第12章

排序

此前各章已结合具体的数据结构，循序渐进地介绍过多种基本的排序算法：2.8节和3.5节分别针对向量和列表，统一以排序器的形式实现过起泡排序、归并排序、插入排序以及选择排序等算法；9.4.1节也曾按照散列的思路与手法，实现过桶排序算法；9.4.3节还将桶排序推广至基数排序算法；10.2.5节也曾完美地利用完全二叉堆的特长，实现过就地堆排序算法。

本章着重于高级排序算法。与以上基本算法一样，其构思与技巧各具特色，在不同应用中的效率也各有千秋。因此在学习过程中，唯有更多地关注不同算法之间细微而本质的差异，留意体会其优势与不足，方能做到运用自如，并结合实际问题的需要，合理取舍与并适当改造。

§12.1 快速排序

12.1.1 分治策略

与归并排序算法一样，快速排序（quicksort）算法[①]也是分治策略的典型应用，但二者之间也有本质区别。2.8.3节曾指出，归并排序的计算量主要消耗于有序子向量的归并操作，而子向量的划分却几乎不费时间。快速排序恰好相反，它可以在$O(1)$时间内，由子问题的解直接得到原问题的解；但为了将原问题划分为两个子问题，却需要$O(n)$时间。

快速排序算法虽然能够确保，划分出来的子任务彼此独立，并且其规模总和保持渐进不变，却不能保证两个子任务的规模大体相当——实际上，甚至有可能极不平衡。因此，该算法并不能保证最坏情况下的$O(n\log n)$时间复杂度。尽管如此，它仍然受到人们的青睐，并在实际应用中往往成为首选的排序算法。究其原因在于，快速排序算法易于实现，代码结构紧凑简练，而且对于按通常规律随机分布的输入序列，快速排序算法实际的平均运行时间较之同类算法更少。

下面结合向量介绍该算法的原理，并针对实际需求相应地给出不同的实现版本。

12.1.2 轴点

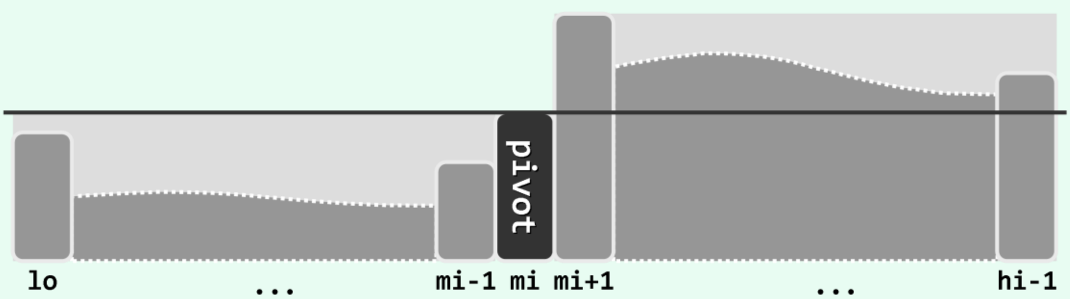

图12.1 序列的轴点（这里用高度来表示各元素的大小）

① 由英国计算机科学家、1980年图灵奖得主C. A. R. Hoare爵士于1960年发明[64]

如图12.1所示，考查任一向量区间S[lo, hi)。对于任何lo ≤ mi < hi，以元素S[mi]为界，都可分割出前、后两个子向量S[lo, mi)和S(mi, hi)。若S[lo, mi)中的元素均不大于S[mi]，且S(mi, hi)中的元素均不小于S[mi]，则元素S[mi]称作向量S的一个轴点（pivot）。

设向量S经排序可转化为有序向量S'。不难看出，轴点位置mi必然满足如下充要条件：

a) S[mi] = S'[mi]
b) S[lo, mi)和S'[lo, mi)的成员完全相同
c) S(mi, hi)和S'(mi, hi)的成员完全相同

因此，不仅以轴点S[mi]为界，前、后子向量的排序可各自独立地进行，而且更重要的是，一旦前、后子向量各自完成排序，即可立即（在$\mathcal{O}(1)$时间内）得到整个向量的排序结果。

采用分治策略，递归地利用轴点的以上特性，便可完成原向量的整体排序。

12.1.3 快速排序算法

按照以上思路，可作为向量的一种排序器，实现快速排序算法如代码12.1所示。

```
template <typename T> //向量快速排序
void Vector<T>::quickSort ( Rank lo, Rank hi ) { //0 <= lo < hi <= size
   if ( hi - lo < 2 ) return; //单元素区间自然有序，否则...
   Rank mi = partition ( lo, hi - 1 ); //在[lo, hi - 1]内构造轴点
   quickSort ( lo, mi ); //对前缀递归排序
   quickSort ( mi + 1, hi ); //对后缀递归排序
}
```

代码12.1 向量的快速排序

可见，轴点的位置一旦确定，则只需以轴点为界，分别递归地对前、后子向量实施快速排序；子向量的排序结果就地返回之后，原向量的整体排序即告完成。算法的核心与关键在于：

轴点构造算法partition()应如何实现？可以达到多高的效率？

12.1.4 快速划分算法

■ 反例

事情远非如此简单，我们首先遇到的困难就是，并非每个向量都必然含有轴点。以如图12.2所示长度为9的向量为例，不难验证，其中任何元素都不是轴点。

1	2	3	4	5	6	7	8	0
0	1	2	3	4	5	6	7	8

图12.2 有序向量经循环左移一个单元后，将不含任何轴点

事实上根据此前的分析，任一元素作为轴点的必要条件之一是，其在初始向量S与排序后有序向量S'中的秩应当相同。因此反过来一般地，只要向量中所有元素都是错位的——即所谓的错排序列——则任何元素都不可能是轴点。

由上可见，若保持原向量的次序不变，则不能保证总是能够找到轴点。因此反过来，唯有通过适当地调整向量中各元素的位置，方可“人为地”构造出一个轴点。

■ 思路

为在区间[lo, hi]内构造出一个轴点，首先需要任取某一元素m作为“培养对象”。

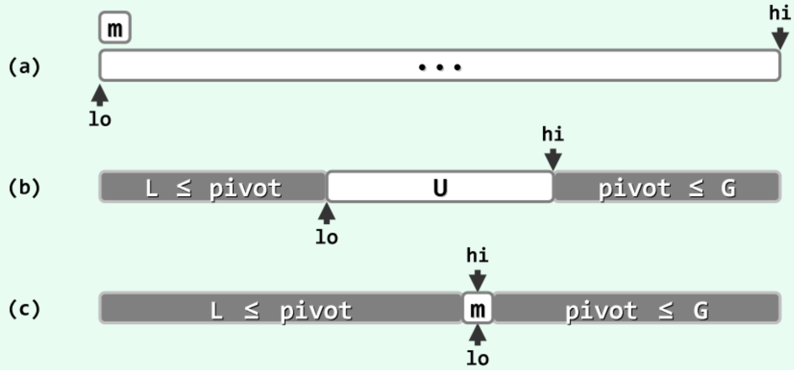

图12.3 轴点构造算法的构思

如图12.3(a)所示，不妨取首元素m = S[lo]作为候选，将其从向量中取出并做备份，腾出的空闲单元便于其它元素的位置调整。然后如图(b)所示，不断试图移动lo和hi，使之相互靠拢。当然，整个移动过程中，需始终保证lo（hi）左侧（右侧）的元素均不大于（不小于）m。

最后如图(c)所示，当lo与hi彼此重合时，只需将原备份的m回填至这一位置，则S[lo = hi] = m便成为一个名副其实的轴点。

以上过程在构造出轴点的同时，也按照相对于轴点的大小关系，将原向量划分为左、右两个子向量，故亦称作快速划分（quick partitioning）算法。

■ 实现

按照以上思路，快速划分算法可实现如代码12.2所示。

```
template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo, hi]的轴点，并返回其秩
Rank Vector<T>::partition ( Rank lo, Rank hi ) { //版本A：基本形式
   swap ( _elem[lo], _elem[lo + rand() % ( hi - lo + 1 ) ] ); //任选一个元素与首元素交换
   T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
   while ( lo < hi ) { //从向量的两端交替地向中间扫描
      while ( ( lo < hi ) && ( pivot <= _elem[hi] ) ) //在不小于pivot的前提下
         hi--; //向左拓展右端子向量
      _elem[lo] = _elem[hi]; //小于pivot者归入左侧子序列
      while ( ( lo < hi ) && ( _elem[lo] <= pivot ) ) //在不大于pivot的前提下
         lo++; //向右拓展左端子向量
      _elem[hi] = _elem[lo]; //大于pivot者归入右侧子序列
   } //assert: lo == hi
   _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
   return lo; //返回轴点的秩
}
```

