
    {
        // 将较大的值移到末尾
        for(int i = left; i < right; i++)
        {
            if(a[i] > a[i + 1])
            {
                temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                shift = i;
            }
            right = shift;
        }
        // 将较小的值移到开头
        for(int i = right - 1; i >= left; i--)
        {
        //接下页代码
```

```
        //接上页代码
            if(a[i] > a[i + 1])
            {
                temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                shift = i + 1;
            }
            left = shift;
        }
    }
}
```

5.2优化后的选择排序算法

在分析其排序思想时，发现在其每一次遍历是仅仅找出一个最大的数或者最小的数进行交换，这样显然不是很高效的算法。进而设想，能否在一次遍历的过程中，同时找到最大值与最小值，并对其与收尾元素进行交换。

```
int left = 0;
    int right = size-1;
    while (left < right)
    {
        int min = left;
        int max = right;
        for (int i = left; i <= right; i++)
        {
            if (a[i] < a[min])
                min = i;
            if(a[i] > a[max])
                max = i;
        }
        swap(a[max], a[right]);
        //考虑修正的情况，最大值在最小位置，最小值在最大位置。
        if (min == right)
        {
            min = max;
        }
        swap(a[min], a[left]);
        left++;
        right--;
    }
}
```

根据这一想法，可以得到了优化后的选择排序算法，其核心代码如下

此段代码中，在一次遍历过程中找到了最大值与最小值，并将其分别与首位元素进行了交换。同时也对最大值在最小位置，而最小值在最大位置的极端情况进行了修正优化。

5.3快速排序的优化

5.3.1 三数取中的快速排序

5.3.1.1 设计原理

在快速排序的过程中，每一次要取一个元素作为枢纽值，以这个数字来将序列划分为两部分。在原代码中，使用了数组末项（a[right])作为切分点（中心点，pivot），并通过从左向右(left++)与从右向左(right--)的方向搜索并交换特定元素，使得切分点一侧的元素值均小于等于该切分点，另一侧则大于等于切分点。在实际应用中，发现快速排序在应对原先已有部分或完全排序的数组时效率不佳，对于完全顺序时，原始的快速排序速度甚至可能慢于冒泡排序与选择排序。这是因为对于完全顺序数组，每次选取切分点为数组的最大值（最小值），每次只能将数组分为 1 个和其他所有元素两个数组，为最糟状况（$O(n^2)$）。而当切分点取到数组中间值时，为最佳情况。故对此弊端，对此快速排序进行优化。考虑到找到数组元素的中间值非常困难，故选择选取数组元素首项，位置中间值和末项并取中间值来增加将数组平均分配的概率，以提高快速排序速度。在代码数据中，使用字符'1'标记了实现三向切分的算法，记为"Qsort_1"。

5.3.1.2 算法实现原理

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[right],a[mid],a[left]中选取中间值作为 pivot，通过大小比较与交换使得 a[mid]<a[left]=pivot<a[right]。

5.3.1.3. 三数取中预处理处理的核心代码

```
void QsortInt_1(int* a, int size) {
    int pivot, temp;
    int left = 0, right = size - 1;                 // 下标（整数）
    int mid = size / 2;
    if (size <= 1) return;
    if (a[right] < a[left])
    {
        temp = a[left]; a[left] = a[right]; a[right] = temp;
    }
    if (a[mid] < a[right])
    {
        temp = a[right]; a[right] = a[mid]; a[mid] = temp;
    }
    if (a[mid] < a[left])
    {
        temp = a[mid]; a[mid] = a[left]; a[left] = temp;
    }
    pivot = a[right];
    // 选择最后一个值为分界值

    //接下页代码
```

```
    //接上页代码
    do
    {
        while (left < right && a[left] <= pivot) {
            left++;
        }// 此处 "<=" 是让与分界值相等的元素暂时留在原地
        while (left < right && a[right] >= pivot) {
            right--;
        }// 此处 ">=" 是让与分界值相等的元素暂时留在原地
        if (left < right)
        {
            temp = a[left]; a[left] = a[right]; a[right] = temp;
        }
    } while (left < right);
    a[size - 1] = a[left]; a[left] = pivot;
    // 找到分界点 left
    QsortInt_1(a, left);                                    // 递归调用(左侧部分)
    QsortInt_1(a + left + 1, size - left - 1);              // 递归调用(右侧部分)
}
```

5.3.2 嵌入插入排序的快速排序

5.3.2.1 设计原理

在数据量较小时，快速排序速度慢于插入排序。故当快速排序递归产生的数组项速足够小时，调用插入排序以增加运算速度。在代码数据中，使用字符'2'标记了实现嵌入插入排序的算法，记为“Qsort_2”。

5.3.2.2 算法实现原理

通过查阅文献，发现当数组项数$n <= 7$时，快速排序的速度便慢于插入排序，故在$n <= 7$，时，调用了插入排序。

5.3.2.3 嵌入插入排序的快速排序的核心代码

```
void QsortInt_2(int* a, int size)                   // 快速排序
{
    int pivot, temp;
    int left = 0, right = size - 1;                 // 下标（整数）
    int mid = size / 2;
    if (size <= 1) return;
    else if (size <= 7)
    {
        InsertInt(a, size);                         //调用已经写好的插入排序
        return;
    }


    //接下页代码
```

```
    //接上页代码
    if (a[right] < a[left])
    {
        temp = a[left]; a[left] = a[right]; a[right] = temp;
    }
    if (a[mid] < a[right])
    {
        temp = a[right]; a[right] = a[mid]; a[mid] = temp;
    }
    if (a[mid] < a[left])
    {
        temp = a[mid]; a[mid] = a[left]; a[left] = temp;
    }
    pivot = a[right];
    // 选择最后一个值为分界值
    // 选择最后一个值为分界值
    do
    {
        while (left < right && a[left] <= pivot)
        {
            left++;
        }// 此处 "<=" 是让与分界值相等的元素暂时留在原地
        while (left < right && a[right] >= pivot)
        {
            right--;

        }// 此处 ">=" 是让与分界值相等的元素暂时留在原地
        if (left < right)
        {
            temp = a[left]; a[left] = a[right]; a[right] = temp;
        }
    } while (left < right);
    a[size - 1] = a[left]; a[left] = pivot;
    // 找到分界点 left
    QsortInt_2(a, left);                                    // 递归调用(左侧部分)
    QsortInt_2(a + left + 1, size - left - 1);              // 递归调用(右侧部分)
}
```

5.3.1 三向切分的快速排序

5.3.2.1 设计原理

虽然通过现有优化，发现快速排序的速度已经有了显著的提高，特别是对于完全正序和逆序数组，但是，在处理具有重复数字的正态分布数组，仍有较大的改进空间。在一次循环中，与 pivot 相同的数组元素会留在原地不动，而在下一次循环中，才作为最大（最小）值移动到 pivot 旁边，这将多次调用该数组元素，降低运算效率，故选择按照大于，等于，小于 pivot 切分数组。照此切分的方法简称为三向切分。在代码数据中，使用字符'3'标记了实现三向切分的算法，记为"Qsort_3".

5.3.2.2 算法实现原理

三向切分在原有快速排序算法上保留左指针 left 与右指针 right，同时新增了指针 i。但值得注意的是，这里指针 left 为小于 pivot 的指针，right 为大于 pivot 的指针，i 为等于 pivot 的指针。

在切分点 pivot 的寻找上，采用上述的三数取中的方法，a[left]为中间值等于 pivot。故 i=left+1。

切分过程：

① 从 i 开始，寻找第一个大于 pivot 的数，若 a[i]小于 pivot，则将 a[left]与 a[i]交换，并右移 left 和 i 指针（i++，left++）

② [i]等于 pivot，则右移 i 指针（i++），不对 left 指针进行操作。

③ 从 right 开始，寻找第一个小于等于 pivot 的值。

④ [i]与 a[right]交换，使得 a[i]小于等于 pivot，a[right]大于 pivot。

5.3.2.3. 三向切分的快速排序的核心代码

