****

远程与继续教育学院

毕业设计（论文）

**题 目： 各种排序算法比较分析**

**指导教师： 卢萍**

**站 点： 杭州华博特专修学校学习中心**

**学 号： W201900025**

**专 业： 计算机科学与技术**

**年 级： 2019春**

**姓 名： 胡小伟**

**2019 年 3 月**

**毕业设计（论文）诚信承诺书**

**本人郑重承诺网络教育 专升本 层次 计算机科学与技术专业的毕业论文《各种排序算法比较分析》的主要观点和思想系本人独立思考完成，并在此申明我愿承担与上述承诺相违背的事实所引起的一切消极后果。**

**签名：**

**2021年 3月15日**

**各种排序算法比较分析**

**Comparison and Analysis of Various Sort Algorithms**

摘要

排序算法是数据结构这门课程核心内容之一。并在这些领域中占有重要的地位。排序算法是程序设计、数据结构、数据库系统、操作系统、软件工程、网络编程以及人工智能等的重要基础，广泛的应用于软件开发、软件工程等各种领域。学习排序算法是为了将实际问题中涉及的对象在计算机中进行处理。本毕业论文对冒泡排序、直接插入排序、简单选择排序、希尔排序、堆排序和快速排序算法进行比较。

为了简化起见，我将所有的测试数据都设置为整数，用不同规模的测试数据做测试比较，数据对象由随机数生成。比较的指标为各个排序算法排序所消耗的时间，以下所有程序都是基于java语言编写实现的，开发工具使用的是IDEA，开发环境使用的JDK1.8。

可以很明显的看到当数据规模不断增加时，各种算法之间的差别是很大的。这四种算法中，快速排序所消耗的时间是最少的。也是最快的一种排序方法，然后是堆排序；插入排序和选择排序次之，最慢的就是冒泡排序，要比希尔排序，堆排序和快速排序所消耗的时间多上几个数量级。

**关键词：** 冒泡排序；直接插入排序；简单选择排序；希尔排序；堆排序；快速排序；

**Abstract**

Sorting algorithm is one of the core contents of data structure course. And occupies an important position in these fields. Sorting algorithm is an important foundation of programming, data structure, database system, operating system , software engineering, network programming and artificial intelligence. It is widely used in various fields such as software development, software engineering and so on. Learning sorting algorithms is to process objects involved in practical problems in a computer. This thesis compares bubble sorting, direct insertion sorting, simple selection sorting, Shell sorting, heap sorting and quick sorting algorithms.

For the sake of simplification, I set all the test data as integers, compare the tests with different scale test data, and the data objects are generated by random numbers. The index of comparison is the time consumed by sorting algorithms. All the following programs are written and implemented based on java language. The development tools use the JDK1.8. used IDEA, the development environment

It is obvious that when the data scale increases, the difference between the various algorithms is very large. among the four algorithms, fast sorting takes the least time. It is also the fastest sorting method, followed by heap sorting; insert sorting and selection sorting are followed by bubbling sorting, which takes several orders of magnitude more time than Shell sorting, heap sorting and quick sorting.

**Keywords:** Bubble sort; Direct insert sort; Simple select sort; Shell sort; Heap

sort;Quick sort

**目 录**

摘要 I**Abstract** II1 绪论 81．1 研究背景和意义 81．2 排序算法的发展和现状 82 冒泡排序 92．1 基本原理 92．2 具体程序设计实现 9

2．3 冒泡排序的优化 11

2．4 时间复杂度分析 12

3 直接插入排序 133．1 基本原理 133．2 具体程序设计实现 143．3 时间复杂度分析 144 简单选择排序 154．1 基本原理 154．2 具体程序设计实现 164．3 时间复杂度分析 17

5希尔排序 175．1 基本原理 175．2 具体程序设计实现 195．3 时间复杂度分析 20

6堆排序 216．1 基本原理 216．2 具体程序设计实现 236．3 时间复杂度分析 26

7快速排序 297．1 基本原理 297．2 具体程序设计实现 327．3 时间复杂度分析 34

8运行和测试 348．1 测试数据生成程序设计 358．2 测试程序模块划分和实现 359 总结 39

结束语 40致谢 40参考文献 41

1 绪论

1.1研究背景和意义

排序是计算机程序设计中的一种重要操作，也是我们生活中经常面对的问题。例如生活中老师会按学生学号点名；班级考试成绩出来会按成绩总分降序依次排列等。简单的来说就是它可以将一个数据元素构成的任意序列，按关键字重新排列成另外一个有序的序列。排序的严格定义是假设含 n 个记录的序列为{R1, R2, ..., Rn}，其相应的关键字序列为{K1, K2, ..., Kn}，这些关键字相互之间可以进行比较，即：在它们之间存在着这样一个关系Kp1 <= Kp2 <= ... <= Kpn，按此固有关系将上式记录序列重新排列为  {Rp1, Rp2, ..., Rpn}的操作称为排序。

排序算法还在整个计算机科学与技术领域被广泛使用，并在这些领域中占有重要的地位。排序算法是程序设计、数据结构、数据库系统、操作系统、软件工程、网络编程以及人工智能等的重要基础，广泛的应用于软件开发、软件工程等各种领域。本人在研究各种排序算法的过程中，对其原理、时间复杂度、适用场景等在不同规模的数据集上做较全面的比较和分析，并以代码和图形的方式动态展示一些经典排序算法的运行过程，目的主要有以下五个方面：做算法研究，提升自己的技术基础深度；培养自己一个良好的程序设计和软件开发习惯；进一步掌握软件开发过程中的需求分析、概要设计、软件编码、测试用例等基本方法和技能；加深对理论知识的理解，提高自己独立分析和解决问题的能力，知识不能仅仅停留在表面上；为后面的软件开发工作服务做准备，后续处理相对应的排序问题可能会有一定帮助。

1.2 排序算法的发展和现状

从计算的一开始，排序问题就吸引了大量的研究，也许是由于尽管它有简单、熟悉的陈述，且有效地解决它的复杂性。在1951年左右，早期排序算法的作者中有贝蒂·霍尔伯顿（Betty Holberton），他曾在ENIAC和UNIVAC上工作。冒泡排序的分析早于1956年。希尔排序(Shell sort),由D.L.Shell在1959年提出，是突破了O(n2)时间复杂度的第一批算法；堆排序(Heap Sort)是由Floyd和Williams在1964年共同发明的，于此同时他们发明了“堆”这个数据结构；快速排序(Quick Sort)最早是由图灵奖获得者Tony Hoare设计出来的，被列为20世纪十大算法之一。到现在21世纪，新的算法仍在发明中，例如现在广泛使用的TimSort可以追溯到2002年，由Tim Peter发明，是一个自适应的、混合的、稳定的排序算法，在现实世界的数据中有着特别优秀的表现；图书馆排序(Library Sort),该算法由Michael A. Bender，MartinFarach-Colton和Miguel Mosteiro在2004年提出，并在2006年首次发布；相信新的排序算法在后面也会出现的越来越多，也值得我们花时间和精力与学习和掌握。

2 冒泡排序

2.1 基本原理

冒泡排序只会操作相邻的两个元素。每次冒泡操作都会对相邻的两个元素进行比较，看这两个元素是否满足所排序大小的关系要求。如果不满足要求，就将这两个元素的位置进行互换，这样每次冒泡操作都会使至少有一个元素能移动到正确的位置，就这样不断重复只到所有元素都排列正确为止。

2.2 具体程序设计实现

下面我用一个具体的案例来演示冒泡排序的整个过程；假设我们要对一组数据86，74，65，52，47，39进行从小到大的排序(下面红色表示两元素刚刚发生比较且位置进行了交换；蓝色表示只进行了比较，没有进行交换)。

初始数据： 86 74 65 52 47 39

第一轮冒泡：第一次比较后 74 86 65 52 47 39

第二次比较后 74 65 86 52 47 39

第三次比较后 74 65 52 86 47 39

第四次比较后 74 65 52 47 86 39

第五次比较后 74 65 52 47 39 86

第二轮冒泡：第一次比较后 65 74 52 47 39 86

第二次比较后： 65 52 74 47 39 86

第三次比较后： 65 52 47 74 39 86

第四次比较后： 65 52 47 39 74 86

第五次比较后： 65 52 47 39 74 86

第三轮冒泡：第一次比较后： 52 62 47 39 74 86

第二次比较后： 52 47 62 39 74 86

第三次比较后： 52 47 39 62 74 86

第四次比较后： 52 47 39 62 74 86

第五次比较后： 52 47 39 65 74 86

依次类推

第四轮冒泡后的排序结果： 47 39 52 65 74 86

第五轮冒泡后的排序结果： 39 47 52 65 74 86

从上面的分析我们可以发现，一个序列中有6个数时，我们使用冒泡排序总共冒泡了五轮，每一轮都比较了五次；那么我们试着推广到一般情况，假设一个序列有n个数据，那么我们进行冒泡排序时，会经历n – 1 轮冒泡，每一轮冒泡会比较n – 1 次，按照这个思路实现的代码如下：

*//元素交换***public static void** swap(**int**[] arr,**int** i,**int** j) {  
 **int** temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
}  
  
