排序算法的优化及 MIDI 可视化处理

李昀哲 邱逸辰 倪思远 丁乐俊 杜佳杰

20123101 20123102 20123103 20123104 20123105

(计算机工程与科学学院)

摘　要　根据课程目标，本文设计三种排序函数，即：冒泡排序、选择排序、快速排序的算法优化，用不同规模（如：从 1024 开始，逐次倍增直至 65536）、多种构型（如：正序、逆序、均匀分布、正态分布、结构体数组）测试数据进行算法测试并进行速度的优化；对字符串的排序及存储空间不同所带来算法不同的分析；以及将三种算法使用 MIDI 可视化的处理。

关键词　排序优化、不同存储方式字符串排序、排序可视化

Optimization of sorting algorithm and MIDI visual processing

Yunzhe Li, Yichen Qiu, Siyuan Ni, Lejun Ding, Jiajie Du

(School of Computer Engineering and Science)

Abstract According to the course objectives, this passage has designed three kinds of sort function, i.e., the bubble sort, selection sort, quick sort algorithm, with different size (range from 1024, successive multiplication to 65536), a variety of configurations of data (such as ordinal sequence, inverted sequence, uniform distribution, normal distribution, structure arrays) test and speed the optimizatio n of the algorithm; The analysis of the different algorithms caused by the sorting of strings and the different storage space; And the thr ee algorithms using MIDI visualization processing.

Key words optimization of sorting algorithm, different ways to store a string sort, visualization of sorts

1 引言

本文基于三种基础排序算法进行优化，并从多个维度进行对比测试，分别从同种算法优化前后同种数据类型及数据规模的测试、同种算法优化前后不同种数据类型的测试、不同种算法对于不同种数据类型及数据规模的测试。进而得出某一算法下优化的核心关键的结论以及某一数据类型、数据集下，最优解的排序方式。

再通过对于不同存储方式下字符串的分析，设计出 2 种针对指针数组和数组指针不同的排序方式。进一步对不同数据类型在计算机中的存储方式有了进一步了解。

最后，通过对于 MIDI 可视化程序的研究分析，将优化后的两种算法（冒泡排序和选择排序）的排序方式可视化处理，帮助读者更加便捷地理解排序算法的运行过程和设计思路。

2 基于基础冒泡排序算法的优化及实验

2.1 冒泡排序的优化

冒泡排序属于一种典型的交换排序。

交换排序顾名思义就是通过元素的两两比较，判断是否符合要求，如过不符合就交换位置来达到排序的目的。冒泡排序名字的由来就是因为在交换过程中，类似水冒泡，小（大）的元素经过不断的交换由水底慢慢的浮到水的顶端。

冒泡排序的思想就是利用的比较交换，利用循环将第 i 小或者大的元素归位，归位操作利用的是对 n 个元素中相邻的两个进行比较，如果顺序正确就不交换，如果顺序错误就进行位置的交换。通过重复的循环访问数组，直到没有可以交换的元素，那么整个排序就已经完成了。

2.2 冒泡排序常规版-代码实现

下面详细分析一下常规版的冒泡排序，整个算法流程其实就是上面实例所分析的过程。可以看出，我们在进行每一次大循环的时候，还要进行一个小循环来遍历相邻元素并交换。所以我们的代码中首先要有两层循环。

外层循环：即主循环，需要辅助我们找到当前第 i 小的元素来让它归位。所以我们会一直遍历 n-2 次，这样可以保证前 n-1 个元素都在正确的位置上，那么最后一个也可以落在正确的位置上了。

内层循环：即副循环，需要辅助我们进行相邻元素之间的比较和换位，把大的或者小的浮到水面上。所以我们会一直遍历 n-1-i 次这样可以保证没有归位的尽量归位，而归位的就不用再比较了。

而上面的问题，出现的原因也来源于这两次无脑的循环，正是因为循环不顾一切的向下执行，所以会导致在一些特殊情况下得多余。例如 5，4，3，1，2 的情况下，常规版会进行四次循环，但实际上第一次就已经完成排序了。

2.3算法的第一次优化

首先针对第一个问题，当我们进行完第三遍循环的时候，实际上整个排序都已经结束了了，但是未优化版还是会继续排序。

为了解决这个问题，我们可以设置一个标志位，用来表示当前第 i 趟是否有交换，如果有则要进行 i+1 趟，如果没有，则说明当前数组已经完成排序。

因为标志位的存在，上面的循环只会进行一遍，若 flag 没有变成 1，整个算法就结束了，此时的时间复杂度就会变为 O(n) 。

```
if (arr[j]>arr[j + 1])
    {
        temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
        flag = 1;//加入标记
    }
if (flag == 0)//如果没有交换过元素，则已经有序,直接结束
    {
        break;}
```

2.4 算法的第二次优化

除了上面这个问题，在冒泡排序中还有一个问题存在，就是第 i 趟排的第 i 小或者大的元素已经在第 i 位上了，甚至可能第 i-1 位也已经归位了，那么在内层循环的时候，有这种情况出现就会导致多余的比较出现。例如：6，4，7，5，1，3，2，当我们进行第一次排序的时候，结果为 6，7，5，4，3，2，1，实际上后面有很多次交换比较都是多余的，因为没有产生交换操作。

针对上述的问题，我们可以想到，利用一个标志位 pos，记录一下当前第 i 趟所交换的最后一个位置的下标，在进行第 i+1 趟的时候，只需要内循环到这个下标的位置就可以了，因为后面位置上的元素在上一趟中没有换位，这一次也不可能会换位置了。基于这个原因，我们可以进一步优化我们的代码。

```
if (arr[j]>arr[j + 1])
    {
        temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
        flag = 1;//加入标记
        pos = j;//交换元素，记录最后一次交换的位置
    }
if (flag == 0)//如果没有交换过元素，则已经有序,直接结束
    {
        break;
    }
k = pos;//下一次比较到记录位置即可
```

2.5 算法的第三次优化

在第二次优化过后，其实程序的效率已经有了很大的提升，但是还有进一步的优化空间。大致思想就是一次排序可以确定两个值，正向冒泡找到最大值交换到最后，同时反向扫描找到最小值交换到最前面。此时的时间复杂度虽然仍然为 O(n)，但是效率大大提升，可以很好的解决一个数组的前半段大致递增，而最小值在数列末端的情况。

2.6 优化前后对比试验

2.6.1 int 类型正态分布优化前后对比

优化后的算法在不同规模数据集下都有较为显著的提升：其中 1024 规模的数据集优化提速约 50%；2048 规模的数据集优化提速约 1/3；4096 规模的数据集优化提速约 21.7%；8192 规模的数据集优化提速约 11.6%；16384 规模的数据集优化提速约 17.2%；32184 规模的数据集优化提速约 28.0%；65536 规模的数据集优化提速约 30.7%，详见**图 1，表 1**。在数据规模较小和较大的情况下最为显著。

表 1. 冒泡排序-正态分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.002	0.001	50
2048	0.006	0.004	0.001
4096	0.023	0.018	0.001
8192	0.095	0.084	0.004
16384	0.453	0.375	0.008
32768	2.156	1.553	0.016
65536	9.058	6.273	0.035

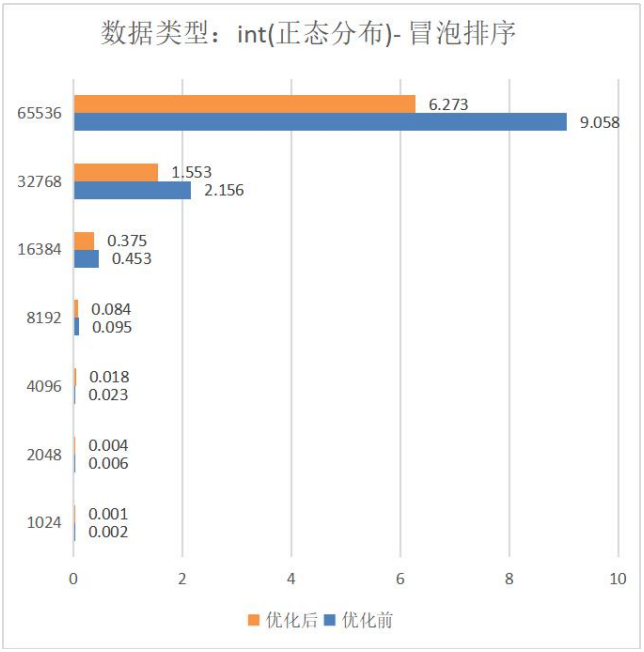

图 1. 冒泡排序-正态分布整型数据排序时间（秒）

2.6.2 int 类型均匀分布优化前后对比

对于均匀分布的情况，优化后的算法在不同规模数据集下也均有较为显著的提升：其中 1024 规模的数据集优化提速约 50%；2048 规模的数据集优化提速约 16.7%；4096 规模的数据集优化提速约 21.7%；8192 规模的数据集优化提速约 26.1%；16384 规模的数据集优化提速约 19.2%；32184 规模的数据集优化提速约 25.9%；65536 规模的数据集优化提速约 29.5%，详见**图 2，表 2**。在数据规模较小和较大的情况下最为显著。

表 2. 冒泡排序-均匀分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.002	0.001	50.00
2048	0.006	0.005	16.67
4096	0.023	0.017	21.74
8192	0.096	0.078	26.09
16384	0.453	0.366	19.21
32768	2.113	1.566	25.89
65536	9.015	6.353	29.53

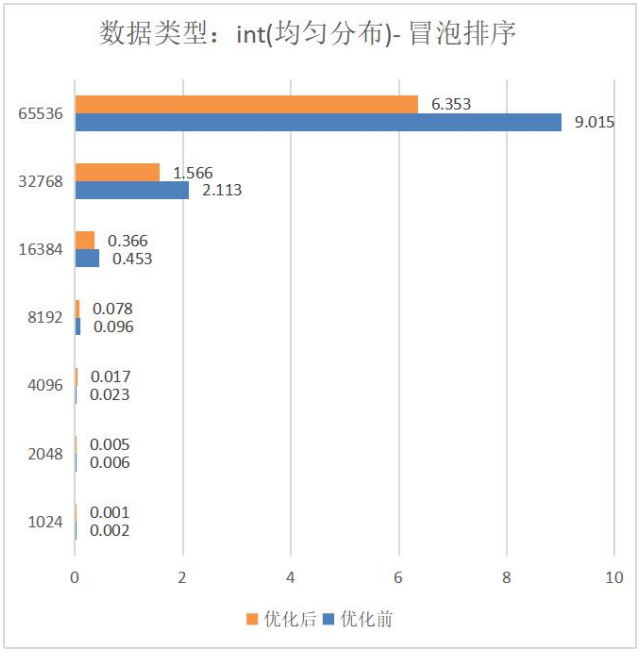

图 2. 冒泡排序-均匀分布整型数据排序时间（秒）

2.6.3 int 类型完全顺序优化前后对比

优化后的算法通过设立标记位的方法可以检验数组数据是否有序来提高效率；因此在完全顺序的情况下代码的优化效率可以提高至 100%，详见**图 3，表 3**。

表 3. 冒泡排序-完全顺序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.001	0	100
2048	0.003	0	100
4096	0.01	0	100
8192	0.045	0	100
16384	0.174	0	100
32768	0.692	0	100
65536	2.774	0	100