```
void QsortInt_3(int* a, int size)                    // 三向切分快速排序
{
    int pivot, temp;
    int left = 0, right = size - 1;
    // 下标（整数）
    int mid = size / 2;
    if (size <= 1) return;
    else if (size <= 7)
    {
        InsertInt(a, size);
        return;
    }
    if (a[right] < a[left])
    {
        temp = a[left]; a[left] = a[right]; a[right] = temp;
    }
    if (a[mid] > a[right])
    {
        temp = a[right]; a[right] = a[mid]; a[mid] = temp;
    }

    //接下页代码
```

```
    //接上页代码
    if (a[mid] > a[left])
    {
        temp = a[mid]; a[mid] = a[left]; a[left] = temp;
    }
    pivot = a[left];
    int i = 1;
    // 选择最后一个值为分界值
    do
    {
        while (i < right && a[i] <= pivot)
        {
            if (a[i] == pivot) {

                i++;
            }
            else
           {
                a[left] = a[i]; a[i] = pivot;
                i++;
                left++;
            }

        }
        while (i < right && a[right] > pivot)
        {
            right--;
        }
        if (i < right)
        {
            temp = a[i]; a[i] = a[right]; a[right] = temp;
        }
    } while (i < right);
    // 找到分界点 left
    QsortInt_3(a, left);                    // 递归调用(左侧部分)
    QsortInt_3(a + i, size - i);            // 递归调用(右侧部分)
}
```

5.4 插入排序的二分优化

直接插入排序是对数组进行升序排序的时候，未排序数列中的元素逐一与已排序数组中的元素进行逐一比较，但这样线性查找的时间度为$O(n)$。通过思考后，发现既然要插入的数组已经是有序的了，但就可以采用二分法进行排序，这样，时间复杂度就从原先的$O(n)$降为了$O(log\ n)$。

以下是二分优化插入排序的核心代码

```
void I_InsertSortpro(int *a,int size)     //插入排序的二分优化
{
    int i,j,left,mid,right,temp;
    for(i=1;i<size;i++)
    {
        temp=a[i];
        left=0;
        right=i-1;

