

第2章

向量

[2-1] 关于某个算法，甲证明“其平均时间复杂度为 $\Theta(n)$ ”，乙证明“其分摊时间复杂度为 $\Theta(n)$ ”。若他们的结论均正确无误，则是甲的结论蕴含乙的结论，乙的结论蕴含甲的结论，还是互不蕴含？

【解答】

两个结论之间不存在蕴含关系，但相对而言，后一结论更为可靠和可信。

所谓平均复杂度，是指在假定各种输入实例的出现符合某种概率分布之后，进而估算出的加权复杂度均值。比如在教材的第12.1.5节中，将基于“待排序的元素服从独立均匀随机分布”这一假设，估算出快速排序算法在各种情况下的加权平均复杂度。

所谓分摊复杂度，则是纵观连续的足够多次操作，并将其间总体所需的运行时间分摊至各次操作。与平均复杂度的本质不同在于，这里强调，操作序列必须是的确能够真实发生的，其中各次操作之间应存在前后连贯的时序关系。比如在参考文献[41]中，Tarjan采用势能分析法对伸展树所有可能的插入、删除操作序列进行分析，并估算出在此意义下单一操作的分摊执行时间。

由此可见，前者不必考查加权平均的各种情况出现的次序，甚至其针对概率分布所做的假设未必符合真实情况；后者不再割裂同一算法或数据结构的各次操作之间的因果关系，更加关注其整体的性能。综合而言，基于后一尺度得出的分析结论，应该更符合于真实情况，也更为可信。

以教材2.4节中基于容量加倍策略的可扩充向量为例。若采用平均分析，则很可能因为所做的概率分布假定与实际不符，而导致不准确的结论。比如若采用通常的均匀分布假设，认为扩容与不扩容事件的概率各半，则会得出该策略效率极低的错误结论。实际上，只要假定这两类事件出现的概率各为常数，就必然导致这种误判。而实际情况是，采用加倍扩容策略后，在其生命期内随着该数据结构的容量不断增加，扩容事件出现的概率将以几何级数的速度迅速趋近于零。对于此类算法和数据结构，唯有借助分摊分析，方能对其性能做出综合的客观评价。

[2-2] 教材32页代码2.2的copyFrom()算法中，目标数组_elem[]是通过new操作由系统另行分配的，故可保证在物理上与来源数组A[]相互独立。若不能保证这种独立性，该算法需要做哪些调整？

【解答】

若不能保证目标数组与来源数组之间的独立性，则二者可能在空间上有所重叠，出现所谓的“搭接”现象。此时，需要区分两种情况分别处理。若目标数组的某一后缀与来源数组的某一前缀重叠搭接，则需要按“从前到后”的次序逐一转移各元素；反之，若目标数组的某一前缀与来源数组的某一后缀重叠搭接，则应改用“从后到前”的次序。

[2-3] 假设将教材34页代码2.4中expand()算法的扩容策略改为“每次追加固定数目的单元”。

a) 试证明，在最坏情况下，单次操作中消耗于扩容的分摊时间为 $\Theta(n)$ ，其中n为向量规模；

【解答】

假定每次追加d个单元。于是，只要每隔固定的常数k次操作就发生一次扩容，则初始容量为 n_0 的向量在经过连续的 $N \gg k$ 次操作之后，容量将增加至：

$$n = n_0 + d \cdot (N/k)$$

在此过程中，消耗于扩容操作的时间合计为：

$$\begin{aligned} T(N) &= (n_0 + d) + (n_0 + 2d) + (n_0 + 3d) + \dots + (n_0 + d \cdot (N/k)) \\ &= n_0 \cdot (N/k) + d \cdot (N/k)(N/k - 1) \end{aligned}$$

均摊至单次操作，所需时间为：

$$\begin{aligned} T(N)/N &= n_0/k + (d/k)(N/k - 1) \\ &= [n_0 + d \cdot (N/k)]/k - d/k \\ &= n/k - d/k \\ &= \Theta(n) \end{aligned}$$

b) 试举例说明，这种最坏情况的确可能发生。

【解答】

考查初始为空 ($n_0 = 0$) 的向量，假定持续地对其执行插入操作。于是，每经过 d 次操作，都需要花费线性时间进行扩容。于是就分摊意义而言，单次操作需花费 $\Theta(n)$ 时间用于扩容。

[2-4] 试证明，教材 36 页代码 2.5 中 shrink() 算法具有分摊的常数时间复杂度。

【解答】

与教材 2.4.4 节对 expand() 算法的分析方法完全一致。

[2-5] 设某算法中设有一个无符号 32 位整型变量 $count = b_{31}b_{30}\dots b_1b_0$ ，其功能是作为计数器，不断地递增 ($count++$ ，溢出后循环)。每经一次递增， $count$ 的某些比特位都会在 0 和 1 之间翻转。

比如，若当前有： $count = 43_{(10)} = 0\dots0101\underline{011}_{(2)}$

则下次递增之后将有： $count = 44_{(10)} = 0\dots0101\underline{100}_{(2)}$

在此过程中，共有（最末尾的）三个比特发生翻转。

现在，考查对 c 连续的足够多次递增操作。纵观这一系列的操作，试证明：

a) 每经过 2^k 次递增， b_k 恰好翻转一次；

【解答】

每经过 2^k 次递增，计数器的数值恰好增加 2^k 。体现在其二进制展开中，对应于数位 b_k 的（由 0 到 1 或由 1 到 0）翻转。

b) 对于每次递增操作，就分摊的意义而言， $count$ 只有 $O(1)$ 个比特位发生翻转。

【解答】

不妨从 0 开始，考查该计数器的连续 $N \gg 2$ 次递增操作。我们将在此期间的所有数位翻转，分别“记账”至对应的数位。于是根据以上 a) 所证的性质，所有数位的翻转次数总和为：

$$\begin{aligned} & N/2^{31} + N/2^{30} + \dots + N/4 + N/2 + N \\ &= N \cdot (1/2^{31} + 1/2^{30} + \dots + 1/4 + 1/2 + 1) \\ &< 2N \end{aligned}$$

因此就分摊意义而言，单次递增操作仅引发 $O(1)$ 次数位翻转。

与基于加倍策略的可扩充向量同理，这里的关键在于参与累计的各项构成（以 $1/2$ 为倍数的）几何级数，正如我们已知的，就渐进意义而言其总和同阶于其中的最大项。因此无论这里的计数器有多少个二进制数位组成，上述性质与结论均可成立。实际上，即便假设计数器拥有无穷个数位（故永不溢出），亦是如此。

[2-6] 考查教材 37 页代码 2.7 中的 `permute()` 算法，假设 `rand()` 为理想的随机数发生器，试证明：

a) 通过反复调用 `permute()` 算法，可以生成向量 $V[0, n]$ 的所有 $n!$ 种排列；

【解答】

通过对向量规模 n 的数学归纳予以证明。假定该命题对于规模不足 n 的任意向量均成立。作为归纳的基础，规模 $n = 1$ 的情况不证自明。以下考查规模为 n 的任意向量 $V[]$ ：

$V[0], V[1], V[2], \dots, V[n - 1]$

任取该向量的一个排列：

$V[a_0], V[a_1], V[a_2], \dots, V[a_{n-1}]$

只需证明，该排列有可能被 `permute()` 算法生成。

实际上，在该算法的第一次迭代中，有可能取 $\text{rand}() \% n = a_{n-1}$ 。于是，首次参与交换的将是 $V[n - 1]$ 和 $V[\text{rand}() \% i] = V[a_{n-1}]$ ，且如此交换之后的向量成为：

$V[0], V[1], \dots, V[a_{n-1} - 1], V[n - 1], V[a_{n-1} + 1], \dots, V[n - 2], V[a_{n-1}]$

不难看出，自此之后的计算过程完全等效于，对于其中前 $n - 1$ 个元素组成的子向量置乱。也就是说，可等效于将向量：

$V[0], V[1], \dots, V[a_{n-1} - 1], V[n - 1], V[a_{n-1} + 1], \dots, V[n - 2]$

置乱为：

$V[a_0], V[a_1], V[a_2], \dots, V[a_{n-2}]$

由归纳假设，`permute()` 算法可以生成长度为 $n - 1$ 的该排列（以及长度为 n 的整个排列）。

b) 由该算法生成的排列中，各元素处于任一位置的概率均为 $1/n$ ；

【解答】

可以按照各元素在 `permute()` 算法中（自后向前）就位的次序，归纳证明这一命题。

首先，鉴于 `rand()` 的随机均匀性，最早就位的元素 $V[a_{n-1}]$ 必以相等的概率选自整个向量，故原向量中每个元素最终出现在该位置的概率为 $1/n$ 。

不妨假定该命题对前 k 个 ($0 \leq k < n$) 就位的元素均成立，即它们均是以 $1/n$ 的等概率取自原向量中各元素。以下，考查下一个就位的元素 $X = V[a_{n-k-1}]$ 。



图x2.1 `permute()` 算法中第 $k + 1$ 个就位元素，应等概率地随机选自当时的前 $n - k$ 个元素

如图x2.1所示，按照算法流程，元素 X 应随机地选自当时的前 $n - k$ 个元素（包含其自身），且其中各元素被选中的概率均为 $1/(n - k)$ 。

请注意，当时的这前 $n - k$ 个元素均有可能参与过此前的 k 次随机交换。这些元素都是截至

当前尚未就位者，原向量中的任何一个元素，都有 $(n - k)/n$ 的概率成为它们中的一员。因此，原向量中每个元素接下来被随机选中且随即交换成为 $V[a_{n-1}]$ 的概率应是：

$$(n - k)/n \times 1/(n - k) = 1/n$$

c) 该算法生成各排列的概率均为 $1/n!$ 。

【解答】

既然以上已经证明，原向量中各元素最终就位于各位置的概率均等，`permute()`算法就应以相等的概率，随机地生成所有可能的排列。

对于规模为 n 的向量，可能的排列共计 $n!$ 种，故概率分别为 $1/n!$ 。

[2-7] 在C语言标准库中，Brian W. Kernighan 和 Dennis M. Ritchie 设计的随机数发生器如下：

```
unsigned long int next = 1;

/* rand:  return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

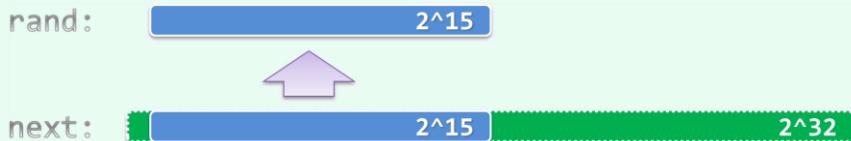
/* srand:  set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

a) 阅读这段代码，并理解其原理；

【解答】

该算法维护一个32位的无符号长整数`next`，随着`next`的“随意”变化，不断输出伪随机数。通过`srand(seed)`，可以设置`next`的初始值（随机种子）。

此后`rand()`的每一次执行过程，均如图x2.2所示。



图x2.2 Brian W. Kernighan和Dennis M. Ritchie所设计随机数发生器的原理

首先，在`next`当前值的基础上乘以 $1103515245 = 3^5 \times 5 \times 7 \times 129749$ ，并加上 12345 。然后，通过整除运算在该长整数的二进制展开中截取高16位，进而通过模余运算抹除最高比特位。经如此的“混沌化”处理之后，即可作为“随机数”返回。

**b) 试说明，若采用 `rand()` 的这个版本实现 `permute()` 算法，则上题的结论 a) 和 b) 并不能兑现；
(提示：绝大多数的排列实际上根本无法由该算法生成)**

【解答】

不难注意到，以上 `rand()` 算法的返回值尽管具有一定的随机性，但远非理想的随机。实际上更严格地讲，其返回值是确定的：只要知道当前的 `next` 值，即可确定地得出下一 `next` 值。

反观如教材 37 页代码 2.7 所示的 `permute` 算法，其对每一个向量的置乱结果，应完全取决于其间对 `rand()` 函数 $n = V.size()$ 次调用所返回的 n 个“随机数”。但使用如上实现的 `rand()`，这些返回值完全取决于所设定的起始种子 `seed`。

`permute()` 算法如需兑现上题中结论 a) 和 b) 所述的性质，本应保证能够（通过 `rand()`）获得 n 个彼此独立的随机数。然而不幸的是，由以上分析可见这一条件并不成立。

实际上我们甚至可以确定，如此可能获得的长度为 n 的“随机数”序列有多少个——其总数不超过 `seed` 的取值范围，就此例而言即为：

$$2^{16} = 65,536 < 9! = 362,880$$

这就意味着，即便是长度 $n = 9$ 的向量，借助该版本的 `rand()` 也无法枚举出所有可能的置乱排列；而对于更长的向量，这个算法就更是无能为力了。

c) 试说明，采用此类伪随机数发生器实现 `permute()` 算法，上题的结论 a) 和 b) 必然无法兑现；

【解答】

由以上分析可知，只要继续沿用这种“种子 + 迭代”的模式，增加 `rand()` 输出整数的位宽亦是徒然——这种“改进”并不能有效地克服上述缺陷。比如，不难验证有：

$$2^{64} = 2^{60+4} < 20 \times 10^{18} < 21! = 51,090,942,171,709,440,000$$

也就是说，即便使用 64 位的无符号整数，在向量的规模超过 20 之后，借助这种模式的随机数发生器就无法覆盖所有可能的置乱排列。进一步地，随着向量规模 n 的进一步扩大，如此可枚举出来的排列，在所有 $n!$ 种排列中所占的比例将迅速下降，并很快趋近于零。

d) 针对 b) 和 c) 所指出的不足，应如何改进 `rand()` 和 `permute()` 算法？

【解答】

以上所介绍基于“种子 + 迭代”模式的随机数发生器，在 `permute()` 之类的算法中之所以显得力不从心，关键在于它们无法保证所生成随机数之间的独立性（`independence`）。反过来，这也给我们指出了改进的大方向。

当然，要在独立性与高效性之间达到足够令人满意的平衡绝非易事，有待于我们的持续探索。

a) 在最好情况下，该算法需要运行多少时间？为什么？

【解答】

若首次接受比对的元素（即向量区间的末元素 `_elem[hi - 1]`）恰好就是目标元素，则算法可以随即因查找成功而终止。这自然地属于最好情况——此种情况下，累计仅需常数时间。

b) 若仅考查成功的查找，则平均需要运行多少时间？为什么？

【解答】

这种顺序式的查找算法，可能成功地终止于向量区间内的任意位置。此外，各元素对应的查找长度，自后向前构成一个递增的等差数列：

$$1, 2, 3, \dots, n = hi - lo$$

于是若按照默认惯例，假定所有元素作为目标元素的概率均等，则其查找长度的均值（数学期望）应渐进地与其中的最高项同阶，为 $\mathcal{O}(n)$ 。详细（但嫌复杂）的分析，亦是殊途同归：

$$(1 + 2 + 3 + \dots + n)/n = (n + 1)/2 = \mathcal{O}(n)$$

[2-9] 考查教材 40 页代码 2.11 中的无序向量插入算法 insert(r, e)。

试证明，若插入位置 r 等概率分布，则该算法的平均时间复杂度为 $\mathcal{O}(n)$ ，n 为向量的规模。

【解答】

这里，运行时间主要来自于后继元素顺次后移的操作。因此对于每个插入位置而言，对应的移动操作次数恰好等于其后继元素（包含自身）的数目。不难看出它们也构成一个等差数列，故在等概率的假设条件下，其均值（数学期望）应渐进地与其中的最高项同阶，为 $\mathcal{O}(n)$ 。

详细（但嫌复杂）的分析，亦是殊途同归。

[2-10] 考查教材 41 页代码 2.12 中的无序向量删除算法 remove(lo, hi)。

a) 若以自后向前的次序逐个前移后继元素，可能出现什么问题？

【解答】

位置靠前的元素，可能被位置靠后（优先移动）的元素覆盖，从而造成数据的丢失。

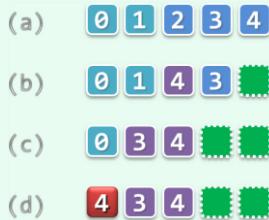
b) 何时出现这类问题？试举一例。（提示：后继元素多于待删除元素时，部分单元会相互覆盖）

【解答】

如图x2.3(a)所示，考查规模为5的向量：

$$V[0, 5] = \{0, 1, 2, 3, 4\}$$

假设我们试图通过调用`V.remove(0, 2)`以删除其中的前两个元素。若采用题中所述不当的次序，则数据元素的移动过程应如该图(b~d)所示。可见，原数据元素`V[2] = 2`并未顺利转移至输出向量中的`V[0]`，即出现数据的丢失现象。



图x2.3 无序向量删除算法remove(lo, hi)中，采用自后向前的次序移动可能造成数据丢失

不难验证，只要按照教材的建议颠倒移动的次序，即可便捷地避免这类错误。

[2-11] `Vector::deduplicate()` 算法的如下实现是否正确？为什么？