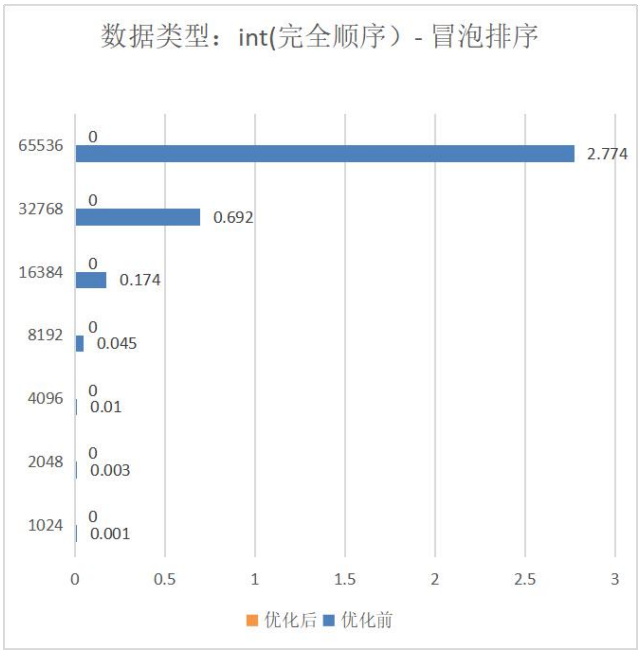

图 3. 冒泡排序-完全顺序整型数据排序时间（秒）

2.6.4 int 类型完全逆序优化前后对比

优化后的算法在不同规模数据集下的效率都稍有下降，详见**图 4，表 4**。经过测试分析可能原因如下：由于代码采用正反双向排序，在完全逆序的情况下相当于要额外经历一遍循环，扫描次数大约为均匀分布情况下的两倍，因此优化后的效率反而较优化前的效率略低。

表 4. 冒泡排序-完全逆序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.001	0.001	0
2048	0.006	0.007	-16.67
4096	0.024	0.025	-4.17
8192	0.1	0.101	-1.00
16384	0.399	0.411	-3.01
32768	1.601	1.638	-2.31
65536	6.352	6.629	-5.94

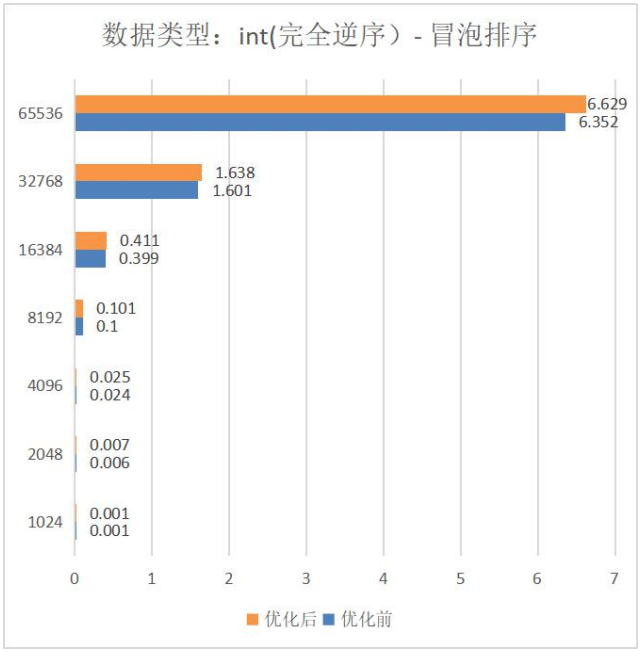

图 4. 冒泡排序-完全逆序整型数据排序时间（秒）

2.6.5 double 类型正态分布优化前后对比

同 int 类型正态分布情况类似，优化后的算法在不同规模数据集下都有较为显著的提升，详见**图 5，表 5**：其中 1024 规模的数据集优化提速约 50%；2048 规模的数据集优化没有提速；4096 规模的数据集优化提速约 17.9%；8192 规模的数据集优化提速约 7.8%；16384 规模的数据集优化提速约 3.2%；32184 规模的数据集优化提速约 19.8%；65536 规模的数据集优化提速约 25.2%。在数据规模较小和较大的情况下最为显著。

表 5. 冒泡排序-正态分布 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.002	0.001	50.00
2048	0.006	0.006	0
4096	0.028	0.023	17.86
8192	0.116	0.107	7.76
16384	0.591	0.572	3.21
32768	2.9	2.325	19.83
65536	12.508	9.357	25.19

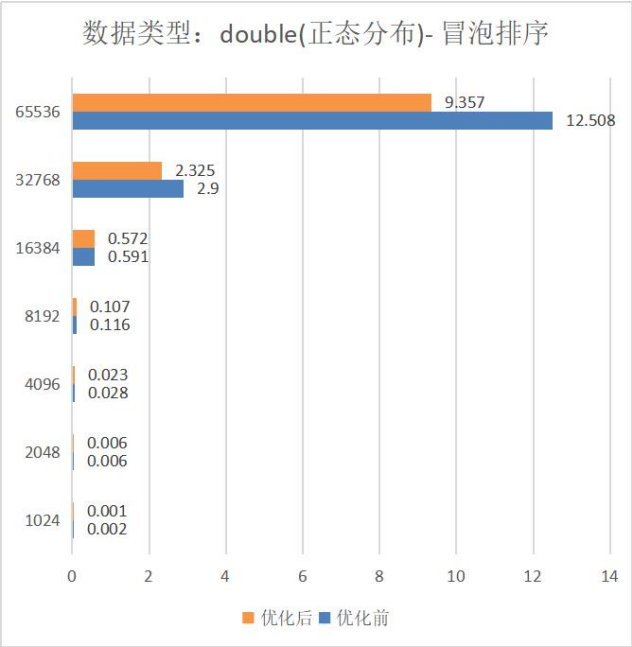

图 5. 冒泡排序-正态分布 double 型数据排序时间（秒）

2.6.6 double 类型均匀分布优化前后对比

同 int 类型均匀分布情况类似，对于均匀分布的情况，优化后的算法在不同规模数据集下也均有较为显著的提升：其中 1024 规模的数据集优化提速约 50%；2048 规模的数据集优化提速约 16.7%；4096 规模的数据集优化提速约 21.7%；8192 规模的数据集优化提速约 26.1%；16384 规模的数据集优化提速约 19.2%；32184 规模的数据集优化提速约 25.9%；65536 规模的数据集优化提速约 29.5%，详见**图 6，表 6**。在数据规模较小和较大的情况下最为显著。

表 6. 冒泡排序-均匀分布 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.002	0.001	50
2048	0.006	0.005	16.67
4096	0.026	0.025	3.85
8192	0.118	0.107	9.32
16384	0.57	0.521	8.60
32768	2.903	2.303	20.67
65536	12.456	9.291	25.41

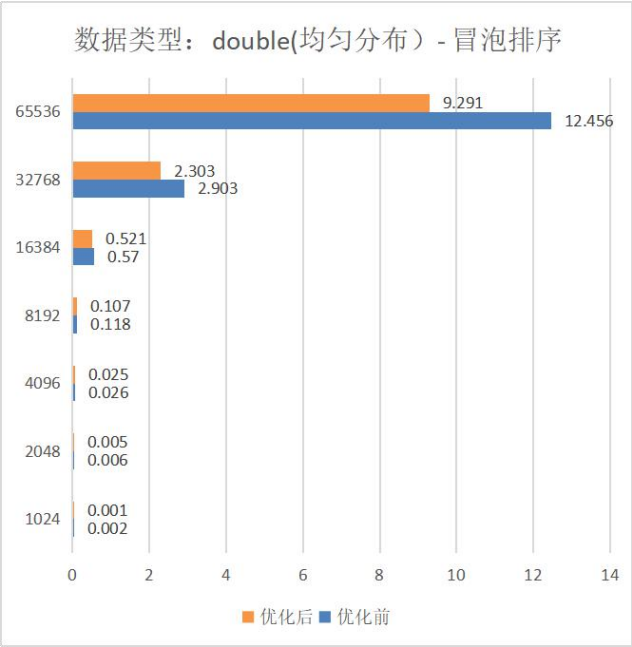

图 6. 冒泡排序-均匀分布 double 型数据排序时间（秒）

2.6.7 double 类型完全顺序优化前后对比

同 int 类型完全顺序情况类似，优化后的算法通过设立标记位的方法可以检验数组数据是否有序来提高效率；因此在完全顺序的情况下代码的优化效率可以提高至 100%，详见**图 7，表 7**。

表 7. 冒泡排序-完全顺序 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.001	0	100
2048	0.003	0	100
4096	0.011	0	100
8192	0.045	0	100
16384	0.172	0	100
32768	0.69	0	100
65536	2.87	0	100

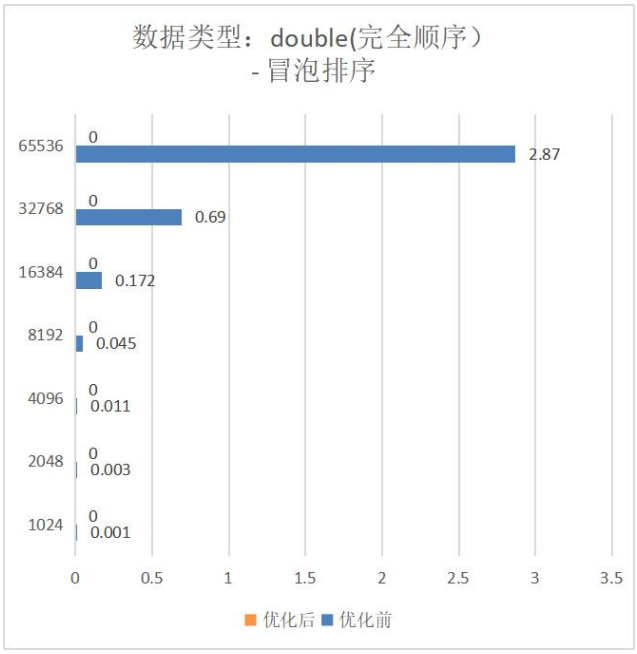

图 7. 冒泡排序-完全顺序 double 型数据排序时间（秒）

2.6.8 double 类型完全逆序优化前后对比

同 int 类型完全逆序情况类似，优化后的算法在不同规模数据集下的效率都稍有下降，详见**图 8，表 8**。经过测试分析可能原因如下：由于代码采用正反双向排序，在完全逆序的情况下相当于要额外经历一遍循环，扫描次数大约为均匀分布情况下的两倍，因此优化后的效率反而较优化前的效率略低。