*//冒泡排序初级版***public static void** bubbleSort1(**int**[] arr){  
 **for**(**int** i = 0;i<arr.**length**-1;i++){ *//有n个数时，需要冒泡n-1次；(外层循环用来控制冒泡次数)* **for** (**int** j = 0;j<arr.**length**-1;j++){ *//每次冒泡比较n-1次(内层循环用来控制每次冒泡中比较的次数)* **if** (arr[j] > arr[j+1]){  
 *swap*(arr,j,j+1); *//发现前面的数字大于后面的数字，就开始交换* }  
 }  
 }  
}

2.3 冒泡排序的优化

（1）第一次优化

从上面的排序过程中我们发现，随着每次冒泡操作，都会有一个元素移动到正确的位置；在上面的第三轮冒泡操作中，发现在第四次比较时后面部分的序列就已经是正确的排序了，后面的比较也没有再交换位置；说明后面的元素已经达到完全有序，不需要再进行比较了。其实除了第一轮冒泡需要比较n – 1次外，第二轮冒泡其实就只需要比较n – 2次,第三轮冒泡只需要比较n – 3次，依次类推；因此我们可以按照这个思路在前面代码的基础上进行如下优化。

*//冒泡排序第一次优化***public static void** bubbleSortOptimize1(**int**[] arr){  
 **for** (**int** i = 0;i<arr.**length**-1;i++){  
 *//每轮冒泡完成后，都会有一个元素排列到正确的位置，除了第一轮冒泡需要比较n-1次外，  
 //后面的冒泡只需要比较n-1-i次即可,i表示前面已完成的冒泡次数* **for**(**int** j = 0;j< arr.**length**-1-i;j++){  
 **if** (arr[j] > arr[j+1]){  
 *swap*(arr,j,j+1);  
 }  
 }  
 }  
}

（2）第二次优化

现在我们看看前面的冒泡排序是否能进一步优化；假设我们待排序的序列数据是这样的{2，1，3，4，5，6}，除了第一个元素和第二个元素需要交换外，其他都已经是正常的顺序了；但是按照刚刚的程序进行执行的话，i=0时交换了2和1，此时序列已经是有序的序列了，但是算法仍然会将i=1到5以及每个循环中的j循环都执行一遍，尽管没有交换数据，但仍然是做了很多无用的比较，因此我们可以将上面的算法进行如下改进。

*//冒泡排序第二次优化***public static void** bubbleSortOptimize2(**int**[] arr){  
 **if** (arr.**length**<=1) { *//只有一个元素直接返回* **return**;  
 }  
 **for** (**int** i = 0;i<arr.**length**-1;i++){  
 **boolean** flag = **true**; *//增加标记变量区分冒泡中是否交换了元素* **for**(**int** j = 0;j< arr.**length**-1-i;j++){  
 **if** (arr[j] > arr[j+1]){  
 flag = **false**;  
 *swap*(arr,j,j+1);  
 }  
 }  
 **if** (flag){ *//只要是true说明前面冒泡没有交换元素，说明元素都已经是有序，打断循环* **break**;  
 }  
 }  
}

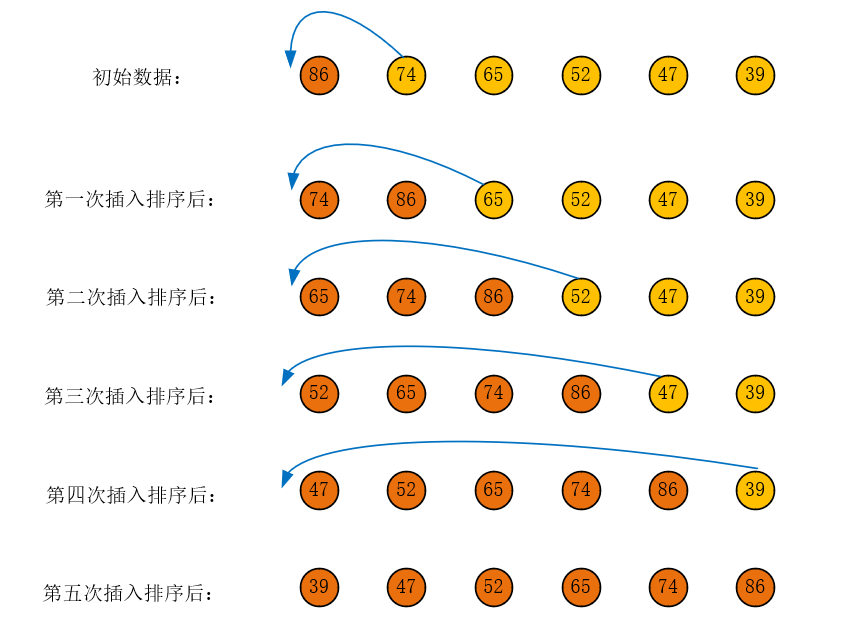
2.4 时间复杂度分析

现在分析一下冒泡排序的时间复杂度；当最好的情况下，即最佳时间复杂度，即已经是有序时，只需进行一次冒泡，n个元素需要执行n – 1次交换，因此最佳时间复杂度是O(n); 当遇到最坏的情况时，最差时间复杂度，即刚好是逆序，n个元素需要执行n – 1次冒泡，比较次数随着冒泡的轮数依次递减形成等差数列；执行交换的元素次数为n – 1 (第一轮冒泡的比较次数) + n – 2(第二轮冒泡的比较次数)+ n – 3(第三轮冒泡的比较次数)+....+1(最后一轮冒泡的比较次数)=n(n-1)/2;因此最差时间复杂度是O(n2)。

3 直接插入排序

3.1 基本原理

直接插入排序的原理很简单，就类似玩扑克摸牌时的理牌一样；首先我们需要将待排序的数据分为两个小序列；已排序序列和未排序序列。插入排序的算法的中心思想就是将未排序序列中的元素取出来，到已排序序列中找到其合适的位置将其插入，在此过程中保证已排序序列的数据一直有序并不断重复这个操作，直到未排序序列中的元素为空为止；下面我还是以86，74，65，52，47，39这组数据为例如图演示插入排序的整个过程。



3.2 具体程序设计实现

从上面的演示图中可以看出插入排序也包含两个操作，一种是元素的比较，另一种是元素的移动，当我们需要将一个元素放入已排序序列时，需要将该元素与已排序序列的元素依次比较大小为其找到正确的插入位置。找到后还需要将插入点之后的元素往后都移动一位，来为该元素提供空间插入，程序设计如下。

*//直接插入排序***public static void** directInsertSort(**int**[] arr){  
 **int** n = arr.**length**; *//获取数组长度* **if** (n <= 1) { *//如果数组长度为1，无需排序，直接返回* **return**;  
 }  
 **int** j,temp;  
 **for** (**int** i = 1;i< arr.**length**;i++){ *//外层循环用来控制插入排序的次数，共需要排序n-1次* **if** (arr[i-1]>arr[i]){ *//如果前面的元素比当前元素大* temp = arr[i]; *//将当前元素放入临时变量* **for** (j = i - 1;j>=0 && arr[j]>temp;j--){ *//内层循环用来控制有序序列元素移动的次数(将有序序列的元素依次与当前元素比较)，这里j>=0为了防止数组越界，* arr[j+1] = arr[j]; *//将有序序列元素往后移动,直到arr[j]<temp* }  
 *//内层循环执行完说明现在的arr[j]<temp，则现在的j+1就是temp当前元素所要插入的位置* arr[j+1] = temp;  
 }  
 }  
}

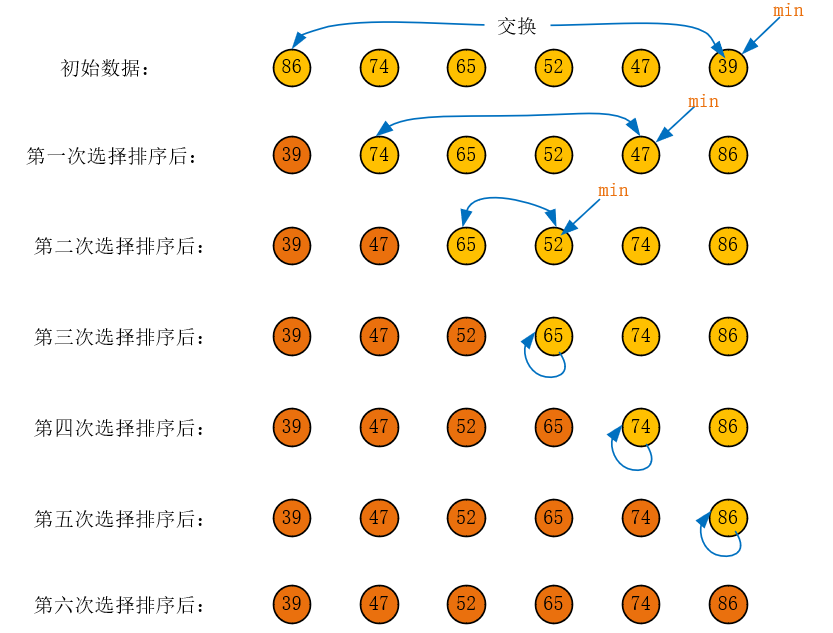
3.3 时间复杂度分析

现在分析一下直接插入排序的时间复杂度；当最好的情况时，也就是要排序的数据本身就是有序的了，这时我们并不需要移动任何的元素。如果我们从尾到头查找插入位置只需要比较一个元素就能确定正确的插入位置；而如果我们从头到尾查找，最多也只需要比较n – 1次就能确定正确的插入位置且没有发生元素移动，因此在这种情况下，最好的时间复杂度是O(n)；当最坏的情况下，也就是要排序的数据是逆序的情况，这时相当于在数组中插入元素，循环执行n – 1次数组插入操作，而数组中插入元素的时间复杂度是O(n)，因此直接插入排序最坏的时间复杂度就是O(n2)。

