# 第二章 排序

## 初级排序算法

### 游戏规则

* 排序成本模型：在研究排序算法时，我们需要计算**比较和交换的数量**。对于不交换元素的算法，我们会计算**访问数组的次数**。
* 内存使用：排序算法的额外的内存开销和运行时间是同等重要的。排序算法可以分为两类：除了函数调用所需的栈和固定数目的实例变量之外无需额外内存的**原地排序算法**，以及需要额外内存空间来存储另一份数组副本的**其他排序算法**。

### 选择排序

首先，找到数组中最小的元素；其次，将它和数组的第一个元素交换位置（如果第一个元素就是最小的元素那么它就和自己交换）；再次，在剩下的元素中找到最小的元素，将它和数据的第二个元素交换位置；如此反复，知道将真个数组排序。这种方法叫做选择排序。



**命题A。**对于长度为的数组，选择排序大约需要次比较和次交换。

**证明。**

选择排序有两个鲜明的特点。**运行时间和输入无关**，为了找出最小的元素而扫描一遍数组并不能为下一遍扫描提供什么信息。一个已经有序的数组或一个主键全部相等的数组和一个随机排列的数组所用的排序时间一样长。**数据移动是最少的**，每次交换都会改变两个元素的值，因此选择排序用了N次交换——交换次数和数组的大小是线性关系。

### 插入排序

将每一个元素插入到其他已经有序的数组中的适当位置，为了给要插入的元素腾出空间，我们需要将其余所有元素在插入之前都向右移动一位。这种算法叫做插入排序。

与选择排序一样，当前索引左边的所有元素都是有序的，但它们的最终位置还不确定，为了给更小的元素腾出空间，它们可能会被移动。但是当索引到达数组的最右端时，数组排序就完成了。

和选择排序不同的是，插入排序所需的时间取决于输入中元素的初始顺序。一个很大且其中的元素已经有序（或接近有序）的数组进行排序将会比随机顺序的数组或是逆序数组进行排序要快的多。



**命题B。**对于随机排列的长度为的主键不重复的数组，平均情况下插入排序需要次比较以及次交换。（即使最坏情况也比选择排序快）

**证明。**

**命题C。**插入排序需要的交换操作和数组中倒置的数量相同，需要的比较次数大于等于倒置的数量，小于等于倒置的数量加上数组的大小在减1。

换一种说法，对于随机排列的长度为的主键不重复的数组，若其倒置数量为，则交换次数为，比较次数**。**当且仅当顺序时左边等号成立，当且仅当逆序时右边等号成立。

**证明。**每次交换都改变了两个顺序颠倒的元素的位置，相当于减少了一对倒置，当倒置数量为0时，排序就完成了。每次交换都对应着一次比较，且1到N-1的每个i可能都需要一次额外的比较。（这时a[i]已经插入到位，如果a[i]到达数组最左边就不需要一次额外比较，如果a[i]没有到达最左边，需要额外一次比较a[i]和a[i-1]，以确认a[i]已经到位）。

**命题D。**对于随机排序的无重复主键的数组，插入排序和选择排序的运行时间是平方级别的，两者只比应该是一个较小的常数。

### 希尔排序

希尔排序是一种基于插入排序的快速的排序算法。对于大规模的乱序数组插入排序很慢，因为它只会交换相邻的元素，因此元素只能一点一点地从数组的一端移动到另一端。

希尔排序的思想是使数组中任意间隔为h的元素都是有序的。这样的数组被称为**h有序数组**。一个h有序数组就是h个互相独立的有序数组编织在一起组成的一个数组。在进行排序时，如果h很大，就能将元素移动到很远的地方，为实现更小的h有序数组创造方便。用这种方式，对于任意以1结尾的h序列，都能将数组排序。

实现希尔排序的一种方法是对于每个h，用插入排序将h个字数组独立地排序。但因为子数组是相互独立的，一个更简单的方法是在h子数组中将每个元素交换到比它大的元素之前去（交换a[i]和a[i-h]）。只需要在插入排序的代码中将移动元素的距离由1改为h即可。这样，希尔排序的实现就转化为一个类似于插入排序但使用不同增量的过程。