表 8. 冒泡排序-完全逆序 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比
1024	0.004	0.002	50.00
2048	0.01	0.012	-5.00
4096	0.046	0.047	-2.17
8192	0.179	0.183	-2.23
16384	0.709	0.737	-3.95
32768	2.821	2.904	-2.94
65536	11.367	11.786	-3.69

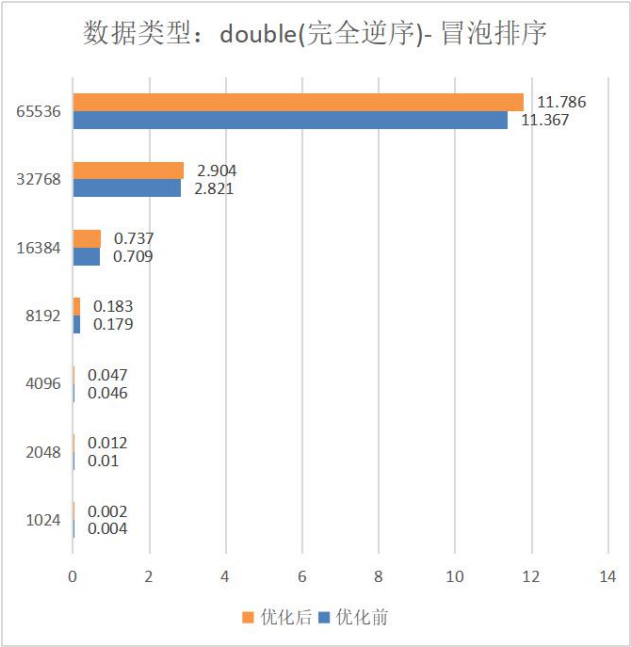

图 8. 冒泡排序-完全逆序 double 型数据排序时间（秒）

3 基于基础选择排序的优化及实验

3.1 选择排序的优化

选择排序第一次循环首先固定首个元素，从头到尾扫描序列，通过比较与首个元素大小，确定当前范围内"最小"元素的下标，若"最小"元素不是固定元素，则两者交换。然后进行第二次循环，以未排序数列首个元素（即第二个元素）为固定元素排序，以此类推……理论上有 n 个待排序元素需进行 n-1 次排序。

```
for(i=1; i<size; i++)
{
    for(j=i; j<size; j++)
     {
```

```
        if(a[j] < a[k])
                k = j;
        if(k!=i-1)
        {
            temp = a[k];
            a[k] = a[i-1];
            a[i-1] = temp;
        }
        k = i;
      }
        }
```

原选择排序的交换元素代码于内循环中，每次交换排一个元素，交换总次数为 N，则总的时间复杂度是 O(N*N)。

简单选择排序算法每一次循环只能确定一个元素的最终位置，经过优化后的选择排序，在遍历剩余元素的过程中，不仅排定最小元素，还排定了最大值。

```
    for (int i = 0; i < k; ++i,--k) {
        min = i;
        max = i;
        for (j = i+1; j < k; ++j) {
            if (a[j] < a[min]) {
                min = j;                //排定最小
            }
            if (a[j] > a[max]) {
                max = j;                //排定最大
            }
        }
```

每次排序从无序数组区间中选出最大最小两个空间，把最小元素换至数组无序部分表头，把最大元素换至数组无序部分表尾，这样在下一次循环过程中无序数组元素个数从 n 变为 n-2，这样理论上 n 个待排元素只需进行 n/2 次排序。若最小值非无序数组表头，则进行交换；

```
   if (min != i) {
            temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }
```

最大值非无序数组表尾亦然，

```
if (max == i) {
        max = min;
    }
    if (max != k-1) {
        temp = a[k-1];
        a[k-1] = a[max];
        a[max] = temp;
    }
}
```

算法总的比较次数为 O（$\frac{N}{2}*\frac{N}{2}$）。经过优化虽然时间复杂度仍是平方级别，但运行时间已有减少，空间复杂度不发生改变，仍为 O（1）。

3.2 优化前后对比试验

在 int 的数据类型下，各数据量速度都有一些提升，随着数据规模的扩大，提速百分比并未呈现一定的规律性，基本稳定在类似数值附近。

在 double 的数据类型下，各数据量速度都有一些提升，随着数据规模的扩大，提速百分比并未呈现一定的规律性，基本稳定在类似数值附近。

3.2.1 int 类型正态分布优化前后对比

int 类型正态分布数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 22.2%，详见**表 9，图 9**

表 9. 选择排序正态分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100.0
2048	0.004	0.003	25.0
4096	0.014	0.01	28.6
8192	0.048	0.04	16.7
16384	0.205	0.162	21.0
32768	0.806	0.652	22.5
65536	3.091	2.486	19.6

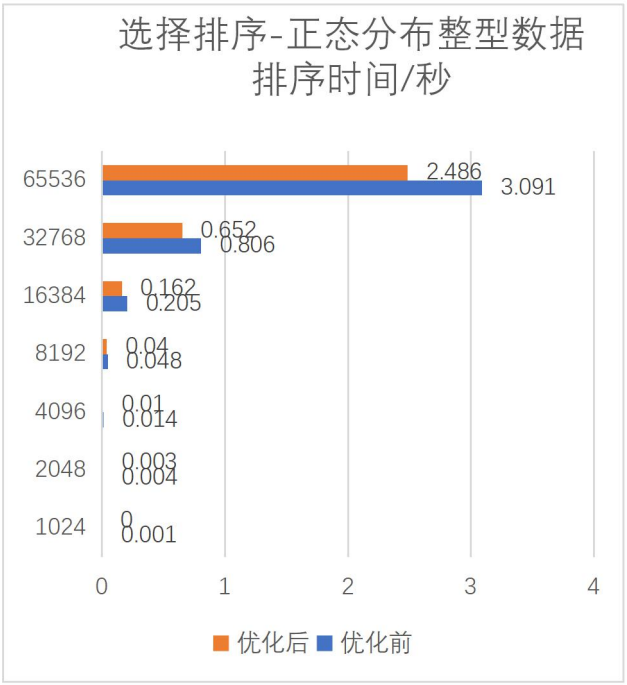

图 9. 选择排序-完全顺序整型数据排序时间（秒）

3.2.2 int 类型均匀分布优化前后对比

int 类型均匀分布数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 19.6%，详见**表 10，图 10**

表 10. 选择排序-均匀分布分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0.0
2048	0.003	0.002	33.3
4096	0.011	0.01	9.1
8192	0.046	0.038	17.4
16384	0.188	0.15	20.2
32768	0.767	0.62	19.2
65536	3.012	2.454	18.5

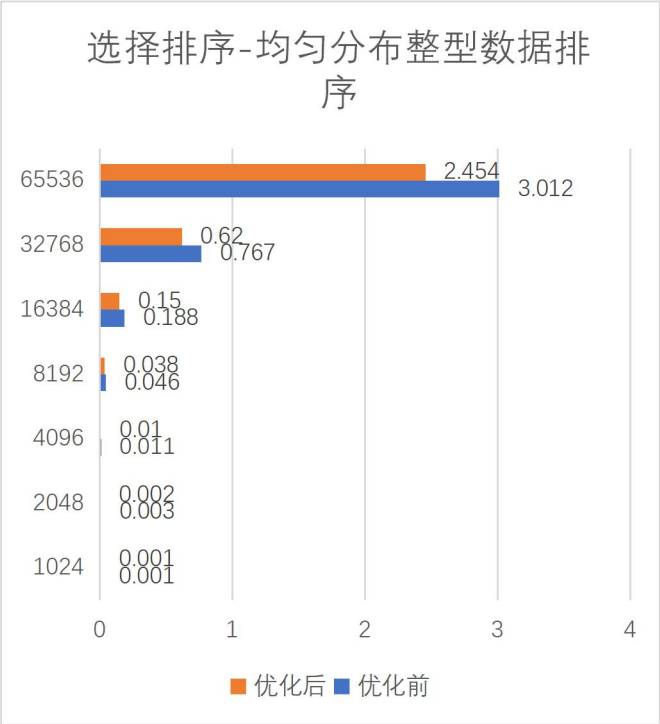

图 10. 选择排序-均匀分布整型数据排序时间（秒）

3.2.3 int 类型完全顺序优化前后对比

int 类型完全顺序数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 14.1%，详见**表 11，图 11**

表 11. 选择排序-完全顺序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100.0
2048	0.003	0.003	0.0
4096	0.011	0.009	18.3
8192	0.049	0.042	14.3
16384	0.184	0.163	11.4
32768	0.735	0.659	10.3
65536	3.124	2.616	16.3

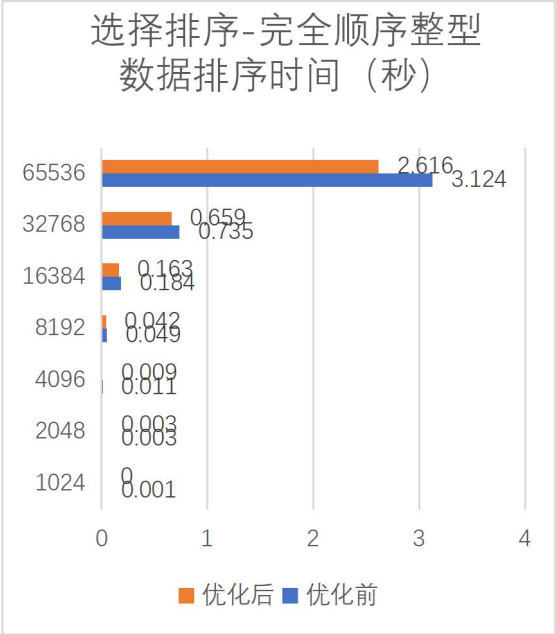

图 11. 选择排序-完全顺序整型数据排序时间（秒）

3.2.4 int 类型完全逆序优化前后对比

int 类型完全顺序数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 20.9%，详见**表 12，图 12**

数据规模上看，优化前后排序时间基本随数据规模呈现平方倍的线性关系。

表 12. 选择排序-完全逆序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100.0
2048	0.003	0.002	33.3
4096	0.012	0.009	25.0
8192	0.046	0.037	19.6
16384	0.191	0.161	15.7
32768	0.782	0.655	16.2
65536	3.082	2.601	15.6