```

1 template <typename T> int Vector<T>::deduplicate() { //删除无序向量中重复元素（错误版）
2     int oldSize = _size; //记录原规模
3     for (Rank i = 1; i < _size; i++) { //逐一考查 elem[i]
4         Rank j = find(_elem[i], 0, i); //在 elem[i] 的前驱中寻找与之雷同者（至多一个）
5         if (0 <= j) remove(j); //若存在，则删除之（但在此种情况，下一迭代不必做 i++）
6     }
7     return oldSize - _size; //向量规模变化量，即被删除元素总数
8 }
```

【解答】

按照这一“算法”，若果真发现雷同的前驱 `elem[j]`，虽然的确可以剔除之，但由此产生的副作用是，秩为 `i` 的单元所对应的将不再是原先的 `elem[i]`（而是其直接后继 `elem[i + 1]`）。

这里即便假设，如教材代码 2.14 所实现的同名算法那样，仍能保证接受检查的每个 `elem[i]` 至多只有一个雷同的前驱，但因如上所述的副作用，原 `elem[i + 1]` 也将被跳过，从而导致遗漏与之雷同的元素。实际上，该“算法”每发现一次雷同，都有可能如此遗漏另一个雷同。

[2-12] 考查教材 42 页代码 2.14 中的无序向量唯一化算法 `deduplicate()`。

a) 试证明，即便在最好情况下，该算法也需要运行 $\Omega(n^2)$ 时间；

【解答】

该算法由 $\mathcal{O}(n)$ 次迭代构成，每次迭代都需要做一次查找操作，以及可能的一次删除操作。对于元素 `elem[k]`，若需做删除操作，则为此需花费 $\mathcal{O}(n - k)$ 时间移动所有的后继元素；反之，若不需要做删除操作，则意味着此前的查找操作以失败告终，其间已经花费了 $\mathcal{O}(k)$ 时间。总而言之，无论如何，各轮迭代至少需要 $\Omega(\min(n - k, k))$ 时间，累计 $\Omega(n^2)$ 。

b) 试参照教材 46 页代码 2.19 中有序向量唯一化算法 `uniqueify()` 的技巧，改进该算法，并分析其时间复杂度；

【解答】

比如，在发现重复元素后不必立即剔除，而是借助位图 `Bitmap` 结构（习题 [2-34]）将其标注为“待删除”。待所有雷同元素均已筛选完毕，再经一趟遍历在 $\mathcal{O}(n)$ 时间内统一删除。

经如此改进之后，尽管渐进的时间复杂度依然为 $\mathcal{O}(n^2)$ ，但因为时间消耗主要来源于静态的查找操作，故实际的运行速度仍将大幅提高。

c) 试继续改进该算法，使其时间复杂度降至 $\mathcal{O}(n \log n)$ ；

【解答】

最为直接的方法，就是先调用 `sort()` 接口对向量做整体排序，然后调用 `uniqueify()` 接口做唯一化处理。只要实现得当（比如采用教材所介绍的优化算法），这两步均只需 $\mathcal{O}(n \log n)$ 时间。

当然，由此将产生的一个副作用——保留下来的元素，未必延续此前的相对排列次序。实际上，完全可以在不增加渐进时间复杂度的前提下，消除这一副作用。请读者独立完成这一任务。

d) 这一效率是否还有改进的余地？为什么？

【解答】

如果没有其它的附加条件，那么在图灵机等通常的计算模型下，可以证明“无序向量唯一化”问题的复杂度下界（难度）为 $\Omega(n \log n)$ 。为此，我们可以借助归约的技巧。

一般地，考查难度待界定的问题B。若另一问题A满足以下性质：

- 1) 问题A的任一输入，都可以在线性时间内转换为问题B的输入
- 2) 问题B的任一输出，都可以在线性时间内转换为问题A的输出

则称“问题A可在线性时间内归约为问题B”，或简称作“问题A可线性归约为问题B”，或者称“从问题A到问题B，存在一个线性（时间）归约（linear-time reduction）关系”，记作：

$$A \leq_N B$$

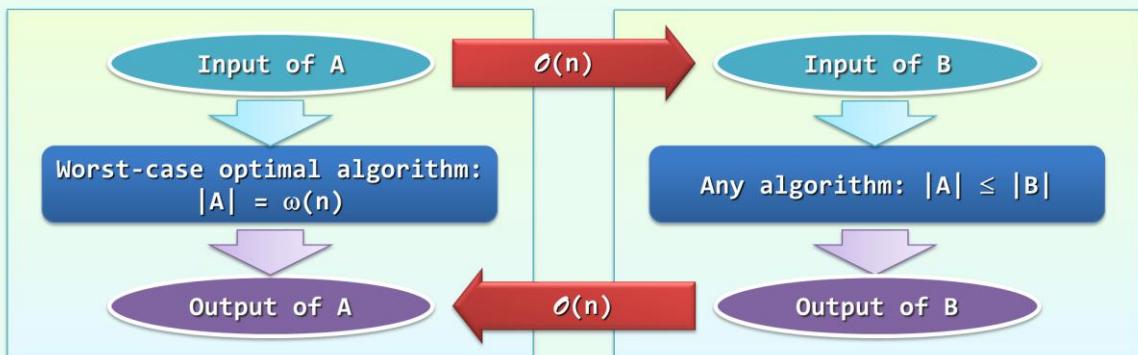
此时，若问题A的难度（记作 $|A|$ ）已界定为严格地高于 $\Omega(n)$ ，亦即：

$$|A| = \Omega(f(n)) = \omega(n)^{\textcircled{1}}$$

则问题B的难度（记作 $|B|$ ）也不会低于这个复杂度下界，亦即：

$$|B| \geq |A| = \Omega(f(n))$$

实际上，若问题A果真可以线性归约为问题B，则由后者的任一算法，必然同时也可以导出前者的一个算法。这一结论，可由图x2.4直接看出：为求解问题A，可将其输入转化为问题B的输入，再调用后者的算法，最后将输出转化为前者的输出。



图x2.4 从问题A到问题B的线性归约

因此，假若问题B具有一个更低的下界，则至少存在一个 $\omega(f(n))^{\textcircled{2}}$ 的算法，于是由上可知，问题A也存在一个 $\omega(f(n))$ 的算法——这与问题A已知的 $\omega(n)$ 下界相悖。

^① 这里的“小 ω 记号”（small-omega notation），也是界定复杂度的尺度。

准确地，若存在函数 $f(n)$ ，使得对于任何正的常数 c ，当 n 足够大之后都有 $T(n) > c \cdot f(n)$ ，则可认为 $f(n)$ 给出了 $T(n)$ 增长速度的一个严格非紧的渐进下界，记作 $T(n) = \omega(f(n))$ 。请注意小 ω 记号与大 Ω 记号的微妙区别。

比如， $2n(n+1) = \Omega(n \log n)$ 而且 $2n(n+1) = \omega(n \log n)$ ， $2n \log n = \Omega(n \log n)$ 但 $2n \log n \neq \omega(n \log n)$

^② 这里的“小 ω 记号”（small- ω notation），也是界定复杂度的尺度。

准确地，若存在函数 $f(n)$ ，使得对于任何正的常数 c ，当 n 足够大之后都有 $T(n) < c \cdot f(n)$ ，则可认为 $f(n)$ 给出了 $T(n)$ 增长速度的一个严格非紧的渐进上界，记作 $T(n) = \omega(f(n))$ 。请注意小 ω 记号与大 ω 记号的微妙区别。

比如， $2n = \omega(n \log n)$ 而且 $2n = \omega(n \log n)$ ， $2n \log n = \omega(n \log n)$ 但 $2n \log n \neq \omega(n \log n)$

归纳起来，为运用线性归约界定问题B的难度下界，须经以下步骤：

- 1) 找到难度已知为 $\omega(n)$ 的问题A
- 2) 证明问题A可线性归约为问题B——其输入、输出可在线性时间内完成转换

就本题而言，“无序向量唯一化”即是难度待界定的问题B，将其简记作UNIQ。作为参照，考查所谓的元素唯一性（Element Uniqueness，简称EU）问题A：

对于任意n个实数，判定其中是否有重复者

作为EU问题的输入，任意n个实数都可在线性时间内组织为一个无序向量，从而转换为UNIQ问题的输入；另一方面，一旦得到UNIQ问题的输出（即去重之后的向量），只需花费线性时间，核对向量的规模是否依然为n，即可判定原实数中是否含有重复者（亦即，得到EU问题的输出）。因此，EU问题可以线性归约为UNIQ问题，亦即：

$$A \leq_N B$$

实际上，算法复杂度理论业已证明，EU问题具有 $\Omega(n \log n)$ 的复杂度下界，故这也是UNIQ问题的一个下界。反过来，以上所给的 $O(n \log n)$ 算法，已属最优。

[2-13] 试参照函数对象 Increase (教材 44 页代码 2.16) 重载运算符 “()” 的方式，基于无序向量的遍历接口 traverse()，实现以下操作（假定向量元素类型支持算术运算）：

a) **decrease():所有元素数值减一；**

【解答】

一种可行的实现方式，如代码x2.1所示。

```
1 template <typename T> struct Decrease //函数对象：递减一个T类对象
2   { virtual void operator() ( T& e ) { e--; } }; //假设T可直接递减或已重载--
3
4 template <typename T> void decrease ( Vector<T> & V ) //统一递减向量中的各元素
5 { V.traverse ( Decrease<T>() ); } //以Decrease<T>()为基本操作进行遍历
```

代码x2.1 基于遍历实现向量的decrease()功能

与教材代码2.16中increase()接口的实现方式同理，这里首先定义一个名为Decrease的函数对象，然后以此为基本操作，通过遍历接口对向量的所有元素逐一处理。

b) **double():所有元素数值加倍。**

【解答】

一种可行的实现方式，如代码x2.2所示。

```
1 template <typename T> struct Double //函数对象：倍增一个T类对象
2   { virtual void operator() ( T& e ) { e *= 2; } }; //假设T可直接倍增
3
4 template <typename T> void double ( Vector<T> & V ) //统一加倍向量中的各元素
```