**命题E。**使用递增序列1,4,13,40...的希尔排序所需的比较次数不会超过N的若干倍乘以递增序列的长度。

## 归并排序

**命题F。**对于长度为N的任意数组，自顶向下的归并排序需要至次比较。

**证明。**令表示一个长度为N的数组排序时所需要的比较次数。我们有，对于N>0，通过递归的sort()方法我们可以由相应的归纳关系得到比较次数的上限：

右边第一项是将数组的左半部分排序所用的比较次数，第二项是将数组的右半部分排序所用的比较次数，第三项是归并所用的比较次数。因为归并所需的比较次数最少为，最多为，比较次数的下限是：

当N是2的幂（即，）时，因为，可以得到上限：

两边同除以，得到：

得到：

对于下限：

两边同除以，得到：

得到：

**命题G。**对于长度为N的任意数组，自顶向下的归并排序最多需要访问数组次。

**证明。**每次归并最多需要访问数组次（次用来复制，用来将排好序的元素移动回去，另外最多比较次），根据命题F即可得到这个命题的结果。

### 原地并归的抽象方法



该方法先将所有元素复制到aux[]中，然后在对并回a[]中。方法在归并时进行了4个条件判断：左半边用尽（取右半边元素）、右半边用尽（取左半边元素）、有半边的当前元素小于左半边的当前元素（取右半边的元素）以及右半边的当前元素大于等于左半边的当前元素（取左半边元素）。

### 自顶向下的归并排序



* 对小规模的子数组使用插入排序。使用插入排序处理小规模的子数组（比如长度小于15）一般可以将归并排序的运算时间缩短10%~15%。
* 测试数组是否已经有序。可以添加一个判断条件，如果a[mid]小于等于a[mid+1]，我们认为数组已经是有序的并跳过merge()方法。
* 不将元素复制到辅助数组。可以节省用于将数组元素复制到用于归并的辅助数组所组成的时间（但空间不行）。要做到这一点我们要调用两种排序方法，一种将数组从输入数组排序到辅助数组，一种将数组从辅助数组排序到输入数组。这种方法需要一些技巧，我们要在递归调用的每个层次交换输入数组和辅助数组的角色。

### 自底向上的归并排序

实现归并排序的另一种方法是先归并那些微型数组，然后在成对归并得到的字数组，如此这般，知道我们将整个数组归并在一起。这种实现方法比标准递归方法所需要的代码两更少。首先我们进行的是两两归并（把每个元素看成是一个大小为1的数组），然后是四四归并（将两个大小为2的数组归并成一个有4个元素的数组），然后是八八的归并，一直下去。在每一轮归并中，最后一次归并的第二个子数组可能比第一个子数组要小，如果不是的话所有的归并中两个数组大小都应该一样，而在下一轮中字数组的大小会翻倍。

**命题H。**对于长度为N的任意数组，自底向上的归并排序需要至次比较，最多访问数组次。

**证明。**处理一个数组的遍数正好是（即中的n）。每一遍会访问数组6N次，比较次数在N/2和N之间。

当数组长度为2的幂时，自顶向下和自底向上的归并排序所用的比较次数和数组访问次数正好相同，只是顺序不同。其他时候，两种方法的比较和数组访问的次序会有所不同。

自底向上的归并排序比较适合用链表组织的数据。将链表先按大小为1的子链表进行排序，然后是大小为2的子链表，然后是大小为4的子链表等。这种方法只需要重新组织链表链接就能将链表原地排序。

### 排序算法的复杂度

归并排序是证明计算复杂度领域的一个重要结论的基础，而计算复杂性能够帮助我们理解排序自身固有的难易程度。

研究复杂度的第一步是建立一个计算模型。一般来讲，研究者会尽量寻找一个和问题相关的最简单的模型。对排序来说，我们的研究对象是基于比较的算法，它们对数组元素的操作方式是由主键的比较决定的。一个基于比较的算法在两次比较之间可能会进行任意规模的计算，但它只能通过主键之间的比较得到关于某个主键的信息。

**命题I。**没有任何基于比价的算法能够保证使用少于次比较将长度为N的数组排序。

**证明。**首先，假设没有重复的主键，因为任何排序算法都必须能够处理这种情况。我们使用二叉树来表示所有的比较。树中的节点要么是一片叶子，表示排序完成且原输入的排列顺序是，要么是一个内部节点，表示a[i]和a[j]之间的一次比较操作，它的左子树表示a[i]小于a[j]时进行的其他的比较，右子树表示a[i]大于a[j]时进行的其他比较。从根节点到叶子节点每一条路径都对应着算法在建立叶子节点说是的顺序时进行的所有比较。