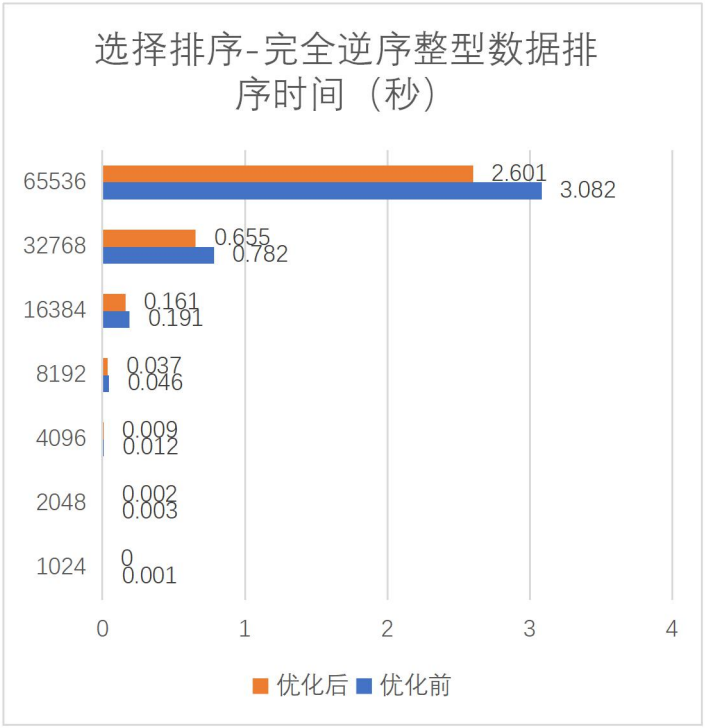

图 12. 选择排序-完全逆序整型数据排序时间（秒）

3.2.5 double 类型完全逆序优化前后对比

double 类型正态分布数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 19.8%，详见**表 13，图 13**

表 13. 选择排序-正态分布 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100.0
2048	0.003	0.002	33.3
4096	0.011	0.009	18.2
8192	0.047	0.039	17.0
16384	0.183	0.154	15.8
32768	0.758	0.637	16.0
65536	2.969	2.415	18.7

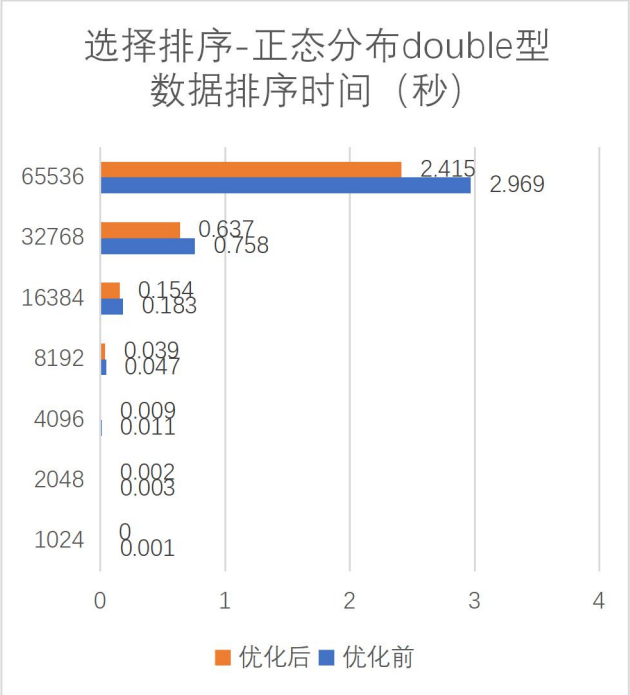

图 13. 选择排序-正态分布 double 型数据排序时间（秒）

3.2.6 double 类型均匀分布优化前后对比

double 类型均匀分布数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 16.8%，详见**表 14，图 14**

表 14. 选择排序-均匀分布 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0.0
2048	0.003	0.002	33.3
4096	0.011	0.01	9.1
8192	0.044	0.042	4.5
16384	0.181	0.152	16.0
32768	0.745	0.603	19.1
65536	3.009	2.437	19.0

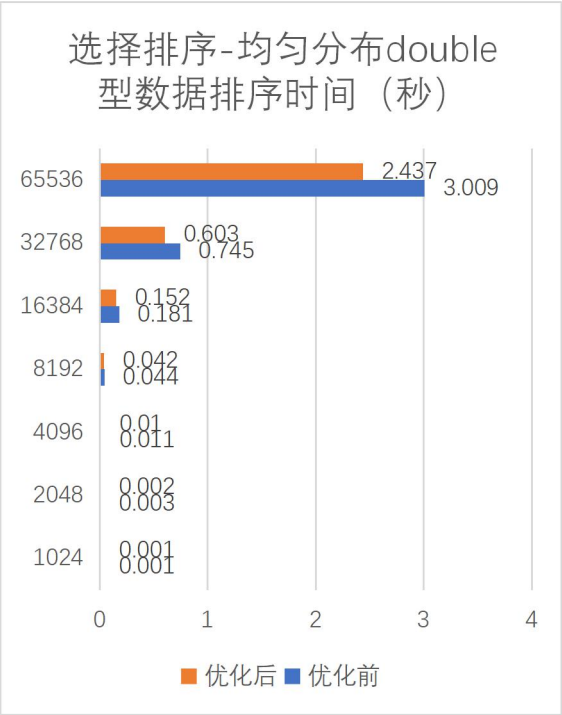

图 14. 选择排序-均匀分布 double 型数据排序时间（秒）

3.2.7 double 类型完全顺序优化前后对比

double 类型完全顺序数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 15.9%，详见**表 15，图 15**

表 15. 选择排序-完全顺序 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0.0
2048	0.003	0.002	33.3
4096	0.011	0.01	9.1
8192	0.048	0.038	20.8
16384	0.177	0.168	5.1
32768	0.739	0.648	12.3
65536	3.018	2.58	14.5

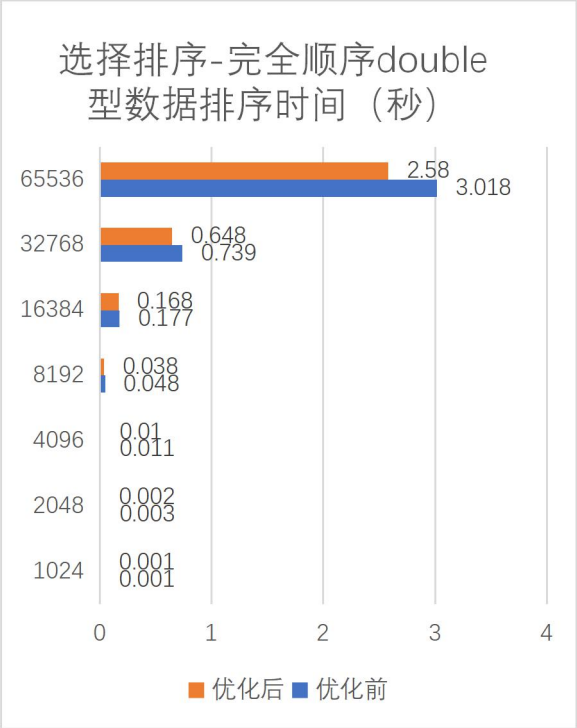

图 15. 选择排序-完全顺序 double 型数据排序时间（秒）

3.2.8 double 类型完全逆序优化前后对比

double 类型完全顺序数据优化前后提速百分比的平均值（排除选择排序优化后排序时间为 0 和优化前排序时间极小（<0.005）时的等值的极端情况）为 20.55%，详见**表 16，图 16**

数据规模上看，优化前后排序时间基本随数据规模呈现平方倍的线性关系。

表 16. 选择排序-完全逆序 double 型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0.0
2048	0.003	0.002	33.3
4096	0.012	0.01	16.7
8192	0.048	0.038	20.8
16384	0.194	0.156	19.6
32768	0.78	0.646	17.2
65536	3.1	2.613	15.7

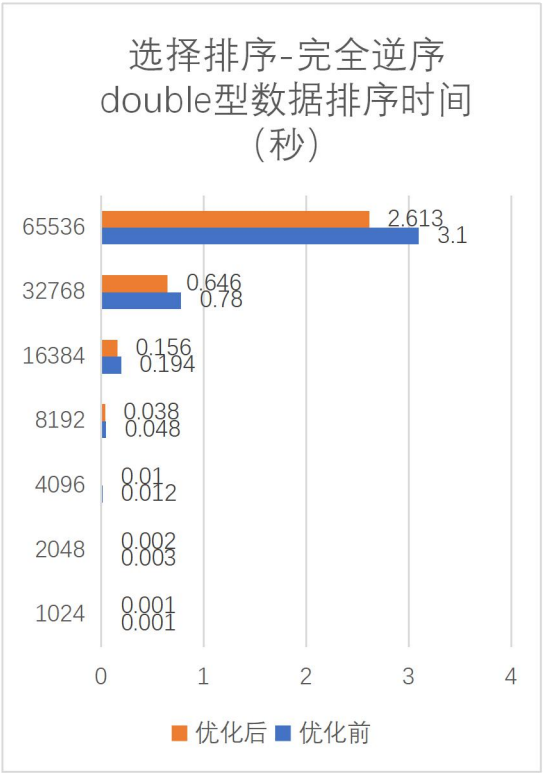

图 16. 选择排序-完全顺序 double 型数据排序时间（秒）

4 基于基础快速排序的优化及实验

4.1基本原理

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小。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，以此达到整个数据变成有序序列。

```
void qSort(int arr[], int low, int high)
{
    int pivot = 0;
    if(low < high)
    {
        pivot = partition(arr, low, high);

        //递归
        qSort(arr, low, pivot - 1);
        qSort(arr, pivot + 1, high);
    }
}
```

快速排序的平均时间复杂度和最坏时间复杂度分别是 O(nlgn)、O(n^2)。

当排序已经成为基本有序状态时，快速排序退化为O(n^2)，一般情况下，排序为指数复杂度。

4.2优化

原排序中的用于分割的标杆数据一般是取数组第一个元素。

第一个优化点：在于将不选择第一个元素为哨兵，使用随机元素。由于标杆数据的位置是随机的，那么产生的分割也不会总是会出现劣质的分割。在整个数组数字全相等时，仍然是最坏情况，时间复杂度是O(n^2)。实际上，随机化快速排序得到理论最坏情况的可能性仅为1/(2^n）。所以随机化快速排序可以对于绝大多数输入数据达到O(nlogn）的期望时间复杂度。

第二个优化点：由于快速排序一般需要用到递归操作，而函数调用的开销导致递归效率低。在一个函数调用之前需要准备函数内局部变量使用的空间、传入函数的参数,每次调用函数都需要进行,因此会产生额外开销导致递归效率偏低。因此在递归过程中，对于很小和部分有序的数组，快排不如插排好。当待排序序列的长度分割到一定大小后，继续分割的效率比插入排序要差，此时可以使用插排而不是快排。

插入排序函数：

```
void insertSort(T arr[], int startindex, int endindex)
{
    int tmp = 0;
    int i = startindex + 1;
    int j = i - 1;
    for (i; i <= endindex; ++i)
    {
        tmp = arr[i];
        for (j = i - 1; j >= startindex && arr[j] > tmp; --j)
        {
            arr[j + 1] = arr[j];
        }
        arr[j + 1] = tmp;
    }
}
```

4.3 优化前后对比试验

4.3.1 int 类型正态分布优化前后对比

在 int（正态分布）的数据类型下，无论各数据量与原算法速度相比都有一些提升，在大数据情况下速度提升比较明显，小数据量相对速度比较缓慢，详见**表 17，图 17**

