

专业课程实验报告

课程名称： 算法分析与设计

开课学期： 2020 至 2021 学年 第 一 学期

专业： 软件工程 年级班级： 1班

学生姓名： 宋行健 学号： 222018321062006

实验教师： 曹严元

计算机与信息科学学院 软件学院

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目名称 | | 排序问题 | | | |
| 实验时间 | 2020年9月17日 | | 实验类型 | | □验证性 □设计性 综合性 |
| 一、实验目的  1. 了解并掌握排序的概念，并熟悉常见的几种排序算法；  2. 掌握常见的几种排序算法的基本思想方法；  3. 对问题实例运用不同的排序算法进行求解，并能比较不同方法的效率好坏和适用的应用场景；  4. 设计并实现排序算法。  二、实验要求  1. 预习实验指导书及教材的有关内容，查阅相关资料，了解不同排序算法的基本思想；  2. 严格按照实验内容进行实验，培养良好的算法设计和编程的习惯；  3. 认真听讲，服从安排，独立思考并完成实验。 | | | | | |
| 三、实验内容与设计（主要内容，操作步骤、算法描述或程序代码）   1. **查阅相关资料，了解有哪些排序算法，掌握常见的几种排序算法的基本思想；** 2. 冒泡排序（Bubble Sort）   它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。  冒泡排序的平均时间复杂度为，最好情况为，最坏情况为，是一种比较稳定且基础的排序算法。   1. 选择排序（Selection Sort）   第一次从待排序的数组中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。  选择排序的平均时间复杂度为，最好情况和最坏情况均为。这是一种不稳定但是简单的排序算法。   1. 插入排序（Insertion Sort）   插入排序的原理很简单，就是遍历待排序的数组，每次将每个元素插入合适的位置。  插入排序的平均时间复杂度为，最好情况为，最坏情况为，是一种比较稳定且基础的排序算法。   1. 快速排序（Quick Sort）   快速排序的基本思想是通过一趟排序将待排序的数组分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。主要用到的是分治的方法。其具体的步骤如下：   1. 从数列中挑出一个元素，称为“基准”； 2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区操作； 3. 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。   快速排序的平均时间复杂度为，最好情况为，即每次选到的基准值都出于待排序的中间。最坏情况为，即每次选到的基准值都出于待排序的极值位置。这是一种不稳定但快捷的排序算法。   1. 归并排序（Merge Sort）   归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。具体步骤如下：   1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列； 2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置； 3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置； 4. 重复步骤 c 直到某一指针达到序列尾； 5. 将另一序列剩下的所有元素直接复制到合并序列尾。     归并排序的平均时间复杂度为，最好情况和最坏情况均为。这是一种稳定的排序算法。   1. 堆排序（Heap Sort）   堆排序是指利用“堆”这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。堆排序可以说是一种利用堆的概念来排序的选择排序。  堆排序的平均时间复杂度和最好最坏的时间复杂度均为。   1. 希尔排序（Shell Sort）   希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法，平均时间复杂度为。  希尔排序是基于插入排序的以下两点性质而提出改进方法的：   1. 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； 2. 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；   希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。  希尔排序   1. 计数排序（Counting Sort）   计数排序的核心在于将输入的数据值转化为键，存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。算法的步骤如下：   1. 找出待排序的数组中最大和最小的元素； 2. 统计数组中每个值为的元素出现的次数，存入数组C的第项； 3. 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）； 4. 反向填充目标数组：将每个元放在新数组的第项，每放一个元素就将减去。 5. 桶排序（Bucket Sort）   桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。为了使桶排序更加高效，需要做到这两点：   1. 在额外空间充足的情况下，尽量增大桶的数量； 2. 使用的映射函数能够将输入的N个数据均匀的分配到K个桶中；   当输入的数据可以均匀的分配到每一个桶中时，排序速度最快；当输入的数据被分配到了同一个桶中，排序速度最慢。但是当桶的容量过小时，其退化成计数排序。     1. 基数排序（Radix Sort）   基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。  基数排序，计数排序和桶排序，这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：   * 基数排序：根据键值的每位数字来分配桶； * 计数排序：每个桶只存储单一键值； * 桶排序：每个桶存储一定范围的数值。   **2. 对下面实例进行排序，用至少5种排序方法进行，写出每种方法的排序过程；**   |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 |  1. 冒泡排序   在第n步中，将第n大的数冒泡到最后，并归位。   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | n | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 1 | 1 | 3 | 1 | 4 | 5 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 9 | | 2 | 1 | 1 | 3 | 4 | 2 | 5 | 5 | 3 | 5 | 6 | 8 | 9 | 9 | | 3 | 1 | 1 | 3 | 2 | 4 | 5 | 3 | 5 | 5 | 6 | 8 | 9 | 9 | | 4 | 1 | 1 | 2 | 3 | 4 | 3 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 5 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 6 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 7 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 8 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 9 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 10 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 11 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 12 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 |  1. 