代码12.2 轴点构造算法（版本A）

为便于和稍后的改进版本进行比较，不妨称作版本A。

■ 过程

可见，算法的主体框架为循环迭代；主循环的内部，通过两轮迭代交替地移动lo和hi。

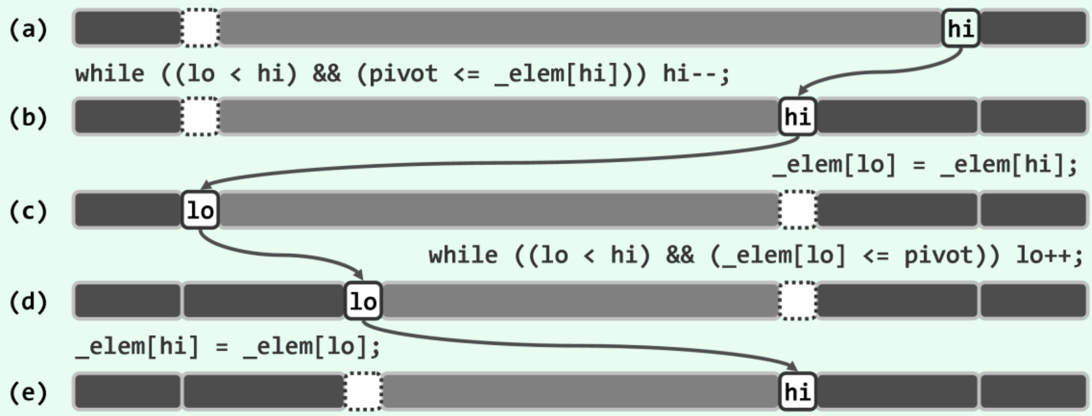

图12.4 轴点构造过程

各迭代的初始状态如图12.4(a)所示。反复地将候选轴点pivot与当前的_elem[hi]做比较，只要前者不大于后者，就不断向左移动hi（除非hi即将越过lo）。hi无法移动继续时，当如图(b)所示。于是接下来如图(c)所示，将_elem[hi]转移至_elem[lo]，并归入左侧子向量。

随后对称地，将_elem[lo]与pivot做比较，只要前者不大于后者，就不断向右移动lo（除非lo即将越过hi）。lo无法继续移动时，当如图(d)所示。于是接下来如图(e)所示，将_elem[lo]转移至_elem[hi]，并归入右侧子向量。

每经过这样的两轮移动，lo与hi的间距都会缩短，故该算法迟早会终止。当然，若如图(e)所示lo与hi仍未重合，则可再做两轮移动。不难验证，在任一时刻，在以lo和hi为界的三个子向量中，左、右子向量分别满足12.1.2节所列的轴点充要条件b)和c)。而随着算法的持续推进，中间子向量的范围则不断压缩。当主循环退出时lo和hi重合，充要条件a)也随即满足。至此，只需将pivot“镶嵌”于左、右子向量之间，即实现了对原向量的一次轴点划分。

该算法的运行时间线性正比于被移动元素的数目，线性正比于原向量的规模$\mathcal{O}$(hi - lo)。

■ 实例

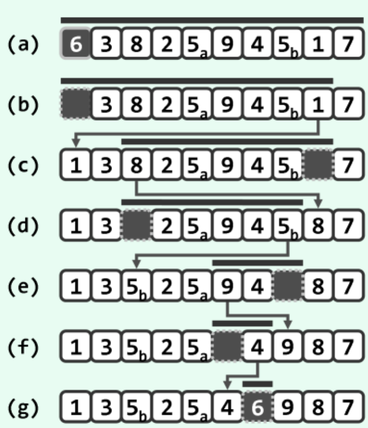

图12.5 轴点构造算法实例

快速划分算法的一次完整运行过程，如图12.5所示。输入序列A如图(a)长度为10，选择A[0] = 6作为轴点候选。以下，hi和lo的第一趟交替移动的过程及结果如图(b~c)所示，第二趟交替移动的过程及结果如图(d~e)所示，最后一趟交替移动的过程及结果如图(f~g)所示。

由于lo和hi的移动方向相反，故原处于向量右（左）端较小（大）的元素将按颠倒的次序转移至左（右）端；特别地，重复的元素也将按颠倒的次序转移至相对的一端，因而不再保持其原有的相对次序。由此可见，如此实现的快速排序算法并不稳定。从图12.5实例中数值为5的两个元素的移动过程与最终效果，不难看出这一点。

12.1.5 复杂度

■ 最坏情况

上节的分析结论指出，采用代码12.2中的partition()算法，可在线性时间内将原向量的排序问题分解为两个相互独立、总体规模保持线性的子向量排序问题；而且根据轴点的性质，由各自排序后的子向量，可在常数时间内得到整个有序向量。也就是说，分治策略得以高效实现的两个必要条件——子问题划分的高效性及其相互之间的独立性——均可保证。然而尽管如此，另一项关键的必要条件——子任务规模接近——在这里却无法保证。事实上，由partition()算法划分出的子任务在规模上不仅不能保证接近，而且可能相差悬殊。

反观partition()算法不难发现，其划分所得子序列的长度与划分的具体过程无关，而是完全取决于入口处所选的候选轴点。具体地，若在最终有序向量中该候选元素的秩为r，则子向量的规模必为r和n - r - 1。特别地，r = 0时子向量规模分别为0和n - 1——左侧子向量为空，而右侧子向量与原向量几乎等长。当然，对称的r = n - 1亦属最坏情况。

更糟糕的是，这类最坏情况可能持续发生。比如，若每次都是简单地选择最左端元素_elem[lo]作为候选轴点，则对于完全（或几乎完全）有序的输入向量，每次（或几乎每次）划分的结果都是如此。这种情况下，若将快速排序算法处理规模为n的向量所需的时间记作T(n)，则如下递推关系始终成立：

$$T(n) = T(0) + T(n - 1) + \mathcal{O}(n) = T(n - 1) + \mathcal{O}(n)$$

综合考虑到其常数复杂度的递归基，与以上递推关系联立即可解得：

$$T(n) = T(n - 2) + 2\cdot\mathcal{O}(n) = \ldots = T(0) + n\cdot\mathcal{O}(n) = \mathcal{O}(n^2)$$

也就是说，其效率居然低到与起泡排序相近。

■ 降低最坏情况概率

那么，如何才能降低上述最坏情况出现的概率呢？读者可能已注意到，代码12.2的partition()算法在入口处增加了swap()一句，在区间内任选一个元素与_elem[lo]交换。就其效果而言，这使得后续的处理等同于随机选择一个候选轴点，从而在一定程度上降低上述最坏情况出现的概率。这种方法称作随机法。

类似地，也可采用所谓三者取中法：从待排序向量中任取三个元素，将数值居中者作为候选轴点。理论分析及实验统计均表明，较之固定选取某个元素或随机选取单个元素的策略，如此选出的轴点在最终有序向量中秩过小或过大的概率更低——尽管还不能彻底杜绝最坏情况。

■ 平均运行时间

以上关于最坏情况下效率仅为$\mathcal{O}(n^2)$的结论不免令人沮丧，难道快速排序名不副实？实际上，更为细致的分析与实验统计都一致地显示，在大多数情况下，快速排序算法的平均效率依然可以达到$\mathcal{O}(n\log n)$；而且较之其它排序算法，其时间复杂度中的常系数更小。以下就以最常见的场景为例，对采用随机法确定候选轴点的快速排序算法的平均效率做一估算。

假设待排序的元素服从独立均匀随机分布。于是，partition()算法在经过n - 1次比较和至多n + 1次移动操作之后，对规模为n的向量的划分结果无非n种可能，划分所得左侧子序列的长度分别是0, 1, ..., n - 1，分别决定于所取候选元素在最终有序序列中的秩。按假定条件，每种情况的概率均为1/n，故若将算法的平均运行时间记作$\hat{T}(n)$，则有：

$$\hat{T}(n) = (n + 1) + (1/n) \times \sum_{k=1}^{n}[\hat{T}(k - 1) + \hat{T}(n - k)]$$

$$= (n + 1) + (2/n) \times \sum_{k=1}^{n}\hat{T}(k - 1)$$

等式两侧同时乘以n，则有：

$$n\cdot\hat{T}(n) = (n + 1)\cdot n + 2\cdot\sum_{k=1}^{n}\hat{T}(k - 1)$$

以及同理：

$$(n - 1)\cdot\hat{T}(n - 1) = (n - 1)\cdot n + 2\cdot\sum_{k=1}^{n-1}\hat{T}(k - 1)$$

以上两式相减，即得：

$$n\cdot\hat{T}(n) - (n - 1)\cdot\hat{T}(n - 1) = 2n + 2\cdot\hat{T}(n - 1)$$

$$n\cdot\hat{T}(n) = (n + 1)\cdot\hat{T}(n - 1) + 2n$$

$$\begin{aligned}\hat{T}(n)/(n + 1) &= \hat{T}(n - 1)/n + 2/(n + 1)\\ &= \hat{T}(n - 2)/(n - 1) + 2/(n + 1) + 2/n\\ &= \hat{T}(n - 3)/(n - 2) + 2/(n + 1) + 2/n + 2/(n - 1)\\ &= \dots\\ &= \hat{T}(0)/1 + 2/(n + 1) + 2/n + 2/(n - 1) + \dots + 2/2\\ &= 2\cdot\sum_{k=1}^{n+1}(1/k) - 1\\ &\overset{②}{=} \mathcal{O}(2\cdot\ln n) = \mathcal{O}(2\cdot\ln 2\cdot\log_2 n) = \mathcal{O}(1.386\cdot\log_2 n)\end{aligned}$$