4 简单选择排序

4.1 基本原理

选择排序算法和之前直接插入排序的算法比较类似，也是将数据分为已排序序列和未排序序列两部分处理。但是选择排序是每次排序时会从未排序序列中找到最小值，再将其最小值放入已排序序列的最后一位并进行位置交换，然后不断重复循环此操作，直到未排序序序列的元素取完为止。具体就是将待排序的一组数据元素中，选出最小的一个数据元素与第一个位置的数据元素交换；然后在剩下的数据元素当中再找最小的与第二个位置的数据元素交换，循环到只剩下最后一个数据元素为止；第一次排序从n个元素中找出最小的元素与第1个元素交换；第二次排序，从第二个元素开始的n – 1个元素中再选出最小的元素与第二个元素交换；如此，在第 i 次排序时，则从第n – i – 1个元素开始的 i + l个元素中选出最小的记录与第 i 个元素交换，直到所有记录排好序，我们仍然使用上面的数据来演示选择排序的整个过程。



4.2 具体程序设计

**public static void** selectionSort(**int**[] arr) {  
 **int** i,j,min;  
 **for** (i=0;i<arr.**length**-1;++i){ *//如果有n个无序的数字，则选出最小值只需比较n-1次* min = i; *//假设当前下标的值为最小值(i就是待交换的)* **for** (j = i+1;j<arr.**length**;++j){ *//内层循环是用来查找最小值* **if** (arr[min]>arr[j]){  
 min = j; *//给最小值重新赋值* }  
 }  
 *//内层循环执行完，说明找到了未排序序列的最小值* **if**(i!=min){ *//说明最小值不是自己，开始交换位置  
 swap*(arr,i,min);  
 }  
 }  
}

**public static void** swap(**int**[] arr,**int** i,**int** j) {  
 **int** temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
}

4.3 时间复杂度分析

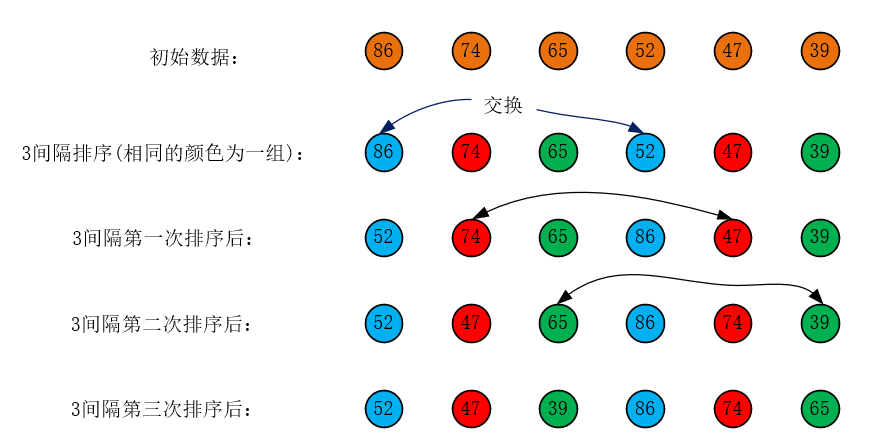
从简单选择排序算法的执行过程来看，它最大的优点就是交换移动移动数据次数相当的少，这样可以节约很多的时间；现在分析下它的时间复杂度，无论简单选择排序的最好情况还是最差的情况，他们的比较次数都是一样的，第i轮简单选择排序需要进行n – i次元素的比较,而整个排序过程需要n-1轮，因此整个简单选择排序的总比较次数为n-1+n-2+…+1=n(n-1)/2次。而对于简单选择排序的元素交换次数；最好的时候，交换次数为0，最差的时候，交换次数为n-1次，因此简单选择排序的时间复杂度是O(n2)。

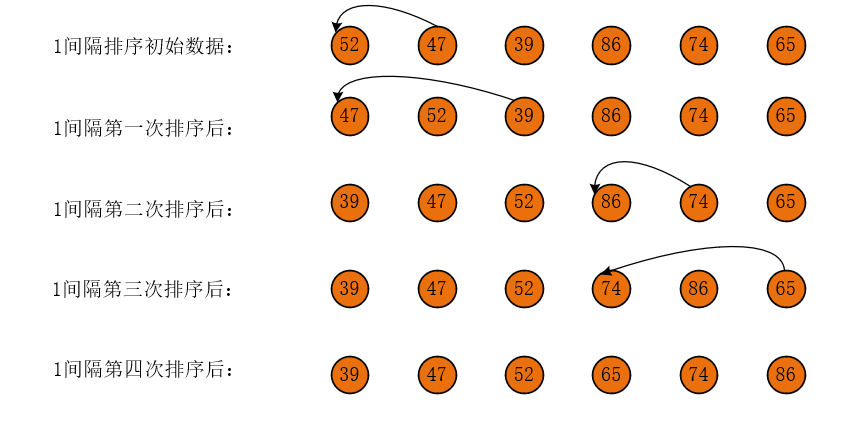
5 希尔排序

5.1 基本原理

现在开始希尔排序，希尔排序是第一批突破算法时间复杂度O(n2)的，因此它在排序的发展历史上还是具有比较重要的意义的；在前面第三节中我讲到了直接插入排序算法，直接插入排序在某些条件下它的效率是很高的，例如，一组数据中自己基本是有序的，这个情况下只会产生少量的插入操作就可以完成整个数据的排序工作。还有一个就是元素个数比较少时，直接插入的效率也很高。因此为了充分发挥直接插入排序的这两项优势，希尔排序就是在直接插入排序的基础上按照这条思路发展出来的。

希尔排序具体是这样操作的，我们将直接插入排序的算法进行改进，将原本拥有大量数据的元素进行分组。将其分割成若干个子序列，这样每个子序列待排序的元素个数就变少了，然后我们在将这些子序列分别进行直接插入排序，完成后整个序列变的基本上是有序了，要提醒下不是完全有序，然后我们再对所有的子序列合并，最后对所有元素进行一次直接插入排序。这样我们就人为创造了直接插入排序的两个条件“基本有序”和“元素个数少”，从而有效的提高了插入排序的效率，下面我演示具体的执行过程。





从上面开始我是每3个间隔分一组，一共将6个元素分成3组，每组子序列包含2个元素，然后将每组子序列进行插入排序，执行完后得到初步有序的序列52，47，39，86，74，65，然后再1间隔进行插入排序，也就变成直接插入排序。从上面的演示可以发现，希尔排序就是在直接插入排序的基础上将相隔某个“增量”的元素组成一个子序列，来实现跳跃式的移动，使得排序效率提高，而且最后一次排序必须时间隔为1，这样才能保证最终排序是正确的，且增量序列必须是递减的；值得注意的是这里“增量间隔”的选取是相当关键的，增量序列选取的好坏直接决定了希尔排序的效率好坏，而到目前为止还没有人能找到一种最好的增量序列；虽然没有找到最好的增量序列，但是前人整理了一批增量序列，这些增量序列都有不错的效率，这些增量序列有：Hibbard增量序列、Knuth增量序列、Sedgewick增量序列等等，下面我以Hibbard增量序列为主来实现希尔排序的程序。

5.2 具体程序设计

**public class** LogarithmUtils {  
  
 */\*\*  
 \** ***@param value*** *真数  
 \** ***@param base*** *底数  
 \** ***@return*** *求以base为底，value的对数  
 \*/* **public static double** log(**double** value, **double** base) {  
 **return** Math.*log*(value) / Math.*log*(base); *//根据对数换底公式 logx(y) =loge(y) / loge(x)* }  
}

*/\*\*  
 \** ***@param arr*** *待排序的数组 (该方法使用的Hibbard增量序列)  
 \*/***public static void** shellSort(**int**[] arr) {  
 **double** log = LogarithmUtils.*log*(arr.**length** + 1, 2); *//根据数组的长度得到接近k的值 2^k-1=length转换成求对数 k = log2(length+1),这一步是重点* **int** k = (**int**)Math.*round*(log); *//四舍五入获取k(使用Hibbard增量序列2^k-1中的k值)* System.***out***.println(**"k值: "** + k);  
 *//获取到k值后面就简单了，只需要进行k次增量插入排序即可* **for** (**int** i=k;i>=1;--i){  
 **int** increment = (**int**)Math.*pow*(2,i)-1; *//计算2^k-1 得到当前的增量  
 //System.out.println(increment);  
 shellInsert*(arr,increment);  
 }  
}  
*/\*\*  
 \* 增量插入排序  
 \** ***@param arr*** *数组  
 \** ***@param increment*** *增量  
 \*/***public static void** shellInsert(**int**[] arr,**int** increment){  
 **int** i,j,temp;  
 **for** (i=increment;i<arr.**length**;++i) {  
 temp = arr[i]; *//注意是将要排序的数字保存到临时变量* j = i-increment;  
 **for** (;j>=0 && temp<arr[j];j=j-increment){ *//如果后面的元素小于前面元素，向后移动，还要注意的是j>=0这个判断条件是为了防止越界，因为temp如果交换的话每次都会前移* arr[j+increment] = arr[j];  
 }  
 arr[j+increment] = temp;  
 }  
}