从比较树观察得到的第一个重要结论是这棵树应该至少有N！个叶子节点，因为N个不同的主键会有N！种不同的排列。如果叶子节点少于N！，那肯定有一些排列顺序被遗漏了。算法对于那些被遗漏的输入肯定会失败。

从根节点到叶子节点的一条路径上的内部节点的数量即是某种输入下算法进行比较的次数。我们感兴趣的是这种路径有多长（也就是树的高度），因为这也就是算法比较次数的最坏情况。二叉树的一个基本的组合学性质就是高度为h的树最多只能可能有个叶子节点，拥有个叶子节点的树是完美平衡的，或称为完全树。

结合以上分析可知：

h的值就是最坏情况下的比较次数，因此不等式两边取对数即可得到任意算法的比价次数至少是。根据斯特灵公式对阶乘函数的近似可得。

**命题J。**归并排序是一种渐进最优的基于比较的排序的算法。

**证明。**更准确的说，这句话的意思是，归并排序在最坏的情况下的比较次数和任意基于比较的排序算法所需的最少比较次数都是。命题H和命题I证明了这些结论。

局限：

* 归并排序的空间复杂度不是最优的。
* 在实践中不一定会遇到最坏情况。
* 处理比较，算法的其他操作（例如访问数组）也可能很重要。
* 不进行比较也能将某些数组排序。

## 快速排序

快速排序是应用最广泛的排序算法。快速排序流行的原因是它实现简单、适用于各种不同的输入数据且在一般应用中比其他排序算法都要快得多。快速排序引人注目的特点是它是原地排序（只需要一个很小的辅助栈），且将长度为N的数组排序所需的时间和成正比。其他排序算法都无法将这两个优点结合在一起。另外，快速排序的内循环比大多数排序算法都要短小，这意味着它无论在理论还是实际中都要更快。它的主要缺点是非常脆弱，在实现时要非常小心才能避免低劣的性能。

### 基本算法

快速排序是一种分治的排序算法。它将数组分成两个子数组，将两部分独立的排序。快速排序和对并排序是互补的：归并排序将数组分成连个字数组分别排序，并将有序的子数组归并以将整个数组排序；而快速排序将数组排序的方式则是当两个子数组都有序时整个数组也就自然有序了。在第一种情况中，递归调用发生在处理整个数组之前；在第二种情况中，递归调用发生在处理整个数组之后。在归并排序中，一个数组被等分为两半；在快速排序中，切分的位置取决于数组的内容。



该方法的关键在于切分，这个过程使得数组满足下面三个条件：

* 对于某个j，a[j]已经排定；
* a[lo]到a[j-1]中的所有元素都不大于a[j]；
* a[j+1]到a[hi]中的所有元素都不小于a[j]。

要完成这个实现，需要实现切分方法。一般策略是先随意的取a[lo]作为切分元素，即那个将会被排定的元素，然后我们从数组的左端开始向右扫描直到找到一个大于等于它的元素，在从数组的右端开始向左边扫描直到找到一个小于等于它的元素。这两个元素显然是没有排定的，因此我们交换它们的位置。如此继续，我们就可以保证左指针i的左侧元素都不大于切分元素，右指针j的右侧元素都不小于切分元素。当两个指针相遇时，我们只需要将切分元素a[lo]和左子数组最右侧的元素（a[j]）交换然后返回j即可。

命题K。将长度为N的无重复数组排序，快速排序平均需要次比较（以及1/6的交换）。

证明。令为将N个不同元素排序平均所需的比较次数。显然==0，对于N>1，由递归程序可以得到以下归纳关系：

第一项是切分的成本（总是N+1），第二项是将左子数组（长度可能是0到N-1）排序的平均成本，第三项是将右字数组（长度和左子数组长度相加为N-1）排序的平均成本。将等式两边乘以N得到：