正因为其良好的平均性能，加上其形象直观和易于实现的特点，快速排序算法自诞生起就一直受到人们的青睐，并被集成到Linux和STL等环境中。

12.1.6 应对退化

■ **重复元素**

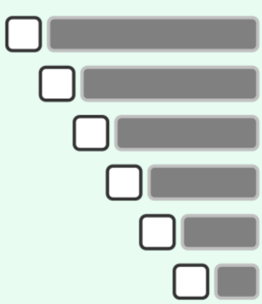

图12.6 partition()算法的退化情况，也是最坏情况

考查所有（或几乎所有）元素均重复的退化情况。对照代码12.2不难发现，partition()算法的版本A对此类输入的处理完全等效于此前所举的最坏情况。事实上对于此类向量，主循环内部前一子循环的条件中“pivot <= _elem[hi]”形同虚设，故该子循环将持续执行，直至“lo < hi”不再满足。当然，在此之后另一内循环及主循环也将随即结束。

如图12.6所示，如此划分的结果必然是以最左端元素为轴点，原向量被分为极不对称的两个子向量。更糟糕的是，这一最坏情况还可能持续发生，从而使整个算法过程等效地退化为线性递归，递归深度为$\mathcal{O}(n)$，导致总体运行时间高达$\mathcal{O}(n^2)$。

② 若记$h(n) = 1 + 1/2 + 1/3 + \dots + 1/n$，则有$\ln(n+1) = \int_{i=1}^{n+1}(1/x) < h(n) < 1 + \int_{i=1}^{n}(1/x) = 1 + \ln n$

当然，可以在每次深入递归之前做统一核验，若确属退化情况，则无需继续递归而直接返回。但在重复元素不多时，如此不仅不能改进性能，反而会增加额外的计算量，总体权衡后得不偿失。

■ 改进

轴点构造算法可行的一种改进方案如代码12.3所示。为与如代码12.2所示同名算法版本A相区别，不妨称作版本B。

```
template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo, hi]的轴点，并返回其秩
Rank Vector<T>::partition ( Rank lo, Rank hi ) { //版本B：可优化处理多个关键码雷同的退化情况
   swap ( _elem[lo], _elem[lo + rand() % ( hi - lo + 1 ) ] ); //任选一个元素与首元素交换
   T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
   while ( lo < hi ) { //从向量的两端交替地向中间扫描
      while ( lo < hi )
         if ( pivot < _elem[hi] ) //在大于pivot的前提下
            hi--; //向左拓展右端子向量
         else //直至遇到不大于pivot者
            { _elem[lo++] = _elem[hi]; break; } //将其归入左端子向量
      while ( lo < hi )
         if ( _elem[lo] < pivot ) //在小于pivot的前提下
            lo++; //向右拓展左端子向量
         else //直至遇到不小于pivot者
            { _elem[hi--] = _elem[lo]; break; } //将其归入右端子向量
   } //assert: lo == hi
   _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
   return lo; //返回轴点的秩
}
```

代码12.3 轴点构造算法（版本B）

较之版本A，版本B主要是调整了两个内循环的终止条件。以前一内循环为例，原条件

```
pivot <= _elem[hi]
```

在此更改为：

```
pivot < _elem[hi]
```

也就是说，一旦遇到重复元素，右端子向量随即终止拓展，并将右端重复元素转移至左端。因此，若将版本A的策略归纳为“勤于拓展、懒于交换”，版本B的策略则是“懒于拓展、勤于交换”。

■ 效果及性能

对照代码12.3不难验证，对于由重复元素构成的输入向量，以上版本B将交替地将右（左）侧元素转移至左（右）侧，并最终恰好将轴点置于正中央的位置。这就意味着，退化的输入向量能够始终被均衡的切分，如此反而转为最好情况，排序所需时间为$\mathcal{O}$(nlogn)。

当然，以上改进并非没有代价。比如，单趟partition()算法需做更多的元素交换操作。好在这并不影响该算法的线性复杂度。另外，版本B倾向于反复交换重复的元素，故它们在原输入向量中的相对次序更难保持，快速排序算法稳定性的不足更是雪上加霜。

§12.2 *选取与中位数

12.2.1 概述

■ k-选取

考查如下问题：

在任意一组可比较大小的元素中，如何找出由小到大次序为k者？

如图12.7(a)所示，也就是要从与这组元素对应的有序序列S中，找出秩为k的元素S[k]，故称作选取（selection）问题。若将目标元素的秩记作k，则亦称作k-选取（k-selection）问题。以无序向量A = { 3, 13, 2, 5, 8 }为例，对应的有序向量为S = { 2, 3, 5, 8, 13 }，其中的元素依次与k = { 0, 1, 2, 3, 4 }相对应。

图12.7 选取与中位数

作为k-选取问题的特例，0-选取即通常的最小值问题，而(n - 1)-选取问题即通常的最大值问题。这两个问题都有平凡的最优解，例如List::selectMax()（82页代码3.21）。

在允许元素重复的场合，秩为k的元素可能同时存在多个副本。此时不妨约定，其中任何一个都可作为解答输出。

■ 中位数

如图12.7(b)所示，在长度为n的有序序列S中，位序居中的元素S[⌊n/2⌋]称作中值或中位数（median）。例如，有序序列S = { 2, 3, $\boxed{5}$, 8, 13 }的中位数，为S[⌊5/2⌋] = S[2] = 5；而有序序列S = { 2, 3, 5, $\boxed{8}$, 13, 21 }的中位数，则为S[⌊6/2⌋] = S[3] = 8。

即便对于尚未排序的序列，也可定义中位数——也就是在对原数据集排序之后，对应的有序序列的中位数。例如，无序序列A = { 3, 13, 2, $\boxed{5}$, 8 }的中位数为元素A[3] = 5。

由于中位数可将原数据集（原问题）划分为大小明确、规模相仿且彼此独立的两个子集（子问题），故能否高效地确定中位数，将直接关系到采用分治策略的算法能否高效地实现。

■ 蛮力算法

由中位数的定义，可直接得到查找中位数的如下直觉算法：对所有元素做排序，将其转换为有序序列S；于是，S[⌊n/2⌋]便是所要找的中位数。然而根据2.7.5节的结论，该算法在最坏情况下需要Ω(nlogn)时间。于是，基于该算法的任何分治算法，时间复杂度都会不低于：

$$T(n) \ = \ n\log n + 2\cdot T(n/2) \ = \ \mathcal{O}(n\log^2 n)$$

这一效率难以令人接受。

综上可见，中位数查找问题的挑战恰恰就在于：

如何在避免全排序的前提下，在$\mathcal{o}$(nlogn)时间内找出中位数？

不难看出，所谓中位数查找问题，也可以理解为是选取问题在k = ⌊n/2⌋时的特例。稍后我们将看到，中位数查找问题既是选取问题的特例，同时也是选取问题中的难度最大者。

以下先结合若干特定情况讨论中位数的定位算法，然后再回到一般性的选取问题。

12.2.2 众数

■ 问题

为达到热身的目的，不妨先来讨论中位数问题的一个简化版本。在任一无序向量A中，若有一半以上元素的数值同为m，则将m称作A的众数（majority）。例如，向量{ 5, $\boxed{3}$, 9, $\boxed{3}$, $\boxed{3}$, 2, $\boxed{3}$, $\boxed{3}$ }的众数为3；而虽然3在向量{ 5, $\boxed{3}$, 9, $\boxed{3}$, 1, 2, $\boxed{3}$, $\boxed{3}$ }中最多，确非众数。

那么，任给无序向量，如何快速判断其中是否存在众数，并在存在时将其找出？尽管只是以整数向量为例，以下算法不难推广至元素类型支持判等和比较操作的任意向量。

■ 必要性与充分性

不难理解但容易忽略的一个事实是：若众数存在，则必然同时也是中位数。否则，在对应的有序向量中，总数超过半数的众数必然被中位数分隔为非空的两组——与向量的有序性相悖。

```
template <typename T> bool majority ( Vector<T> A, T& maj ) { //众数查找算法：T可比较可判等
   maj = majEleCandidate ( A ); //必要性：选出候选者maj
   return majEleCheck ( A, maj ); //充分性：验证maj是否的确当选
}
```

