# 排序

## 希尔排序

* 算法介绍

**希尔排序的实质就是分组插入排序**，该方法又称缩小增量排序，因DL．Shell于1959年提出而得名。

* 基本思想

**先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前三种方法有较大提高。**

|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //希尔排序  void ShellSort**(**int arr**[],**int len**){**    int increasement **=** len**;**  **do{**  //缩减增量  increasement **=** increasement **/** 3 **+** 1**;**  **for** **(**int i **=** 0**;** i **<** increasement**;** i **++){**  //遍历每一组  **for** **(**int j **=** i **+** increasement**;** j **<** len**;** j **+=** increasement**){**    **if** **(**arr**[**j**]** **<** arr**[**j **-** increasement**]){**    int temp **=** arr**[**j**];**  int k **=** j **-** increasement**;**  **for** **(;** k **>=** 0 **&&** temp **<** arr**[**k**];**k **-=** increasement**){**  arr**[**k **+** increasement**]** **=** arr**[**k**];**  **}**  arr**[**k **+** increasement**]** **=** temp**;**  **}**  **}**  **}**  **}** **while** **(**increasement **>** 1**);**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  ShellSort**(**arr**,** MAX**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

通过这段代码，大家应该能够看到在希尔排序中我们并不是随便的把序列分组，然后对每个子序列分别排序，而是将相隔“增量”的记录组成一个子序列，实现跳跃式移动，使得排序效率提高。

这个增量的选取就非常关键了。我们案例中是用Increasement = increasement / 3 + 1,至于这个增量如何选取非常难，没有明确的确定，但是通过前人的研究，我们用

步长 = 步长 / 3 + 1。

## 快速排序

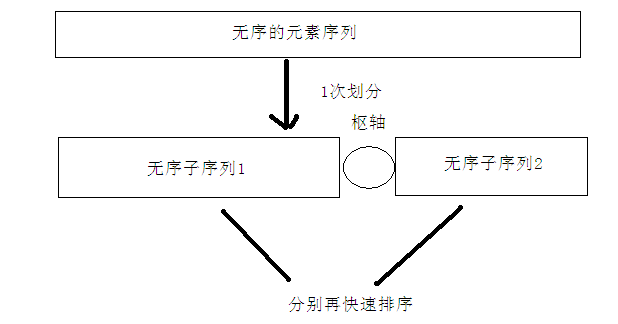
* 算法介绍

快速排序是C.R.A.Hoare于1962年提出的**一种划分交换排序**。它**采用了一种分治的策略**，通常称其为分治法(Divide-and-ConquerMethod)。快速排序被称为20世界十大算法之一。快速排序和冒泡排序类似，不断的通过比较和移动交换来实现排序，只不过他的实现增大了比较和移动的距离，把较大的数据移动到后面，较小的数据移动到前面，从而减少了了总的比较和移动交换次数。该算法是目前实践中使用最频繁、实践效率最好的排序算法。

* 分治法基本思想

通过一趟排序将待排序的数据分成两部分，其中一部分的数据均比另一部分小，然后再分别对这两部分进行排序，以达到整个序列有序的目的。

* + 先从数列中取出一个数作为基准数（枢轴）。
  + **分区过程将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。(升序)**
  + 再对左右区间重复第二步，直到各区间只有一个数。



|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //快速排序  void QuickSort**(**int arr**[],**int start**,** int end**){**  int i **=** start**;**  int j **=** end**;**  //取基准数  int target **=** arr**[**start**];**  **if** **(**i **<** j**){**    **while** **(**i **<** j**){**    //从右向左找比基准数大的  **while** **(**i **<** j **&&** arr**[**j**]** **<** target**){**  j**--;**  **}**  **if** **(**i **<** j**){**  arr**[**i**]** **=** arr**[**j**];**  i**++;**  **}**  //从左向右找比基准数小的  **while** **(**i **<** j **&&** arr**[**i**]** **>** target**){**  i**++;**  **}**  **if** **(**i **<** j**){**  arr**[**j**]** **=** arr**[**i**];**  j**--;**  **}**  **}**    //i的位置就是基准数最合适的位置  arr**[**i**]** **=** target**;**  //快速排序左半部分  QuickSort**(**arr**,**start**,**i **-** 1**);**  //快速排序有半部分  QuickSort**(**arr**,** i **+** 1**,** end**);**  **}**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  QuickSort**(**arr**,** 0**,**MAX **-** 1**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