将该式减去N-1时的相同等式可得：

将等式两边同除以，得到：

依归纳法可得：

括号内的量是曲线2/x下从3到N的离散近似面积加1，积分得到。注意到，也就是说平均比较次数只比最好情况多39%。

尽管快速排序有很多优点，它的基本实现仍有一个潜在的缺点:在切分不平衡的时这个程序可能会极为低效。例如：假如第一次从最小的元素切分，第二次从第二小的元素切分，如此这般，每次调用只会移除一个元素。这会导致一个大子数组需要切分很多次。

命题L。快速排序最多需要约次比较，但随机打乱数组能够预防这种情况。

证明。根据命题K，在每次切分后两个子数组之一总是空的情况下，比较次数为：

这不仅说明算法所需的时间是平方级别的，也显示了算法所需的空间是线性的，而这对于大数组来说是不可接受的。但是通过扩展对一般情况的分析我们可以得到比较次数的标准差是0.65N。因此，随着N的增大，运行时间会趋近于平均数，且不可能与平均数偏离太大。

### 算法改进

#### 切换到插入排序

和大多数递归排序算法一样，改进快速排序性能的一个简单办法是基于以下两点：

* 对于小数组，快速排序比插入排序慢；
* 因为递归，快速排序的sort方法在小数组中也会调用自己。

因此，在排序小数组时应该切换到插入排序。

#### 三样取切分

改进快速排序性能的第二个办法是使用子数组的一小部分元素的中位数来切分数组。这样做得到的切分更好，但代价是需要计算中位数。人们发现将取样大小设为3并用大小居中的元素切分的效果最好。我们还可以将取样元素放在数组末尾作为“哨兵”来去掉partition中的数组边界测试。

#### 熵最优的排序

在有大量重复元素的情况下，快速排序的递归性会使元素全部重复的子数组经常出现，这就有很大的改进潜力，将当前线性对数级的性能提高到线性级别。

一个简单的想法是将数组切分为三部分，分别对应于小于、等于和大于切分元素的数组元素。这种切分实现起来比二分法更复杂。

Dijkstra从左到右遍历数组一次，维护一个指针lt使得a[lo...lt]中的元素都小于v，一个指针gt使得a[gt+1...hi]中的元素都大于v，一个指针i使得a[lt...i-1]中的元素都等于v，a[i...gt]中的元素都还未确定。一开始i和lo相等，对a[i]进行三向比较来直接处理以下情况：

* a[i]小于v，将a[lt]和a[i]交换，将lt和i加1；
* a[i]大于v，将a[gt]和a[i]交换，将gt减1；
* a[i]等于v，将i加1。

这些操作都会保证数组元素不变且缩小gt-i的值（这样循环才会结束）。另外，除非和切分元素相等，其他元素都会被交换。

## 优先队列

许多应用程序都需要处理有序的元素，但不一定要求他们全部有序，或是不一定要一次就将它们排序。很多情况下我们会收集一些元素，处理当前键值最大的元素，然后再收集更多的元素，在处理当前键值最大的元素，如此这般。

在这种情况下，一个合适的数据结构应该支持两种操作：删除最大元素和插入元素。这种数据结构叫做优先队列。

### 堆的定义

数据结构**二叉堆**能够很好地实现优先队列的基本操作。在二叉堆的数组中，每个元素都要保证大于等于另两个特定位置的元素。相应地，这些位置的元素又至少要大于等于数组中的另两个元素。

**定义。**当一个二叉树的每个结点都大于等于它的两个子结点时，它被称为堆有序。

相应得，在堆有序的二叉树中，每个结点都小于等于它的父结点（如果有的话）。从任意结点向上，我们都能得到一列非递减的元素；从任意结点向上，我们都能得到一列非递增的元素。

**命题O。**根结点是堆有序的二叉树中的最大结点。

**定义。二叉堆**是一组能够用堆有序的完全二叉树排序的元素，并在数组中按照层级储存（不使用数组的第一个位置）。

在一个堆中，位置k的结点的父结点的位置为[k/2]，而它的两个子结点的位置则分别为2k和2k+1。

命题P。一棵大小为N的完全二叉树的高度为[log2N]。

### 堆的算法

遍历堆并按照要求将堆恢复的有序状态叫做**堆的有序化**。

#### 由下至上的堆有序化（上浮）

如果堆的有序状态因为某个结点变得比它的的父结点更大而被打破，那么我们就需要通过交换它和它的父结点来修复堆。交换后，这个结点比它的两个子结点都大，但这个结点仍然可能比它现在的父结点更大。我们可以一遍遍地用同样的办法恢复秩序。将这个结点不断的向上移动直到我们遇到一个更大的父结点。