我解释下上面的程序，LogarithmUtils的log方法是根据数组元素的长度获取Hibbard增量序列的k值，因为Hibbard增量序列的取法为Dk=2^k−1：{1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191...}，因此我用数组的长度通过对数的换底公式求出k值，然后就能用k值得到Hibbard的增量序列了，后面就是用在直接插入排序的基础上增加Hibbard的增量来实现。

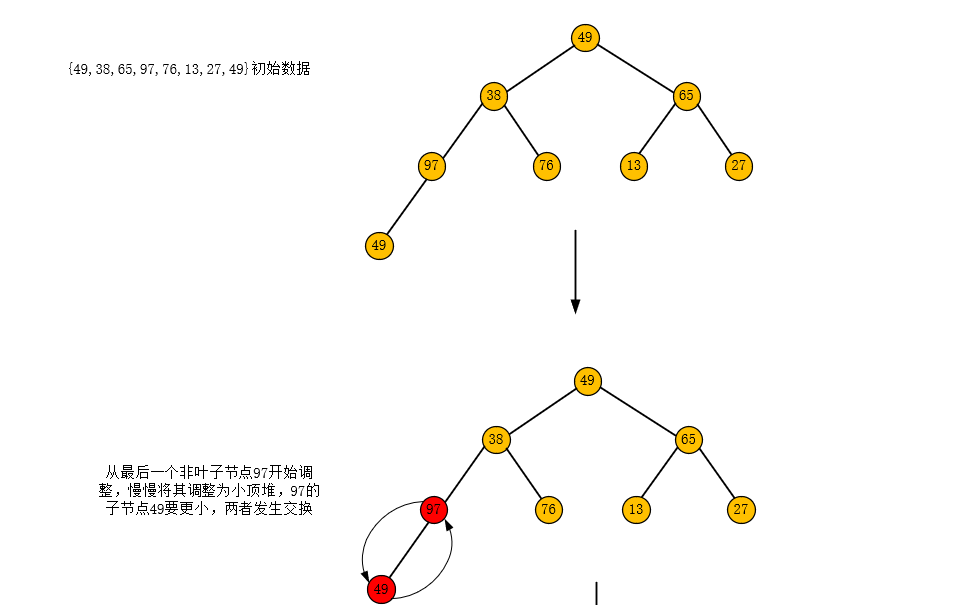
5.3 时间复杂度分析

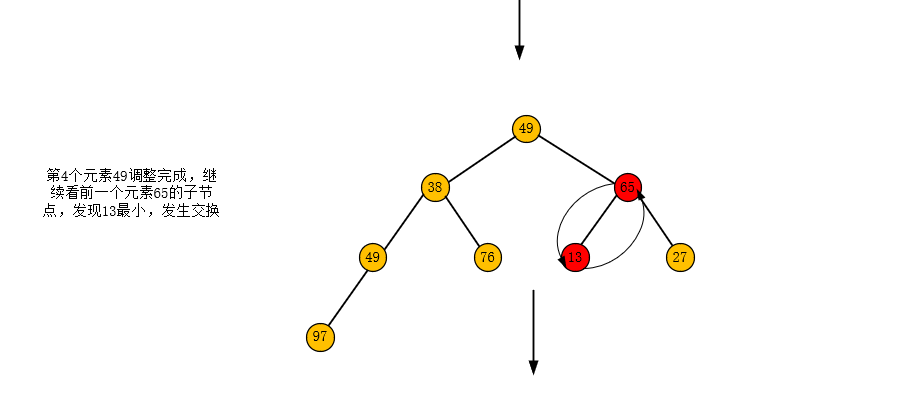
希尔排序是按照不同增量对元素进行插入排序，最开始元素无序的时候，增量最大，所以插入排序的元素个数也是最少，速度很快；当元素基本有序时，增量很小，插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比o(n2)好，我以上面Hibbard增量序列实现的希尔排序为例，它的最坏情况下时间复杂度大约是O(n^3/2)。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性会被打乱，所以希尔排序是不稳定的。

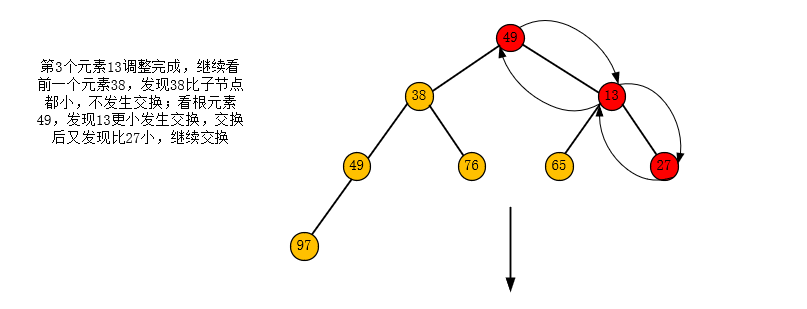
6 堆排序

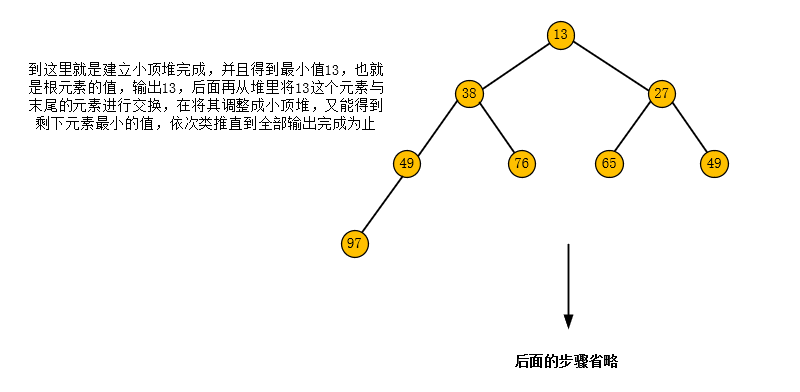
6.1 基本原理

堆排序的实际上就是使用堆这个数据结构进行排序的方法。它的基本思想是将待排序元素的序列构造成一个大顶堆(可以是小顶堆，也可以是大顶堆);大顶堆的根节点就是序列元素中的最大值。然后将其与堆中的末尾元素进行交换，此时末尾元素就是最大值了，然后再将剩余的元素重新构造成一个大顶堆，这样就会得到剩下元素中的最大值，重复进行直到只剩下一个元素市，便能得到一个有序的序列了。下面我演示下具体的执行过程









6.2 具体程序设计

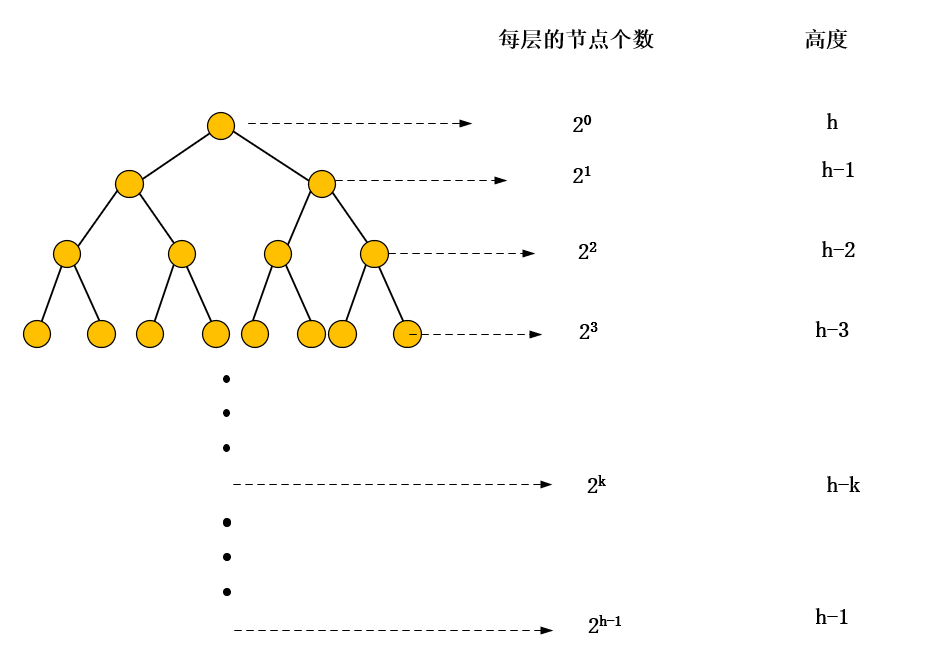
*/\*\*  
 \* 小顶堆排序，排序结果是逆序的  
 \** ***@param arr*** *\*/***public static void** smallHeapSort(**int**[] arr) {  
 *//构造小顶堆  
 buildSmallHeap*(arr);  
 *//排序，将最小的节点放在堆尾，然后从根节点重新调整* **for** (**int** i = arr.**length** - 1; i >= 1; i--) { *//当i=0时，就只剩最后一个元素了，无须再调整，所以i>=1即可  
 swap*(arr,0,i); *//交换元素，将小顶堆里的根元素与堆尾元素交换  
 smallHeapAdjust*(arr, 0, i); *//由于之前已经输出了一个元素，剩下总的元素个数就是i = arr.length - 1* }  
}  
  