## 归并排序

* 算法介绍

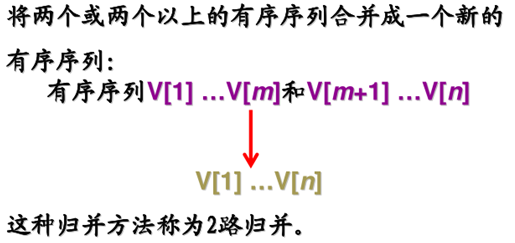
归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

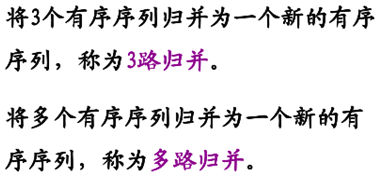
* 基本思想

**基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序**。如何让这二组组内数据有序了？

**可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。**

* **归并的定义**





* **如何合并连个有序序列？？？**

只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //合并两个有序序列  void Merge**(**int arr**[],** int start**,** int end**,** int mid**,** int tempSpace**[]){**    int iStart **=** start**;**  int iEnd **=** mid**;**  int jStart **=** mid**+**1**;**  int jEnd **=** end**;**  int length **=** 0**;**  **while** **(**iStart **<=** iEnd **&&** jStart **<=** jEnd**){**    **if** **(**arr**[**iStart**]** **<** arr**[**jStart**]){**  tempSpace**[**length**]** **=** arr**[**iStart**];**  iStart**++;**  **}**  **else{**  tempSpace**[**length**]** **=** arr**[**jStart**];**  jStart**++;**  **}**  length**++;**  **}**  //两个序列中有一个还有剩余元素  **while** **(**iStart **<=** iEnd**){**  tempSpace**[**length**]** **=** arr**[**iStart**];**  iStart**++;**  length**++;**  **}**  **while** **(**jStart **<=** jEnd**){**  tempSpace**[**length**]** **=** arr**[**jStart**];**  jStart**++;**  length**++;**  **}**  //覆盖原空间的数据  **for** **(**int i **=** 0**;** i **<** length**;**i**++){**  arr**[**start **+** i**]** **=** tempSpace**[**i**];**  **}**  **}**  //归并排序  void MergeSort**(**int arr**[],**int start**,**int end**,**int tempSpace**[]){**  **if** **(**start **==** end**){**  **return;**  **}**  int mid **=** **(**start **+** end**)** **/** 2**;**  //拆分左半部分  MergeSort**(**arr**,** start**,** mid**,** tempSpace**);**  //拆分右半部分  MergeSort**(**arr**,** mid **+** 1**,** end**,** tempSpace**);**  //拆分完了 合并有序序列  Merge**(**arr**,**start**,**end**,**mid**,**tempSpace**);**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  int tempSpace**[**MAX**];**  MergeSort**(**arr**,** 0**,** MAX **-** 1**,** tempSpace**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

## 排序总结

各种排序算法的比较需要考虑诸多的因素，在实际操作中，我们主要考虑时间复杂度和空间复杂度这两个方面。虽然仅仅靠这两个参数是无法判断一个算法的优劣，算法的实际性能还受运行的硬件、数据规模和数据特点的影响，但是这两个方面的比较对于算法的选择仍然具有很好的指导意义。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **排序算法** | **平均时间**  **复杂度** | **最坏时间**  **复杂度** | **平均空间**  **复杂度** | **稳定性** |
| 选择排序 | O(n2) | O(n2) | O(1) | 不稳定 |
| 冒泡排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 插入排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 希尔排序 | O(nlogn) | O(n2) | O(1) | 不稳定 |
| 快速排序 | O(nlogn) | O(n2) | O(logn) | 不稳定 |
| 归并排序 | O(nlogn) | O(nlogn) | O(n) | 稳定 |
| 堆排序 | O(nlogn) | O(nlogn) | O(1) | 不稳定 |

在了解了各种算法的理论性能之后，我们对算法的选择标准还要参考待排序数据的规模、存储空间的紧张程度和待排序数据的初识状态。综合考虑这些因素，我们给出以下几条建议：

* 在通常的情况下，即输出数据是随机的，快速排序、归并排序、希尔排序和堆排序的运行速度较快。其中堆排序是原地排序最省空间，快速排序的速度最快。所以如果在内存空间不紧张的情况下，一般采用快速排序，如果需要节省空间则采用堆排序，希尔排序不适合用在链表数据结构上。
* 如果待排序规模不大而且基本有序，插入排序和冒泡排序的运行时间很好，可选用这两排序算法。
* 从待排序的规模方面考虑，规模较小的时候，采用简单的排序算法比较合适，其整体性能比较高；如果规模较大，采用改进型的算法比较合适。

综合看来，每种算法都有其优缺点，很难说那种算法绝对比另一种算法好，我们需要根据具体情况具体分析，从而选择最合适项目的排序算法。