**数据结构课程设计**

**-8种排序算法比较**

# 题目描述

随机函数产生一百，一千，一万和十万个随机数，用冒泡排序、选择排序、直接插入、希尔排序、快速排序、堆排序、归并排序和基数排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数，并且显示他们的比较次数。

请在文档中记录上述数据量，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

# 总体思路

根据用户输入的随机数数量生成随机数，储存为源数据。写一个sort类，其中包括了8种排序算法，而每次排序前需要将源数据copy一份，以保证不同排序算法排序使用的是同一份数据，不同的排序对这个拷贝数组进行排序，统计排序时间和交换次数等信息。

# 代码类设计

主函数实现排序算法的选择和随机数的数量输入，还包含了使用计算机硬件计数器实现的排序算法用时记录。

头文件sort.h包含了类sort，其中有8种以排序算法名字命名的成员函数和交换次数、比较次数的输出。

# 具体实现

## 随机数生成

在输入了正确的随机数个数num后，以num为初始参数定义sort类实例。随机数函数使用rand即可，范围为。

## 排序算法用时

添加头文件获取计数器频率tick和初始硬件计数器的计数begin和排序结束后硬件计数器计数end，计数差值再除以频率即排序算法的用时，单位为秒。计时代码如下：

## 冒泡排序

每轮遍历将未排序序列的最大值逐渐向序列末尾移动，算法过程形象的描述为序列按值的大小比喻为水中气泡的大小，在水中逐渐把大气泡冒出水面形成排序序列，因此此算法被称为冒泡排序。代码实现就是对序列进行n-1次遍历，每次遍历从序列开头遍历到最后一个没有排序过元素的索引，遍历时如果下一元素小于当前元素，交换二者。

## 选择排序

顾名思义，每次选择未排序序列中的最小值，排到对应位置。循环n-1次选择最小值，变量i既是循环次数，又对应着每次循环找出的最小值应该在序列中存放的位置，即将最小值的下标和下标为i的元素交换位置。

## 直接插入排序

假设序列左侧已经排序完成，右侧为未排序的元素，插入排序的意思就是向前遍历已排序好的序列，将未排序的元素找到对应位置插入。具体实现其实就是初始步长增量为1的希尔排序。

## 希尔排序

插入排序在对几乎已经排好序的数据操作时效率高，即可以达到线性排序的效率。但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。基于此希尔排序被提出，他对序列所有步长增量为step的子序列进行直接插入排序，step从总序列长度的一半开始，每次减半直至为1。代码实现时，用私有成员函数，有一个参数add，内容为步长增量为add的直接插入排序。直接插入排序可直接调用此函数，参数add为1即可实现。希尔排序在外部循环步长增量step从到1，每次循环调用并传参step。

函数代码实现如下：

## 快速排序

对一个未排序的序列，选定第一个元素为基准值，将序列中所有小于基准值的移至序列左侧，大于基准值的移至序列右侧。再对每个小于区间和大于区间进行快速排序。具体实现为使用两个指针low和high，在一个大while循环中，先从high开始向前遍历直至遇到小于基准值的元素，交换此元素和low指向的元素，low指针加一；再从low开始向后遍历直至找到大于基准值的元素，交换此元素和high指向的元素，high指针减一，循环终止条件为两指针相遇。至此循环结束后序列第一个元素为基准值，然后是小于区间和大于区间。两个指针均指向小于区间的最后一个元素，然后将基准值和指针所指元素交换，继续对小于区间和大于区间做快速排序。

以下是子序列的快速排序：

## 堆排序

在堆的基础上衍生的排序算法，先建立一个大根堆，然后对调根节点和尾结点，从上至下调整大根堆，再对调根节点和倒数第二个节点，重复上述操作，直到对调调整了n-1次。代码编写方面先统一写一个对序列start到end进行向下调整的私有成员函数maxHeap。如果用i代表子树的根节点，那么j=2\*i+1和2\*i+2就是此节点的两个子节点，找到子节点中最大的那个作为j，如果i指向的节点小于j指向的节点，交换i和j。然后令当前根节点为原先j所在位置，继续如上向下调整。

回到堆排序，建大根堆的过程实际上就是对所有非叶子节点到尾结点做向下调整，非叶子节点的调整由最后一个开始往根节点调整。大根堆建立好之后，从尾结点开始依次往前，与根节点交换再以根节点和此节点为范围向下调整，直至此节点和根节点重合。

从start到end向下调整的函数maxHeap代码如下：

## 归并排序

基础思想是将序列分成两份，用两指针遍历两子序列，按序将元素放入一个新数组中完成排序。但只二分一次效率不高，因此对序列进行分治，不断二分直至一个子序列只有一个元素，然后不断对子序列进行上述的合并操作。因此此算法被称作归并排序。