```
5 { V.traverse ( Double<T>() ); } //以Double<T>()为基本操作进行遍历
```

代码x2.2 基于遍历实现向量的double()功能

与以上Decrease的实现方式相仿，这里的键依然在于定义一个名为Double的函数对象。

[2-14] 字符串、复数、矢量等类型没有提供自然的比较规则，但仍能人为地对其强制定义某种大小关系(即次序关系)。试分别为这三种类型的对象定义“人工的”次序。

【解答】

比如，可以采用字典序来确定字符串之间的次序。

复数与复平面上的点一一对应，故可以按照“先实部、后虚部”的原则定义复数之间的次序；复数也与极坐标平面上的点一一对应，故也可以按照“先幅值、后极角”的原则定义其次序。

二维矢量与复数彼此对应，故定义于复数之间任何一种次序，也应适用于矢量。反之亦然。读者可参照以上方式，结合具体应用的特点及需求，定义不同的次序。

[2-15] 考查采用CBA式算法对4个整数的排序。

a) 试证明，最坏情况下不可能少于5次比较；

【解答】

既然是考查最坏情况，不妨假定所有整数互异，此时的CBA式算法经每次比较之后，在对应的比较树（comparison tree）中只有两个有效的分支。

此时共有 $4!$ 种可能的输出，故算法对应的比较树至少拥有24个叶节点，因此树高至少是：

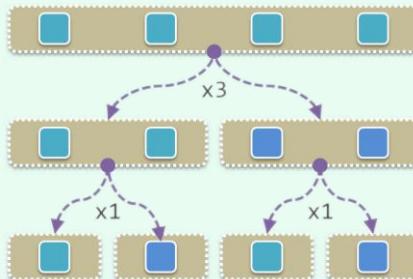
$$\lceil \log_2 24 \rceil = 5$$

b) 试设计这样的一个CBA式算法，即便在最坏情况下，至多只需5次比较。

【解答】

比如，可以采用教材61页2.8.3节介绍的归并排序（mergesort）算法。

如图x2.5所示，当输入规模为4时，归并排序算法的递归深度为2。



图x2.5 输入规模为4时的归并排序过程

底层的两次二路归并，各自仅需1次比较；顶层的一次二路归并，最坏情况下只需3次比较。总体合计，不过5次比较。

[2-16] `search(e, lo, hi)` 算法版本 C (教材 56 页代码 2.24) 所返回的秩，均符合接口规范。试针对以下情况，分别验证这一结论：

- a) [lo, hi] 中的元素均小于 e；
- b) [lo, hi] 中的元素均等于 e；
- c) [lo, hi] 中的元素均大于 e；
- d) [lo, hi] 中既包含小于 e 的元素，也包含大于 e 的元素，但不含等于 e 的元素。

【解答】

请读者对照该算法对应的代码，独立完成分析和验证任务。

[2-17] 考虑用向量存放一组字符串。

为在其中进行二分查找，可依据字典序确定字符串之间的次序：

设字符串 $a = a_1a_2\dots a_n$ 和 $b = b_1b_2\dots b_m$

则 $a < b$ 当且仅当 $n = 0 < m$ ，或者 $a_1 < b_1$ ，或者 $a_1 = b_1$ 且 $a_2\dots a_n < b_2\dots b_m$

也就是说，两个字符串之间的大小关系，取决于它们（按首字符对齐后）第一对互异的字符。

- a) 试实现一个字符串类，并提供相应的比较器，或者重载对应的操作符；

【解答】

请读者仿照教材代码 2.9，独立完成编码与调试任务。

- b) 若共有 n 个字符串，二分查找的复杂度是否仍为 $\mathcal{O}(\log n)$ ？

【解答】

总体而言，此时的二分查找，未必可以保证在 $\mathcal{O}(\log n)$ 时间内完成。

尽管二分查找算法的流程、迭代次数均保持不变，但与整数、字符之类的基本数据类型不同，字符串的长度并不确定，且无上限。因此，“每次比较仅需 $\mathcal{O}(1)$ 时间”的假设条件不再成立，换而言之，此时字符串之间的比较已经不能继续视作基本操作。

[2-18] 设采用实现如教材 48 页代码 2.21 所示的二分查找 `binSearch()` 算法版本 A，针对独立均匀分布于 $[0, 2n]$ 内的整数目标，在固定的有序向量 { 1, 3, 5, …, 2n - 1 } 中查找。

- a) 若将平均的成功和失败查找长度分别记作 S 和 F，试证明： $(S + 1) \cdot n = F \cdot (n + 1)$ ；

【解答】

对向量规模 n 做数学归纳。

假定对于规模小于 n 的所有向量，以上命题均成立。以下考查规模为 n 的向量。

实际上，我们可以考查 `binSearch()` 算法对应的比较树。一般地，若向量的规模为 n ，则对应的比较树应由 n 个内部节点（成功的返回）以及 $n + 1$ 个叶节点（失败的返回）。

特别地，规模为 $n - 1$ 和 n 的向量所对应的比较树 (CT_{n-1} 和 CT_n) 应该分别如图 x2.6(a) 和 (b) 所示。二者之间的差异仅在于，前者的某一外部节点 x ，被替换为由一个内部节点 x 和两个外部节点 a 与 b 组成的子树。也就是说，原先的某一查找失败情况，现在对应于一种成功情况，另加两种失败情况。综合而言，成功情况及失败情况各自增加一种。

比如，对于向量 { 1, 3 }，共计有 2 种成功情况 { 1, 3 } 以及 3 种失败情况 { 0, 2, 4 }；

而对于向量{ 1, 3, 5 }，则共计有3种成功情况{ 1, 3, 5 }以及4种失败情况{ 0, 2, 4, 6 }。



图x2.6 二分查找binSearch()算法版本A所对应的比较树，在向量规模递增后的结构变化

设在 CT_{n-1} 中，失败情况 x 所对应的查找长度为 d 。于是根据算法流程，在 CT_n 中成功情况 x 对应的查找长度应为 $d + 2$ ，而新的两种失败情况对应的查找长度为 $d + 1$ 和 $d + 2$ 。

若在 CT_{n-1} 中，内部节点、外部节点所对应的成功查找总长度、失败查找总长度应分别为：

$$S \cdot (n - 1)$$

$$F \cdot n$$

则在 CT_n 中，内部节点、外部节点所对应的成功查找总长度、失败查找总长度应分别为：

$$S \cdot (n - 1) + (d + 2)$$

$$F \cdot n + (d + 1) + (d + 2) - d = F \cdot n + (d + 3)$$

于是在 CT_n 中，成功查找、失败查找的平均长度应分别为：

$$S' = [S \cdot (n - 1) + (d + 2)]/n$$

$$F' = [F \cdot n + (d + 3)]/(n + 1)$$

故有：

$$(S' + 1) \cdot n = (S + 1) \cdot (n - 1) + (d + 3)$$

$$F' \cdot (n + 1) = F \cdot n + (d + 3)$$

根据归纳假设，应有：

$$(S + 1) \cdot (n - 1) = F \cdot n$$

故有：

$$(S' + 1) \cdot n = F' \cdot (n + 1)$$

至于以上证明的归纳基础，该命题的平凡情况不难验证。我们将此留给读者完成。

b) 上述结论，是否适用于 binSearch() 算法的其它版本？为什么？

【解答】

仍然适用。

证明方法完全类似，只不过在从 CT_{n-1} 转换至 CT_n 时，内部节点、叶节点所对应的成功、失败查找长度的计算口径不同。

c) 上述结论，是否适用于 `fibSearch()` 算法的各个版本？为什么？

【解答】

依然适用。

考查在从 CT_{n-1} 转换至 CT_n 后，（二者有所差异的）局部子树对成功、失败查找总长度的贡献。实际上在命题中恒等式的两端，只要这两方面的贡献相互抵消，恒等式即可继续成立。

不难验证，`fibSearch()` 依然具有这种特性。我们也将此留给读者完成。

d) 若待查找的整数按照其它的随机规律分布，以上结论又应如何调整？

【解答】

命题中的恒等式需加入各种情况对应的概率权重。具体的调整形式，留给读者完成。

实际上，原命题中的恒等式，也可视作一种特殊情况——在等概率分布下，所有权重均等。

[2-19] 为做 Fibonacci 查找，未必非要严格地将向量整理为 `fib(n) - 1` 形式的长度。

比如，可考虑以下策略：

- a) 按照黄金分割比，取 $mi = \lfloor 0.382 * lo + 0.618 * hi \rfloor$
- b) 按照近似的黄金分割比，取 $mi = \lfloor (lo + 2 * hi) / 3 \rfloor$
- c) 按照近似的黄金分割比，取 $mi = (lo + (lo \ll 1) + hi + (hi \ll 2)) \gg 3$

这几种替代策略，综合性能各有什么优劣？为什么？

【解答】

在如代码2.22（教材53页）所示的 `fibSearch()` 算法中，首先需要调用 `Fib` 类的初始化接口，找到一个尽可能小，却亦足够覆盖整个向量 $V[0, n]$ 的 `Fibonacci` 数，作为初始查找范围的宽度：

$$N \geq hi - lo$$

如代码x1.12所示，`Fib` 类对象的初始化只需 $\mathcal{O}(\log \Phi(n))$ 时间（分摊至后续的查找过程，每次递归仅增加 $\mathcal{O}(1)$ 时间）。接下来在迭代式逐层深入地查找过程中，还需通过一个内循环确定合适的黄金分割点——实际上每个分割点只需不超过两次迭代。

尽管以上足以说明 `fibSearch()` 算法的高效性，但就算法流程的简洁性而言，却远不如标准的二分查找 `binSearch()` 算法。

究其原因在于，目前实现的版本对 `Fibonacci` 查找思想的理解和贯彻过于机械。实际上，本题所建议的几种方式都能在保持渐进效率的前提下适当地灵活变通，使算法的流程得以简化和清晰。建议的三种改进方案中，方案a)采用近似值快速地估算出切分点，方案b)可更好地发挥整数运算的优势，而方案c)则通过移位操作替代更为耗时的乘、除法运算。

48

[2-20] 试分别针对二分查找算法版本 A (代码 2.21) 及 Fibonacci 算法 (代码 2.22)，推导其失败查找长度的显式公式，并就此方面的性能对二者做一对比。

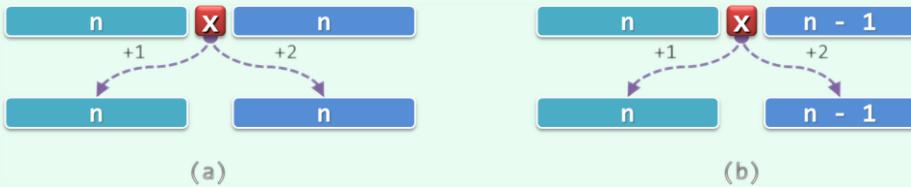
【解答】

可以证明：对于规模为 n 的有序向量，二分查找在失败情况下的平均比较次数不超过：

$$1.5 \cdot \log_2(n + 1) = \mathcal{O}(1.5 \cdot \log n)$$

为此，我们采用数学归纳法。作为归纳基，这一命题对长度为 1 的向量显然。

以下考查二分查找的第一步迭代，如图x2.7所示无非两种情况。



图x2.7 二分查找失败情况的递归分类

首先，考查左、右子向量规模均为 n 的情况。此时如图(a)所示，左侧子向量总共包含 $n + 1$ 种失败情况，由归纳假设，其平均比较长度不超过：

$$1 + 1.5 \cdot \log_2(n + 1)$$

右侧子向量也总共包含 $n + 1$ 种失败情况，由归纳假设，其平均比较长度不超过：

$$2 + 1.5 \cdot \log_2(n + 1)$$

综合所有失败情况，总体的平均查找长度不超过：

$$\begin{aligned} & [(n + 1) \cdot (1 + 1.5 \cdot \log_2(n + 1)) + (n + 1) \cdot (2 + 1.5 \cdot \log_2(n + 1))] / (2n + 2) \\ & = 1.5 \cdot \log_2(2n + 2) = O(1.5 \cdot \log(2n + 1)) \end{aligned}$$

再考查左、右子向量规模分别为 n 和 $n - 1$ 的情况。此时如图(b)所示，左侧子向量总共包含 $n + 1$ 种失败情况，由归纳假设，其平均比较长度不超过：

$$1 + 1.5 \cdot \log_2(n + 1)$$

右侧子向量总共包含 n 种失败情况，由归纳假设，其平均比较长度不超过：

$$2 + 1.5 \cdot \log^2 n$$

综合所有失败情况，总体的平均查找长度不超过：

$$\begin{aligned} & [(n + 1) \cdot (1 + 1.5 \cdot \log_2(n + 1)) + n \cdot (2 + 1.5 \cdot \log_2 n)] / (2n + 1) \\ & = [(3n + 1) + 1.5 \cdot ((n + 1) \cdot \log_2(n + 1) + n \cdot \log_2 n)] / (2n + 1) \\ & \sim [(3n + 1) + 1.5 \cdot 2 \cdot (n + 1/2) \cdot \log_2(n + 1/2)] / (2n + 1) \\ & = (3n + 1)/(2n + 1) + 1.5 \cdot \log_2(n + 1/2) \\ & \sim 1.5 \cdot [1 + \log_2(n + 1/2)] \\ & = 1.5 \cdot \log_2(2n + 1) = O(1.5 \cdot \log(2n)) \end{aligned}$$

类似地还可以证明：Fibonacci查找在失败情况下的平均比较次数不超过：

$$\lambda \cdot \log_2(n + 1) = O(\lambda \cdot \log n)$$

其中

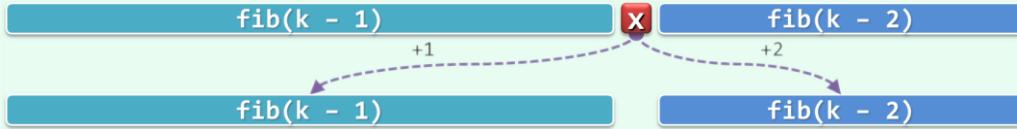
$$\lambda = 1 + 1/\Phi^2 = (2 + \Phi)/(1 + \Phi) = 3 - \Phi = 1.382$$

$$\Phi = (\sqrt{5} + 1) / 2 = 1.618$$

我们依然采用数学归纳法。作为归纳基，这一命题对长度为1的向量显然。

^③ 本书约定，使用符号“~”表示渐进同阶

以下如图x2.8所示，设向量长度为 $n = \text{fib}(k) - 1$ ，考查Fibonacci查找的第一步迭代。



图x2.8 Fibonacci查找失败情况的递归分类

此时，左侧子向量共计 $\text{fib}(k - 1)$ 种失败情况，由归纳假设，其平均比较长度不超过：

$$1 + \lambda \cdot \log_2 \text{fib}(k - 1)$$

右侧子向量共计 $\text{fib}(k - 2)$ 种失败情况，其平均比较长度不超过：

$$2 + \lambda \cdot \log_2 \text{fib}(k - 2)$$

综合所有失败情况，平均查找长度不超过：

$$\begin{aligned} & [\text{fib}(k-1) \cdot (1 + \lambda \cdot \log_2 \text{fib}(k-1)) + \text{fib}(k-2) \cdot (2 + \lambda \cdot \log_2 \text{fib}(k-2))] / \text{fib}(k) \\ &= [(\text{fib}(k) + \text{fib}(k-2)) + \lambda \cdot (\text{fib}(k-1) \cdot \log_2 \text{fib}(k-1) \\ &\quad + \text{fib}(k-2) \cdot \log_2 \text{fib}(k-2))] / \text{fib}(k) \\ &\sim [\lambda \cdot \text{fib}(k) + \lambda \cdot \text{fib}(k) \cdot (\log_2 \text{fib}(k) - 1)] / \text{fib}(k) \\ &= \lambda \cdot \log_2 \text{fib}(k) \end{aligned}$$

由上可见，就失败情况而言，尽管两种查找算法的渐进时间复杂度均为 $\mathcal{O}(\log n)$ ，但常系数却又一定的差异——Fibonacci查找的 $\lambda = 1.382$ ，较之二分查找的1.5更小。

[2-21] 设 $A[0, n)$ 为一个非降的正整数向量。

试设计并实现算法 `expSearch(int x)`，对于任意给定的正整数 $x \leq A[n - 1]$ ，从该向量中找出一个元素 $A[k]$ ，使得 $A[k] \leq x \leq A[\min(n - 1, k^2)]$ 。

若有两个满足这一条件的 k ，只需返回其中任何一个，但查找时间不得超过 $\mathcal{O}(\log(\log k))$ 。

(提示：指数查找 (exponential search))

【解答】

我们令 k 从 1 开始不断递增 ($A[k]$ 亦相应地非降变化)，直至 $A[k]$ 首次超过查找目标 x 。

当然，这里不能采用顺序的逐一递增 ($k = k + 1$) 模式：

$$k = 1, 2, 3, 4, 5, 6\dots$$

显然，在抵达 $A[k] \leq x \leq A[k + 1]$ 之前，必然已经花费了 $\mathcal{O}(k)$ 时间。可见，为尽快抵达目标位置，必须加大各试探位置的间距。

然而类似地，采用一般的递增模式仍不足够，比如加倍的递增 ($k = 2 * k$) 模式：

$$k = 1, 2, 4, 8, 16, 32, \dots$$

因为在抵达 $A[k] \leq x \leq A[2k]$ 之前，如此必然已经花费了 $\mathcal{O}(\log k)$ 时间。

为进一步加大各次试探位置的间距，可以采用指数递增 ($k = k * k$) 模式：

$$k = 1, 2, 4, 16, 256, 65536, \dots$$

如此，在抵达 $A[k] \leq x \leq A[k^2]$ 之前，仅需试探的步数为：

$$\mathcal{O}(\log(\log k)) = \mathcal{O}(\log \log k)$$

[2-22] 设 $A[\theta, n][\theta, n]$ 为整数矩阵（即二维向量）， $A[\theta][\theta] = \theta$ 且任何一行（列）都严格递增。

- a) 试设计一个算法，对于任一整数 $x \geq \theta$ ，在 $O(r + s + \log n)$ 时间内，从该矩阵中找出并报告所有值为 x 的元素（的位置），其中 $A[\theta][r]$ ($A[s][\theta]$) 为第 θ 行（列）中不大于 x 的最大者；
 (提示：马鞍查找 (saddleback search))

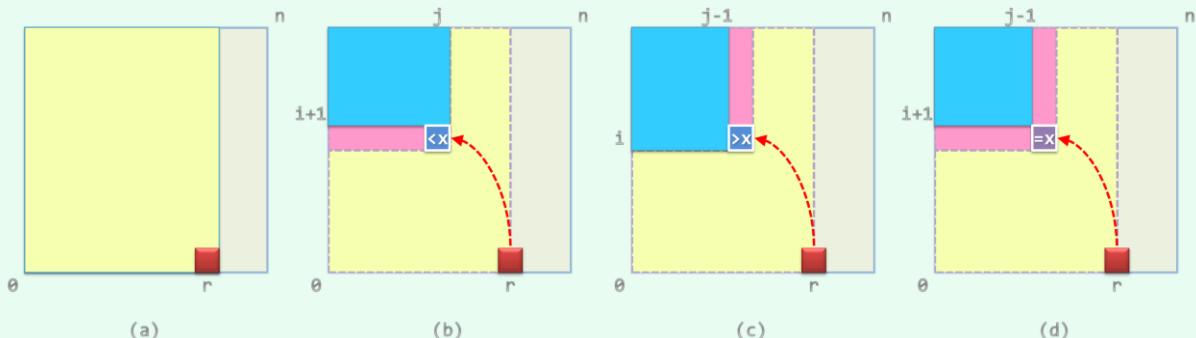
【解答】

一种可行的算法，过程大致如算法x2.1所示。

