四种排序算法的优化与可视化

**摘 要** 归并排序和快速排序都适合处理大数据，在面对“混乱度”较低的数据时均表现出更高效率，但归并排序是稳定排序而快速排序不是。冒泡排序和选择排序都不太适合处理大数据，其中冒泡排序具有稳定性而选择排序不具稳定性。我们对上述的排序方式进行了优化和验证。此外，我们还调用了原函数中的ShowText，SWAP，ShowBars等函数来实现排序的可视化。

**关键词** 排序 优化 可视化

Optimization and visualization of four sorting algorithms

**Abstract** Both merge sort and quick sort are suitable for dealing with big data, and they are more efficient in the face of data with low "confusion". But merge sort is stable sort, but quick sort is not. Bubble sort and selection sort are not suitable for big data processing. Bubble sort is stable while selection sort is not. We optimize and verify the above sorting method. In addition, we also call the original functions such as Showtext, SWAP, ShowBars to realize the visualization of sorting.

**Key words** sort optimize visualization

1 引言

排序就是对元素集合建立某种有序排列的过程。排序在计算机软件系统设计中占有相当重要的地位，尤其是在事务处理中，经常需要对有关数据进行排序，用以提高检索、调用等操作的效率。

根据关键字递增、递减的顺序，把元素依次排列起来，使任意次序排列的数据表变成按照其关键字进行有序排列的数据表称为排序表。如果在排序表中，任意两个不同的元素拥有相同的关键字，且它们在排序过后的前后相对位置没有发生改变，则称这种排序方式是稳定的，否则就称这种排序方式是不稳定的。

交换排序的基本思想是对排序表中两个元素的关键字进行比较，如果发生逆序则交换两者的位置，直到所有的元素都完成排序为止。冒泡排序和快速排序等都是交换排序。

2 排序及优化

2.1 归并排序

（1）概念：归并排序是一种先通过递归，将一个大的序列数组分解为每一个单独的序列元素后，再两两合并为一个更大的序列，逐层进行，最后得到一个有序的序列的排序算法。

（2）设计思路：依据归并排序的概念，我们不难将归并排序分为三个部分：对左侧序列递归分解，对右侧数列递归分解，合并操作（详见图2.1-1）。同时，在递归中，只有当每一个待合并的序列都有序时，我们才能确保最后的序列也同样有序。因此，只有当递归到序列元素数量为1时，方可退出递归，开始进行合并操作。

*if (left >= right)*

*{*

*return;*

*}*

*int mid = left + (right - left) / 2;*

*\_mergeSort(arr, temparr, left, mid, assignCnt, compareCnt);*

*\_mergeSort(arr, temparr, mid + 1, right, assignCnt, compareCnt);*

*merge(arr, temparr, left, mid, right, assignCnt, compareCnt);*

图 2.1-1 归并排序三大板块及递归出口

在单趟合并过程前，为了节省空间，我们事先创建了一个与原序列相同的辅助序列，并将其传入。在单趟合并过程中，借助传入的参数，我们能够得到需要合并的范围，通过比较该范围内辅助序列中的元素并将其依次赋值给原数列，我们便能够将原序列的某一部分进行排序。通过不断合并、排序，我们最后便能够得到一个有序的序列。

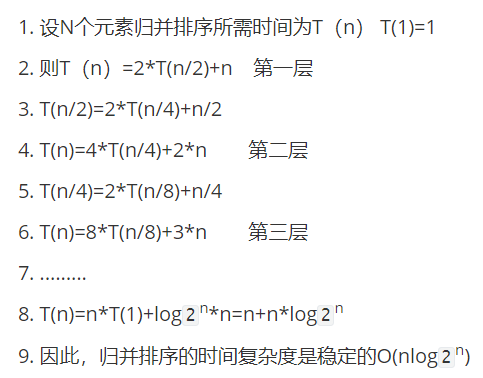


图 2.1-2 归并排序时间复杂度推导

（3）理论分析：在稳定性方面，由于在每次合并过程中，等值元素总是遵循依次放入原序列的规则，维持了等值元素在合并前后的存放顺序，因此，在理论上，归并排序是一种稳定的排序算法。在时间复杂度方面，归并排序算法是一种高效的算法，通过用分治的思想，用空间换时间。通过数学计算，我们可以发现归并排序的时间复杂度为O(nlogn)（详见图2.1-2）。

1. 优化改进：
2. 在小规模数据时终止递归而用插入排序替代。在处理小规模数据时，归并排序由于还要开辟栈区，费时费空间，而插入排序效率更高，因此，在递归到小规模数据时利用插入排序算法替代更深层次的递归，可以减少栈的使用，在防止栈溢出的同时降低空间复杂度（详见图2.1-3）。

*if (left + 10 >= right) {*

*insertSort(arr, left, right, compareCnt, assignCnt);*

*}*

图 2.1-3 小规模数据采用插入排序

1. 对完全顺序进行特判。在创建临时数组前利用bool型变量对完全顺序的数据进行特判，经过试验后发现该特判对其他数据消耗的时间几乎没有影响，而对完全顺序的数据可以大幅度改进，利大于弊。
2. 在递归过程中通过交换原序列与辅助数组参数的传入，去除单趟合并中对原数组到辅助数组的拷贝[1]。为mergeSortPlus和\_mergePlus的参数顺序是相同的， 所以，无论递归过程中辅助数组和原数组的角色如何替换，对最后一次调用的\_mergePlus而言,，最终被排为有序的都是原数组，而不是辅助数组（详见图2.3-4）。

*template <typename T> void \_mergeSort(****T\* arr, T\* temparr****, int left, int right, ULL& compareCnt, ULL& assignCnt)*