        while(left<=right)                    //二分搜索以便找到插入点
       {
            mid=(left+right)/2;
            if(a[i]<a[mid])
            {            //左侧
                right=mid-1;
            }
            else
            {           //右侧
                left=mid+1;
            }
        }
        for(j=i-1;j>=left;j--)
        {
            a[j+1]=a[j];
        }
        a[left]=temp;
    }
 }
```

5.5 已优化算法的效率分析

5.5.1 冒泡排序与双向冒泡排序算法执行时间比较——以整型数据为例

对于不同规模均匀分布的整形数据，冒泡排序算法以及其优化版本——双向冒泡排序算法的运行时间如下图

图 5. 冒泡排序与双向冒泡排序算法在面对不同规模正态分布整形数据时的运行时间

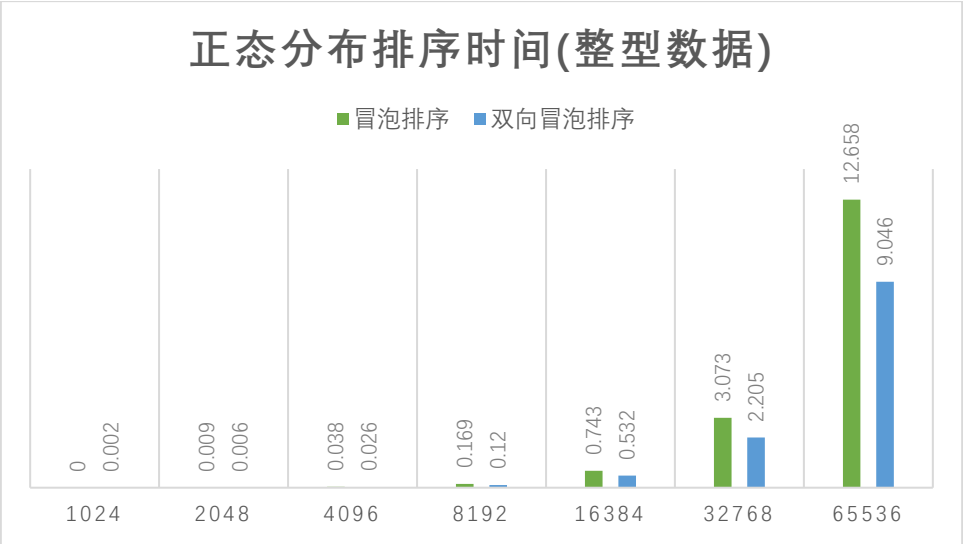


图 6. 冒泡排序与双向冒泡排序算法在面对不同规模均匀分布整形数据时的运行时间

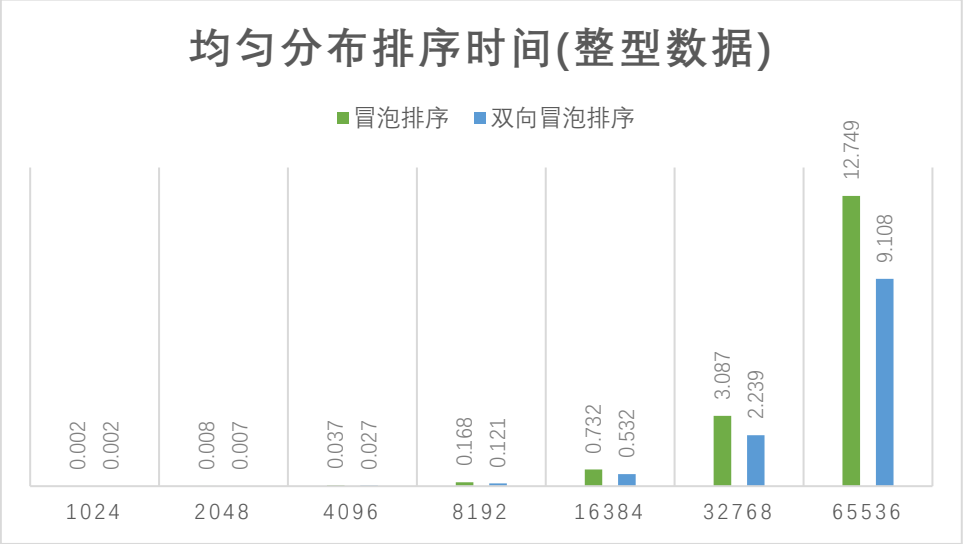


图 7. 冒泡排序与双向冒泡排序算法在面对不同规模完全顺序分布整形数据时的运行时间

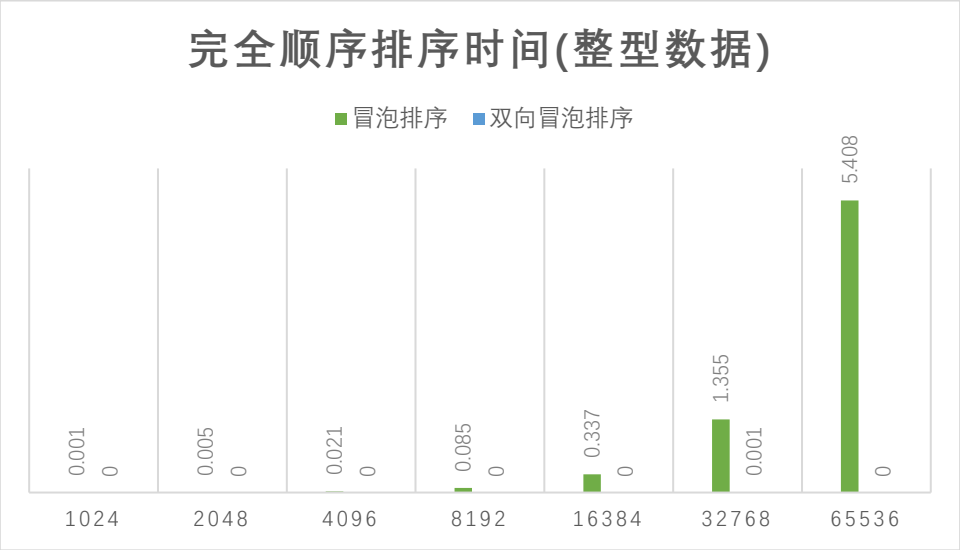
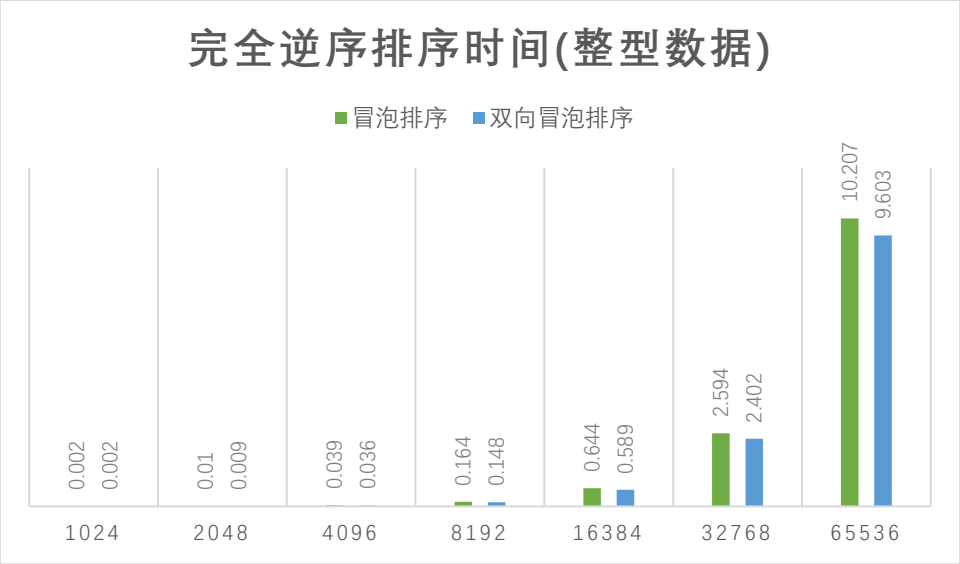


图 8. 冒泡排序与双向冒泡排序算法在面对不同规模完全逆序分布整形数据时的运行时间



对于不同规模正态分布和均匀分布数据的处理分析

①在数据量 16384 以下，双向冒泡排序和冒泡排序所用时间相差无几；

②当数据量达到 16384 且逐渐变大，双向冒泡排序和冒泡排序的效率差异会逐渐显著，双向排序算法会有更优的效率（效率=数据量/处理时间）。

对于不同规模完全正序分布整型数据的处理分析

①双向冒泡排序时间始终为“0”；

②冒泡排序时间与数据量呈正相关。

对于不同规模完全逆序分布整型数据的处理分析

①在数据量 8192 以下，双向冒泡排序和冒泡排序所用时间相差无几；

②当数据量达到 16384 且逐渐变大，双向冒泡排序和冒泡排序的效率差异会逐渐显著，双向排序算法会有更优的效率（效率=数据量/处理时间）。

5.5.2 冒泡排序与双向冒泡排序算法排序算法操作次数比较

详细数据请见**附录 B 未优化与已优化的冒泡排序算法的比较**

总操作次数=比较次数+赋值次数

面对正态分布、均匀分布、完全正序、完全逆序四种整型数据

①赋值次数：冒泡排序和双向冒泡始终一致；

②比较次数：冒泡排序始终大于双向冒泡排序且随着数据量增大两者在该参数上差异越来越明显；

③总操作次数：冒泡排序>双向冒泡；

④特殊情况：面对完全正序整型数据，冒泡排序和双向冒泡排序的赋值次数皆为“0”。

5.5.3 快速排序算法与其优化版本算法的执行时间比较——以整型数据为例

对于不同规模均匀分布的整形数据，快速排序算法以及其优化版本的排序算法的运行时间如下图