```
1 saddleback( int A[n][n], int x ) {
2     int i = 0; //不变性：有效查找范围始终为左上角的子矩形A[i, n][0, j]
3     int j = binSearch(A[0][], x); //借助二分查找，在O(log n)时间内，从A的第θ行中找到不大于x的最大者
4     while ( i < n && -1 < j ) { //以下，反复根据A[i][j]与x的比较结果，不断收缩查找范围A[i, n][0, j)
5         if ( A[i][j] < x ) i++; //矩形区域的底边上移
6         else if ( x < A[i][j] ) j--; //矩形区域的右边左移
7         else { report( A[i][j] ); i++; j--; } //报告当前命中元素，矩形区域的底边上移、右边左移
8     }
9 }
```

算法x2.1 马鞍查找

该算法的原理及过程，如图x2.9所示。若将待查找矩阵A视作二维矩形区域，则在算法的每一次迭代中，搜索的范围始终可精简为该矩形左上角的某一子矩形（以阴影示意）。当然，该子矩形在初始情况下即为矩阵对应的整个区域。



图x2.9 马鞍查找算法的原理及过程

算法首先通过二分查找，花费 $O(\log n)$ 的时间在首行 $A[\theta][\cdot]$ 中确定起始元素 $A[\theta][j = r]$ 。于是如图(a)所示，根据该矩阵的单调性，查找范围即可收缩至 $A[i = \theta, n][0, j = r]$ 。

接下来，反复地根据子矩形右下角元素 $A[i][j]$ 与目标元素 x 的大小关系，不断收缩子矩形。既然该矩阵在两个维度均具有单调性，故若 $A[i][j] < x$ ，则如图(b)所示，意味着当前子矩形的底边可以向上收缩一行；若 $x < A[i][j]$ ，则如图(c)所示，意味着当前子矩形的右边可以向左收缩一列；而倘若 $A[i][j] = x$ ，则如图(d)所示，不仅意味着找到了一个新的命中元素，而且当前子矩形的底边和右边可以同时收缩。

总而言之无论如何，每经过一次迭代，搜索的范围都可有效地收缩。由此可见，该算法也采用了减而治之 (decrease-and-conquer) 的策略。

为估计这一过程所需进行的迭代次数，我们不妨考查观察量： $k = j - i$ 。

开始迭代之前，设二分查找返回值为 $j = r < n$ ，则当时应有：

$$k = j - i = r - \theta < n$$

此后每经过一次迭代，或者 j 减一，或者 i 加一，或者二者同时如此变化。总而言之，无论如何观察量 k 都至少减一。另一方面根据循环条件，最后一次迭代中必然有 $i = s$ 和 $\theta \leq j$ ，即：

$$k = j - i \geq \theta - s > -n$$

由此可见，这个过程中的迭代次数不超过：

$$r + s < 2n$$

请注意，单调性在此扮演了重要的角色。实际上，如果将矩阵理解为某一地区，将其中各单元的数值视作对应位置的高度，则该地区的地形将类似于马鞍的形状，该算法也因此得名。

从这一视角来看，所有命中的元素应该就是输入指定高度 x 所对应的一条等高线。于是，上述查找过程，则等效于从某一端的起点出发，逐点遍历该等高线。因为元素数值的严格单调性，该等高线与每一行、每一列至多相交于一个单元。这些单元也就是算法需要遍历并检查的单元，若等高线的起点和终点分别为 $A[\theta][r]$ 和 $A[s][\theta]$ ，则其包含的单元应不超过 $r + s$ 个。这一结论，与前面的分析殊途同归。

b) 若 A 的各行 (列) 只是非减 (而不是严格递增)，你的算法需做何调整？复杂度有何变化？

【解答】

如果带查询矩阵的严格单调性不能保证，则在 $A[i][j] = x$ 的情况下不能继续有效地收缩查找范围。实际上在此种条件下，命中的元素可能多达 $\Omega(n^2)$ 个，故仅在上述算法的基础上做修补，已很难保证整体的效率。

为此，不妨改用其它策略，比如采用第8章将要介绍的kd-树结构或四叉树等数据结构及相应的算法。

[2-23] 教材 2.6 节针对有序向量介绍的各种查找算法，落实减而治之策略的形式均大同小异：反复地“猜测”某一元素 $s[m]$ ，并通过将目标元素与之比较的结果，确定查找范围收缩的方向。然而在某些特殊的场合，沿前、后两个方向深入的代价并不对称，甚至其中之一只允许常数次。

比如，在仅能使用直尺的情况下，可通过反复实验，用鸡蛋刚能摔碎的下落高度（比如精确到毫米）来度量蛋壳的硬度。尽管可以假定在破裂之前蛋壳的硬度保持不变，但毕竟破裂是不可逆的。故若仅有一枚鸡蛋，则我们不得不从 0 开始，以 1 毫米为单位逐步增加下落的高度。若蛋壳的硬度不超过 n 毫米，则需要进行 $\Theta(n)$ 次实验。就效率而言，这等价于退化到无序向量的顺序查找。

a) 若你拥有两枚鸡蛋（假定它们硬度完全相同），所需实验可减少到多少次？试给出对应的算法；

【解答】

不妨以 \sqrt{n} 为间距，将区间 $[1, n]$ 均匀地划分为 \sqrt{n} 个区间。于是，借助第一枚鸡蛋，即可在 $\Theta(\sqrt{n})$ 时间内确定其硬度值所属的区间。接下来，再利用第二枚鸡蛋，花费 $\Theta(\sqrt{n})$ 时间在此区间之内精确地确定其硬度值。

两步合计，共需花费 $\Theta(2 \cdot \sqrt{n}) = \Theta(\sqrt{n})$ 时间。

b) 进一步地，如果你拥有三枚鸡蛋呢？

【解答】

仿照上述思路，以 $n^{1/3}$ 为间距，将区间 $[1, n]$ 均匀地划分为 $n^{1/3}$ 个区间（宽度各为 $n^{2/3}$ ）；然后，再将每个区间继续均匀地细分为 $n^{1/3}$ 个子区间（宽度各为 $n^{1/3}$ ）。

类似地，借助第一枚鸡蛋，在 $\mathcal{O}(n^{1/3})$ 时间内将查找范围缩减至 $n^{2/3}$ ；接下来，再利用第二枚鸡蛋，在 $\mathcal{O}(n^{1/3})$ 时间内将查找范围缩减至 $n^{1/3}$ ；最后，利用第三枚鸡蛋，花费 $\mathcal{O}(n^{1/3})$ 时间在此区间之内精确地确定硬度值。

综合以上三步，总共耗时不过：

$$\mathcal{O}(3 \cdot n^{1/3}) = \mathcal{O}(n^{1/3})$$

c) 一般地，如果共有 d 枚鸡蛋可用呢？

【解答】

将以上方法推广，也就是对区间 $[1, n]$ 逐层细分。每深入一层，都将当前层的每个子区间均匀地细分为 $n^{1/d}$ 个更小的子区间。累计共分为 d 层。

查找也是逐层进行，不断深入：每花费 $\mathcal{O}(n^{1/d})$ 时间，查找范围的宽度都收缩至此前的 $n^{-1/d}$ 。

纵观整个查找过程，共计 d 次复杂度为 $\mathcal{O}(n^{1/d})$ 顺序查找，累计耗时不过：

$$\mathcal{O}(d \cdot n^{1/d}) = \mathcal{O}(n^{1/d})$$

需特别留意的是，为实现子区间的分层细分，只需根据输入参数 d 确定规则，并不需要进行实质性的计算，因此这部分时间无需考虑。

[2-24] 在实际应用中，有序向量内的元素不仅单调排列，而且往往还服从某种概率分布。若能利用这一性质，则可以更快地完成查询。

以查阅英文字典为例，单词"Data"应大致位于前 $1/5$ 和 $1/4$ 之间，而"Structure"则应大致位于后 $1/5$ 和 $1/4$ 之间。对元素的分布规律掌握得越准确，这种加速效果也就越加可观。

此类方法的原理大同小异，无非是利用向量元素的分布规律，根据目标数值，通过插值估计出其大致所对应的秩，从而迅速缩小搜索范围，故称作插值查找（interpolation search）。

a) 若有序向量中的元素均独立且等概率地取自某一数值区间，试证明它们应大致按线性规律分布；

【解答】

既然是均匀且独立的分布，样本数量（向量区间内元素的数量）自然应线性正比于取值范围（向量区间两端点的数值之差）。

b) 针对此类有序向量，如何通过插值来估计待查找元素的秩？试给出具体的计算公式；

【解答】

若查找区间为 $[lo, hi]$ ，且查找目标为 Y ($A[lo] \leq Y < A[hi]$)，则 Y 的秩可以近似地估计为：

$$mi = lo + (hi - lo) * (Y - A[lo]) / (A[hi] - A[lo])$$

c) 试证明:对于此类向量,每经一次插值和比较,待搜索区间的宽度大致以平方根的速度递减^[25].

【解答】

设查找的目标为Y。

该算法通过不断迭代逐步逼近最终位置，故不妨考查其中第j步迭代 s_j , $j = 1, 2, \dots$ 。

若此步迭代对应于子区间 $V_j = [L_j, H_j]$, 区间宽度 $N_j = H_j - L_j$, 则按照上述估计公式, 接受比较的元素的秩为:

$$K_j = L_j + N_j \cdot P_j$$

其中，

$$P_i = (Y - A[L_i]) / (A[H_i] - A[L_i])$$

这里的 P_j , 既可看作 Y 在区间 V_j 内的相对位置, 同时也是均匀分布于 V_j 之内的每一随机变量取值不大于 Y 的概率。

我们将 V_j 中不大于 Y 的元素数目记作 I_j 。既然该区间内的 N_j 个元素相互独立，故 I_j 应该就是它们取值不大于 Y 的概率总和。更准确地， I_j 可视作一个符合二项式分布的随机变量，于是 I_j 的期望值为 $N_j \cdot P_j$ ，方差为 $N_j \cdot P_j \cdot (1 - P_j)$ 。

再来考查查找目标Y，将其在整个区间中的秩记作 K^* ——亦即，总共恰有 K^* 个元素不大于Y。

于是，在查找范围业已收缩至 V_j 时， $K^* - L_j$ 就是上述符合二项式分布的随机变量。因此若如上将第 j 个接受比较的元素的秩记作 K_j ，则按照该算法的原理， K_j 即是在经过此前各步迭代而进入状态 S_j 的情况下，取 K^* 的条件期望，亦即：

以下考查前后相邻的两次试探位置的间距，令：

$$D_i = |K_{i+1} - K_i|$$

实际上，根据该算法的原理不难看出，以下两个等式中，总是必有其一成立：

$$K_i = L_{i+1}$$

$$K_i = H_{i+1}$$

相应地，以下两式之一也必然成立：

$$D_j = N_{j+1} \cdot P_{j+1}$$

$$D_i = N_{i+1} \cdot (1 - P_{i+1})$$

因此无论如何，总是有：

由以上的定义，还可以导出：

$$\begin{aligned} D_j &= |K_{j+1} - K_j| \\ &= |E(K^* \mid S_1, S_2, \dots, S_j, S_{j+1}) - K_j| \\ &= |E(K^* - K_j \mid S_1, S_2, \dots, S_j, S_{j+1})| \end{aligned}$$

由柯西不等式可知：

$$D_j^2 = [E(K^* - K_j \mid S_1, S_2, \dots, S_j, S_{j+1})]^2$$

$$\leq [E([K^* - K_j]^2 \mid S_1, S_2, \dots, S_j, S_{j+1})]$$

根据条件期望值的性质，进一步地有：

$$E(D_i^2 \mid S_1, S_2, \dots, S_i)$$

$$\begin{aligned} &\leq E([E([K^* - K_j]^2 \mid S_1, S_2, \dots, S_j, S_{j+1}]) \mid S_1, S_2, \dots, S_j) \\ &= E([K^* - K_j]^2 \mid S_1, S_2, \dots, S_j) \end{aligned}$$

由(1)式，在依次转入 S_1, S_2, \dots, S_j 状态后，随机变量 K^* 的期望值为 K_j ，故上式也就是 K^* 在此时的条件方差。于是由(2)式，继续有：

最后，再次根据柯西不等式，并利用条件期望值的性质，由(3)式有：

$$\begin{aligned} [\mathbb{E}(D_j \mid S_1)]^2 &\leq \mathbb{E}(D_j^2 \mid S_1) \\ &= \mathbb{E}(\mathbb{E}(D_j^2 \mid S_1, S_2, \dots, S_{j-1}) \mid S_1) \\ &\leq \mathbb{E}(D_{j-1} \mid S_1) \end{aligned}$$

亦即：

$$E(D_j \mid S_1) \leq \sqrt{E(D_{j-1} \mid S_1)}$$

这就意味着，从进入第一步迭代之后，随后各步迭代所对应查找区间的宽度，将以平方根的速度逐次递减。

实际上，第一步迭代所对应的子区间也具有这一性质。请读者参照以上方法，独立补充证明。

d) 试证明：对于长度为 n 的此类向量，插值查找的期望运行时间为 $\theta(\log \log n)$ ；

【解答】

针对宽度为n的向量做插值查找时，记所需的时间为T(n)。于是有以下边界条件及递推方程：

$$\begin{aligned} T(1) &= \mathcal{O}(1) \\ T(n) &= T(\sqrt{n}) + \mathcal{O}(1) \end{aligned}$$

令：

$$S(n) = T(2^n)$$

则有：

$$\begin{aligned} S(1) &= \mathcal{O}(1) \\ S(n) &= S(n/2) + \mathcal{O}(1) \end{aligned}$$

解之可得：

$$S(n) = o(\log n)$$

对应地：

$$T(n) = S(\log n) = O(\log \log n)$$

e) 按照以上思路实现对应的插值查找算法，并通过实际测量，与二分查找等算法做一效率对比；

【解答】

请读者独立完成算法的编码与调试任务。

f) 你的实测对比结果，与理论分析是否吻合？若不吻合，原因何在？

【解答】

请读者独立完成实验，依据统计结果加以验证，并作出分析与解释。

[2-25] 对于几乎有序的向量，如教材代码 2.26 (60 页) 和代码 2.27 (60 页) 所示的起泡排序算法，都显得效率不足。

比如，即便乱序元素仅限于 $A[0, \sqrt{n}]$ 区间，最坏情况下仍需调用 `bubble()` 做 $\Omega(\sqrt{n})$ 次调用，共做 $\Omega(n)$ 次交换操作和 $\Omega(n^{3/2})$ 次比较操作，因此累计运行 $\Omega(n^{3/2})$ 时间。

a) 试改进原算法，使之在上述情况下仅需 $\mathcal{O}(n)$ 时间；

【解答】

可改进如代码x2.3和代码x2.4所示：