*{*

*if (left >= right)*

*{*

*return;*

*}*

*int mid = left + (right - left) / 2;*

*\_mergeSort(****arr, temparr****, left, mid, assignCnt, compareCnt);*

*\_mergeSort(****arr, temparr****, mid + 1, right, assignCnt, compareCnt);*

*merge(****arr, temparr****, left, mid, right, assignCnt, compareCnt);*

*}*

图 2.1-4 交换参数去除辅助数组的拷贝

1. 实际运行&分析：

经过分析发现，经过了上述优化后，归并排序的效率平均提升了大约40%，而在对完全顺序类型的数据进行排序时，由于特判的存在，优化率达到惊人的100%（详见表2.1-1）。同时，我们发现，在优化前后的归并排序算法中，归并排序对完全逆序数据的处理效率要超过正态分布和均匀分布数据，并且赋值和比较的次数也小于他们两个。并且，在数据数量不断增大时，归并排序也展现出了优异的处理能力，并没有发生随着数据量增大消耗时间呈几何倍增长的趋势（详见表2.1-1、2.1-2）。而在稳定性方面，归并排序也正如之前理论分析的那样，呈现出了稳定的特征，未找到不稳定的特例（详见表2.1-3）。

由以上的发现，我们可以知道：归并排序适合应用于处理大数据或需要确保数据的稳定性的场景，在面对“混乱度”较低的数据时，效率会更高。

表2.1-1 优化前后整形数据排序时间（Release配置版）（秒）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型: | 正态分布 | | 均匀分布 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 0.001 | 0 | 0 | 0 |
| 2048 | 0 | 0 | 0 | 0 |
| 4096 | 0 | 0 | 0.001 | 0 |
| 8192 | 0.001 | 0 | 0.001 | 0 |
| 16384 | 0 | 0.001 | 0.001 | 0.002 |
| 32768 | 0.003 | 0.002 | 0.003 | 0.002 |
| 65536 | 0.006 | 0.004 | 0.005 | 0.003 |
| 数据类型: | 完全顺序 | | 完全逆序 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 0 | 0 | 0 | 0 |
| 2048 | 0 | 0 | 0 | 0 |
| 4096 | 0 | 0 | 0 | 0 |
| 8192 | 0 | 0 | 0.001 | 0 |
| 16384 | 0.002 | 0 | 0.001 | 0 |
| 32768 | 0.002 | 0 | 0.002 | 0.001 |
| 65536 | 0.003 | 0 | 0.004 | 0.002 |

表2.1-2 整形数据比较和赋值操作次数（Release配置版）（次）（以优化后为例）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型: | 正态分布 | | 均匀分布 | |
|  | 比较次数 | 赋值次数 | 比较次数 | 赋值次数 |
| 1024 | 25123 | 23752 | 25074 | 23686 |
| 2048 | 56987 | 50952 | 56914 | 50974 |
| 4096 | 120796 | 115325 | 120691 | 115404 |
| 8192 | 268877 | 244793 | 268395 | 244926 |
| 16384 | 565905 | 543776 | 564083 | 543313 |
| 32768 | 1239629 | 1143273 | 1236229 | 1142694 |
| 65536 | 2590324 | 2502024 | 2584262 | 2501654 |
| 数据类型: | 完全顺序 | | 完全逆序 | |
|  | 比较次数 | 赋值次数 | 比较次数 | 赋值次数 |
| 1024 | 13823 | 12800 | 21887 | 23552 |
| 2048 | 31743 | 26624 | 48895 | 50176 |
| 4096 | 65535 | 61440 | 101887 | 112640 |
| 8192 | 147455 | 126976 | 224255 | 237568 |
| 16384 | 303103 | 286720 | 464895 | 524288 |
| 32768 | 671743 | 589824 | 1011711 | 1097728 |
| 65536 | 1376255 | 1310720 | 2088959 | 2392064 |

表2.1-3 从结构体数据看稳定性（65536大小）（Release配置版）（分）

|  |
| --- |
| 学 号 总分 语 数 外 理 化 |
| 00051320 413 88 84 72 84 85 |
| 00058292 412 74 86 84 85 83 |
| 00047306 411 84 86 82 77 82 |
| 00027406 411 77 93 71 86 84 |
| 00039516 409 75 93 76 78 87 |
| 00034325 409 81 76 90 82 80 |
| 00001693 408 73 85 85 86 79 |
| 00062585 407 90 76 80 78 83 |
| 00055381 407 76 83 77 79 92 |
| 00053188 407 73 86 78 78 92 |
| 比较次数：1521734 赋值次数：1540800 |

2.2 冒泡排序

（1）算法原理

冒泡排序，顾名思义数列中的元素会像气泡一样逐渐“浮”到数组某个位置。它重复遍历数组的每一个元素通过比较前后两个元素的大小判断是否发生交换，如果某次遍历没有发生交换则该数组已经有序。

（2）算法稳定性

冒泡排序的比较是相邻两个元素比较，交换也发生在该两个元素之间。相邻且相等两个元素是不会发生交换的；如果相等的元素没有相邻，前一个元素最终也会交换到后一个元素的前一个位置，前后顺序没有发生改变，所以冒泡排序是一种稳定的算法。

（3）算法分析

未优化的冒泡排序

经过第一次遍历后最大的元素已经移动到数组的末端，因此下次遍历应该到数组的倒数第二个位置截止。以此类推，最后一次遍历时只发生一次比较，即可达到有序。未优化的冒泡排序，在原本有序的数组排序时达到最优时间复杂度为O（n）。如果原数组逆序，则每次遍历都要发生n-i次交换，这种情况是冒泡排序最不想看到的，此时时间复杂度为O（n2）。因此未优化的冒泡排序平均复杂度为O（n2）。