表 17. 快速排序-正态分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0
2048	0.002	0.001	100
4096	0.003	0.003	0
8192	0.008	0.007	12.5
16384	0.023	0.022	4.3
32768	0.073	0.072	1.4
65536	0.259	0.244	5.7

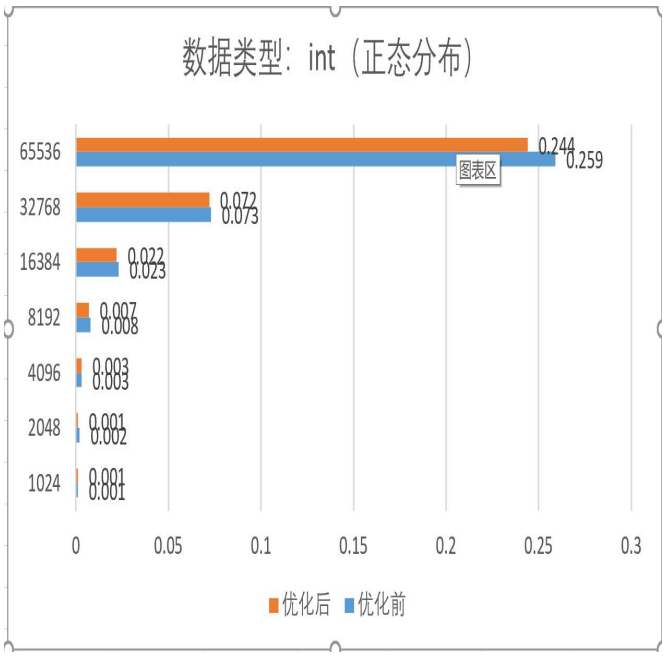

图 17. 快速排序-正态分布整型数据排序时间（秒）

4.3.2 int 类型均匀分布优化前后对比

对于 int 类型均匀分布的数据，在小数据方面优化算法由于递归的原因，速度较为不稳定，在数据规模较大时速度略有提升，总体来说提升较为不明显，有待改进，详见**表 18，图 18**

表 18. 快速排序-均匀分布整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100
2048	0.002	0.001	50
4096	0.002	0.003	-50
8192	0.007	0.006	14.3
16384	0.018	0.016	11.1
32768	0.048	0.05	4.1
65536	0.159	0.168	5.7

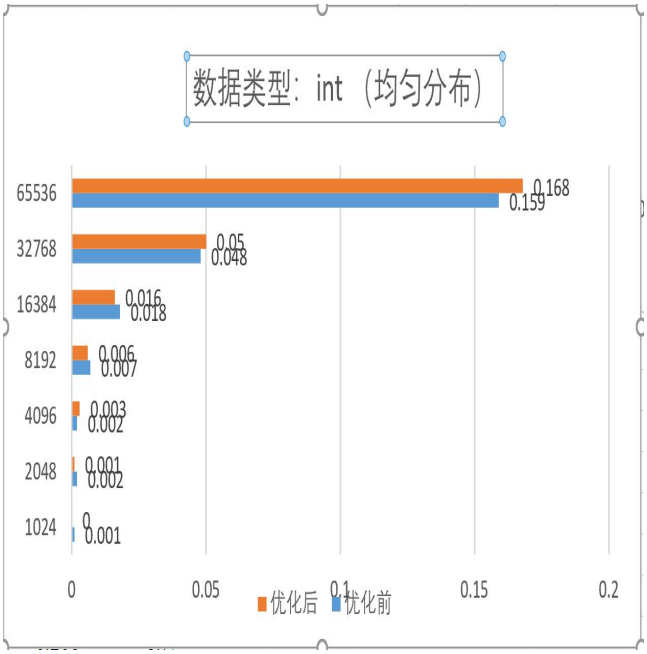

图 18. 快速排序-正态分布整型数据排序时间（秒）

4.3.3 int 类型均匀分布优化前后对比

对于 int 类型均匀分布的数据，小数据较优化前的提升不明显，运算速度比较缓慢，在大数据量下速度稍有提升，快速排序优化更适合在大数据量下发挥一定的优势，详见**表 19，图 19**

表 19. 快速排序-完全顺序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0.001	0
2048	0.005	0.005	0
4096	0.02	0.019	5
8192	0.078	0.078	0
16384	0.299	0.297	6.7
32768	1.179	1.181	1.7
65536	4.72	4.68	8.5

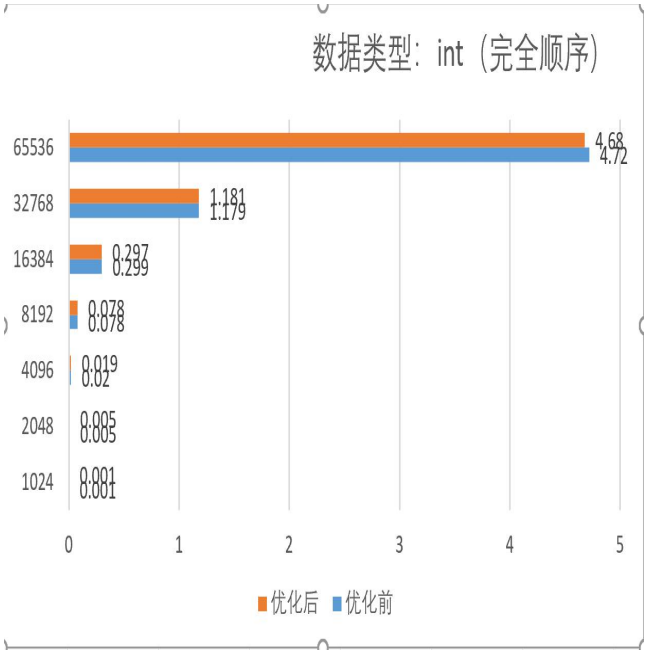

图 19. 快速排序-完全顺序整型数据排序时间（秒）

4.3.4 int 类型完全逆序优化前后对比

对于 int 类型完全逆序的数据而言，各规模数据下提升的速度都微乎其微，优化算法在这一数据类型下的测试并不理想，未能解决快速排序在完全逆序数据下的及其缓慢的问题，详见**表 20，图 20**

表 20. 快速排序-完全逆序整型数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.021	0.023	0
2048	0.083	0.083	0
4096	0.336	0.338	0
8192	1.345	1.333	0
16384	5.395	5.392	0.5
32768	21.685	22.462	0
65536	88.907	86.87	2.3

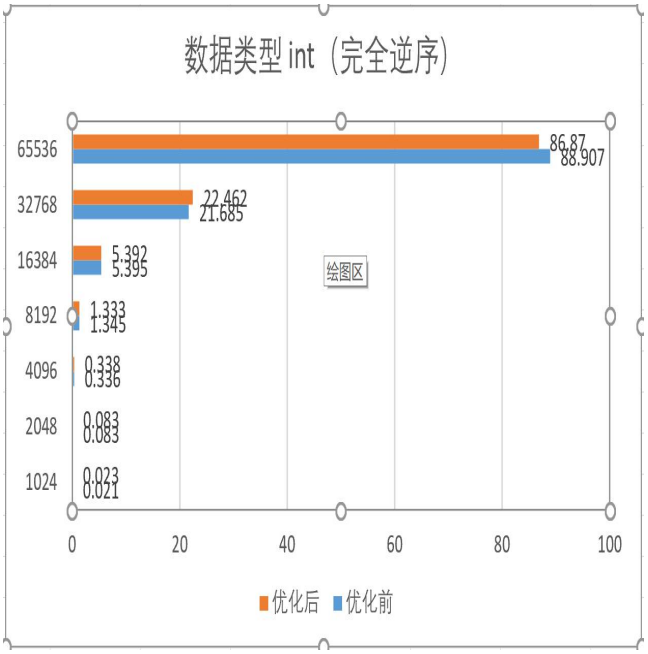

图 20. 快速排序-完全逆序整型数据排序时间（秒）

4.3.5 double 类型正态分布优化前后对比

对于 double 类型正态分布数据而言，在小规模数据下速度提升最为显著，而在大规模数据下优化算法也相对加快了运算速度，详见**表 21，图 21**

表 21. 快速排序-正态分布 double 数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100
2048	0.002	0.001	50
4096	0.003	0.003	0
8192	0.008	0.007	12.5
16384	0.016	0.015	6.3
32768	0.043	0.039	9.3
65536	0.066	0.066	0

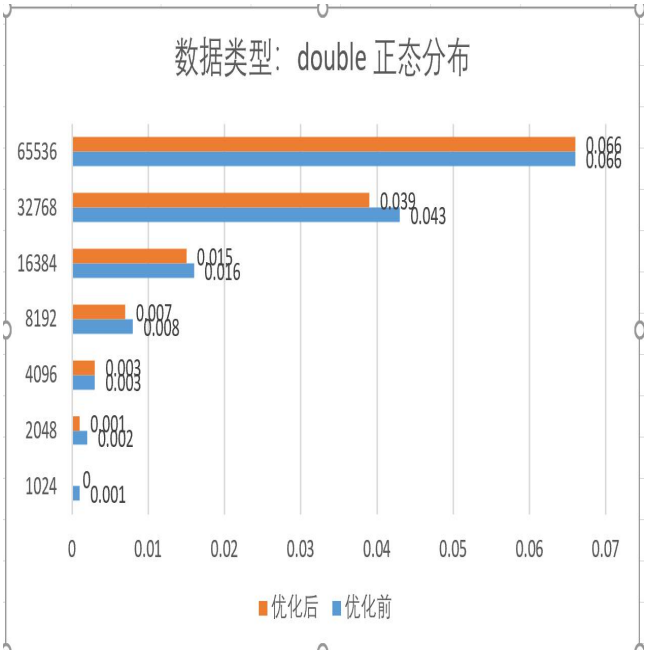

图 21. 快速排序-正态分布 double 数据排序时间（秒）

4.3.6 double 类型均匀分布优化前后对比

对于 double 类型均匀分布数据而言，在小规模数据下速度提升最为显著，而在大规模数据下优化算法的速度也相对提升，优化算法在该数据类型下运作理想，详见**表 22，图 22**

表 22. 快速排序-均匀分布 double 数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.001	0	100
2048	0.002	0.001	50
4096	0.004	0.003	25
8192	0.008	0.007	12.5
16384	0.016	0.014	12.5
32768	0.032	0.03	6.2
65536	0.068	0.065	4.4