代码12.4 众数查找算法主体框架

因此可如代码12.4所示，通过调用majEleCandidate()，从向量A中找到中位数maj（如果的确可以高效地查找到的话），并将其作为众数的唯一候选者。

然后再如代码12.5所示，调用majEleCheck()在线性时间内扫描一遍向量，通过统计该中位数出现的次数，即可验证其作为众数的充分性，从而最终判断向量A的众数是否的确存在。

```
template <typename T> bool majEleCheck ( Vector<T> A, T maj ) { //验证候选者是否确为众数
   int occurrence = 0; //maj在A[]中出现的次数
   for ( int i = 0; i < A.size(); i++ ) //逐一遍历A[]的各个元素
      if ( A[i] == maj ) occurrence++; //每遇到一次maj，均更新计数器
   return 2 * occurrence > A.size(); //根据最终的计数值，即可判断是否的确当选
}
```

代码12.5 候选众数核对算法

那么，在尚未得到高效的中位数查找算法之前，又该如何解决众数问题呢？

■ 减而治之

关于众数的另一重要事实，如图12.8所示：

设P为向量A中长度为2m的前缀。若元素x在P中恰好出现m次，则A有众数仅当后缀A-P拥有众数，且A-P的众数就是A的众数。

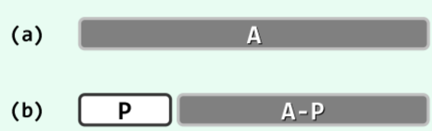

图12.8 通过减治策略计算众数

既然最终总会针对充分性另作一次核对，故不必担心A不含众数的情况，而只需验证A的确拥有众数的两种情况。若A的众数就是x，则在剪除前缀P之后，x与非众数均减少相同的数目，二者数目的差距在后缀A-P中保持不变。反过来，若A的众数为y ≠ x，则在剪除前缀P之后，y减少的数目也不致多于非众数减少的数目，二者数目的差距在后缀A-P中也不会缩小。

■ 实现

以上减而治之策略，可以实现为如代码12.6所示的majEleCandidate()算法。利用该算法，自左向右地扫描一遍整个向量，即可唯一确定满足如上必要条件的某个候选者。

```
template <typename T> T majEleCandidate ( Vector<T> A ) { //选出具备必要条件的众数候选者
   T maj; //众数候选者
// 线性扫描：借助计数器c，记录maj与其它元素的数量差额
   for ( int c = 0, i = 0; i < A.size(); i++ )
      if ( 0 == c ) { //每当c归零，都意味着此时的前缀P可以剪除
         maj = A[i]; c = 1; //众数候选者改为新的当前元素
      } else //否则
         maj == A[i] ? c++ : c--; //相应地更新差额计数器
   return maj; //至此，原向量的众数若存在，则只能是maj —— 尽管反之不然
}
```

代码12.6 候选众数选取算法

其中，变量maj始终为当前前缀中出现次数不少于一半的某个元素；c则始终记录该元素与其它元素的数目之差。一旦c归零，则意味着如图12.8(b)所示，在当前向量中找到了一个可剪除的前缀P。在剪除该前缀之后，问题范围将相应地缩小至A-P。此后，只需将maj重新初始化为后缀A-P的首元素，并令c = 1，即可继续重复上述迭代过程。

对于向量的每个秩i，该算法迭代且仅迭代一步。故其运行时间，因线性正比于向量规模。

12.2.3 归并向量的中位数

■ 问题

本节继续讨论中位数问题的另一简化版本。考查如下问题：

任给有序向量S_1和S_2，如何找出它们归并后所得有序向量$S = S_1 \cup S_2$的中位数？

■ 蛮力算法

```
// 中位数算法蛮力版：效率低，仅适用于max(n1, n2)较小的情况
template <typename T> //子向量S1[lo1, lo1 + n1)和S2[lo2, lo2 + n2)分别有序，数据项可能重复
T trivialMedian ( Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2 ) {
   int hi1 = lo1 + n1, hi2 = lo2 + n2;
   Vector<T> S; //将两个有序子向量归并为一个有序向量
   while ( ( lo1 < hi1 ) && ( lo2 < hi2 ) ) {
      while ( ( lo1 < hi1 ) && S1[lo1] <= S2[lo2] ) S.insert ( S1[lo1 ++] );
      while ( ( lo2 < hi2 ) && S2[lo2] <= S1[lo1] ) S.insert ( S2[lo2 ++] );
   }
   while ( lo1 < hi1 ) S.insert ( S1[lo1 ++] );
   while ( lo2 < hi2 ) S.insert ( S1[lo2 ++] );
   return S[ ( n1 + n2 ) / 2]; //直接返回归并向量的中位数
}
```

代码12.7 中位数蛮力查找算法

诚然，有序向量S中的元素$S[\lfloor(n_1 + n_2)/2\rfloor]$即为中位数，但若果真按代码12.7中蛮力算法trivialMedian()将二者归并，则需花费$\mathcal{O}(n_1 + n_2)$时间。这一效率虽不算太低，但毕竟未能充分利用“两个子向量已经有序”的条件。那么，能否更快地完成这一任务呢？

以下首先讨论S_1和S_2长度同为n的情况，稍后再推广至不等长的情况。

■ 减而治之

如图12.9所示，考查S_1的中位数$m_1 = S_1[\lfloor n/2\rfloor]$和$S_2$的逆向中位数$m_2 = S_2[\lceil n/2\rceil - 1] = S_2[\lfloor(n - 1)/2\rfloor]$，并比较其大小。n为偶数和奇数的情况，分别如图(a)和图(b)所示。

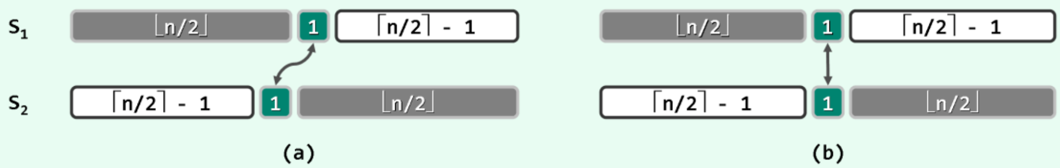

图12.9 采用减治策略，计算等长有序向量归并后的中位数

若$m_1 = m_2$，则在$S = S_1 \cup S_2$中，各有$\lfloor n/2\rfloor + (\lceil n/2\rceil - 1) = n - 1$个元素不大于和不小于它们，故$m_1$和$m_2$就是S的中位数。若$m_1 < m_2$，则意味着在S中各有$\lfloor n/2\rfloor$个元素（图中以灰色示意）不大于和不小于它们。可见，这些元素或者不是S的中位数，或者与m_1或m_2同为S的中位数。无论如何，在清除这些元素之后，S中位数的数值均保持不变。$m_1 > m_2$的对称情况，与此类似。

综合以上分析，只需进行一次比较，即可将原问题的规模缩减大致一半。利用这一性质，如此反复递归，问题的规模将持续地以1/2为比例，按几何级数的速度递减，直至平凡的递归基。

整个算法呈线性递归的形式，递归深度不超过$\log_2 n$，每一递归实例仅需常数时间，故总体时间复杂度仅为$\mathcal{O}(\log n)$——这一效率远远高于蛮力算法。

■ 实现

以上减而治之策略，可以实现为如代码12.8所示的median()算法。

```
template <typename T> //序列S1[lo1, lo1 + n)和S2[lo2, lo2 + n)分别有序，n > 0，数据项可能重复
T median ( Vector<T>& S1, int lo1, Vector<T>& S2, int lo2, int n ) { //中位数算法（高效版）
   if ( n < 3 ) return trivialMedian ( S1, lo1, n, S2, lo2, n ); //递归基
   int mi1 = lo1 + n / 2, mi2 = lo2 + ( n - 1 ) / 2; //长度（接近）减半
   if ( S1[mi1] < S2[mi2] )
      return median ( S1, mi1, S2, lo2, n + lo1 - mi1 ); //取S1右半、S2左半
   else if ( S1[mi1] > S2[mi2] )
      return median ( S1, lo1, S2, mi2, n + lo2 - mi2 ); //取S1左半、S2右半
   else
      return S1[mi1];
}
```

代码12.8 等长有序向量归并后中位数算法

在向量长度小于3之后，即调用蛮力算法trivialMedian直接计算中位数。否则，分别取出m_1和m_2，并分三种情况继续线性递归。请体会“循秩访问”方式在此所起的关键性作用。