一、结束位置的优化

记录每次发生交换的两个元素的较小下标，下次遍历在标记点结束。这种方法可以避免在数组末已经有序但是还在比较的情况，可以减少循环次数来节省时间。实际情况在int类数组下，该方案整体在时间上并不能优化，反而增加了排序时间，虽然赋值次数没有变化，但是比较次数却能明显减少。更加诡异的是double类数组该方案却能优化26%，限于专业知识我并不能解释这一种情况。完全顺序下无论是int还是double或是结构体类型的数据，优化后都明显减少排序时间，几乎瞬间完成，从理论和赋值次数中都可以看出只遍历一次就结束（详见表2.2-1、2.2-2）。

表. 2.2-1 整形数据时间表（Release版）（次）（以优化后为例）

表2.2-2 整形数据比较操作次数（Release版）（次）（以优化后为例）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据构型: | 正态分布 | | 均匀分布 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 523776 | 522186 | 523776 | 522212 |
| 2048 | 2096128 | 2086278 | 2096128 | 2093350 |
| 4096 | 8386560 | 8366155 | 8386560 | 8373947 |
| 8192 | 33550336 | 33535989 | 33550336 | 33496202 |
| 16384 | 134209536 | 134144937 | 134209536 | 134021395 |
| 32768 | 536854528 | 536776961 | 536854528 | 536387087 |
| 65536 | 2147450880 | 2146915514 | 2147450880 | 2145489237 |
| 数据构型: | 完全顺序 | | 完全逆序 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 523776 | 1023 | 523776 | 523776 |
| 2048 | 2096128 | 2047 | 2096128 | 2096128 |
| 4096 | 8386560 | 4095 | 8386560 | 8386560 |
| 8192 | 33550336 | 8191 | 33550336 | 33550336 |
| 16384 | 134209536 | 16383 | 134209536 | 134209536 |
| 32768 | 536854528 | 32767 | 536854528 | 536854528 |
| 65536 | 2147450880 | 65535 | 2147450880 | 2147450880 |

二、双向优化

通过正向和逆向遍历一次将最大和最小分别放到数组的尾和首[2]。同时结合第一种方案，分别记录正向和逆向交换的最后一次交换位置，将左右边界跳至记录点。经过实验发现该方案对于int、double和结构体类型的数组时间都能优化30%左右，在完全顺序时最优时间复杂度为O（n），几乎瞬间完成。而在完全逆序时情况最差，每次都要交换n-i次,不仅不能优化反而增加了10%的时间（详见表2.2-3、2.2-4）。

表 2.2-3 整形数据时间表（Release版）（次）（以正态分布例）

表 2.2- 4 整形数据比较操作次数（Release版）（次）（以优化后为例）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据构型: | 正态分布 | | 均匀分布 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 523776 | 337529 | 523776 | 345508 |
| 2048 | 2096128 | 1352696 | 2096128 | 1395261 |
| 4096 | 8386560 | 5542541 | 8386560 | 5424699 |
| 8192 | 33550336 | 21882826 | 33550336 | 22266233 |
| 16384 | 134209536 | 87845075 | 134209536 | 88126671 |
| 32768 | 536854528 | 350019503 | 536854528 | 354462341 |
| 65536 | 2147450880 | 1403617009 | 2147450880 | 1417925154 |
| 数据构型: | 完全顺序 | | 完全逆序 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 1024 | 523776 | 1023 | 523776 | 523776 |
| 2048 | 2096128 | 2047 | 2096128 | 2096128 |
| 4096 | 8386560 | 4095 | 8386560 | 8386560 |
| 8192 | 33550336 | 8191 | 33550336 | 33550336 |
| 16384 | 134209536 | 16383 | 134209536 | 134209536 |
| 32768 | 536854528 | 32767 | 536854528 | 536854528 |
| 65536 | 2147450880 | 65535 | 2147450880 | 2147450880 |

在稳定性方面无论是优化前还是优化后都没有发生发生变化，在实验数据中也并不能找的特例证明冒泡排序不稳定这与上述理论分析相一致。而时间上冒泡排序仅能进行简单的优化，无法降低算法的时间复杂度，时间上与数据规模呈现平方倍的差别。

2.3 选择排序

选择排序算法是通过将未排序段的最值与已排序段的端点交换实现的。传统的选择排序在每次扫描完未排序段后得到一个最值，这样效率很低，而一种比较经典的优化是在每次扫描完未排序段后同时获得最大值和最小值。相较于传统的选择排序，优化的选择排序可以更加充分地利用每一次扫描，从而减少扫描的次数。传统的选择排序在扫描完一次后可以使有序段的长度加一，故排列整个数组需要扫描n次；而优化后的选择排序在扫描完一次后可以使有序段长度加二，故排列整个数组只需要扫描⎣n/2⎦次。

在排序的稳定性方面，由于选择排序将最值与端点值进行交换，而端点值的稳定性无法得到保证。为了保证算法具有较高的执行效率以及较小的空间复杂度，我这里的选择排序并未在算法稳定性方面进行优化。

由于每次扫描结束后需要维护两个有序序列，优化后的选择排序在细节上有更多的要求。下面将着重分析优化选择排序的细节问题。在具体说明之前，为了方便起见，我们令left为未排序段的左端点的下标，right为未排序段的右端点的下标，minp表示当前最小值的下标，maxp表示当前最大值的下标，且我们假定要将该序列升序排列。

首先是对最值的假定问题。理论上选择未排序段中的任意位置都可以排序，但为了减少不必要的交换，我们令minp=left，maxp=right。这样，若在一轮扫描结束后minp或maxp不变，则可以直接扩展相应的有序序列，从而减少不必要的交换。