图 9. 快速排序算法与其优化版本算法在面对不同规模完全逆序分布整形数据时的运行时间

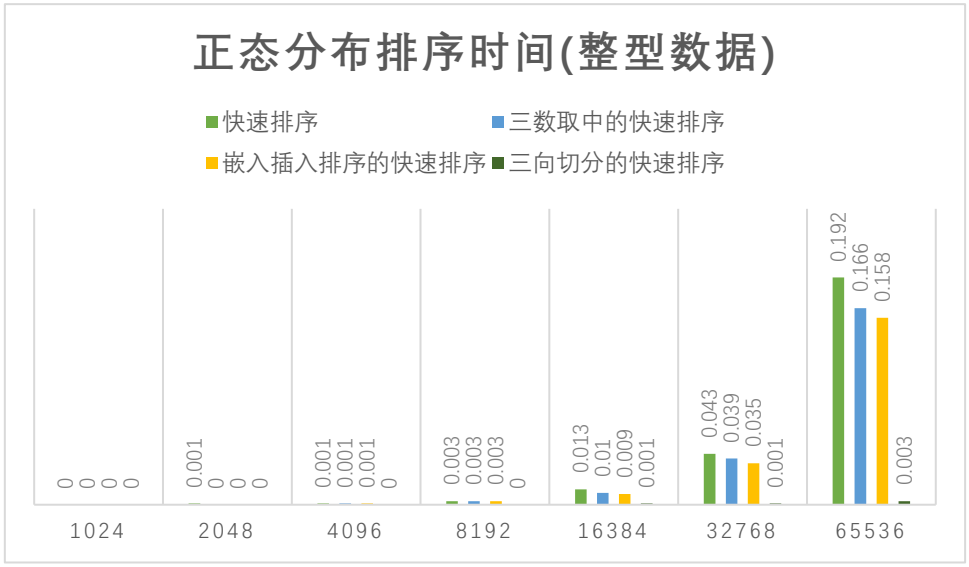


图 10. 快速排序算法与其优化版本算法在面对不同规模完全逆序分布整形数据时的运行时间

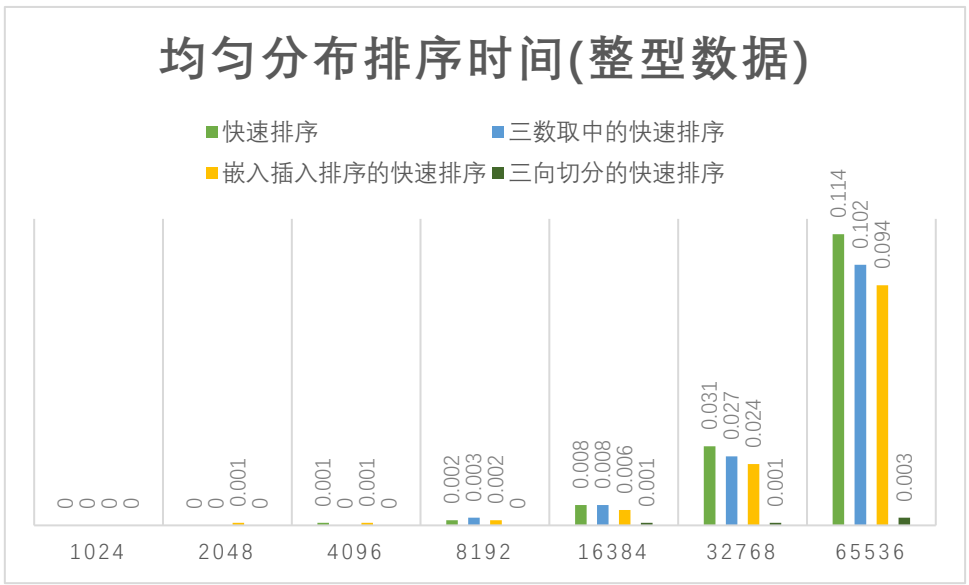


图 11. 快速排序算法与其优化版本算法在面对不同规模完全逆序分布整形数据时的运行时间

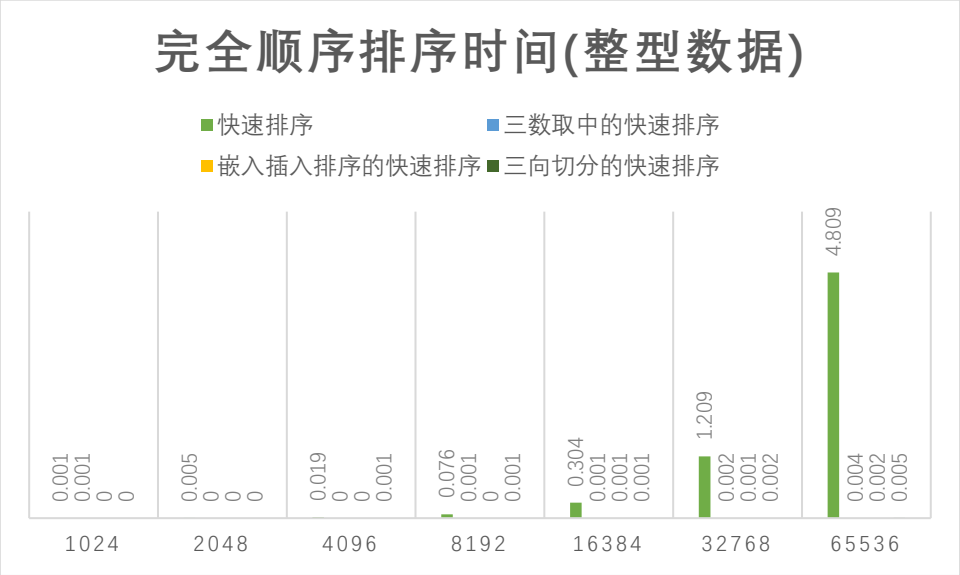
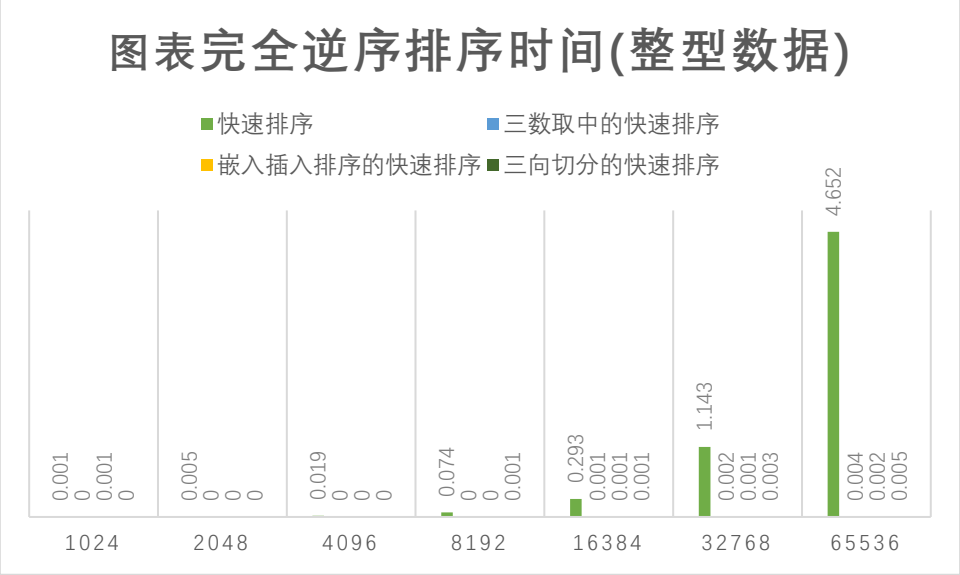


图 12. 快速排序算法与其优化版本算法在面对不同规模完全逆序分布整形数据时的运行时间



面对不同规模正态和均匀分布数据

①数据量 8192 以下，四种排序方式所用时间相差无几，且都趋于 0；

②数据量 16384 及以上，处理时间：快排>三数取中>插入型快排>三向切分，且随着数据量增大不同算法两两间差异越来越大；

③三向切分处理时间远小于其它算法。

面对不同规模完全正序和逆序分布数据

①数据量 8192 以下，所有算法处理时间相差无几且都趋于或等于 0；

②数据量 16384 以上，快排原始算法处理时间大于所有快排优化算法；且随着数据量增大，所有优化算法处理时间会有微小增加但仍趋于 0，而原始算法处理时间会有显著增加。

5.5.4 快速排序算法与其优化版本算法操作次数比较

详细数据请见**附录 C 不同优化思路的快速排序算法的比较**

总操作次数=比较次数+赋值次数

①快速排序（原始）与三向取中的快速排序（优化）

面对正态分布的整型数据

处理数据 1024，比较次数：原始 <优化，赋值次数：原始<优化，总操作数：原始<优化；

处理 2048 及以上，比较次数：原始>优化，赋值次数：原始<优化，总操作数：原始>优化。

面对完全正序整型数据

赋值次数：原始与优化始终一致，比较次数：原始>>优化，总操作数：原始>>优化；

三数取中的快速排序（优化）与嵌入插入排序+三数取中的快速排序（二级优化）。

面对完全正序整型数据

比较次数：优化>二级优化，赋值次数：优化<二级优化，总操作次数：优化>二级优化

面对正态分布整型数据

比较次数：优化>二级优化，复制次数：优化>二级优化，总操作次数：优化>二级优化

②三数取中的+嵌入插入排序的快速排序（二级优化）与三数取中的+嵌入插入排序的快速排序+三向切分的快速排序（三级优化）

面对正态分布整型数据

比较次数：二级优化>三级优化，复制次数：二级优化>三级优化，总操作次数：二级优化>三级优化

面对完全正序整型数据

比较次数：二级优化<三级优化，复制次数：二级优化<三级优化，总操作次数：二级优化<三级优化

6 C-字符串处理

6.1 C-字符串处理的起点

要研究 C-字符串的处理函数并进行操作，首先就一定要理解字符串在 C 语言中的存储方式与引用方式。字符串一般是按照字符数组的形式存放在内存空间中，在 C 语言中，字符串通常使用串首存储位置的字符指针进行表示，字符串的末尾以'\0'作为串结束符，字符串的内容在内存中则是连续存放。在读取字符串内容时，指针从串的起始位置开始逐位向后移动并读取，直到读取到串结束符为止。在修改串内容时，需要保证存放结果的字符指针后面有足够多的空闲内存空间，保障在写入字符串内容时不会发生越界错误。字符串在 C 语言里有两种定义方式，即用指针指向字符串和用数组储存字符串。如以下所示代码。