因属于尾递归，故不难将该算法改写为迭代形式（习题[12-6]）。

一般情况

以上算法可如代码12.9所示推广至一般情况，即允许有序向量S_1和S_2的长度不等。

```
template <typename T> //向量S1[lo1, lo1 + n1)和S2[lo2, lo2 + n2)分别有序，数据项可能重复
T median ( Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2 ) { //中位数算法
   if ( n1 > n2 ) return median ( S2, lo2, n2, S1, lo1, n1 ); //确保n1 <= n2
   if ( n2 < 6 ) //递归基：1 <= n1 <= n2 <= 5
      return trivialMedian ( S1, lo1, n1, S2, lo2, n2 );
   ///////////////////////////////////////////////////////////////////////
   //                lo1            lo1 + n1/2      lo1 + n1 - 1
   //                 |                 |                 |
   //                 X >>>>>>>>>>>>>>> X >>>>>>>>>>>>>>> X
   // Y .. trimmed .. Y >>>>>>>>>>>>>>> Y >>>>>>>>>>>>>>> Y .. trimmed .. Y
   // |               |                 |                 |               |
   // lo2     lo2 + (n2-n1)/2       lo2 + n2/2     lo2 + (n2+n1)/2    lo2 + n2 -1
   ///////////////////////////////////////////////////////////////////////
   if ( 2 * n1 < n2 ) //若两个向量的长度相差悬殊，则长者（S2）的两翼可直接截除
      return median ( S1, lo1, n1, S2, lo2 + ( n2 - n1 - 1 ) / 2, n1 + 2 - ( n2 - n1 ) % 2 );
   ///////////////////////////////////////////////////////////////////////
   //    lo1                  lo1 + n1/2              lo1 + n1 - 1
   //     |                       |                       |
   //     X >>>>>>>>>>>>>>>>>>>>> X >>>>>>>>>>>>>>>>>>>>> X
   //                             |
   //                            m1
   ///////////////////////////////////////////////////////////////////////
   //                            mi2b
   //                             |
   // lo2 + n2 - 1         lo2 + n2 - 1 - n1/2
   //     |                       |
   //     Y <<<<<<<<<<<<<<<<<<<<< Y ...
   //                                .
   //                               .
   //                              .
   //                             .
   //                            .
   //                           .
   //                          .
   //                         ... Y <<<<<<<<<<<<<<<<<<<<< Y
   //                             |                       |
   //                       lo2 + (n1-1)/2               lo2
   //                             |
   //                            mi2a
   ///////////////////////////////////////////////////////////////////////
```

```
   int mi1  = lo1 + n1 / 2;
   int mi2a = lo2 + ( n1 - 1 ) / 2;
   int mi2b = lo2 + n2 - 1 - n1 / 2;
   if ( S1[mi1] > S2[mi2b] ) //取S1左半、S2右半
      return median ( S1, lo1, n1 / 2 + 1, S2, mi2a, n2 - ( n1 - 1 ) / 2 );
   else if ( S1[mi1] < S2[mi2a] ) //取S1右半、S2左半
      return median ( S1, mi1, ( n1 + 1 ) / 2, S2, lo2, n2 - n1 / 2 );
   else //S1保留，S2左右同时缩短
      return median ( S1, lo1, n1, S2, mi2a, n2 - ( n1 - 1 ) / 2 * 2 );
}
```

代码12.9 不等长有序向量归并后中位数算法

这一算法与代码12.8中同名算法的思路基本一致，请参照注释分析和验证其功能。

这里也采用了减而治之的策略，可使问题的规模大致按几何级数递减，故总体复杂度亦为$\mathcal{O}(\log(n_1 + n_2))$。更精确地，其复杂度应为$\mathcal{O}(\log(\min(n_1, n_2)))$（习题[12-7]）——也就是说，子向量长度相等或接近时，此类问题的难度更大。

12.2.4 基于优先级队列的选取

■ 信息量与计算成本

回到一般性的选取问题。蛮力算法的效率之所以无法令人满意，可以解释为：“一组元素中第k大的元素”所包含的信息量，远远少于经过全排序后得到的整个有序序列。

花费足以全排序的计算成本，却仅得到了少量的局部信息，未免得不偿失。由此看来，既然只需获取原数据集的局部信息，为何不采用更适宜于这类计算需求的优先级队列结构呢？

■ 堆

以堆结构为例。如图12.10所示，基于堆结构的选取算法大致有三种。

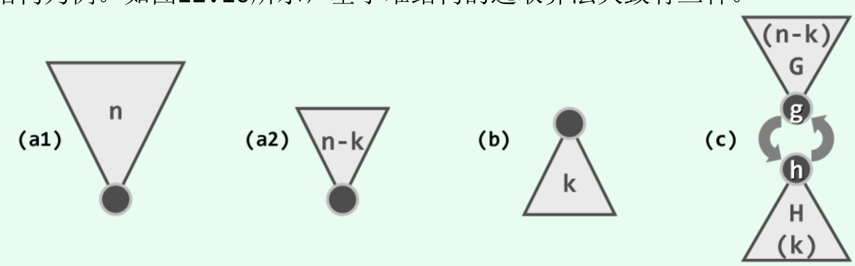

图12.10 基于堆结构的选取算法

第一种算法如图(a1)所示。首先，花费$\mathcal{O}(n)$时间将全体元素组织为一个小顶堆；然后，经过k次delMin()操作，则如图(a2)所示得到位序为k的元素。这一算法的运行时间为：

$$\mathcal{O}(n) + k\cdot\mathcal{O}(\log n) = \mathcal{O}(n + k\log n)$$

另一算法如图(b)所示。任取k个元素，并在$\mathcal{O}(k)$时间以内将其组织为大顶堆。然后将剩余的n - k个元素逐个插入堆中；每插入一个，随即删除堆顶，以使堆的规模恢复为k。待所有元素处理完毕之后，堆顶即为目标元素。该算法的运行时间为：

$$\mathcal{O}(k) + 2(n - k)\cdot\mathcal{O}(\log k) = \mathcal{O}(k + 2(n - k)\log k)$$

最后一种方法如图(c)。首先将全体元素分为两组，分别构建一个规模为n - k的小顶堆G和一个规模为k的大顶堆H。接下来，反复比较它们的堆顶g和h，只要g < h，则将二者交换，并重新调整两个堆。如此，G的堆顶g将持续增大，H的堆顶h将持续减小。当g ≥ h时，h即为所要找的元素。这一算法的运行时间为：

$$\mathcal{O}(n - k) + \mathcal{O}(k) + \min(k, n - k) \cdot 2 \cdot (\mathcal{O}(\log k + \log(n - k)))$$

在目标元素的秩很小或很大（即$|n/2 - k| \approx n/2$）时，上述算法的性能都还不错。比如，$k \approx 0$时，前两种算法均只需$\mathcal{O}(n)$时间。然而很遗憾，当$k \approx n/2$时，以上算法的复杂度均退化至蛮力算法的$\mathcal{O}(n\log n)$。因此，我们不得不转而从其它角度寻找突破口。

12.2.5 基于快速划分的选取

■ 秩、轴点与快速划分

选取问题所查找元素的位序k，就是其在对应的有序序列中的秩。就这一性质而言，该元素与轴点颇为相似。尽管12.1.4节的快速划分算法只能随机地构造一个轴点，但若反复应用这一算法，应该可以逐步逼近目标k。

■ 逐步逼近

以上构思可细化如下。首先，调用算法partition()构造向量A的一个轴点A[i] = x。若i = k，则该轴点恰好就是待选取的目标元素，即可直接将其返回。

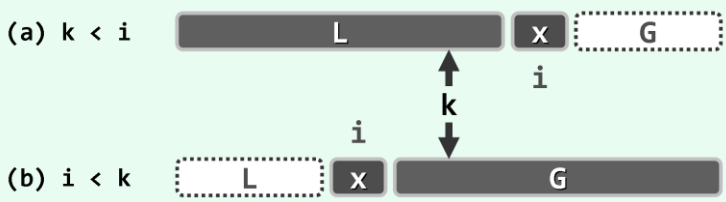

图12.11 基于快速划分算法逐步逼近选取目标元素

反之，若如图12.11所示i ≠ k，则无非两种情况。若如图(a)，k < i，则选取的目标元素不可能（仅）来自于处于x右侧、不小于x的子向量（白色）G中。此时，不妨将子向量G剪除，然后递归地在剩余区间继续做k-选取。反之若如图(b)，i < k，则选取的目标元素不可能（仅）来自于处于x左侧、不大于x的子向量（白色）L中。同理，此时也可将子向量L剪除，然后递归地在剩余区间继续做(k - i)-选取。

■ 实现

基于以上减而治之、逐步逼近的思路，可实现quickSelect()算法如代码12.10所示。

```
template <typename T> void quickSelect ( Vector<T> & A, Rank k ) { //基于快速划分的k选取算法
   for ( Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
      Rank i = lo, j = hi; T pivot = A[lo];
      while ( i < j ) { //O(hi - lo + 1) = O(n)
         while ( ( i < j ) && ( pivot <= A[j] ) ) j--; A[i] = A[j];
         while ( ( i < j ) && ( A[i] <= pivot ) ) i++; A[j] = A[i];
      } //assert: i == j
      A[i] = pivot;
```

```
        if ( k <= i ) hi = i - 1;
        if ( i <= k ) lo = i + 1;
    } //A[k] is now a pivot
}
```

代码12.10 基于快速划分的k-选取算法

该算法的流程，与代码12.2中的partition()算法（版本A）如出一辙。每经过一步主迭代，都会构造出一个轴点A[i]，然后lo和hi将彼此靠拢，查找范围将收缩至A[i]的某一侧。当轴点的秩i恰为k时，算法随即终止。如此，A[k]即是待查找的目标元素。