其次是扫描的边界问题。在传统的选择排序中，我们可以直接跳过假定的那个最小值，直接从下一个数开始扫描。而在优化的选择排序中，我们分别对left和right进行假定，则对于left和right，它们分别只对应一种假设，但我们无法确定left和right的相对关系，故我们不能跳过这两个数而必须从头扫描到尾，不能遗漏任何数，否则就会导致排序错误。由于比传统的选择排序多扫描了一次端点值（这里是指left），因此优化的选择排序的比较次数会略高于传统的选择排序。另外还有一个边界问题，那就是何时结束总循环，即循环条件是left<right还是left<=right。显然当left==right时，这意味着未排序段的长度为1，而根据选择排序的性质，未排序段的所有值必然大于等于小段（left左边那一段）的所有值，小于等于大段（right右边那一段）的所有值，显然该未排序数已经满足升序的条件，不必进行任何操作，故为了节省开销应将循环条件设未left<right。

接下来是优化选择排序的关键一步。考虑这种情况：在扫描完一次序列后发现minp!=left且minp!=right，而maxp==left。我们假设先将最小值与left的值交换，然后最大值与right的值交换。如果直接将最小值与left的值交换，最大值与right的值交换则会导致最终交换后，最小值在right位置，而left位置存放的是原先right位置存放的元素且其不是最小值。为了解决这个问题，我们需要额外进行一次判断，使在这种情况下right的值与minp的值进行交换。代码如下：

maxp == left ? swap(a[minp], a[right]):swap(a[right],a[maxp]);

然后是对特殊情况的优化处理。若在扫描完一次序列后，发现序列的最小值在right位置，最大值在left位置。按照上述做法，算法会先将right的值与left的值进行交换（最小值和left交换），然后触发上述的三目运算符，right与right交换（最大值与right交换）。为了消除自己与自己进行的交换，我们可以添加一个条件，只有当最大值和最小值不同时在left和right位置上时才对right进行交换。

经过这样的优化后可以使优化的选择排序的交换次数不大于优化前的选择排序。表2.3-1列出了当数据规模为65536时在各种序列下优化前后的比较和赋值次数。

表 2.3-1 优化前后在65536的数据规模下不同数据分布的比较次数与赋值次数

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 优化前 | 优化后 |
| 正态分布 | 比较次数 | 2147450880 | 2147549184 |
| 赋值次数 | 196578 | 181764 |
| 均匀分布 | 比较次数 | 2147450880 | 2147549184 |
| 赋值次数 | 193488 | 187269 |
| 完全顺序 | 比较次数 | 2147450880 | 2147549184 |
| 赋值次数 | 0 | 0 |
| 完全逆序 | 比较次数 | 2147450880 | 2147549184 |
| 赋值次数 | 98304 | 98304 |

可以看出优化前后的选择排序在相同的数据规模下的比较次数是不变的，且优化后的比较次数略大于优化前的比较次数，符合我们的预期。在赋值次数方面，优化后的赋值次数不超过优化前的赋值次数，当完全顺序和完全逆序的情况下两者相等。

最后是时间复杂度与空间复杂度的分析。优化前后的选择排序都涉及两重循环，我们容易分析得出它们的时间复杂度都为，由于都没有开辟额外的数组空间，它们的空间复杂度都为。对于优化后的选择排序，其扫描次数为⎣n/2⎦且每次的扫描长度都会减2，因此其时间复杂度在常系数上小于优化前的选择排序。但优化后的选择排序开辟了更多的变量，因此其空间复杂度在常量上大于优化前的选择排序。

表 2.3-2 用优化的选择排序排列结构体后降序输出前十名的结果（Release配置版）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **学号** | **总分** | **语** | **数** | **外** | **理** | **化** |
| 00029906 | 415 | 81 | 87 | 83 | 84 | 80 |
| 00002486 | 413 | 85 | 77 | 88 | 82 | 81 |
| 00057417 | 413 | 85 | 81 | 77 | 85 | 85 |
| 00031602 | 410 | 79 | 85 | 77 | 85 | 84 |
| 00000711 | 409 | 84 | 80 | 83 | 86 | 76 |
| 00025067 | 409 | 93 | 74 | 87 | 78 | 77 |
| 00051880 | 409 | 87 | 89 | 90 | 71 | 72 |
| 00013180 | 408 | 83 | 76 | 84 | 91 | 74 |
| 00018340 | 408 | 79 | 73 | 80 | 89 | 87 |
| 00028498 | 408 | 81 | 79 | 82 | 91 | 75 |
| 比较次数：2147549184 | | |  | 赋值次数：184398 | |  |

从表2.3-2可以看出，优化选择排序不具有稳定性，如总分409分的学生随总分减少的方向，学号递增。

表 2.3-3 优化前后的选择排序在用时上的区别（Release配置版）（秒）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 数据规模 | 优化前 | 优化后 |
| 正态分布 | int | 4096 | 0.017 | 0.01 |
| 16384 | 0.257 | 0.177 |
| 65536 | 4.303 | 3.012 |
| double | 4096 | 0.019 | 0.013 |
| 16384 | 0.297 | 0.221 |
| 65536 | 4.76 | 3.48 |
| 均匀分布 | int | 4096 | 0.017 | 0.016 |
| 16384 | 0.308 | 0.21 |
| 65536 | 4.242 | 2.863 |
| double | 4096 | 0.019 | 0.014 |
| 16384 | 0.305 | 0.223 |
| 65536 | 4.722 | 3.47 |
| 完全顺序 | int | 4096 | 0.015 | 0.01 |
| 16384 | 0.257 | 0.179 |
| 65536 | 4.235 | 2.918 |
| double | 4096 | 0.018 | 0.013 |
| 16384 | 0.29 | 0.215 |
| 65536 | 4.73 | 3.475 |
| 完全逆序 | int | 4096 | 0.018 | 0.011 |
| 16384 | 0.284 | 0.197 |
| 65536 | 4.475 | 2.962 |
| double | 4096 | 0.019 | 0.013 |
| 16384 | 0.302 | 0.22 |
| 65536 | 4.97 | 3.866 |