选择排序   在第n步中，将第n小的数与待排序的最小索引交换，并归位。   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | n | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 1 | 1 | 3 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 2 | 1 | 1 | 4 | 3 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 3 | 1 | 1 | 2 | 3 | 5 | 9 | 4 | 6 | 5 | 3 | 5 | 8 | 9 | | 4 | 1 | 1 | 2 | 3 | 3 | 9 | 4 | 6 | 5 | 5 | 5 | 8 | 9 | | 5 | 1 | 1 | 2 | 3 | 3 | 4 | 9 | 6 | 5 | 5 | 5 | 8 | 9 | | 6 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 9 | 5 | 5 | 8 | 9 | | 7 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 9 | 6 | 5 | 8 | 9 | | 8 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 9 | 8 | 9 | | 9 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 10 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 11 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 9 | 8 | 9 | | 12 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 |  1. 插入排序   在第n步中，将第n个数插入到待排序列的合适位置，并归位。   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | n | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 1 | 1 | 3 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 2 | 1 | 1 | 3 | 4 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | | 3 | 1 | 1 | 2 | 3 | 4 | 5 | 9 | 6 | 5 | 3 | 5 | 8 | 9 | | 4 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 9 | 5 | 3 | 5 | 8 | 9 | | 5 | 1 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 9 | 3 | 5 | 8 | 9 | | 6 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 9 | 5 | 8 | 9 | | 7 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 9 | 8 | 9 | | 8 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 9 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 10 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 | | 11 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 9 | 8 | 9 |  1. 快速排序  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | n | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | |  | 从第6个开始分 | | | | | | | | | | | | | | 1 | 1 | 1 | 2 | 3 | 5 | 9 | 9 | 6 | 5 | 3 | 5 | 8 | 4 | |  | 从第8个开始分 | | | | | | | | | | | | | | 2 | 1 | 1 | 2 | 3 | 5 | 4 | 3 | 5 | 5 | 9 | 6 | 8 | 9 | |  | 从第5个开始分 | | | | | | | | | | | | | | 3 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 9 | 6 | 8 | 9 | |  | 从第7个开始分 | | | | | | | | | | | | | | 4 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 9 | 6 | 8 | 9 | |  | 从第11个开始分 | | | | | | | | | | | | | | 5 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 8 | 9 | 9 |  1. 计数排序  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 桶 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | 个数 | 2 | 1 | 2 | 1 | 3 | 1 | 0 | 1 | 2 |  1. **选择你认为最好的一种排序算法，用自己熟悉的编程语言实现。**   我认为最好的排序算法是“归并排序”。它是是采用分治法的一个非常典型的应用。其平均时间复杂度为，而且很稳定。  