#### 由上至下的堆有序化（下沉）

如果堆的有序状态因为某个结点变得变得比它的两个子结点或是其中之一更小了而被打破了，那么我们通过它和它的两个子结点中的较大者交换来恢复堆。交换可能会在子结点处继续打破有序状态，因此我们需要不断地用相同的方式将其修复，将结点向下移动直到它的子结点都比它更小或是到达了堆的底部。

插入元素。我们将新元素加到数组末尾，增加堆的大小并让这个新元素上浮到合适的位置。

删除最大元素。我们从数组顶端删去最大的元素并将数组的最后一个元素放到顶端，减小堆的大小并让这个元素下沉到合适的位置。

基于堆的优先队列能够保证插入元素和删除最大元素这两个操作的用时和队列的大小仅成对数关系。

**命题Q。**对于一个含有N个元素的基于堆的优先队列，插入元素操作只需要不超过（log2N+1）次比较，删除最大元素的操作需要不超过2log2N次比较。

**证明。**由命题P可知，两种操作都需要在根结点和堆底之间移动元素，而路径的长度不超过log2N。对于路径上的每个结点，删除最大元素需要两次比较（除了堆底元素），一次用来找出较大的子结点，一次用来去顶该子结点是否需要上浮。

命题Q意味着使用有序或是无序的优先队列的初级实现总是需要线性时间来完成其中的一种操作，但基于堆的实现则能够保证在对数时间内完成它们。

#### 多叉堆

#### 调整数组大小

#### 元素的不可变性

#### 索引优先队列

### 堆排序

我们可以把任意优先队列变成一种排序方法。将所有元素插入一个查找最小元素的优先队列，然后再重复调用删除最小元素的操作来将他们按顺序删去。用无序数组实现的优先队列这么做相当于进行一次插入排序。用基于堆的优先队列这样做是一种全新的排序方法。

堆排序可以分成两个阶段。在**堆的构造**阶段中，我们将原始数组重新组织安排进一个堆；然后在**下沉阶段**，我们从堆中按递减顺序取出所有元素并得到排序结果。我们在排序时可以将需要排序的数组本身作为堆，直接使用swim和sink操作，无需额外的空间。

#### 堆的构造

我们可以在与Nlog2N成正比的时间内构造一个堆，只需要从左至右遍历数组，用swim保证扫描指针左侧的所有元素已经是一颗堆有序的完全树即可，就像连续向优先队列中插入元素一样。

更聪明更高效的办法是从右至左用sink函数构造子堆。数组的每个位置都已经是一个子堆的根结点了，sink对于这些子堆也适用。如果一个结点的两个子结点都已经是堆了，那么在该结点调用sink可以将它们变成一个堆。这个过程会递归的建立起堆的秩序。开始时我们只需要扫描数组中的一半元素，因为我们可以跳过大小为1的子堆。最后我们在位置1上调用sink方法，扫描结束。

**命题R。**用下沉操作由N个元素构造堆只需少于2N次比较以及少于N次交换。

证明。观察可知，构造过程中处理的堆都比较小。例如，要构造一个127个元素的堆，我们会处理32个大小为3的堆，16个大小为7的堆，8个大小为15的堆，4个大小为31的堆，2个大小为63的堆和1个大小为127的堆，因此（最坏情况下）需要32×1+16×2+8×3+4×4+2×5+1×6=120次交换（以及两倍的比较）。

#### 下沉排序

堆排序的主要工作都在第二阶段完成。我们将堆中的最大元素删除，然后放入堆缩小后空出的位置。这个过程和选择排序有些类似，但所需的比较要少的多，因为堆提供了一种从未排序的部分找到最大元素的有效方法。

**命题S。**将N个元素排序，堆排序只需要少于（2Nlog2N+2N）次比较（以及一般次数的交换）。

**证明。**2N项来自于堆的构造。2Nlog2N项来自于每次下沉操作最大可能需要2log2N次比较。

#### 先下沉后上浮

堆排序在排序复杂性的研究中有着重要的位置。