从表2.3-3可以看出，优化选择排序的用时比优化前优化了约30%，数据规模越大差距越明显。同时，可以看出，在完全顺序的情况下优化前后的选择排序都需要花较长的时间排序，没有由数据的特殊性使排序时间发生显著变化。

2.4 快速排序

（1）概念

快速排序是一种将一个大的序列数组分拆为2个小的序列数组，其中一个数组中的所有关键字都大于（小于）另一个数组，并通过递归依次逐渐将数组拆分为单个数组元素，最后得到一个有序序列的排序算法。

（2）设计思路

依据快速排序的概念，我们不难将归并排序分为三个部分：将序列数组分拆分一大一小两个部分；继续对小数组进行递归拆分；合并序列数组。同时，在递归中，只有当每一个被拆分出的序列都有序时，我们才能确保最后的序列也同样有序。因此，，只有当递归到的序列元素数量为1时，方可退出递归，开始进行合并操作。（见图2.4-1）

*if (size <= 1) return;*

*Do{...}*

*while (left < right);*

*a[size - 1] = a[left]; a[left] = pivot; // 找到分界点 left*

*Qsort(a, left, compareCnt, assignCnt); // 递归调用(左侧部分)*

*Qsort(a + left + 1, size - left - 1, compareCnt, assignCnt);// 递归调用(右侧部分)*

图 2.4-1 快速排序递归出入口

在快速排序的单趟拆分中，必须选取一个用于比较的数。在课件给出的原函数中，这个数时序列数组的最后一个数。借助此参数与序列数组中的数做逐个比较大小，我们便能够判断某个元素此时应该置前还是置后。

（3）理论分析

在稳定性方面，对于两个等大且均大于参数的数，它们在后置完成时顺序将发生改变。由此可见快速排序是一种不稳定的排序方式。在时间复杂度方面，快速排序是一种高效的算法，通过用分治的思想，用空间换时间。通过数学计算，我们可以发现快速排序的时间复杂度为O(nlog2n)（见图2.4-2）

文本, 信件

描述已自动生成

图2.4-2 快速排序时间复杂度推导

（4）优化改进

对于课件中给出的原算法，如果给出的数组中最后一个元素就是最大元素，那么当次分拆出的新数组将只有一个元素，分拆过程过于缓慢，将会导致程序递归层次过多。新算法中引入了中央位点“mid”，使用数组中的第mid个元素为固定的对照元素“pivot”做比较解决了这一问题。并且在快速排序置前，先将第一个元素，第mid个元素和最后一个元素进行比较，使得拆分后的两个数组的长度尽可能相近。

（5）实际运行和分析

经过分析发现，经过了上述优化后，快速排序的效率较优化之前的算法平均提升了大约40%，而在对完全顺序类型的数据进行排序时，其优化率达到惊人的100%（详见表2.4-1）。同时，我们也发现，优化过后的快速排序对完全逆序数据的处理效率较之优化前提升并不显著，甚至在部分案例中表现出效率下降。此外，在数据数量不断增大时，快速排序也展现出了其时间复杂度较小的优势，随着数据量的增大，其排序消耗时间并没有呈现几何倍增长的趋势（详见表2.4-1、2.4-2）。而在稳定性方面，快速排序也正如之前理论分析的那样，呈现出了不稳定的特征，这与快速排序对优先度相同的数据会因其所处的不同位置而做出不同处理有关（详见表2.4-3）。

由以上的发现，我们可以知道：快速排序适合应用于处理大数据的场景，在面对“混乱度”较低的数据时，效率会更高，但其缺点是缺乏稳定性。

表2.4-1 优化前后整形数据排序时间（Release配置版）（秒）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型: | 正态分布 | | 均匀分布 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 2048 | 0 | 0 | 0.015 | 0 |
| 4096 | 0.015 | 0.016 | 0.016 | 0 |
| 8192 | 0.078 | 0.047 | 0.078 | 0.064 |
| 16384 | 0.313 | 0.245 | 0.322 | 0.22 |
| 32768 | 1.368 | 0.949 | 1.371 | 0.928 |
| 65536 | 5.869 | 3.788 | 5.642 | 3.787 |
| 数据类型: | 完全顺序 | | 完全逆序 | |
|  | 优化前 | 优化后 | 优化前 | 优化后 |
| 2048 | 0 | 0 | 0 | 0 |
| 4096 | 0 | 0 | 0.015 | 0.016 |
| 8192 | 0.047 | 0 | 0.047 | 0.063 |
| 16384 | 0.182 | 0 | 0.203 | 0.229 |
| 32768 | 0.699 | 0 | 0.829 | 0.918 |
| 65536 | 2.789 | 0 | 3.283 | 3.662 |

表2.4-2 整形数据比较和赋值操作次数（Release配置版）（次）（以优化后为例）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型: | 正态分布 | | 均匀分布 | |
|  | 比较次数 | 赋值次数 | 比较次数 | 赋值次数 |
| 4096 | 5499115 | 12022146 | 5612047 | 12345006 |
| 8192 | 21993943 | 48122106 | 22186721 | 48849612 |
| 16384 | 88446587 | 193499319 | 88588355 | 195912780 |
| 32768 | 351493022 | 768654372 | 354267504 | 781692456 |
| 65536 | 1403433755 | 3062225727 | 1423337570 | 3147054987 |
| 数据类型: | 完全顺序 | | 完全逆序 | |
|  | 比较次数 | 赋值次数 | 比较次数 | 赋值次数 |
| 4096 | 4095 | 0 | 8386560 | 25159680 |
| 8192 | 8191 | 0 | 33550336 | 100651008 |
| 16384 | 16383 | 0 | 134209536 | 402628608 |
| 32768 | 32767 | 0 | 536854528 | 1610563584 |
| 65536 | 65535 | 0 | 2147450880 | 6442352640 |