```
char *str = "Hello world!";        //一个给定的字符串是字符串常量
```

对于以上代码，str 是一个字符指针。由于在程序设计时 str 指向的是内存的常量区，在运行时这个字符串（“Hello world!”）将会被储存在内存的常量池中，即该字符串地址固定，不可以被程序修改。即不可以直接对 str 指向的字符串做修改（但可以访问）所以被称为字符串常量。

```
char str[21] = "Hello world!";   //储存有字符串的字符数组是字符串变量
```

以上代码中，定义了一个字符数组 str[21]，因为这里定义的是变量数组，str 作为数组的首地址便会指向内存的变量区（根据定义位置和定义方式的不同，可能储存在局部变量区，也可能储存在全局变量区）。这种方式定义的字符串是可以被程序进行修改的，所以被称为字符串变量。

以下代码为实例

```
char *str1 = "Hello world!";
//定义指向字符串常量的指针 ，注意 char *str1 != char *str1[]
str1[6] = 'W';
//违法操作，程序不可以对常量池中的常量进行修改,会报错退出
char str2[20] = "Hello world!";
//定义指向字符数组（字符串变量）的指针
str2[6] = 'W';
//合法操作，程序可以对变量区中的变量进行修改 printf("%s",str2);
//输出结果 hello World（修改生效了）
```

根据以上特点，在了解了字符串的存储方式后，就可以设计字符串处理和操作的函数。

6.2 C-字符串处理第一部分——内置字符串处理函数的重新设计

定义在 C 标准库<string.h>中的常见字符串处理函数有如下

表 3. 头文件<string.h>中定义的部分函数

序号	函数 & 描述	
1	字符串长度计算函数	int StrLen(const char *str);
2	字符串拷贝函数	char *StrCpy(char *a, const char *b);
3	字符串拼接函数	char *StrCat(char *a, const char *b);
4	字符串比较函数（字典序）	int *StrCmp(const char *a, const char *b);

下面是对上述函数的自定义

6.2.1 字符串长度函数

```
int StrLen(const char *str) //字符串长度计算函数
{
    int len;
    for(len = 0; str[len] != '\0'; len++);
   //这里需要对数组、指针、地址三者间关系有足够认识，指针+下标可看成数组,进行逐项统计
    return len;
}
```

6.2.2 字符串拷贝函数

```
char *StrCpy(char *a, const char *b)
//返回值为地址用指针函数，此处返回的是数组首地址，且该数组为字符型数组
{
    int i, len;
    len = StrLen(b);
    //直接获得 b 字符串的长度，减少下一步循环中的次数，降低时间复杂度
    for(i=0; i < len; i++) a[i] = b[i];
        a[len] = '\0';          //补充串尾结束符，使之成为完整的字符数组
    return a;//返回拷贝结果（保存对传入地址对应数据的改动）
}
```

6.2.3 字符串拼接函数

版本一

```
 char *StrCat(char *a, const char *b)
{
       char *p;
       p = a;
       while(*p != '\0') p++;      //寻找串尾结束符所对应地址，作为拼接首地址
       StrCpy(p, b);               //拷贝 b 数组首地址到上述位置
       return a;
}
```

版本二

```
char *StrCat1(char *a, const char *b)
{
    int t;
    t = StrLen(a);                 //寻找串尾结束符所对应地址，作为拼接首地址
    StrCpy(a+t, b);                //拷贝 b 数组首地址到上述位置
    return a;
}
```

6.2.4 字符串比较函数（字典序与长度序）

6.2.4.1 字符串比较函数（字典序）

```
int StrCmp(const char *a, const char *b)
{
    int add = 0;
    while((unsigned char) *(a + add) == *(b + add))
        add++;              // 比较两字符串相同位置对应字符是否相同
    if ((unsigned char) *(a + add) > (unsigned char) *(b + add))
        return 1;
    // unsigned char 强制字符转换获得地址对应字符，不可缺少，这里可以扩大表示整数的范围，防止某些中文字符对应 ascii 码过大导致数据溢出。字符与 Ascii 码一一对应，因此此处按字典序比较
    else if ((unsigned char) *(a + add) == (unsigned char) *(b + add))
        return 0;
    else if ((unsigned char) *(a + add) < (unsigned char) *(b + add))
    return -1;
}
```

6.2.4.1 字符串比较函数（长度序）

```
int StrCmp1(const char *a, const char *b)
{
    if (StrLen(a) > StrLen(b))  return 1;
    if (StrLen(a) < StrLen(b))  return -1;
    if (StrLen(a) == StrLen(b)) return 0;
}
```

6.2.5 字符串转换成长整型函数

此外，在 C 标准库<stdlib.h>中的库函数 long int atol (const char *str)，可以把参数 str 所指向的字符串转换为一个长整数（类型为 long int 型）。

atol()会扫描参数 str 字符串，跳过前面的空白字符（例如空格，tab 缩进等，可以通过 isspace()函数来检测），直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。

atol()函数返回转换后的长整型数。如果 str 不能转换成 long 或者 str 为空字符串，那么将返回 0。

下面是对此函数的自定义

```
int Atol(const char *str)
{
    int flag = 0, num = 0;
    const char *p;//定义临时指针
    p = str;//是指针指向字符数组首地址
    while(*p == ' ' || *p == '\t')    p++;
                                      //'t'表示制表符。此处作用为遇空格即跳跃。
    if(*p == '-')                     //如果遇到的第一个符号为-
    {
        flag = 1;
        p++;
    }
    if(*p == '+')                     //如果遇到第一个符号为+，决定之后的符号
    {
        flag = 0;
        p++;
    }
    while((unsigned char) *p <= '9' && (unsigned char) *p >= '0')
    //如果某地址对应字符为数，决定整数的大小。注意'0' <= (unsigned char) *p <= '9'写法不成立
    {
        num = num * 10 + (*p - '0');//输入由左向右数位由高至低
        p++;
    }
    if(flag) num = -num;
    return num;
}
```

6.2.6 字符串交换函数

6.2.6.1 直接交换

```
void Swap(char *a, char *b)          //这里把字符数组当成数组，传入数组首地址
{
    char str[N];
    //开的数组中元素数量需要为一合适的值 N，不能开太大，易造成空间大量浪费，同时，该数组也需要先初始化,初始化字符串为任意的，再进行引用
    StrCpy(str, a);                  //建立辅助数组进行数组拷贝
    StrCpy(a, b);
    StrCpy(b, str);
}
```

注意此处需要传入的实参为两个数组的首地址，并且这里的交换是直接对两个字符数组的元素进行改变。

6.2.6.2 间接交换

```
void Swapu(char** a, char** b)
 //二级指针指向指针变量，访问的变量元素为地址。传参方式为 Swapu(&str1, &str2)
{
    char *str = NULL;
    //空指针初始化，NULL 要大写，此处保证了代码运行的安全（未知指针直接引用是不安全的）
    str = *a;                    //交换 a,b 所指向的地址，不改变字符串常量的存储地址
    *a = *b;
    *b = str;                    //间接交换两个数组的首地址
}
```

这里与上一种方式不同的是需要传入的实参为字符数组的地址，这里可以把该数组看成指向自己首地址的指针变量，从而 a 表示指向指针的指针变量，取其地址所获取的即为该数组的首地址值，交换地址进而实现对两个字符串的间接交换。

6.2.6 字符串排序函数

接下来是字符串的排序函数。将字符串组合成为数组的方式也分为字符指针数组和二维字符数组两种形式。针对两种不同的形式，需要采用不同的参数定义方式，实现参量的传递。

6.2.6.1 字符串排序函数（版本一）