```
1 template <typename T> //向量的起泡排序
2 void Vector<T>::bubbleSort ( Rank lo, Rank hi ) //assert: 0 <= lo < hi <= size
3 { while ( lo < ( hi = bubble ( lo, hi ) ) ); } //逐趟做扫描交换，直至全序
```

代码x2.3 向量的起泡排序（改进版）

```
1 template <typename T> Rank Vector<T>::bubble ( Rank lo, Rank hi ) { //一趟扫描交换
2     Rank last = lo; //最右侧的逆序对初始化为 [lo - 1, lo]
3     while ( ++lo < hi ) //自左向右，逐一检查各对相邻元素
4         if ( _elem[lo - 1] > _elem[lo] ) { //若逆序，则
5             last = lo; //更新最右侧逆序对位置记录，并
6             swap ( _elem[lo - 1], _elem[lo] ); //通过交换使局部有序
7         }
8     return last; //返回最右侧的逆序对位置
9 }
```

代码x2.4 单趟扫描交换（改进版）

较之教材中的代码2.26和2.27，这里将逻辑型标志 `sorted` 改为秩 `last`，以记录各趟扫描交换所遇到的最后（最右）逆序元素。如此，在乱序元素仅限于 $A[0, \sqrt{n}]$ 区间时，仅需一趟扫描交换，即可将问题范围缩减至这一区间。累计耗时：

$$\mathcal{O}(n + (\sqrt{n})^2) = \mathcal{O}(n)$$

b) 继续改进，使之在如下情况下仅需 $\mathcal{O}(n)$ 时间：乱序元素仅限于 $A[n - \sqrt{n}, n]$ 区间；

【解答】

仿照a) 的思路与技巧，将扫描交换的方向调换为自后（右）向前（左），记录最前（最左）逆序元素。请读者独立完成这一改进。

c) 综合以上改进，使之在如下情况下仅需 $\mathcal{O}(n)$ 时间：乱序元素仅限于任意的 $A[m, m+\sqrt{n}]$ 区间。

【解答】

综合以上a) 和b) 的思路与技巧，方向交替地执行扫描交换，同时动态地记录和更新最左和最右的逆序元素。请读者独立完成这一改进。

[2-26] 根据教材 2.8.3 节所给递推关系以及边界条件试证明 ,如教材 62 页代码 2.28 所示 mergeSort() 算法的运行时间 $T(n) = \Theta(n \log n)$ 。

【解答】

教材中已针对该算法, 给出了如下边界条件及递推方程:

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 2 \times T(n/2) + \Theta(n) \end{aligned}$$

或等价地

$$T(n)/n = T(n/2)/(n/2) + \Theta(1)$$

以下若令:

$$S(n) = T(n)/n$$

则有:

$$\begin{aligned} S(1) &= \Theta(1) \\ S(n) &= S(n/2) + \Theta(1) \\ &= S(n/4) + \Theta(2) \\ &= \dots \\ &= S(n/2^k) + \Theta(k) \\ &= \Theta(\log n) \end{aligned}$$

于是有:

$$\begin{aligned} T(n) &= n \cdot S(n) \\ &= \Theta(n \log n) \end{aligned}$$

归并排序的边界条件及递推方程, 在算法复杂度分析中非常典型, 以上解法也极具有代表性。因此, 读者不妨记住这一递推模式, 并在今后作为基本结论直接应用。

[2-27] 如教材 62 页代码 2.28 所示 mergeSort() 算法, 即便在最好情况下依然需要 $\Omega(n \log n)$ 时间。

实际上略微修改这段代码, 即可使之在(子)序列业已有序时仅需线性时间。为此, mergeSort() 的每个递归实例仅需增加常数的时间, 且其它情况下的总体计算时间仍然保持 $\Theta(n \log n)$ 。

试给出你的改进方法, 并说明其原理。

【解答】

只需将原算法中的

```
merge(lo, mi, hi);
```

一句改为:

