第2章

向量

数据结构是数据项的结构化集合，其结构性表现为数据项之间的相互联系及作用，也可以理解为定义于数据项之间的某种逻辑次序。根据这种逻辑次序的复杂程度，大致可以将各种数据结构划分为线性结构、半线性结构与非线性结构三大类。在线性结构中，各数据项按照一个线性次序构成一个整体。最为基本的线性结构统称为序列（sequence），根据其中数据项的逻辑次序与其物理存储地址的对应关系不同，又可进一步地将序列区分为向量（vector）和列表（list）。在向量中，所有数据项的物理存放位置与其逻辑次序完全吻合，此时的逻辑次序也称作秩（rank）；而在列表中，逻辑上相邻的数据项在物理上未必相邻，而是采用间接定址的方式通过封装后的位置（position）相互引用。

本章的讲解将围绕向量结构的高效实现而逐步展开，包括其作为抽象数据类型的接口规范以及对应的算法，尤其是高效维护动态向量的技巧。此外，还将针对有序向量，系统介绍经典的查找与排序算法，并就其性能做一分析对比，这也是本章的重点与难点所在。最后，还将引入复杂度下界的概念，并通过建立比较树模型，针对基于比较式算法给出复杂度下界的统一界定方法。

§2.1 从数组到向量

2.1.1 数组

C、C++和Java等程序设计语言，都将数组作为一种内置的数据类型，支持对一组相关元素的存储组织与访问操作。具体地，若集合S由n个元素组成，且各元素之间具有一个线性次序，则可将它们存放于起始于地址A、物理位置连续的一段存储空间，并统称作数组（array），通常以A作为该数组的标识。具体地，数组A[]中的每一元素都唯一对应于某一下标编号，在多数高级程序设计语言中，一般都是从0开始编号，依次是0号、1号、2号、...、n-1号元素，记作：

$A = \{ a_0, a_1, ..., a_{n-1} \}$ 或者

$A[0, n) = \{ A[0], A[1], ..., A[n - 1] \}$

其中，对于任何$0 \le i < j < n$，A[i]都是A[j]的前驱（predecessor），A[j]都是A[i]的后继（successor）。特别地，对于任何$i \ge 1$，A[i - 1]称作A[i]的直接前驱（immediate predecessor）；对于任何$i \le n - 2$，A[i + 1]称作A[i]的直接后继（immediate successor）。任一元素的所有前驱构成其前缀（prefix），所有后继构成其后缀（suffix）。

采用这一编号规范，不仅可以使得每个元素都通过下标唯一指代，而且可以使我们直接访问到任一元素。这里所说的“访问”包含读取、修改等基本操作，而“直接”则是指这些操作都可以在常数时间内完成——只要从数组所在空间的起始地址A出发，即可根据每一元素的编号，经过一次乘法运算和一次加法运算，获得待访问元素的物理地址。具体地，若数组A[]存放空间的起始地址为A，且每个元素占用s个单位的空间，则元素A[i]对应的物理地址为：

$A + i \times s$

因其中元素的物理地址与其下标之间满足这种线性关系，故亦称作线性数组（linear array）。

2.1.2 向量

按照面向对象思想中的数据抽象原则，可对以上的数组结构做一般性推广，使得其以上特性更具普遍性。向量（vector）就是线性数组的一种抽象与泛化，它也是由具有线性次序的一组元素构成的集合$V = \{ v_0, v_1, \ldots, v_{n-1} \}$，其中的元素分别由秩相互区分。

各元素的秩（rank）互异，且均为[0, n)内的整数。具体地，若元素e的前驱元素共计r个，则其秩就是r。以此前介绍的线性递归为例，运行过程中所出现过的所有递归实例，按照相互调用的关系可构成一个线性序列。在此序列中，各递归实例的秩反映了它们各自被创建的时间先后，每一递归实例的秩等于早于它出现的实例总数。反过来，通过r亦可唯一确定$e = v_r$。这是向量特有的元素访问方式，称作“循秩访问”（call-by-rank）。

经如此抽象之后，我们不再限定同一向量中的各元素都属于同一基本类型，它们本身可以是来自于更具一般性的某一类的对象。另外，各元素也不见得同时具有某一数值属性，故而并不保证它们之间能够相互比较大小。

以下首先从向量最基本的接口出发，设计并实现与之对应的向量模板类。然后在元素之间具有大小可比性的假设前提下，通过引入通用比较器或重载对应的操作符明确定义元素之间的大小判断依据，并强制要求它们按此次序排列，从而得到所谓有序向量，并介绍和分析此类向量的相关算法及其针对不同要求的各种实现版本。

§2.2 接口

2.2.1 ADT接口

作为一种抽象数据类型，向量对象应支持如下操作接口。

表2.1 向量ADT支持的操作接口

操 作 接 口	功 能	适 用 对 象
size()	报告向量当前的规模（元素总数）	向量
get(r)	获取秩为r的元素	向量
put(r, e)	用e替换秩为r元素的数值	向量
insert(r, e)	e作为秩为r元素插入，原后继元素依次后移	向量
remove(r)	删除秩为r的元素，返回该元素中原存放的对象	向量
disordered()	判断所有元素是否已按非降序排列	向量
sort()	调整各元素的位置，使之按非降序排列	向量
find(e)	查找等于e且秩最大的元素	向量
search(e)	查找目标元素e，返回不大于e且秩最大的元素	有序向量
deduplicate()	剔除重复元素	向量
uniquify()	剔除重复元素	有序向量
traverse()	遍历向量并统一处理所有元素，处理方法由函数对象指定	向量

以上向量操作接口，可能有多种具体的实现方式，计算复杂度也不尽相同。而在引入秩的概念并将外部接口与内部实现分离之后，无论采用何种具体的方式，符合统一外部接口规范的任一实现均可直接地相互调用和集成。

2.2.2 操作实例

按照表2.1定义的ADT接口，表2.2给出了一个整数向量从被创建开始，通过ADT接口依次实施一系列操作的过程。请留意观察，向量内部各元素秩的逐步变化过程。

表2.2 向量操作实例

操作	输出	向量组成（自左向右）	操作	输出	向量组成（自左向右）
初始化			disordered()	3	4 3 7 4 9 6
insert(0, 9)		9	find(9)	4	4 7 4 9 6
insert(0, 4)		4 9	find(5)	-1	4 3 7 4 9 6
insert(1, 5)		4 5 9	sort()		4 4 6 7 9
put(1, 2)		4 2 9	disordered()	0	3 4 4 6 7 9
get(2)	9	4 2 9	search(1)	-1	3 4 4 6 7 9
insert(3, 6)		4 2 9 6	search(4)	2	3 4 4 6 7 9
insert(1, 7)		4 7 2 9 6	search(8)	4	3 4 4 6 7 9
remove(2)	2	4 7 9 6	search(9)	5	3 4 4 6 7 9
insert(1, 3)		4 3 7 9 6	search(10)	5	3 4 4 6 7 9
insert(3, 4)		4 3 7 4 9 6	uniquify()		3 4 6 7 9
size()	6	4 3 7 4 9 6	search(9)	4	3 4 6 7 9

2.2.3 Vector模板类

按照表2.1确定的向量ADT接口，可定义Vector模板类如代码2.1所示。

```
1  typedef int Rank; //秩
2  #define DEFAULT_CAPACITY  3 //默认的初始容量（实际应用中可设置为更大）
3
4  template <typename T> class Vector { //向量模板类
5  protected:
6     Rank _size; int _capacity;  T* _elem; //规模、容量、数据区
7     void copyFrom ( T const* A, Rank lo, Rank hi ); //复制数组区间A[lo, hi)
8     void expand(); //空间不足时扩容
9     void shrink(); //装填因子过小时压缩
10    bool bubble ( Rank lo, Rank hi ); //扫描交换
11    void bubbleSort ( Rank lo, Rank hi ); //起泡排序算法
12    Rank max ( Rank lo, Rank hi ); //选取最大元素
13    void selectionSort ( Rank lo, Rank hi ); //选择排序算法
14    void merge ( Rank lo, Rank mi, Rank hi ); //归并算法
15    void mergeSort ( Rank lo, Rank hi ); //归并排序算法
16    Rank partition ( Rank lo, Rank hi ); //轴点构造算法
17    void quickSort ( Rank lo, Rank hi ); //快速排序算法
18    void heapSort ( Rank lo, Rank hi ); //堆排序（稍后结合完全堆讲解）
19 public:
```

```
20 // 构造函数
21    Vector ( int c = DEFAULT_CAPACITY, int s = 0, T v = 0 ) //容量为c、规模为s、所有元素初始为v
22    { _elem = new T[_capacity = c]; for ( _size = 0; _size < s; _elem[_size++] = v ); } //s<=c
23    Vector ( T const* A, Rank n ) { copyFrom ( A, 0, n ); } //数组整体复制
24    Vector ( T const* A, Rank lo, Rank hi ) { copyFrom ( A, lo, hi ); } //区间
25    Vector ( Vector<T> const& V ) { copyFrom ( V._elem, 0, V._size ); } //向量整体复制
26    Vector ( Vector<T> const& V, Rank lo, Rank hi ) { copyFrom ( V._elem, lo, hi ); } //区间
27 // 析构函数
28    ~Vector() { delete [] _elem; } //释放内部空间
29 // 只读访问接口
30    Rank size() const { return _size; } //规模
31    bool empty() const { return !_size; } //判空
32    int disordered() const; //判断向量是否已排序
33    Rank find ( T const& e ) const { return find ( e, 0, _size ); } //无序向量整体查找
34    Rank find ( T const& e, Rank lo, Rank hi ) const; //无序向量区间查找
35    Rank search ( T const& e ) const //有序向量整体查找
36    { return ( 0 >= _size ) ? -1 : search ( e, 0, _size ); }
37    Rank search ( T const& e, Rank lo, Rank hi ) const; //有序向量区间查找
38 // 可写访问接口
39    T& operator[] ( Rank r ) const; //重载下标操作符，可以类似于数组形式引用各元素
40    Vector<T> & operator= ( Vector<T> const& ); //重载赋值操作符，以便直接克隆向量
41    T remove ( Rank r ); //删除秩为r的元素
42    int remove ( Rank lo, Rank hi ); //删除秩在区间[lo, hi)之内的元素
43    Rank insert ( Rank r, T const& e ); //插入元素
44    Rank insert ( T const& e ) { return insert ( _size, e ); } //默认作为末元素插入
45    void sort ( Rank lo, Rank hi ); //对[lo, hi)排序
46    void sort() { sort ( 0, _size ); } //整体排序
47    void unsort ( Rank lo, Rank hi ); //对[lo, hi)置乱
48    void unsort() { unsort ( 0, _size ); } //整体置乱
49    int deduplicate(); //无序去重
50    int uniquify(); //有序去重
51 // 遍历
52    void traverse ( void (* ) ( T& ) ); //遍历（使用函数指针，只读或局部性修改）
53    template <typename VST> void traverse ( VST& ); //遍历（使用函数对象，可全局性修改）
54 }; //Vector
```

代码2.1 向量模板类Vector

这里通过模板参数T，指定向量元素的类型。于是，以Vector<int>或Vector<float>之类的形式，可便捷地引入存放整数或浮点数的向量；而以Vector<Vector<char>>之类的形式，则可直接定义存放字符的二维向量等。这一技巧有利于提高数据结构选用的灵活性和运行效率，并减少出错，因此将在本书中频繁使用。

在表2.1所列基本操作接口的基础上，这里还扩充了一些接口。比如，基于size()直接实现

的判空接口empty()，以及区间删除接口remove(lo, hi)、区间查找接口find(e, lo, hi)等。它们多为上述基本接口的扩展或变型，可使代码更为简洁易读，并提高计算效率。

这里还提供了sort()接口，以将向量转化为有序向量。为此可有多种排序算法供选用，本章及后续章节，将陆续介绍它们的原理、实现并分析其效率。排序之后，向量的很多操作都可更加高效地完成，其中最基本和最常用的莫过于查找。因此，这里还针对有序向量提供了search()接口，并将详细介绍若干种相关的算法。为便于对sort()算法的测试，这里还设有一个unsort()接口，以将向量随机置乱。在讨论这些接口之前，我们首先介绍基本接口的实现。

§2.3 构造与析构

由代码2.1可见，向量结构在内部维护一个元素类型为T的私有数组_elem[]：其容量由私有变量_capacity指示；有效元素的数量（即向量当前的实际规模），则由_size指示。此外还进一步地约定，在向量元素的秩、数组单元的逻辑编号以及物理地址之间，具有如下对应关系：

向量中秩为r的元素，对应于内部数组中的_elem[r]，其物理地址为_elem + r

因此，向量对象的构造与析构，将主要围绕这些私有变量和数据区的初始化与销毁展开。

2.3.1 默认构造方法

与所有的对象一样，向量在使用之前也需首先被系统创建——借助构造函数（constructor）做初始化（initialization）。由代码2.1可见，这里为向量重载了多个构造函数。

其中默认的构造方法是，首先根据创建者指定的初始容量，向系统申请空间，以创建内部私有数组_elem[]；若容量未明确指定，则使用默认值DEFAULT_CAPACITY。接下来，鉴于初生的向量尚不包含任何元素，故将指示规模的变量_size初始化为0。

整个过程顺序进行，没有任何迭代，故若忽略用于分配数组空间的时间，共需常数时间。

2.3.2 基于复制的构造方法

向量的另一典型创建方式，是以某个已有的向量或数组为蓝本，进行（局部或整体的）克隆。代码2.1中虽为此功能重载了多个接口，但无论是已封装的向量或未封装的数组，无论是整体还是区间，在入口参数合法的前提下，都可归于如代码2.2所示的统一的copyFrom()方法：

```
1 template <typename T> //元素类型
2 void Vector<T>::copyFrom ( T const* A, Rank lo, Rank hi ) { //以数组区间A[lo, hi)为蓝本复制向量
3    _elem = new T[_capacity = 2 * ( hi - lo ) ]; _size = 0; //分配空间，规模清零
4    while ( lo < hi ) //A[lo, hi)内的元素逐一
5       _elem[_size++] = A[lo++]; //复制至_elem[0, hi - lo)
6 }
```

代码2.2 基于复制的向量构造器

copyFrom()首先根据待复制区间的边界，换算出新向量的初始规模；再以双倍的容量，为内部数组_elem[]申请空间。最后通过一趟迭代，完成区间A[lo, hi)内各元素的顺次复制。