```
void BubbleA(char (*str)[NUM], int size)
{
    for (int i=0; i < size; i++)
    {
        for(int j = i+1; j < n1; j++)
        {
            if(StrCmp(str1[i], str1[j]) > 0)
                Swap(str1[i], str1[j]); //交换地址对应数据
        }
    }
}
```

上述实例用数组指针来处理多个字符串，缺点为开辟的 y 维数（如[8]是需要自己定义的），所以有很大可能会造成内存空间的浪费,由此提出了一个优化版本。

6.2.6.2 字符串排序函数（版本二）

```
void BubbleB(char *str2[], int size)
//需要传递的实参为字符指针数组，但传参规则与一般数组一致，即 sort2(str1, n1)
{
    for(int i = 0; i < n2; i++)
    {
        for(int j = i+1; j < size; j++)
        {
            if(StrCmp(str2[i], str2[j]) > 0)
            {
                Swapu(&str2[i], &str2[j]); //直接交换地址
            }
        }
    }
}
```

此方式改变了形参的数据结构，但没改变冒泡排序算法本身，这里所做的是调用 Swapu 函数，传入两个数组指针（地址）的地址，进行逐个交换，进而实现字符串的排序。

6.3 思考题

6.3.1

下面的测试函数中，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？

```
char strA[][NUM]={"enter", "number", "size", "begin", "of", "cat", "case",
"program", "certain", "a", "cake", "side"};
char *strB[]={"enter", "number", "size", "begin", "of", "cat", "case",
"program", "certain", "an", "cake", "side"};
char **strC, **strD;
```

答：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。

6.3.2

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？

堆区、栈区。

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。

6.3.3

GetStringsA, GetStringB 和 FreeStrings 函数的第一个参数为什么要使用三级指针？如果不用三级指针，会有什么结果？

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，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。

7 个人贡献、心得体会与反思

7.1 个人贡献

成员	内容	贡献比
严昕宇	计数与输出模块；冒泡排序算法优化；撰写论文	25%
史诚鑫	插入排序算法优化；撰写论文；论文校对	25%
刘桓菘	C-字符串处理；数据分析；撰写论文；	25%
奚仲璞	快速排序算法优化；数据处理与图表化；	25%

7.2 心得体会

姓名	心得体会
严昕宇	经历了两周的计算机编程实训，使我较为扎实地掌握了指针，函数，字符串等概念，并且大大增强了我阅读代码的能力。我完成的小组任务是计数模块的编写、部分算法的优化和论文撰写。 最初看到这份代码时，我感到一头雾水，对于整个工程文件中代码的逻辑感到迷惑与陌生。从内心的煎熬，到在经过自己的修修改改后，熟悉的代码运行出正确的答案后，心中的成就感油然而生。这次实训课，是我们第一次进行真正意义上的算法研究，和以往在不断做算法题不同，算法题是要你得出一个明确的答案。但我们这次进行的算法优化实验，没有所谓的标准答案，我们要通过查阅文献，与自己的不断尝试去的出一个相对更佳的结果。这次实训研究，也启示了我们，在今后的学习和实践中，自强不息，迎难而上，不厌其烦地找出问题、解决问题，这才是一个合格的学习者所应有的态度。
史诚鑫	本次编程实训，是我对算法有了更深入的理解。我在小组中的任务是快速排序的优化以及试图将算法可视化，刚开始思考如何对插入算法进行优化的时候，感到无从下手，于是开始查询各种资料以及参考文献。之后便开始着手写代码，通过自己的不断尝试以及思考最终基本解决了问题，即使写完运行时往往能出很多问题，但在解决了这些问题以后会有很强的满足感和自豪感。
刘桓菘	整个实训的过程其实就是做一个项目，让我们能够提前接触一个计算机工程项目，从某种角度来看，这与程序设计夏季赛的意义一致，都是为计算机专业同学之后选择方向打下基础。做算法竞赛，做项目实现，做开发等等都是很好的路径。所以说，这样的课程安排能够帮助我们很好地打开视野，改变我们的认知。 回顾整个项目的实现，简单来说就是解决一个个困难的过程。给出需求、遇到问题、设计方法、解决问题、实现功能 5 个方面组合成一个基本的实现单元。过程中我遇到的问题主要分布在语法和编译规则两个方面，在明确了具体问题后，我选择在 CSDN、博客源等资源网站上主动学习相关知识，经过一段时间的理解与尝试，功能基本都能够实现。当然，有时我也会遇到一些难以解决的问题，这个时候与小组的其他成员进行交流就尤为重要，不仅能缓解无法解决问题的烦恼，还能从中获得更好的点子和动力，更好地完成之后的工作。
奚仲璞	我在这次项目中，主要负责快速排序的优化。通过查阅文献及对未优化数据进行分析，我对快速排序进行了三种优化：三数取中，嵌入插入排序和三向切分，较大的提高了快速排序的速度，特别是对于特定的部分（完全）排序的数组。在这个过程中，通过查阅书籍和浏览各类专业性的网站,如 CSDN、编程中国，我对快速排序算法有了深入的了解，在对于排序优化的过程中，锻炼了我自我学习的能力。同时，此次项目也是对于团队合作的锻炼。

7.3 反思

7.3.1 多文件程序的编写问题

在之前的程序设计过程中，我们小组中没有成员接触过额外附加头文件的多文件程序编写方式，而此次实训课是我们第一次尝试采用这种方式编写程序。在参考了老师给予的代码文件和网上的相关资料①之后，我成功的编写了这次的作业程序。

多文件程序的组合由主程序文件（即包含 main()函数的*.c 文件）、头文件（*.h 文件）和与头文件名称相同的程序文件（*.c 文件）组成。举一个简单的例子，以下文件就可以构成一个多文件程序

```
Project---
| main.c            //主程序文件
| my_header.h       //自定义头文件
| my_header.c       //与头文件名称相同的程序文件
```

对于这类多文件程序，我们通常采用以下规范进行编写：

①整个程序中只有主程序文件有 main 函数，其他文件内没有 main 函数。编译得到的程序将会从主程序文件中的 main 函数开始执行。

②我们可以根据不同的功能特点、不同的开发者等，对我们自定义的函数进行分组存放。自定义函数的声明一般会被放置在自定义的头文件中，而函数定义则在同名的程序文件中进行。

③在进行多文件程序开发时，如果我们重复引用了同一个库文件，可能会导致编译出错。我们可以采用预定义宏的方式来避免这一问题。参见如下代码。

```
#ifndefHEADER_FILE      //预编译命令，如果引用过本头文件就不再引用
#defineHEADER_FILE      //第一次引用时定义宏，宏名一般与头文件名相同
//定义的头文件内容
#endif
```

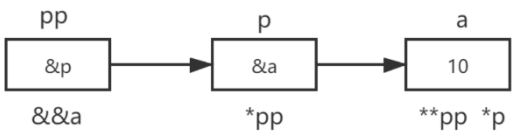
7.3.2 对 C 语言语法的不熟悉

在函数设计与测试过程中，由于对 C 语言部分语法知识的生疏，我们小组遇到了许多坎坷，而在指针与函数两个方面的问题尤为突出。在指针中，主要表现在多级指针的理解与应用；函数中，则主要体现在函数的定义、调用与声明规则。通过书籍的阅读与 CSDN 等资源网站的帮助，我们很快就理解了指针和函数的核心概念并通过一些范例的练习实现了基本的应用。在寻求帮助的过程中我们注意到图解对于我们理解某些计算机概念的重要性。通过图像说明，很多抽象的概念都变得可视化，进而辅助我们理解。

下面对二级指针的图解就是一个很好的范例。