尽管内循环仅需$\mathcal{O}$(hi - lo + 1)时间，但很遗憾，外循环的次数却无法有效控制。与快速排序算法一样，最坏情况下外循环需执行Ω(n)次（习题[12-11]），总体运行时间为$\mathcal{O}(n^2)$。

12.2.6 k-选取算法

以上从多个角度所做的尝试尽管有所收获，但就k-选取问题在最坏情况下的求解效率这一最终指标而言，均无实质性的突破。本节将延续以上quickSelect()算法的思路，介绍一个在最坏情况下运行时间依然为$\mathcal{O}(n)$的k-选取算法。

■ 算法

该方法的主要计算流程，可描述如算法12.1所示。

```
select(A, k)
输入：规模为n的无序序列A，秩k ≥ 0
输出：A所对应有序序列中秩为k的元素
{
    0) if (n = |A| < Q) return trivialSelect(A, k); //递归基：序列规模不大时直接使用蛮力算法
    1) 将A均匀地划分为n/Q个子序列，各含Q个元素; //Q为一个不大的常数，其具体数值稍后给出
    2) 各子序列分别排序，计算中位数，并将这些中位数组成一个序列; //可采用任何排序算法，比如选择排序
    3) 通过递归调用select()，计算出中位数序列的中位数，记作M;
    4) 根据其相对于M的大小，将A中元素分为三个子集：L（小于）、E（相等）和G（大于）;
    5) if (|L| ≥ k) return select(L, k);
       else if (|L| + |E| ≥ k) return M;
       else return select(G, k - |L| - |E|);
}
```

算法12.1 线性时间的k-选取

■ 正确性

该算法正确性的关键，在于其中第5)步中所涉及的递归。

实际上如图12.12所示，在第4)步依据全局中位数M对所有元素做过分类之后，可以假想地将三个子序列L、E和G按照大小次序自左向右排列。尽管这三个子集都有可能是空集，但无论如何，k-选取目标元素的位置无非三种可能。

其一如图(a)，子序列L足够长（|L| ≥ k）。此时，子序列E和G的存在与否与k-选取的结果无关，故可将它们剪除，并在L中继续做递归的k-选取。

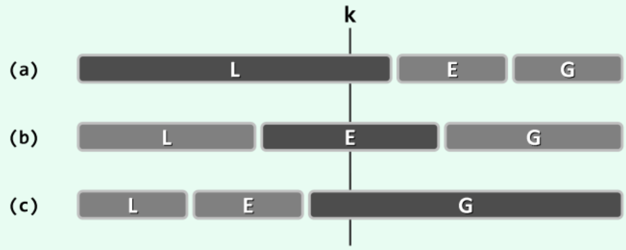

图12.12 k-选取目标元素所处位置的三种可能情况

其次如图(b)，子序列L长度不足k，但在加入子序列E之后可以覆盖k。此时，E中任何一个元素（均等于全局中位数M）都是所要查找的目标元素，故可直接返回M。

最后如图(c)，子序列L和E的长度总和仍不足k。此时，目标元素必然落在子序列G中，故可将L和E剪除，并在G中继续做递归的(k - |L| - |E|)-选取。

■ 复杂度

将该select()算法在最坏情况下的运行时间记作T(n)，其中n为输入序列A的规模。

显然，第1)步只需$\mathcal{O}(n)$时间。既然Q为常数，故在第2)步中，每一子序列的排序及中位数的计算只需常数时间，累计不过$\mathcal{O}(n)$。第3)步为递归调用，因子序列长度为n/Q，故经过T(n/Q)时间即可得到全局的中位数M。第4)步依据M对所有元素做分类，为此只需做一趟线性遍历，累计亦不过$\mathcal{O}(n)$时间。

那么，第5)步需要运行多少时间呢？考查第2)步所得各子序列的中位数。若按照这n/Q个中位数（标记为m）的大小次序，将其所属子序列顺序排列，大致应如图12.13所示。在这些中位数中的居中者，即为第3)步计算出的全局中位数M。

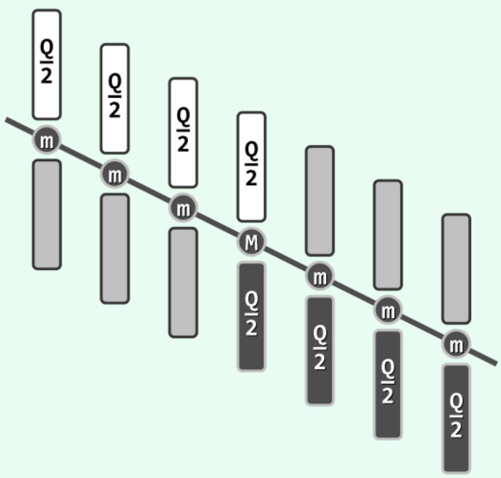

图12.13 各子序列的中位数以及全局中位数

由该图不难发现，至少有一半的子序列中，有半数的元素不小于M（在图中以白色示意）。同理，也至少有一半的子序列中，有半数的元素不大于M（在图中以黑色示意）。反过来，这两条性质也意味着，严格大于（小于）M的元素在全体元素中所占比例不会超过75%。

由此可知，子序列L与G的规模均不超过3n/4。也就是说，算法的第5)步尽管会发生递归，但需进一步处理的序列的规模，绝不致超过原序列的3/4。

综上，可得递推关系如下：

$$T(n) = cn + T(n/Q) + T(3n/4)，c为常数$$

若取Q = 5，则有

$$T(n) = cn + T(n/5) + T(3n/4) = \mathcal{O}(20cn) = \mathcal{O}(n)$$

■ 综合评价

上述selection()算法从理论上证实，的确可以在线性时间内完成k-选取。然而很遗憾，其线性复杂度中的常系数项过大，以致在通常规模的应用中难以真正体现出效率的优势。

该算法的核心技巧在于第2)和3)步，通过高效地将元素分组，分别计算中位数，并递归计算出这些中位数的中位数M，使问题的规模得以按几何级数的速度递减，从而实现整体的高性能。

由此也可看出，中位数算法在一般性k-选取问题的求解过程中扮演着关键性角色，尽管前者只不过是后者的一个特例，但反过来也是其中难度最大者。

§12.3 *希尔排序

12.3.1 递减增量策略

■ 增量

希尔排序[③]（Shellsort）算法首先将整个待排序向量A[]等效地视作一个二维矩阵B[][]。

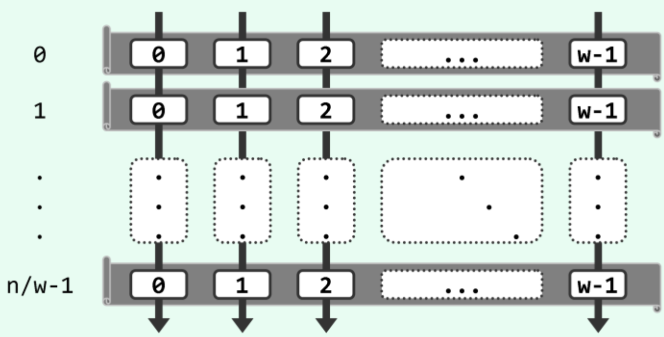

图12.14 将待排序向量视作二维矩阵

于是如图12.14所示，若原一维向量为A[0, n)，则对于任一固定的矩阵宽度w，A与B中元素之间总有一一对应关系：

$$B[i][j] = A[i + jw]$$

或

$$A[k] = B[k \% w][k / w]$$

从秩的角度来看，矩阵B的各列依次对应于整数子集[0, n)关于宽度w的某一同余类。这也等效于从上到下、自左而右地将原向量A中的元素，依次填入矩阵B的各个单元。

为简化起见，以下不妨假设w整除n。如此，B中同属一列的元素自上而下依次对应于A中以w为间隔的n/w个元素。因此，矩阵的宽度w亦称作增量（increment）。

③ 最初版本由D. L. Shell于1959年发明[65]

■ 算法框架

希尔排序的算法框架，可以扼要地描述如下：

```
Shellsort(A, n)
输入：规模为n的无序向量A
输出：A对应的有序向量
{
   取一个递增的增量序列：H = { w_1 = 1, w_2, w_3, ..., w_k, ... }
   设k = max{i | w_i < n}，即w_k为增量序列H中小于n的最后一项
   for (t = k; t > 0; t--) {
      将向量A视作以w_t为宽度的矩阵B_t
      对B_t的每一列分别排序：B_t[i]，i = 0, 1, ..., w_t - 1
   }
}
```

