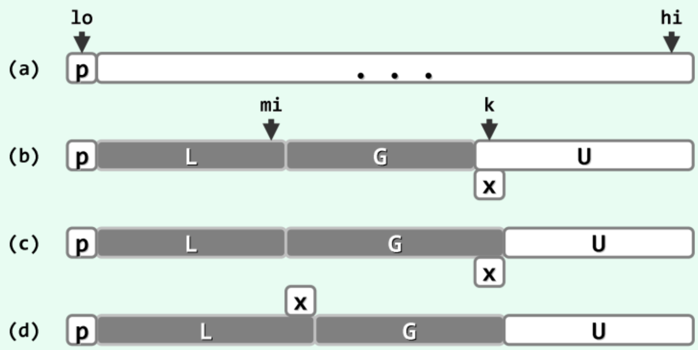
第12章

排序

[12-1] 构造轴点的另一更为快捷的策略，思路如图 x12.1 所示：

> **始终将整个向量 V[lo, hi]划分为四个区间：**
>
> **V[lo]，L = V(lo, mi]，G = V(mi, k)，U = V[k, hi]**
>
> **其中 V[lo]为候选轴点，L/G 中的元素均不大/不小于 V[lo]，U 中元素的大小未知**

初始时取 k - 1 = mi = lo，L 和 G 均为空；此后随着 k 不断递增，逐一检查元素 V[k]，并根据 V[k]相对于候选轴点的大小，相应地扩展区间 L（图(d)）或区间 G（图(c)），同时压缩区间 U。最终当 k - 1 = hi 时，U 不含任何元素，于是只需将候选轴点放至 V[mi]，即成为真正的轴点。



图x12.1 轴点构造算法（版本C）

a） 试依此思路，实现对应的划分算法 Vector::partition()；

【解答】

一种可行的实现方式，如代码x12.1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C
   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
   int mi = lo;
   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   //   [ ---- < [lo] ----- ] [ ----- [lo] <= --- ] [ ----- unknown ----- ]
   // X x . . . . . . . . . x . . . . . . . . . . . x . . . . . . . . . . x
   // |                     |                       |                     |
   // lo (pivot)            mi                      k                     hi
   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   for ( int k = lo + 1; k <= hi; k++ ) //自左向右扫描
      if ( _elem[k] < pivot ) //若当前元素_elem[k]小于pivot，则
         swap ( _elem[++mi], _elem[k] ); //将_elem[k]交换至原mi之后，使L子序列向右扩展
   //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   //   [ --------- < [lo] ---------- ] [ ----------- [lo] <= ----------- ]
```