```
int a = 10;
int* p = &a;
int** p = &p;
```

上述代码定义了 3 个变量 a、p 和 pp 并初始化。一级指针 p 指向整型变量 a，二级指针 pp 指向一级指针 p，如下图。



7.3.3 测试中比较案例的不完整

在设计完字符串比较函数（字典序），我们随之进行了测试，但其中测试案例只包括字母型字符串间的比较，这就导致我们小组在之后的二审中发现如果比较对象中有中文字符串，就会得到与比较理想相反的结果。分析其原因，我们发现汉字在计算机中一般会以 GB2312-80 编码进行储存，在这一编码方式中，汉字属于扩展 ASCII 字符集字符，由两个字节，分别储存’160+’区码和’160+’位码进行表示。由于通常使用的 char 类型表示整数的范围是从-128~127，在储存超过 127 的数字时就会因为溢出而变为负数。使得汉字编码总会比非汉字编码小，比较结果与预计相反。为避免这一问题的出现，我们只需要将传入的字符串指针转换为 unsigned char 类型（表示范围 0~255），就不会出现比较结果相反的错误了。
这个错误提醒我们要考虑到一个函数功能的完备性，应该在测试时设置多种情况来考虑它。

7.3.4 编译器的差异

在进行整个字符串处理函数设计工作时，部分小组成员 Mingw Developer Studio 使用的是，程序运行一切顺利。但将工程文件在某位小组成员使用的 Visual Studio 下进行编译时，却会出现报错的现象，我们发现不同集成开发环境在编译规则上有着许多差异。

因此我们探索了集成开发环境的定义。

集成开发环境（IDE Integrated Development Environment）是用于提供程序开发环境的应用程序，一般包括代码编写器、编译器、调试器和图形用户界面等工具。它集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套。所有具备这一特性的软件或者软件套都可以叫集成开发环境。

在不同编译环境中，纵使底层编译器是一致的，也难免在调试器或图形用户界面等模块有规则上的差异。所以在更换 IDE 编译代码时，要注意进行细节的修改。

8 结语

在本论文中，我们对排序算法的优化及 C-字符串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-字符串处理有了更深入的了解，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] 上海大学“网上教学平台”2020-2021 夏《计算机程序设计实训》课程 http://www.elearning.shu.edu.cn/portal

[2] CSDN技术社区

[3] CSDN技术社区 https://blog.csdn.net/hansionz/article/details/80822494

[4] CSDN 技术社区 https://blog.csdn.net/lemon_tree12138/article/details/50591859

[5] CSDN 技术社区 https://blog.csdn.net/weixin_42048417/article/details/81452732

[6] CSDN 技术社区 https://blog.csdn.net/qq_33289077/article/details/90370899

[7] CSDN 技术社区 https://blog.csdn.net/left_la/article/details/8656425

[8] CSDN 技术社区 https://blog.csdn.net/qq_38222051/article/details/109438886

[9] 博客园 https://www.cnblogs.com/ayqy/p/3862938.html

[10] CSDN 技术社区 https://blog.csdn.net/syyyy712/article/details/89918319

附录 A 未优化排序算法的操作次数

表 A.1 不同未优化排序算法在面对正态分布的整型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	708366	523776	3048	3528	5292	237145	238168
2048	2096128	3005202	2096128	6108	6974	10461	1003781	1005828
4096	8386560	12049788	8386560	12246	14120	21180	4020691	4024786
8192	33550336	48594357	33550336	24540	28536	42804	16206310	16214501
16384	134209536	190981950	134209536	49122	57318	85977	63677033	63693416
32768	536854528	769047369	536854528	98244	114300	171450	256381890	256414657
65536	2147450880	3078712956	2147450880	196572	227274	340911	1026303187	1026368722

表 A.2 不同未优化排序算法在面对均匀分布的整型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	752388	523776	2988	3730	5595	251819	252842
2048	2096128	3008499	2096128	5952	7680	11520	1004880	1006927
4096	8386560	12055674	8386560	11955	15438	23157	4022653	4026748
8192	33550336	48468954	33550336	23898	30764	46146	16164509	16172700
16384	134209536	193240536	134209536	47790	62072	93108	64429895	64446278
32768	536854528	787715877	536854528	95559	123026	184539	262604726	262637493
65536	2147450880	3131860302	2147450880	196584	247034	370551	1044018969	1044084504

表 A.3 不同未优化排序算法在面对完全顺序的整型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	0	523776	0	2046	3069	1023	2046
2048	2096128	0	2096128	0	4094	6141	2047	4094
4096	8386560	0	8386560	0	8190	12285	4095	8190
8192	33550336	0	33550336	0	16382	24573	8191	16382
16384	134209536	0	134209536	0	32766	49149	16383	32766
32768	536854528	0	536854528	0	65534	98301	32767	65534
65536	2147450880	0	2147450880	0	131070	196605	65535	131070

表 A.4 不同未优化排序算法在面对完全逆序的整型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	1571328	523776	1536	2046	3069	524799	525822
2048	2096128	6288384	2096128	3072	4094	6141	2098175	2100222
4096	8386560	25159680	8386560	6144	8190	12285	8390655	8394750
8192	33550336	100651008	33550336	12288	16382	24573	33558527	33566718
16384	134209536	402628608	134209536	24576	32766	49149	134225919	134242302
32768	536854528	1610563584	536854528	49152	65534	98301	536887295	536920062
65536	2147450880	6442352640	2147450880	98304	131070	196605	2147516415	2147581950

表 A.5 不同未优化排序算法在面对正态分布的浮点型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	802767	523776	3063	11543	7314	268612	269635
2048	2096128	3117420	2096128	6120	26687	15957	1041187	1043234
4096	8386560	12504252	8386560	12270	57286	34995	4172179	4176274
8192	33550336	50201229	33550336	24549	118445	76206	16741934	16750125
16384	134209536	202250052	134209536	49128	302835	161541	67433067	67449450
32768	536854528	810768978	536854528	98283	569258	351576	270289093	270321860
65536	2147450880	3215120757	2147450880	196575	1303522	741195	1071772454	1071837989

表 A.6 不同未优化排序算法在面对均匀分布的浮点型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	789267	523776	3036	10918	7422	264112	265135
2048	2096128	3235986	2096128	6108	25635	16251	1080709	1082756
4096	8386560	12651096	8386560	12273	62109	34671	4221127	4225222
8192	33550336	49717377	33550336	24558	125159	75135	16580650	16588841
16384	134209536	202729743	134209536	49122	269235	162006	67592964	67609347
32768	536854528	812325141	536854528	98283	573365	347154	270807814	270840581
65536	2147450880	3222544809	2147450880	196575	1286528	724500	1074247138	1074312673

表 A.7 不同未优化排序算法在面对完全顺序的浮点型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	0	523776	0	523776	3069	1023	2046
2048	2096128	0	2096128	0	2096128	6141	2047	4094
4096	8386560	0	8386560	0	8386560	12285	4095	8190
8192	33550336	0	33550336	0	33550336	24573	8191	16382
16384	134209536	0	134209536	0	134209536	49149	16383	32766
32768	536854528	0	536854528	0	536854528	98301	32767	65534
65536	2147450880	0	2147450880	0	2147450880	196605	65535	131070

表 A.8 不同未优化排序算法在面对完全逆序的浮点型数据时的操作次数

数据规模	冒泡排序		直接插入排序		选择排序		快速排序	
	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数	比较次数	赋值次数
1024	523776	1571328	523776	1536	523776	3069	524799	525822
2048	2096128	6288384	2096128	3072	2096128	6141	2098175	2100222
4096	8386560	25159680	8386560	6144	8386560	12285	8390655	8394750