```
if (_elem[mi - 1] > _elem[mi]) merge(lo, mi, hi);
```

实际上按照原算法的流程, 在即将调用 `merge()` 接口对业已各自有序的向量区间 `[lo, mi)` 和 `[mi, hi)` 做二路归并之前, `_elem[mi - 1]` 即是前一 (左侧) 区间的末 (最靠右) 元素, 而 `_elem[mi]` 则是后一 (右侧) 区间的首 (最靠左) 元素。

于是，若属于本题所指（业已整体有序）的情况，则必有`_elem[mi - 1] ≤ _elem[mi]`；反之亦然。因此只需加入如上的比较判断，即可在这种情况下省略对`merge()`的调用。

不难看出，如此并不会增加该算法的渐进时间复杂度。

[2-28] 教材 63 页代码 2.29 中的二路归并算法 `merge()`，反复地通过 `new` 和 `delete` 操作申请和释放辅助空间。然而实验统计表明，这类操作的实际时间成本，大约是常规运算的 100 倍，故往往成为制约效率提高的瓶颈。

a) 试改写该算法，通过尽量减少此类操作，进一步优化整体效率；

【解答】

可以在算法启动时，统一申请一个足够大的缓冲区作为辅助向量`B[]`，并作为全局变量为所有递归实例公用；归并算法完成之后，再统一释放。

如此可以将动态空间申请的次数降至 $O(1)$ ，而不再与递归实例的总数 $O(n)$ 相关。当然，这样会在一定程度上降低代码的规范性和简洁性，代码调试的难度也会有所增加。

b) 试通过实测，对比并验证你的改进效果。

【解答】

请读者独立完成实验测试，并给出验证结论。

[2-29] 二路归并算法 `merge()`（教材 63 页代码 2.29）中的循环体内，两条并列语句的判断逻辑，并非完全对称。

a) 若将后一句中的“`C[k] < B[j]`”改为“`C[k] ≤ B[j]`”，对算法将有何影响？

【解答】

经如此调整之后，虽不致影响算法的正确性（仍可排序），但不再能够保证各趟二路归并的稳定性，整个归并排序算法的稳定性也因此不能保证。

原算法的控制逻辑可以保证稳定性。实际上，若两个子区间当前接受比较的元素分别为`B[j]`和`C[k]`，则唯有在前者严格大于后者时，才会将后者转移至`A[i++]`；反之，只要前者不大于后者（包含二者相等的情况），都会优先转移前者。由此可见，无论是子区间内部（相邻）的重复元素，还是子区间之间的重复元素，在归并之后依然能够保持其在原向量中的相对次序。

b) 若将前一句中的“`B[j] <= C[k]`”改为“`B[j] < C[k]`”，对算法将有何影响？

【解答】

当待归并的子向量之间有重复元素时，循环体内的两条处理语句均会失效，两个子向量的首元素都不会被转移，算法将在此处进入死循环。

c) 若同时做以上修改，对算法又将有何影响？

【解答】

不影响算法的正确性，仍可排序。然而每经过一趟归并，子向量之间的重复元素都会颠倒前后的次序，从而进一步地破坏整个归并排序算法的稳定性。

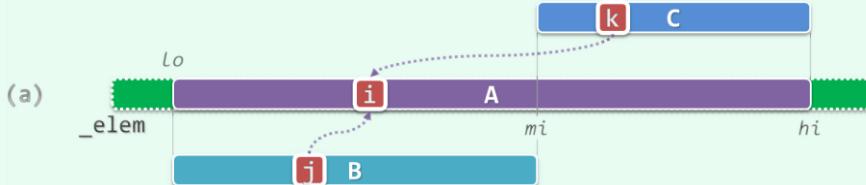
[2-30] 二路归并算法 `merge()` (教材 63 页代码 2.29) 中的循环体，虽然形式上简洁，但流程控制逻辑却较为复杂。

a) 试分情况验证并解释该算法的正确性；

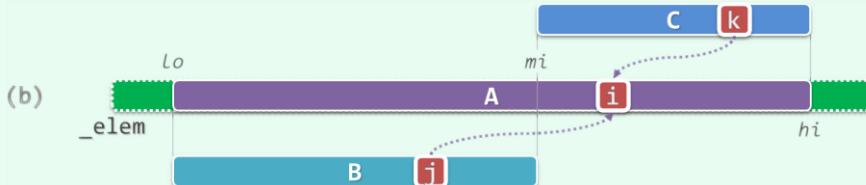
【解答】

这里之所以引入了较为复杂的控制逻辑，目的是为了统一对不同情况的处理。尽管如此可使代码在形式上更为简洁，但同时也会在一定程度上造成运行效率的下降。

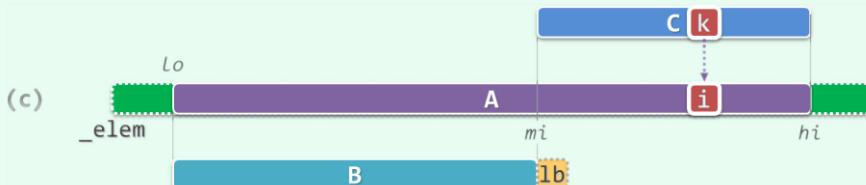
实际上，二路归并算法过程中可能出现的情况，如图x2.10至图x2.13所示无非四种。



图x2.10 B[]和C[]中的元素均未耗尽，且已转入A[]的元素总数i ≤ 1b



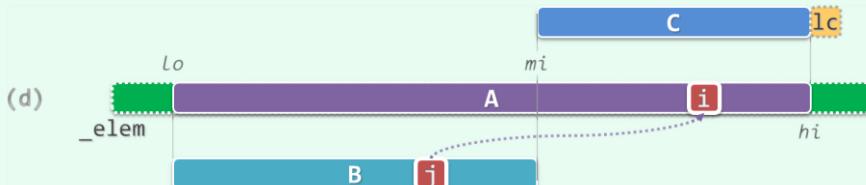
图x2.11 B[]和C[]中的元素均未耗尽，且已转入A[]的元素总数i > 1b



图x2.12 B[]中的元素先于C[]耗尽

就本章特定的二路归并而言，在情况(c)下C[]中剩余元素已不必移动。

此时，为更好地理解循环体内的控制逻辑 “ $1b \leq j \text{ || } (C[k] < B[j])$ ”，不妨假想地设置一个哨兵 $B[1b] = +\infty$ 。如此，“ $1b \leq j$ ” 即可作为特殊情况归入 “ $C[k] < B[j]$ ”。



图x2.13 C[]中的元素先于B[]耗尽

类似地，此时为更好地理解循环体内的控制逻辑 “ $1c \leq k \text{ || } (B[j] \leq C[k])$ ”，不妨假想地设置一个哨兵 $C[1c] = +\infty$ ，即可将 “ $1c \leq k$ ” 作为特殊情况归入 “ $B[j] < C[k]$ ”。

b) 基于以上理解，该循环体可以如何简化？

【解答】

可以考虑精简为如下形式（为便于对比，这里插入了一些空格）：

```
1 for ( Rank i = 0, j = 0, k = 0; j < 1b; ) { //将B[j]和C[k]中的小者续至A末尾
2     if ( ( k < 1c ) && ( C[k] < B[j] ) ) A[i++] = C[k++];
3     if ( ( 1c <= k ) || ( B[j] <= C[k] ) ) A[i++] = B[j++];
4 }
```

代码x2.5 有序向量二路归并算法的简化

请读者对照以上所列各种情况独立验证，尽管这里交换了循环体内两句的次序，同时删除了一些判断条件，却并不影响该算法的正确性和稳定性。

c) 如果从代码可维护性及运行效率的角度出发，该算法应该如何实现？

【解答】

不难看出，上述四种情况的发生次序，必然首先是(a)，然后（可能会）经过(b)，最后以(c)或(d)结束。因此从代码可维护性及运行效率的角度出发，不妨将(a和b)与(c或d)分为两个阶段，分别处理。如此虽然会增加代码量，但因判断逻辑可以进一步精简，反而会在一定程度上提高运行效率。

[2-31] 找到(v2.4之前版本)Python的**bisect**模块，阅读其中**bisect_right()**接口的实现代码。

a) 试以增加注释的形式，说明该接口的输入输出、功能语义、实现策略和算法实现；

【解答】

bisect模块增加注释之后的源代码，如代码x2.6所示：

```
def bisect_right(a, x, lo=0, hi=None): # 在有序向量区间a[lo, hi)中，采用二分策略查找x
    if hi is None: # hi未予明确指定时，默认值取作
        hi = len(a) # a的长度——因lo默认取作0，故默认时等效于对整个向量做查找
    while lo < hi: # 每步迭代仅需做一次比较判断，有两个分支
        mid = (lo+hi)//2 # 以中点为轴点
        if x < a[mid]: hi = mid # 经一比较后，若x小于轴点，则向左深入[lo, mi)
        else: lo = mid + 1 # 否则向右深入(mi, hi)
    return lo # lo为x在a[lo, hi)中适当的插入位置
```

代码x2.6 增加注释后，Python的**bisect**模块中**bisect_right**接口的源代码

需说明的是，出于效率的考虑，在Python v2.4之后版本中，该接口已改用C语言实现。

b) 就以上方面而言，该接口与本章向量的哪个接口基本类似？同时，又有什么区别？

【解答】

bisect_right()的功能语义、算法原理及流程，与**Vector::binsearch()**的版本C（教材56页代码2.24）几乎如出一辙。二者之间的差异无非在于，前者返回的秩比后者大一。

[2-32] 自学 C++ STL 中 vector 容器的使用方法，阅读对应的源代码。

【解答】

请读者独立完成相关代码的阅读和分析。

[2-33] 自学 Java 语言中的 Java.util.ArrayList 和 java.util.Vector 类，并阅读对应的源代码。

【解答】

请读者独立完成相关代码的阅读和分析任务。

[2-34] 位图（Bitmap）是一种特殊的序列结构，可用以动态地表示由一组（无符号）整数构成的集合。

其长度无限，且其中每个元素的取值均为布尔型（初始均为 false），支持的操作接口主要包括：

```
void set(int i); //将第i位置为true ( 将整数i加入当前集合 )
void clear(int i); //将第i位置为false ( 从当前集合中删除整数i )
bool test(int i); //测试第i位是否为true ( 判断整数i是否属于当前集合 )
```

a) 试给出 Bitmap 类的定义，并具体实现以上接口；

【解答】

一种可行的实现方式，如代码x2.7所示。

```
1 class Bitmap { //位图Bitmap类
2 private:
3     char* M; int N; //比特图所存放的空间M[]，容量为N*sizeof(char)*8比特
4 protected:
5     void init ( int n ) { M = new char[N = ( n + 7 ) / 8]; memset ( M, 0, N ); }
6 public:
7     Bitmap ( int n = 8 ) { init ( n ); } //按指定或默认规模创建比特图（为测试暂时选用较小的默认值）
8     Bitmap ( char* file, int n = 8 ) //按指定或默认规模，从指定文件中读取比特图
9     { init ( n ); FILE* fp = fopen ( file, "r" ); fread ( M, sizeof ( char ), N, fp ); fclose
( fp ); }
10    ~Bitmap() { delete [] M; M = NULL; } //析构时释放比特图空间
11
12    void set ( int k ) { expand ( k ); M[k >> 3] |= ( 0x80 >> ( k & 0x07 ) ); }
13    void clear ( int k ) { expand ( k ); M[k >> 3] &= ~ ( 0x80 >> ( k & 0x07 ) ); }
14    bool test ( int k ) { expand ( k ); return M[k >> 3] & ( 0x80 >> ( k & 0x07 ) ); }
15
16    void dump ( char* file ) //将位图整体导出至指定的文件，以便对此后的新位图批量初始化
17    { FILE* fp = fopen ( file, "w" ); fwrite ( M, sizeof ( char ), N, fp ); fclose ( fp ); }
18    char* bits2string ( int n ) { //将前n位转换为字符串——
19        expand ( n - 1 ); //此时可能被访问的最高位为bitmap[n - 1]
20        char* s = new char[n + 1]; s[n] = '\0'; //字符串所占空间，由上层调用者负责释放
21        for ( int i = 0; i < n; i++ ) s[i] = test ( i ) ? '1' : '0';
22        return s; //返回字符串位置
23    }
```



```

24 void expand ( int k ) { //若被访问的Bitmap[k]已出界，则需扩容
25     if ( k < 8 * N ) return; //仍在界内，无需扩容
26     int oldN = N; char* oldM = M;
27     init ( 2 * k ); //与向量类似，加倍策略
28     memcpy_s ( M, N, oldM, oldN ); delete [] oldM; //原数据转移至新空间
29 }
30 };

```

代码x2.7 位图Bitmap类

这里使用了一段动态申请的连续空间M[], 并依次将其中的各比特位与位图集合中的各整数一一对应：若集合中包含整数k，则该段空间中的第k个比特位为1；否则该比特位为0。

在实现上述一一对应关系时，这里借助了高效的整数移位和位运算。鉴于每个字节通常包含8个比特，故通过移位运算：

`k >> 3`

即可确定对应的比特位所属字节的秩；通过逻辑位与运算：

`k & 0x07`

即可确定该比特位在此字节中的位置；通过移位操作：

`0x80 >> (k & 0x07)`

即可得到该比特位在此字节中对应的数值掩码（mask）。

得益于这种简明的对应关系，只需在局部将此字节与上述掩码做逻辑或运算，即可将整数k所对应的比特位设置为1；将此字节与上述掩码做逻辑与运算，即可测试该比特位的状态；将此字节与上述掩码的反码做逻辑位与运算，即可将该比特位设置为0。

这里还提供了一个dump()接口，可以将位图整体导出至指定的文件，以便对此后的新位图批量初始化。例如在后面9.3节实现高效的散列表结构时，经常需要快速地找出不小于某一整数的最小素数。为此，可以借助Eratosthenes算法，事先以位图形式筛选出足够多个候选素数，并通过dump()接口将此集合保存至文件。此后在使用散列表时，可一次性地读入该文件，即可按照需要反复地快速确定合适的素数。

与可扩充向量一样，一旦即将发生溢出，这里将调用expand()接口扩容。可见，这里采用的也是“加倍”的扩容策略。

b) 试针对你的实现，分析各接口的时间和空间复杂度；

62

【解答】

根据以上分析，`set()`、`clear()`和`test()`等接口仅涉及常数次基本运算，故其时间复杂度均为 $O(1)$ 。可见，这种实现方式巧妙地发挥了向量之“循秩访问”方式的优势。

此外，相对于四则运算等常规运算，这里所涉及的整数移位和位运算更为高效，因此该数据结构实际的运行效率非常高，该结构也是一种典型的实用数据结构。

这里，位图向量所占的空间线性正比于集合的取值范围——在很多应用中，这一范围就是问题本身的规模，故通常不会导致渐进空间复杂度的增加。

c) 创建 Bitmap 对象时，如何节省下为初始化所有元素所需的时间？(提示：参考文献[4][9])

【解答】

首先考查简单的情况：位图结构只需提供`test()`和`set()`接口，暂且不需要`clear()`接口。针对此类需求，一种可行的方法大致如代码x2.8所示。