*//元素位置交换***public static void** swap(**int**[] arr,**int** i,**int** j) {  
 **int** temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
}  
  
*/\*\*  
 \* 将无序的元素构建成小顶堆  
 \** ***@param arr*** *\*/***public static void** buildSmallHeap(**int**[] arr){  
 **int** i,n = arr.**length**;  
 *//i=n/2表示是从最后一个非叶子节点开始调整(因为最后一个叶子节点的位置就是n)，但是这里需要减1，因为是从0开始的  
 //每调整完一个元素后就需要移动到前面一个元素为基准进行调整，直到根元素被调整完* **for** (i = n/2-1; i >= 0; i--) {  
 *smallHeapAdjust*(arr, i, n);  
 }  
  
}  
  
*/\*\*  
 \* 小顶堆调整方法 找出当前元素的左右子节点的最小值并进行交换，直到最后自己是最小值为止  
 \** ***@param arr*** *判断  
 \** ***@param index*** *最后一个非叶子节点的索引  
 \** ***@param len*** *元素个数  
 \*/***private static void** smallHeapAdjust(**int**[] arr, **int** index,**int** len) {  
 **while** (**true**) {  
 *//左子节点的索引* **int** leftNode = 2 \* index + 1;  
 *//右子节点的索引* **int** rightNode = 2 \* index + 2;  
 *//设置最小值* **int** min = index;  
  
 *//左子节点比较，如果左子节点值更小，就将左子节点作为最小值* **if**(leftNode<len && arr[leftNode]<arr[min]){ *//这里leftNode<len是为了判断index下是否有左子节点* min = leftNode;  
 }  
 *//和右子节点比较，如果右子节点值更小，就将右子节点作为最小值* **if**(rightNode<len && arr[rightNode]<arr[min]){ *//这里rightNode<len是为了判断index下是否有右子节点* min = rightNode;  
 }  
 *//如果index不是最小值的话，在前面程序执行的过程中，min的值就会改变，表示需要交换数据；否则就无须交换，说明index就是最小值* **if** (min == index){  
 **break**;  
 }  
 *//执行到这说明找到最小值的位置了并且不是index，开始替换  
 swap*(arr,min,index);  
 index = min; *//交换位置后，将当前元素的索引更新，然后下次循环在这个新的位置进行调整* }  
  
}  
  
  
*/\*\*  
 \* 大顶堆排序 ，排序结果是正序的  
 \** ***@param arr*** *\*/***public static void** bigHeapSort(**int**[] arr) {  
 *//构造大顶堆  
 buildBigHeap*(arr);  
 *//排序，将最小的节点放在堆尾，然后从根节点重新调整* **for** (**int** i = arr.**length** - 1; i >= 1; i--) { *//当i=0时，就只剩最后一个元素了，无须再调整，所以i>=1即可  
 swap*(arr,0,i); *//交换元素，将大顶堆里的根元素与堆尾元素交换  
 bigHeapAdjust*(arr, 0, i); *//由于之前已经输出了一个元素，剩下总的元素个数就是i = arr.length - 1* }  
}  
  
  
*/\*\*  
 \* 将无序的元素创建成大顶堆  
 \** ***@param arr*** *\*/***public static void** buildBigHeap(**int**[] arr){  
 **int** i,n = arr.**length**;  
 *//i=n/2表示是从最后一个非叶子节点开始调整(因为最后一个叶子节点的位置就是n)，  
 //每调整完一个元素后就需要移动到前面一个元素为基准进行调整，直到根元素被调整完* **for** (i = n/2-1; i >= 0; i--) {  
 *bigHeapAdjust*(arr, i, n);  
 }  
}  
  
*/\*\*  
 \* 大顶堆调整  
 \** ***@param arr*** *\** ***@param index*** *最后一个叶子节点的父节点的索引(也是最后一个非叶子节点)  
 \** ***@param len*** *元素总个数  
 \*/***public static void** bigHeapAdjust(**int**[] arr,**int** index,**int** len){  
 **while** (**true**) {  
 **int** max = index; *//假设当前位置是最大值* **int** leftChild = 2 \* index + 1; *// 左孩子索引。* **int** rightChild = 2 \* index + 2; *// 右孩子索引。  
  
 // 若左孩子大于最大值，则更新最大值。* **if** (leftChild < len && arr[leftChild] > arr[max]) {  
 max = leftChild;  
 }  
  
 *// 若右孩子大于最大值，则更新最大值。* **if** (rightChild < len && arr[rightChild] > arr[max]) {  
 max = rightChild;  
 }  
  
 **if** (max == index) { *//说明已经是最大值* **break**;  
 }  
 *swap*(arr, index, max); *//执行到这说明找到最大值的位置了并且不是index，开始替换* index = max; *//交换位置后，将当前元素的索引更新，然后下次循环在这个新的位置进行调整* }  
}

上面的程序是将小顶堆和大顶堆基于这两种堆的排序都实现了。

6.3 时间复杂度分析

现在我们来分析一下堆排序的时间复杂度，堆排序的时间复杂度分析起来比较复杂；由于堆排序分为两个步骤，一个是构建堆，第二是排序；因此我们分析时间复杂度时也分成这两个步骤；

构建堆的时间复杂度分析：我们知道堆实际上就是一个完全二叉树，我们在构建堆的过程中就是顺着节点所在路径进行比较和交换，直到满足每个节点的值都大于或小于它的左右子节点的值，这个过程我们称为堆化。当树的高度越高，堆化所需要比较和交换次数就越多，花费的时间也就越多，所以构造堆的时间复杂度是跟树的高度成正比的关系；根据完全二叉树的性质，我们知道当完全二叉树的总节点数量是n个时，树的高度是不大于log2n +1的，因此堆里节点堆化的时间复杂度就是O(logn)；我们在构建堆的时候我们是从最后的非叶子节点开始进行堆化的，即是堆下标从从n/2开始堆化的，因为n/2后面的节点都是叶子节点，不需要堆化。因此需要堆化的节点从倒数第二层开始，每个节点堆化的过程中需要比较和交换节点的个数和这个节点的高度(假设为k)成正比的。因此我们只需要将每个节点的高度进行求和，就能算出构建堆的时间复杂度。我假设有n个元素，其构成完全二叉树的高度设为h，每一层的高度设为常量1，则如下图所示：



将每一个非叶子节点的高度求和得到如下公式：

S1=1\*h+2\*(h-1)+22\*(h-2)+…+2k\*(h-k)+…+2h-1\*1

我们将公式S1左右都乘以2得到另一个公式2S1。我们将其错位对齐，并用2S1减去S1得到：

S1=1\*h+2\*(h-1)+22\*(h-2)+…+2k\*(h-k)+…+2h-1\*1

2S1= 2\*h + 22\*(h-1)+…+2k\*(h-k-1)+…+2h-1\*2+2h\*1

S1=2S1-S1= -h+2+22+23+…+2k+…+2h-1+2h

这样我们就得到了一个等比数列，所以我们可以用等比数列的求和公式得到

S1=2h+1 - 2 - h，然后我们将h=log2n代入其中得到S1=4n-2-h，因此构建堆的时间复杂度是O(n)。

排序的时间复杂度分析：当堆创建完成后，就开始正式排序，在第一次堆排序时，因为移除了最大值(或者最小值)，所以就是n-1个节点中的根节点进行堆化，后面每次排序，剩下的堆节点都会减少，树的高度也会跟着减少，总高度为：

S=log2(n-1)+log2(n-2)+…+log2(n-k)+…+log2(1)=log2(n-1)\*(n-2)\*…\*(n-k)\*…\*1

=log2(n-1)!