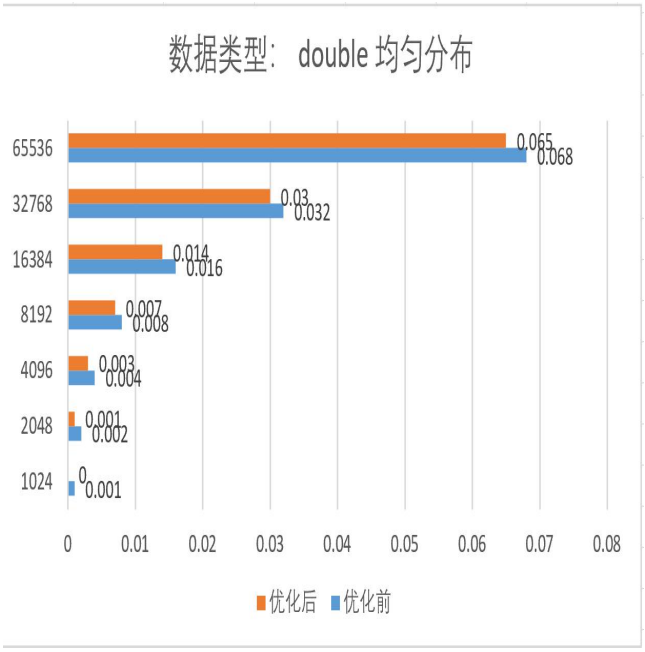

图 22. 快速排序-均匀分布 double 数据排序时间（秒）

4.3.7 double 类型完全正序优化前后对比

对于 double 类型完全正序数据而言，在各规模数据中速度略有提升，但不明显，优化算法的运行效果不如 double 类型均匀分布的数据，详见**表 23，图 23**

表 23. 快速排序-完全正序 double 数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0. 002	0. 001	50
2048	0. 005	0. 005	0
4096	0. 02	0. 019	5
8192	0. 082	0. 077	6. 9
16384	0. 307	0. 307	0
32768	1. 222	1. 217	0. 4
65536	4. 852	4. 832	0. 4

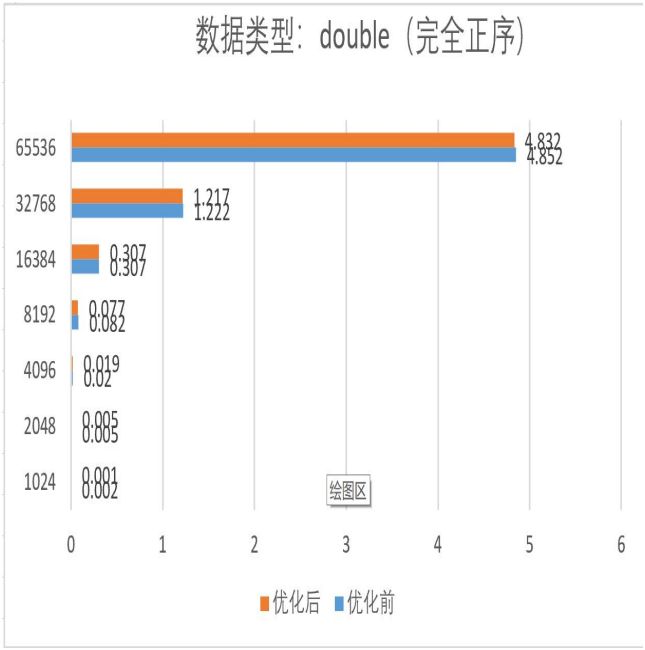

图 23. 快速排序-完全正序分布 double 数据排序时间（秒）

4.3.8 double 类型完全逆序优化前后对比

对于 double 类型完全逆序数据而言，在各规模数据中速度略有提升，但不明显，优化算法的运行效果不如 double 类型均匀分布的数据，详见**表 24，图 24**

表 24. 快速排序-完全逆序 double 数据排序时间（秒）

数据规模	优化前	优化后	提速百分比/%
1024	0.023	0.022	4.3
2048	0.09	0.088	2.2
4096	0.035	0.348	0.5
8192	1.392	1.394	0
16384	5.527	5.638	0
32768	22.332	22.213	0.1
65536	97.67	89.448	8.4

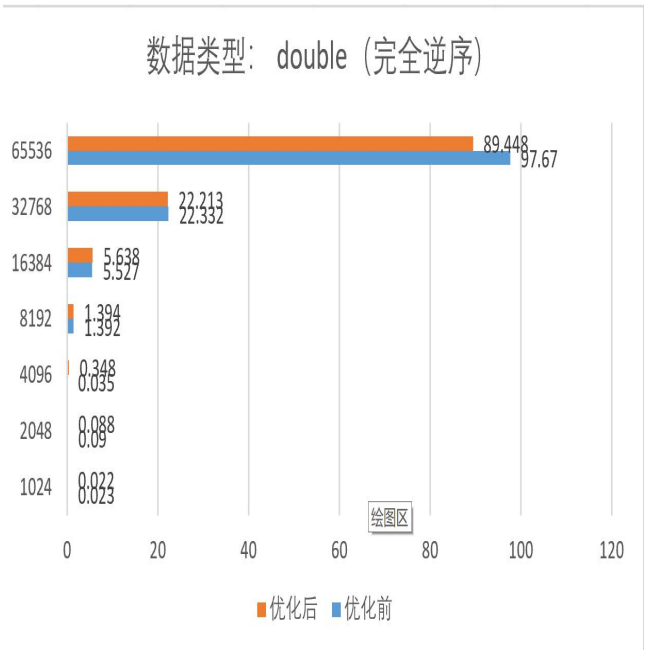

图 24. 快速排序-完全逆序分布 double 数据排序时间（秒）

5 测试不同优化后排序算法对于同种类型、规模数据的分析

5.4.1 int类型数据测试

对于正态分布的整型数据，任意规模的数据都是快速排序最快，选择排序次之，冒泡排序最慢，且在大规模数据集下，缓慢程度较为严重，详见**图 25**，**表 25**。

表 25. 正态分布整型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.002	0	0.001
2048	0.007	0.002	0.001
4096	0.028	0.007	0.002
8192	0.126	0.03	0.003
16384	0.6	0.119	0.01
32768	2.509	0.475	0.034
65536	9.416	1.909	0.114

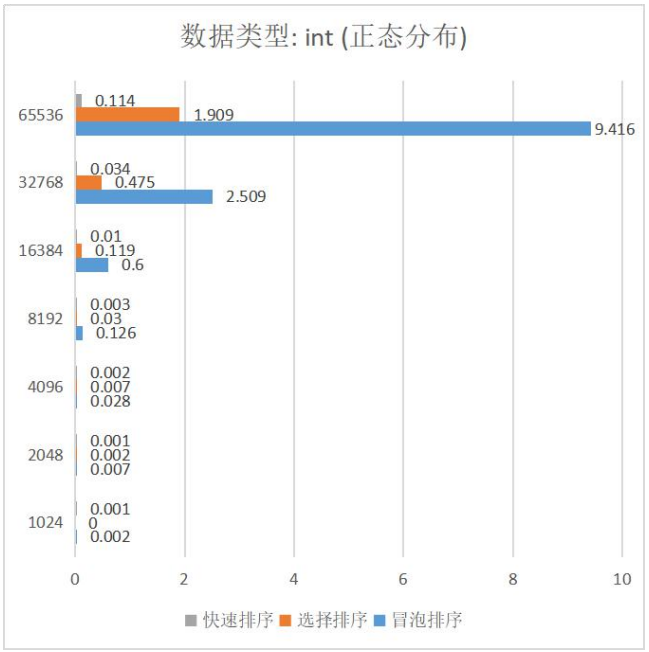

图 25. 正态分布整型数据排序时间（秒）

对于均匀分布的整型数据，任意规模的数据都是快速排序最快，选择排序次之，冒泡排序最慢，且在大规模数据集下，缓慢程度较为严重，但相较于正态分布的数据，速度都有 10%左右的加快，详见**图 26，表 26。**

表 26. 均匀分布整型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.002	0.001	0
2048	0.006	0.002	0.001
4096	0.025	0.008	0.001
8192	0.113	0.03	0.003
16384	0.528	0.118	0.009
32768	2.252	0.473	0.024
65536	8.842	1.902	0.075

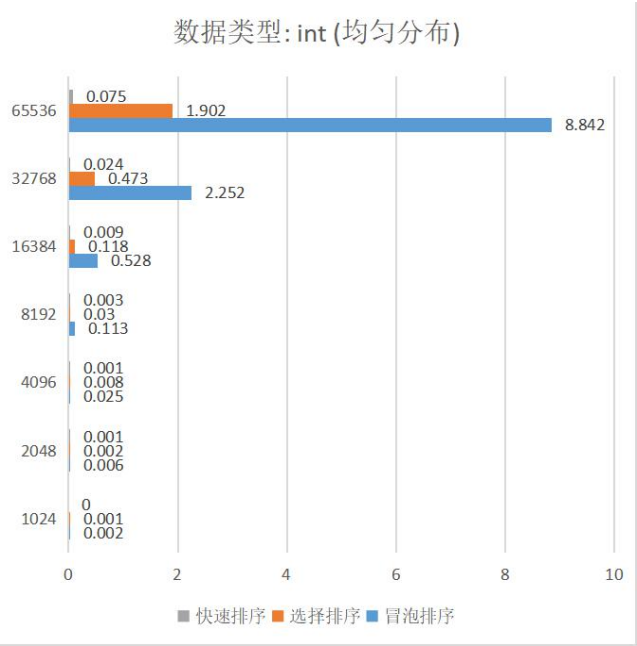

图 26. 均匀分布整型数据排序时间（秒）

而在完全顺序的情况下，由于冒泡排序的优化，算法会自动检测是否有序，因此在有序情况下运行时间任意规模数据下几乎为零，而快速排序和选择排序时间相近，且并不缓慢，详见**表 27，图 27**。

表 27. 完全顺序分布整型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0	0	0.001
2048	0	0.002	0.002
4096	0	0.008	0.008
8192	0	0.031	0.034
16384	0	0.126	0.131
32768	0	0.505	0.514
65536	0	2.027	2.041

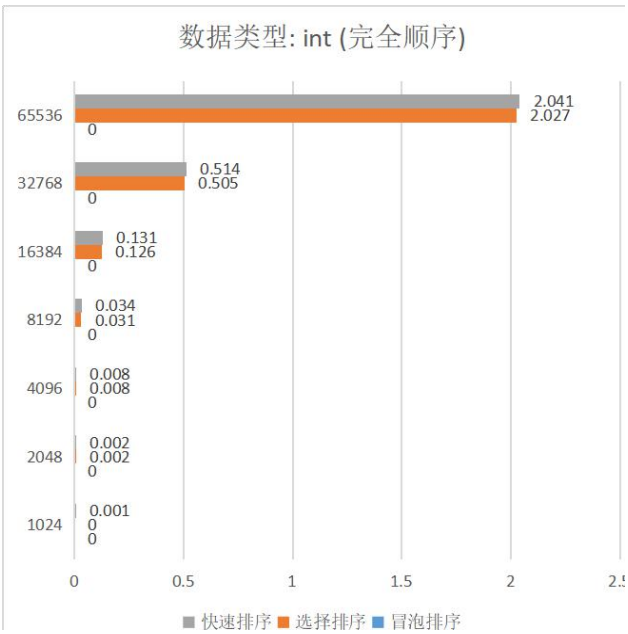

图 27. 完全顺序分布整型数据排序时间（秒）

而在完全逆序下，快速排序递归程度过深的劣势完全体现，大规模数据下运行时间较为缓慢，而最快的则是选择排序，详见**表 28，图 28。**

表 28. 均匀分布整型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.002	0.001	0.009
2048	0.009	0.002	0.039
4096	0.035	0.008	0.155
8192	0.136	0.032	0.624
16384	0.391	0.126	2.475
32768	2.302	0.503	9.899
65536	8.043	2.015	39.636