归并排序C++代码：   1. // 归并排序 2. **int** \*MergeSort(**int** \*array, **int** len) 3. { 4. **int** \*a = array; 5. **int** \*b = **new** **int**[len]; 6. **for** (**int** seg = 1; seg < len; seg += seg) 7. { 8. **for** (**int** start = 0; start < len; start += seg + seg) 9. { 10. **int** low = start, mid = min(start + seg, len), high = min(start + seg + seg, len); 11. **int** k = low; 12. **int** start1 = low, end1 = mid; 13. **int** start2 = mid, end2 = high; 14. **while** (start1 < end1 && start2 < end2) 15. b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++]; 16. **while** (start1 < end1) 17. b[k++] = a[start1++]; 18. **while** (start2 < end2) 19. b[k++] = a[start2++]; 20. } 21. **int** \*temp = a; 22. a = b; 23. b = temp; 24. } 25. **if** (a != array) 26. { 27. **for** (**int** i = 0; i < len; i++) 28. b[i] = a[i]; 29. b = a; 30. } 31. **return** b; 32. } | | | | | |
| 三、测试数据和执行结果 （在给定数据下，执行操作、算法和程序的结果，可使用数据、图表、截图等给出）   1. C++程序代码：   在后面的代码中，我尽己所能实现了第一部分提到的10种排序算法中的7种算法，分别为：冒泡排序、选择排序、插入排序、快速排序、归并排序、希尔排序、计数排序。   1. #include <iostream> 2. #include <stdlib.h> 4. **using** **namespace** std; 6. // 打印数组 7. **void** printArr(**int** \*arr, **int** len) 8. { 9. **for** (**int** i = 0; i < len; ++i) 10. { 11. cout << \*(arr + i) << " "; 12. } 13. cout << "\n" 14. << endl; 15. } 17. // 冒泡排序 18. **int** \*BubbleSort(**int** \*array, **int** len) 19. { 20. **int** temp; 21. **for** (**int** i = 0; i < len; i++) 22. { 23. **for** (**int** j = 0; j < len - 1 - i; j++) 24. { 25. **if** (array[j + 1] < array[j]) 26. { 27. temp = array[j + 1]; 28. array[j + 1] = array[j]; 29. array[j] = temp; 30. } 31. } 32. } 33. **return** array; 34. } 36. // 选择排序 37. **int** \*SelectionSort(**int** \*array, **int** len) 38. { 39. **for** (**int** i = 0; i < len; i++) 40. { 41. **int** minIndex = i; 42. **for** (**int** j = i; j < len; j++) 43. { 44. **if** (array[j] < array[minIndex]) //找到最小的数 45. minIndex = j;               //将最小数的索引保存 46. } 47. **int** temp = array[minIndex]; 48. array[minIndex] = array[i]; 49. array[i] = temp; 50. } 51. **return** array; 52. } 54. // 插入排序 55. **int** \*InsertionSort(**int** \*array, **int** len) 56. { 57. **int** current; 58. **for** (**int** i = 0; i < len - 1; i++) 59. { 60. current = array[i + 1]; 61. **int** preIndex = i; 62. **while** (preIndex >= 0 && current < array[preIndex]) 63. { 64. array[preIndex + 1] = array[preIndex]; 65. preIndex--; 66. } 67. array[preIndex + 1] = current; 68. } 69. **return** array; 70. } 72. // 快速排序子算法——partition 73. **int** partition(**int** \*array, **int** start, **int** end) 74. { 75. **int** pivot = (**int**)(start + 0.5 \* (end - start + 1)); 76. **int** smallIndex = start - 1; 77. swap(array[pivot], array[end]); 78. **for** (**int** i = start; i <= end; i++) 79. **if** (array[i] <= array[end]) 80. { 81. smallIndex++; 82. **if** (i > smallIndex) 83. swap(array[i], array[smallIndex]); 84. } 85. **return** smallIndex; 86. } 88. // 快速排序方法 89. **int** \*QuickSort(**int** \*array, **int** start, **int** end) 90. { 91. **int** smallIndex = partition(array, start, end); 92. **if** (smallIndex > start) 93. QuickSort(array, start, smallIndex - 1); 94. **if** (smallIndex < end) 95. QuickSort(array, smallIndex + 1, end); 96. **return** array; 97. } 99. // 归并排序 100. **int** \*MergeSort(**int** \*array, **int** len) 101. { 102. **int** \*a = array; 103. **int** \*b = **new** **int**[len]; 104. **for** (**int** seg = 1; seg < len; seg += seg) 105. { 106. **for** (**int** start = 0; start < len; start += seg + seg) 107. { 108. **int** low = start, mid = min(start + seg, len), high = min(start + seg + seg, len); 109. **int** k = low; 110. **int** start1 = low, end1 = mid; 111. **int** start2 = mid, end2 = high; 112. **while** (start1 < end1 && start2 < end2) 113. b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++]; 114. **while** (start1 < end1) 115. b[k++] = a[start1++]; 116. **while** (start2 < end2) 117. b[k++] = a[start2++]; 118. } 119. **int** \*temp = a; 120. a = b; 121. b = temp; 122. } 123. **if** (a != array) 124. { 125. **for** (**int** i = 0; i < len; i++) 126. b[i] = a[i]; 127. b = a; 128. } 129. **return** b; 130. } 132. // 希尔排序 133. **int** \*ShellSort(**int** \*array, **int** len) 134. { 135. **int** h = 1; 136. **while** (h < len / 3) 137. { 138. h = 3 \* h + 1; 139. } 140. **while** (h >= 1) 141. { 142. **for** (**int** i = h; i < len; i++) 143. { 144. **for** (**int** j = i; j >= h && array[j] < array[j - h]; j -= h) 145. { 146. swap(array[j], array[j - h]); 147. } 148. } 149. h = h / 3; 150. } 151. **return** array; 152. } 154. // 计数排序 155. **int** \*CountingSort(**int** \*array, **int** len) 156. { 157. **int** i, j, k; 158. **int** max = -10000; 159. **for** (i = 0; i < len; i++) 160. { 161. **if** (array[i] > max) 162. max = array[i]; 163. } 164. **int** \*count\_arr = **new** **int**[max]; 165. **int** \*sorted\_arr = **new** **int**[len]; 166. **for** (k = 0; k < 100; k++) 167. count\_arr[k] = 0; 168. **for** (i = 0; i < len; i++) 169. count\_arr[array[i]]++; 170. **for** (k = 1; k < 100; k++) 171. count\_arr[k] += count\_arr[k - 1]; 172. **for** (j = len; j > 0; j--) 173. sorted\_arr[--count\_arr[array[j - 1]]] = array[j - 1]; 174. **return** sorted\_arr; 175. } 177. **int** main() 178. { 179. **int** arr[13] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9}; 180. **int** len = **sizeof**(arr) / **sizeof**(**int**); 181. **int** \*arrSort; // 排序后的数组 183. cout << "Bubble Sort:" << endl; 184. arrSort = BubbleSort(arr, len); 185. printArr(arrSort, len); 187. cout << "Select Sort:" << endl; 188. arrSort = SelectionSort(arr, len); 189. printArr(arrSort, len); 191. cout << "Insertion Sort:" << endl; 192. arrSort = InsertionSort(arr, len); 193. printArr(arrSort, len); 195. cout << "Quick Sort:" << endl; 196. arrSort = QuickSort(arr, 0, 12); 197. printArr(arrSort, len); 199. cout << "Merge Sort:" << endl; 200. arrSort = MergeSort(arr, len); 201. printArr(arrSort, len); 203. cout << "Shell Sort:" << endl; 204. arrSort = ShellSort(arr, len); 205. printArr(arrSort, len); 207. cout << "Counting Sort:" << endl; 208. arrSort = CountingSort(arr, len); 209. printArr(arrSort, len); 210. } 211. 运行结果 | | | | | |
| 四、实验结果分析及总结（对实验的结果是否达到预期进行分析，总结实验的收获和存在的问题等）  通过本次实验，我对排序算法有了更深入、更系统的了解。同时，我选择了用C++语言来实现这些排序算法，对C++中指针的传递、数组的传递方式也进行了复习。  在实验的过程中，我主要参考的是RUNOOD（<https://www.runoob.com/w3cnote/ten-sorting-algorithm.html>）这个网站。其对10中种典型的排序算法进行了详细的介绍，并且总结的十分清晰。同时，我也参照了在学习Java时，自己在博客上整理的一些笔记（<https://blog.csdn.net/SongXJ_01/article/details/105573188>），将之前用Java实现的算法，用C++重构了一遍，也体会到了两种语言之间的差异。相比之下，我个人还是更喜欢C++一些。  对于每个算法的时间复杂度，我也体会到不同的计算方法之间，计算所用的步骤数和复杂度差异很大。仅针对第一部分中提到的10种排序算法，时间复杂度较小的为希尔排序、归并排序、快速排序、堆排序。它们的平均时间复杂度均为。但是除了归并排序，另外三种并不是很稳定。  至于冒泡排序、选择排序、插入排序，它们是3种基础的排序方法，容易理解，空间复杂度低，但是时间复杂度较大，其平均时间复杂度为。  至于基数排序、计数排序、桶排序，这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：   * 计数排序：每个桶只存储单一键值； * 桶排序：每个桶存储一定范围的数值； * 基数排序：根据键值的每位数字来分配桶。 | | | | | |
| 教  师  评  阅 | 实验内容和设计（A-E）： | | |  | |
| 操作过程、算法或代码（A-E）： | | |  | |
| 实验结果（A-E）： | | |  | |
| 实验分析和总结（A-E）： | | |  | |
| 实验成绩（A-E）：  反馈评语： | | | | |