```
    // X x . . . . . . . . . . . . . . x . . . . . . . . . . . . . . . . . x
    // |                                |                                  |
    // lo                               mi                                 hi
    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    swap ( _elem[lo], _elem[mi] ); //候选轴点归位
    return mi; //返回轴点的秩
}
```

代码x12.1 轴点构造算法（版本C）

b） 基于该算法的快速排序是否稳定？

【解答】

不稳定。按照以上算法划分向量的过程中，子向量L和R都是向右侧“延伸”，新元素都是插至各自的末尾。除此之外，子向量L不会有任何修改，故其中所有元素之间的相对次序，必然与原向量完全一致。然而，在子向量L的每次生长之前，子向量R都需要相应地向前“滚动”一个单元，故可能造成雷同元素之间相对次序的紊乱。

c） 基于该算法的快速排序，能否高效地处理大量元素重复之类的退化情况？

【解答】

在元素大量甚至完全重复的情况下，以上划分算法虽不致出错，但划分所得子向量的规模相差悬殊，快速排序算法几乎退化为起泡排序算法，整体运行时间将增加到$\mathcal{O}(n^2)$。

[12-2] 考查 majEleCandidate()算法（教材 343 页代码 12.6）的返回值 maj。

a） 该候选者尽管不见得必然是众数，但是否一定是原向量中出现最频繁者？为什么？

【解答】

未必。该算法采用减而治之的策略，原向量被等效地切分为若干区段，各区段的首元素分别在其中占至少50%的比例（不妨称作“准众数”）。因此，最终返回的maj，实际上只是最后一个区段的准众数，未必就是整个向量的（准）众数。

b） 该返回值在向量中出现的次数最少可能是多少？试就此举一实例。

【解答】

实际上，无论原向量的长度如何，只要其中的确不包含众数，则最终返回的maj都有可能仅出现一次。作为一个实例，我们考查如下长度为n = 22的向量A[]：

$$\{\ \boxed{0,\ 1};\ \boxed{0,\ 0,\ 1,\ 1};\ \boxed{0,\ 0,\ 0,\ 1,\ 1,\ 1};\ \boxed{0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1};\ \boxed{2,\ 1}\ \}$$

其中，元素0、1和2分别出现了10次、11次和1次。若采用majEleCandidate()算法，整个向量将等效于被分成5个区间：前4个区间A[0, 2)、A[2, 6)、A[6, 12)和A[12, 20)，均以0作为maj候选；最后的A[20, 22)以2作为maj候选，并最终返回2。显然，仅出现一次的元素2在这里既非频繁数，更非（准）众数。

读者可以仿照此例，构造出更长的实例。

[12-3] 按照教材 12.2.2 节的定义，众数应严格地多于其它元素。若将“多于”改为“不少于”，则

a） 该节所设计的算法框架是否依然可以沿用？或者，需如何调整？

【解答】

可以继续沿用。

请注意，在目前的总体框架majority()（教材342页代码12.4）中，最终一步都会调用majEleCheck()（教材342页代码12.5），通过对原向量一趟遍历，针对候选者做严格的甄别。因此，只要majEleCandidate()算法能在此之前筛选出唯一的候选者，就不致误判或漏判。

b） majEleCandidate()算法（教材 343 页代码 12.6）可否继续沿用？或者，需如何调整？

【解答】

如此放宽众数的标准之后，我们需要计算的，实际上就是习题[12-2]之a）所定义的准众数。但若继续沿用目前的majEleCandidate()算法，则有可能造成漏判。

继续考查习题[12-2]之b）所给的实例，可以看到一个有趣的现象：其中的元素1明明是准众数（在所有n = 22个元素中，它恰好出现了n/2 = 11次），但却未被任何一个区间选作maj。这就意味着，majEleCandidate()算法注定无法将该元素作为maj返回，从而造成遗漏。

显然，当向量规模n为奇数时，准众数必然就是众数，因此不妨只考查n为偶数的情况（如上例）。此时针对准众数的查找，对原众数查找算法的一种简明调整方法是：首先任选一个元素（比如末元素），并在$\mathcal{O}$(n)时间内甄别其是否为准众数。不妨设该元素不是准众数，于是只需将其忽略（原向量的有效长度减至奇数n - 1），即可将在原向量中查找准众数的问题，转化为在这个长度为n - 1的向量中查找众数的问题。