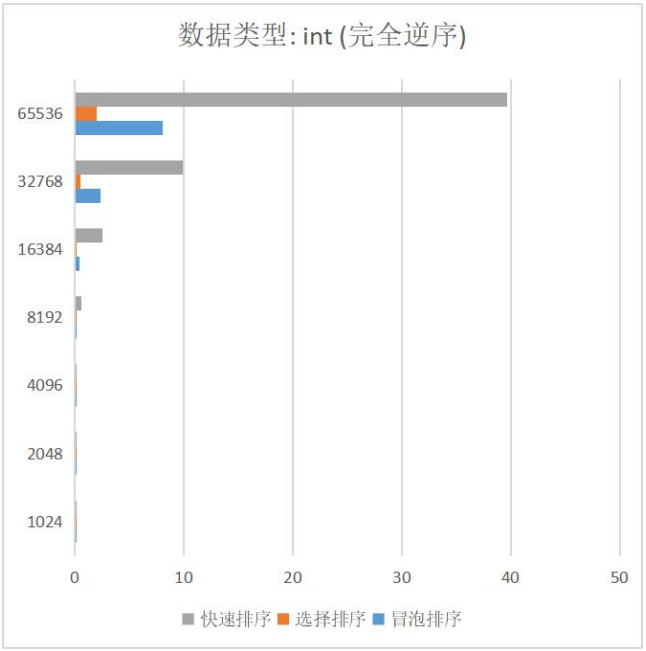

图 28. 完全逆序分布整型数据排序时间（秒）

5.4.2 Double 类型数据测试

正态分布的 double 型数据同整型下一样，速度方面同样是快速排序最快，选择排序次之，冒泡排序最慢，详见**图 29 表 29**。

表 29. 正态分布 double 型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.003	0.001	0
2048	0.008	0.002	0.001
4096	0.035	0.008	0.001
8192	0.156	0.03	0.004
16384	0.756	0.119	0.008
32768	3.259	0.479	0.016
65536	8.391	1.903	0.035

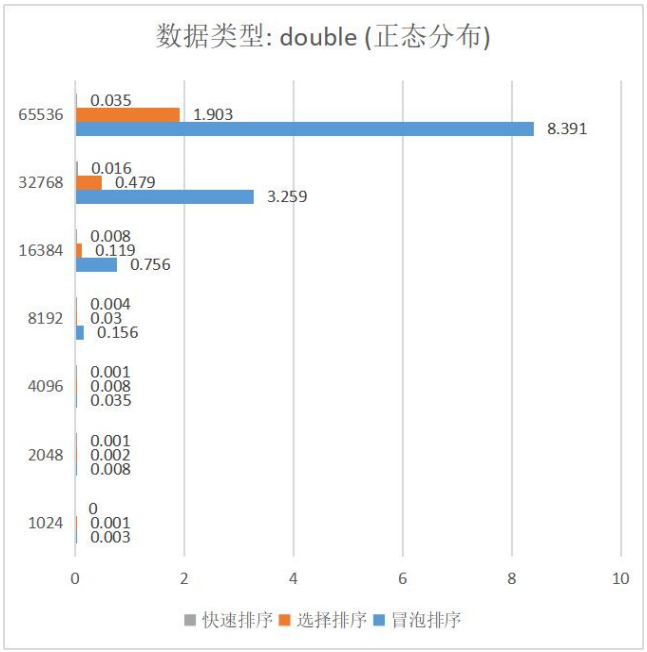

图 29. 正态分布 double 型数据排序时间（秒）

均匀分布的 double 型数据同整型下一样，速度方面同样是快速排序最快，选择排序次之，冒泡排序最慢，详见**图 30 表 30**。相较于正态分布下的 double 型数据则慢了 8.9%左右。

表 30. 均匀分布 double 型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.002	0	0.001
2048	0.007	0.002	0.001
4096	0.031	0.007	0.002
8192	0.13	0.03	0.004
16384	0.667	0.12	0.007
32768	2.853	0.476	0.016
65536	9.567	1.921	0.034

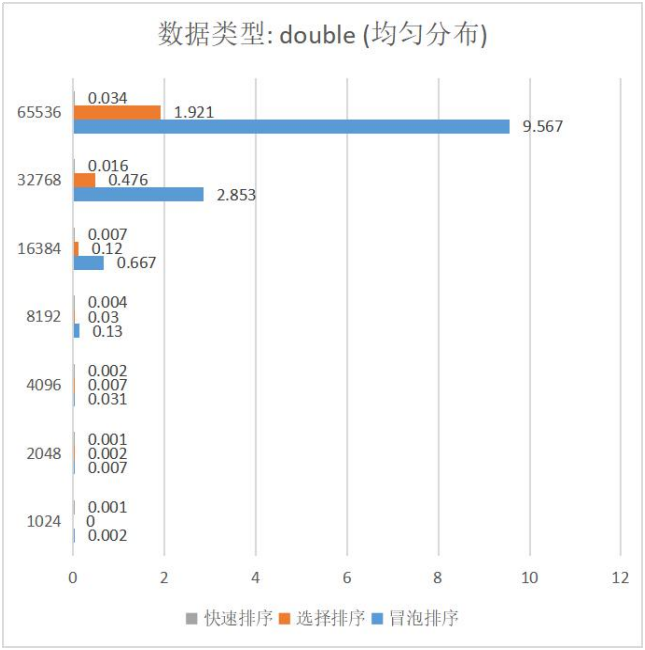

图 30. 均匀分布 double 型数据排序时间（秒）

完全顺序分布下，冒泡排序仍为最优解，选择排序、快速排序次之且相差不多，详见**表 31，图 31**。

表 31. 完全顺序分布 double 型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0	0.001	0
2048	0	0.002	0.003
4096	0	0.008	0.008
8192	0	0.034	0.029
16384	0	0.127	0.117
32768	0	0.514	0.458
65536	0	2.037	1.821

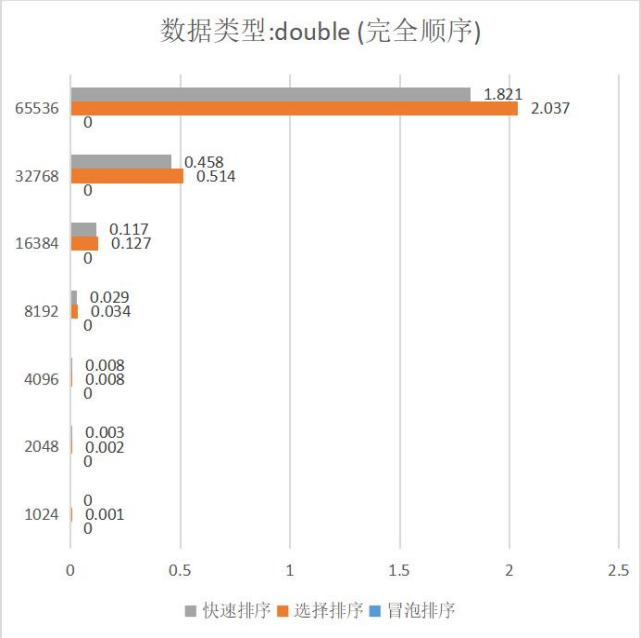

图 31. 完全顺序分布 double 型数据排序时间（秒）

而在完全逆序下，快速排序递归的劣势完全体现，大规模数据下运行时间都较为缓慢，而最快的则是选择排序，详见**表 32，图 32**。

表 32. 完全逆序分布 double 型数据排序时间（秒）

数据规模	冒泡排序	选择排序	快速排序
1024	0.003	0.001	0.01
2048	0.013	0.002	0.039
4096	0.05	0.008	0.157
8192	0.2	0.032	0.629
16384	0.54	0.127	2.508
32768	2.896	0.507	10.005
65536	10.442	2.019	40.046

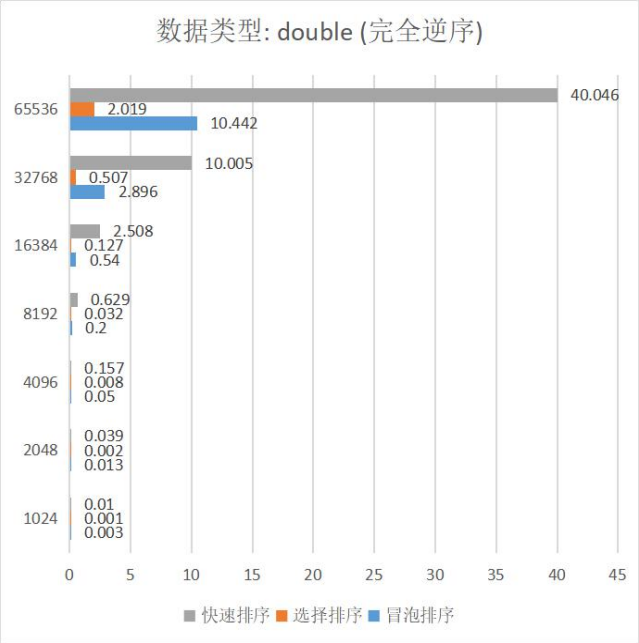

图 32. 完全逆序分布 double 型数据排序时间（秒）

5.4.3 横向比较 int，double 类型数据测试

在正态分布下，冒泡排序在 int 型较小数据集下都是较快的，而在较大数据集下则是 double 型较快；选择排序在 int 和 double 类型在任意数据集下都相差不大；

快速排序则是在 double 型下都相对较快，但也相差不大，详见**表 33**

表 33. 正态分布数据排序时间（秒）

数据规模	int（冒泡）	double	int（选择）	double	int （快排）	double
1024	**0.002**	0.003	**0**	0.001	0.001	**0**
2048	**0.006**	0.008	0.002	0.002	0.001	0.001
4096	**0.025**	0.035	**0.007**	0.008	0.002	**0.001**
8192	**0.113**	0.156	0.03	0.03	**0.003**	0.004
16384	**0.528**	0.756	0.119	0.119	0.01	**0.008**
32768	**2.252**	3.259	**0.475**	0.479	0.034	**0.016**
65536	8.842	**8.391**	1.909	**1.903**	0.114	**0.035**