若忽略开辟新空间所需的时间，运行时间应正比于区间宽度，即$\mathcal{O}$(hi - lo) = $\mathcal{O}$(_size)。

需强调的是，由于向量内部含有动态分配的空间，默认的运算符"="不足以支持向量之间的直接赋值。例如，6.3节将以二维向量形式实现图邻接表，其主向量中的每一元素本身都是一维向量，故通过默认赋值运算符，并不能复制向量内部的数据区。

为适应此类赋值操作的需求，可如代码2.3所示，重载向量的赋值运算符。

```
1 template <typename T> Vector<T>& Vector<T>::operator= ( Vector<T> const& V ) { //重载
2    if ( _elem ) delete [] _elem; //释放原有内容
3    copyFrom ( V._elem, 0, V.size() ); //整体复制
4    return *this; //返回当前对象的引用，以便链式赋值
5 }
```

代码2.3 重载向量赋值操作符

2.3.3 析构方法

与所有对象一样，不再需要的向量，应借助析构函数（destructor）及时清理（cleanup），以释放其占用的系统资源。与构造函数不同，同一对象只能有一个析构函数，不得重载。

向量对象的析构过程，如代码2.1中的方法~Vector()所示：只需释放用于存放元素的内部数组_elem[]，将其占用的空间交还操作系统。_capacity和_size之类的内部变量无需做任何处理，它们将作为向量对象自身的一部分被系统回收，此后既无需也无法被引用。

若不计系统用于空间回收的时间，整个析构过程只需$\mathcal{O}(1)$时间。

同样地，向量中的元素可能不是程序语言直接支持的基本类型。比如，可能是指向动态分配对象的指针或引用，故在向量析构之前应该提前释放对应的空间。出于简化的考虑，这里约定并遵照“谁申请谁释放”的原则。究竟应释放掉向量各元素所指的对象，还是需要保留这些对象，以便通过其它指针继续引用它们，应由上层调用者负责确定。

§2.4 动态空间管理

2.4.1 静态空间管理

内部数组所占物理空间的容量，若在向量的生命期内不允许调整，则称作静态空间管理策略。很遗憾，该策略的空间效率难以保证。一方面，既然容量固定，总有可能在此后的某一时刻，无法加入更多的新元素——即导致所谓的上溢（overflow）。例如，若使用向量来记录网络访问日志，则由于插入操作远多于删除操作，必然频繁溢出。注意，造成此类溢出的原因，并非系统不能提供更多的空间。另一方面反过来，即便愿意为降低这种风险而预留出部分空间，也很难在程序执行之前，明确界定一个合理的预留量。以上述copyFrom()方法为例，即便将容量取作初始规模的两倍，也只能保证在此后足够长的一段时间内（而并非永远）不致溢出。

向量实际规模与其内部数组容量的比值（即_size/_capacity），亦称作装填因子（load factor），它是衡量空间利用率的重要指标。从这一角度，上述难题可归纳为：

如何才能保证向量的装填因子既不致于超过1，也不致于太接近于0？

为此，需要改用动态空间管理策略。其中一种有效的方法，即使用所谓的可扩充向量。

2.4.2 可扩充向量

经过一段时间的生长，每当身体无法继续为其外壳所容纳，蝉就会蜕去外壳，同时换上一身更大的外壳。扩充向量（extendable vector）的原理，与之相仿。若内部数组仍有空余，则操作可照常执行。每经一次插入（删除），可用空间都会减少（增加）一个单元（图2.1(a)）。一旦可用空间耗尽（图(b)），就动态地扩大内部数组的容量。这里的难点及关键在于：

如何实现扩容？新的容量取作多少才算适宜？

首先解决前一问题。直接在原有物理空间的基础上追加空间？这并不现实。数组特有的定址方式要求，物理空间必须地址连续，而我们却无法保证，其尾部总是预留了足够空间可供拓展。

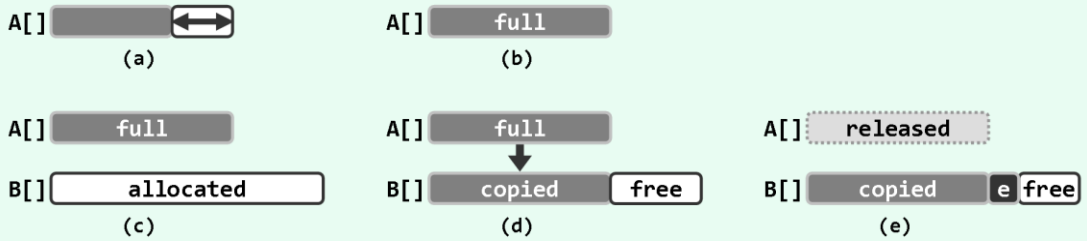

图2.1 可扩充向量的溢出处理

一种可行的方法，如图2.1(c~e)所示。我们需要另行申请一个容量更大的数组B[]（图(c)），并将原数组中的成员集体搬迁至新的空间（图(d)），此后方可顺利地插入新元素e而不致溢出（图(e)）。当然，原数组所占的空间，需要及时释放并归还操作系统。

2.4.3 扩容

基于以上策略的扩容算法expand()，可实现如代码2.4所示。

```
1 template <typename T> void Vector<T>::expand() { //向量空间不足时扩容
2    if ( _size < _capacity ) return; //尚未满员时，不必扩容
3    if ( _capacity < DEFAULT_CAPACITY ) _capacity = DEFAULT_CAPACITY; //不低于最小容量
4    T* oldElem = _elem;  _elem = new T[_capacity <<= 1]; //容量加倍
5    for ( int i = 0; i < _size; i++ )
6       _elem[i] = oldElem[i]; //复制原向量内容（T为基本类型，或已重载赋值操作符'='）
7    delete [] oldElem; //释放原空间
8 }
```

代码2.4 向量内部数组动态扩容算法expand()

实际上，在调用insert()接口插入新元素之前，都要先调用该算法，检查内部数组的可用容量。一旦当前数据区已满（_size == _capacity），则将原数组替换为一个更大的数组。

请注意，新数组的地址由操作系统分配，与原数据区没有直接的关系。这种情况下，若直接引用数组，往往会导致共同指向原数组的其它指针失效，成为野指针（wild pointer）；而经封装为向量之后，即可继续准确地引用各元素，从而有效地避免野指针的风险。

这里的关键在于，新数组的容量总是取作原数组的两倍——这正是上述后一问题的答案。

2.4.4 分摊分析

■ 时间代价

与常规数组实现相比，可扩充向量更加灵活：只要系统尚有可用空间，其规模将不再受限于初始容量。不过，这并非没有代价——每次扩容，元素的搬迁都需要花费额外的时间。

准确地，每一次由n到2n的扩容，都需要花费$O(2n) = O(n)$时间——这也是最坏情况下，单次插入操作所需的时间。表面看来，这一扩容策略似乎效率很低，但这不过是一种错觉。

请注意，按照此处的约定，每花费$O(n)$时间实施一次扩容，数组的容量都会加倍。这就意味着，至少要再经过n次插入操作，才会因为可能溢出而再次扩容。也就是说，随着向量规模的不断扩大，在执行插入操作之前需要进行扩容的概率，也将迅速降低。故就某种平均意义而言，用于扩容的时间成本不至很高。以下不妨就此做一严格的分析。

■ 分摊复杂度

这里，不妨考查对可扩充向量的足够多次连续操作，并将其间所消耗的时间，分摊至所有的操作。如此分摊平均至单次操作的时间成本，称作分摊运行时间（amortized running time）。

请注意，这一指标与平均运行时间（average running time）有着本质的区别（习题[2-1]）。后者是按照某种假定的概率分布，对各种情况下所需执行时间的加权平均，故亦称作期望运行时间（expected running time）。而前者则要求，参与分摊的操作必须构成和来自一个真实可行的操作序列，而且该序列还必须足够地长。

相对而言，分摊复杂度可以针对计算成本和效率，做出更为客观而准确的估计。比如在这里，在任何一个可扩充向量的生命期内，在任何足够长的连续操作序列中，以任何固定间隔连续出现上述最坏情况的概率均为0，故常规的平均复杂度根本不具任何参考意义。作为评定算法性能的一种重要尺度，分摊分析（amortized analysis）的相关方法与技巧将在后续章节陆续介绍。

■ $O(1)$分摊时间

以可扩充向量为例，可以考查对该结构的连续n次（查询、插入或删除等）操作，将所有操作中用于内部数组扩容的时间累计起来，然后除以n。只要n足够大，这一平均时间就是用于扩容处理的分摊时间成本。以下我们将看到，即便排除查询和删除操作而仅考查插入操作，在可扩充向量单次操作中，用于扩容处理的分摊时间成本也不过$O(1)$。

假定数组的初始容量为某一常数N。既然是估计复杂度的上界，故不妨设向量的初始规模也为N——即将溢出。另外不难看出，除插入操作外，向量其余的接口操作既不会直接导致溢出，也不会增加此后溢出的可能性，因此不妨考查最坏的情况，假设在此后需要连续地进行n次insert()操作，n >> N。首先定义如下函数：

```
size(n)  =  连续插入n个元素后向量的规模
capacity(n)  =  连续插入n个元素后数组的容量
T(n) = 为连续插入n个元素而花费于扩容的时间
```

其中，向量规模从N开始随着操作的进程逐步递增，故有：

$$size(n) = N + n$$

既然不致溢出，故装填因子绝不会超过100%。同时，这里的扩容采用了“懒惰”策略——只有在的确即将发生溢出时，才不得不将容量加倍——因此装填因子也始终不低于50%。

概括起来，始终应有：

$$size(n) \leq capacity(n) < 2 \cdot size(n)$$

考虑到N为常数，故有：

$$capacity(n) = \Theta(size(n)) = \Theta(n)$$

容量以2为比例按指数速度增长，在容量达到capacity(n)之前，共做过$\Theta(\log_2 n)$次扩容，每次扩容所需时间线性正比于当时的容量（或规模），且同样以2为比例按指数速度增长。因此，消耗于扩容的时间累计不过：

$$T(n) = 2N + 4N + 8N + ... + capacity(n) < 2 \cdot capacity(n) = \Theta(n)$$

将其分摊到其间的连续n次操作，单次操作所需的分摊运行时间应为$\mathcal{O}(1)$。

■ 其它扩容策略

以上分析确凿地说明，基于加倍策略的动态扩充数组不仅可行，而且就分摊复杂度而言效率也足以令人满意。当然，并非任何扩容策略都能保证如此高的效率。比如，早期可扩充向量多采用另一策略：一旦有必要，则追加固定数目的单元。实际上，无论采用的固定常数多大，在最坏情况下，此类数组单次操作的分摊时间复杂度都将高达$\Omega(n)$（习题[2-3]）。

2.4.5 缩容

导致低效率的另一情况是，向量的实际规模可能远远小于内部数组的容量。比如在连续的一系列操作过程中，若删除操作远多于插入操作，则装填因子极有可能远远小于100%，甚至非常接近于0。当装填因子低于某一阈值时，我们称数组发生了下溢（underflow）。

尽管下溢不属于必须解决的问题，但在格外关注空间利用率的场合，发生下溢时也有必要适当缩减内部数组容量。代码2.5给出了一个动态缩容shrink()算法：

```
1 template <typename T> void Vector<T>::shrink() { //装填因子过小时压缩向量所占空间
2    if ( _capacity < DEFAULT_CAPACITY << 1 ) return; //不致收缩到DEFAULT_CAPACITY以下
3    if ( _size << 2 > _capacity ) return; //以25%为界
4    T* oldElem = _elem;  _elem = new T[_capacity >>= 1]; //容量减半
5    for ( int i = 0; i < _size; i++ ) _elem[i] = oldElem[i]; //复制原向量内容
6    delete [] oldElem; //释放原空间
7 }
```

代码2.5 向量内部功能shrink()

可见，每次删除操作之后，一旦空间利用率已降至某一阈值以下，该算法随即申请一个容量减半的新数组，将原数组中的元素逐一搬迁至其中，最后将原数组所占空间交还操作系统。这里以25%作为装填因子的下限，但在实际应用中，为避免出现频繁交替扩容和缩容的情况，可以选用更低的阈值，甚至取作0（相当于禁止缩容）。

与expand()操作类似，尽管单次shrink()操作需要线性量级的时间，但其分摊复杂度亦为$\mathcal{O}(1)$（习题[2-4]）。实际上shrink()过程等效于expand()的逆过程，这两个算法相互配合，在不致实质地增加接口操作复杂度的前提下，保证了向量内部空间的高效利用。当然，就单次扩容或缩容操作而言，所需时间的确会高达$\Omega(n)$，因此在对单次操作的执行速度极其敏感的应用场合以上策略并不适用，其中缩容操作甚至可以完全不予考虑。

§2.5 常规向量

2.5.1 直接引用元素

与数组直接通过下标访问元素的方式（形如“A[i]”）相比，向量ADT所设置的get()和put()接口都显得不甚自然。毕竟，前一访问方式不仅更为我们所熟悉，同时也更加直观和便捷。那么，在经过封装之后，对向量元素的访问可否沿用数组的方式呢？答案是肯定的。

解决的方法之一就是重载操作符“[]”，具体实现如代码2.6所示。

```
1 template <typename T> T& Vector<T>::operator[] ( Rank r ) const //重载下标操作符
2 { return _elem[r]; } // assert: 0 <= r < _size
```

代码2.6 重载向量操作符[]

2.5.2 置乱器

■ 置乱算法

可见，经重载后操作符“[]”返回的是对数组元素的引用，这就意味着它既可以取代get()操作（通常作为赋值表达式的右值），也可以取代set()操作（通常作为左值）。例如，采用这种形式，可以简明清晰地描述和实现如代码2.7所示的向量置乱算法。

```
1 template <typename T> void permute ( Vector<T>& V ) { //随机置乱向量，使各元素等概率出现于各位置
2    for ( int i = V.size(); i > 0; i-- ) //自后向前
3       swap ( V[i - 1], V[rand() % i] ); //V[i - 1]与V[0, i)中某一随机元素交换
4 }
```

代码2.7 向量整体置乱算法permute()