当然，调整的方法不一而足，读者不妨从其它角度出发，设计并实现自己的改进方法。

[12-4] 在微软 Office 套件中，Excel 提供了一系列的统计查询函数。

试通过查阅手册，了解 large(range, rank)、median(range)和 mode(range)等函数的功能；这些功能，分别对应于本章所讨论的哪些问题？

【解答】

large(range, rank)：

在range所指示的范围内按数值大小找出第rank大者，等效于k-选取算法。

median(range)：

在range所指示的范围内按数值大小找出居中者，等效于中位数算法。

mode(range)：

在range所指示的范围内找出出现最多的数值，等效于众数算法。

[12-5] 实际上，trivialMedian()算法（教材 343 页代码 12.7）只需迭代$(n_1 + n_2)/2$步即可终止。

a） 照此思路，改进该算法；

【解答】

这里计算的目标，是归并之后向量中的中位数，然而这并不意味着一定要显式地完成归并。实际上就此计算任务而言，只需设置一个计数器，而不必真地引入并维护一个向量结构。

具体地，依然可以沿用原算法的主体流程，向量S只是假想式地存在。于是，我们无需真地将子向量中的元素注意转移至S中，而是只需动态地记录这一假想向量的规模：每当有一个元素假想式地归入其中，则计数器相应地递增。一旦计数器抵达$\lfloor (n_1 + n_2)/2 \rfloor$，即可忽略后续元素并立即返回假想向量的末元素——亦即，两个子向量当前元素之间的更小者。

请读者根据以上分析与提示，独立完成该算法的改进任务。

b） 如此改进之后，算法总体的渐进时间复杂度是否有所降低？

【解答】

没有实质的降低。

改进后的算法仍需迭代$\lfloor (n_1 + n_2)/2 \rfloor$步，总体的渐进时间复杂度依然是$\mathcal{O}(n_1 + n_2)$。

[12-6] 如教材 344 页代码 12.8 所示的 median()算法属于尾递归形式，试将其改写为迭代形式。

【解答】

请读者按照消除尾递归的一般性方法，独立完成编码和调试任务。

[12-7] 如教材 346 页代码 12.9 所示的 median()算法，针对两个向量长度相差悬殊的情况做了优化处理。

a） 试分析该方法的原理，并证明其正确性；

【解答】

该算法首先比较n_1和n_2的大小，并在必要时交换两个向量，从而保证有$n_1 \leq n_2$。

以下，若两个向量的长度相差悬殊，则可对称地适当截除长者（S_2）的两翼，以保证有：

$$n_1 \leq n_2 \leq 2 \cdot n_1$$

因为S_2两翼截除的长度相等，所以此后$S_1 \cup S_2$的中位数，依然是原先$S_1 \cup S_2$的中位数。

b） 试证明，复杂度的精确上界应为$\mathcal{O}(\log(\min(n_1, n_2)))$。

【解答】

由以上分析可见，无论是交换两个向量，还是截短S_2，都只需常数时间。因此实质的计算，只是针对长度均同阶于$\min(n_1, n_2)$的一对向量计算中位数。

与教材中对这一减而治之策略的分析同理，此后每做一次比较，即可将问题的规模缩减大致一半。因此，问题的规模将以1/2为比例按几何级数的速度递减，直至平凡的递归基。整个算法的递归深度不超过$\log_2(\min(n_1, n_2))$，总体时间复杂度为$\mathcal{O}(\log(\min(n_1, n_2)))$。

[12-8] 若输入的有序序列 S_1 和 S_2 以列表（而非向量）的方式实现，则：

a） 如教材 344 页代码 12.8 和 346 页代码 12.9 所示的两个 median()算法，分别应做哪些调整？

【解答】

这里的关键在于，列表仅支持“循位置访问”的方式，不能像“循秩访问”那样在常数时间内访问任一元素。特别地，在读取每个元素之前，都要沿着列表进行计数查找。

b） 调整之后的计算效率如何？

【解答】

为保证$|S_1| \leq |S_2|$而交换两个序列（的名称），依然只需$O(1)$时间；然而，序列S_2两翼的截短则大致需要$O(n_2 - n_1)$时间。而更重要的是，在此后的递归过程中，每一次为将问题规模缩减一半，都必须花费线性的时间。

因此，总体需要$O(n_1 + n_2)$时间——这一效率，已经降低到与蛮力算法trivialMedian（教材343页代码12.7）相同。

[12-9] 若输入的有序序列 S_1 和 S_2 以平衡二叉搜索树（而非序列）的方式给出，则：

a） 如教材 344 页代码 12.8 和 346 页代码 12.9 所示的两个 median()算法，分别应做哪些调整？

【解答】

为此，需要给平衡二叉搜索树增加以下接口：