```

1 class Bitmap { //位图Bitmap类：以空间作为补偿，节省初始化时间（仅允许插入，不支持删除）
2 private:
3     Rank* F; Rank N; //规模为N的向量F，记录[k]被标记的次序（即其在栈T[]中的秩）
4     Rank* T; Rank top; //容量为N的栈T，记录被标记各位秩的栈，以及栈顶指针
5
6 protected:
7     inline bool valid ( Rank r ) { return ( 0 <= r ) && ( r < top ); }
8
9 public:
10    Bitmap ( Rank n = 8 ) //按指定（或默认）规模创建比特图（为测试暂时选用较小的默认值）
11    { N = n; F = new Rank[N]; T = new Rank[N]; top = 0; } //在O(1)时间内隐式地初始化
12    ~Bitmap() { delete [] F; delete [] T; } //析构时释放空间
13
14 // 接口
15    inline void set ( Rank k ) { //插入
16        if ( test ( k ) ) return; //忽略已带标记的位
17        F[k] = top++; T[ F[k] ] = k; //建立校验环
18    }
19    inline bool test ( Rank k ) //测试
20    { return valid ( F[k] ) && ( k == T[ F[k] ] ); }
21 };

```



代码x2.8 可快速初始化的Bitmap对象（仅支持set()操作）

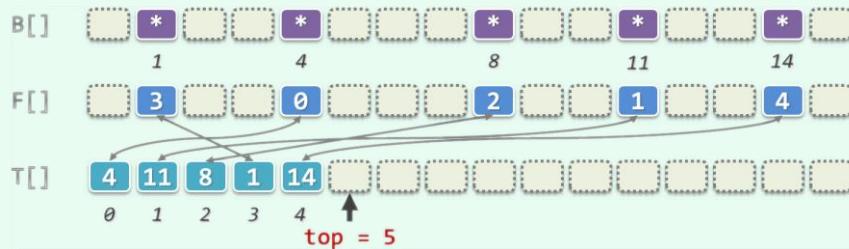
首先，将代码x2.7中`Bitmap`类的内部空间`M[]`，代替换为一对向量`F[]`和`T[]`，其中元素均为`Rank`类型，其规模`N`均与（逻辑上的）位图结构`B[]`相同。实际上，`T[]`的工作方式将等效于栈，栈顶由`top`指示，初始`top = 0`。

请注意，与代码x2.7的版本相比，这里对两个向量均未做显式的初始化。

此后，每当需要调用`set(k)`标记新的`B[k]`位时，即可将`k`压入栈`T[]`中，并将该元素（当前的顶元素）在栈中的秩存入`F[k]`。

如此产生的效果是，在`k`与`T[F[k]]`之间建立了一个校验环路。也就是说，当`F[k]`指向栈`T[]`中的某个有效元素（`valid(F[k])`），而且该元素`T[F[k]]`恰好就等于`k`时，在逻辑上必然等效于`B[k] = true`；更重要的是，反之亦然。因此如代码x2.8所示，`test(k)`接口只需判断以上两个条件是否同时成立。

经如此改造之后，位图结构的一个运转实例如图x2.14所示。



图x2.14 通过引入两个等长的向量，在 $\mathcal{O}(1)$ 时间内初始化Bitmap对象

按以上方法，首先在 $\mathcal{O}(1)$ 时间内对该位图结构做初始化。接下来，依次标记：

$B[4], B[11], B[8], B[1], B[14], \dots$

相应地依次压入栈T[]中的分别是：

$T[0] = 4, T[1] = 11, T[2] = 8, T[3] = 1, T[4] = 14, \dots$

向量F[]中依次存入的秩为：

$F[4] = 0, F[11] = 1, F[8] = 2, F[1] = 3, F[14] = 4, \dots$

不难看出，整个过程中，凡可通过test(k)逻辑的任何比特位k，均被标记过；反之亦然。

由上同时可见，经如上改造之后的test()和set()接口，各自仍然仅需 $\mathcal{O}(1)$ 时间。

当然，以上方法仅限于标记操作set()，尚不支持清除操作clear()。如需兼顾这两个接口，就必须有效地辨别两种无标记的位：从未标记过的，以及曾经一度被标记后来又被清除的。否则，每次为无标记的位增加标记时，若简单地套用目前的set()接口为其增加一个校验环，则栈T[]的规模将线性正比于累积的操作次数，从而无法限制在N以内（尽管向量F[]仍可以），整个结构的空间复杂度也将随着操作次数的增加严格单调的增加。

能够有效区分以上两种无标记位的一种Bitmap类，可实现如代码x2.9所示。



```

1 class Bitmap { //位图Bitmap类：以空间作为补偿，节省初始化时间（既允许插入，亦支持删除）
2 private:
3     Rank* F; Rank N; //规模为N的向量F，记录[k]被标记的次序（即其在栈T[]中的秩）
4     Rank* T; Rank top; //容量为N的栈T，记录被标记各位秩的栈，以及栈顶指针
5
6 protected:
7     inline bool valid ( Rank r ) { return ( 0 <= r ) && ( r < top ); }
8     inline bool erased ( Rank k ) //判断[k]是否曾被标记过，且后来又被清除
9     { return valid ( F[k] ) && ! ( T[ F[k] ] + 1 + k ); } //这里约定：T[ F[k] ] = -1 - k
10
11 public:
12     Bitmap ( Rank n = 8 ) //按指定（或默认）规模创建比特图（为测试暂时选用较小的默认值）
13     { N = n; F = new Rank[N]; T = new Rank[N]; top = 0; } //在O(1)时间内隐式地初始化
14     ~Bitmap() { delete [] F; delete [] T; } //析构时释放空间

```

```

15
16 // 接口
17     inline void set ( Rank k ) { //插入
18         if ( test ( k ) ) return; //忽略已带标记的位
19         if ( !erased ( k ) ) F[k] = top++; //若系初次标记，则创建新校验环
20         T[ F[k] ] = k; //若系曾经标记后被清除，则恢复原校验环
21     }
22     inline void clear ( Rank k ) //删除
23     { if ( test ( k ) ) T[ F[k] ] = - 1 - k; } //忽略不带标记的位
24     inline bool test ( Rank k ) //测试
25     { return valid ( F[k] ) && ( k == T[ F[k] ] ); }
26 };

```

代码x2.9 可快速初始化的Bitmap对象（兼顾set()和clear()操作）

这里的clear()接口并非简单地破坏原校验环，而是将T[F[k]]取负之后再减一。也就是说，可以在与正常校验环不相冲突的前提下，就地继续保留原校验环的信息。

基于这一约定，set(k)接口只需调用erase(k)，即可简明地判定[k]究竟属于哪种类型。若系从未标记过的，则按此前的方法新建一个校验环；否则可以直接恢复原先的校验环。

不难再次确认，在扩充后的版本中，各接口依然保持常数的时间复杂度。

最后，考查经改进之后Bitmap结构的空间复杂度。尽管表面上看，F[]和T[]的规模均不超过N，但这并不意味着整个结构所占空间的总量渐进不变。关键在于，两个向量的元素类型已不再是比特位或逻辑位，而是秩。二者的本质区别在于，前一类元素自身所占空间与整体规模无关，而后者有关。具体来说，这里Rank类型的取值必须足以涵盖Bitmap的规模；反之，可用Bitmap的最大规模也不能超越Rank类型的取值范围。比如，若Rank为四个字节的整数，则Bitmap的规模无法超过 $2^{31} - 1 = \mathcal{O}(10^9)$ ，否则Rank自身的字宽必须相应加大。所幸，目前的多数应用尚不致超越这个规模，因此仍可近似地认为以上改进版Bitmap具有线性的空间复杂度。

与之相关的另一问题是，目前的版本仍不支持动态扩容。我们将这一任务留给读者，并请读者对可扩容版Bitmap结构的空间、时间复杂度做一分析。

[2-35] 利用Bitmap类设计算法，在 $\mathcal{O}(n)$ 时间内剔除n个ASCII字符中的重复字符，各字符仅保留一份。

【解答】

将非重复的ASCII字符视作一个集合，并将其组织为一个Bitmap结构——ASCII编码为k的字符，对应于其中第k个比特位。

初始时，该集合为空，Bitmap结构中的所有比特位均处于0状态。以下，只需在 $\mathcal{O}(n)$ 时间内遍历所有的输入字符，并对ASCII编码为k的字符，通过set(k)接口将其加入集合。

请注意，这里使用的Bitmap结构只需128个比特位。因此，最后只需再花费 $\mathcal{O}(128) = \mathcal{O}(1)$ 时间遍历一趟所有的比特位，并输出所有通过test()测试的比特位，即可完成字符集的去重。

[2-36] 利用 Bitmap 类设计算法，快速地计算不大于 10^8 的所有素数。（提示：Eratosthenes 筛法）

【解答】

比如，可以采用Eratosthenes^④筛法。该算法一种可行的实现方式如代码x2.10所示。