我们最终得到排序总高度的和是log2(n-1)!，又因为log(n!)和nlogn是同阶函数(可证明)，因此最终堆第二步排序的时间复杂度就是O(nlogn),因为O(nlogn)大于O(n),因此堆排序最终的时间复杂度就是O(nlogn)。

关于证明log2n!和nlog2n是同阶函数如下：

上限：log(n!) = log(1)+log(2)+…+log(n-1)+log(n)<=log(n)+…+log(n)=n\*log(n)

下限：log(n!) = log(1)+log(2)+…+log(n/2)+…+log(n-1)+log(n)

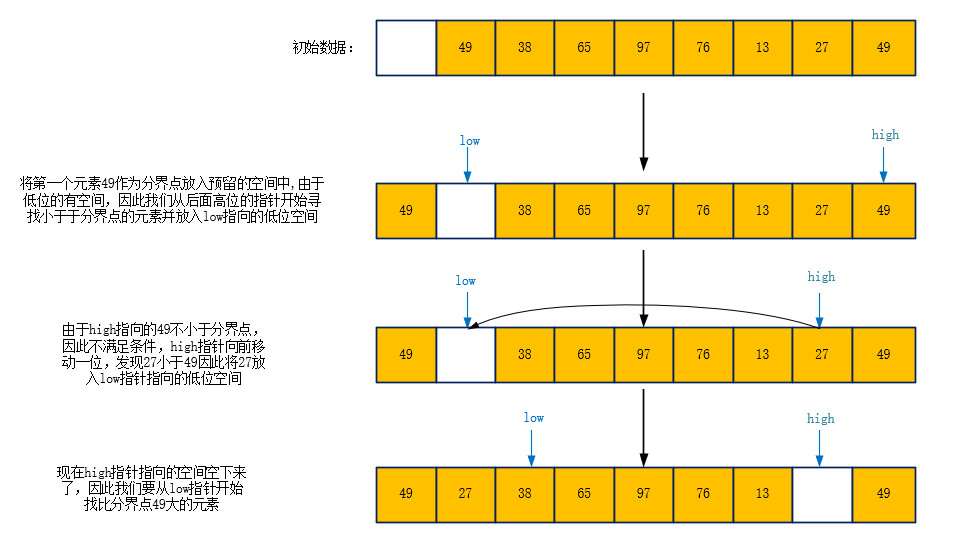
>=log(n/2)+log(n/2+1)+…+log(n-1)+log(n)>=log(n/2)+…+log(n/2)

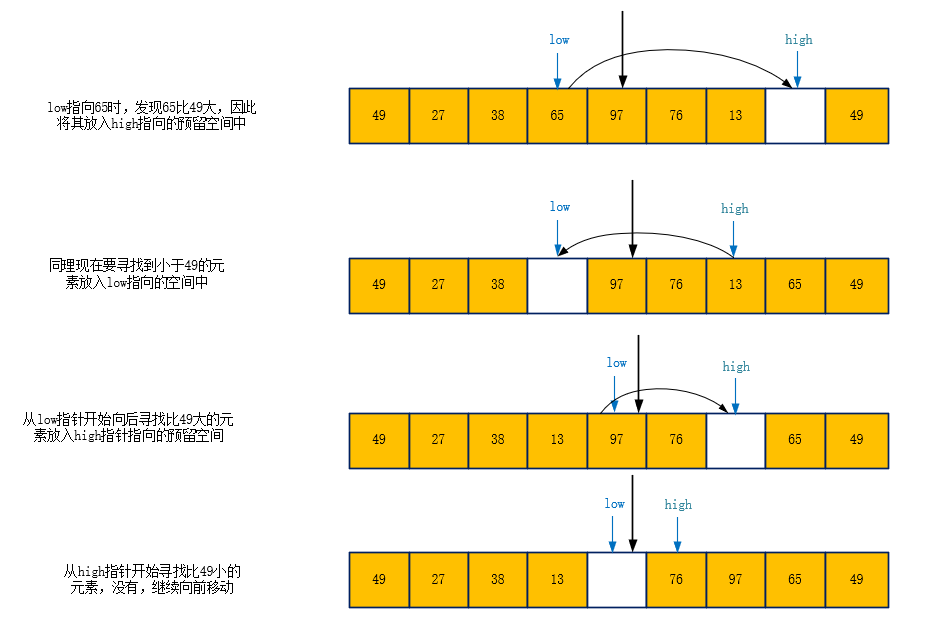
=n/2 \* log(n/2)

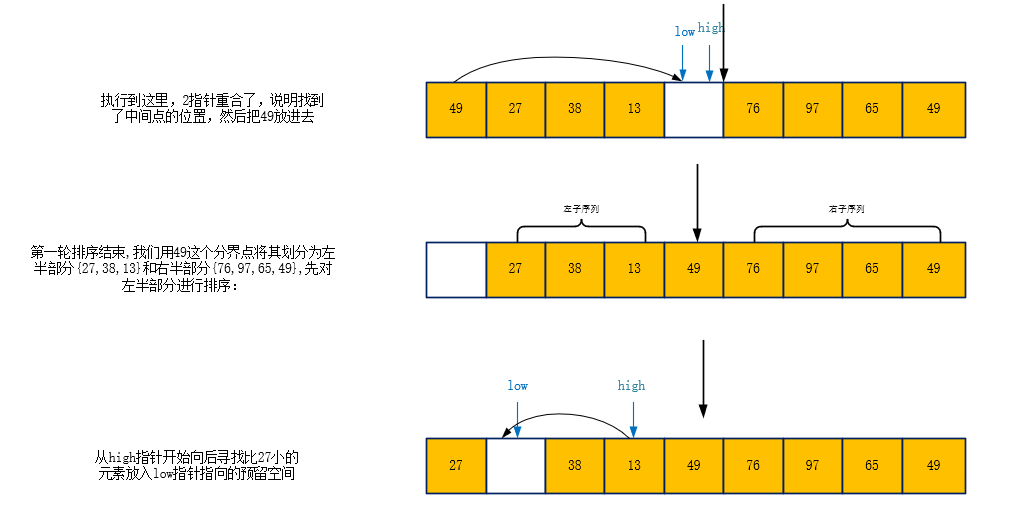
7 快速排序

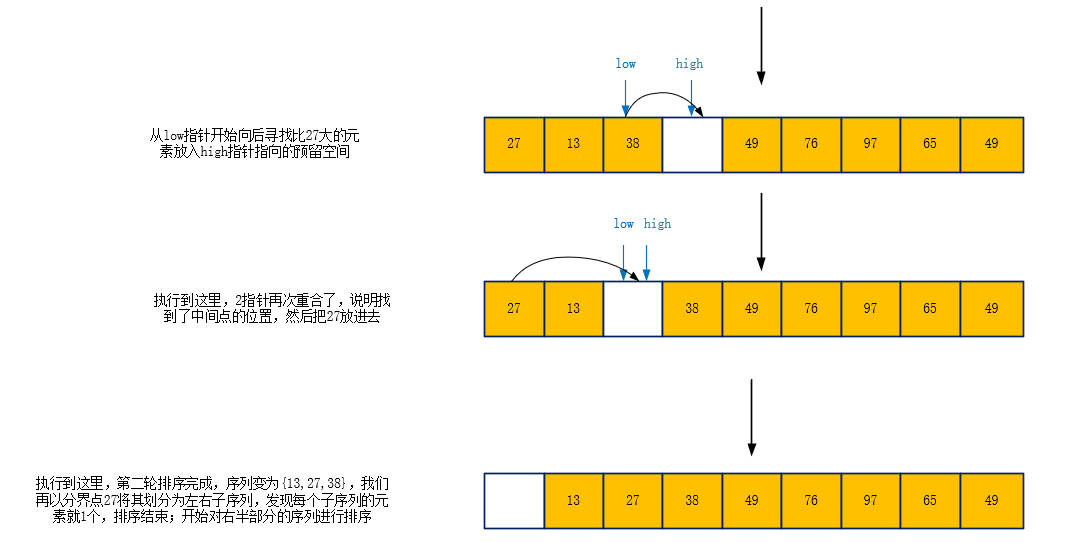
7.1 基本原理

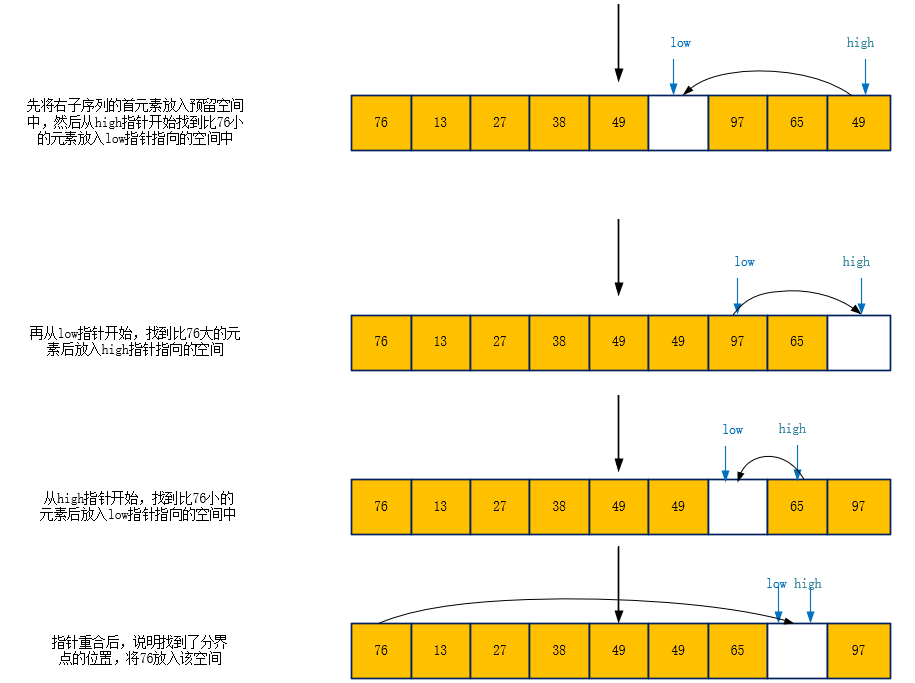
前面我们讲到了希尔排序和堆排序；实际上希尔排序相当于直接插入排序的升级版，它们同属于插入排序类，堆排序相当于简单选择排序的升级，它们同属于选择排序类。而快速排序其实就是前面冒泡排序的升级版，它们都属于交换排序类。快速排序的基本思想是通过一轮排序将待排序的序列分成独立的两个子序列，其中一个子序列的所有元素都要比另一个子序列的所有元素都小，再分别将着两个子序列继续进行排序，以达到整个序列有序的母的。下面我演示下具体的执行过程：