```
template <typename T> BinNodePosi(T) & BBST<T>::search(Rank r); //查找并返回树中第r大的节点
template <typename T> BinNodePosi(T) & BBST<T>::removeMin(int k); //从树中删除最小的k个节点
template <typename T> BinNodePosi(T) & BBST<T>::removeMax(int k); //从树中删除最大的k个节点
```

b） 调整之后的计算效率如何？

【解答】

仿照quickSelect()算法（教材348页代码12.10），不难实现一个效率为$O(\log n)$的search(r)接口。然而，高效的removeMin(k)和removeMax(k)接口并不容易实现。

实际上，一种简明的策略是：首先通过中序遍历，将平衡二叉搜索树中的所有元素转化为有序向量，然后套用以上算法计算中位数。

当然，按照这一策略，运行时间主要消耗于遍历，整体为$O(n_1 + n_2)$——与教材343页代码12.7中的蛮力算法trivialMedian()相同。

[12-10] a） 基于教材 346 页代码 12.9 中的 median()算法，添加整型输入参数 k，实现在 $S_1 \cup S_2$ 中选取第 k 个元素的功能；

【解答】

记 $n_1 = |S_1|$ 和 $n_2 = |S_2|$，不失一般性，设 $n_1 \le n_2$。

进一步地，不妨设 $2k \le n_1 + n_2$——否则，可以颠倒比较器的方向，原问题即转化为在 $S_1 \cup S_2$ 中选取第 $n_1 + n_2 - k$ 个元素，与以下方法同理。

若 $k \le n_1 = \min(n_1, n_2)$，则只需令：

$S_1' = S_1[0, k)$

$S_2' = S_2[0, k)$

于是原问题即转换为计算 $S_1' \cup S_2'$ 的中位数。

否则，若 $n_1 < k < n_2$，则可令

$S_1' = S_1[0, n_1)$

$S_2' = S_2[0, 2k - n_1)$

于是原问题即转换为计算 $S_1 \cup S_2'$ 的中位数。

可见，无论如何，针对 $S_1 \cup S_2$ 的k-选取问题总是可以在常数时间内，转换为中位数问题，并进而直接调用相应的算法。

b） 新算法的时间复杂度是多少？

【解答】

由上可见，无论如何，都可在 $\mathcal{O}(1)$ 时间内将原问题转换为中位数的计算问题。借助median()算法，如此只需要 $\mathcal{O}(\log(\min(n_1, n_2)))$ 时间。

[12-11] 考查如教材 348 页代码 12.10 所示的 quickSelect()算法。

a） 试举例说明，最坏情况下该算法的外循环需要执行 $\Omega(n)$ 次；

【解答】

在最坏情况下，每一次随机选取的候选轴点pivot = A[lo]都不是查找的目标，而且偏巧就是当前的最小者或最大者。于是，对向量的每一次划分都将极不均匀，其中的左侧或右侧子向量长度为0。如此，每个元素都会被当做轴点的候选，并执行一趟划分，累计 $\Omega(n)$ 次。

从算法策略的角度来看，原拟定的“分而治之”策略未能落实，实际效果反而等同于采用了“减而治之”策略。

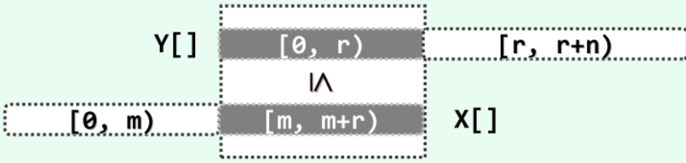
b） 在各元素独立等概率分布的条件下，该算法的平均时间复杂度是多少？

【解答】

仿照教材12.1.5节对quickSort()算法的分析方法，同样可以证明，quickSelect()算法的平均运行时间为 $\mathcal{O}(n)$——在平均意义上，与算法12.1（教材348页）相当。

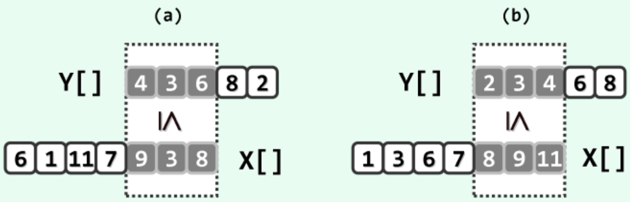
[12-12] 如图x12.2所示，设有向量X[0, m + r)和Y[0, r + n)，且满足：

对于任何0 ≤ j < r，都有Y[j] ≤ X[m + j]



图x12.2 在向量X和Y各自排序后，对齐元素之间的次序依然保持

试证明，在X和Y分别（按非降次序）排序并转换为X'和Y'之后（如图x12.3的实例所示），对于任何0 ≤ j < r依然有Y'[j] ≤ X'[m + j]成立。（提示：习题[2-41]的推广）



图x12.3 (a)排序前有Y[0, 3) ≤ X[4, 7)，(b)排序后仍有Y'[0, 3) ≤ X'[4, 7)

【解答】

对于任意的0 ≤ j < r，考查元素X'[m + j]。

一方面，在有序的X'（以及无序的X）中，显然应该恰有m + j个元素不大于X'[m + j]。

而另一方面，由图x12.2可见，其中至少存在j个元素，各自不小于无序的Y（以及有序的Y'）中的某一元素，而且Y（Y'）中的这些元素互不重复。也就是说，Y'（Y）中至少存在j个元素不大于X'[m + j]，故必有：

Y'[j] ≤ X'[m + j]

当然，仿照习题[2-41]的两种证明方法，亦可得出同样结论。

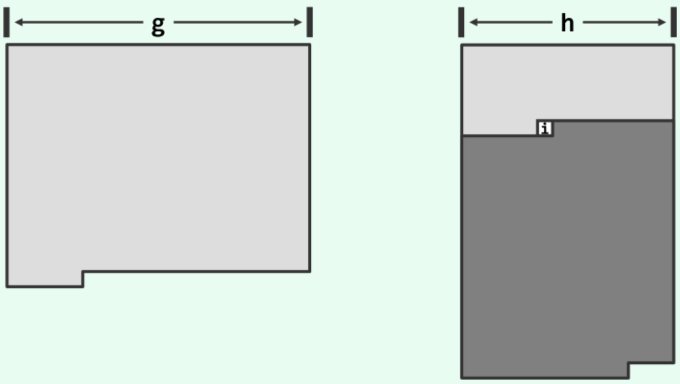
有兴趣的读者，不妨照此提示，从其它角度独立给出证明。

[12-13] 试证明，g-有序的向量再经h-排序之后，依然保持g-有序。

【解答】

在已经h-排序之后的向量中，考查任一元素A[i]，我们欲证总有A[i] ≤ A[i + g]。

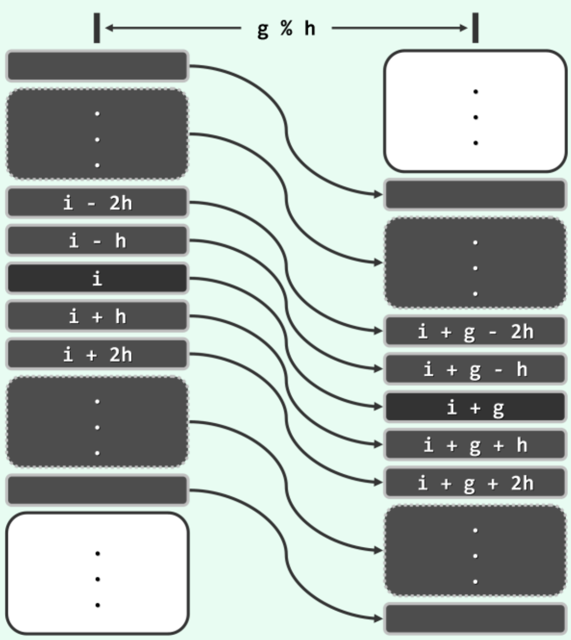
如图x12.4所示，考查g-排序以及h-排序（在逻辑上）各自对应的二维矩阵。于是，在后一矩阵中，A[i + g]必然落在深色阴影区域内部。我们继续在该矩阵中，考查A[i]以及A[i + g]各自所属的列。



图x12.4 g-有序的向量A[ ]再经h-排序后，A[i + g]必然来自阴影区域

根据g-有序性，如图x12.5所示，两个列的前缀与后缀必然一一对应地有序，亦即：