表2.4-3 从结构体数据看稳定性（65536大小）（Release配置版）（分）

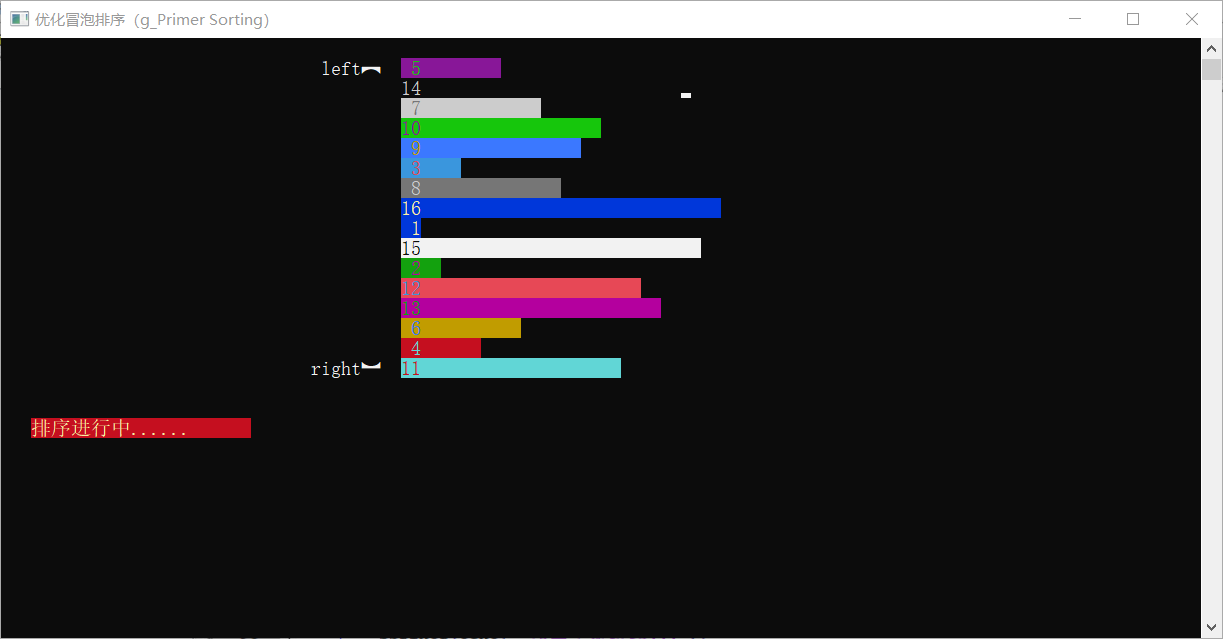
|  |
| --- |
| 学 号 总分 语 数 外 理 化 |
| 00026879 419 82 84 79 86 88 |
| 00061720 418 85 82 93 89 69 |
| 00054749 412 77 81 78 90 86 |
| 00006090 412 87 72 88 87 78 |
| 00054003 412 74 83 82 89 84 |
| 00021604 410 83 84 86 77 80 |
| 00025718 409 79 92 85 75 78 |
| 00004523 409 84 76 81 88 80 |
| 00056629 408 86 72 82 82 86 |
| 00048599 408 81 85 79 83 80 |
| 比较次数：2147450880  赋值次数：6442352640 |

3 排序秀

3.1 优化后冒泡排序可视化

基于优化后冒泡排序的算法，我们调用了原函数中的ShowText，SWAP，ShowBars等函数来实现可视化。

在程序开始先显示数组中的内容，并在将要排序刚时在数组首尾分别显示"︻"和"︼"。当进入while循环时显示左右边界分别用left和right标记。以第一个for循环为例，比较数组前后两个元素的大小，如果需要交换则调用SWAP函数，并刷新right的位置，即原位置用空白代替，新位置显示right标记。由于left和right显示在同一列中，可能会出现相互覆盖的情况，在程序中添加了if语句判断是否需要重新显示被覆盖的内容（此处即left）。如果不需要交换则显示原来的内容。for循环结束后将右边界刷新，原位置用空白覆盖，在right标记处显示"︼"。这里要提到的是，left和right标记与"︻"和"︼"标记并不在同一列中，所以并不需要特别的改动。第二个循环内容和上述类似，此处不再赘述。

图 3.1 冒泡排序秀展示图

3.2 优化的选择排序秀

程序调用了ShowText、ShowBars、SWAP三个函数用于实现排序秀。每次扫描未排序数组前首先调用ShowText在命令提示符上显示待排序数组的范围，其中“︻”和“︼”分别表示待排序数组的上界和下界（不包括字符本身的位置）。然后在“︻”的下方一个单位再次调用ShowText显示“min”表示最小值，在“︼”的上方一个单位调用ShowText显示“max”表示最大值。随着扫描的进行，最小值和最大值都会变化，用“min”和“max”的位置变化来表示当前发现的最小值和最大值。

初始化完成后对未排序数组进行扫描。首先调用ShowBars表面当前位置的数正在与最小值进行比较，同时根据两者的位置关系播放音乐。若判断需要进行更新，则先将“min”位置的所有字符删除，接着判断“max”是否也在原来“min”的位置，如果是，则需要将“max”重新显示到原来的位置。删除完“min”后需要在新位置添加“min”，若新位置上已经有“max”存在，则直接调用ShowText在这个位置显示“max,min”并覆盖原来的“max”表示此时最大值和最小值在同一个位置。若新位置上为空，则直接在上面显示“min”。在更新完“min”的状态后，再次调用ShowBars，表面此时正在和最大值进行比较。若判断最大值需要进行交换，则与“min”同理进行更新操作即可。在扫描完未排序数组后，数组中的最小值旁将显示“min”，最大值旁将显示“max”。