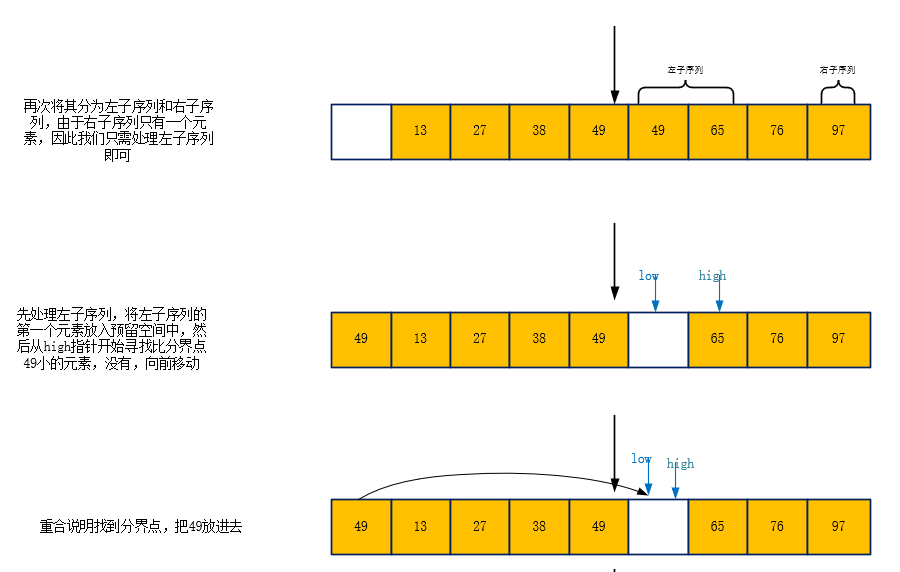


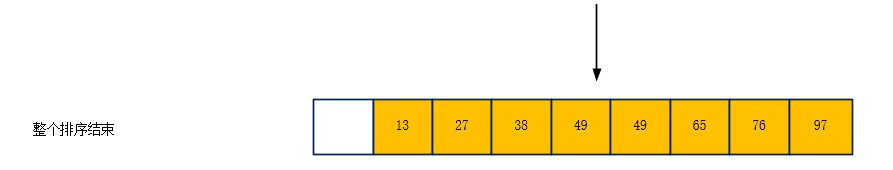












7.2 具体程序设计

*/\*\*  
 \*  
 \** ***@param arr*** *待排序的数组  
 \** ***@param low*** *低位指针  
 \** ***@param high*** *高位指针  
 \*/***public static void** quickSort(**int**[] arr,**int** low,**int** high){  
 **int** pivot;  
 **if** (low < high){ *//如果low不小于high，说明2指针重合，这一轮排序结束* pivot = *partition*(arr,low,high); *//将序列一分为二，pivot为排好序的中心元素的位置  
 quickSort*(arr,low,pivot-1); *//对低序列递归排序  
 quickSort*(arr,pivot+1,high); *//对高序列递归排序* }  
}  
  
*//元素位置交换***public static void** swap(**int**[] arr,**int** i,**int** j) {  
 **int** temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
}

*/\*\*  
 \* 找序列中心点的位置(不设置预留空间)  
 \** ***@param arr*** *\** ***@param low*** *\** ***@param high*** *\** ***@return*** *\*/***private static int** partition(**int**[] arr, **int** low, **int** high) {  
 **int** pivotValue = arr[low]; *//得到分界点的元素值* **while** (low < high){  
 *//如果分界点小于high指针指向的元素，则将high指针指向的元素放入low指针指向的空间中，并且，low指针向后移动一位;  
 //如果不满足pivotValue < arr[high]，则high指针前移，直到满足条件，因此这必须是一个循环* **while** (low < high && arr[high] >= pivotValue ){  
 high--; *//每次执行到这里，意味着high指针没有找到比分界点小的元素，high指针需要向前移动一位* }  
 *//执行到这里说明high指针找到了比分界点小的元素，将high指针指向的元素与low指针指向的空间进行交换，并且low指针向后移动一位  
 swap*(arr,low,high);  
 *//如果分界点大于low指针指向的元素，则将low指针指向的元素放入high指针指向的空间中，并且high指针向前移动一位* **while** (low < high && arr[low] <= pivotValue){  
 low++; *//每次执行到这里，意味着low指针没有找到比分界点大的元素，low指针需要向后移动一位* }  
 *//执行到这里说明low指针找到了比分界点大的元素，将low指针指向的元素与high指针指向的空间进行交换，并且high指针向前移动一位  
 swap*(arr,high,low);  
 }  
 *//执行到这里说明已经重合，将分界点元素放入分界的位置  
 //swap(arr,high,0);* **return** high;  
}

7.3 时间复杂度分析

现在分析下快速排序的时间复杂度，快速排序的时间性能主要取决于快速排序递归的深度，假设我们要排序n个元素，快速排序需要时间为T(n)的话，我们第一次划分需要对整个数组比较一遍，即需要n次比较，获得的分界点将序列一分为二，那么各自还需要T(n/2)的时间。那么依次类推：

T(n) = 2\*T(n/2) + n 第一次划分

= 2\*(2\*T(n/4) + n/2) + n = 4\*T(n/4) + 2\*n 第二次划分

= 4\*(2\*T(n/8) + n/4) + 2\*n = 8\*T(n/8) + 3\*n 第三次划分

= 8\*(2\*T(n/16) + n/8) + 3\*n = 16\*T(n/16) + 4\*n 第四次划分

......

= 2^k \* T(n/2^k) + k \* n 根据数学归纳法第k次划分

......

为了求出具体的时间复杂度，假设快速排序一个元素所需要花费的时间是常量C即：T(1)=C，那么快速排序执行到最后阶段就只剩下一个元素，即T(n/2^k)=T(1)=C时，这样n/2^k=1即n=2^k。根据对数的知识我们就知道k=log2n，然后我们将该公式带入后得到T(n)=n\*C + nlog2n;因此时间复杂度是O(nlogn),但是这是最好情况下的时间复杂度；假设我们要排序的数据是正序或者逆序的，每次划分得到的其中一个子序列的元素个数只有1个，另外一个子序列的元素是n-1个，后面每次划分都会变成这样，每次比较也只能找到一个元素排序，这样就退化成了冒泡排序了。这时我们排序消耗的时间变成了T(n)=n-1+n-2+….+1=n(n+1)/2,时间复杂度变成了O(n2)；而且由于快速排序的比较和交换是跳跃进行的，因此快速排序是一种不稳定的排序。

8 运行和测试

8.1 测试数据生成程序设计

**public class** GenerateArray {  
  
 **public static int**[] generateArray(**int** n) {  
 **int**[] a=**new int**[n];  
 Random r=**new** Random();  
 **for**(**int** i=0;i<n;i++) {  
 a[i]=r.nextInt(n);  
 }  
 **return** a;  
 }  
 */\*\*  
 \* 默认产生10000个随机数  
 \** ***@return*** *\*/* **public static int**[] generateArray() {  
 **int**[] a=**new int**[10000];  
 Random r=**new** Random();  
 **for**(**int** i=0;i<10000;i++) {  
 a[i]=r.nextInt(10000);  
 }  
 **return** a;  
 }  
  
}

上面的程序可以随机生成指定数量长度的随机数组，专门用来提供不同规模大小的测试数据。

8.2 测试程序模块划分和实现

我将模块按排序类别分为六个模块，每个模块都有一个test方法用来计算排序所消耗的时间。

冒泡排序测试方法：

**public static void** test(**int**[] arr){  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"冒泡排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *bubbleSortOptimize2*(ints);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"冒泡排序结束，花费时间："**+ l + **"毫秒"**);  
}

直接插入排序测试方法：

**public static void** test(**int**[] arr){  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"插入排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *directInsertSort*(ints);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"插入排序结束，花费时间："**+ l + **"毫秒"**);  
}

简单选择排序测试方法：

**public static void** test(**int**[] arr) {  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"选择排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *selectionSort*(ints);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"选择排序结束，花费时间："**+ l + **"毫秒"**);  
}

希尔排序测试方法：

**public static void** test(**int**[] arr){  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"希尔排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *shellSort*(ints);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"希尔排序结束，花费时间："**+ l + **"毫秒"**);  
}