算法12.2 希尔排序

■ 增量序列

如图12.15所示，希尔排序是个迭代式重复的过程。

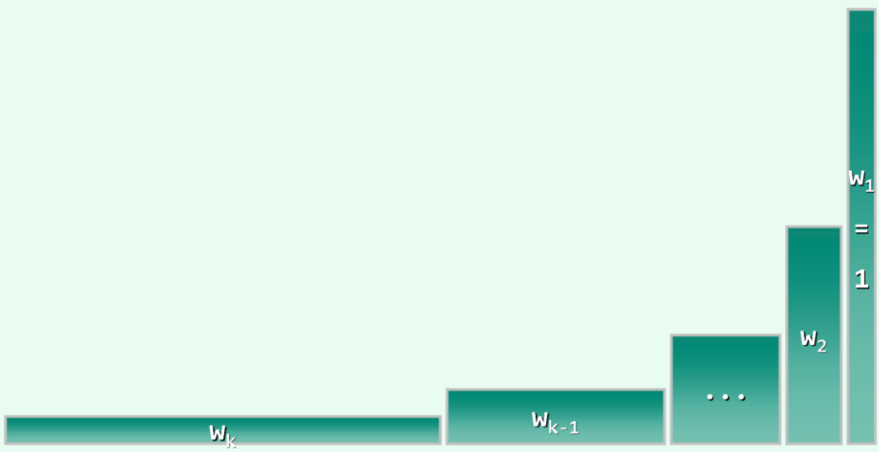

图12.15 递减增量、逐渐逼近策略

每一步迭代中，都从事先设定的某个整数序列中取出一项，并以该项为宽度，将输入向量重排为对应宽度的二维矩阵，然后逐列分别排序。当然，各步迭代并不需要真地从物理上重排原向量。事实上，借助以上一一对应关系，即可便捷地从逻辑上根据其在B[][]中的下标，访问统一保存于A[]中的元素。

不过，为便于对算法的理解，以下我们不妨仍然假想地进行这一重排转换。

因为增量序列中的各项是逆向取出的，所以各步迭代中矩阵的宽度呈缩减的趋势，直至最终使用$w_1 = 1$。矩阵每缩减一次并逐列排序一轮，向量整体的有序性就得以进一步改善。当增量缩减至1时，如图12.15最右侧所示，矩阵退化为单独的一列，故最后一步迭代中的“逐列排序”等效于对整个向量执行一次排序。这种通过不断缩减矩阵宽度而逐渐逼近最终输出的策略，称作递减增量（diminishing increment）算法，这也是希尔排序的另一名称。

以长度为13的向量：

{ 80, 23, 19, 40, 85, 1, 18, 92, 71, 8, 96, 46, 12 }

为例，对应的希尔排序过程及结果如图12.16所示。

秩k	列号	0	1	2	3	4	5	6	7	8	9	10	11	12
元素A[k]		80	23	19	40	85	1	18	92	71	8	96	46	12
分8列逐列排序之后	0	71								80				
	1		8								23			
	2			19								96		
	3				40								46	
	4					12								85
	5						1							
	6							18						
	7								92					
		71	8	19	40	12	1	18	92	80	23	96	46	85
分5列逐列排序之后	0	1					71					96		
	1		8					18					46	
	2			19					85					92
	3				40					80				
	4					12					23			
		1	8	19	40	12	71	18	85	80	23	96	46	92
分3列逐列排序之后	0	1			18			23			40			92
	1		8			12			85			96		
	2			19			46			71			80	
		1	8	19	18	12	46	23	85	71	40	96	80	92
分2列逐列排序之后	0	1		12		19		23		71		92		96
	1		8		18		40		46		80		85	
		1	8	12	18	19	40	23	46	71	80	92	85	96
分1列逐列排序之后		1	8	12	18	19	23	40	46	71	80	85	92	96

图12.16 希尔排序实例：采用增量序列{ 1, 2, 3, 5, 8, 13, 21, ... }

■ 底层算法

最后一轮迭代等效于向量的整体排序，故无论此前各步如何迭代，最终必然输出有序向量，希尔排序的正确性毋庸置疑。然而反过来，我们却不禁有个疑问：既然如此，此前各步迭代中的逐列排序又有何必要？为何不直接做最后一次排序呢？这涉及到底层排序算法的特性。能够有效支持希尔排序的底层排序算法，必须是输入敏感的，比如3.5.2节所介绍的插入排序算法。

尽管该算法在最坏情况下需要运行$\mathcal{O}(n^2)$时间，但随着向量的有序性不断提高（即逆序对的不断减少），运行时间将会锐减。具体地，根据习题[3-11]的结论，当逆序元素的间距均不超过k时，插入排序仅需$\mathcal{O}(kn)$的运行时间。仍以图12.16为例，最后一步迭代（整体排序）之前，向量仅含两对逆序元素（40和23、92和85），其间距为1，故该步迭代仅需线性时间。

正是得益于这一特性，各步迭代对向量有序性的改善效果，方能不断积累下来，后续各步迭代的计算成本也能得以降低，并最终将总体成本控制在足以令人满意的范围。

12.3.2 增量序列

如算法12.2所示，希尔排序算法的主体框架已经固定，唯一可以调整的只是增量序列的设计与选用。事实上这一点也的确十分关键，不同的增量序列对插入排序以上特性的利用程度各异，算法的整体效率也相应地差异极大。以下将介绍几种典型的增量序列。

■ Shell序列

首先考查Shell本人在提出希尔算法之初所使用的序列：

$$\mathcal{H}_{shell} = \{ 1, 2, 4, 8, 16, 32, ..., 2^k, ... \}$$

我们将看到，若使用这一序列，希尔排序算法在最坏情况下的性能并不好。

不妨取$[0, 2^N)$内所有的$n = 2^N$个整数，将其分为$[0, 2^{N-1})$和$[2^{N-1}, 2^N)$两组，再分别打乱次序后组成两个随机子向量，最后将两个子向量逐项交替地归并为一个向量。比如N = 4时，得到的向量可能如下（为便于区分，这里及以下，对两个子向量的元素分别做了提升和下移）：

$$11\ {}_{4}\ 14\ {}_{3}\ 10\ {}_{0}\ 15\ {}_{1}\ 9\ {}_{6}\ 8\ {}_{7}\ 13\ {}_{2}\ 12\ {}_{5}$$

请注意，在$\mathcal{H}_{shell}$中，首项之外的其余各项均为偶数。因此，在最后一步迭代之前，这两组元素的秩依然保持最初的奇偶性不变。如果把它们分别比作井水与河水，则尽管井水与河水各自都在流动，但毕竟“井水不犯河水”。

特别地，在经过倒数第二步迭代（$w_2 = 2$）之后，尽管两组元素已经分别排序，但二者依然恪守各自的秩的奇偶性。仍以N = 4为例，此时向量中各元素应排列如下：

$$8\ {}_{0}\ 9\ {}_{1}\ 10\ {}_{2}\ 11\ {}_{3}\ 12\ {}_{4}\ 13\ {}_{5}\ 14\ {}_{6}\ 15\ {}_{7}$$

准确地，此时元素k的秩为$(2k + 1) \% (2^N + 1)$。对于每一$1 \le k \le 2^{N-1}$，与其在最终有序向量中相距k个单元的元素各有2个，故最后一轮插入排序所做比较操作次数共计：

$$2 \times (1 + 2 + 3 + ... + 2^{N-1}) = 2^{N-1} \cdot (2^{N-1} + 1) = \mathcal{O}(n^2)$$

反观这一实例可见，导致最后一轮排序低效的直接原因在于，此前的各步迭代尽管可以改善两组元素各自内部的有序性，但对二者之间有序性的改善却于事无补。究其根源在于，序列$\mathcal{H}_{shell}$中除首项外各项均被2整除。由此我们可以得到启发——为改进希尔排序的总体性能，首先必须尽可能减少不同增量值之间的公共因子。为此，一种彻底的方法就是保证它们之间两两互素。

不过，为更好地理解和分析如此设计的其它增量序列，需要略做一番准备。

■ 邮资问题

考查如下问题：

> 假设在某个国家，邮局仅发行面值分别为4分和13分的两种邮票，那么
> 1）准备邮寄平信的你，可否用这两种邮票组合出对应的50分邮资？
> 2）准备邮寄明信片的你，可否用这两种邮票组合出对应的35分邮资？

略作思考，即不难给出前一问的解答：使用六张4分面值的邮票，另加两张13分的。但对于后一问题，无论你如何绞尽脑汁，也不可能给出一种恰好的组合方案。

■ 线性组合

用数论的语言，以上问题可描述为：$4m + 13n = 35$是否存在自然数（非负整数）解？

对于任意自然数g和h，只要m和n也是自然数，则$f = mg + nh$都称作g和h的一个组合（combination）。我们将不能由g和h组合生成出来的最大自然数记作$x(g, h)$。

这里需要用到数论的一个基本结论：如果g和h互素，则必有

$$x(g, h) = (g - 1)\cdot(h - 1) - 1 = gh - g - h$$

就以上邮资问题而言，$g = 4$与$h = 13$互素，故有

$$x(4, 13) = 3 \times 12 - 1 = 35$$

也就是说，35恰为无法由4和13组合生成的最大自然数。

■ h-有序与h-排序

在向量$S[0, n)$中，若$S[i] \leq S[i + h]$对任何$0 \leq i < n - h$均成立，则称该向量h-有序（h-ordered）。也就是说，其中相距h个单元的每对元素之间均有序。