接下来进行交换操作。首先确定最小值是否发生了变化，如果没变则直接进行下一步，如果变了则调用SWAP函数将“min”的值和“︻”下方一个元素的值交换。这样就确保“︻”下方的数是未排序数组的最小值。接着更改“min”的显示格式，将其该字体改为白底黑字以表示“min”已经处理完成。接着处理“max”。若“max”恰好在“︻”的下方一个位置，则将“min”与“︼”上方一个单位进行交换，否则将“max”与“︼”上方一个单位进行交换。

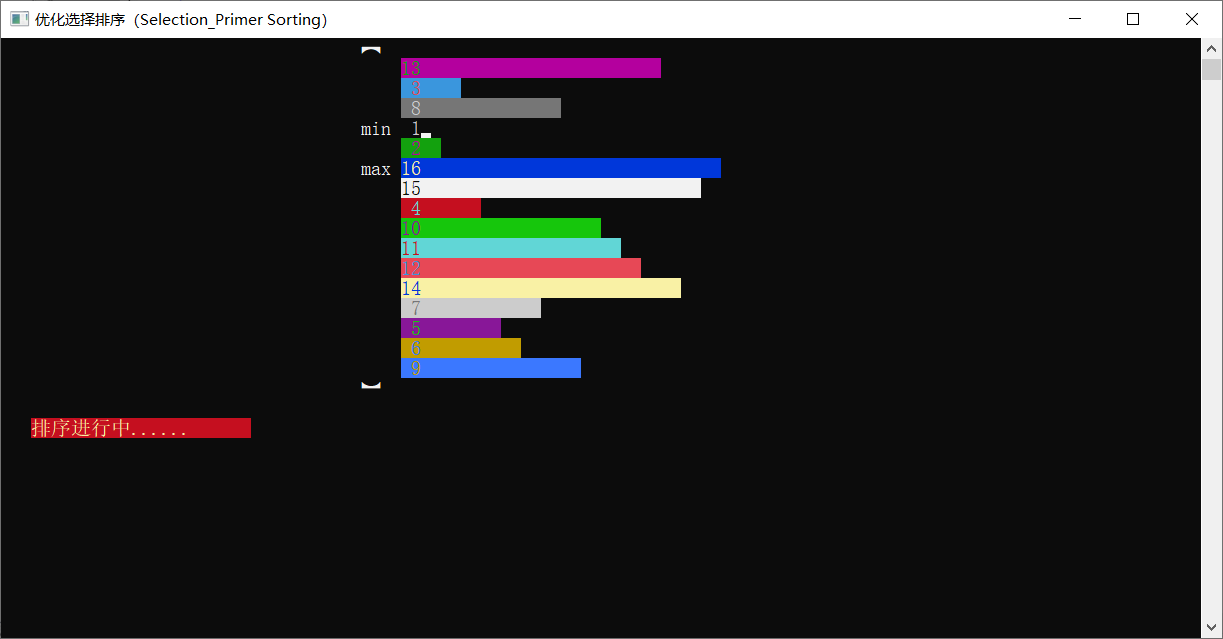
处理完“min”和“max”后，将所有字符清空，调用ShowText在所有字符位置填充空格来清除字符。然后开始下一轮循环。图3.2展示了最终效果。

图 3.2 选择排序秀展示图

4 关于不同存储方式的C-字符串数组排序研究

（1）不同储存方式在内存中的分配：首先，对于一个二维字符串数组（char a[][]）而言，其具有一个首地址，被储存在栈区，其内部的数据分布是连续的(详见图4-1)。而在对其进行初始化后，其中的每一个数据都会被储存在数据区。

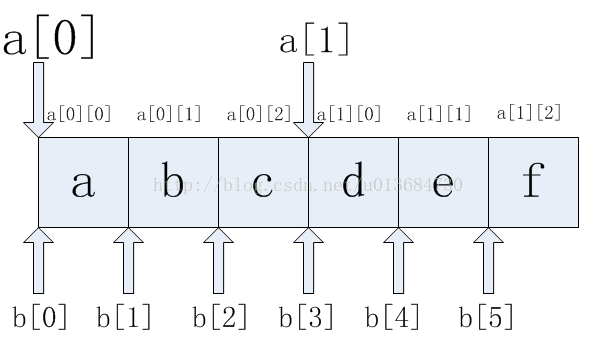


图4-1 二维数组在内存中的分配

其次，对于一个指针数组(char \*a[])而言，由于“[]”的优先度比“\*”高，因此，a先与“[]”结合为一个数组，再与“\*”结合为一个指针类型的数组，其数组中的每一个元素都是一个指针，指向各自分离的内存，其内部并不是连续的（详见图4-2）。

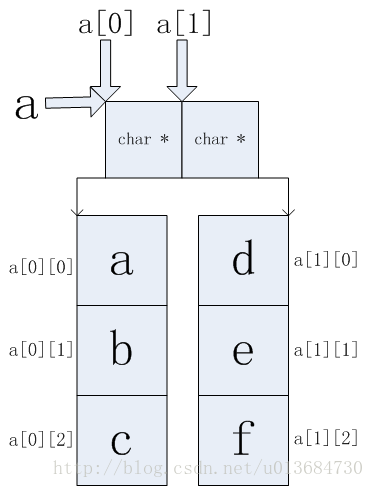


图4-2 指针数组在内存中的分配

接着，对于数组指针(char (\*a)[])来说，由于“( )”的作用，a会首先和“( )”结合为指针，接着指向一个数组，该数组中内存的分布和普通的二维数组一样，都是连续分布的。但值得一提的是，数组指针的“a”与二维数组的“a”并不相同，数组指针的“a”指向的是一个二维数组的地址，而二维数组的“a”指向的却是二维数组的首元素地址。两个“a”虽然都是指针，开辟在栈区，但指向不同。