```
              ...
A[i - 2h]  ≤  A[i + g - 2h]
A[i -  h]  ≤  A[i + g -  h]
A[i      ]  ≤  A[i + g      ]
A[i +  h]  ≤  A[i + g +  h]
A[i + 2h]  ≤  A[i + g + 2h]
              ...
```



图x12.5 g-有序的向量A[ ]按照h列重排之后，A[i]所属列的前缀，必然与A[i + g]所属列的后缀，逐个元素地对应有序

于是根据本章第[12-12]题的结论，在经过h-排序之后，这两列的前缀和后缀之间的对应有序关系依然成立，g-有序性得以延续。

[12-14] 设使用 Pratt 序列：

$$\mathcal{H}_{pratt} = \{ 1, 2, 3, 4, 6, 8, 9, 12, 16, ..., 2^p3^q, ... \}$$

对长度为 n 的任一向量 S 做希尔排序。

试证明：

a）若 S 已是(2, 3)-有序，则只需 $\mathcal{O}(n)$时间即可使之完全有序；

【解答】

根据教材第12.3.2节的分析结论，在(2, 3)-有序的序列中，逆序元素之间的间距不超过：

$$(2 - 1) \times (3 - 1) - 1 = 1$$

也就是说，整个向量中包含的逆序对不过$\mathcal{O}(n)$个。

于是根据习题[3-11]的结论，此后对该向量的1-排序仅需$\mathcal{O}(n)$时间。

b）对任何 $h_k \in \mathcal{H}_{pratt}$，若 S 已是($2h_k$, $3h_k$)-有序，则只需 $\mathcal{O}(n)$时间即可使之 h_k-有序；

【解答】

既然所有元素的秩取值于[0, n)范围内，故若照相对于h_k的模余值，它们可以划分为h_k个同余类；相应地，原整个向量可以“拆分为”h_k个接近等长的子向量。

不难看出，其中每个子向量都是(2, 3)-有序的，根据上一问的结论，均可在线性时间内转换为各自1-有序的；就其总体效果而言，等同于在$\mathcal{O}(n)$时间内转换为全局的h_k-有序。

c）针对$\mathcal{H}_{pratt}$序列中的前 $\mathcal{O}(\log^2 n)$项，希尔排序算法需要分别迭代一轮；

【解答】

$\mathcal{H}_{pratt}$序列中的各项无非是2^p和3^q的乘积组合，因此其中不大于n项数至多不超过：

$$\log_2 n \times \log_3 n = \mathcal{O}(\log^2 n)$$

d）总体的时间复杂度为 $\mathcal{O}(n\log^2 n)$。

【解答】

综合b）和c）的结论，在采用$\mathcal{H}_{pratt}$序列的希尔排序过程中，每一轮耗时不超过$\mathcal{O}(n)$，累计至多迭代$\mathcal{O}(\log^2 n)$轮，因此，总体耗时不超过$\mathcal{O}(n\log^2 n)$。