堆排序测试方法：

**public static void** test(**int**[] arr){  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"堆排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *bigHeapSort*(ints);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"堆排序结束，花费时间："**+ l + **"毫秒"**);  
}

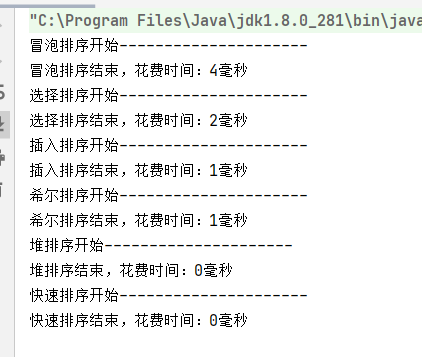
快速排序测试方法：

**public static void** test(**int**[] arr){  
 **int**[] ints = Arrays.*copyOf*(arr, arr.**length**);  
 System.***out***.println(**"快速排序开始---------------------"**);  
 **long** l1 = System.*currentTimeMillis*();  
 *quickSort*(ints,0,ints.**length**-1);  
 **long** l2 = System.*currentTimeMillis*();  
 **long** l = l2 - l1;  
 System.***out***.println(**"快速排序结束，花费时间："**+ l + **"毫秒"**);  
}

各自排序的测试类

**public class** Test {  
  
 **public static void** main(String[] args){  
 **int**[] testarray = GenerateArray.*generateArray*(200000);  
 BubbleSort.*test*(testarray); *//冒泡排序* SelectionSort.*test*(testarray); *//选择排序* InsertionSort.*test*(testarray); *//插入排序* ShellSort.*test*(testarray); *//希尔排序* HeapSort.*test*(testarray); *//堆排序* QuickSort.*test*(testarray); *//快速排序* }  
}

下面来看我先生成数量元素为1000的元素进行测试来对比各个排序所消耗的时间



可以看出1000个元素对各个排序之间所消耗的时间差别并不是很大，下面我增加数据规模，随机生成10万个元素的数据进行排序后再进行比较。



可以看到当数据规模增加到10万时，冒泡排序，选择排序，插入排序依次相差1个数量级；而希尔排序，堆排序和快速排序所消耗时间基本一样，下面我继续增加数据规模，将其元素个数增加到100万时我们在进行比较。



从上面的测试结果可以看出，冒泡排序，选择排序，插入排序和之前一样依次相差一个数量级；而三个排序中即使是最快的插入排序与希尔排序，堆排序和快速排序竟也相差两个数量级；在希尔排序，堆排序和快速排序中快速排序名副其实是三者中效率最高的。

总结

排序算法是是计算机程序设计、数据库、操作系统、编译原理及人工智能等的重要基础，广泛应用于信息学、系统工程等各种领域。学习排序算法是为了将实际问题中涉及的对象在计算机中进行处理。本毕业论文对直接插入排序、简单选择排序、冒泡排序、希尔排序、堆排序和快速排序算法进行测试。待排序表的元素关键字为整数类型，用不同的测试数据做测试比较。比较的指标为所消耗的时间多少。

经过比较我们发现，当规模不断增加时，各种算法之间的差别是很大的。这四种算法中，各种算法之间的差别是很大的。这六种算法中，快速排序所消耗的时间是最少的。也是最快的一种排序方法，然后是堆排序和希尔。插入排序和选择排序次之，最慢的就是冒泡排序，要比希尔排序，堆排序和快速排序所消耗的时间多上几个数量级。

结束语

在自己不断独自思考和查询资料的情况下，我独自完成了排序算法分析和对排序算法以图形的形式展现其执行过程。本软件能随机生成测试数据，选择数种经常使用排序算法中的一种的动态演示，还能把该算法的伪代码显示出来；能选择随机生成数据进行各种排序算法需要时间比较，比较的结果通过消耗时间的大小展示出来。还能够通过图画的形式将主要排序算法的排序过程形象的演示出来。从而使学习排序算法变得轻松愉快。对于以上程序在 jdk1.8环境中运行正常。通过毕业设计的锻炼，学到了很多东西： ① 巩固和加深了对数据结构的理解，提高综合运用本课程所学知识的能力。 ② 培养了我选用参考书，查阅手册及文献资料的能力。培养独立思考，深入研究，分析问题、解决问题的能力。 ③ 通过自己编写的程序对各种排序性能的比较让我更深入理解了他们的应用④ 够按要求编写毕业设计报告书，能正确阐述设计和实验结果。毕业设计是把我们所学的理论知识进行系统的总结并应用于实践的良好机会，有利于加强我们用知识理论来分析实际问题的能力，进而加强了我们对知识认识的实践度，巩固了我们的理论知识，深化了对知识的认识，并为走向社会打下一个良好的基础。

致谢

光阴似箭，日月如梭，不知不觉就已经到了快毕业的时间。期间，在老师和同学们的关怀和帮助下，我丰富了知识、扩大了视野、提高了能力，为今后的学习与发展奠定了基础。

在此我要特别的感谢我们的卢萍老师。他给予我学业上的无私教诲。同时，感谢所有教导过、关心过、帮助过我的老师们，是他们使我有更多的机会尝试着站在理论和实践的新起点上进行思考。

在此，还要感谢在求学期间认识的所有同学和朋友们给予的帮助。

最后，我要感谢我的父母，是他们一直在背后默默地支持我。

参考文献

[1] 程杰，《大话数据结构》，清华大学出版社2008

[2] 马克·艾伦·维斯，数据结构与算法分析-Java语言描述（第 3 版）．机械工业出版社 2007

[3] Bruce Eckel（美）Java编程思想（第4版）机械工业出版社 2007

[4] 严蔚敏，吴伟民， 《数据结构题集（C语言版）》，清华大学出版社 2009

[5]朱继红，数据结构算法动态示系统的设计与实现 【J】信息工程学院学报1998

[6] 张群哲， 《数据结构（C语言版）》，西安电子科技大学出版社2008

[7] 严蔚敏，《数据结构》（ C 语言版），清华大学出版社2005

[8] Cay S.Horstman(美)， Java核心技术卷1基础知识（第10版），机械工业出版社 2016

[9] Cay S.Horstman(美)， Java核心技术卷2高级特性（第10版），机械工业出版社 2016

. [10] 塞若贝茨著，《Head First Java》（第2版），中国电力出版社 2007

[11] 约书亚·布洛克，《Effective Java》（第3版），机械工业出版2018

**毕业设计（论文）成绩评定**

|  |
| --- |
| **一、答辩前每个学生都要将直接的毕业设计（论文）在指定的时间内交给指导老师，，由指导老师审阅，写出评语并预评分。**  **二、答辩工作结束后，答辩小组应举行专门会议按统一的评分标准和评分办法，在参考指导教师评语结果的基础上，评定每个学生的成绩。在有争议是，应由小组成员进行表决；综合成绩由答辩小组确定。**  **三、成绩评定采用以下评分的办法，即由指导教师跟答辩小组分别给分（以百分计），两部分成绩比例由各院系根据实际情况确定，最后按优秀（90～100分）；良好（80～89分）；中等（70～79分）；及格（60～69分）；不及格（59分以下）。**  **四、综合成绩由两部分组成，指导教师评定成绩占60%，答辩小组评定成绩占40%。** |

**毕业设计（论文）成绩评定**

|  |
| --- |
| **学生姓名：**  **综合成绩： 分（折合等级 ）**  **答辩小组长签字: 年 月 日** |

**一、评语（根据学生答辩情况及其论文质量综合写出）：**

**二、评分（按下表要求综合评定）：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **评分项目**  **（分值）** | **答辩情况** | | **论文质量** | | **合计（100分）** |
| **表达情况**  **（15分）** | **回答问题情况（25分）** | **规范要求与文字表达（20分）** | **学术（或技术水平）（40分）** |
| **得分** |  |  |  |  |  |

**答辩小组长签字： 年 月 日**

**毕业答辩说明**

**1、答辩前，答辩小组应详细审阅每个学生的毕业设计（论文），为答辩做好准备；并根据毕业设计（论文）质量给出分数。**

**2、严肃认真组织答辩，开好答辩会，给出答辩成绩。**

**3、指导教师应参加所指导学生的答辩，但评定其成绩时宜回避。**

**4、答辩中要作好记录。**

**一、对毕业设计（论文）的学术评语（应具体、确切、实事求是）：**

**二、对毕业设计（论文）评分[按下表要求综合评定]：**

**（1）理工科评分表**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **评分项目(分值)** | **调研论证(10分)** | **技术水平与实际能力(25分)** | **基本理论与专业知识(25分)** | **创新**  **(15分)** | **论文(说明说)撰写质量(15分)** | **学习态度(10分)** | **合计**  **(100分)** |
| **得分** |  |  |  |  |  |  |  |

**（2）文科评分表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **评分项目(分值)** | **文献阅读与**  **文献综述**  **(10分)** | **学术水平**  **与创新**  **(30分)** | **论文能力**  **(30分)** | **论文撰写质量**  **(20分)** | **学习态度**  **(10分)** | **合计**  **(100分)** |
| **得分** |  |  |  |  |  |  |

**指导老师签字： 年 月 日**