总之，指针数组比普通的二维数组对空间的利用率更高，普通的二维数组会存在浪费空间的问题。

1. 不同储存方式的操作方法：对于一个普通的二维数组(char a[][]))来说，若想要取得其中第i行第j列的元素，则只需利用a[i][j]即可。

而对于一个指针数组来说(char (\*a)[])，其内部存放的是指针，而为了表示一个二维数组，每个指针都需要指向一个独立的数组，因此，我们可以用一个二阶指针（\*\*b）来进行获取、操作其中的元素。为了获取“第i行，第j列”的元素，一方面，我们可以使用b[i][j]，它指的是第i个存放int \*指针所指向地址中的第j个元素，另一方面，我们也可以直接利用地址——\*(\*(b+i)+j)来操作其中的元素。

最后，对于一个数组指针(char (\*a)[])来说，想要获取其中第i行，第j列的元素，除了在使用时需要“(\*a)[i][j]”外，与普通的二维数组并无区别。

1. 对多个字符串进行字典序排序：在正式排序前，为了后面通过二阶指针进行排序试验，我们需要事先创建两个二阶指针，分别用来拷贝指针数组与普通的二维数组内的内容。

*char strA[][NUM]={"enter", "number", "size", "begin", "of", "cat"};*

*char \*strB[] ={"enter", "number", "size", "begin", "of", "cat"};*

*char \*\*strC, \*\*strD;*

*int n1 = sizeof(strA)/sizeof(\*strA), n2 = sizeof(strB)/sizeof(\*strB);*

*GetStrings(strC, strA, n1);*

*GetStrings(strD, strB, n2);*

图4-3 用二阶指针拷贝

通过GetStrings()模板函数（详见图4-4），面对相似类型的操作时，我们便可以重复使用，而不用再写一个新的函数进行操作，提高了复用性。在该函数中，第一个参数利用传入一个二阶指针的引用，便可以防止在该执行完函数后dest释放导致拷贝无效的问题，而第二个参数，在第一次执行时，将实例化为二维数组的首地址（指向数组内第一个元素——一个数字），而在第二次执行时，将实例化为指针数组首元素的地址（指向数组内第一个元素——一个指针）。（详见图4-3、4-4）。

*template <typename T> void GetStrings(char \*\*&dest, const T \*source, int n){*

*dest = new char\*[n];*

*if(dest == NULL) return;*

*int len;*

*for(int i=0; i<n; i++){*

*len = strlen(source[i]);*

*dest[i] = new char[len+1];*

*strcpy(dest[i], source[i]);*

*}*

*}*

图4-4 GetStrings()函数

正式开始排序时，为了真正改变原有的数组，我们需要传入一个相应的指针或是引用。因此，在面对普通的二维数组时，函数中参数需要设计为一个数组指针，而在面对指针数组和二阶指针时，在函数中，我们既可以把参数设置为指针数组，也可以设置为二阶指针，这两个是等价的。在本次实验中，我们选择将该参数设置为指针数组（详见图4-5）。

*void BubbleA(char (\*str)[NUM], int size);*

*void BubbleB(char\* str[], int size) ;*

图4-5 函数参数设计

此外，为了简化字符串之间的比较，我们还设计了bool类型的cmp函数进行比较（详见图4-6）。

*bool cmp(char\* a, char\* b) {*

*if (a == NULL) return false;*

*if (b == NULL) return true;*

*if (\*a == \*b) return cmp(a + 1, b + 1);*

*return \*a > \*b;*

*}*

*}*

图4-6 cmp函数设计

对于普通二维数组的排序：当两个字符串首元素相等时，利用cmp函数对字符串内部元素继续进行比较，通过返回的bool型数据决定是否交换。若需要交换，由于二维数组每个char类型元素都有相应的地址，因此交换时需要用循环将每个元素依次交换，来达成目的。

而对于指针数组与二阶指针拷贝的数组而言，由于其内部的元素是指针，因此，在交换时仅需要将两个指针的指向交换即可（详见图4-7）。

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

*{*

*for(int i=1;i<size;++i){*

*for(int j=0;j<size-i;++j){*

*if(!cmp(str[j+1],str[j])){*

*char temp[20];*

*strcpy(temp,str[j]);*

*strcpy(str[j],str[j+1]);*

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

*}*

*}*

*}*

*}*

*void BubbleB(char\* str[], int size) {*

*for (int i = 1; i < size; ++i) {*

*for (int j = 0; j < size - i; ++j) {*

*if (!cmp(str[j + 1], str[j])) {*

*char\* temp = str[j];*

*str[j] = str[j + 1];*

*str[j + 1] = temp;*

*}*

*}*

*}*

*}*

图4-7 数组指针与指针数组的交换过程

5 感想

5 总结

在本次课程中，我们小组成员通力合作共同设计探讨了排序的相关问题，实现了排序的优化，也加深了我们对排序的理解。同时，在进行优化的时候我们也遇到了暂时无法解决的问题，如我们的程序在release版本下会出现中断而在debug版本下则可以正常运行，这或许是由release与debug之间的不同导致的，而我们暂时无法解决，希望通过后续学习能解答这一问题。

**致谢** 感谢小组成员十分负责地完成了自己的工作，同时也感谢老师对我们的指导。

**参 考 文 献**

[1] 【算法】归并排序算法的编码及优化 <https://www.cnblogs.com/penghuwan/p/7940440.html>

[2] 冒泡排序及优化 [https://www.cnblogs.com/jeffery336699/p/9306243.html](https://www.cnblogs.com/jeffery336699/p/9306243.html%20)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 小组成员 |  |  |  |  |
| 分工 |  |  |  |  |
| 签名 |  |  |  |  |