该算法从待置乱区间的末元素开始，逆序地向前逐一处理各元素。如图2.2(a)所示，对每一个当前元素V[i - 1]，先通过调用rand()函数在[0, i)之间等概率地随机选取一个元素，再令二者互换位置。注意，这里的交换操作swap()，隐含了三次基于重载操作符“[]”的赋值。

于是如图(b)所示，每经过一步这样的迭代，置乱区间都会向前拓展一个单元。因此经过$O(n)$步迭代之后，即实现了整个向量的置乱。

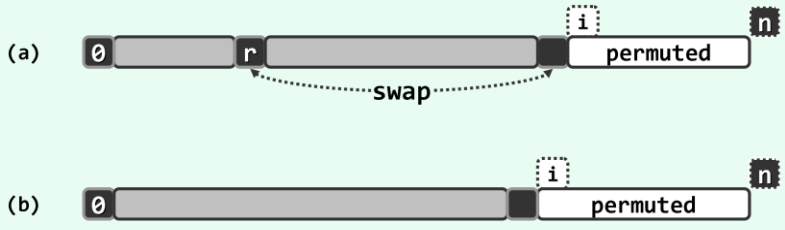

图2.2 向量整体置乱算法permute()的迭代过程

在软件测试、仿真模拟等应用中，随机向量的生成都是一项至关重要的基本操作，直接影响到测试的覆盖面或仿真的真实性。从理论上说，使用这里的算法permute()，不仅可以枚举出同一向量所有可能的排列，而且能够保证生成各种排列的概率均等（习题[2-6]）。

■ 区间置乱接口

为便于对各种向量算法的测试与比较，这里不妨将以上permute()算法封装至向量ADT中，并如代码2.8所示，对外提供向量的置乱操作接口Vector::unsort()。

```
1 template <typename T> void Vector<T>::unsort ( Rank lo, Rank hi ) { //等概率随机置乱区间[lo, hi)
2    T* V = _elem + lo; //将子向量_elem[lo, hi)视作另一向量V[0, hi - lo)
3    for ( Rank i = hi - lo; i > 0; i-- ) //自后向前
4       swap ( V[i - 1], V[rand() % i] ); //将V[i - 1]与V[0, i)中某一元素随机交换
5 }
```

代码2.8 向量区间置乱接口unsort()

通过该接口，可以均匀地置乱任一向量区间[lo, hi)内的元素，故通用性有所提高。可见，只要将该区间等效地视作另一向量V，即可从形式上完整地套用以上permute()算法的流程。

尽管如此，还请特别留意代码2.7与代码2.8的细微差异：后者是通过下标，直接访问内部数组的元素；而前者则是借助重载的操作符“[]”，通过秩间接地访问向量的元素。

2.5.3 判等器与比较器

从算法的角度来看，“判断两个对象是否相等”与“判断两个对象的相对大小”都是至关重要的操作，它们直接控制着算法执行的分支方向，因此也是算法的“灵魂”所在。在本书中为了以示区别，前者多称作“比对”操作，后者多称作“比较”操作。当然，这两种操作之间既有联系也有区别，不能相互替代。比如，有些对象只能比对但不能比较；反之，支持比较的对象未必支持比对。不过，出于简化的考虑，在很多场合并不需要严格地将二者区分开来。

算法实现的简洁性与通用性，在很大程度上体现于：针对整数等特定数据类型的某种实现，可否推广至可比较或可比对的任何数据类型，而不必关心如何定义以及判定其大小或相等关系。若能如此，我们就可以将比对和比较操作的具体实现剥离出来，直接讨论算法流程本身。

为此，通常可以采用两种方法。其一，将比对操作和比较操作分别封装成通用的判等器和比较器。其二，在定义对应的数据类型时，通过重载"<"和"=="之类的操作符，给出大小和相等关系的具体定义及其判别方法。本书将主要采用后一方式。为节省篇幅，这里只给出涉及到的比较和判等操作符，读者可以根据实际需要，参照给出的代码加以扩充。

```
1 template <typename T> static bool lt ( T* a, T* b ) { return lt ( *a, *b ); } //less than
2 template <typename T> static bool lt ( T& a, T& b ) { return a < b; } //less than
3 template <typename T> static bool eq ( T* a, T* b ) { return eq ( *a, *b ); } //equal
4 template <typename T> static bool eq ( T& a, T& b ) { return a == b; } //equal
```

代码2.9 重载比较器以便比较对象指针

在一些复杂的数据结构中，内部元素本身的类型可能就是指向其它对象的指针；而从外部更多关注的，则往往是其所指对象的大小。若不加处理而直接根据指针的数值（即被指对象的物理地址）进行比较，则所得结果将毫无意义。

为此，这里不妨通过如代码2.9所示的机制，将这种情况与一般情况予以区分，并且约定在这种情况下，统一按照被指对象的大小做出判断。

2.5.4 无序查找

■ 判等器

代码2.1中Vector::find(e)接口，功能语义为“查找与数据对象e相等的元素”。这同时也暗示着，向量元素可通过相互比对判等——比如，元素类型T或为基本类型，或已重载操作符“==”或“!=”。这类仅支持比对，但未必支持比较的向量，称作无序向量（unsorted vector）。

■ 顺序查找

在无序向量中查找任意指定元素e时，因为没有更多的信息可以借助，故在最坏情况下——比如向量中并不包含e时——只有在访遍所有元素之后，才能得出查找结论。

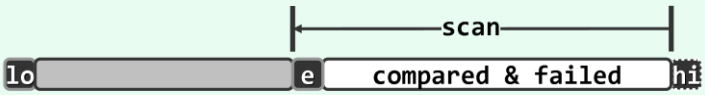

图2.3 无序向量的顺序查找

因此不妨如图2.3所示，从末元素起自后向前，逐一取出各个元素并与目标元素e进行比对，直至发现与之相等者（查找成功），或者直至检查过所有元素之后仍未找到相等者（查找失败）。这种依次逐个比对的查找方式，称作顺序查找（sequential search）。

■ 实现

针对向量的整体或区间，代码2.1分别定义了一个顺序查找操作的入口，其中前者作为特例，可直接通过调用后者而实现。因此，只需如代码2.10所示，实现针对向量区间的查找算法。

```
1 template <typename T> //无序向量的顺序查找：返回最后一个元素e的位置；失败时，返回lo - 1
2 Rank Vector<T>::find ( T const& e, Rank lo, Rank hi ) const { //assert: 0 <= lo < hi <= _size
3    while ( ( lo < hi-- ) && ( e != _elem[hi] ) ); //从后向前，顺序查找
4    return hi; //若hi < lo，则意味着失败；否则hi即命中元素的秩
5 }
```

代码2.10 无序向量元素查找接口find()

其中若干细微之处，需要体会。比如，当同时有多个命中元素时，本书统一约定返回其中秩最大者——稍后介绍的查找接口find()亦是如此——故这里采用了自后向前的查找次序。如此，一旦命中即可立即返回，从而省略掉不必要的比对。另外，查找失败时约定统一返回-1。这不仅简化了对查找失败情况的判别，同时也使此时的返回结果更加易于理解——只要假想着在秩为-1处植入一个与任何对象都相等的哨兵元素，则返回该元素的秩当且仅当查找失败。

最后还有一处需要留意。while循环的控制逻辑由两部分组成，首先判断是否已抵达通配符，再判断当前元素与目标元素是否相等。得益于C/C++语言中逻辑表达式的短路求值特性，在前一判断非真后循环会立即终止，而不致因试图引用已越界的秩（-1）而出错。

■ 复杂度

最坏情况下，查找终止于首元素_elem[lo]，运行时间为$\mathcal{O}$(hi - lo) = $\mathcal{O}$(n)。最好情况下，查找命中于末元素_elem[hi - 1]，仅需$\mathcal{O}$(1)时间。对于规模相同、内部组成不同的输入，渐进运行时间却有本质区别，故此类算法也称作输入敏感的（input sensitive）算法。

2.5.5 插入

■ 实现

按照代码2.1的ADT定义，插入操作insert(r, e)负责将任意给定的元素e插到任意指定的秩为r的单元。整个操作的过程，可具体实现如代码2.11所示。

```
1 template <typename T> //将e作为秩为r元素插入
2 Rank Vector<T>::insert ( Rank r, T const& e ) { //assert: 0 <= r <= size
3    expand(); //若有必要，扩容
4    for ( int i = _size; i > r; i-- ) _elem[i] = _elem[i-1]; //自后向前，后继元素顺次后移一个单元
5    _elem[r] = e; _size++; //置入新元素并更新容量
6    return r; //返回秩
7 }
```

代码2.11 向量元素插入接口insert()

如前所述，插入之前必须首先调用expand()算法，核对是否即将溢出；若有必要（图2.4(a)），则加倍扩容（图2.4(b)）。

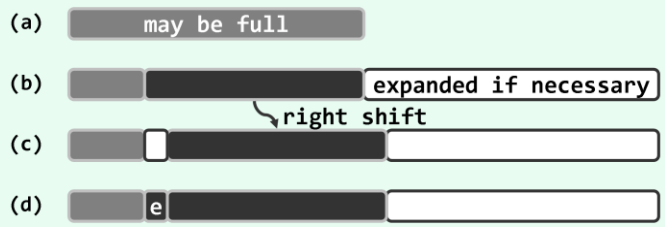

图2.4 向量元素插入操作insert(r, e)的过程

为保证数组元素物理地址连续的特性，随后需要将后缀_elem[r, _size)（如果非空）整体后移一个单元（图(c)）。这些后继元素自后向前的搬迁次序不能颠倒，否则会因元素被覆盖而造成数据丢失。在单元_elem[r]腾出之后，方可将待插入对象e置入其中（图(d)）。

■ 复杂度

时间主要消耗于后继元素的后移，线性正比于后缀的长度，故总体为O(_size - r + 1)。

可见，新插入元素越靠后（前）所需时间越短（长）。特别地，r取最大值_size时为最好情况，只需O(1)时间；r取最小值0时为最坏情况，需要O(_size)时间。一般地，若插入位置等概率分布，则平均运行时间为O(_size) = O(n)（习题[2-9]），线性正比于向量的实际规模。

2.5.6 删除

删除操作重载有两个接口，remove(lo, hi)用以删除区间[lo, hi)内的元素，而remove(r)用以删除秩为r的单个元素。乍看起来，利用后者即可实现前者：令r从hi - 1到lo递减，反复调用remove(r)。不幸的是，这一思路似是而非。

因数组中元素的地址必须连续，故每删除一个元素，所有后继元素都需向前移动一个单元。若后继元素共有m = _size - hi个，则对remove(r)的每次调用都需移动m次；对于整个区间，元素移动的次数累计将达到m*(hi - lo)，为后缀长度和待删除区间宽度的乘积。

实际可行的思路恰好相反，应将单元素删除视作区间删除的特例，并基于后者来实现前者。

稍后就会看到，如此可将移动操作的总次数控制在$O(m)$以内，而与待删除区间的宽度无关。

■ 区间删除：remove(lo, hi)

向量区间删除接口remove(lo, hi)，可实现如代码2.12所示。

```
1 template <typename T> int Vector<T>::remove ( Rank lo, Rank hi ) { //删除区间[lo, hi)
2    if ( lo == hi ) return 0; //出于效率考虑，单独处理退化情况，比如remove(0, 0)
3    while ( hi < _size ) _elem[lo++] = _elem[hi++]; //[hi, _size)顺次前移hi - lo个单元
4    _size = lo; //更新规模，直接丢弃尾部[lo, _size = hi)区间
5    shrink(); //若有必要，则缩容
6    return hi - lo; //返回被删除元素的数目
7 }
```

代码2.12 向量区间删除接口remove(lo, hi)

设[lo, hi)为向量（图2.5(a)）的合法区间（图(b)），则其后缀[hi, n)需整体前移hi - lo个单元（图(c)）。与插入算法同理，这里后继元素自前向后的移动次序也不能颠倒（习题[2-10]）。

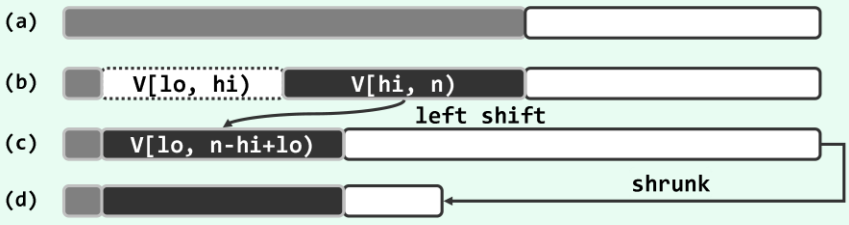

图2.5 向量区间删除操作remove(lo, hi)的过程

向量规模更新为_size - hi + lo后，还要调用shrink()，若有必要则做缩容处理（图(d)）。

■ 单元素删除remove(r)

利用以上remove(lo, hi)通用接口，通过重载即可实现如下另一同名接口remove(r)。

```
1 template <typename T> T Vector<T>::remove ( Rank r ) { //删除向量中秩为r的元素，0 <= r < size
2    T e = _elem[r]; //备份被删除元素
3    remove ( r, r + 1 ); //调用区间删除算法，等效于对区间[r, r + 1)的删除
4    return e; //返回被删除元素
5 }
```

代码2.13 向量单元素删除接口remove()

■ 复杂度

remove(lo, hi)的计算成本，主要消耗于后续元素的前移，线性正比于后缀的长度，总体不过$O(m + 1) = O(_size - hi + 1)$。

这与此前的预期完全吻合：区间删除操作所需的时间，应该仅取决于后继元素的数目，而与被删除区间本身的宽度无关。特别地，基于该接口实现的单元素删除接口remove(r)需耗时$O(_size - r)$。也就是说，被删除元素在向量中的位置越靠后（前）所需时间越短（长），最好为$O(1)$，最坏为$O(n) = O(_size)$。

■ 错误及意外处理

请注意，上述操作接口对输入都有一定的限制和约定。其中指定的待删除区间，必须落在合

法范围[0, _size)之内，为此输入参数必须满足0 ≤ lo ≤ hi ≤ _size。

一般地，输入参数超出接口所约定合法范围的此类问题，都属于典型的错误（error）或意外（exception）。除了以注释的形式加以说明，还应该尽可能对此类情况做更为周全的处理。

尽管如此，本书还是沿用了相对简化的处置方式，将入口参数合法性检查的责任统一交由上层调用例程，这也是出于对本书的重点——算法效率、讲解重点以及叙述简洁——的优先考虑。当然，在充分掌握了本书的内容之后，读者不妨再按照软件工程的规范，就此做进一步的完善。

2.5.7 唯一化

很多应用中，在进一步处理之前都要求数据元素互异。以网络搜索引擎为例，多个计算节点各自获得的局部搜索结果，需首先剔除其中重复的项目，方可合并为一份完整的报告。类似地，所谓向量的唯一化处理，就是剔除其中的重复元素，即表2.1所列deduplicate()接口的功能。

■ **实现**

视向量是否有序，该功能有两种实现方式，以下首先介绍针对无序向量的唯一化算法。

```
1 template <typename T> int Vector<T>::deduplicate() { //删除无序向量中重复元素（高效版）
2    int oldSize = _size; //记录原规模
3    Rank i = 1; //从_elem[1]开始
4    while ( i < _size ) //自前向后逐一考查各元素_elem[i]
5       ( find ( _elem[i], 0, i ) < 0 ) ? //在其前缀中寻找与之雷同者（至多一个）
6       i++ : remove ( i ); //若无雷同则继续考查其后继，否则删除雷同者
7    return oldSize - _size; //向量规模变化量，即被删除元素总数
8 }
```

代码2.14 无序向量清除重复元素接口deduplicate()

如代码2.14所示，该算法自前向后逐一考查各元素_elem[i]，并通过调用find()接口，在其前缀中寻找与之雷同者。若找到，则随即删除；否则，转而考查当前元素的后继。

■ **正确性**

算法的正确性由以下不变性保证：

在while循环中，在当前元素的前缀_elem[0, i)内，所有元素彼此互异

初次进入循环时i = 1，只有唯一的前驱_elem[0]，故不变性自然满足。

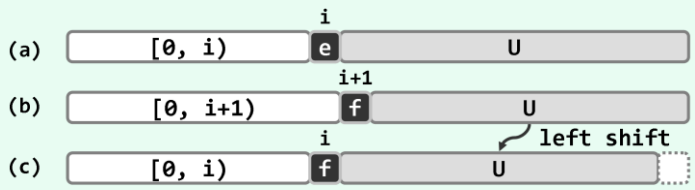

图2.6 无序向量deduplicate()算法原理

一般地如图2.6(a)所示，假设在转至元素e = _elem[i]之前不变性一直成立。于是，经过针对该元素的一步迭代之后，无非两种结果：

1）若元素e的前缀_elem[0, i)中不含与之雷同的元素，则如图(b)，在做过i++之后，新的前缀_elem[0, i)将继续满足不变性，而且其规模增加一个单位。

2）反之，若含存在与e雷同的元素，则由此前一直满足的不变性可知，这样的雷同元素不超过一个。因此如图(c)，在删除e之后，前缀_elem[0, i)依然保持不变性。

■ 复杂度

由图2.6(a)和(b)也可看出该算法过程所具有的单调性：

随着循环的不断进行，当前元素的后继持续地严格减少

因此，经过n - 2步迭代之后该算法必然终止。

这里所需的时间，主要消耗于find()和remove()两个接口。根据2.5.4节的结论，前一部分时间应线性正比于查找区间的宽度，即前驱的总数；根据2.5.6节的结论，后一部分时间应线性正比于后继的总数。因此，每步迭代所需时间为$O(n)$，总体复杂度应为$O(n^2)$。

经预排序转换之后，借助2.6.3节将要介绍的相关算法，还可以进一步提高向量唯一化处理的效率（习题[2-12]）。

2.5.8 遍历

■ 功能

在很多算法中，往往需要将向量作为一个整体，对其中所有元素实施某种统一的操作，比如输出向量中的所有元素，或者按照某种运算流程统一修改所有元素的数值（习题[2-13]）。针对此类操作，可为向量专门设置一个遍历接口traverse()。

■ 实现

向量的遍历操作接口，可实现如代码2.15所示。

```
1 template <typename T> void Vector<T>::traverse ( void ( *visit ) ( T& ) ) //借助函数指针机制
2 { for ( int i = 0; i < _size; i++ ) visit ( _elem[i] ); } //遍历向量
3
4 template <typename T> template <typename VST> //元素类型、操作器
5 void Vector<T>::traverse ( VST& visit ) //借助函数对象机制
6 { for ( int i = 0; i < _size; i++ ) visit ( _elem[i] ); } //遍历向量
```

代码2.15 向量遍历接口traverse()

可见，traverse()遍历的过程，实质上就是自前向后地逐一对各元素实施同一基本操作。而具体采用何种操作，可通过两种方式指定。前一种方式借助函数指针*visit()指定某一函数，该函数只有一个参数，其类型为对向量元素的引用，故通过该函数即可直接访问或修改向量元素。另外，也可以函数对象的形式，指定具体的遍历操作。这类对象的操作符“()”经重载之后，在形式上等效于一个函数接口，故此得名。

相比较而言，后一形式的功能更强，适用范围更广。比如，函数对象的形式支持对向量元素的关联修改。也就是说，对各元素的修改不仅可以相互独立地进行，也可以根据某个（些）元素的数值相应地修改另一元素。前一形式虽也可实现这类功能，但要繁琐很多。

■ 实例

在代码2.16中，Increase<T>()即是按函数对象形式指定的基本操作，其功能是将作为参数的引用对象的数值加一（假定元素类型T可直接递增或已重载操作符"++"）。于是可如

increase()函数那样，以此基本操作做遍历即可使向量内所有元素的数值同步加一。

```
1 template <typename T> struct Increase //函数对象：递增一个T类对象
2    {  virtual void operator() ( T& e ) { e++; }  }; //假设T可直接递增或已重载++
3
4 template <typename T> void increase ( Vector<T> & V ) //统一递增向量中的各元素
5 {  V.traverse ( Increase<T>() );  } //以Increase<T>()为基本操作进行遍历
```

代码2.16 基于遍历实现increase()功能

■ **复杂度**

遍历操作本身只包含一层线性的循环迭代，故除了向量规模的因素之外，遍历所需时间应线性正比于所统一指定的基本操作所需的时间。比如在上例中，统一的基本操作Increase<T>()只需常数时间，故这一遍历的总体时间复杂度为$O(n)$。

§2.6 有序向量

若向量S[0, n)中的所有元素不仅按线性次序存放，而且其数值大小也按此次序单调分布，则称作有序向量（sorted vector）。例如，所有学生的学籍记录可按学号构成一个有序向量（学生名单），使用同一跑道的所有航班可按起飞时间构成一个有序向量（航班时刻表），第二十九届奥运会男子跳高决赛中各选手的记录可按最终跳过的高度构成一个（非增）序列（名次表）。与通常的向量一样，有序向量依然不要求元素互异，故通常约定其中的元素自前（左）向后（右）构成一个非降序列，即对任意$0 \le i < j < n$都有$S[i] \le S[j]$。

2.6.1 比较器

当然，除了与无序向量一样需要支持元素之间的“判等”操作，有序向量的定义中实际上还隐含了另一更强的先决条件：各元素之间必须能够比较大小。这一条件构成了有序向量中“次序”概念的基础，否则所谓的“有序”将无从谈起。

多数高级程序语言所提供的基本数据类型都满足上述条件，比如C++语言中的整型、浮点型和字符型等，然而字符串、复数、矢量以及更为复杂的类型，则未必直接提供了某种自然的大小比较规则。采用很多方法，都可以使得大小比较操作对这些复杂数据对象可以明确定义并且可行，比如最常见的就是在内部指定某一（些）可比较的数据项，并由此确立比较的规则。这里沿用2.5.3节的约定，假设复杂数据对象已经重载了"<"和"<="等操作符。

2.6.2 有序性甄别

作为无序向量的特例，有序向量自然可以沿用无序向量的查找算法。然而，得益于元素之间的有序性，有序向量的查找、唯一化等操作都可更快地完成。因此在实施此类操作之前，都有必要先判断当前向量是否已经有序，以便确定是否可采用更为高效的接口。

```
1 template <typename T> int Vector<T>::disordered() const { //返回向量中逆序相邻元素对的总数
2    int n = 0; //计数器
3    for ( int i = 1; i < _size; i++ ) //逐一检查_size - 1对相邻元素
```

```
4       if ( _elem[i - 1] > _elem[i] ) n++; //逆序则计数
5    return n; //向量有序当且仅当n = 0
6 }
```

代码2.17 有序向量甄别算法disordered()

代码2.17即为有序向量的一个甄别算法，其原理与1.1.3节起泡排序算法相同：顺序扫描整个向量，逐一比较每一对相邻元素——向量已经有序，当且仅当它们都是顺序的。

2.6.3 唯一化

相对于无序向量，有序向量中清除重复元素的操作更为重要。正如2.5.7节所指出的，出于效率的考虑，为清除无序向量中的重复元素，一般做法往往是首先将其转化为有序向量。

■ 低效版

```
1 template <typename T> int Vector<T>::uniquify() { //有序向量重复元素剔除算法（低效版）
2    int oldSize = _size; int i = 1; //当前比对元素的秩，起始于首元素
3    while ( i < _size ) //从前向后，逐一比对各对相邻元素
4       _elem[i - 1] == _elem[i] ? remove ( i ) : i++; //若雷同，则删除后者；否则，转至后一元素
5    return oldSize - _size; //向量规模变化量，即被删除元素总数
6 }
```

代码2.18 有序向量uniquify()接口的平凡实现

唯一化算法可实现如代码2.18所示，其正确性基于如下事实：有序向量中的重复元素必然前后紧邻。于是，可以自前向后地逐一检查各对相邻元素：若二者雷同则调用remove()接口删除靠后者，否则转向下一对相邻元素。如此，扫描结束后向量中将不再含有重复元素。

这里的运行时间，主要消耗于while循环，共需迭代_size - 1 = n - 1步。此外，在最坏情况下，每次循环都需执行一次remove()操作，由2.3节的分析结论，其复杂度线性正比于被删除元素的后继元素总数。因此如图2.7所示，当大量甚至所有元素均雷同时，用于所有这些remove()操作的时间总量将高达：

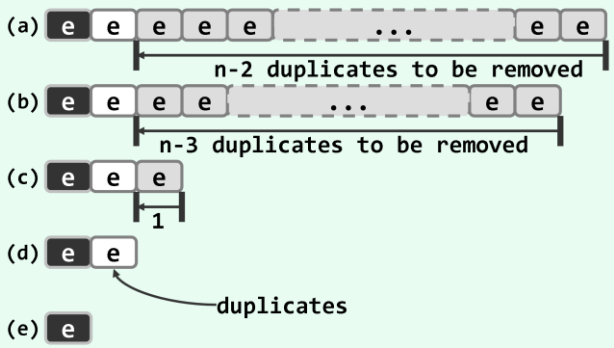

图2.7 低效版uniquify()算法的最坏情况

$$\begin{aligned} & (n - 2) + (n - 3) + \ldots + 2 + 1 \\ = \ & \mathcal{O}(n^2) \end{aligned}$$

这一效率竟与向量未排序时相同，说明该方法未能充分利用此时向量的有序性。

■ 改进思路

稍加分析即不难看出，以上唯一化过程复杂度过高的根源是，在对remove()接口的各次调用中，同一元素可能作为后继元素向前移动多次，且每次仅移动一个单元。

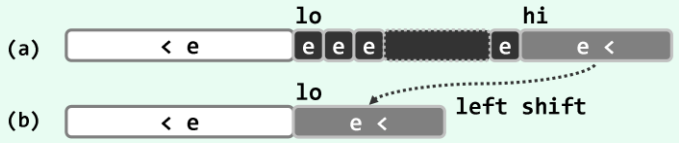

图2.8 有序向量中的重复元素可批量删除

如上所言，此时的每一组重复元素，都必然前后紧邻地集中分布。因此如图2.8所示，可以区间为单位成批地删除前后紧邻的各组重复元素，并将其后继元素（若存在）统一地大跨度前移。具体地，若V[lo, hi)为一组紧邻的重复元素，则所有的后继元素V[hi, _size)可统一地整体前移hi - lo - 1个单元。

■ 高效版

按照上述思路，可如代码2.19所示得到唯一化算法的新版本。

```
1 template <typename T> int Vector<T>::uniquify() { //有序向量重复元素剔除算法（高效版）
2    Rank i = 0, j = 0; //各对互异“相邻”元素的秩
3    while ( ++j < _size ) //逐一扫描，直至末元素
4       if ( _elem[i] != _elem[j] ) //跳过雷同者
5          _elem[++i] = _elem[j]; //发现不同元素时，向前移至紧邻于前者右侧
6    _size = ++i; shrink(); //直接截除尾部多余元素
7    return j - i; //向量规模变化量，即被删除元素总数
8 }
```

代码2.19 有序向量uniquify()接口的高效实现

图2.9针对一个有序向量的实例，完整地给出了该算法对应的执行过程。

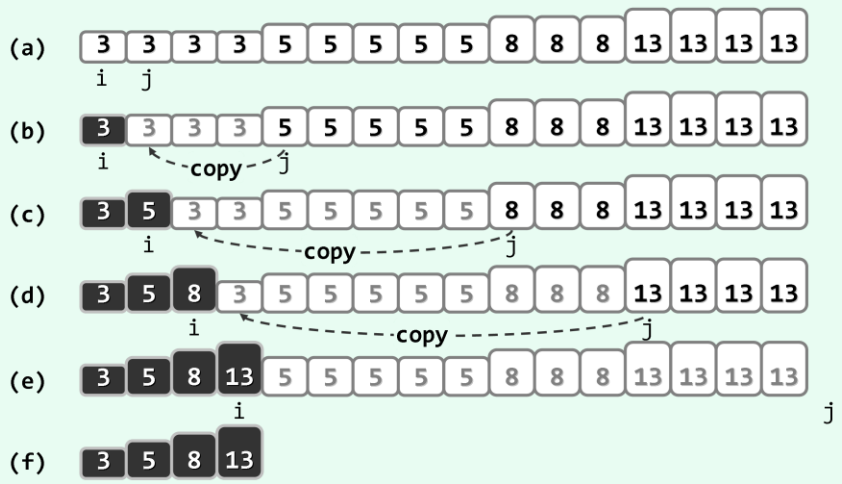

图2.9 在有序向量中查找互异的相邻元素

同样地，既然各组重复元素必然彼此相邻地构成一个子区间，故只需依次保留各区间的起始元素。于是，这里引入了变量i和j。每经过若干次移动，i和j都将分别指向下一对相邻子区间的首元素；在将后者转移至前者的后继位置之后，即可重复上述过程。具体地如图(a)所示，初始时i = 0和j = 1分别指向最靠前两个元素。

接下来，逐位后移j，直至指向A[j=4] = 5 ≠ A[i=0]。如图(b)，此时可见，i和j的确分别指向3和5所在分组的首元素。接下来，令i = 1，并将A[j=4] = 5前移至A[i=1]处。此时的i指向刚被前移的A[1] = 5；令j = j + 1 = 5指向待扫描的下一元素A[5] = 5，并继续比较。如图(c)，此轮比较终止于 A[j=9] = 8 ≠ A[i=1] = 5。

于是，令i = i + 1 = 2，并将A[j=9] = 8前移至A[i=2]处。此时的i指向刚被前移的A[2] = 8；令j = j + 1 = 10指向待扫描的下一元素A[10] = 8，并继续比较。如图(d)，此轮比较终止于A[12] = 13 ≠ A[i=2] = 8。于是，令i = i + 1 = 3，并将A[j=12] = 13前移至A[i=3]处。此时的i指向刚被前移的A[3] = 13；令j = j + 1 = 13指向待扫描的下一元素A[13] = 13，并继续比较。如图(e)，至j = 16 ≥ _size时，循环结束。最后如图(f)，只需将向量规模更新为_size = i + 1 = 4，算法随即结束。鉴于在删除重复元素之后内部数组的空间利用率可能下降很多，故需调用shrink()，如有必要则做缩容处理。

■ **复杂度**

while循环的每一步迭代，仅需对元素数值做一次比较，向后移动一到两个位置指针，并至多向前复制一个元素，故只需常数时间。而在整个算法过程中，每经过一步迭代秩j都必然加一，鉴于j不能超过向量的规模n，故共需迭代n次。由此可知，uniquify()算法的时间复杂度应为$\mathcal{O}(n)$，较之uniquifySlow()的$\mathcal{O}(n^2)$，效率整整提高了一个线性因子。

反过来，在遍历所有元素之前不可能确定是否有重复元素，故就渐进复杂度而言，能在$\mathcal{O}(n)$时间内完成向量的唯一化已属最优。当然，之所以能够做到这一点，关键在于向量已经排序。

2.6.4 查找

有序向量S中的元素不再随机分布，秩r是S[r]在S中按大小的相对位次，位于S[r]前（后）方的元素均不致于更大（小）。当所有元素互异时，r即是S中小于S[r]的元素数目。一般地，若小于、等于S[r]的元素各有i、k个，则该元素及其雷同元素应集中分布于S[i, i + k)。

利用上述性质，有序向量的查找操作可以更加高效地完成。尽管在最坏情况下，无序向量的查找操作需要线性时间，但我们很快就会看到，有序向量的这一效率可以提升至$\mathcal{O}(\log n)$。

为区别于无序向量的查找接口find()，有序向量的查找接口将统一命名为search()。与find()一样，代码2.1也针对有序向量的整体或区间查找重载了两个search()接口，且前者作为特例可直接调用后者。因此，只需如代码2.20所示实现其中的区间查找接口。

```
1 template <typename T> //在有序向量的区间[lo, hi)内，确定不大于e的最后一个节点的秩
2 Rank Vector<T>::search ( T const& e, Rank lo, Rank hi ) const { //assert: 0 <= lo < hi <= _size
3    return ( rand() % 2 ) ? //按各50%的概率随机使用二分查找或Fibonacci查找
4           binSearch ( _elem, e, lo, hi ) : fibSearch ( _elem, e, lo, hi );
5 }
```

代码2.20 有序向量各种查找算法的统一search()接口

鉴于有序查找的算法多样且各具特点，为便于测试，这里的接口不妨随机选择查找算法。实际应用中可根据问题的特点具体确定，并做适当微调。以下将介绍两类典型的查找算法。

2.6.5 二分查找（版本A）

■ 减而治之

循秩访问的特点加上有序性，使得我们可将“减而治之” 策略运用于有序向量的查找。具体地如图2.10所示，假设在区间S[lo, hi)中查找目标元素e。

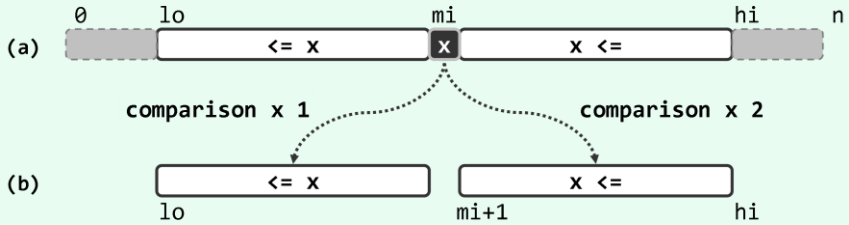

图2.10 基于减治策略的有序向量二分查找算法（版本A）

以任一元素S[mi] = x为界，都可将区间分为三部分，且根据此时的有序性必有：

$$S[lo, mi) \le S[mi] \le S(mi, hi)$$

于是，只需将目标元素e与x做一比较，即可视比较结果分三种情况做进一步处理：1）若e < x，则目标元素如果存在，必属于左侧子区间S[lo, mi)，故可深入其中继续查找；2）若x < e，则目标元素如果存在，必属于右侧子区间S(mi, hi)，故也可深入其中继续查找；3）若e = x，则意味着已经在此处命中，故查找随即终止。

也就是说，每经过至多两次比较操作，我们或者已经找到目标元素，或者可以将查找问题简化为一个规模更小的新问题。如此，借助递归机制即可便捷地描述和实现此类算法。实际上，以下将要介绍的各种查找算法都可归入这一模式，不同点仅在于其对切分点mi的选取策略，以及每次深入递归之前所做比较操作的次数。

■ 实现

按上述思路实现的第一个算法如代码2.21所示。为区别于稍后将要介绍的同类算法的其它版本，不妨将其记作版本A。

```
1 // 二分查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
3    while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支
4       Rank mi = ( lo + hi ) >> 1; //以中点为轴点
5       if      ( e < A[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
6       else if ( A[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)继续查找
7       else                  return mi; //在mi处命中
8    } //成功查找可以提前终止
9    return -1; //查找失败
10 } //有多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败的位置
```

代码2.21 二分查找算法（版本A）

为在有序向量区间[lo, hi)内查找元素e，该算法以中点mi = (lo + hi) / 2为界，将其大致平均地分为前、后两个子向量。随后通过一至两次比较操作，确定问题转化的方向。通过快捷的整数移位操作回避了相对更加耗时的除法运算。另外，通过引入lo、hi和mi等变量，将减

治算法通常的递归模式改成了迭代模式。

■ 实例

如图2.11左侧所示，设通过调用search(8, 0, 7)，在有序向量区间S[0, 7)内查找目标元素8。第一步迭代如图(a1)所示，取mi = (0 + 7)/2 = 3，经过1次失败的比较另加1次成功的比较后确认S[mi = 3] = 7 < 8，故深入后半段S[4, 7)。第二步迭代如图(a2)所示，取mi = (4 + 7)/2 = 5，经过1次成功的比较后确认8 < S[mi = 5] = 9，故深入前半段S[4, 5)。最后一步迭代如图(a3)所示，取mi = (4 + 5)/2 = 4，经过2次失败的比较后确认8 = S[mi = 4]。前后总共经过3步迭代和5次比较操作，最终通过返回合法的秩mi = 4，指示对目标元素8的查找在元素S[4]处成功命中。

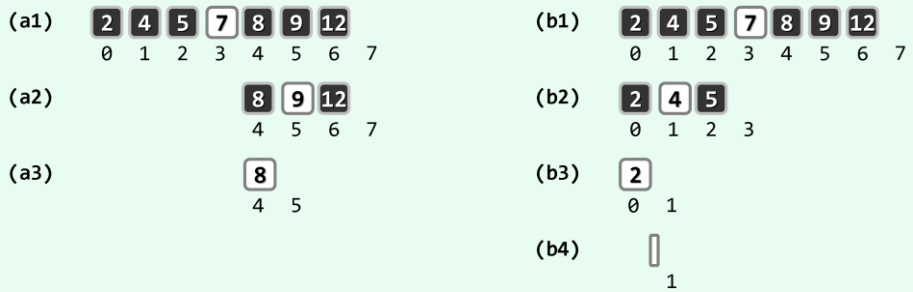

图2.11 二分查找算法（版本A）实例：search(8, 0, 7)成功，search(3, 0, 7)失败

再如图2.11右侧所示，设通过调用search(3, 0, 7)，在同一向量区间内查找目标元素3。第一步迭代如图(b1)所示，取mi = (0 + 7) / 2 = 3，经过1次成功的比较后确认3 < S[mi = 3] = 7，故深入前半段S[0, 3)。第二步迭代如图(b2)所示，取mi = (0 + 3) / 2 = 1，经过1次成功的比较后确认3 < S[mi = 1] = 4，故深入前半段S[0, 1)。第三步迭代如图(b3)所示，取mi = (0 + 1) / 2 = 0，经过1次失败的比较另加1次成功的比较后确认S[mi = 0] = 2 < 3，故深入“后半段”S[1, 1)。此时因为lo = 1 = hi，故最后一步迭代实际上并不会执行；while循环退出后，算法通过返回非法的秩-1指示查找失败。纵观整个查找过程，前后总共经过4步迭代和4次比较操作。

■ 复杂度

以上算法采取的策略可概括为，以“当前区间内居中的元素”作为目标元素的试探对象。从应对最坏情况的保守角度来看，这一策略是最优的——每一步迭代之后无论沿着哪个方向深入，新问题的规模都将缩小一半。因此，这一策略亦称作二分查找（binary search）。

也就是说，随着迭代的不断深入，有效的查找区间宽度将按1/2的比例以几何级数的速度递减。于是，经过至多$\log_2(hi - lo)$步迭代后，算法必然终止。鉴于每步迭代仅需常数时间，故总体时间复杂度不超过：

$$\mathcal{O}(\log_2(hi - lo)) = \mathcal{O}(\log n)$$

与代码2.10中顺序查找算法的$\mathcal{O}(n)$复杂度相比，$\mathcal{O}(\log n)$几乎改进了一个线性因子。

■ 查找长度

以上迭代过程所涉及的计算，主要分为两类：元素的大小比较、秩的算术运算及其赋值。虽然二者均属于$\mathcal{O}(1)$复杂度的基本操作，但元素的秩无非是（无符号）整数，而向量元素的类型

则通常更为复杂，甚至复杂到未必能够保证在常数时间内完成（习题[2-17]）。因此就时间复杂度的常系数而言，前一类计算的权重远远高于后者，而查找算法的整体效率也更主要地取决于其中所执行的元素大小比较操作的次数，即所谓查找长度（search length）。

通常，可针对查找成功或失败等情况，从最好、最坏和平均情况等角度，分别测算查找长度，并凭此对查找算法的总体性能做一评估。

■ 成功查找长度

对于长度为n的有序向量，共有n种可能的成功查找，分别对应于某一元素。实际上，每一种成功查找所对应的查找长度，仅取决于n以及目标元素所对应的秩，而与元素的具体数值无关。比如，回顾图2.11中的实例不难看出，无论怎样修改那7个元素的数值，只要它们依然顺序排列，则针对S[4]的查找过程（包括各步迭代的比较次数以及随后的深入方向）必然与在原例中执行search(8, 0, 7)的过程完全一致。

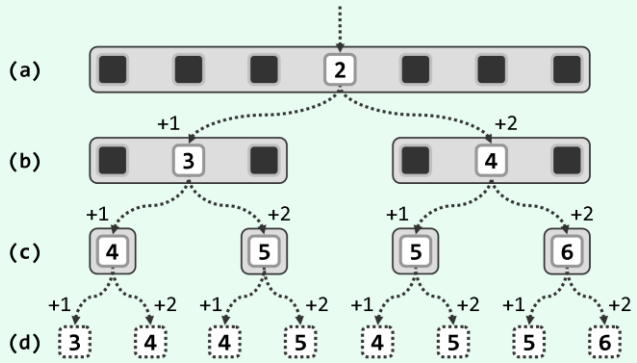

图2.12 二分查找算法（版本A）的查找长度（成功、失败查找分别以实线、虚线白色方框示意）

当n = 7时由图2.12不难验证，各元素所对应的成功查找长度分别应为：

{ 4, 3, 5, 2, 5, 4, 6 }

若假定查找的目标元素按等概率分布，则平均查找长度即为：

(4 + 3 + 5 + 2 + 5 + 4 + 6) / 7 = 29 / 7 = 4.14

为了估计出一般情况下的成功查找长度，不失一般性地，仍在等概率条件下考查长度为$n = 2^k - 1$的有序向量，并将其对应的平均成功查找长度记作$c_{average}(k)$，将所有元素对应的查找长度总和记作$C(k) = c_{average}(k)\cdot(2^k - 1)$。

特别地，当k = 1时向量长度n = 1，成功查找仅有一种情况，故有边界条件：

$c_{average}(1) = C(1) = 2$

以下采用递推分析法。对于长度为$n = 2^k - 1$的有序向量，每步迭代都有三种可能的分支：经过1次成功的比较后，转化为一个规模为$2^{k-1} - 1$的新问题（图2.12中的左侧分支）；经2次失败的比较后，终止于向量中的某一元素，并确认在此处成功命中；经1次失败的比较另加1次成功的比较后，转化为另一个规模为$2^{k-1} - 1$的新问题（图2.12中的右侧分支）。

根据以上递推分析的结论，可得递推式如下：

$C(k) = [C(k - 1) + (2^{k-1} - 1)] + 2 + [C(k - 1) + 2\times(2^{k-1} - 1)]$（式2-1）

$= 2\cdot C(k - 1) + 3\cdot 2^{k-1} - 1$

若令：

$$F(k) = C(k) - 3k \cdot 2^{k-1} - 1$$

则有：

$$F(1) = -2$$

$$F(k) = 2 \cdot F(k-1) = 2^2 \cdot F(k-2) = 2^3 \cdot F(k-3) = \dots = 2^{k-1} \cdot F(1) = -2^k$$

于是：

$$C(k) = F(k) + 3k \times 2^{k-1} + 1 = -2^k + 3k \times 2^{k-1} + 1 = (3k/2 - 1) \cdot (2^k - 1) + 3k/2$$

进而：

$$c_{average}(k) = C(k) / (2^k - 1) = 3k/2 - 1 + 3k/2/(2^k - 1) = 3k/2 - 1 + \mathcal{O}(\varepsilon)$$

也就是说，若忽略末尾趋于收敛的波动项，平均查找长度应为：

$$\mathcal{O}(1.5k) = \mathcal{O}(1.5 \cdot \log_2 n)$$

■ 失败查找长度

按照代码2.21，失败查找的终止条件必然是“lo ≥ hi”，也就是说，只有在有效区间宽度缩减至0时，查找方以失败告终。因此，失败查找的时间复杂度应为确定的$\Theta(\log n)$。

不难发现，就各步迭代后分支方向的组合而言，失败查找可能的情况，恰好比成功查找可能的情况多出一种。例如在图2.12中，失败查找共有7 + 1 = 8种可能。由图2.12不难验证，各种可能所对应的查找长度分别为：

{ 3, 4, 4, 5, 4, 5, 5, 6 }

其中，最好情况下需要做3次元素比较，最坏情况下需要做6次元素比较。若同样地假定待查找目标元素按等概率分布，则平均查找长度应为：

(3 + 4 + 4 + 5 + 4 + 5 + 5 + 6) / 8 = 36 / 8 = 4.50

仿照以上对平均成功查找长度的递推分析方法，不难证明（习题[2-20]），一般情况下的平均失败查找长度亦为$\mathcal{O}(1.5 \cdot \log_2 n)$。

■ 不足

尽管二分查找算法（版本A）即便在最坏情况下也可保证$\mathcal{O}(\log n)$的渐进时间复杂度，但就其常系数1.5而言仍有改进余地。以成功查找为例，即便是迭代次数相同的情况，对应的查找长度也不尽相等。究其根源在于，在每一步迭代中为确定左、右分支方向，分别需要做1次或2次元素比较，从而造成不同情况所对应查找长度的不均衡。尽管该版本从表面上看完全均衡，但我们通过以上细致的分析已经看出，最短和最长分支所对应的查找长度相差约两倍。

那么，能否实现更好的均衡呢？具体又应如何实现呢？

2.6.6 Fibonacci查找

■ 递推方程

递推方程法既是复杂度分析的重要方法，也是我们优化算法时确定突破口的有力武器。为改

进以上二分查找算法的版本A，不妨从刻画查找长度随向量长度递推变化的式2-1入手。

实际上，最终求解所得到的平均复杂度，在很大程度上取决于这一等式。更准确地讲，主要取决于$(2^{k-1} - 1)$和$2 \times (2^{k-1} - 1)$两项，其中的$(2^{k-1} - 1)$为子向量的宽度，而系数1和2则是算法为深入前、后子向量，所需做的比较操作次数。以此前的二分查找算法版本A为例，之所以存在均衡性方面的缺陷，根源正在于这两项的大小不相匹配。

基于这一理解，不难找到解决问题的思路，具体地不外乎两种：

其一，调整前、后区域的宽度，适当地加长（缩短）前（后）子向量
其二，统一沿两个方向深入所需要执行的比较次数，比如都统一为一次

后一思路的实现将在稍后介绍，以下首先介绍前一思路的具体实现。

■ 黄金分割

实际上，减治策略本身并不要求子向量切分点mi必须居中，故按上述改进思路，不妨按黄金分割比来确定mi。为简化起见，不妨设向量长度n = fib(k) - 1。

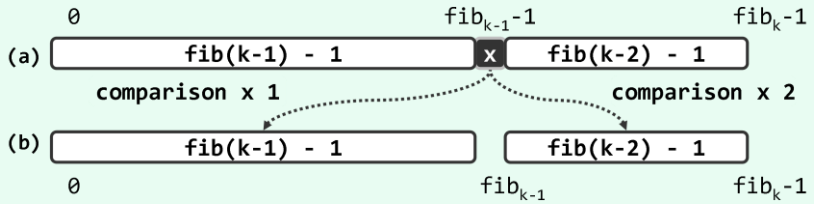

图2.13 Fibonacci查找算法原理

于是如图2.13所示，fibSearch(e, 0, n)查找可以mi = fib(k - 1) - 1作为前、后子向量的切分点。如此，前、后子向量的长度将分别是：

fib(k - 1) - 1

fib(k - 2) - 1 = (fib(k) - 1) - (fib(k - 1) - 1) - 1

于是，无论朝哪个方向深入，新向量的长度从形式上都依然是某个Fibonacci数减一，故这一处理手法可以反复套用，直至因在S[mi]处命中或向量长度收缩至零而终止。这种查找算法，亦称作Fibonacci查找（Fibonaccian search）。

■ 实现

按照以上思路，可实现Fibonacci查找算法如代码2.22所示。

```
1  #include "..\fibonacci\Fib.h" //引入Fib数列类
2  // Fibonacci查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
3  template <typename T> static Rank fibSearch ( T* A, T const& e, Rank lo, Rank hi ) {
4     Fib fib ( hi - lo ); //用O(log_phi(n = hi - lo)时间创建Fib数列
5     while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支
6        while ( hi - lo < fib.get() ) fib.prev(); //通过向前顺序查找（分摊O(1)）——至多迭代几次？
7        Rank mi = lo + fib.get() - 1; //确定形如Fib(k) - 1的轴点
8        if      ( e < A[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
9        else if ( A[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)继续查找
10       else                  return mi; //在mi处命中
11    } //成功查找可以提前终止
```

```
12     return -1; //查找失败
13 } //有多个命中元素时，不能保证返回秩最大者；失败时，简单地返回-1，而不能指示失败的位置
```

代码2.22 Fibonacci查找算法

算法主体框架与二分查找大致相同，主要区别在于以黄金分割点取代中点作为切分点。为此，需要借助Fib对象（习题[1-22]），实现对Fibonacci数的高效设置与获取。

尽管以下的分析多以长度为fib(k) - 1的向量为例，但这一实现完全可适用于长度任意的向量中的任意子向量。为此，只需在进入循环之前调用构造器Fib(n = hi - lo)，将初始长度设置为“不小于n的最小Fibonacci项”。这一步所需花费的$\mathcal{O}(\log_{\Phi} n)$时间，分摊到后续的$\mathcal{O}(\log_{\Phi} n)$步迭代中，并不影响算法整体的渐进复杂度。

■ 定性比较

可见，Fibonacci查找倾向于适当加长（缩短）需1次（2次）比较方可确定的前端（后端）子向量。故定性地粗略估计，应可（在常系数的意义上）进一步提高查找的效率。

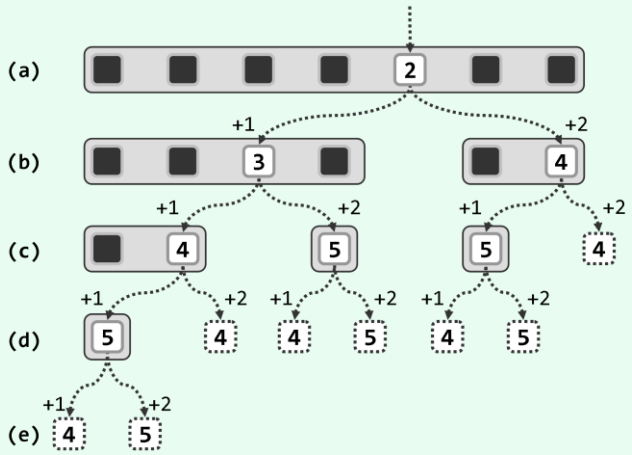

图2.14 Fibonacci查找算法的查找长度（成功、失败查找分别以实线、虚线白色方框示意）

作为验证，不妨仍以n = fib(6) - 1 = 7为例，就平均查找长度与二分查找做一对比。

如图2.14所示，7种成功情况、8种失败情况的查找长度分别为：

```
{ 5, 4, 3, 5, 2, 5, 4 }
{ 4, 5, 4, 4, 5, 4, 5, 4 }
```

若依然假定各种情况出现的概率相等，则平均成功查找长度和平均失败查找长度应分别为：

```
(5 + 4 + 3 + 5 + 2 + 5 + 4) / 7  =  28 / 7  =  4.00
(4 + 5 + 4 + 4 + 5 + 4 + 5 + 4) / 8  =  35 / 8  =  4.38
```

相对于二分查找算法版本A实例（图2.12）的4.14和4.50，的确有所改进。

■ 定量分析

参照2.6.5节的方法，也可对Fibonacci查找算法的成功查找长度做出最更为精确的分析。其中关于最好、最坏情况的结论完全一致，故以下仅讨论等概率条件下的平均情况。

依然将长度为n = fib(k) - 1的有序向量的平均成功查找长度记作$c_{average}(k)$，将所有元素对应的查找长度总和记作$C(k) = c_{average}(k)\cdot(fib(k) - 1)$。

同理，可得边界条件及递推式如下：

$$c_{average}(2) = C(2) = 0$$

$$c_{average}(3) = C(3) = 2$$

$$C(k) = [C(k-1) + (fib(k-1) - 1)] + 2 + [C(k-2) + 2\times(fib(k-2) - 1)]$$

$$= C(k - 2) + C(k - 1) + fib(k - 2) + fib(k) - 1$$

结合以上边界条件，可以解得：

$$C(k) =^{①} k\cdot fib(k) - fib(k + 2) + 1 = (k - \Phi^2)\cdot fib(k) + 1 + \mathcal{O}(\varepsilon)$$

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

于是

$$c_{average}(k) = C(k)/(fib(k) - 1)$$

$$= k - \Phi^2 + 1 + (k - \Phi^2)/(fib(k) - 1) + \mathcal{O}(\varepsilon)$$

$$= k - \Phi^2 + 1 + \mathcal{O}(\varepsilon)$$

也就是说，忽略末尾趋于收敛的波动项，平均查找长度的增长趋势为：

$$\mathcal{O}(k) = \mathcal{O}(\log_\Phi n) = \mathcal{O}(\log_\Phi 2 \cdot \log_2 n) = \mathcal{O}(1.44\cdot\log_2 n)$$

较之2.6.5节二分查找算法（版本A）的$\mathcal{O}(1.50\cdot\log_2 n)$，效率略有提高。

2.6.7 二分查找（版本B）

■ 从三分支到两分支

2.6.6节开篇曾指出，二分查找算法版本A的不均衡性体现为复杂度递推式中$(2^{k-1} - 1)$和$2\times(2^{k-1} - 1)$两项的不均衡。为此，Fibonacci查找算法已通过采用黄金分割点，在一定程度上降低了时间复杂度的常系数。实际上还有另一更为直接的方法，即令以上两项的常系数同时等于1。也就是说，无论朝哪个方向深入，都只需做1次元素的大小比较。相应地，算法在每步迭代中（或递归层次上）都只有两个分支方向，而不再是三个。

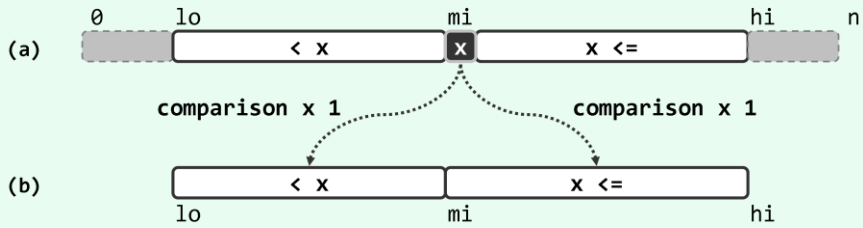

图2.15 基于减治策略的有序向量二分查找算法（版本B）

具体过程如图2.15所示，与二分查找算法的版本A基本类似。不同之处是，在每个切分点A[mi]处，仅做一次元素比较。具体地，若目标元素小于A[mi]，则深入前端子向量A[lo, mi)继续查找；否则，深入后端子向量A[mi, hi)继续查找。

■ 实现

按照上述思路，可将二分查找算法改进为如代码2.23所示的版本B。

① 令F(k) = -C(k) + k·fib(k) + 1，则有F(0) = 1，F(1) = 2，F(k) = F(k-1) + F(k-2) = fib(k+2)

```
1 // 二分查找算法（版本B）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
3    while ( 1 < hi - lo ) { //每步迭代仅需做一次比较判断，有两个分支；成功查找不能提前终止
4       Rank mi = ( lo + hi ) >> 1; //以中点为轴点
5       ( e < A[mi] ) ? hi = mi : lo = mi; //经比较后确定深入[lo, mi)或[mi, hi)
6    } //出口时hi = lo + 1，查找区间仅含一个元素A[lo]
7    return ( e == A[lo] ) ? lo : -1 ; //查找成功时返回对应的秩；否则统一返回-1
8 } //有多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败的位置
```

代码2.23 二分查找算法（版本B）

请再次留意与代码2.21中版本A的差异。首先，每一步迭代只需判断是否e < A[mi]，即可相应地更新有效查找区间的右边界（hi = mi）或左边界（lo = mi）。另外，只有等到区间的宽度已不足2个单元时迭代才会终止，最后再通过一次比对判断查找是否成功。

■ **性能**

尽管版本B中的后端子向量需要加入A[mi]，但得益于mi总是位于中央位置，整个算法$\mathcal{O}(\log n)$的渐进复杂度不受任何影响。

在这一版本中，只有在向量有效区间宽度缩短至1个单元时算法才会终止，而不能如版本A那样，一旦命中就能及时返回。因此，最好情况下的效率有所倒退。当然，作为补偿，最坏情况下的效率相应地有所提高。实际上无论是成功查找或失败查找，版本B各分支的查找长度更加接近，故整体性能更趋稳定。

■ **进一步的要求**

在更多细微之处，此前实现的二分查找算法（版本A和B）及Fibonacci查找算法仍有改进的余地。比如，当目标元素在向量中重复出现时，它们只能“随机”地报告其一，具体选取何者取决于算法的分支策略以及当时向量的组成。而在很多场合中，重复元素之间往往会隐含地定义有某种优先级次序，而且算法调用者的确可能希望得到其中优先级最高者。比如按照表2.1的定义，在有多个命中元素时，向量的search()接口应以它们的秩为优先级，并返回其中最靠后者。

这种进一步的要求并非多余。以有序向量的插入操作为例，若通过查找操作不仅能够确定可行的插入位置，而且能够在同时存在多个可行位置时保证返回其中的秩最大者，则不仅可以尽可能低减少需移动的后继元素，更可保证重复的元素按其插入的相对次序排列。对于向量的插入排序等算法（习题[3-8]）的稳定性而言，这一性质更是至关重要。

另外，对失败查找的处理方式也可以改进。查找失败时，以上算法都是简单地统一返回一个标识“-1”。同样地，若在插入新元素e之前通过查找确定适当的插入位置，则希望在查找失败时返回不大（小）于e的最后（前）一个元素，以便将e作为其后继（前驱）插入向量。同样地，此类约定也使得插入排序等算法的实现更为便捷和自然。

2.6.8 二分查找（版本C）

■ **实现**

在版本B的基础上略作修改，即可得到如代码2.24所示二分查找算法的版本C。

```
1 // 二分查找算法（版本C）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
2 template <typename T> static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
3    while ( lo < hi ) { //每步迭代仅需做一次比较判断，有两个分支
4       Rank mi = ( lo + hi ) >> 1; //以中点为轴点
5       ( e < A[mi] ) ? hi = mi : lo = mi + 1; //经比较后确定深入[lo, mi)或(mi, hi)
6    } //成功查找不能提前终止
7    return --lo; //循环结束时，lo为大于e的元素的最小秩，故lo - 1即不大于e的元素的最大秩
8 } //有多个命中元素时，总能保证返回秩最大者；查找失败时，能够返回失败的位置
```

代码2.24 二分查找算法（版本C）

该版本的主体结构与版本B一致，故不难理解，二者的时间复杂度相同。

■ **正确性**

版本C与版本B的差异，主要有三点。首先，只有当有效区间的宽度缩短至0（而不是1）时，查找方告终止。另外，在每次转入后端分支时，子向量的左边界取作mi + 1而不是mi。

表面上看，后一调整存在风险——此时只能确定切分点A[mi] ≤ e，“贸然”地将A[mi]排除在进一步的查找范围之外，似乎可能因遗漏这些元素，而导致本应成功的查找以失败告终。

然而这种担心大可不必。通过数学归纳可以证明，版本C中的循环体，具有如下不变性：

A[0, lo)中的元素皆不大于e；A[hi, n)中的元素皆大于e

首次迭代时，lo = 0且hi = n，A[0, lo)和A[hi, n)均空，不变性自然成立。

如图2.16(a)所示，设在某次进入循环时以上不变性成立，以下无非两种情况。若e < A[mi]，则如图(b)，在令hi = mi并使A[hi, n)向左扩展之后，该区间内的元素皆不小于A[mi]，当然也仍然大于e。反之，若A[mi] ≤ e，则如图(c)，在令lo = mi + 1并使A[0, lo)向右拓展之后，该区间内的元素皆不大于A[mi]，当然也仍然不大于e。总之，上述不变性必然得以延续。

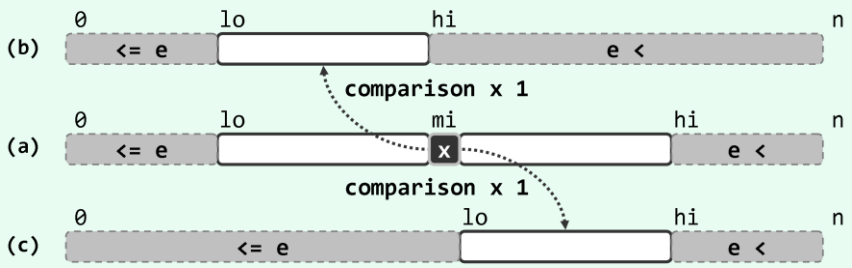

图2.16 基于减治策略的有序向量二分查找算法（版本C）

循环终止时，lo = hi。考查此时的元素A[lo - 1]和A[lo]：作为A[0, lo)内的最后一个元素，A[lo - 1]必不大于e；作为A[lo, n) = A[hi, n)内的第一个元素，A[lo]必大于e。也就是说，A[lo - 1]即是原向量中不大于e的最后一个元素。因此在循环结束之后，无论成功与否，只需返回lo - 1即可——这也是版本C与版本B的第三点差异。

§2.7 *排序与下界

2.7.1 有序性

从数据处理的角度看，有序性在很多场合都能够极大地提高计算的效率。以查找算法为例，对于无序向量，正如此前的分析结论，代码2.10中Vector::find()算法$\mathcal{O}(n)$的复杂度已属最优。而对于有序向量，代码2.20中Vector::search()接口的效率，则可优化到$\mathcal{O}(\log n)$。我们知道，为此需要借助二分查找策略，而之所以这一策略可行，正是因为所有元素已按次序排列。

2.7.2 排序及其分类

由以上介绍可见，有序向量的诸如查找等操作，效率远高于一般向量。因此在解决许多应用问题时我们普遍采用的一种策略就是，首先将向量转换为有序向量，再调用有序向量支持的各种高效算法。这一过程的本质就是向量的排序。为此，正如2.6.1节所指出的，向量元素之间必须能够定义某种全序关系，以保证它们可相互比较大小。

排序算法是个庞大的家族，可从多个角度对其中的成员进行分类。比如，根据其处理数据的规模与存储的特点不同，可分为内部排序算法和外部排序算法：前者处理的数据规模相对不大，内存足以容纳；后者处理的数据规模很大，必须将借助外部甚至分布式存储器，在排序计算过程的任一时刻，内存中只能容纳其中一小部分数据。

又如，根据输入形式的不同，排序算法也可分为离线算法（offline algorithm）和在线算法（online algorithm）。前一情况下，待排序的数据以批处理的形式整体给出；而在网络计算之类的环境中，待排序的数据通常需要实时生成，在排序算法启动后数据才陆续到达。再如，针对所依赖的体系结构不同，又可分为串行和并行两大类排序算法。另外，根据排序算法是否采用随机策略，还有确定式和随机式之分。

本书讨论的范围，主要集中于确定式串行脱机的内部排序算法。

2.7.3 下界

根据1.2.2节的分析，1.1.3节起泡排序算法的复杂度为$\mathcal{O}(n^2)$。那么，这一效率是否已经足够高？能否以更快的速度完成排序？实际上，在着手优化算法之前，这都是首先必须回答的问题。以下结合具体实例，从复杂度下界的角度介绍回答此类问题的一般性方法。

■ 苹果鉴别

考虑如下问题：三只苹果外观一样，其中两只重量相同另一只不同，利用一架天平如何从中找出重量不同的那只？一种直观的方法可以描述为算法2.1。

该算法的可行性、正确性毋庸置疑。该算法在最好情况下仅需执行一次比对操作，最坏情况下两次。那么，是否存在其它算法，即便在最坏情况下也至多只需一次比对呢？

```
identifyApple(A, B, C)
输入：三只苹果A、B和C，其中两只重量相同，另一只不同
输出：找出重量不同的那只苹果
{
   称量A和B；若A和B重量相等，则返回C；
   称量A和C；若A和C重量相等，则返回B；
   否则，返回A；
}
```

算法2.1 从三个苹果中选出重量不同者

■ 复杂度下界

尽管很多算法都可以优化，但有一个简单的事实却往往为人所忽略：对任一特定的应用问题，随着算法的不断改进，其效率的提高必然存在某一极限。毕竟，我们不能奢望不劳而获。这一极限不仅必然存在，而且其具体的数值，应取决于应用问题本身以及所采用的计算模型。

一般地，任一问题在最坏情况下的最低计算成本，即为该问题的复杂度下界（lower bound）。一旦某一算法的性能达到这一下界，即意味着它已是最坏情况下最优的（worst-case optimal）。可见，尽早确定一个问题的复杂度下界，对相关算法的优化无疑会有巨大的裨益。比如上例所提出的问题，就是从最坏情况的角度，质疑“2次比对操作”是否为解决这一问题的最低复杂度。

以下结合比较树模型，介绍界定问题复杂度下界的一种重要方法。

2.7.4 比较树

■ 基于比较的分支

如果用节点（圆圈）表示算法过程中的不同状态，用有方向的边（直线段或弧线段）表示不同状态之间的相互转换，就可以将以上算法2.1转化为图2.17的树形结构（第5章）。

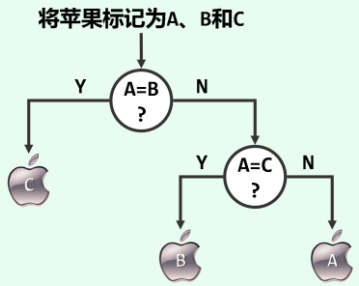

图2.17 从三只苹果中挑出重量不同者

这一转化方法也可以推广并应用于其它算法。一般地，树根节点对应于算法入口处的起始状态（如此处三个苹果已做好标记）；内部节点（即非末端节点，图中以白色大圈示意）对应于过程中的某步计算，通常属于基本操作；叶节点（即末端节点，图中以黑色小圈示意）则对应于经一系列计算后某次运行的终止状态。如此借助这一树形结构，可以涵盖对应算法所有可能的执行流程。

仍以图2.17为例，从根节点到叶节点C的路径对应于，在经过一次称量比较并确定A与B等重后，即可断定C是所要查找的苹果。再如，从根节点到叶节点B的路径对应于，在经过两次称量比较并确定A与B不等重、A与C等重之后，即可判定B是所要查找的苹果。

■ 比较树

算法所有可能的执行过程，都可涵盖于这一树形结构中。具体地，该树具有以下性质：

① 每一内部节点各对应于一次比对（称量）操作；

② 内部节点的左、右分支，分别对应于在两种比对结果（是否等重）下的执行方向；

③ 叶节点（或等效地，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。

按上述规则与算法相对应的树，称作比较树（comparison tree）。

不难理解，无论什么算法，只要其中的分支都如算法2.1那样，完全取决于不同变量或常量的比对或比较结果，则该算法所有可能的执行过程都可表示和概括为一棵比较树。反之，凡可如此描述的算法，都称作基于比较式算法（comparison-based algorithm），简称CBA式算法。比如在本书中，除散列之外的算法大多属于此类。

以下我们将看到，CBA式算法在最坏情况下的最低执行成本，可由对应的比较树界定。

2.7.5 估计下界

■ **最小树高**

考查任一CBA式算法A，设CT(A)为与之对应的一棵比较树。

根据比较树的性质，算法A每一次运行所需的时间，将取决于其对应叶节点到根节点的距离（称作叶节点的深度）；而算法A在最坏情况下的运行时间，将取决于比较树中所有叶节点的最大深度（称作该树的高度，记作h(CT(A))）。因此就渐进的意义而言，算法A的时间复杂度应不低于Ω(h(CT(A)))。

对于存在CBA式算法的计算问题，既然其任一CBA式算法均对应于某棵比较树，该问题的复杂度下界就应等于这些比较树的最小高度。问题在于，如何估计这些比较树的最小高度呢？

为此，只需考查树中所含叶节点（可能的输出结果）的数目。具体地，在一棵高度为h的二叉树中，叶节点的数目不可能多于2^h。因此反过来，若某一问题的输出结果不少于N种，则比较树中叶节点也不可能少于N个，树高h不可能低于$\log_2 N$（习题[7-3]）。

■ **苹果鉴别**

仍以算法2.1为例。就该问题而言，可能的输出结果共计N = 3种（不同的苹果分别为A、B或C），故解决该问题的任一CBA式算法所对应比较树的高度为：

$$h \geq \lceil \log_2 3 \rceil = 2$$

因此，只要是采用CBA式算法来求解该问题，则无论如何优化，在最坏情况下都至少需要2次称量——尽管最好情况下的确仍可能仅需1次。这也意味着，算法2.1虽平淡无奇，却已是解决苹果鉴别问题的最佳CBA式算法。

■ **排序**

再以CBA式排序算法为例。就n个元素的排序问题而言，可能的输出共有N = n!种。与上例略有不同之处在于，元素之间不仅可以判等而且可以比较大小，故此时的比较树应属于三叉树，即每个内部节点都有三个分支（分别对应小于、相等和大于的情况）。不过，这并不影响上述分析方法的运用。按照以上思路，任一CBA式排序算法所对应比较树的高度应为：

$$h \geq \lceil \log_3(n!) \rceil = \lceil \log_3 e \cdot \ln(n!) \rceil \overset{②}{=} \Omega(n \log n)$$

可见，最坏情况下CBA式排序算法至少需要Ω(nlogn)时间，其中n为待排序元素数目。

需强调的是，这一Ω(nlogn)下界是针对比较树模型而言的。事实上，还有很多不属此类的排序算法（比如9.4.1节的桶排序算法和9.4.3节的基数排序算法），而且其中一些算法在最坏情况下的运行时间，有可能低于这一下界，但与上述结论并不矛盾。

§2.8 排序器

2.8.1 统一入口

鉴于排序在算法设计与实际应用中的重要地位和作用，排序操作自然应当纳入向量基本接口的范围。这类接口也是将无序向量转换为有序向量的基本方法和主要途径。

② 由Stirling逼近公式，$n! \sim \sqrt{2\pi n} \cdot (n/e)^n$

```
template <typename T> void Vector<T>::sort ( Rank lo, Rank hi ) { //向量区间[lo, hi)排序
   switch ( rand() % 5 ) { //随机选取排序算法。可根据具体问题的特点灵活选取或扩充
      case 1:  bubbleSort ( lo, hi ); break; //起泡排序
      case 2:  selectionSort ( lo, hi ); break; //选择排序（习题）
      case 3:  mergeSort ( lo, hi ); break; //归并排序
      case 4:  heapSort ( lo, hi ); break; //堆排序（稍后介绍）
      default: quickSort ( lo, hi ); break; //快速排序（稍后介绍）
   }
}
```

代码2.25 向量排序器接口

针对任意合法向量区间的排序需求，代码2.25定义了统一的入口，并提供起泡排序、选择排序（习题[3-9]）、归并排序、堆排序（10.2.5节）和快速排序（12.1节）等多种算法。为便于测试和对比，这里暂以随机方式确定每次调用的具体算法。在了解这些算法各自所长之后，读者可结合各自具体的应用，根据实际需求灵活地加以选用。

以下先将起泡排序算法集成至向量ADT中，然后讲解归并排序算法的原理、实现。

2.8.2 起泡排序

起泡排序算法已在1.1.3节讲解并实现，这里只需将其集成至向量ADT中。

■ 起泡排序

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

代码2.26 向量的起泡排序

代码2.26给出了起泡排序算法的主体框架，其功能等效于代码1.1中的外层循环：反复调用单趟扫描交换算法，直至逆序现象完全消除。

■ 扫描交换

单趟扫描交换算法，可实现如代码2.27所示。

```
template <typename T> bool Vector<T>::bubble ( Rank lo, Rank hi ) { //一趟扫描交换
   bool sorted = true; //整体有序标志
   while ( ++lo < hi ) //自左向右，逐一检查各对相邻元素
      if ( _elem[lo - 1] > _elem[lo] ) { //若逆序，则
         sorted = false; //意味着尚未整体有序，并需要
         swap ( _elem[lo - 1], _elem[lo] ); //通过交换使局部有序
      }
   return sorted; //返回有序标志
}
```

代码2.27 单趟扫描交换

该算法的功能等效于第5页代码1.1中bubblesort1A()的内层循环：依次比较各对相邻元素，每当发现逆序即令二者彼此交换；一旦经过某趟扫描之后未发现任何逆序的相邻元素，即意味着排序任务已经完成，则通过返回标志“sorted”，以便主算法及时终止。

■ 重复元素与稳定性

稳定性（stability）是对排序算法更为细致的要求，重在考查算法对重复元素的处理效果。具体地，在将向量A转换为有序向量S之后，设A[i]对应于$S[k_i]$。若对于A中每一对重复元素A[i] = A[j]（相应地$S[k_i] = S[k_j]$），都有$i < j$当且仅当$k_i < k_j$，则称该排序算法是稳定算法（stable algorithm）。简而言之，稳定算法的特征是，重复元素之间的相对次序在排序前后保持一致。反之，不具有这一特征的排序算法都是不稳定算法（unstable algorithm）。

比如，依此标准反观起泡排序可以发现，该算法过程中元素相对位置有所调整的唯一可能是，某元素_elem[i - 1]严格大于其后继_elem[i]。也就是说，在这种亦步亦趋的交换过程中，重复元素虽可能相互靠拢，但绝对不会相互跨越。由此可知，起泡排序属于稳定算法。

稳定的排序算法，可用以实现同时对多个关键码按照字典序的排序。比如，后面9.4.3节基数排序算法的正确性，就完全建立在桶排序稳定性的基础之上。

若需兼顾其它方面的性能，以上起泡排序仍有改进的余地（习题[2-25]）。

2.8.3 归并排序

■ 历史与发展

归并排序[③]（mergesort）的构思朴实却亦深刻，作为一个算法既古老又仍不失生命力。在排序算法发展的历史上，归并排序具有特殊的地位，它是第一个可以在最坏情况下依然保持$O(n \log n)$运行时间的确定性排序算法。

时至今日，在计算机早期发展过程中曾经出现的一些难题在更大尺度上再次呈现，归并排序因此重新焕发青春。比如，早期计算机的存储能力有限，以至于高速存储器不能容纳所有的数据，或者只能使用磁带机或卡片之类的顺序存储设备，这些既促进了归并排序的诞生，也为该算法提供了施展的舞台。信息化无处不在的今天，我们再次发现，人类所拥有信息之庞大，不仅迫使我们更多地将它们存放和组织于分布式平台之上，而且对海量信息的处理也必须首先考虑，如何在跨节点的环境中高效地协同计算。因此在许多新算法和技术的背后，都可以看到归并排序的影子。

■ 有序向量的二路归并

与起泡排序通过反复调用单趟扫描交换类似，归并排序也可以理解为是通过反复调用所谓二路归并（2-way merge）算法而实现的。所谓二路归并，就是将两个有序序列合并成为一个有序序列。这里的序列既可以是向量，也可以是第3章将要介绍的列表，这里首先考虑有序向量。归并排序所需的时间，也主要决定于各趟二路归并所需时间的总和。

二路归并属于迭代式算法。每步迭代中，只需比较两个待归并向量的首元素，将小者取出并追加到输出向量的末尾，该元素在原向量中的后继则成为新的首元素。如此往复，直到某一向量为空。最后，将另一非空的向量整体接至输出向量的末尾。

③ 由冯•诺依曼于1945年在EDVAC上首次编程实现

如图2.18(a)所示，设拟归并的有序向量为{ 5, 8, 13, 21 }和{ 2, 4, 10, 29 }。

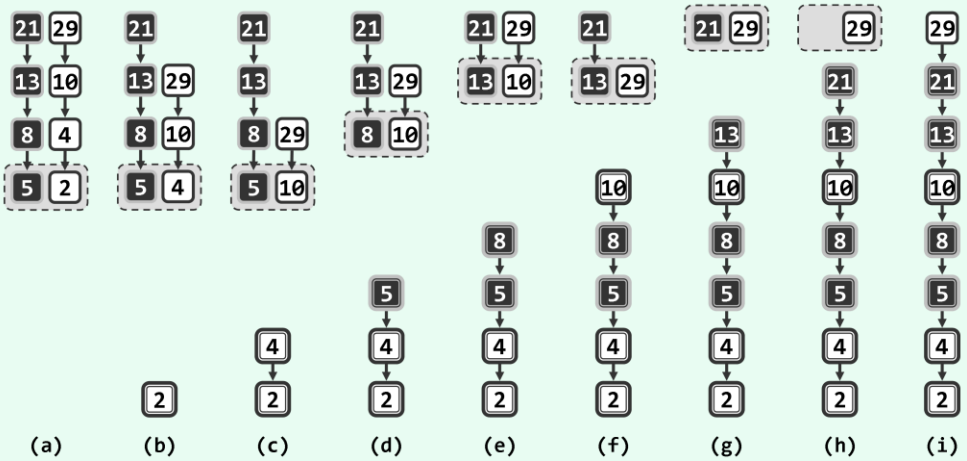

图2.18 有序向量的二路归并实例

（来自两个向量的元素分别以黑、白方框区分，其各自的当前首元素则以灰色长方形示意）

第一步迭代经比较，取出右侧向量首元素2并归入输出向量，同时其首元素更新为4（图(b)）。此后各步迭代均与此类似，都需比较首元素，将小者取出，并更新对应的首元素（图(c~h)）。如此，即可最终实现整体归并（图(i)）。

可见，二路归并算法在任何时刻只需载入两个向量的首元素，故除了归并输出的向量外，仅需要常数规模的辅助空间。另外，该算法始终严格地按顺序处理输入和输出向量，故特别适用于使用磁带机等顺序存储器的场合。

■ 分治策略

归并排序的主体结构属典型的分治策略，可递归地描述和实现如代码2.28所示。

```
1 template <typename T> //向量归并排序
2 void Vector<T>::mergeSort ( Rank lo, Rank hi ) { //0 <= lo < hi <= size
3    if ( hi - lo < 2 ) return; //单元素区间自然有序，否则...
4    int mi = ( lo + hi ) >> 1; mergeSort ( lo, mi ); mergeSort ( mi, hi ); //以中点为界分别排序
5    merge ( lo, mi, hi ); //归并
6 }
```

代码2.28 向量的归并排序

可见，为将向量S[lo, hi)转换为有序向量，可以均匀地将其划分为两个子向量：

S[lo, mi)

S[mi, hi)

以下，只要通过递归调用将二者分别转换为有序向量，即可借助以上的二路归并算法，得到与原向量S对应的整个有序向量。

请注意，这里的递归终止条件是当前向量长度：

n = hi - lo = 1

既然仅含单个元素的向量必然有序，这一处理分支自然也就可以作为递归基。

■ 实例

归并算法的一个完整实例，如图2.19所示。从递归的角度，也可将图2.19看作对该算法的递归跟踪，其中绘出了所有的递归实例，并按照递归调用关系将其排列成一个层次化结构。

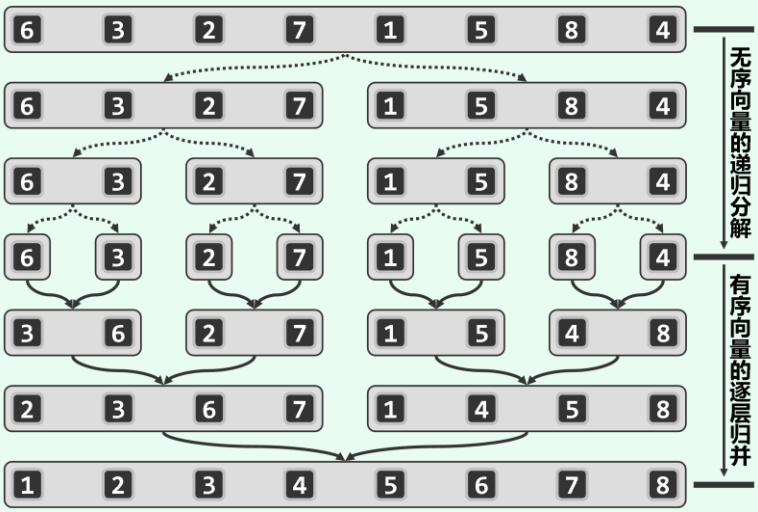

图2.19 归并排序实例：S = { 6, 3, 2, 7, 1, 5, 8, 4 }

可以看出，上半部分对应于递归的不断深入过程：不断地均匀划分（子）向量，直到其规模缩减至1从而抵达递归基。此后如图中下半部分所示，开始递归返回。通过反复调用二路归并算法，相邻且等长的子向量不断地捉对合并为规模更大的有序向量，直至最终得到整个有序向量。由此可见，归并排序可否实现、可否高效实现，关键在于二路归并算法。

■ 二路归并接口的实现

针对有序向量结构，代码2.29给出了二路归并算法的一种实现。

```
1  template <typename T> //有序向量的归并
2  void Vector<T>::merge ( Rank lo, Rank mi, Rank hi ) { //各自有序的子向量[lo, mi)和[mi, hi)
3     T* A = _elem + lo; //合并后的向量A[0, hi - lo) = _elem[lo, hi)
4     int lb = mi - lo; T* B = new T[lb]; //前子向量B[0, lb) = _elem[lo, mi)
5     for ( Rank i = 0; i < lb; B[i] = A[i++] ); //复制前子向量
6     int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = _elem[mi, hi)
7     for ( Rank i = 0, j = 0, k = 0; ( j < lb ) || ( k < lc ); ) { //B[j]和C[k]中的小者续至A末尾
8        if ( ( j < lb ) && ( ! ( k < lc ) || ( B[j] <= C[k] ) ) ) A[i++] = B[j++];
9        if ( ( k < lc ) && ( ! ( j < lb ) || ( C[k] <  B[j] ) ) ) A[i++] = C[k++];
10    }
11    delete [] B; //释放临时空间B
12 } //归并后得到完整的有序向量[lo, hi)
```

代码2.29 有序向量的二路归并

这里约定，参与归并的子向量在原向量中总是前、后相邻的，故借助三个入口参数即可界定其范围[lo, mi)和[mi, hi)。另外，为保证归并所得的子向量能够原地保存以便继续参与更高层的归并，这里使用了临时数组B[]存放前一向量[lo, mi)的副本（习题[2-28]）。

■ 归并时间

不难看出，以上二路归并算法merge()的渐进时间成本，取决于其中循环迭代的总次数。

实际上，每经过一次迭代，B[j]和C[k]之间的小者都会被移出并接至A的末尾（习题[2-29]和[2-30]）。这意味着，每经过一次迭代，总和s = j + k都会加一。

考查这一总和s在迭代过程中的变化。初始时，有s = 0 + 0 = 0；而在迭代期间，始终有：

$$s < lb + lc = (mi - lo) + (hi - mi) = hi - lo$$

因此，迭代次数及所需时间均不超过$\mathcal{O}(hi - mi) = \mathcal{O}(n)$。

反之，按照算法的流程控制逻辑，无论子向量的内部元素组成及其相对大小如何，只有待到s = hi - lo时迭代方能终止。因此，该算法在最好情况下仍需$\Omega(n)$时间，概括而言应为$\Theta(n)$。

请注意，借助二路归并算法可在严格少于$\Omega(n\log n)$时间内完成排序的这一事实，与此前2.7.3节关于排序算法下界的结论并不矛盾——毕竟，这里的输入并非一组完全随机的元素，而是已经划分为各自有序的两组，故就总体而言已具有相当程度的有序性。

■ 推广

二路归并只需线性时间的结论，并不限于相邻且等长的子向量。实际上，即便子向量在物理空间上并非前后衔接，且长度相差悬殊，该算法也依然可行且仅需线性时间。

更重要地，正如我们在后面（82页代码3.22）将要看到的，这一算法框架也可应用于另一类典型的序列结构——列表——而且同样可以达到线性的时间效率。

■ 排序时间

那么，基于以上二路归并的线性算法，归并排序算法的时间复杂度又是多少呢？

不妨采用递推方程分析法，为此首先将归并排序算法处理长度为n的向量所需的时间记作T(n)。根据算法构思与流程，为对长度为n的向量归并排序，需递归地对长度各为n/2的两个子向量做归并排序，再花费线性时间做一次二路归并。如此，可得以下递推关系：

$$T(n) = 2 \times T(n/2) + \mathcal{O}(n)$$

另外，当子向量长度缩短到1时，递归即可终止并直接返回该向量。故有边界条件

$$T(1) = \mathcal{O}(1)$$

联立以上递推式，可以解得（习题[2-26]）：

$$T(n) = \mathcal{O}(n\log n)$$

也就是说，归并排序算法可在$\mathcal{O}(n\log n)$时间内对长度为n的向量完成排序。因二路归并算法的效率稳定在$\Theta(n)$，故更准确地讲，归并排序算法的时间复杂度应为$\Theta(n\log n)$。

实际上，利用图2.19中算法整个执行过程的递归跟踪图，也可殊途同归。为此只需如该图所示，按照规模大小将各递归实例分层排列。既然每次二路归并均只需线性时间，故同层的所有二路归并累计也只需线性时间（当然，这两个“线性”的含义不同：前者是指线性正比于一对待归并子向量长度之和，后者则是指线性正比于所有参与归并的子向量长度之和）。由图不难看出，原向量中每个元素在同一层次恰好出现一次，故同层递归实例所消耗时间之和应为$\Theta(n)$。另外，递归实例的规模以2为倍数按几何级数逐层变化，故共有$\Theta(\log_2 n)$层，共计$\Theta(n\log n)$时间。