根据算法思想，需要一个用于归并的子私有成员函数merge，参数有序列的开头start、分割位mid和终止end。使用双指针i和j分别从start和mid+1出发，将两指针中指向元素较小的那个添加进合并数组，再自增往前，终止条件为其中一个指针到达该子序列末尾。将未遍历完的子序列剩余元素全部添至合并数组。对于分治，子序列长度rangeLen从1开始到n-1结束，每次循环翻倍，在每次循环中将全序列按的划分标准分治归并，注意mid不能超过n-1，否则构成不了序列的两侧子序列，即右序列为空，就无法正常归并。

对序列进行归并，代码如下：

## 基数排序

对于所有整数的排序，最终结果体现为所有数的最高位呈升序排列，在同一最高位下，次高位仍按升序排序，以此比较至个位最终形成排序。基数排序正是利用这个思想。遍历序列，从个位开始最高位结束，将同位数上数字相同的数添加至对应队列，再从数字小的队列以此出队列，最终到最高位操作完成后就是排好序的序列。最高位由函数maxDigit完成，实现是找到序列最大值，然后返回该数的最大位数。

在具体实现中可能会出现所有数的某位数均相同，这样队列就会很长，使用严格意义上的链表写的队列过于大材小用。我们考虑矩阵转置时添加的定位数组思想：对每个位数不同数字出现的个数先进行统计，这样就可在入队列时定位到相应位置。

我们定义一个大小为n的总队列，和两个大小为10的counter和startPos数组。对每一位进行操作时，先遍历一遍序列，统计每个位数数字出现次数到counter数组中，然后根据counter数组算出。再次遍历序列时，根据位数找到startPos中此数字的起始位置添加到总队列中，对应的startPos中的数加一，这样形成的总队列天然满足队列的先进先出。最终将总队列复制到结果数组中，位数加一。

# 排序算法性能测试

表 一 100个随机数排序结果

|  |  |  |  |
| --- | --- | --- | --- |
| 算法名称 | 时间开销/s | 转移次数 | 比较次数 |
| 冒泡排序 | 0.002953 | 7203 | 4884 |
| 选择排序 | 0.002047 | 291 | 4950 |
| 直接插入排序 | 0.001625 | 2500 | 2401 |
| 希尔排序 | 0.003236 | 966 | 463 |
| 快速排序 | 0.003678 | 301 | 370 |
| 堆排序 | 0.003100 | 1743 | 738 |
| 归并排序 | 0.003896 | 1376 | 559 |
| 基数排序 | 0.003738 | 0 | 0 |

表 二 1000个随机数排序结果

|  |  |  |  |
| --- | --- | --- | --- |
| 算法名称 | 时间开销/s | 转移次数 | 比较次数 |
| 冒泡排序 | 0.004693 | 715338 | 499380 |
| 选择排序 | 0.003422 | 2979 | 499500 |
| 直接插入排序 | 0.002556 | 239445 | 239446 |
| 希尔排序 | 0.003675 | 15256 | 7250 |
| 快速排序 | 0.002795 | 4432 | 6981 |
| 堆排序 | 0.003143 | 27246 | 12489 |
| 归并排序 | 0.002957 | 19968 | 8731 |
| 基数排序 | 0.003054 | 0 | 0 |

表 三 10000个随机数排序结果

|  |  |  |  |
| --- | --- | --- | --- |
| 算法名称 | 时间开销/s | 转移次数 | 比较次数 |
| 冒泡排序 | 0.222018 | 74828388 | 49989114 |
| 选择排序 | 0.104205 | 29964 | 49995000 |
| 直接插入排序 | 0.072978 | 24952795 | 24942796 |
| 希尔排序 | 0.003609 | 262589 | 142584 |
| 快速排序 | 0.003890 | 58855 | 103480 |
| 堆排序 | 0.004364 | 372558 | 174762 |
| 归并排序 | 0.005699 | 272640 | 123696 |
| 基数排序 | 0.002936 | 0 | 0 |

表 四 100000个随机数排序结果

|  |  |  |  |
| --- | --- | --- | --- |
| 算法名称 | 时间开销/s | 转移次数 | 比较次数 |
| 冒泡排序 | 31.436138 | 7498653198 | 4999861169 |
| 选择排序 | 10.841844 | 299949 | 4999950000 |
| 直接插入排序 | 8.117889 | 2499651065 | 2499551066 |
| 希尔排序 | 0.024652 | 4239000 | 2738994 |
| 快速排序 | 0.012596 | 709031 | 1404282 |
| 堆排序 | 0.020862 | 4723755 | 2245294 |
| 归并排序 | 0.040327 | 3385984 | 1566727 |
| 基数排序 | 0.006106 | 0 | 0 |