8192	33550336	100651008	33550336	1228	33550336	24573	33558527	33566718
16384	134209536	402628608	134209536	2457	134209536	49149	134225919	134242302
32768	536854528	1610563584	536854528	4915	536854528	98301	536887295	536920062
65536	2147450880	6442352640	2147450880	98304	2147450880	196605	2147516415	2147581950

附录 B 未优化与已优化的冒泡排序算法的比较

表 B.1 未优化与已优化的冒泡排序算法在面对正态分布的整型数据时的操作次数

数据规模	冒泡排序				双向冒泡排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	523776	771513	1295289	0.002	399288	771513	1170801
2048	0.009	2096128	2997882	5094010	0.006	1570814	2997882	4568696
4096	0.038	8386560	11962128	20348688	0.026	6192000	11962128	18154128
8192	0.169	33550336	47955087	81505423	0.12	25190382	47955087	73145469
16384	0.743	134209536	192075948	326285484	0.532	100252446	192075948	292328394
32768	3.073	536854528	764910501	1301765029	2.205	401832734	764910501	1166743235
65536	12.658	2147450880	3066608262	5214059142	9.046	1614276480	3066608262	4680884742

表 B.2 未优化与已优化的冒泡排序算法在面对均匀分布的整型数据时的操作次数

数据规模	冒泡排序				双向冒泡排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	523776	771513	1295289	0.002	399288	771513	1170801
2048	0.009	2096128	2997882	5094010	0.006	1570814	2997882	4568696
4096	0.038	8386560	11962128	20348688	0.026	6192000	11962128	18154128
8192	0.169	33550336	47955087	81505423	0.12	25190382	47955087	73145469
16384	0.743	134209536	192075948	326285484	0.532	100252446	192075948	292328394
32768	3.073	536854528	764910501	1301765029	2.205	401832734	764910501	1166743235
65536	12.658	2147450880	3066608262	5214059142	9.046	1614276480	3066608262	4680884742

表 B.3 未优化与已优化的冒泡排序算法在面对完全正序的整型数据时的操作次数

数据规模	冒泡排序				双向冒泡排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0.001	523776	0	523776	0	2046	0	2046
2048	0.005	2096128	0	2096128	0	4094	0	4094
4096	0.021	8386560	0	8386560	0	8190	0	8190
8192	0.085	33550336	0	33550336	0	16382	0	16382
16384	0.337	134209536	0	134209536	0	32766	0	32766
32768	1.355	536854528	0	536854528	0.001	65534	0	65534
65536	5.408	2147450880	0	2147450880	0	131070	0	131070

表 B.4 未优化与已优化的冒泡排序算法在面对完全逆序的整型数据时的操作次数

数据规模	冒泡排序				双向冒泡排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0.002	523776	1571328	2095104	0.002	524288	1571328	2095616
2048	0.01	2096128	6288384	8384512	0.009	2097152	6288384	8385536
4096	0.039	8386560	25159680	33546240	0.036	8388608	25159680	33548288
8192	0.164	33550336	100651008	134201344	0.148	33554432	100651008	134205440
16384	0.644	134209536	402628608	536838144	0.589	134217728	402628608	536846336
32768	2.594	536854528	1610563584	2147418112	2.402	536870912	1610563584	2147434496
65536	12.658	2147450880	3066608262	5214059142	9.046	1614276480	3066608262	4680884742

附录 C 不同优化思路的快速排序算法的比较

表 C.1 未优化与已优化的快速排序算法在面对正态分布的整型数据时的操作次数

数据规模	快速排序				三数取中的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	30494	5178	35672	0	32646	5559	38205
2048	0.001	92767	10440	103207	0	87902	11268	99170
4096	0.001	389569	20877	410446	0.001	326630	22530	349160
8192	0.003	1377054	43890	1420944	0.003	1224991	45534	1270525
16384	0.013	5582180	85689	5667869	0.01	4247127	91257	4338384
32768	0.043	18916926	171192	19088118	0.039	16963767	180072	17143839
65536	0.192	85357612	342945	85700557	0.166	75176762	360888	75537650

表 C.2 未优化与已优化的快速排序算法在面对完全顺序的整型数据时的操作次数

数据规模	快速排序				三数取中的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0.001	525822	3069	528891	0.001	10764	3069	13833
2048	0.005	2100222	6141	2106363	0	23565	6141	29706
4096	0.019	8394750	12285	8407035	0	51214	12285	63499
8192	0.076	33566718	24573	33591291	0.001	110607	24573	135180
16384	0.304	134242302	49149	134291451	0.001	237584	49149	286733
32768	1.209	536920062	98301	537018363	0.002	507921	98301	606222
65536	4.809	2147581950	196605	2147778555	0.004	1081362	196605	1277967

表 C.3 未优化与已优化的快速排序算法在面对完全顺序的整型数据时的操作次数

数据规模	三数取中的快速排序				三数取中的+嵌入插入排序的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0.001	525822	3069	528891	0	11784	4605	16389
2048	0.005	2100222	6141	2106363	0	25609	9213	34822
4096	0.019	8394750	12285	8407035	0	55306	18429	73735
8192	0.074	33566718	24573	33591291	0	118795	36861	155656
16384	0.293	134242302	49149	134291451	0.001	253964	73725	327689
32768	1.143	536920062	98301	537018363	0.002	540685	147453	688138
65536	4.652	2147581950	196605	2147778555	0.004	1146894	294909	1441803

表 C.4 未优化与已优化的快速排序算法在面对正态分布的整型数据时的操作次数

数据规模	三数取中的快速排序				三数取中的+嵌入插入排序的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	32646	5559	38205	0	22630	5335	27965
2048	0	87902	11268	99170	0	56907	10817	67724
4096	0.001	326630	22530	349160	0.001	222042	21744	243786
8192	0.003	1224991	45534	1270525	0.003	874283	44231	918514
16384	0.01	4247127	91257	4338384	0.009	2871295	88176	2959471
32768	0.039	16963767	180072	17143839	0.035	11651905	174230	11826135
65536	0.166	75176762	360888	75537650	0.158	55198177	350833	55549010

表 C.5 不同优化思路的快速排序算法在面对正态分布的整型数据时的操作次数

数据规模	三数取中的+嵌入插入排序的快速排序				三数取中的+嵌入插入排序的快速排序 +三向切分的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	22630	5335	27965	0	11113	4187	15300
2048	0	56907	10817	67724	0	21909	6821	28730
4096	0.001	222042	21744	243786	0	44366	12737	57103
8192	0.003	874283	44231	918514	0	86887	24561	111448
16384	0.009	2871295	88176	2959471	0.001	170657	47398	218055
32768	0.035	11651905	174230	11826135	0.001	339970	82469	422439
65536	0.158	55198177	350833	55549010	0.003	737255	270329	1007584

表 C.6 不同优化思路的快速排序算法在面对完全顺序的整型数据时的操作次数

数据规模	三数取中的+嵌入插入排序的快速排序				三数取中的+嵌入插入排序 +三向切分的快速排序			
	时间	比较次数	赋值次数	总操作次数	时间	比较次数	赋值次数	总操作次数
1024	0	4875	2302	7177	0	18519	10668	29187
2048	0	10764	4606	15370	0	42227	23276	65503
4096	0	23565	9214	32779	0.001	94858	50463	145321
8192	0	51214	18430	69644	0.001	210488	108799	319287
16384	0.001	110607	36862	147469	0.001	462341	233272	695613
32768	0.001	237584	73726	311310	0.002	1007253	497638	1504891
65536	0.002	507921	147454	655375	0.005	2179783	1057262	3237045