```

1 #include "../Bitmap/Bitmap.h" //引入Bitmap结构
2
3 ****
4 * 筛法求素数
5 * 计算出不大于n的所有素数
6 * 不计内循环，外循环自身每次仅一次加法、两次判断，累计O(n)
7 * 内循环每趟迭代O(n/i)步，由素数定理至多n/ln(n)趟，累计耗时不过
8 *      n/2 + n/3 + n/5 + n/7 + n/11 + ...
9 *      < n/2 + n/3 + n/4 + n/6 + n/7 + ... + n/(n/ln(n))
10 *     = O(n(ln(n/ln(n)) - 1))
11 *     = O(nln(n) - nln(ln(n)) - 1)
12 *     = O(nlog(n))
13 * 如下实现做了进一步优化，内循环从i * i而非i + i开始，迭代步数由O(n / i)降至O(max(1, n / i - i))
14 ****
15 void Eratosthenes ( int n, char* file ) {
16     Bitmap B ( n ); B.set ( 0 ); B.set ( 1 ); //0和1都不是素数
17     for ( int i = 2; i < n; i++ ) //反复地，从下一
18         if ( !B.test ( i ) ) //可认定的素数i起
19             for ( int j = __min ( i, 46340 ) * __min ( i, 46340 ); j < n; j += i ) //以i为间隔
20                 B.set ( j ); //将下一个数标记为合数
21     B.dump ( file ); //将所有整数的筛选标记统一存入指定文件，以便日后直接导入
22 }
```

代码x2.10 Eratosthenes素数筛选算法

这里的Bitmap结构B，相当于Eratosthenes的羊皮纸。初始时，（除0和1之外的）所有整数都有可能是素数，正如羊皮纸在开始前是完好无损的。

算法的主循环启动之后，将逐一检测各整数i（即与之对应的比特位[i]）。若当前的整数i已可被判定为合数（即B.test(i) = true，亦相当于羊皮纸上对应的方格已穿洞），则忽略之。否则，整数i应为素数，故须从2i开始以i为间隔，将后续所有形如j = ki ($k \geq 2$) 的整数j逐一标记为合数（即B.set(j)，亦相当于在羊皮纸上对应方格中穿孔）。

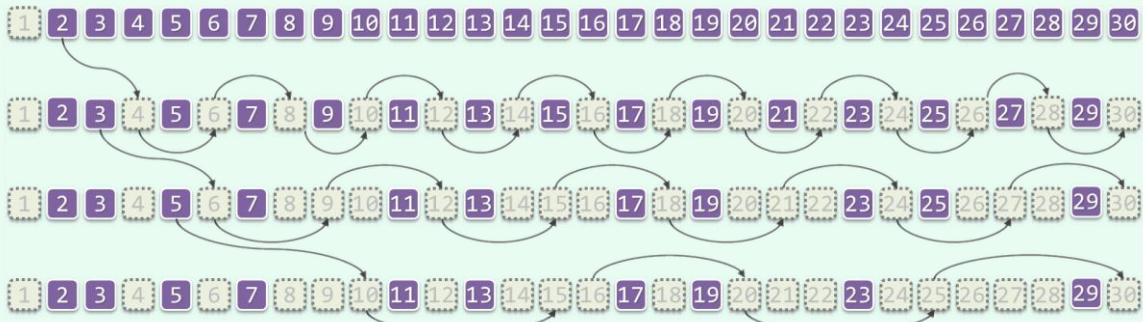
图x2.15逐行地依次给出了该算法在初始状态以及前三次迭代之后，Bitmap位图结构（Eratosthenes筛子）所对应的内部组成和状态。其中，白色的比特位（有穿孔的方格）所对应的整数已确认为合数；跨行的箭头联线，表示在确认了一个新的素数i之后，开始从2i开始筛

^④ Eratosthenes (276-194 B.C.)，埃拉托斯特尼，古希腊先哲。经纬坐标系的发明者。

其对地球半径简捷而精准地计算，也是封底估算方法的经典实例。

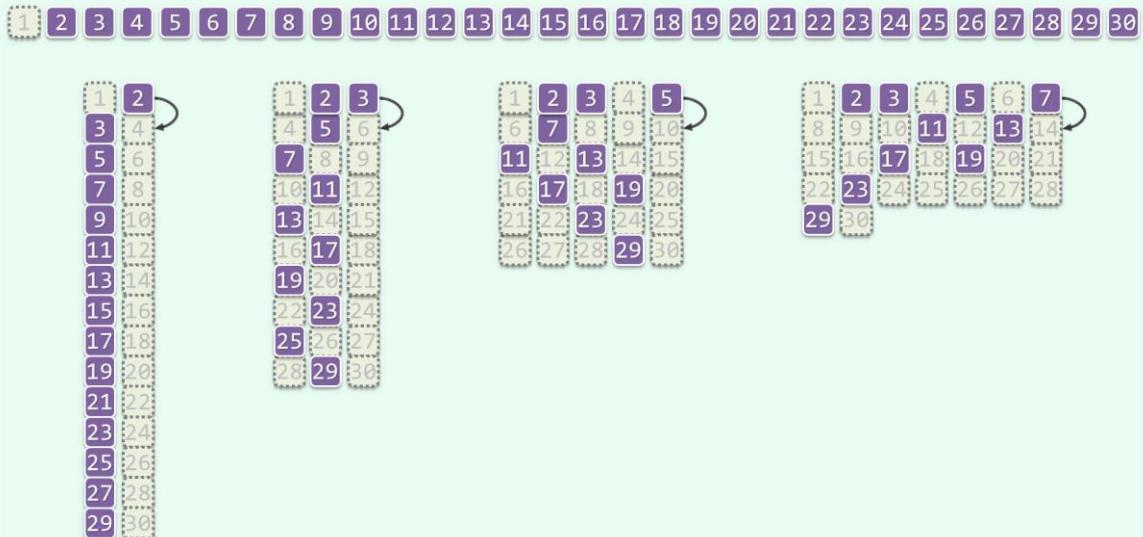
此处实现的算法也源自于他，并被形象地称作埃拉托斯特尼的筛子（the sieve of Eratosthenes）。

除其所有的整倍数；而同行内的箭头，则对应于逐个筛除其整倍数的过程。



图x2.15 Eratosthenes算法的实例

通过不断重排所有整数，我们可以更好地理解该算法的原理及过程。在算法确认了一个新的素数*i*之后，不妨将所有整数以*i*个为一行，顺次排成*i*列的矩阵（当然，理论上应有无穷行）。如此，各次迭代所对应的重排结果应如图x2.16所示，依次为2列、3列、5列、7列、....。



图x2.16 Eratosthenes算法：每次迭代中所筛除的整数，恰好就是重排矩形的最右侧一列

该算法可在 $O(n \log n)$ 时间内计算出不超过n的所有素数，具体分析可参见代码所附注释。

实际上，这里所实现的版本已在以上思路的基础上做了改进（尽管不是渐进意义上的改进）。

不难发现，在从对应于素数*i*的重排矩阵中筛除其最右侧一列时，完全可以直接从*i*²（而不是 $2i$ ）出发。以*i = 5*为例，如图x2.17所示完全可以从25（而不是10）出发。实际上，该列中介于 $[2i, i^2)$ 之间的整数，均应该已经在此前的某次迭代中被筛除了。

同理，若只是考查不超过n的素数，则当*i > \sqrt{n}*后，外循环即可终止。比如在上例中，当*i > |\sqrt{n}| = |\sqrt{30}| = 5*后，外循环即可终止。也就是说，最后一趟（*i = 7*）的筛除完全可以省略。



图x2.17 Eratosthenes算法的改进

[2-37] 教材 12 页算法 1.3 中，在选出三个数之后还需对它们做排序。试证明：

a) 至多只需比对元素的大小三次，即可完成排序；

【解答】

首先，任取两个数并比较大小，记作 $a < b$ 。如此，整个数轴被分为三个区间：

$$(-\infty, a) \quad (a, b) \quad (b, +\infty)$$

此后，至多再做两次比较，即可确定第三个数落在其中哪个区间。

b) 在最坏情况下，的确至少需要比对元素的大小三次，才能完成排序。

【解答】

3个数的排序结果，共有 $3! = 6$ 种。也就是说，该问题的任何一个算法所对应的比较树 (comparison tree) 中，至少应有6个叶节点，故树高不致低于：

$$\lceil \log_2 6 \rceil = 3$$

[2-38] 代数判定树 (algebraic decision tree, ADT) 是比较树的推广，其中的节点分别对应于根据某一代数表达式做出的判断。例如，比较树中各节点所对应的“ $a == b$ ”式判等以及“ $a < b$ ”式比较，均可统一为根据一次代数表达式“ $a - b$ ”取值符号的判断。

a) 对应于教材 2.7.4 节所列比较树的性质，代数判定树有哪些相似的性质？

【解答】

与比较树类似地，代数判定树也具有如下性质：

- ① 每一内部节点各对应于一次（基于代数计算数值正负符号的）判定操作
- ② 内部节点的左、右分支，分别对应于在不同的数值符号下的执行方向
- ③ 叶节点（亦即，根到叶节点的路径）对应于算法某次执行的完整过程及输出
- ④ 反过来，算法的每一运行过程都对应于从根到某一叶节点的路径

b) 2.7.5 节中基于比较树模型的下界估计方法^{[27][28]}，可否推广至代数判定树？如何推广？

【解答】

完全可以推广。

需要特别注意的是，对一般代数式的求值本身未必仍然属于基本操作，故不见得可以在常数时间内完成。例如在高维空间中，为计算两个点之间的欧氏距离所需的时间应线性正比于空间的维度 d ，若不将 d 视作常数，则此时欧氏距离的计算即不属于基本操作。

因此在代数判定树中，应根据各节点所对应代数计算操作的复杂度，以根节点到叶节点通路的加权长度，来度量各种输出所对应的计算成本。

[2-39] 任给 12 个互异的整数，其中 10 个已组织为一个有序序列，现需要插入剩余的两个以完成整体排序。若采用 CBA 式算法，最坏情况下至少需做几次比较？为什么？

【解答】

对于该问题的任一算法，都可以将其中所有的分支描述为一棵代数判定树。根据排列组合中

基础知识，可能的输出数目应为：

前一整数可能的插入位置数 \times 后一整数可能的插入位置数 = 11 \times 12 = 132
这也是该判定树应含叶节点数目的下限。

于是对应地，判定树的高度应至少是：

$$\lceil \log_2 132 \rceil = 8$$

这也是此类算法在最坏情况下需做比较操作次数的下限。

[2-40] 经过至多 $(n - 1) + (n - 2) = 2n - 3$ 次比较，不难从任何存有 n 个整数的向量中找出最大者和次大者。试改进这一算法，使所需的比较次数（即便在最坏情况下）也不超过 $\lceil 3n/2 \rceil - 2$ 。

【解答】

可以采用分治策略，通过二分递归解决该问题。

算法的具体过程为：将原问题划分为两个子问题，分别对应于向量的前半部分和后半部分。以下，在递归地求解两个子问题（即找出两个子向量各自的最大和次大元素）后，只需两次比较操作，即可得到原问题的解（即确定整个向量中的最大和次大元素）。

实际上，若前一子向量中的最大、次大元素分别为 a_1 和 a_2 ，后一子向量中的最大、次大元素分别为 b_1 和 b_2 ，则全局的最大元素必然选自 a_1 和 b_1 之间。不失一般性地，设：

$$a_1 = \max(a_1, b_1)$$

于是全局的次大元素必然选自 a_2 和 b_1 ，亦即：

$$\max(a_2, b_1)$$

若将该算法的运行时间记作 $T(n)$ ，则根据以上分析，可得边界条件及递推方程如下：

$$T(2) = 1$$

$$T(n) = 2*T(n/2) + 2$$

若令：

$$S(n) = [T(n) + 2]/n$$

则有：

$$S(n) = S(n/2) = S(n/4) = \dots = S(2) = 3/2$$

故有：

$$T(n) = \lceil 3n/2 \rceil - 2$$

比如，若利用以上算法从任意8个整数中找出最大、次大元素，则即便是在最坏情况下，也只需 $T(8) = 10$ 次比较操作。

请注意，这里的关键性技巧在于，为合并子问题的解，可以仅需2次而不是3次比较操作。
否则，对应的递推关系应是：

$$T(n) = 2*T(n/2) + 3$$

解之即得：

$$T(n) = 2n - 3$$

仍以8个整数为例，在最坏情况下可能需要进行13次比较操作。

[2-41] 试证明，对于任一 $n \times m$ 的整数矩阵 M ，若首先对每一列分别排序，则继续对每一行分别排序后，其中的各列将依然有序（一个实例如图x2.18所示）。（提示：只需考查 $n = 2$ 的情况）

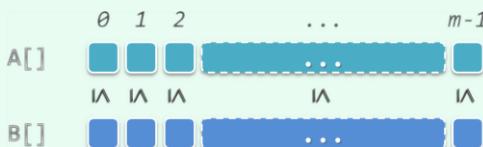
5	8	7	3	5	0	3	1	3	2	0	1	2	3	3
1	5	2	8	8	1	5	2	4	5	1	2	4	5	5
0	9	4	6	2	5	8	4	6	7	4	5	6	7	8
6	3	1	4	7	6	9	7	8	8	6	7	8	8	9

图x2.18 4×5的矩阵实例：经逐列排序再逐行排序后，每行、每列均各自有序

【解答】

因各行的排序独立进行，故只需证明以上命题对任意两行成立。

在已逐列排序的矩阵中，任取两行 $A[\theta, m)$ 和 $B[\theta, m)$ 。如图x2.19所示，不妨设 $A[]$ 位于 $B[]$ 之上方，于是在经逐列排序之后，对所有的 $\theta \leq k < m$ 均有 $A[k] \leq B[k]$ 成立。

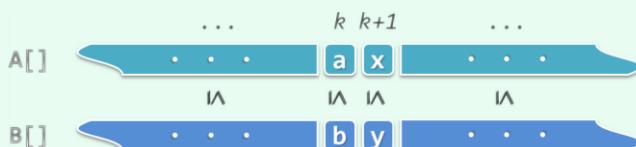


图x2.19 只需考查沿纵向捉对有序的任意两行

以下将通过两种方法证明，在继而再做逐行排序之后，两行的元素沿纵向依然捉对有序。

【证法 A】

不妨假想地采用起泡排序算法，同步地对各行实施排序。如此只需证明，该算法每向前迭代一步， A 和 B 中的元素沿纵向依次捉对的有序性，依然继续保持。



图x2.20 起泡排序的每一步，都是考查一对相邻元素

实际上如图x2.20所示，在每一步迭代中，该算法都仅逐行考查同一对相邻元素，比如：

$$A[k] = a \text{ 和 } A[k + 1] = x$$

$$B[k] = b \text{ 和 } B[k + 1] = y$$

不妨假设，在此之前已有：

$$a \leq b \text{ 和 } x \leq y$$

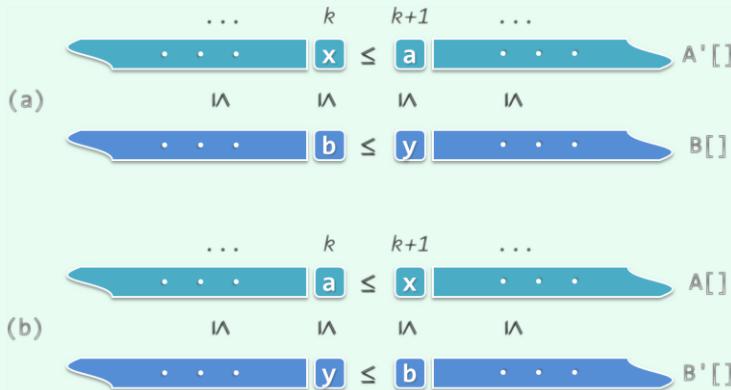
经过此步迭代之后，尽管 a 和 x 可能互换位置， b 和 y 也可能互换位置，但总体而言无非四种情况。其中，对于两行均无交换和同时交换的情况，命题显然成立。

故以下如图x2.21所示，只需考查其中只有一行交换的两种情况。

首先，设仅交换 a 和 x 。于是如图x2.21(a)所示，必有：

$$A'[k] = x \leq A'[k+1] = a \leq b = B[k] = B'[k]$$

$$A'[k+1] = a \leq b \leq y = B[k+1] = B'[k+1]$$



图x2.21 只需考查仅有两行进行交换的两种情况：(a) a 和 x 交换；(b) b 和 y 交换

反之，设仅交换 b 和 y 。于是如图x2.21(b)所示，必有：

$$A'[k] = A[k] = a \leq A[k+1] = x \leq y = B'[k]$$

$$A'[k+1] = A[k+1] = x \leq y = B'[k] \leq b = B'[k+1]$$

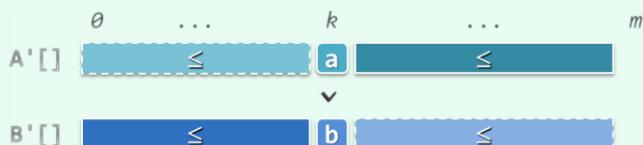
原命题故此得证。

需特别强调的是，这里不能采用插入排序或选择排序等其它算法——它们与起泡排序不同，不能保证在排序过程中，各行之间能够严格同步地进行比较，故不能直接支持以上推导过程。

【证法B】

反证。假设如图x2.22所示，在经逐行排序之后有：

$$A'[k] = a > b = B'[k]$$



图x2.22 假设逐行排序之后，沿纵向出现一对逆序元素 a 和 b

然而根据以上假设条件，以下可以导出，这两类元素的总数不少于 $m + 1$ ，从而导致悖论。

首先考查并统计小于 a 的元素。为此可以注意到以下事实：

$$B'[0, k] \leq B'[k] = b < a = A'[k]$$

这就意味着，对于 $B'[0, k]$ （亦来自于原 $B[]$ ）中的每一元素，在原 $A[]$ 中都应有一个不大于 b （亦即小于 a ）的元素与之对应——也就是说，在 $A[]$ 中至少有 $k + 1$ 个元素小于 a 。

再来统计不小于 a 的元素。既然 $a \leq A[k, m]$ ，故 $A[]$ 中至少有 $m - k$ 个元素不小于 a 。

因此，两类元素合计总数至少应为：

$$(k + 1) + (m - k) = m + 1 > m$$