在均匀分布下，冒泡排序在任意数据集下 int 型都更快；

选择排序则相差不大，小数据集下 double 型较快，大数据集下 int 型较快；

快速排序则是小数据集下 int 型较快，大数据集下 double 型较快，详见**表 34**

表 34. 均匀分布数据排序时间（秒）

数据规模	int(冒泡)	double	int（选择）	double	int （快排）	double
1024	0.002	0.002	0.001	**0**	**0**	0.001
2048	**0.006**	0.007	0.002	0.002	0.001	0.001
4096	**0.025**	0.031	0.008	**0.007**	**0.001**	0.002
8192	**0.113**	0.13	0.03	0.03	**0.003**	0.004
16384	**0.528**	0.667	**0.118**	0.12	0.009	**0.007**
32768	**2.252**	2.853	**0.473**	0.476	0.024	**0.016**
65536	**8.842**	9.567	**1.902**	1.921	0.075	**0.034**

完全顺序下，冒泡排序由于算法会检测是否已经有序因此会最快完成。

选择排序则是 int 型在任意数据集下都较快；

快速排序在小数据集下 int 型较快，而大数据集下 double 型较快，详见**表 31**

表 31. 完全顺序分布数据排序时间（秒）

数据规模	int(冒泡)	double	int（选择）	double	int （快排）	double
1024	0	0	**0**	0.001	0.001	**0**
2048	0	0	0.002	0.002	**0.002**	0.003
4096	0	0	0.008	0.008	0.008	0.008
8192	0	0	**0.031**	0.034	0.034	**0.029**
16384	0	0	**0.126**	0.127	0.131	**0.117**
32768	0	0	**0.505**	0.514	0.514	**0.458**
65536	0	0	**2.027**	2.037	2.041	**1.821**

完全逆序下，所有排序方式都是 int 型较快；

冒泡排序中两种类型速度相差较大；选择排序两种类型相差不大；快速排序大数据集下相差较大，详见**表 35**

表 35. 完全逆序分布数据排序时间（秒）

数据规模	int(冒泡)	double	int（选择）	double	int （快排）	double
1024	**0.002**	0.003	0.001	0.001	**0.009**	0.01
2048	**0.009**	0.013	0.002	0.002	**0.039**	0.039
4096	**0.035**	0.05	0.008	0.008	**0.155**	0.157
8192	**0.136**	0.2	0.032	0.032	**0.624**	0.629
16384	**0.391**	0.54	**0.126**	0.127	**2.475**	2.508
32768	**2.302**	2.896	**0.503**	0.507	**9.899**	10.005
65536	**8.043**	10.442	**2.015**	2.019	**39.636**	40.046

6 字符串的排序及存储空间不同所带来算法不同的分析

字符串的排序中所给两个函数，分别是数组指针和指针数组形式。数组指针是指它指向一个包含 NUM 个 char 类型数据的数组；指针数组指的是该数组包含 NUM 个指向 char 类型数据的指针。

2.1 数组指针 `void BubbleA(char (*str)[NUM], int size)`

对于数组指针类型，通过比较数组中每个元素的首字母来判断大小，即：

```
if (*str[j] > *str[j + 1])
```

在函数中直接定义

```
Char temp[20];
```

作为交换用的变量。

若判断结果正确，则直接交换数组中元素的内容，用 strcpy 函数直接交换，即：

```
strcpy(temp, str[j]);
strcpy(str[j], str[j + 1]);
strcpy(str[j + 1], temp);
```

如果出现前几位字母都相同的情况，则添加判断条件，即：

```
else if (*str[j] == *str[j + 1])
        {
            int flag = 1;
            while (flag != 0)
        }
```

flag 作为标识符，当它非零时不断循环比较，直到最后一位，即：

```
            for (int k = 1; ; k++)
            {
                if (str[j][k] > str[j + 1][k])      //比较首字母后的每一个字母
                {
                    strcpy(temp, str[j]);
                    strcpy(str[j], str[j + 1]);
                    strcpy(str[j + 1], temp);
                    flag = 0;
                    break;
                }
                else if (str[j][k] = str[j + 1][k])
                {
                    continue;
                }
            }
```

2.2 指针数组 `void BubbleB(char *str1[], int size)`

对于指针数组，比较条件仍然相同，即：

```
if (*str1[j] > *str1[j + 1])
```

但在后续处理交换两个元素内容时则不是直接使用 strcpy 函数，**因为字符串常量区的内容不可以重写，只能修改指针**。因此，在函数中定义的作为交换用的变量是

```
char* temp;
```

一个指向 char 类型的指针，用以交换指针变量。

交换方式即：

```
temp = str1[j];
str1[j] = str1[j + 1];
str1[j+1] = temp;
```

若前几位字母相同，则使用相同的逻辑判断，即：

```
else if (*str1[j] == *str1[j + 1])
        {
            int flag = 1;
            while (flag != 0)
        }
```

但交换时同样是交换指针，即：

```
for (int k = 1; ; k++)
{
    if (*(str1[j] + k) > *(str1[j+1] + k))      //比较首字母后的每一个字母
    {
        temp = str1[j];
        str1[j] = str1[j + 1];
        str1[j + 1] =  temp;
        flag = 0;
        break;
    }
    else if (*(str1[j]+k) == *(str1[j+1]+k))
    {
        continue;
    }
}
```

7 MIDI 可视化

3.1 实验过程

1、了解可视化相关的函数，在资料基础上，学习参数的意义以及返回的数据类型

2、与负责算法优化的小组成员进行沟通，了解算法背后的逻辑关系

3、将可视化函数与算法结合，完成可视化

3.2 可视化实现过程

1. 生成内容，例：

```
ShowText(col1 - 4, minIndex + 1, 0, 15, "min");
```

2. 删除内容，例：

```
ShowText(col1 - 4, minIndex + 1, 0, 15, "   ");
```

3.交换内容，例：

```
SWAP(array, start, minIndex);
```

4.内容闪烁，例：

```
ShowChar(col1-6-2*layer, top+a-array, 0, 15, lStr[layer]);
```

3.3 实验中遇到的问题与解决方案

在学习资料中所给的函数时，会遇到大量无法完全理解的函数，我才用控制变量法来学习其中参数的作用，具体方法是：将函数的参数进行简单的修改实验。

例如 showtext 这一函数，第一个参数是列数，第二个参数是行数，第三、四个参数决定颜色，第四个参数决定图案。

通过微调参数可以快速的学会如何使用这个函数。

3.4 实验结果优缺点

优点：将优化后的算法直观的体现在屏幕上

缺点：无法手动撤回到上一步

8 分工及个人心得体会

分工：李昀哲完成字符串的排序及不同存储方式的分析，撰写报告中字符串排序部分及联合测试部分、整合成员报告、整合程序、整合数据；邱逸辰完成快速排序程序编写优化及快速排序部分论文撰写；倪思远完成排序可视化程序及可视化部分论文撰写；丁乐俊完成冒泡排序程序优化及冒泡排序部分论文撰写；杜佳杰完成选择排序程序及选择排序部分论文撰写。

心得体会：

李昀哲：本次实训课程起到了承上启下的作用，不仅帮助我们回忆起了第一学期的代码知识，也很好地起到了为下学期面向对象做准备的作用。实训中能激发我们解决问题的兴趣，外人看代码，看到的只是作品的效率与效果，并不会过多纠结于代码本身，而对于我们程序员来说，做好交互只是其中的一部分，提升代码效率和整体算力才是重中之重。这个课题，对我们团队合作能力是极大的提升，世界不是靠一个人建立的，而是靠千千万万的共同努力构建的，这次的研讨也提升了我们的交流沟通能力，不仅要耐心听取别人的想法，也要善于表达自己的态度，这对于我们以后的职业生涯都是至关重要的。我们有时也不需要自己创造一个世界，而是在原有基础上锦上添花，这也是计算机行业的核心理念，在前人基础上迭代往往会有更好的效果。这次的算法设计多个方面，也让我们懂得有时不需要面面俱到，做好自己的那一部分，并将那一部分做到极致，也就对整个项目都做出了极大的贡献。每个算法都有一定的复杂度，思考方式上也需要一定的思维深度，对我们来说想要全部理解可能将花费较多的时间，因此，每个人的分工合作就能

帮助我们解决这个问题，今后踏入项目开发，我们负责的可能就是一个很小的模块，却也需要和不同部门的同事沟通进度。

邱逸辰：在本次的排序方法编程实训中，我负责的是快速排序算法的优化。本来对于快速排序一知半解的我上网查阅资料，花了不少时间才将基本的算法设计思路搞懂，弄清了快排的递归是怎样进行的。为了优化算法，首先我尝试着自己对其思路进行一些改进，发现没有这个水平后上网查阅资料，对网上提供的优化思路和部分代码进行整合，调试。经过本次实训，我认为一个 IT 从业者并不需要看懂每一行代码，这也几乎做不到。最关键是理解算法每一块的作用，弄清函数的形参，实参，以及函数返回了什么值。本课程使我受益匪浅，多有启发！

倪思远：研究排序算法可视化的过程中，我认为最重要的就是和队友之间的交流，做可视化的过程需要理解他们的算法思想。由于各位组员的代码风格不一，所以更需要面对面的交流，当然代码的规范性也很重要，在以后的工作中，代码如果能加上注释，会减少他人很大的工作量。研究排序算法可视化的过程中，我认为最重要的就是和队友之间的交流，做可视化的过程需要理解他们的算法思想。由于各位组员的代码风格不一，所以更需要面对面的交流，当然代码的规范性也很重要，在以后的工作中，代码如果能加上注释，会减少他人很大的工作量。

丁乐俊：通过暑假的编程实训，我还是受益匪浅的，一来我重新拾起了已经生疏半年的编程，才让我明白：身为一名计算机学院的学生，码代码排 bug 的熬夜生活才是真实的快乐；二来在室友的通力合作下，进一步让我体会到那种合作的快乐以及温暖的同学情谊。分流的学生已经在开小灶全力刷 OJ 了，我们直招的学生也不能甘居人后，得开始通过编编代码来找回写感觉了！

杜佳杰：通过夏季学期为期两周多的短暂的学习,使得我慢慢地找回了编程的感觉。由于长达两个学期没有专业课，对于高级程序语言上学习的一些知识已经有了不少遗忘，在自己实践算法优化的过程中也稳习了那些遗忘的知识。

由于第一学期没有和室友选择到一门课以及刚进学校对一切还略感陌生，第一学期的高级程序语言的小组大作业因为人员陌生与交流方式不同等原因，没有办法真正切身地融入。在这次夏季学期的实践课程中，除了实践能力的提高，我还知道了如何更好地团队分工，如何更好地陈述自己的观点，如何更好地团队合作.....

同时这几周的学习也使得我意识到了一些知识的遗漏，包括遗忘的与之前就比较薄弱的，也提醒着我暑假需要更加努力，提升自己。

致谢 感谢本小组同学的工作和积极参与。同时也感谢张景峤老师悉心的指点和授课。感谢上海大学计算机学院开设的夏季实践课培养我们的不畏艰难困苦的精神和团队意识！

参 考 文 献

[1] https://blog.csdn.net/Ciellee/article/details/107532171

[2] https://www.cnblogs.com/jyroy/p/11248691.html

[3] https://blog.csdn.net/qq_41431406/article/details/84452453?utm_source=app&app_version=4.10.0