考查希尔排序中对应于任一增量h的迭代。如前所述，该步迭代需将原向量“折叠”成宽度为h的矩阵，并对各列分别排序。就效果而言，这等同于在原向量中以h为间隔排序，故这一过程称作h-排序（h-sorting）。不难看出，经h-排序之后的向量必然h-有序。

关于h-有序和h-排序，Knuth[3]给出了一个重要结论（习题[12-12]和[12-13]）：

> 已经g-有序的向量，再经h-排序之后，依然保持g-有序

也就是说，此时该向量既是g-有序的，也是h-有序的，称作(g, h)-有序。

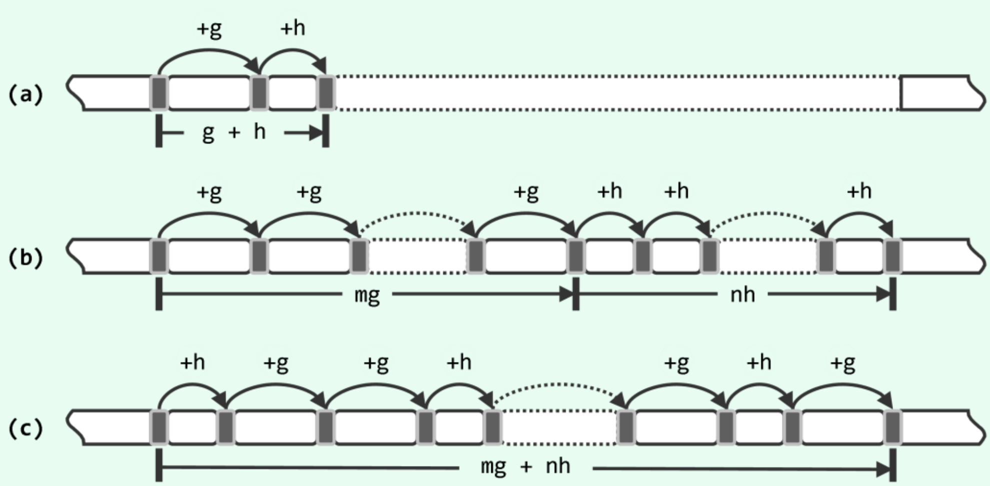

图12.17 (g, h)-有序向量必然(mg + nh)-有序

考查(g, h)-有序的任一向量S。如图12.17(a)所示，借助有序性的传递律可知，相距g + h的任何一对元素都必有序，故S必然(g + h)-有序。推而广之，如图(b)和(c)所示可知，对于任意非负整数m和n，相距mg + nh的任何一对元素都必有序，故S必然(mg + nh)-有序。

有序性的保持与加强

根据以上Knuth所指出的性质，随着h不断递减，h-有序向量整体的有序性必然逐步改善。特别地，最终1-有序的向量，即是全局有序的向量。

为更准确地验证以上判断，可如图12.18所示，考查与任一元素S[i]构成逆序对（习题[3-11]）的后继元素。

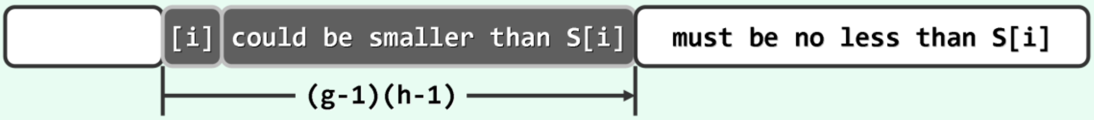

图12.18 经多步迭代，逆序元素可能的范围必然不断缩小

在分别做过g-排序与h-排序之后，根据Knuth的结论可知该向量必已(g, h)-有序。由以上分析，对于g和h的任一线性组合mg + nh，该向量也应(mg + nh)-有序。因此反过来，逆序对的间距必不可能是g和h的组合。而根据此前所引数论中的结论，只要g和h互素，则如图12.18所示，逆序对的间距就绝不可能大于(g - 1)·(h - 1)。

由此可见，希尔排序过程中向量的有序性之所以会不断积累并改善，其原因可解释为，向量中每个元素所能参与构成的逆序对持续减少，整个向量所含逆序对的总数也持续减少。与此同时，随着逆序对的减少，底层所采用的插入排序算法的实际执行时间，也将不断减少，从而提高希尔排序的整体效率。以下结合具体的增量序列，就此做出定量的估计。

(g, h)-有序与排序成本

设某向量S已属(g, h)-有序，且假设g和h的数值均处于$\mathcal{O}$(d)数量级，以下考查对该向量做d-排序所需的时间成本。

据其定义，d-排序需将S等间距地划分为长度各为$\mathcal{O}$(n / d)的d个子向量，并分别排序。由以上分析，在(g, h)-有序的向量中，逆序对的间距不超过

$$(g - 1)\cdot(h - 1)$$

故就任何一个子向量的内部而言，逆序对的间距应不超过

$$(g - 1)\cdot(h - 1) / d = \mathcal{O}(d)$$

再次根据习题[3-11]的结论，采用插入排序算法可在：

$$\mathcal{O}(d)\cdot(n / d) = \mathcal{O}(n)$$

的时间内，完成每一子向量的排序；于是，所有子向量的排序总体消耗的时间应不超过$\mathcal{O}$(dn)。

Papernov-Stasevic序列

现在，可以回到增量序列的优化设计问题。按照此前“尽力避免增量值之间公共因子”的思路，Papernov和Stasevic于1965年提出了另一增量序列：

$$\mathcal{H}_{ps} = \{ 1, 3, 7, 15, 31, 63, ..., 2^k - 1, ... \}$$

不难看出，其中相邻各项的确互素。我们将看到，采用这一增量序列，希尔排序算法的性能可以改进至$\mathcal{O}(n^{3/2})$，其中n为待排序向量的规模。

在序列$\mathcal{H}_{ps}$的各项中，设w_t为与$n^{1/2}$最接近者，亦即$w_t = \Theta(n^{1/2})$。以下将希尔排序算法过程中的所有迭代分为两类，分别估计其运行时间。

首先，考查在w_t之前执行的各步迭代。

这类迭代所对应的增量均满足$w_k > w_t$，或等价地，$k > t$。在每一次这类迭代中，矩阵共有w_k列，各列包含$\mathcal{O}(n/w_k)$个元素。因此，若采用插入排序算法，各列分别耗时$\mathcal{O}((n/w_k)^2)$，所有列共计耗时$\mathcal{O}(n^2/w_k)$。于是，此类迭代各自所需的时间$\mathcal{O}(n^2/w_k)$构成一个大致以2为比例的几何级数，其总和应线性正比于其中最大的一项，亦即不超过

$$\mathcal{O}(2 \cdot n^2/w_t) \quad = \quad \mathcal{O}(n^{3/2})$$

对称地，再来考查w_t之后的各步迭代。

这类迭代所对应的增量均满足$w_k < w_t$，或等价地，$k < t$。考虑到此前刚刚完成w_{k+1}-排序和w_{k+2}-排序，而来自$\mathcal{H}_{ps}$序列的w_{k+1}和w_{k+2}必然互素，且与w_k同处一个数量级。因此根据此前结论，每一次这样的迭代至多需要$\mathcal{O}(n \cdot w_k)$时间。同样地，这类迭代所需的时间$\mathcal{O}(n \cdot w_k)$也构成一个大致以2为比例的几何级数，其总和也应线性正比于其中最大的一项，亦即不超过

$$\mathcal{O}(2 \cdot n \cdot w_t) \quad = \quad \mathcal{O}(n^{3/2})$$

综上可知，采用$\mathcal{H}_{ps}$序列的希尔排序算法，在最坏情况下的运行时间不超过$\mathcal{O}(n^{3/2})$。

■ Pratt序列

Pratt于1971年也提出了自己的增量序列：

$$\mathcal{H}_{pratt} \quad = \quad \{ 1, 2, 3, 4, 6, 8, 9, 12, 16, ..., 2^p 3^q, ... \}$$

可见，其中各项除2和3外均不含其它素因子。

可以证明，采用$\mathcal{H}_{pratt}$序列，希尔排序算法至多运行$\mathcal{O}(n\log^2 n)$时间（习题[12-14]）。

■ Sedgewick序列

尽管Pratt序列的效率较高，但因其中各项的间距太小，会导致迭代趟数过多。为此，Sedgewick[66]综合Papernov-Stasevic序列与Pratt序列的优点，提出了以下增量序列：

$$\mathcal{H}_{sedgewick} \quad = \quad \{ 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, ... \}$$

其中各项，均为：

$$9 \cdot 4^k - 9 \cdot 2^k + 1$$

或

$$4^k - 3 \cdot 2^k + 1$$

的形式。

如此改进之后，希尔排序算法在最坏情况下的时间复杂度为$\mathcal{O}(n^{4/3})$，平均复杂度为$\mathcal{O}(n^{7/6})$。更重要的是，在通常的应用环境中，这一增量序列的综合效率最佳。