# 项目小结

1. 三个基本排序算法

冒泡排序、选择排序和直接插入排序都比较基础，代码实现较为简单。由实现思路可知均有两层循环嵌套，因此平均时间开销均为。其中冒泡排序在初始序列已经有序时，只需要遍历一趟就可以完成，比较次数为n。选择排序的排序码比较次数和序列的初始排序无关，总是；而元素移动次数有关，最好情况下可以一次都不移动，最坏情况下移动次数不超过3(n-1)。直接插入排序排序码比较次数和元素移动次数一致，均与序列初始排序有关，最好情况可以只比较n-1次不需要交换元素；而平均和最坏情况都是水平。

这三种排序算法除了使用了一个辅助元素外，无其他额外内存。从稳定性来看，冒泡排序和直接插入排序是稳定的，而选择排序因为选择最小值时可能会打乱原有相等元素的先后顺序，不是稳定的。从测试结果来看，这三种算法在n小于10K时效率没有和其他排序算法有较大差距，甚至在n很小的时候反而效率更高。

1. 快速排序

快速排序是较为通用的内部排序算法，平均情况下时间复杂度为，一般情况下需要额外的空间来存储基准值。但快速排序不稳定，因为划分时前后的相同元素可能会被分到相反的区间。在序列已经有序时，快速排序仍然需要划分递归等操作，时间开销会退化到，空间复杂度也会变为。改进方法可以是取序列首个元素、中间和末尾元素的中间值来尽可能避免。当序列中含有大量重复值时，因为原始的快速排序对等于基准值的元素仍要进行划分，因此可以使用三路划分快排来优化，此优化在小于区间和大于区间外增加了等于区间，可以保证性能最坏情况和不优化的一致。

1. 堆排序

堆排序也是一个高效的内部排序算法，每次对调未排序序列尾元素和根节点元素都需要对堆进行向下调整，总时间复杂度为，且基本不会有什么最坏情况使得堆排序运行明显变慢，且堆排序不需要额外存储空间。但整体平均性能不如快速排序。仍然是因为其不断调整的过程使得此算法不稳定。

1. 归并排序

归并排序也是一个十分高效的内部排序算法，其性能与序列的初始排序无关，总为。但归并排序最大的缺点是归并数组需要的额外内存。因为归并排序过程中没有跨范围的交换元素，因此是稳定的排序算法。

根据实际得到的时间消耗结果和理论分析，可知上述的快速排序、堆排序和归并排序均适用元素个数n很大的情况。

1. 希尔排序

希尔排序的性能分析至今没有精确的数学分析与证明，但Knuth曾通过大量实验统计资料得出，当n很大时，排序码平均比较次数和元素平均移动次数大约在到的范围内，介于基本排序算法和高效排序算法之间，且根据测试可见希尔排序在中等规模的元素序列，即n<1000时，性能表现优越。因为排序过程中对序列子序列进行排序，所以相同元素有可能打乱，希尔排序是不稳定的。

1. 基数排序

基数排序较为特殊，此算法还对排序码的不同部分进行处理和比较。虽然基数排序具有线性增长的时间复杂度，但由于在常规编程环境中，关键字索引统计程序内部循环的操作可能比其他高效算法的内部循环还要多，所以基数排序的时间开销可能不比快速排序好。且基数排序还受到排序码部分的提取算法的影响，适应性不如普通的整体比较和交换操作。

1. 总结
2. 下面的表五是对上述分析的总结：

表 五 8种排序算法的性质

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算法名称 | 平均时间复杂度 | 最坏情况 | 最好情况 | 空间复杂度 | 排序方式 | 稳定性 |
| 冒泡排序 |  |  |  |  | In | 稳定 |
| 选择排序 |  |  |  |  | In | 不稳定 |
| 直接插入排序 |  |  |  |  | In | 稳定 |
| 希尔排序 |  |  |  |  | In | 不稳定 |
| 快速排序 |  |  |  |  | In | 不稳定 |
| 堆排序 |  |  |  |  | In | 不稳定 |
| 归并排序 |  |  |  |  | Out | 稳定 |
| 基数排序 | O(d(n+radix)) | O(d(n+radix)) | O(d(n+radix)) | O(n+radix) | Out | 稳定 |

注：In指原地排序；Out指需要额外空间

1. 以下是用python实现的各排序算法可视化：

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |