目 录

第 1 章　绪论

1.1　对象

作为一门 OOP 的编程语言，C++ 的设计范式总是从类和对象起步。在本书的第一节，我们将以一个“本书所有数据结构和算法的基类”作为引入，讲解如何利用 C++20 中的模块特性设计类。下面是第一个程序，它是一个**模块接口文件**，功能上类似于旧标准 C++ 中的**头文件**。它定义了命名空间`dslab`中的一个`Object`类，作为本书中所有数据结构和算法的基类。

```
1  // Object.ixx
2  module;
3  #include <string>
4  #include <typeinfo>
5  export module Framework.Object;
6  export namespace dslab {
7  class Object {
8  public:
9      virtual std::string type_name() const {
10         return typeid(*this).name();
11     }
12 };
13 }
```

在旧标准 C++ 中，通常在头文件（.h）里写定义，而在源文件（.cpp）里写实现。C++20 里引入了**模块**（module）的概念，用模块接口文件（.ixx）代替头文件。模块的引入大大提高了大型 C++ 项目的编译速度，因为相同的模块只需要编译一次，而被不同源文件`#include`的同一个头文件会引入大量的重复编译工作。本书中的所有代码都使用模块。模块接口文件的后缀名（.ixx）是 MSVC 推荐的后缀名；如果您使用的是 GCC，需要开启-x c++ 选项来支持它；如果您使用的是 Clang，需要修改配置文件才能支持.ixx。您也可以简单地修改后缀名为普通的 C++ 文件名，如.cxx 或.cc。

在模块接口文件的开始，我们需要引入标准库。我们通过一行单独的`module;`进入全局模块环境，然后和往常那样，将需要使用的标准库`#include`到程序中。这种写法适用于版本较低的编译器。一些高版本的编译器允许用户采用`import std;`这样的方法一次性引入整个标准库，这是 C++23 标准中的内容。一次性引入整个标准库看起来很吓人，但在模块的支持下，它比旧标准 C++ 里单独`#include <vector>`更快。

接着，我们需要通过`export module`说明，当前模块接口文件导出的是什么

模块。和 Java 类似，模块名可以用点分隔来表示子模块。本书采用的模块命名规则和 Java 相同，即采用目录名 + 文件名的格式。

本书定义的所有内容定义在命名空间dslab下，因此，我们总是采用**export namespace** dslab的方式导出dslab中的所有内容。如果我们需要导出特定的类或函数，则需要将关键字**export**移动到对应的类或函数的前面。

在上述代码的命名空间内不再含有模块的特性。

我们使用**typeid**(***this**).name()方法来在运行时获取一个对象的类型名,这是因为 C++ 仍然没有加入反射（reflection）这个高级语言喜闻乐见的特性。这个方法没有标准的实现，MSVC 中会返回一个比较可读的结果，而 GCC 和 Clang 中则会返回不那么可读的结果。因此，我们后续实现派生于Object的数据结构和算法的时候，往往需要重写这个方法，将其改为更加可读的版本。最后，值得一提的是，作为 C++98 时代的遗留产物，**typeid**(***this**).name()返回的是 C 风格的字符串类型（**const char***），在本书中会尽可能避免使用 C 风格指针，因此将其隐式转换为 C++ 风格的std::string返回。

下面我们编写一个测试的源文件（.cpp），来导入刚才实现的Object类。

```
// ObjectTest.cpp
#include <iostream>
import Framework.Object;
using namespace std;
class Test : public dslab::Object {};
int main() {
    cout << (Test()).type_name() << endl;
    return 0;
}
```

在源文件（.cpp）中，可以通过import关键字来导入定义的模块。在这个测试文件中，我们使用了一个Test类继承此前定义的Object类，并查看Test类型临时变量的类型名。如前所述，这个程序在不同的编译器下会得到不同的输出结果。

类似于其他高级语言，我们可能希望import Framework;提供一次性引入模块Framework中所有子模块的功能。此时，我们可以建立一个新的模块接口文件，用来实现这个功能。

```
// Framework.ixx
export module Framework;
export import Framework.Object;
```

其中，混合使用**export** import表示，将子模块Framework.Object导入之后，作为亲模块Framework的一部分导出。未来我们增加其他的子模块，只需要在 Framework.ixx 里增加对应的一行，就可以让它一并作为Framework的一部分被导出。

1.2　数据结构

数据结构是什么？这是一个很多新手都不会思考的问题。当然，我们可以简单地把“数据结构”这个词拆分为**数据**和**结构**，即：数据结构是计算机中的数据元素以某种结构化的形式组成的集合。

您可能会觉得这种定义过于草率，或者和您在教材上看到的定义不同。这是因为计算机是一门工程学科，对于不涉及工程实现的问题，都不存在标准化的定义。比如，“数据结构”和“算法”，甚至“计算机”这样的基本概念，都不存在标准化的定义。一些教材会把算盘甚至算筹划归“计算机”的范畴，并把手工算法（如尺规作图，甚至按照菜谱烹饪食物）划归“算法”的范畴。这种概念和定义的争议在计算机领域广泛存在，它主要来自以下几个原因。

1. 为了叙述简便，有些概念会借用一个已经存在的专有名词，从而引发歧义。如**树**（tree）这个词在计算机领域就有常用但迥然不同的两个概念。图论中的“树”出现得比较早，但没有人愿意将工程界经常出现的“树”称为“有限有根有序有标号的树”——英文里这些词并不能缩写为“四有树”。
2. 研究人员各执一词，从而引发歧义。这个现象的典型例子是“计数时从 0 开始还是从 1 开始”。从 0 开始是有一定数学上的优越性的，可以避免一些公式出现刻意的“+1”余项；但从 1 开始计数更符合自然习惯。这个问题直接导致在有些问题（如“树的高度”）上，不同教材的说法不同。考生如果参考了错误的教材就会导致无辜的失分。
3. 随着计算机领域的快速发展，一些概念的含义会发生变化。如众所周知，**字节**（byte）表示 8 个二进制位；但在远古时代，不同计算机采用的“字节”定义互不相同，有些计算机甚至是十进制的，那个时候一个字节可能表示 2 个十进制位。
4. 受计算机科学家的意识形态影响，同一概念的用词有所不同。如“树上的上层邻接和下层邻接节点”这一概念，现在普遍使用的词是 parent 和 child；然而思想老旧的人可能还在用 father 和 son，进步主义者则可能会用 mother 和 daughter。
5. 计算机领域的大多数成果出自美国，而英语翻译为汉语时，不同译者可能采用不同的译法。如 robustness 有音译的**鲁棒性**和意译的**稳健性**，hash 有音译的**哈希**和意译的**散列**。
6. 从业人员为了销售产品或取得投资，存在滥用、炒作部分计算机概念的情况。如“人工智能”“大数据”“云计算”“区块链”“元宇宙”等概念是这个现象的重灾区。

本书在描述无标准化的定义时，将尽可能让您把握住概念的要点。正如：数据结构是**计算机**中的数据元素以某种结构化的形式组成的**集合**。

数据结构中的数据是存储在**计算机**中的数据。我们在解题的时候往往会在纸上画出数据结构的图形，这是为了让自己更好地理解数据在计算机中的组织方式。计算机中的数据和纸上的数据会有很多的不同。比如，计算机处理数据时存在高速缓存（Cache，参见《组成原理》），这会使算法的局部性对算法性能造成重大影响，而这是在纸上无法分析出的。

我们研究的计算机都是二进制计算机，这意味着数据结构的数据元素总是一个二进制数码串，程序会将这些二进制数据转换为有意义的数据类型，比如`char`、`int`、`double`或某个自定义的类。因为一台计算机存储的二进制串不能无限长，所以我们讨论整数数据结构或者浮点数数据结构的时候，其元素的真正取值范围并不会是数学上的 Z 或者 R，而是一个有限集合。这一特性经常被一些数学成绩优秀的学生所忽略，尤其当他们试图用数学方法完成一些证明时。另一个有限性，即数据结构规模（存放的元素个数）的有限性，则不那么容易被忽略。

通常情况下，一个数据结构中的数据元素具有相同的类型，比如，一个数据结构不能既存储`int`又存储`std::string`。一些情况下，我们可能希望元素具有多个可能的类型，此时可以选择 C 语言的`union`或者 C++ 的`std::variant`。更加特殊的情况下，我们可能希望元素是任意类型的，此时可以选择 C 语言的`void*`或者 C++ 的`std::any`。这种特殊的情况更多地被视为一种**编程技巧**，而非**数据结构**中研究的理论问题。

数据结构中的数据元素具有实质上的**互异性**。这是由于，计算机中不可能存在两个完全一样的数据：至少它们的地址不同。这是我们把数据结构定义为数据的**集合**的原因。这种互异性在我们设计算法的时候需要尤其注意，比如，如果我们将序列`[a1,b,c]`修改为了`[a2,b,c]`，其中`a1`和`a2`的值相同、地址不同，看起来好像修改前后这个序列本身没有什么区别，但实际上可能会引起重大的错误。因为我们在修改这个序列的过程中，不能保证没有外部的指针指向`a1`。如果修改之后，`a1`的内存被释放，外部指向它的指针就会变成悬垂指针，造成严重的后果。很多有经验的程序员也容易忽视这一点，在本书中您还将看到更加实际的例子来说明它。

现在，我们回到数据结构的实现中来。现在我们需要继承上一节中实现的`Object`，设计一个数据结构的基类。作为一个抽象的数据结构，我们需要从数据结构的定义中挖掘共性。

1. 数据结构总是存储相同的类型。对于支持泛型编程的 C++ 来说，我们可以使用模板类型作为数据结构中的元素类型。

2. 数据结构是有限多个元素组成的，因此任何数据结构都具有`size()`方法。在 C++ 中，表示规模（size）的类型是`size_t`，它通常是一个无符号整数类型，可能是`uint32_t`或者`uint64_t`。

于是，我们可以这样设计`DataStructure`类：

```
template <typename T>
class DataStructure : public Object {
public:
    virtual size_t size() const = 0;
    virtual bool empty() const {
        return size() == 0;
    }
};
```

其中，获取规模的方法`size`被设计成了纯虚函数（pure virtual function），这意味着它的子类必须要实现这个方法。和规模相关，我们支持判空方法`empty`。

可能有的读者会认为，数据结构作为数据元素的集合，它理应支持增加元素、删除元素、查找元素这样的方法，然而事实却并非如此。有一些数据结构，它们可能一旦建立之后就无法添加或删除元素，或者只能添加和删除特定的元素，即**写受限**。同样地，另一些数据结构可能内部对用户不透明，用户只被允许访问特定的元素，并不能在数据结构中自由地查找，即**读受限**。在本书的后续部分，您将看到写受限和读受限的具体例子。

1.3　算法

1.3.1　算法的定义

和数据结构经常同时出现的另一个名词是**算法**。算法通常指接受某些**输入**，在**有限**步骤内可以产生**输出**的计算机计算方法。输出和输入的关系可以理解为算法的功能。对于同一功能，可能存在多种算法，对于相同的输入，它们通过不同的步骤可以得到**等价**的输出。这里并不一定要求得到相同的输出，比如我们要求一个数的倍数，输入 a 的情况下，输出 $2a$ 和 $3a$ 都是正确的输出，在“倍数”的观点下，这两个输出等价。

通常，算法的定义除了上述的三个要素：输入、输出和有限性之外，还包括可行性和确定性。比如，算法中如果包含了“如果哥德巴赫猜想正确，则 …”，则在当前不满足可行性；如果包含了“任取一个 …”，则不满足确定性（对同一算法，同样的输入必须产生同样的输出）。不过，在计算机上用代码写成的算法，通常都具有可行性和确定性，所以我们一般不讨论它们。有限性可以通过白盒测试和黑盒测试评估。在设计“算法”的基类时，我们只考虑输入和输出。

正如我们所熟知的那样，C++ 有一个概念同样具有输入和输出：函数（function）。但是，直接用函数来表示一个算法并不 OOP，因此我们采用**仿函数**（functor）而不是普通的全局函数。仿函数是一种类，它重载了括号运算符`operator()`。仿函数对象可以像一个普通的函数一样被调用，并且可以被转换为`std::function`。和普通的全局函数相比，仿函数具有两方面的优势：

1. 仿函数可以有成员变量。比如，可以定义一个`m_count`来统计一个仿函数对象被调用的次数。而普通的全局函数则无法做到这一点，非`static`的变量会在函数结束后被释放，而`static`的变量又强制被全局共享。
2. 仿函数可以有成员函数（也称为方法）。当仿函数的功能非常复杂时，它可以将功能拆解为大量的成员函数。因为这些成员函数都处在仿函数内部，所以不会污染外部的命名空间，并且可以清楚地看到它们和仿函数之间的关系。必要的时候，还可以通过嵌套类显式地指明其中的关系。而普通的全局函数，则无法在函数内嵌套一个没有实现的子函数。

于是，我们用仿函数设计了算法类`Algorithm`：

```
template <typename OutputType, typename... InputTypes>
class Algorithm : public Object {
public:
    virtual OutputType operator()(InputTypes... inputs) = 0;
};
```

这个类只定义了一个纯虚的括号运算符重载。这里我们使用了可变参数模板（以“...”表示），以处理不同输入的算法。比如，一个输入两个整数、输出一个整数的算法可以被继承`Algorithm<int, int, int>`。

1.3.2　算法的评价

作为一种解决问题的方法，算法的评价是多维度的，它们构成了算法题的主流题型。即使是在算法设计题中，也经常有“对设计的算法进行评价”的附加要求。本节将简要介绍算法的几个典型的评价维度。

正确性。正确性检验通常分为两个方面：

1. **有限性检验**。如前所述，有限性是算法定义的组成部分之一。有限性检验，主要用于判断带有无限循环（如`while(true)`）或强制跳转（`goto`）的算法是否必定会终止。没有无限循环或强制跳转的情况，有限性是默认的。这一方面对应着算法定义中的**有限性**要求。
2. **结果正确性检验**。即验证输出的结果满足算法的需求。在算法有确定的正确结果时，这一检验是“非黑即白”的；而在算法没有确定的正确结果时，可能需要专用的检验程序甚至人工打分（比如较早的象棋 AI，往往是以高手对

一些局面的形势打分作为基础训练数据的）。这一方面对应着算法定义中的**输入**和**输出**。

算法定义中的另外两点，**可行性**和**确定性**，正如我们之前所讨论的那样，通常都可以被直接默认，而不需要进行检验。

效率。评价算法效率的标准可以简单地概括为多、快、好、省。在《数据结构》中，通常只研究“快”和“省”这两个方面，而《网络原理》则需要考虑全部的四个方面。在《网络原理》中，“多”代表网络流量，“好”代表网络质量。

1. **时间效率**（快）。在计算机上运行算法一定会消耗时间，时间效率高的算法消耗的时间比较短。
2. **空间效率**（省）。在计算机上运行算法一定会消耗空间（硬件资源），空间效率高的算法消耗的硬件资源比较少。

在不同的计算机、不同的操作系统、不同的编程语言实现下，同一算法消耗的时间和空间可能大相径庭。为了抵消这些变量对算法效率评价的干扰作用，在《数据结构》这门学科里进行算法评价时，往往不那么注重真实的时间、空间消耗，而倾向于做**复杂度分析**。关于复杂度的讨论参见后文。

稳健性（robustness，又译健壮性、鲁棒性；在看到英文之前，笔者曾一度认为“鲁棒性”这个词来源于山东大汉身体棒的地域刻板印象）。即算法面对意料之外的输入的能力。

泛用性。即算法是否能很方便地用于设计目的之外的其他场合。

在上述 4 个评价维度中，正确性和效率是《数据结构》学科研究的主要内容，算法评价题也总是围绕正确性检验和复杂度分析命题；这两个问题留到后面的小节里展开讨论。而稳健性和泛用性，则在课程设置上属于《软件工程》讨论的内容，在下一节会用一个实验展示它，后续不再赘述。

1.3.3　实验：从 1 加到 N

算法和实现它的代码（code）或程序（program）有本质区别。按照通常意义的划分，算法更接近于理科的范畴，而实现它的代码更接近工科的范畴。一些同学可能很擅长设计出精妙绝伦的算法，但需要耗费巨大的精力才能实现它，并遗留不计其数的 Bug；另一些同学可能在设计算法上感到举步维艰，但如果拿到已有的设计方案，可以轻松完成一份漂亮的代码。算法设计和工程实现对于计算机学科的研究同等重要。参加过语文高考的学生可能会很有感受：自己有一个绝妙的构思，但没有办法在有限的时间下把它说清楚。专精算法设计而忽略工程实现的同学会有类似的感觉。

上一节中介绍的算法评价维度中，**稳健性**和**泛用性**是高度依赖于算法的实现

的（当然，也有少数情况和算法本身的设计相关）。在本节将通过一个实验作为例子，向读者展示：对于相同的算法，代码实现的不同会影响这两个评价维度。本节的实验可以在 *Sum.cpp* 中找到。

我们讨论一个非常简单的例子：从 1 加到 n 的求和。输入一个正整数 n，输出 $1+2+\cdots+n$ 的和。我们从这个例子出发，介绍本书使用的实验框架。首先，我们定义本问题的基类：

```
class Sum : public Algorithm<int, int> {};
```

因为这个算法不需要用到额外的成员变量或成员函数，所以它简单地继承了我们定义的算法类`Algorithm<int, int>`。我们也可以使用别名（alias）技术来指定`Sum`和`Algorithm<int, int>`是同一个类：

```
using Sum = Algorithm<int, int>;
```

相对于 C 语言的**`typedef`**，使用关键字**`using`**的别名技术更加清晰，并且支持带模板参数的语法。当我们使用 C++ 编程时，应当使用别名替代**`typedef`**。

现在我们回到求和问题上来。作为实验的一部分，您可以自己实现一个类，继承`Sum`，并重写**`operator()`**，来和笔者提供的示例程序做对比。这一过程可以发生在您阅读下面对示例程序的解析之前，也可以发生在之后。在这个实验的示例程序里，笔者设计了几个`Sum`的派生类来完成这个算法功能。

首先，我们可以简单地把每个数字加起来，就像这样：

```
// SumBasic
int operator()(int n) override {
    int sum { 0 };
    for (int i { 1 }; i <= n; ++i) {
        sum += i;
    }
    return sum;
}
```

使用一对大括号（而不是等于号）对数据进行初始化，称为统一初始化，这是现代 C++ 建议的初始化方式。它有助于提示用户这里发生了初始化（调用了构造函数），而不是发生了赋值（调用了**`operator=`**）。另一方面，统一初始化有更加严格的类型检查，这可以防止某些意料之外的隐式类型转换。

此外，如果不需要使用返回值，则建议使用前置运算符`++i`而不是后置运算符`i++`。尽管当`i`是整数的时候编译器往往会自动进行优化，可当`i`是自定义类型时不一定会这样。

另一方面，`override`关键字用于提示编译器该方法是重载方法。永远要记得为您的重载方法添加这个关键字，以避免由于细小的区别（如遗漏**`const`**）而不小

心定义了一个新方法。

上面的这个算法显然称不上好。著名科学家高斯在很小的时候就发现了等差数列求和的一般公式。我们可以使用公式，得到另一个可行的算法。

```
// SumAP
int operator()(int n) override {
    return n * (n + 1) / 2;
}
```

由于这个算法的正确性十分显然，您或许觉得这个程序毫无问题，直到您发现了另外一个程序：

```
// SumAP2
int operator()(int n) override {
    if (n % 2 == 0) {
        return n / 2 * (n + 1);
    } else {
        return (n + 1) / 2 * n;
    }
}
```

通过对比 SumAP 和 SumAP2，您会立刻意识到 SumAP 存在的问题：对于某个区间内的 n，$\frac{n(n+1)}{2}$ 的值不会超过`int`的最大值，但 $n(n+1)$ 会超过这个值。比如，当 $n = 50,000$ 的时候，SumAP2 和 SumBasic 都能输出正确的结果，而算法 SumAP 则会因为数据溢出返回一个负数。*您可以算出使 SumAP 错误而 SumAP2 仍然正确的区间。*

但 SumAP2 也很难称之为无可挑剔，比如说，如果 n 更大一些，比如取 $100,000$，则它也无法输出一个正确的值。这种情况下，甚至最朴素的 SumBasic 也无法输出正确的值。

那么，我们思考一个问题：上述的三个实现，哪些是**正确**的？

在我们实际进行代码实现的时候，通常会倾向于 SumAP2，因为它既有较高的效率（相对于 SumBasic 而言），又保证了在数据不溢出的情况下能输出正确的结果（相对于 SumAP 而言）。但在我们进行算法评估的时候，通常认为这三个实现都是正确的，并且 SumAP 和 SumAP2 实质上是**同一种**算法。也就是说，数据溢出这种问题并不在我们的评估模型之内，算法虽然执行在计算机上，但又是**独立**于计算机的；算法虽然需要代码去实现，但又是**独立**于实现它的代码的。在《数据结构》这门学科中的研究对象，通常和体系结构、操作系统、编程语言等因素都没有关系。

在本节的末尾，提出一个有趣的问题：SumAP2 在 n 非常大的时候也会出现溢出问题。当然，这超过了`int`所能表示的上限。但是，这种情况下，返回什么样的

值是合理的？SumAP2 返回的值（有可能是一个负数）真的合理吗？这是纯粹的工程问题，并不在《数据结构》研究的范围内。

一种可能的方案是，在溢出的时候返回`std::numeric_limits<int>::max()`，这是 C++ 中用来替代 C 语言宏`INT32_MAX`的方法。这种方案称为“饱和”。饱和保证了数值不会因为溢出而变为负值，当数值具有实际意义时，一个不知所云的负值可能会引发连锁的负面反应。比如，路由器可能会认为两个节点之间的距离为负（事实上应当是无穷大），从而完全错误地计算路由。因此，如果实现了饱和，在泛用性上可以得到一定的提升。

还有一个问题是，如果 n 为负数，则应该如何输出？当然，我们的题目要求 n 是正整数，负数是非法输入；但我们也希望程序能输出一个有意义的值。按照朴素的想法（也就是 SumBasic），这个时候应该输出 0，然而 SumAP 和 SumAP2 都做不到这一点。在示例程序中给出了一个考虑了饱和和负数输入的实现，您也可以独立尝试实现这个功能。

1.4　正确性检验

在上一节已经说明，正确性检验可以拆解为两个方面：有限性检验和结果正确性检验。在实际解题的过程中，这两项检验往往可以一并完成，即检验算法是否能在有限时间内输出正确结果。

解决正确性检验的一般方法是**递降法**。它的思想基础是在计算机领域至关重要、并且是《数据结构》学科核心的**递归**思维方法；它的理论依据则是作为在整数公理系统中举足轻重的**数学归纳法**。

在本节中，将从数学归纳法的角度出发介绍递降法的原理，这部分有助于让您对递归思维有更加深刻的理解。当然在实际考试中只要会用递降法解题即可，您也可以跳过数学的部分（如果您对数学感到头疼，这是建议的选项）。

1.4.1　经典的归纳和递降

在高中理科数学中介绍了数学归纳法的经典形式。由于各省教材不同，您可能接触到过两种表述不太一样的数学归纳法：

定理 1.1 (第一归纳法)： 令 $P(n)$ 是一个关于正整数 n 的命题，若满足：

1. $P(1)$。
2. $\forall n \geqslant 1, P(n) \to P(n+1)$。

则 $\forall n, P(n)$。

定理 1.2 (第二归纳法)： 令 $P(n)$ 是一个关于正整数 n 的命题，若满足：

1. $P(1)$。

2. $\forall n \geqslant 1, (\forall k \leqslant n, P(k)) \rightarrow P(n+1)$。

则 $\forall n, P(n)$。

其中，第一归纳法是皮亚诺（Piano）公理体系的一部分，您可以很轻松地用第一归纳法推导出第二归纳法。另一方面，第二归纳法的**归纳假设** $\forall k \leqslant n, P(k)$，显然比第一归纳法的归纳假设 $P(n)$ 更强，所以实际应用数学归纳法进行证明时，通常都使用第二归纳法，而不使用第一归纳法。

在上面的表述中，归纳的过程是从 *1* 开始的，这比较符合数学研究者的工作习惯。在计算机领域，归纳法常常从 *0* 开始（有时甚至从*-1* 开始）。显然，这并不影响它的正确性。本节后续对“正整数”相关问题的讨论，一般替换成“自然数”也同样可用。

我们将第二归纳法称为**经典归纳法**。经典归纳法可以用来处理有关正整数的命题。相应地，对于输入是正整数的算法，可以使用与经典归纳法相对应的**经典递降法**。

定理 1.3 (经典递降法)： 令 $A(n)$ 是一个输入正整数 n 的算法，若满足：

1. $A(1)$ 可在有限时间输出正确结果。

2. $\forall n > 1, A(n)$ 可在有限时间正确地将问题化归为有限个 $A(k_j)$，其中 $k_j < n$。

则 $\forall n, A(n)$ 可在有限时间输出正确结果。

经典递降法的正确性由经典归纳法保证，您可以自己证明这一点。显然，经典递降法的应用范围非常狭小：只能用来处理输入是正整数的算法。对于实际的算法，它的输入数据通常是多个数、乃至于数组和各种数据结构，而非孤零零的一个正整数。因此，需要对经典归纳法进行推广，从而使其可以应用到更广的范围中，并使其相对应的递降法能够处理更加多样的输入数据。

1.4.2　带映射的归纳和递降

在《线性代数》的第一章“行列式”中，您一定见过下面这个经典的命题：

$$D_n \begin{vmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_1^{n-1} & x_2^{n-2} & \cdots & x_n^{n-1} \end{vmatrix} = \prod_{1 \leqslant j < i \leqslant n} (x_i - x_j)$$

通常这个命题都是用数学归纳法证明的，您不会有任何违和感。

但在这个命题中，n 并不是唯一的变量。在 n 之外，还有 $x_1, x_2, \cdots, x_n$ 这 n 个变量。换句话说，这个命题所接受的是一个任意有限长的向量 $(x_1, x_2, \cdots, x_n)$。在

使用数学归纳法进行证明时，实际上指定了一个映射 $f(x_1, x_2, \cdots, x_n) = n$，输入数据是这个映射的原象，而在这个映射的象上做数学归纳法。有些数学书为了指明这一点，会在证明的开头写上“对行列式的阶数 n 做归纳”；而有些书则图省事略去了这句话。

在这个例子中，通过“对行列式的阶数做归纳”，将复杂的输入数据映射到了正整数集，这是一种典型的应用归纳法的技术。类似地，在证明对整数的命题时，可以对它的绝对值做归纳，等等。一般而言，可以将经典归纳法改写成下面的“带映射的”形式：

定理 1.4 (带映射的经典归纳法)： 令 $P(x)$ 是一个关于 $x \in X$ 的命题，$f : X \to \mathrm{N}^+$ 是满射，若满足：

1. $f(x) = 1 \to P(x)$。
2. $[\forall y, f(y) < f(x) \to P(y)] \to P(x)$。

则 $\forall x, P(x)$。

可以对命题 $Q(n) = (\forall x, f(x) = n \to P(x))$ 使用经典的归纳法来证明上面这个带映射的形式。相应地，也存在带映射的经典递降法。

定理 1.5 (带映射的经典递降法)： 令 $A(x)$ 是一个输入 $x \in X$ 的算法，$f : X \to \mathrm{N}^+$ 是满射，若满足：

1. 当 $f(x) = 1$ 时，$A(x)$ 可在有限时间输出正确结果。
2. $\forall x, f(x) > 1$，$A(x)$ 可在有限时间正确地将问题化归为有限个 $A(y_j)$，其中 $f(y_j) < f(x)$。

则 $\forall x, A(x)$ 可在有限时间输出正确结果。

为了让上述定义的每个 $Q(n)$ 都存在，f 需要保证是满射。这又是一项对递降法应用范围的限制，需要想办法清除掉它，得到更加一般的、更加通用的解题方法。

1.4.3　良序关系

在带映射的归纳法中，需要通过满射将定义域 X 映射到正整数集 N^+ 上。很多时候，这样的满射并不容易构造。与其试图用高超的技巧去构造满射，不如从正整数集 N^+ 入手：放宽映射的象的条件，不要求它一定是 N^+，只需要满足一些和 N^+ 相似的性质即可。

对于一般的集合，引入**良序**（well-order）的概念。如果集合 S 上的一个关系 $\preceq$ 满足：

1. **完全性**。$x \preceq y$ 和 $y \preceq x$ 至少有一个成立。
2. **传递性**。如果 $x \preceq y$ 且 $y \preceq z$，那么 $x \preceq z$。

3. **反对称性**。如果 $x \preceq y$ 和 $y \preceq x$ 均成立，那么 $x = y$。
4. **最小值**。对于集合 S 的任意非空子集 A，存在最小值 $\min\ A = x$。即对于 A 中的其他元素 y，总有 $x \preceq y$。

那么称 $\preceq$ 是 S 上的一个**良序关系**，同时称 S 为**良序集**。类似于数值的小于等于“$\leqslant$”和小于“$<$”关系，对于关系 $\preceq$，如果 $x \preceq y$ 且 $x \neq y$，则可以记为 $x \prec y$。

根据上述定义，熟知的小于等于关系“$\leqslant$”在正整数集 N^+ 上是良序的，而在整数集 Z 上不是良序的。当然，可以通过定义“绝对值小于等于”让整数集 Z 成为良序集。同时，熟知的小于等于关系“$\leqslant$”在非负实数集 $\mathrm{R}^+ \cup \{0\}$ 上不是良序的，因为它不满足最小值条件（任取一个开区间作为它的子集，都没有最小值）。

和熟知的正整数集 N^+ 相比，一般的良序集具有下面的相似性质：

定理 1.6 (无穷递降)： 在良序集 S 中，不存在无穷序列 $\{x_n\}$，使得 $x_{j+1} \prec x_j$ 对一切 j 成立。

您可以用反证法证明它。这一性质使得在一般的良序集上做归纳法成为可能，后面的小节会回到这个问题上来。

1.4.4　字典序

在上一小节，您发现熟知的小于等于关系在非负实数集 $\mathrm{R}^+ \cup \{0\}$ 上并不是良序的。如果您感到不服气，想要尝试去构造 R 上的良序关系，几乎一定会无功而返（目前还没有数学家定义出实数集 R 上的显式良序关系）。然而我们有下面的定理：

定理 1.7 (良序定理)： （ZFC）任何集合都存在良序关系。

在集合论的 ZFC 公理体系下，上述定理成立。在 ZF 公理体系下，该定理和**选择公理**（AC）等价。选择公理是有些反直观的（有兴趣的话可以自行搜索），与它等价的良序定理同样反直观，基本上只能用于理论推导，很难实际应用。

幸运的是，在计算机领域的日常研究中，并不需要使用无所不能的良序定理来“设”出一个良序关系。计算机领域中，输入数据所属的集合总是**可数的**（countable），而可数集合总是可以很轻松地定义良序关系。其中一个典型的例子是所谓的**字典序**（lexicographical order）。

定理 1.8 (字典序)： 设 $\{S_n\}$ 是一个良序集序列，$\preceq_j$ 是集合 S_j 上的一个良序关系。则对于无限笛卡尔积 $\prod S_j = S_1 \times S_2 \times S_3 \times \cdots$ 中的两个元素 $a = (a_1, a_2, a_3, \cdots)$ 和 $b = (b_1, b_2, b_3, \cdots)$，定义 $a \preceq b$ 当且仅当：存在某个 k，使得 $a_j = b_j$ 对于 $1 \leq j < k$ 成立，但 $a_k \preceq_k b_k$。那么 $\preceq$ 是 $\prod S_j$ 上的一个良序关系。

上面定义的良序关系，是针对无限长向量的。将它稍微修改一下，就可以用来定义任意长向量。下面给出了一种证明方法，您也可以自己解决这个问题。在每个集合 S_j 中增加一个元素 $\varnothing_j$，让这个元素作为 $S_j \cup \{\varnothing_j\}$ 的最小值。对于非无限长

的向量 $(a_1, a_2, \cdots, a_n)$，将其延伸为 $(a_1, a_2 \cdots, a_n, \varnothing_{n+1}, \varnothing_{n+2}, \cdots)$，就得到了无限长向量。这个映射是一一对应的，从而可以用无限长向量的字典序去定义任意长向量的字典序。特别地，由 $S_j = \{a, b, \cdots, z\}$ 构成的任意长向量的字典序，就是英文字典中排列单词的顺序。

在本节的最后需要说明，实际上计算机领域的输入数据所属集合总是有限的，因为任何硬件设备都存在可承载的数据量上限。但是，在不能用枚举法的情况下，通常都会选择将输入集合从有限集扩大为可数集，从而使用针对可数集的递降法。

1.4.5　超限的归纳和递降

在定义良序关系之后，就可以使用**超限归纳法**来证明命题。超限归纳法本质上是经典数学归纳法在集合论上的一般形式，可由良序关系的定义直接导出，而无需用到良序定理。

定理 1.9 (超限归纳法)： 令 $P(x)$ 是一个关于 $x \in X$ 的命题，$f : X \to S$ 是满射，S 是关于 $\preceq$ 的良序集，若满足：

1. $f(x) = \min S \to P(x)$。
2. $[\forall y, f(y) \prec f(x) \to P(y)] \to P(x)$。

则 $\forall x, P(x)$。

上面的表述和经典的超限归纳法表述有所不同。它融合了之前介绍过的“映射”策略。当 S 取为正整数集 N^+，良序关系取为熟知的“$\leqslant$”时，超限归纳法的特例就是经典归纳法。这是最自然的良序关系和良序集。在考试解题过程中，通常都只需要用到 N^+ 或它的子集。尽管解题的时候，通常只会用到通常良序集，但“良序”的思维，仍然广泛存在于计算机领域的各个学科中。

和超限归纳法对应，可以得到一般形式的递降法。这是我们解决各种算法分析问题的基本方法。

定理 1.10 (递降法)： 令 $A(x)$ 是一个输入 $x \in X$ 的算法，$f : X \to S$ 是满射，S 是关于 $\preceq$ 的良序集，若满足：

1. 当 $f(x) = \min S$ 时，$A(x)$ 可在有限时间输出正确结果。
2. $\forall x, f(x) \neq \min S$，$A(x)$ 可在有限时间正确地将问题化归为有限个 $A(y_j)$，其中 $f(y_j) \prec f(x)$。

则 $\forall x, A(x)$ 可在有限时间输出正确结果。

递降法和**递归**（recursion）是密不可分的。在递降法中，条件（1）对应了**递归边界**，条件（2）则对应了**递归调用**。边界情况通常对应的是最简单的情况，而递归调用则用来将复杂问题拆解成简单问题。只要您有一定的递归编程的经验，那么递降法是非常容易理解的。

递降用于分析算法，而递归用于设计算法，二者思路上相似，只是功能上不同。由于递降法用到的计算机思维事实上是递归，我们将不再区分“递降法”和“递归法”。递归（递降）法是《数据结构》学科最为核心的思维方法，在这一章只是用简单的例子介绍它，后面的章节中还会反复出现，并不断增加问题的难度和思维的深度。

下面的几个小节将用几个实际的例子，来说明递降法在正确性检验中的作用。

1.4.6　实验：最大公因数

本节以求最大公因数为例，展示如何证明递归算法的正确性。相关代码可以在 *Gcd.cpp* 中找到。关于这个问题，如果您有一定编程基础，肯定能一眼就知道如何解决，不妨尝试一下。

如果您没有编程基础，可以参考下面的实现，这是大名鼎鼎的最大公因数算法：欧几里得（Euclid）辗转相除法的递归形式。

```
// GcdEuclid
int operator()(int a, int b) override {
    if (b == 0) {
        return a;
    } else {
        return (*this)(b, a % b);
    }
}
```

在这个算法中，递归边界是 $(a, 0)$，因此可以定义 $f(a, b) = \min(a, b)$，将输入数据映射到通常的良序集 N。接着就可以用递降法处理这个问题了。

1. 如果 $a < b$，那么通过 1 次递归，可变换为等价的 $\gcd(b, a)$。
2. 如果 $a \geqslant b > 0$，那么由于 $b > a\%b$ 对一切正整数 a, b 成立，所以通过 1 次递归，可变换为 $f(\cdot)$ 更小的 $(b, a\%b)$。接下来只要证 $\gcd(a, b) = \gcd(b, a\%b)$。您可以自己完成这一证明。下面提供了一种比较简单的证法。

   设 $a = kb + l$，其中 $l = a\%b$。那么，对于 a 和 b 的公因数 d，设 $a = Ad$，$b = Bd$，则 $l = (A - kB)d$。因此 d 也是 b 和 l 的公因数。反之，对于 b 和 l 的公因数 d’，也可推出 d’ 也是 a 和 b 的公因数。因此 a 和 b 的公因数集合，与 b 和 l 的公因数集合相同；它们的最大值（即最大公因数）显然也相同。
3. 如果 $b = 0$，到达边界，$\gcd(a, 0) = a$ 正确。

如此便完成了 Euclid 辗转相除法的正确性证明。

1.4.7　实验：数组求和

在递降法中允许递归函数调用自身有限次。在上一节中，一个 gcd 只会调用一次自身，这种直接将问题转化成更简单问题的思想称为**化归**（transformation）。这一小节展示了一个多次调用自身的例子。

作为第一章的实验，我们仍然讨论一个简单的求和问题：给定一个规模为 n 的数组 A，求 $A[0]+A[1]+\cdots+A[n-1]$ 的和。相关代码可以在 *ArraySum.cpp* 中找到。

```
class ArraySum : public Algorithm<int, span<const int>> {};
```

您可以将它改为您更熟悉的版本，比如 C 风格用指针`int*`和长度`size_t`定义的数组，或者使用 C++ 的变长数组`std::vector`并结合迭代器。这里我们使用 C++20 引入的`std::span`来表示数组的视图，它既可以处理`int*`定义的数组，又可以处理`std::vector`。

经典的数组求和方法，是定义一个累加器，把每个数逐一加到累加器上。这个算法在 C++ 中被封装成了`std::accumulate`：

```
// ArraySumBasic
int operator()(span<const int> data) override {
    return accumulate(begin(data), end(data), 0);
}
```

在 C++17 中，引入了一个新的累加算法`std::reduce`，它允许乱序计算，从而给了硬件技术（如 SIMD，参见《组成原理》）更高的优化空间。`std::reduce`的语法和`std::accumulate`相似。

```
// ArraySumReduce
int operator()(span<const int> data) override {
    return reduce(begin(data), end(data), 0);
}
```

在这个例子中，如果 n 很大，ArraySumReduce 和 ArraySumBasic 之间可以有好几倍的时间消耗差异（您可以运行参考程序来看到这一点），因此在允许乱序求和的时候，总是应当采用`std::reduce`。

现在回到我们的递归话题来，我们可以设计这样的算法：先算出数组前一半的和，再算出数组后一半的和，最后把这两部分的和相加（这个算法成立的基础是加法结合律）。这是一个递归算法，如下所示。

```
// ArraySumDivideAndConquer
int operator()(span<const int> data) override {
    if (data.size() == 0) {
        return 0;
    } else if (data.size() == 1) {
        return data[0];
    } else {
```

```
8           auto mid { data.size() / 2 };
9           return (*this)(data.first(mid)) + (*this)(data.last(data.
                size() - mid));
10      }
11  }
```

您可以选取合适的映射，将数据范围从数组映射到通常的良序集上，并完成正确性的证明。一个显然的映射是 $f(A) = A.\text{size}()$。这种将数据结构（这里是数组）分拆成几个部分，用得到的部分结果“拼出”整体结果的思想称为**分治**（divide-and-conquer）。

我们注意到，分治算法和普通算法相比，只是修改了加法的运算次序。我们不期待它能达到`std::reduce`的性能，但测试会发现，它竟然连基本算法的性能都远远不如。这是因为*递归调用*本身存在不小的开销。如果等价的迭代方法（在后续章节中，将讲解递归和迭代的相互转换）并不复杂，那么通常用迭代替换递归，以免除递归调用本身的性能开销。

1.4.8　实验：函数零点

以上两个例子都是建立在递归上的算法。很多算法可能并不包含递归；对这些算法做有限性检验，不是要排除无穷递归，而是要排除无限循环。下面展示了一个循环的例子，代码可以在 *ZeroPoint.cpp* 中找到。

我们考虑一个函数的零点，给定一个函数 $f(x)$ 和区间 (l,r)，保证 $f(x)$ 在 (l,r) 上连续，并且 $f(l)\cdot f(r) < 0$。根据*介值定理*，我们知道 $f(x)$ 在 (l,r) 上存在至少一个零点。由于计算机中的浮点数计算有精度限制，我们只需要保证绝对误差不超过给定的误差限 ϵ。为了简单起见，我们现在统一给定 $l = -1, r = 1, \epsilon = 10^{-6}$。

```
1   class ZeroPoint : public Algorithm<double, function<double(
        double)>> {
2       static constexpr double limit_l { -1.0 };
3       static constexpr double limit_r { 1.0 };
4   protected:
5       static constexpr double eps { 1e-6 };
6       virtual double apply(function<double(double)> f, double l,
            double r) = 0;
7   public:
8       double operator()(function<double(double)> f) override {
9           return apply(f, limit_l, limit_r);
10      }
11  };
```

上面的例子展示了我们将`Algorithm`定义为仿函数的优势，我们可以在零点问题的基类中定义私有（private）成员来表示给定的初值 l 和 r；另一个给定的初值 ϵ 在算法的实现中需要用到，则可以定义为受保护的（protected）成员。

函数的零点可以通过二**分**的方法取得，这个方法在高中的数学课程中介绍过。如果您熟悉编程，应该可以自己实现一个版本。

```
// ZeroPointIterative
double apply(function<double(double)> f, double l, double r)
    override {
    while (r - l > eps) {
        double mid { l + (r - l) / 2 };
        if (f(l) * f(mid) <= 0) {
            r = mid;
        } else {
            l = mid;
        }
    }
    return l;
}
```

尽管是经典的算法，但您仍然可以关注其中的一些细节。一个细节是我们定义mid为 $l+\frac{r-l}{2}$ 而不是 $\frac{l+r}{2}$。这两种写法都是推荐的写法。在整数的情况下，前者的优点是不会溢出，并保证结果接近 l（后者存在负数时无法保证，而这一点在写二分算法时有时很致命）；后者的优点是少一次减法计算。浮点数的情况比整数简单很多，两种写法，甚至 $\frac{l}{2}+\frac{r}{2}$（在整数的情况下不应该使用）都是可以的。

另一个细节是我们的判断条件为 $f(l)\cdot f(\text{mid})\leqslant 0$，而没有用严格小于号。您可以自己思考，若改为严格小于，会发生什么样的变化？如果一时没有头绪，您可以实现这样的一个类，拿来和示例代码一起测试。作为一门工程学科，计算机学科非常需要您通过实验建立起来的经验。

分析循环问题的手段，和分析递归问题是相似的。递归函数的参数，在循环问题里就变成了循环变量。在处理上面的迭代算法时，首先找到循环的停止条件：$r-l<\epsilon$。这个条件里，ϵ 作为输入数据，在循环中是不变量，而 l 和 r 是循环中的变量。因此，使用递降法的时候可以将 ϵ 看成常量，而 l 和 r 作为递归参数。

定义映射 $f(l,r)=\lfloor\frac{r-l}{\epsilon}\rfloor$ 就可以将问题映射到通常良序集，后面的做法和前面几个例子基本相同，您可以自己完成正确性检验。请注意在证明结果正确性时要留意 $f(\text{mid})=0$ 的情况。从这个映射中，我们可以发现规定 ϵ 是必要的，即使不考虑**double**的位宽限制，计算机也无法保证在有限时间里找到精确解，只能保证找到满足精度条件的近似解。

这种“将循环视为递归”然后用递降法处理的方法，等价于将上面的迭代算法改写为以下与其等价的递归算法。

```
// ZeroPointRecursive
double apply(function<double(double)> f, double l, double r)
    override {
    if (r - l <= eps) {
```

```
4         return l;
5     } else {
6         double mid { l + (r - l) / 2 };
7         if (f(l) * f(mid) <= 0) {
8             return apply(f, l, mid);
9         } else {
10            return apply(f, mid, r);
11        }
12    }
13 }
```

其通用做法是：找到循环的停止条件，然后将条件中出现的、在循环内部被改变的变量视为递归的参数，以此将循环改写为递归。当然，实际解题的时候犯不着费劲改写成递归再分析，用这个思路直接分析循环就可以了。

1.5　复杂度分析

复杂度（complexity）**分析**的技术被用于评价一个算法的效率。在考试中它出现的频率比正确性检验更高。在上文中提到过，一个算法的真实效率（运行时间、占用的硬件资源）会受到所用计算机、操作系统以及其他条件的影响，因此无法用来直接进行比较。因此，进行复杂度分析时通常不讨论绝对的时间（空间）规模，而是采用**渐进复杂度**来表示其大致的增长速度。

1.5.1　复杂度记号的定义

以下给出复杂度记号的标准定义。

假设在问题规模为 n 的情况下，算法在某一计算机上执行的绝对时间单元（空间单元）的数量为 $T(n)$。

1. 对充分大的 n，如果 $T(n) \leqslant C \cdot f(n)$，其中 $C > 0$ 是和 n 无关的常数，那么记 $T(n) = O(f(n))$。
2. 对充分大的 n，如果 $C_1 \cdot f(n) \leqslant T(n) \leqslant C_2 \cdot f(n)$，其中 $C_2 \geqslant C_1 > 0$ 是和 n 无关的常数，那么记 $T(n) = \Theta(f(n))$。
3. 对充分大的 n，如果 $C \cdot f(n) \leqslant T(n)$，其中 $C > 0$ 是和 n 无关的常数，那么记 $T(n) = \Omega(f(n))$。

用 $O(\cdot)$、$\Theta(\cdot)$ 和 $\Omega(\cdot)$ 记号表示的时间（空间）随输入数据规模的增长速度称为**渐进复杂度**，或简称**复杂度**。

从上述定义中可以得到，当 $T(n)$ 关于 n 单调递增并趋于无穷大时，$f(n)$ 是阶不比它低的无穷大量，$h(n)$ 是阶不比它高的无穷大量，而 $g(n)$ 是和它同阶的无穷大量。当然 $T(n)$ 并不一定单调递增趋于无穷大。一般而言，复杂度记号是在问题

规模充分大的前提下，从增长速度的角度对算法效率的定性评价。

有些教材为图省事，只介绍了 $O(\cdot)$ 一个符号，这非常容易引起理解错误或混淆。在下一小节中会展示很多错误理解的例子。因此，即使您准备参加的考试中不要求另外两种复杂度记号，也希望您理解这些记号。在本书中，这三种复杂度记号都会使用。其中，$\Theta(\cdot)$ 记号包含了比 $O(\cdot)$ 更多的信息，如果您准备参加的考试只要求 $O(\cdot)$ 这一个记号，您可以在不引起歧义的前提下，在作答时将笔记中的 $\Theta(\cdot)$ 用 $O(\cdot)$ 代替。除了这三种复杂度记号之外，还有少见的两种复杂度记号：$o(\cdot)$ 和 $\omega(\cdot)$。因为在科研生活里也极少使用，所以不进行介绍。需要指出，一些错误的理解甚至可能成为 *408* 考试的标准答案，这种情况造成的失分是无法避免的，只能在考试中揣摩出题者的意图了。

在上面的定义中，引入了时间单元和空间单元的概念。因为渐进复杂度的记号表示中不考虑常数，所以这两个单元的大小是可以任取的。例如，一个时间单元可以取成：

- 一秒（毫秒，微秒，纳秒，分钟，小时等绝对时间单位）。
- 一个 CPU 周期。
- 一条汇编语句。
- 一次基本运算（如加减乘除）。
- 一次内存读取。
- 一条普通语句（不含循环、函数调用等）。
- 一组普通语句。

又例如，一个空间单元可以取成：

- 一个比特（字节、半字、字、双字等绝对单位）。
- 一个结构体（固定大小）所占空间。
- 一个页（参见《操作系统》）。
- 一个栈帧（参见《操作系统》）。

这些单位并不一定能直接地相互转换。比如，即使是同一台计算机，它的“一个 CPU 周期”对应的绝对时间也可能会发生变化（CPU 过热时降频）。但是这些单位在转换时，转换倍率必然存在常数的上界。比如，能正常工作的内存绝不可能需要 10^9 个 CPU 周期才能完成读取。

这个常数级别的差距，在复杂度分析里被纳入到了 C、C_1、C_2 中，而不会影响到渐进复杂度。因此，复杂度分析成功回避了硬件、软件、环境条件等“算法外因素”对算法效率的影响。

1.5.2　复杂度记号的常见理解误区

这一小节单独开辟出来，讨论和复杂度记号（尤其是 $O(\cdot)$）有关的注意点。由于复杂度记号总是作为一门学科的背景知识出现，您可能会没有意识到这是一个相当容易混淆的概念，从而陷入某些误区而不自知。

不可交换。已知 $n^3 = O(n^4)$，$n^2 = O(n^4)$，那么，是否有 $n^3 = O(n^4) = n^2$ 呢? 显然是不可能的。等于号“=”的两边通常都是可以交换的，但在复杂度记号这里并非如此。在进行复杂度的连等式计算时，始终需要记住：

1. **等式左边包含的信息不少于右边**。（最核心的性质）
2. 复杂度记号本身损失了常数的信息。因此复杂度记号只能出现在等式的右侧。如果出现在左侧，那么右侧也必须是复杂度记号。
3. 从 $\Theta(\cdot)$ 转换成 $O(\cdot)$ 或 $\Omega(\cdot)$，会损失一侧的信息。因此连等式中 $\Theta(\cdot)$ 只能出现在 $O(\cdot)$ 或 $\Omega(\cdot)$ 的左侧。只有一种情况除外，就是 $O(1) = \Theta(1)$。

例如，$2n^2 = \Theta(n^2) = O(n^3) = O(n^4)$ 是正确的。

不可比较。已知算法 A 的复杂度是 $O(1)$，算法 B 的复杂度是 $O(n)$，那么，算法 A 的复杂度是否低于算法 B?

这是最容易误解的一处，切不能想当然认为算法 A 的复杂度低于算法 B。这是因为，尽管已知条件是“算法 B 的复杂度是 $O(n)$”，但已知条件并没有排除“算法 B 的复杂度同时也是 $O(1)$”的可能。$O(1) = O(n)$，这个式子是正确的。$O(1) < O(n)$ 则是完全错误。注意，*408* 的标准答案似乎也会犯这个错误。

不满足对减法和除法的分配律。思考一下，是否有 $O(H_n) - O(\ln n) = O(H_n - \ln n) = O(1)$ 呢? 这里记号 $H_n = \sum_{i=1}^{n} \frac{1}{i}$ 用来表示调和级数的部分和，在 $n \to \infty$ 时，$H_n - \ln n \to \gamma$，其中 γ 是 Euler 常数。这一性质在复杂度分析的领域非常重要，之后还会再次遇到。

答案是否定的。不论是三种复杂度记号中的哪一种，都不服从对减法的分配律。对加法和乘法，分配律是成立的，但是右侧的复杂度符号数量必须比左侧少（直观地理解是，复杂度记号包裹的范围越大，就会损失越多的信息）。比如 $O(f(n))\cdot O(g(n)) = O(f(n)\cdot g(n))$。您可以自己证明这些公式。

不是所有算法都可以用 $\Theta(\cdot)$ 评价。这个问题很容易从数学角度看出来，比如说 $T(n) = n \cdot (\sin \frac{n\pi}{2} + 1)$，就不存在“与它同阶的无穷大量”。正是因为这个原因，我们更多地使用表示上界的 $O(\cdot)$，而不是看起来可以精确描述增长速度的 $\Theta(\cdot)$。

复杂度记号和情况的好坏无关。也就是说，尽管表示上下界，但 $O(\cdot)$ 和 $\Omega(\cdot)$ 不代表“最坏情况（worst case）”和“最好情况（best case）”。首先，上面的例子 $T(n) = n \cdot (\sin \frac{n\pi}{2} + 1)$。很容易看出这个式子是 $O(n)$ 和 $\Omega(1)$。这个正弦的例子并看

不出来什么。

接下来，我们看另一个例子。假设一个黑箱里有 n 个除颜色外完全相同的球，其中有且仅有一个红球，其他都是黑球。连续不放回地取球，直到取出红球为止。把“一次取球”看成是一个时间单元，则取球次数为 $T(n)$。这个例子中，如果认为上述算法的复杂度为“最坏 $O(n)$”和“最好 $\Omega(1)$”，就会感觉到有一点违和。在最好情况下，不管 n 是多少，都有 $T(n) = 1$，所以，它应当是 $O(1)$ 的；而在最坏的情况下，总是有 $T(n) = n$，最后一个球才取出红球，所以，它应当是 $\Omega(n)$ 的。我们惊讶地发现，在这个例子中，两种复杂度记号“易位”了。

那么，这两个例子有什么不同呢？在正弦的例子中，n 是唯一的变量，并不存在和 n 无关的变量“情况（case）”在影响 $T(n)$。在取球的例子中，n 不再是唯一的变量，在 n 被确定了的情况下，还有可好可坏的情况在影响 $T(n)$。

情况（case）表示和问题规模 n 无关的输入数据特征。在取球的例子中，合理描述算法复杂度的方法应该是“最坏 $\Theta(n)$”和“最好 $\Theta(1)$”。由于通常关心的都是最坏情况的下界和最好情况的上界，所以也可以省略一些信息，表述成“最坏 $\Omega(n)$”和“最好 $O(1)$”。正是因为情况和复杂度记号无关，所以在只使用 $O(\cdot)$ 的书上，无论是最好、最坏还是平均都可以使用这个记号。

1.5.3　实验：判断 2 的幂次

下面几个小节，将通过一些简单的算法，介绍复杂度分析的基本方法。更多的复杂度分析将穿插在整本书中。在这一节我们讨论，判断一个数是否为 2 的幂次，代码可以在 *IsPower2.cpp* 中找到。

```
class IsPower2 : public Algorithm<bool, int> {};
```

这个问题并不难，一种常规的实现是：

```
// IsPower2Basic
bool operator()(int n) override {
    return n > 0 && (n & (n - 1)) == 0;
}
```

这里判断 $n > 0$ 是为了健壮性。您可以从二进制数的特征出发，证明上面这个算法的正确性。对于有经验的程序员来说，写出上面的实现易如反掌，但并不是每一个人都能记住它。C++20 提供了一个专门的函数，判断一个二进制数据是否恰好只有一个非 0 比特位（对于整数来说，这等价于 2 的幂次）。

```
// IsPower2SingleBit
bool operator()(int n) override {
    return n > 0 && has_single_bit(static_cast<unsigned>(n));
}
```

这里std::has_single_bit只接受无符号数，所以需要先判断是否大于 0，再进行静态强制转换。**static_cast**是 C++ 推荐的写法，您也可以采用 C 语言风格的强制类型转换，但 C++ 风格的写法是更加类型安全的，它会在编译期检查转换是否合法。

这种简单的检查方法只需要常数次的计算，时间复杂度和空间复杂度都是 $O(1)$。对于既不熟悉二进制位运算，又不了解标准库的程序员，可能会写出下面的算法来解决这个问题：

```
// IsPower2Recursive
bool operator()(int n) override {
    if (n % 2 == 1) {
        return n == 1;
    } else {
        return (*this)(n / 2);
    }
}
```

这个算法涉及递归，显然它的时间复杂度不再是 $O(1)$。为了证明上述算法的有限性，可以采用前述的递降法，如构造 $f(n) = \max\left(d : 2^d | n\right)$ 来映射到通常良序集。但对这种简单的问题，也可以直接显式地计算 $T(n)$，即函数体的执行次数。显式地计算出有限的 $T(n)$，也就在复杂度分析的同时“顺便”证明了算法的有限性。

设 $n = k \cdot 2^d$，其中 k 为正奇数，d 为自然数。则 $T(n) = T(k \cdot 2^d) = 1 + T(k \cdot 2^{d-1}) = 2 + T(k \cdot 2^{d-2}) = \cdots = d + T(k) = d + 1 = O(d) = O(\log(n/k)) = O(\log n)$。另一边的 $\Omega(1)$ 是显然的。

上面这个 $T(n)$ 的计算过程，本质上仍然是使用了递降法，将 $T(\cdot)$ 的参数不断递降到递归边界（正奇数 k），思路和正确性检验的递降法是一致的，它利用了 $T(\cdot)$ 的递归式去显式地计算这个值。这里注明一点：因为计算机领域广泛使用二进制，所以未指定底数的对数符号“log”，底数默认为 2。而因为换底公式的存在，在复杂度记号下无论使用什么底数都没有区别。您可能会发现，这个算法在给定错误值，如 *0* 和*-1* 的时候会陷入无限递归，这属于稳健性问题，您可以自己改正它。

请注意，如果我们认为 n 是“问题规模”，那么就不能说奇数的情况是“最好情形”，因为此时没有“情形”的概念；这使得我们不能说最好 $O(1)$ 和最坏 $O(n)$。但如果认为 n 的位宽 $\log n$ 是“问题规模”，则可以这么说，这从侧面反映了用位宽，也就是“输入规模”来表示问题规模的好处。邓书上也建议这样表示问题规模；但习惯上（和直观上），如果输入数据只有一个数 n，通常也认为 $O(\log n)$ 的算法具有对数复杂度，而非线性复杂度。这是一个争议性话题，没有标准答案。

同样，可以显式地计算下面这个算法的 $T(n)$。

```
// IsPower2Iterative
```

```
2  bool operator()(int n) override {
3      int m { 1 };
4      while (m < n) {
5          m *= 2;
6      }
7      return m == n;
8  }
```

上面的两个实现，并不是同一算法的迭代形式和递归形式。根据上面的分析可以看到，递归算法的时间复杂度为 $O(\log n)$ 和 $\Omega(1)$，而迭代算法为 $\Theta(\log n)$，您可以自己证明这一点。此外，这个迭代算法也存在稳健性问题，如当输入 $n = 2^{31} - 1$ 时，就会陷入无限循环，您可以想办法改正它。

如果您掌握了对循环和递归相互转换，您可以实现 IsPower2Recursive 的迭代版本，以及 IsPower2Iterative 的递归版本。这两个算法的区别在两个方面。第一，迭代版本的迭代方向是“递推”而不是“递降”；第二，递归边界和循环终止条件不对应。第二点是更加本质的区别，它指示了在最好情况下（n 为奇数时），时间复杂度不同的原因。

另一方面，二者的空间复杂度也有所不同。在迭代算法中，只引入了 1 个临时变量 m，因此空间复杂度是 $O(1)$。而在递归算法中，看似一个临时变量都没有引入，空间复杂度也应该是 $O(1)$，实则不然。递归算法在达到递归边界之前，每一次递归调用的函数都在等待内层递归的返回值。到达递归边界、判断完成后，这一结果被一级一级传上去，途中调用函数占据的空间才会被销毁（参见《操作系统》）。

因此，对于递归算法，递归所占用的空间在复杂度意义上等于最大递归深度。IsPower2Recursive 的空间复杂度同样是 $O(\log n)$ 和 $\Omega(1)$ 的。现代编译器可能会将它优化成迭代形式，但在《数据结构》的解题过程中，永远不要考虑编译器优化问题。

考试时往往更加重视时间复杂度，因为现代计算机的内存通常足够普通的程序使用，而且《数据结构》中涉及的大多数算法，空间复杂度要么显而易见、要么能在计算时间复杂度的时候顺便算出来。但空间效率仍然是衡量数据结构的重要指标。这个空间效率不单指空间复杂度，也包含被复杂度隐藏下去的和数据结构相关的常数。比方说，如果在同一计算机上，数据结构 A 比数据结构 B 的常数低 10 倍，那么它就能存放 10 倍的数据，这个优势是非常大的——即使二者的空间复杂度一致。

1.5.4　实验：快速幂

上一节中讨论到了*问题规模*的概念。在邓书中，使用了快速幂作为举例阐述这个问题，本书也将快速幂作为一个实验。代码可以在 *Power.cpp* 中找到。

快速幂是一个解决求幂问题的算法。求幂问题中，我们输入两个正整数 a 和 b，输出 a^b。

```
class PowerProblem : public Algorithm<int, int, int> {};
```

基本的求幂算法，和求和类似：

```
int operator()(int a, int b) override {
    int result { 1 };
    for (int i { 0 }; i < b; ++i) {
        result *= a;
    }
    return result;
}
```

您可以毫不费力地看出这个算法的时间复杂度是 $\Theta(b)$。当 b 比较大时，这个算法的时间效率很低。这是因为，在计算 a^b 的时候，采用的递推式是 $a^b = (a(a(a(a(a\cdots(a\cdot a)\cdots)))))$，像普通的 b 个数相乘一样简单地循环，没有利用 a^b 的在计算上的自相似性。

事实上，我们可以将这 b 个 a 两两分组。如果 b 是偶数，那么恰好可以分成 $\frac{b}{2}$ 组，于是我们只需要计算 $(a^2)^{\frac{b}{2}}$。奇数的情形需要乘上那个没能进组的 a。于是，我们通过 1 到 2 次乘法，将 b 次幂问题化归到了 $\frac{b}{2}$ 次幂问题。

```
int operator()(int a, int b) override {
    if (b == 0) {
        return 1;
    } else if (b % 2 == 1) {
        return a * (*this)(a * a, b / 2);
    } else {
        return (*this)(a * a, b / 2);
    }
}
```

容易证明这个算法的时间、空间复杂度均为 $\Theta(\log b)$。示例代码还提供了它的迭代版本，*您也可以试着自己将它改为迭代*。通常，$\Theta(\log b)$ 复杂度的求幂算法都称为快速幂，除了上面介绍的这种实现（借助 $a^b = (a^2)^{\frac{b}{2}}$），还有另一种实现（借助 $a^b = \left(a^{\frac{b}{2}}\right)^2$），*您也可以试着去实现它*。

现在问题来了：这个算法的*问题规模*是什么？

最普遍被接受，也最自然的想法是，它的问题规模是 b。这样，上述两种算法的复杂度都可以用这个问题规模表示；至于 a 的值，则被归入“情况”的范畴，并且它也不会影响到这两个算法的复杂度。

另一种学说认为，它的问题规模是 $\log b$。这一学说的依据是：问题规模应当是描述这一问题的输入需要的数据量（即输入规模）。在这个问题中，要描述问题中的 b，在二进制计算机中需要 $\log b$ 个比特的数据，所以问题规模是 $\log b$。

这两种方法各有利弊。第一种学说的优点在于形象直观，容易理解；第二种学说的优点则在于有迹可循，定义统一。比如，如果让快速幂中的 b 允许超出`int`限制，比如使用 Java 中的`BigInteger`，那么 b 势必用数组或者类似的数据结构表示。这个时候，因为实质上输入的是一个数组，即使是“直观派”也会倾向于将“数组的大小”也就是 $\log b$ 作为问题规模。于是，“直观派”无法让问题规模的定义在扩展的情况下保持统一，而“空间派”可以做到这一点。

在大多数情况下，这两种学说并无分歧。邓书上支持“输入规模派”，而大部分书上没有讨论这个问题，通常这个问题对解题也没有任何影响。

1.5.5　实验：复杂度的积分计算

递归算法的复杂度分析在之后还会讨论更多技术。在比较简单的试卷上，命题者通常不会用递归算法命题，而是使用更为简单的迭代算法，命制选择或填空题。这类题目的共同点是迭代的次数非常清晰。比如前面的 IsPower2Iterative，可以一眼就看出来迭代的次数是 $\Theta(\log n)$。

这种题目通常可以用积分计算，而不需要用递降法。下面是一个没什么实际意义的例子。可以在 *ComplexityIntegeral.cpp* 中找到它，并通过测试验证下面计算得到的时间复杂度。

```
int f(int n) {
    int result = 0;
    for (int i { 1 }; i <= n; ++i) {
        for (int j { 1 }; j <= i; ++j)
            for (int k { 1 }; k <= j; ++k)
                for (int l { 1 }; l <= j; l *= 2)
                    result += k * l;
    }
    return result;
}
```

很容易看出，要分析该算法的时间复杂度，只需要算循环体被执行了几次，也就是计算：

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} j \cdot (1 + \lfloor \log j \rfloor)$$

要显式地求出这个和式非常困难。幸运的是，需要求出的是复杂度而不是精确的值，常数和小项都可以在求和过程被省略掉。这给了您解决它的手段：离散的

求和问题可以直接转换成连续的积分问题。

如果我们只需要一个渐进复杂度，那么积分也不需要真的去求，每次积的时候直接乘上一个线性量就可以。如果想要估算常数，则求积分的时候可以每一步只保留最高次项。

比如，上面的求和式可以计算如下：

$$T(n)=\sum_{i=1}^{n}\sum_{j=1}^{i}j\cdot(1+\lfloor\log j\rfloor)=O\left(\int_0^n\int_0^x y(1+\log y)\mathrm{d}y\mathrm{d}x\right)$$

$$=O\left(\int_0^n\int_0^x y\log y\mathrm{d}y\mathrm{d}x\right)=O\left(\int_0^n x^2\log x\mathrm{d}x\right)=O(n^3\log n)$$

其中，第一步是将求和符号转换成积分；消去积分符号的过程是做了两次“乘上一个线性量”的操作。如果想要估算常数，只需要：

$$T(n)=\sum_{i=1}^{n}\sum_{j=1}^{i}j\cdot(1+\lfloor\log j\rfloor)\sim\int_0^n\int_0^x y(1+\log y)\mathrm{d}y\mathrm{d}x$$

$$\sim\int_0^n\int_0^x y\log y\mathrm{d}y\mathrm{d}x\sim\int_0^n\frac{1}{2}x^2\log x\mathrm{d}x\sim\frac{1}{6}n^3\log n$$

这里最后的 $\frac{1}{6}$ 就是常系数；在一些难度较高的算法分析题中，常系数可能会被用到。

第 2 章　向量

2.1　线性表

线性表是指相同类型的有限个数据组成的*序列*。本书按照邓书的方法，将线性表分为**向量**（vector）和**列表**（list）两种形式，分别对应 C++ 中的`std::vector`和`std::list`。在另一些教材中，这两个词被称为**顺序表**和**链表**，分别对应 Java 里的`ArrayList`和`LinkedList`。向量（顺序表）和列表（链表）这两对概念通常可以混用。向量和列表代表着两种最基本的数据结构组织形式：**顺序结构**和**链式结构**。在这一节，将首先让读者对这两种组织形式有一个基本的印象。

首先，我们思考这样的问题：抛开顺序结构或链式结构不谈，*线性表*这个概念本身具有怎样的性质？这将会指导我们设计`LinearList`这个类。

对比它的基类`DataStructure`。一般的数据结构的定义中，“某种结构化的形式”，在线性表这里被具体化为了“序列”。既然是序列，那么它会具有头和尾，会具有“第 i 个”这样的概念。这样看起来已经完备了，但我们忽略了一个问题：应该如何访问序列中的一个元素呢？

把计算机中的内存想像为一座巨大的旅馆，线性表是一个居住在其中的旅游团。现在旅游团预定了一大片连续的房间，比如，从 1000 到 1099，并且让第 1 个游客住在 1000，第 2 个游客住在 1001，以此类推，那么我们就很方便地可以知道第 i 个游客的房间号。这种情况下，我们只需要知道游客的序号，就能知道它们居住的房间号。

但并不是每个人都会按部就班地居住，一些人可能喜欢阳光，一些人可能想和伙伴们做邻居，于是，这些游客开始交换房间。房间被交换之后，我们再也无法直接知道第 i 个游客的房间号。一个可能的想法是，从 1000 到 1099 每一个房间都敲敲门。这种方法虽然可行，但无疑是很低效的。

更加糟糕的是，旅游团可能没订到连续的房间，游客们散落在旅馆的各处。因为我们不可能像推销员那样每个房间都敲门（这会被赶出去的），所以再也无法找到我们想要的第 i 个游客了。为了应对这种情况，旅游团的导游往往会记录下各个游客所在的房间号，以便能够找到他们。

在上面这个比喻中，我们可以看到，如果数据结构中的元素被连续地储存，那么我们可以通过它们的序号（在邓书中，称为秩，rank）来找到它们；如果数据结构中的元素并没有被连续地储存，则我们只能通过它们的位置（position）来找到它们。上面的三种情况，分别是地址连续、且地址和秩相关的顺序结构（向量）；地

址连续、但地址和秩无关的静态链式结构（静态链表）；地址不连续、也和秩无关的动态链式结构（动态链表）。图2.1示意了三种结构的区别。

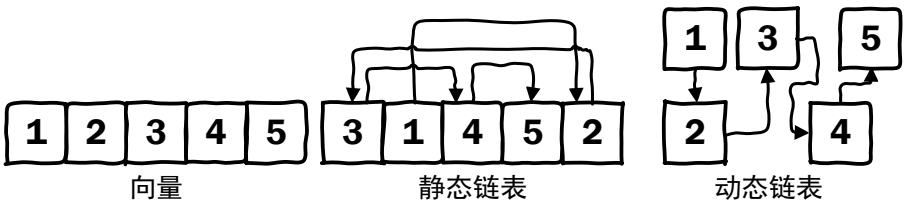

图 2.1　向量、静态链表和动态链表的对比

显然，如果发生了第二种情况，导游通常还是会选择记录房间号而不是逐个敲门。那么既然没有省事，也就没有必要预定一大片房间了。因为旅馆老板，也就是操作系统，可能会乘机宰客。比如，旅游团一次定了 100 个房间，但中途有 50 人提前结束了旅行。由于旅游团定的是整单，老板不允许单独退这 50 个人的房间。于是，旅游团要么承担 50 间空房的代价（空间损失）；要么再定 50 个房间，请剩下的 50 人搬到新房间住（时间损失），然后把原来的 100 个房间一并退掉。

这个例子展示了静态链表是一个不实用的数据结构，因此在邓书中它被删除。在包括静态链表的教材中，它也不是重点的一部分。我们将聚焦在向量和动态链表（列表）上。

如前所述，向量和列表里定位元素的方法是不同的。向量是循秩访问，而列表则循位置访问。因此，我们设计的线性表抽象类中需要反映这一点。

```
1  template <typename T, typename Pos>
2  class AbstractLinearList : public DataStructure<T> {
3  public:
4     virtual T& get(Pos p) = 0;
5     virtual void set(Pos p, const T& e) = 0;
6
7     virtual Pos insert(Pos p, const T& e) = 0;
8     virtual Pos insert(Pos p, T&& e) = 0;
9     virtual Pos find(const T& e) const = 0;
10    virtual T remove(Pos p) = 0;
11    virtual void clear() = 0;
12
13    virtual Pos first() const = 0;
14    virtual Pos last() const = 0;
15    virtual Pos next(Pos p) const = 0;
16    virtual Pos prev(Pos p) const = 0;
17    virtual bool end(Pos p) const = 0;
18
19    virtual T& operator[](Pos p) {
20       return get(p);
21    }
22    virtual T& front() {
23       return get(first());
```

```
24      }
25      virtual T& back() {
26          return get(last());
27      }
28  };
```

如果您查看 AbstractLinearList.ixx，还会看到一些其他的方法。正如序言中所说的那样，那些和《数据结构》研究的内容关系不大的方法将在书里被隐藏。现在我们分析展示出来的这些方法，可以看到它们分为 4 组（实际上是 5 组，还有一组是继承于`DataStructure`、有关规模的方法）。

1. 根据位置 p 访问或修改元素。
2. 借助位置实现元素的增、删、查；以及清空整个线性表。
3. 位置相关的操作。我们可以通过`first`和`last`获取第一个和最后一个元素的位置。对于给定的元素（已知它的位置），我们可以通过`next`和`prev`获取序列中下一个和上一个元素的位置。此外还有一个`end`方法用来判断位置是否到达了表尾。
4. 最后是我们一开始想到的三个方法：第一个、最后一个以及第 i 个。由于秩 i 在列表中并不足以定位到目标，所以“第 i 个”这里的 i 被声明为了`Pos`类型，当实现列表的时候它是一个位置。借助前面定义的方法，可以很容易地写出这三个方法。

您可能会觉得，这个线性表类中包含的内容过多或者过少。如果您感兴趣，我也非常鼓励您实现自己的抽象类模板。本书给出的模板保证您在不参与设计细节的情况下能完成本书设计的有趣实验，您可以随时用自己实现的程序替换其中的一部分。如果您打算这么做，请记得使用版本控制程序或备份，保证您可以回退到程序能正常运行的时刻。

2.2　向量的定义

向量（vector）是一个基于**数组**（array）的数据结构，它在内存中占据的是一段连续的空间。C++ 建议更多地使用标准库中的向量`std::vector`代替数组。向量和数组相比，其最重要的区别在于它是运行时可变长的，而在其他使用上，二者基本可以等同。因此，向量上的算法可以很容易地修改为数组上的算法（即使不使用`std::span`）。在 408 中通常不考虑可变长这一性质，此时讨论的顺序表就是数组，但您同样可以用本章中介绍的向量算法解决相关题目。

作为一种基于数组的线性表，向量的元素次序和数组的元素次序相同。如果一个向量 V 基于长度为 n 的数组 A 构建，那么向量 V 的第 i 个元素就是 $A[i]$。我

们知道，C 语言的数组可以视为指针，于是向量 V 的第 i 个元素的地址就是 $A+i$。

向量里的元素数量 n，称为向量的**规模**（size）；而向量所占有的连续空间能够容纳的元素数量 m，称为向量的**容量**（capacity）。这两者通常是不同的，规模必定不大于容量，而在不超过容量的前提下，向量的规模可以灵活变化，从而赋予了它比数组更高的灵活性。前面的那个旅行团的例子也可以帮助理解规模和容量的关系。一个 n 个人的旅行团订了连续的 m 个房间，这里 m 可以大于 n，这样，如果旅行过程中有新人加入团队，他们就可以直接加入到已经预定的连续房间中来。

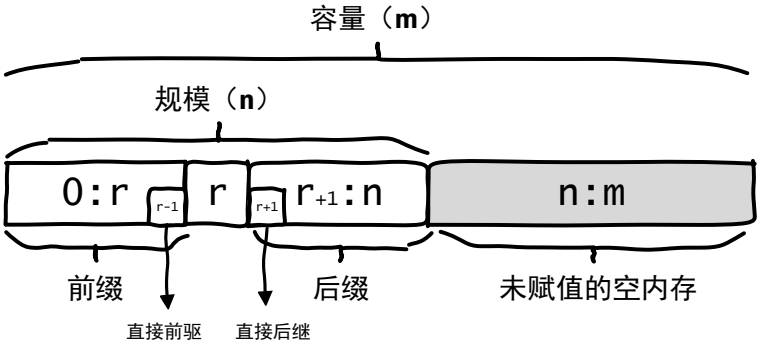

图 2.2　向量的基本概念

如图2.2所示，对于向量中的每一个元素 $V[i]$ 来说，它前面的元素称为它的**前驱**（predecessor），它后面的元素称为它的**后继**（successor）。特别地，和它位置相邻的前驱，也就是**直接前驱**为 $V[i-1]$，相应地，**直接后继**为 $V[i+1]$。所有的前驱构成了**前缀**（prefix），也就是 $V[0{:}i]$；所有的后继构成了**后缀**（suffix），也就是 $V[i+1{:}n]$。本书中我们用 $V[a{:}b]$ 来简记 $V[a], V[a+1], \cdots, V[b-1]$ 这个子序列，这是 Python 的切片（slice）语法，借助它可以简化很多叙述，尤其在记忆一些比较复杂的算法时很有用。

2.3　循秩访问

了解了向量的定义之后，我们将之前实现的线性表抽象类进行细化，构建向量的抽象类`AbstractVector`。随后本书会讲解一个示例实现，您可以自行继承`AbstractVector`进行实现，因为那些不需要您关注的方法都已经在这个抽象类中实现。*比如说迭代器，对于读者而言实现迭代器会浪费很多时间，因此迭代器被完全排除在实验体系之外。我们的目标是学习数据结构，而不是复现 STL。*

向量的核心特征是**循秩访问**。称元素 $V[i]$ 在向量 V 中的序号 i 为它的**秩**（rank）。对于建立在数组 A 上的向量 V，因为 V 和 A 的元素次序是一致的，所以 $V[i] = A[i] = *(A+i)$。因此，只要知道一个元素的秩，就可以在 $O(1)$ 的时间内访问该元

素。通常，我们用size_t类型存储下标。为了强调向量是循秩访问的，我们给它一个别名。

```
using Rank = size_t;
```

接下来，我们开始构筑向量抽象类。首先，除了在DataStructure里定义的规模size之外，我们还需要定义容量capacity。同时，因为向量是可变长的，所以规模和容量都是可以变化的，还需要两个修改它们的方法。有了修改规模的方法，线性表里的clear就可以直接用resize(0)实现。其次，我们需要构筑循秩访问的体系，重写AbstractLinearList里面的get和set方法。

```
template <typename T>
class AbstractVector : public AbstractLinearList<T, Rank> {
protected:
   virtual T* data() = 0;
public:
   virtual size_t capacity() const = 0;
   virtual void reserve(size_t n) = 0;
   virtual void resize(size_t n) = 0;

   T& get(Rank r) override {
      return data()[r];
   }
   void set(Rank r, const T& e) override {
      data()[r] = e;
   }
   void clear() override {
      resize(0);
   }
};
```

操作底层内存的时候，因为和所有权无关，所以不能使用智能指针，只能使用裸指针。为了避免裸指针被外部获取，我们将向量的获取首地址方法data()设置为受保护的，而公共方法只会返回引用。在线性表中定义的第三组方法，如位置的first、last等，您可以用Rank代替Pos，然后自己实现这些方法，这应当非常简单。

现在，我们已经拥有了一个抽象类AbstractVector，它还缺少下面这些组件的实现：获取规模、容量和首地址的方法；修改规模和容量的方法；插入、删除、查找的方法。如果您打算自己实现向量类，只需要在抽象类的基础上补充它们即可；您也可以继承本书提供的示例向量类，重写其中的部分方法。我们的示例类会从下面开始。

```
template <typename T>
class Vector : public AbstractVector<T> {
   std::unique_ptr<T[]> m_data { nullptr };
   size_t m_capacity { 0 };
```

```
    size_t m_size { 0 };

    T* data() override { return m_data.get(); }
public:
    size_t capacity() const override { return m_capacity; }
    size_t size() const override { return m_size; }
}
```

C++14 起，允许用户使用智能指针管理数组，因此我们使用`std::unique_ptr`来申请内存。它的好处是不需要在析构函数里手动释放，减少了手动管理内存的麻烦。

2.4　向量的容量和规模

2.4.1　初始化

当我们建立一个新的数据结构的时候，有几种情况是比较典型的。在这里，以向量为例展示它们，后面讨论其他的数据结构的时候不再赘述。

零初始化。即，生成一个空的数据结构。对于向量来说，这应该包含一个大小为 0 的数组，并把容量和规模都赋值为 0。然而，C++ 不支持大小为 0 的数组，所以只能把`data`赋值为`nullptr`，就像在上一节中展示的默认值那样。

指定大小的初始化。即，给定 n，生成一个规模为 n 的数据结构，其中的每个元素都采用默认值（即元素采用零初始化）。对于向量而言，可以申请一片大小为 n 的内存，如下面的代码所示。

```
Vector(size_t n) : m_data { std::make_unique<T[]>(n) },
    m_capacity { n }, m_size { n } {}
```

复制初始化。即，给定相同数据结构的一个对象，复制该对象里的所有数据及数据之间的结构化关系。对于向量来说，只需要在申请大小为 n 的内存之后，将给定的向量的元素逐一复制进来即可，如下面的代码所示。注意这里在初始化器中显式调用了上面的“指定大小的初始化”的构造函数，为向量进行了初步的初始化，然后再把另一个向量的数据复制进来。

```
Vector(const Vector& rhs) : Vector(rhs.m_size) { std::copy_n(
    rhs.m_data.get(), rhs.m_size, m_data.get()); }
```

移动初始化。即，给定相同数据结构的一个对象，将该对象里的所有数据及数据之间的结构化关系移动到当前对象处。**移动**（move）语义和复制（copy）有显著的不同，因为在移动之后，“被移动”的对象失去了对数据的控制权，我们永远不会从被移动后的对象里访问那些数据。下面展示了向量移动初始化的一个例子。

```
1 Vector(Vector&& rhs) noexcept : m_data { std::move(rhs.m_data)
      }, m_capacity { rhs.m_capacity }, m_size { rhs.m_size } {
2     rhs.m_capacity = 0;
3     rhs.m_size = 0;
4 }
```

通过对智能指针调用std::move，我们可以在常数时间里将被移动对象的数据转移到新对象里。如果您使用的是普通的指针，这包含了指针赋值、被移动的指针置空两个步骤，而智能指针可以将它合成为一个步骤。可以看出，被移动之后，rhs的数据区被复位为空指针，规模和容量都被置 0，就像它被零初始化了一样，成为了一个空向量。

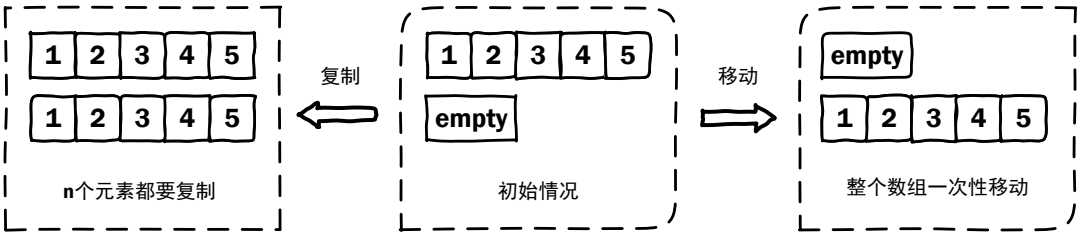

图 2.3　复制和移动的区别

从图2.3中可以直观了解到复制和移动的语义区别。当数据结构里的元素数量很多时，逐元素地复制是一项复杂、琐碎、漫长的工程，而移动则是一项简单、整体、快速的工作。在本书的示例代码中，我们经常会给同一个函数提供一个复制版本和一个移动版本。比如，对于插入（insert），我们定义一个插入const T&类型的方法用于复制（非破坏地插入），又定义了一个插入T&&类型的方法用于移动（破坏地插入）。这些方法之间往往只有一个或几个std::move的区别，因此本书通常省略移动版本的方法，而只展示复制版本的方法。尽管如此，在您的日常编程中需要时刻注意，只要复制和移动的时间成本有可能相差比较远，就应该同时定义并实现复制和移动两个版本的方法，而不能只实现复制。

需要注意的是，析构函数、复制构造函数、移动构造函数、复制赋值运算符、移动赋值运算符这 5 个函数，一旦显式定义其中的一个（比如想要定义复制构造函数），编译器就不会生成其他的函数。处于“一荣俱荣，一损俱损”的关系。因此，我们在日常编程的时候，通常选择不实现它们中间的任意一个函数（0 原则），因为日常编程的时候，通常都使用的是 STL 对象，而 STL 里已经将这些功能实现了。但是，在我们实现一些比较底层的结构时候，没法依靠 STL 里的实现，需要实现这 5 个函数中的一个或几个。此时，就必须要将所有的 5 个函数实现（5 原则）。我们可以使用=default来显式使用自动生成的函数，但如果我们不显式说明它，这些函数将不会被包含在这个类中。

初始化列表初始化。即，使用初始化列表`std::initializer_list`对数据结构进行初始化。初始化列表也是一个典型的线性容器，可以直接使用 STL 方法复制。

```
Vector(std::initializer_list<T> ilist) : Vector(ilist.size())
    {
    std::move(ilist.begin(), ilist.end(), m_data.get());
}
```

支持初始化列表初始化之后，我们就可以用下面的形式来初始化一个向量。

```
Vector V {1, 2, 3};
```

2.4.2　装填因子

设向量的容量为 m，规模为 n，则称比值 $\frac{n}{m}$ 为**装填因子**（load factor）。正常情况下，这是一个 $[0, 1]$ 之间的数。装填因子是衡量向量效率的重要指标。

1. 如果装填因子过小，则会造成内存浪费：申请了巨大的数组，但其中只有少量的单元被向量中的元素用到，其他单元都被闲置了。
2. 如果装填因子过大（超过 1），则会引发数组越界，造成段错误（segmentation fault）。

刚开始的时候，装填因子一定是在 $[0, 1]$ 之间的。但因为数组的容量 m 是固定的，而向量的规模 n 是动态的，所以一开始分配的 m 可能后来会不够用，从而产生装填因子大于 1 的问题，此时就需要令 m 增大，这一操作称为**扩容**（expand）。

由于 408 不讨论变长顺序表的问题，所以下面的几个章节可以略读。

2.4.3　改变容量和规模 *

您可能会想到，除了扩容之外，我们还可以进行**缩容**（shrink），降低 m 的值从而避免装填因子过小，造成内存浪费。但是，现实中很少进行缩容。因为扩容和缩容都需要时间，在扩容的场合是实现可变长特性的必需，但在缩容的场合仅仅是节约了空间而已。有多个原因让我们不愿意缩容：

1. 我们可以接受一定程度的空间浪费，因为很少有程序能占满全部的内存。
2. 如果缩容之后，又因为规模扩大而不得不扩容，一来一回浪费了不少时间，而价值甚微。
3. 当不得不考虑空间时，我们有很多其他方法可以节约出这些空间，不一定要使用缩容的技术。比如复制初始化生成一个新向量，然后清空原向量来释放内存；按照之前介绍的方法，这个新向量的装填因子为 1，处在空间最大利用的状态。

因此，在本书中将不再讨论缩容；您可以在自己的向量类中实现这个特性，并观察它的效果。

无论是扩容还是缩容，我们都需要重新申请一片内存。在扩容的场合，这很好理解。我们预定了 1000 到 1099 的房间，但在我们预定之后，1100 号房间可能被其他旅客占用了。这时如果我们想要预定连续的 200 个房间，就需要重新找一段空房间。缩容的场合，则是为了安全性考虑，不允许释放数组的部分内存。重新申请内存之后，我们将数据复制到新内存中。下面是扩容的一个实现。

```
1 void reserve(size_t n) override {
2     if (n > m_capacity) {
3         auto tmp { std::make_unique<T[]>(n) };
4         std::move(m_data.get(), m_data.get() + m_size, tmp.get()
              );
5         m_data = std::move(tmp);
6         m_capacity = n;
7     }
8 }
```

图2.4展示了当插入元素时如果发现容量不足，所进行的扩容过程。其中释放原先的内存这一步，本书中所采用的智能指针会自动完成，而 C 语言和旧标准 C++ 则需要显式地`delete[]`释放内存。

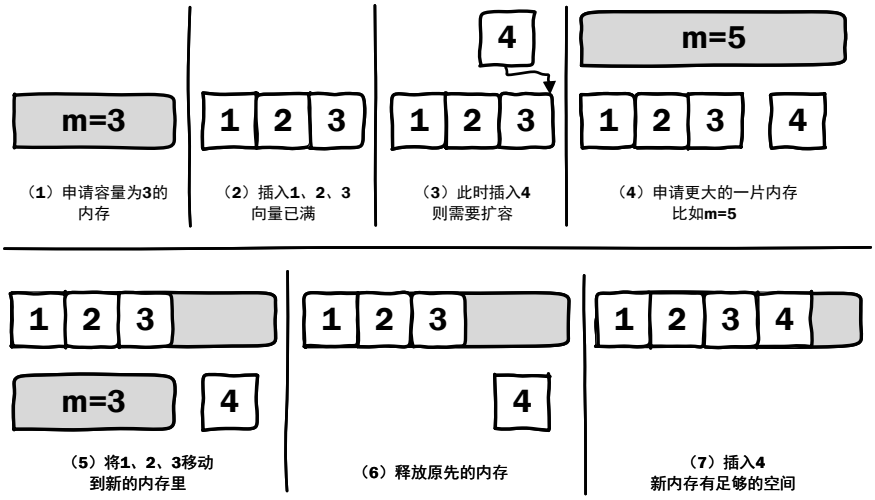

图 2.4　扩容过程

可以看出，扩容是一项成本很高的操作，因为它需要开辟一块新的内存。设扩容之后的容量为 m，则扩容算法的时间复杂度为 $\Theta(m)$。由于 m 可能会很大，我们不希望经常扩容。在下一节里将会讨论一些扩容策略。

我们设计了一个方法`resize`用来改变规模，当规模超过容量（装填因子超过 1）的时候调用`reserve`扩容。需要注意的是，一些其他的方法也会改变规模，比

如插入方法insert会让规模增加 1，而删除方法remove会让规模减 1。我们需要在实现插入方法的时候也考虑扩容问题；如果您实现了缩容，那么在实现删除方法的时候也需要考虑缩容问题。

```
void resize(size_t n) {
   if (n > m_capacity) {
      reserve(n);
   }
   m_size = n;
}
```

2.4.4　扩容策略 *

当我们调用resize的时候，可以立刻知道，需要扩大到多少容量才能容纳目标的规模。但实际情况下，很多时候元素是被一个一个加入到向量中的，这个时候，按照resize的策略，每次都扩容到新的规模，是一个糟糕的选择。假设初始化为了一个规模为 n 的向量，然后元素一个一个被加入，那么按照 $n \to n+1 \to n+2 \to \cdots$ 的次序扩容，每加入一个元素，都会造成至少 n 个元素的复制，时间效率极差。

因此，在面对持续插入的时候，我们需要设计新的扩容策略，以降低扩容发生的频率。这个策略应该是由向量的设计者提供的，而不是用户：如果用户知道更加合适的策略，他们会主动使用reserve进行扩容。但是，用户通常没有精力用在这种细节上；这个时候，向量的设计者提供的扩容策略就会成为一个不错的备选项。

现在我们尝试为扩容策略的问题添加一个抽象的描述。当我们讨论扩容的时候，显然不需要知道向量中的数据内容是什么。因此，扩容策略作为一个算法，输入向量的当前规模 n 和当前容量 m，输出新的容量 m'。这个描述具有良好的可重用性，它同样可以用于缩容。

```
class AbstractVectorAllocator : public Algorithm<size_t, size_t
   , size_t> {
protected:
   virtual size_t expand(size_t capacity, size_t size) const =
      0;
   virtual size_t shrink(size_t capacity, size_t size) const =
      0;
public:
   size_t operator()(size_t capacity, size_t size) override {
      if (capacity <= size) {
         return expand(capacity, size);
      } else {
         return shrink(capacity, size);
      }
   }
};
```

基于上述的分析，我们可以用这个类表示扩容策略。在后面的章节，将继承这个类并重写`expand`方法，实现不同策略的扩容，而缩容方法则直接返回`capacity`（永不缩容）。当您需要实现缩容的时候，也可以使用这个模板，重写`shrink`方法。如果您对如何扩容有一些自己的想法，笔者建议您先实现自己的扩容策略，然后再阅读后面的理论部分。

2.4.5　等差扩容和等比扩容 *

那么，应该如何设计扩容策略呢？一个简单的想法是，既然每次容量 +1 不行，那就加多一点。这种思路可以被概括为*等差数列*扩容方法。如果选取 d 作为公差，那么在本节开始的那个例子中，将按照 $n \to n+d \to n+2d \to \cdots$ 的次序扩容。

```
template <size_t D> requires (D > 0)
class VectorAllocatorAP : public AbstractVectorAllocator {
protected:
    size_t expand(size_t capacity, size_t size) const override
        {
        return capacity + D;
    }
};
```

这里使用了 C++20 引入的`requires`语法，要求用户给定的模板参数 $D > 0$。因为很明显，如果 $D \leqslant 0$，`expand`将永远扩不起来。

既然有了等差数列，另一个很容易想到的方法是按照*等比数列*扩容。如果选取 q 作为公比，则会按照 $n \to qn \to q^2 n \to \cdots$ 的次序扩容。

```
template <typename Q> requires (Q::num > Q::den)
class VectorAllocatorGP : public AbstractVectorAllocator {
protected:
    size_t expand(size_t capacity, size_t size) const override
        {
        size_t newCapacity { capacity * Q::num / Q::den };
        return std::max(newCapacity, capacity + 1);
    }
};
```

这里允许了 C++11 提供的编译期有理数`std::ratio`作为模板参数，`Q::num`表示分子，而`Q::den`表示分母。请注意，需要保证新的容量比原有容量大，否则扩容就没有意义。上面的做法保证了容量至少会扩大 1。比如，当 $Q = \frac{3}{2}$，往容量为 0 的向量里连续插入元素时，容量变化为 $0 \to 1 \to 2 \to 3 \to 4 \to 6 \to 9 \to \cdots$，如果没有容量至少扩大 1 的设计，等比扩容将永远停留在 0 容量。

2.4.6　分摊复杂度分析 *

很显然，进行单次扩容操作的时候，等差扩容的时间复杂度为 $O(n+d)=O(n)$，等比扩容的时间复杂度为 $O(qn)=O(n)$（因为 q 是常数），两者看起来没有区别；甚至和我们已经知道效率很低的情况（$d=1$ 的等差扩容）也没有区别。这也意味着，我们评价时间效率的方法可能出现了一些问题。

问题的关键在于，我们设计扩容策略的目的是按照等差或等比的*数列*扩容，而不是*一次*扩容。所以，评价这两种扩容规则的标准，不是进行一次扩容的效率或进行一次扩容后的装填因子，而是比较一系列扩容操作的总体效率和在这一系列扩容操作中的平均装填因子。用已有的复杂度分析工具不足以对这两种策略的效率进行准确评价。为了对*一系列*操作进行分析，需要引入新的复杂度分析标准。

一般地，假设 $O_1, O_2, \cdots, O_n$ 是连续进行的 n 次操作，则当 $n \to \infty$，这 n 次连续操作所用时间的*平均值*的复杂度，称为这一操作的**分摊复杂度**，对分摊复杂度的分析称为**分摊分析**。分摊分析的原则之一是：使用相同效果的操作序列。所以，要比较上述两种算法，不应该把每次操作取为“进行一次扩容”（因为两种方法扩容量不一样），而应该取为“向量 V 的规模增加 1”。连续进行 n 次操作，就可以考虑向量 V 的规模从 0 增长为 n 的过程。

在等差扩容方法中，容量依次被扩充为 $d, 2d, 3d, \cdots, n$，共进行 $\frac{n}{d}$ 次扩容。因此，分摊复杂度为：

$$T(n)=\frac{d+2d+3d+\cdots+n}{n}=\frac{\left(\frac{n}{d}\right)\cdot d+\frac{\left(\frac{n}{d}\right)\left(\frac{n}{d}-1\right)}{2}\cdot d}{n}=\frac{\frac{n}{d}+1}{2}=\Theta\left(\frac{n}{d}\right)$$

另一方面，进行 k 次扩容之后的装填因子至少为 $\frac{kd}{(k+1)d}=\frac{k}{k+1}$，当 $k \to \infty$ 时，装填因子趋于 100%。

在等比扩容方法中，容量被依次扩充为 $q, q^2, q^3, \cdots, n$，共进行 $\log_q n$ 次扩容。因此，分摊复杂度为：

$$T(n)=\frac{q+q^2+q^3+\cdots+n}{n}=\frac{q\cdot\frac{1-n}{1-q}}{n}=\Theta\left(\frac{q}{q-1}\right)=O(1)$$

另一方面，装填因子不断在 $\left[\frac{1}{q}, 1\right]$ 之间线性增长，平均装填因子为 $\frac{1+q}{2q}$。可以看出，不管怎样选择 q，对分摊复杂度都没有影响，而更小的 q 能够带来更大的平均装填因子。当选择 $q=2$ 时，平均装填因子为 75%。

可以看出，等比扩容的装填因子并没有很低，而换来了分摊时间复杂度上巨大的优化。因此，我们倾向于选择等比扩容。至于等比扩容的公比，则是一个值得讨论的话题。从上面的推导中，我们发现分摊时间复杂度的系数为 $\frac{q}{q-1}$，它会随 q

的增加而降低；另一方面，平均装填因子也会随 q 的增加而降低。因此，选择更大的 q，事实上是以时间换空间的做法。

因为分摊 $O(1)$ 已经很快，所以通常选取的 q 比较小。常见的公比选择有 2 和 $\frac{3}{2}$。邓书上介绍的版本选择了 2，这也是 GCC 和 Clang 的选择；而 MSVC 则采用更节约空间的 $\frac{3}{2}$。

需要指出的是，等比扩容也存在一些劣势：容量越大，装填因子不高带来的空间浪费愈发明显，所以有些对空间要求较高的情况下，也采用二者相结合的方式：在容量比较小时等比扩容、在容量比较大的时候等差扩容。这种思想在《网络原理》里的*慢启动*中得到了应用。

最后，既然有等比扩容，必然也有等比缩容。当我们讨论缩容的时候，通常需要结合一个**缩容阈值**，当装填因子低于这个阈值时才引发缩容，而不是每当规模低于容量时就缩容，否则就丧失了向量的灵活性。尽管本书不实现缩容（相当于缩容阈值为 0），但当缩容和缩容阈值被纳入讨论，可以命制一些有趣的问题。设等比扩容的公比为 $q_1 > 1$，等比缩容的公比为 $q_2 < 1$，缩容阈值为 θ，*您可以进行思考和计算，这三个变量满足什么条件时，才能保证对于任意的、由插入和删除组成的操作序列，扩容和缩容总和的分摊时间复杂度为 $O(1)$。*

2.5　插入、查找和删除

对于任何数据结构，都有三种基本的操作：

1. **插入**（insert）：向数据结构中插入一个元素。
2. **查找**（find）：查找一个元素在数据结构中的位置。
3. **删除**（delete）：从数据结构中移除一个元素。

在这一节中，我们以向量为例介绍这三种基本的操作。

2.5.1　插入一个元素

要将待插入的元素 e 插入到 $V[r]$，那么可以将原来的向量 $V[0{:}n]$ 分成 $V[0{:}r]$ 和 $V[r{:}n]$ 两部分。

- 插入之前，向量是 $V[0{:}r]$，$V[r{:}n]$。
- 插入之后，向量是 $V[0{:}r]$，e，$V[r{:}n]$。

可以发现，在插入的前后，前一段 $V[0{:}r]$ 的位置是不变的，而后一段 $V[r{:}n]$ 需要整体向后移动 1 个单元的位置。据此，可以设计下面的算法，算法的原理如图2.5所示。

```
1 Rank insert(Rank r, const T& e) override {
```

```
    std::move_backward(m_data.get() + r, m_data.get() + m_size,
        m_data.get() + m_size + 1);
    m_data[r] = e;
    ++m_size;
    return r;
}
```

这里需要使用`std::move_backward`而不能用`std::move`，因为向后移动 1 个单元这个过程，需要先移动最后一个元素、再移动倒数第二个元素、以此类推；如果从第一个元素开始移动，就会覆盖掉还没移动的元素。如果您对 STL 算法不熟练，也可以改写为熟悉的循环形式。

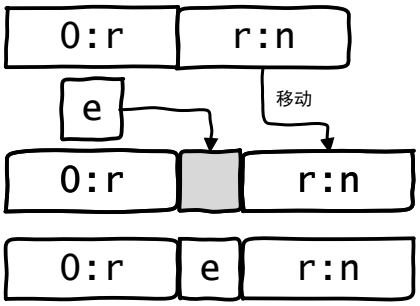

图 2.5　往向量中插入一个元素

在上面的算法中忽略了一点：插入有可能造成装填因子大于 1 的问题。如果准备插入的时候发现向量已满（装填因子等于 1），那么应当先使用上一节讨论的扩容策略，扩大向量的容量，然后再执行插入。您会发现，扩容的过程中包括了一次移动，`std::move_backward`又包括一次移动，这两次移动中会有一些重复的、不必要的赋值操作。不过，因为扩容被调用的次数很少，我们可以忽略这些额外的性能损失。

为了让我们在上一节设计的算法被嵌入到向量里来，我们可以为向量模板增加一个参数，写成下面这样的形式。

```
template <typename T, typename Alloc = VectorAllocatorGP<std::
    ratio<3, 2>>>
    requires std::is_base_of_v<AbstractVectorAllocator, Alloc>
```

在上面这个模板参数表声明中，我们规定模板的第二个参数`Alloc`必须是我们上面实现的`AbstractVectorAllocator`的派生类。用户可以不显式地指定扩容策略，而选择我们默认的策略（公比为 $\frac{3}{2}$ 的等比扩容）。加入这个参数之后，我们只需要在`insert`的开头进行一次判断，就可以自动地在插入满向量时进行扩容。

最后，我们上面实现的插入算法，传入的是**`const`**引用类型，这意味着元素并没有被真正的“插入”，被实际插入的是一个副本。如果我们希望让元素真正被插

入，那么应当使用移动语义。移动语义版本的插入算法和复制语义基本相同。如前所述，以后将不再展示移动语义的版本。

```
Rank insert(Rank r, T&& e) override {
    if (m_size == m_capacity) {
        reserve(Alloc {}(m_capacity, m_size));
    }
    std::move_backward(m_data.get() + r, m_data.get() + m_size,
        m_data.get() + m_size + 1);
    m_data[r] = std::move(e);
    ++m_size;
    return r;
}
```

2.5.2　平均复杂度分析

为了更定量地分析插入操作的时间效率，引入一个新的复杂度分析策略：**平均复杂度**。

在介绍复杂度时曾经强调，复杂度是依赖于数据规模，不依赖于输入情况的分析手段。在上面的插入算法中，数据规模通常认为是 n，而 r 是具体情况带来的参数。为了研究不同具体情况对算法时间效率的影响，有三种常见的分析手段：

1. **最坏时间复杂度**：研究在情况最坏的情况下的复杂度。很多算法有硬性的时间限制（如在复试的机试中，通常要求输出结果的时间不能多于 1s 或 2s），此时常常使用最坏时间复杂度分析。这是最常用的时间复杂度分析。
2. **最好时间复杂度**：研究在情况最好的情况下的复杂度。研究最好时间复杂度的意义远小于最坏时间复杂度。最好时间复杂度有时用于嘲讽某种算法的效率：在最好的情况下，这种算法的复杂度也只能达到（某个复杂度），而我的新算法在最坏的情况下也可以达到（某个更好的复杂度）。
3. **平均时间复杂度**：研究在平均情况下的复杂度。如果没有硬性的时间限制，则平均时间复杂度往往能更好地反映一个算法的总体时间效率。平均时间复杂度需要知道每种情况发生的先验概率，在这个概率的基础上计算 $T(n)$ 的数学期望的复杂度。在针对现实数据的实验研究中，常见的假设包括正态分布、Pareto 分布和 Poisson 分布；而在《数据结构》学科中，通常假设成等可能的分布，以方便进行理论计算。

平均复杂度很容易和分摊复杂度发生混淆，需要加以区分。下面是它们的一些典型的差异：

1. 分摊复杂度是一系列连续操作的平均效率，而平均复杂度是单次操作的期望效率。
2. 分摊复杂度的一系列连续操作是有可能（通常都）存在后效的，而平均复杂

度只讨论单次操作的可能情况。

3. 分摊复杂度需要指定每次进行何种的*基本操作*，而平均复杂度需要指定各种情况的*先验概率*。

最坏、最好、平均时间复杂度对应统计里的*最大值*、*最小值*和*数学期望*。显然，其他统计量，比如*方差*，在分析的时候也是有价值的，也深得科研人员重视。但在《数据结构》的考试中，是不会涉及到这些统计量的分析的，只需要知道最坏、最好和平均时间复杂度的分析技术即可。

现在回到插入的算法，它的时间复杂度是 $\Theta(n-r)$。显然，最好时间复杂度是 $O(1)$（插入在末尾的情况），最坏时间复杂度是 $\Theta(n)$（插入在开头的情况）。这里有略微不严谨的地方，因为 r 的最大值可以取到 n，此时 $n-r=0$，不再符合复杂度记号的定义；不过，因为我们清楚任何算法的时间都不可能为 0，所以一般不在这个细节上做区分。

为了求平均时间复杂度，一个合理的假设是，r 的取值对于 $[0{:}n+1]$ 之间的整数是等概率的（注意有 n 个可以插入的位置，而不是 $n-1$ 个）。在这个假设下，容易算出单次插入的平均时间复杂度为 $\Theta(n)$。

2.5.3　查找一个元素

查找需要返回找到的位置。对向量而言，只需要得到被查找元素的*秩*就可以了。和插入、删除相比，查找具有更加丰富的灵活性，甚至于一些编程语言（如 SQL）的核心就是查找。

最简单的查找是*按值查找*。即，给定被查找元素的值，在数据结构中找到等于这个值的元素。对于更加复杂的查找类型，比如*按区间查找*等，人们设计了更加复杂的数据结构来应对。对于按值查找的问题，最简单的方案就是检测向量中的每个元素是否等于要查找的元素 e，如果等于，就把它的秩返回。

```
1 Rank find(const T& e) const override {
2    for (size_t i { 0 }; i < m_size; ++i) {
3       if (m_data[i] == e) {
4          return i;
5       }
6    }
7    return m_size;
8 }
```

关于找不到的情况，有多种处理方式。上面采用的方法是返回无效的秩，除了返回序列尾部溢出的`m_size`之外，返回序列头部溢出的 -1 也是常见的选择。此外，也可以将返回值的类型改为`std::optional`，当找不到的时候返回一个无效值。

设 e 在向量中的秩为 r，那么在查找成功的情况下，上述算法的时间复杂度为 $\Theta(r)$。在查找失败的情况下，算法的时间复杂度为 $\Theta(n)$。这里可以分析，在等可能条件下，查找成功时的平均时间复杂度是 $\Theta(n)$。注意，查找成功的概率是一个很难假设的值，所以在分析平均时间复杂度时，通常只分析“查找成功时”和“查找失败时”的平均时间复杂度，而不会将它们混为一谈。

因为对于向量 V 和待查找元素 e 的情况没有更多的先验信息，所以暂时也没有比上面更高效的解决方案。**利用信息思考**是计算机领域重要的思维方式。在设计算法时，应尽可能利用更多的先验信息。反之，如果先验信息不足，则算法的效率受到信息论限制，不可能会特别高。这个思维方式在后文介绍各种算法的设计过程时，还会反复出现。

不过，这个算法还是有一些值得推敲的地方：如果 e 在向量 V 中出现了多次，那么这个算法只会返回最小的秩。*您可以思考一下，如何将其修改成返回最大的秩？并分析修改后的算法时间复杂度变化。*

2.5.4　删除一个元素

删除元素是插入元素的逆操作。在插入元素时，让被插入元素的后继*后移*；因此在删除元素的时候，只需要让被删除元素的后继*前移*即可。需要注意前移和后移在方向上的差别，插入时的`std::move_backward`，逆操作应该是`std::move`。

```
1 T remove(Rank r) override {
2    T e { std::move(m_data[r]) };
3    std::move(m_data.get() + r + 1, m_data.get() + m_size,
        m_data.get() + r);
4    --m_size;
5    return e;
6 }
```

和插入一样，可以分析出删除操作时间复杂度 $\Theta(n-r)$，平均时间复杂度 $\Theta(n)$，空间复杂度 $O(1)$。到此为止，我们实现了完整的`Vector`类，可以开始实验了。

2.5.5　实验：插入连续元素

在这个实验中，我们将用实验观察，等差扩容和等比扩容的时间效率差异。因为我们评估的是分摊时间，所以需要构造一个插入连续元素的场景来进行观察。从2.5.1节中我们知道，在位置 r 插入一个元素的时间复杂度为 $\Theta(n-r)$。为了降低插入连续元素这个操作本身对，更好地观察扩容时间，我们固定每次都在向量的末尾插入元素。代码可以在 *VectorInsert.cpp* 中找到。

我们构造下面的类，作为插入连续元素问题的基类。因为在测试过程中，每次连续插入结束之后，需要将向量重置为空（避免已分配的空间影响），所以这里定

义了一个reset方法。

```
class VectorInsertProblem : public Algorithm<size_t, int> {
public:
    virtual void reset() = 0;
};
```

返回值定义为连续插入结束后的向量容量，因为笔者希望观察连续插入结束之后扩容到了多大。您也可以将其改为**void**或者其他您希望观察的变量。接下来，我们让在2.4.5节中定义的等差扩容和等比扩容策略，作为Vector类的参数传入。

```
template <typename Vec>
    requires is_base_of_v<AbstractVector<int>, Vec>
class VectorInsertBasic : public VectorInsertProblem {
    Vec m_vector {};
public:
    size_t operator()(int n) override {
        for (size_t i { 0 }; i < n; ++i) {
            m_vector.insert(m_vector.size(), i);
        }
        return m_vector.capacity();
    }
    void reset() override {
        m_vector = Vec {};
    }
};
```

上面的模板接受一个Vec作为向量类型名，这是为了兼容您自己写的向量类。它用到了零初始化、赋值、插入、获取容量和规模的方法，如果您继承了AbstractVector，这些方法都应当已经实现。您也可以创建自己的测试类参与对比测试。对于本书的示例Vector实现，使用下面的模板。

```
template <typename Alloc>
    requires is_base_of_v<AbstractVectorAllocator, Alloc>
class VectorInsert : public VectorInsertBasic<Vector<int, Alloc
    >> {
public:
    string type_name() const override {
        return Alloc {}.type_name();
    }
};
```

这里重载了type_name，否则由于模板复杂，输出的类型名会非常长。在实验的示例代码中，我们比较了 $D = 64$、$D = 4096$、$Q = \frac{3}{2}$、$Q = 2$、$Q = 4$ 这些情况。您可以发现，当 n 比较小的时候，差异不明显；而当 n 比较大，如达到 10^6 的量级时，$D = 64$ 会极其缓慢，而 $D = 4096$ 也慢慢和三种等比扩容拉开距离（如果您增加一个 $n = 10^7$ 的用例，会更加明显）。相反，三种等比扩容的区别非常微小，我们可以判断出，这已经非常接近插入这些元素本身需要的时间，和扩容的关系不大。

如何证明上面的判断呢？去计算连续插入中原子操作的数量（计算时间复杂度的常数）是困难、繁琐且容易出错的工作。您可以加入一个 $Q = 10^6$ 的向量，它的容量会直接从 1 跳变到 10^6，也就是，在 $n = 10^6$ 的情况下，它没有进行任何多余的扩容。您会发现，即使是这样，也没有和上面三种等比扩容有显著差异，从而能够验证上面的判断。这种实验方法在计算机学科的学习中是一个有力武器。这个实验请您自己完成，您也可以加入一些自己实现的其他扩容策略来观察效果。比较有挑战性的工作是，在实现缩容之后，类似本实验，设计一个新的实验评估缩容的效果（请注意，缩容本身是时间换空间的做法，仅仅测试缩容的时间性能是不合适的）。

需要指出的是，这个实验结果只是粗略性的，因为连续插入而不做其他事情是一个非常特殊的用例。当这种情况真正发生的时候，应当直接`reserve`足够大的空间，而不是让向量按照设计者提供的默认扩容方案慢慢扩容。

2.5.6　实验：向量合并

如果要插入的不是单个元素，而是多个元素，情况会发生什么变化呢？现在，假设我们有两个向量 $V[0{:}n]$ 和 $V_1[0{:}n_1]$，我们希望将整个 V_1 插入到 V 的位置 r 处，实现向量合并。代码可以在 *VectorConcat.cpp* 中找到。这个问题不难，建议您自己实现算法参与对比。

```
class VectorConcat : public Algorithm<Vector<int>&, Rank> {
protected:
   Vector<int> V {}, V1 {};
public:
   void initialize(int n, int n1) {
      V.reserve(static_cast<size_t>(n) + n1);
      V1.reserve(n1);
      V.resize(n);
      V1.resize(n1);
   }
};
```

和上一节类似，我们定义了一个辅助函数用来对问题进行初始化。我们不希望扩容影响实验结果，所以预先给 V 分配了 $n + n_1$ 的空间。

如果我们采用2.5.1节中介绍的方法，将 V_1 中的元素一个一个插入到该位置，那么时间复杂度会高达 $\Theta((n-r)\cdot n_1)$，平均 $\Theta(n\cdot n_1)$，略显笨重。

```
// VectorConcatBasic
Vector<int>& operator()(Rank r) override {
   for (int i : V1) {
      V.insert(r, i);
      ++r;
   }
```

```
7     return V;
8  }
```

您可以敏锐地发现，只要再次使用在讨论单元素插入时的分析方法，就可以得到更加高效的算法。要将待插入的向量 V_1 插入到 $V[r]$，那么可以将原来的向量 $V[0:n]$ 分成 $V[0:r]$ 和 $V[r:n]$ 两部分。

1. 插入之前，向量是 $V[0:r]$，$V[r:n]$。
2. 插入之后，向量是 $V[0:r]$, V_1, $V[r:n]$。其中，$V[r:n]$ 被转移到了 $V[r+n_1:n+n_1]$ 的位置上。

这样设计出了一个批量插入的算法，两种方法的对比如图2.6所示。

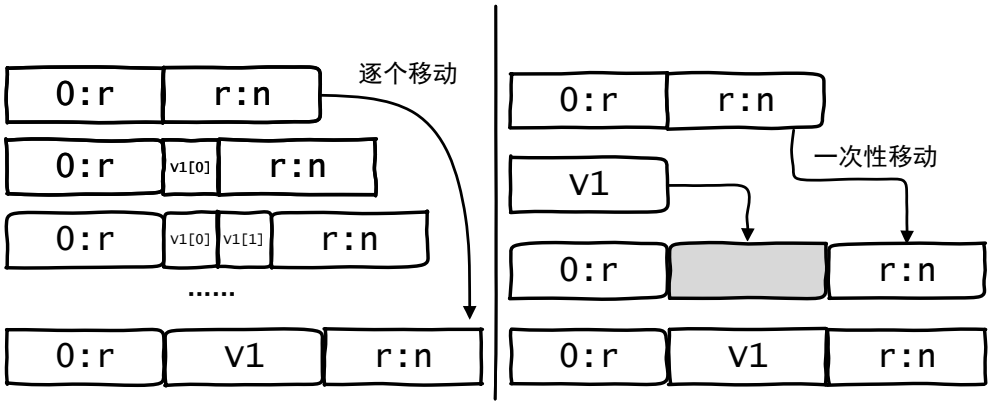

图 2.6　向量合并的两种方法

```
1  // VectorConcatFast
2  Vector<int>& operator()(Rank r) override {
3     V.resize(V.size() + V1.size());
4     std::move_backward(V.begin() + r, V.end() - V1.size(), V.
         end());
5     std::move(V1.begin(), V1.end(), V.begin() + r);
6     return V;
7  }
```

您可以自行分析上述批量插入算法的时间复杂度和平均时间复杂度。因为已经预先reserve，这里resize可以保证在 $O(1)$ 时间里完成。示例程序中进行了四类情况的评估：插入少量元素到开头、插入大量元素到开头、插入少量元素到结尾、插入大量元素到结尾。这里的少量可以认为是常数 $n_1 = O(1)$，而大量可以认为 $n_1 = \Theta(n)$。您可以对比前面得到的时间复杂度，验证您对这两个算法的评估结论。您也可以将上述上述两个示例程序和您自己的实现进行对比。

批量插入的算法中体现出的用块操作代替多次单元操作的思想，在以线性表

为背景的算法设计题中应用广泛。在我们的实现中，插入操作和扩容操作是解耦的：容量已经被预先分配好。在实际情况下，我们也需要同时考虑扩容的成本。按照解耦的实现，在扩容申请了新的数组空间之后，会将原数组的元素复制过去，然后再执行插入，对这些元素进行移动。很显然，$V[r]$ 之后的元素被移动了两次。事实上，如果进行耦合的实现，可以让这些元素只被移动一次：在扩容申请了新的数组空间之后，$V[r]$ 之后的元素可以直接移动到它们的目的位置，为 V_1 里的元素“空出一块空间”。您可以自己实现这个耦合算法。因为需要在外部访问`Vector`里的成员，因此您应当将您的类或函数生声明为友元。

2.5.7　实验：按值删除元素

在2.5.4节，我们讨论的删除是按位置删除，具体到向量就是按秩删除。另一种删除的方式是按值删除，也就是说，我们给定一个元素 e，想要删除向量中和 e 相等的每一个元素。代码可以在 *VectorRemove.cpp* 中找到。同样，建议您自己实现一个算法。

在分析这个问题之前，先对按值操作进行一些深层次的理解。我们知道，数据结构是元素的集合，元素之间是互不相同的，但这并不妨碍它们相等，因为我们可以定义“相等”（反映到 C++ 中，就是重载运算符`operator==`）。比如说，我们定义值相同为“相等”，这样就不需要考虑元素的地址；这是按值删除的思路。顺着这个思路，我们可以扩展按值删除到更加一般的按条件删除。比如说，向量里的元素是一个结构体，我们定义结构体的某个属性相同为“相等”，而其他属性可以不被考虑；此时，某个属性等于给定的值就是我们定义的条件。在 C++ 的 STL 中，按值删除对应的算法是`std::remove`，而按条件删除对应的算法是`std::remove_if`，二者具有高度的相似性。其他的一些算法也有相应的“按条件”版本，您可以根据需要自行了解，这里不再赘述。

下面我们来讨论按值删除元素的问题。为了评估算法的性能，示例程序构造了一个场景：在一个规模为 n 的向量中，第奇数个元素为 1，第偶数个元素为 0，我们的算法将要删除所有的 0，也就是说删除一半的元素。

```
1  class VectorRemove : public Algorithm<size_t, int> {
2  protected:
3     Vector<int> V {};
4     virtual void batchRemove(int e) = 0;
5  public:
6     size_t operator()(int e) override {
7        size_t n = V.size();
8        batchRemove(e);
9        return n - V.size();
10    }
```

```
11      void initialize(size_t n) {
12          V.resize(n);
13          for (size_t i { 0 }; i < n; ++i) {
14              V[i] = i % 2;
15          }
16      }
17  };
```

和上一节一样，我们从最朴素的想法开始：逐个查找、逐个删除。我们在向量 V 中查找要删除的 e，每发现一个就删除一个，直到没有等于 e 的元素为止。如图2.7所示。

```
1   // VectorRemoveBasic
2   void batchRemove(int e) override {
3       Rank r { 0 };
4       while (r = find(begin(V), end(V), e) - begin(V), r < V.size
            ()) {
5           V.remove(r);
6       }
7   }
```

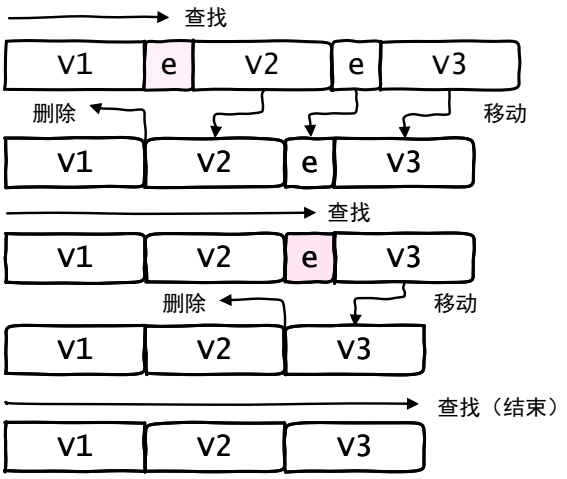

图 2.7　向量按值删除元素：逐个查找、逐个删除

上述朴素算法的空间复杂度是 $O(1)$，但是时间效率是很低的。这里使用的`std::find`，原理和2.5.3节介绍的查找一样，时间复杂度为 $\Theta(r)$。`std::find`返回的是一个迭代器，和起始迭代器`std::begin`相减之后就可以得到找到的位置（秩）。而另一方面，在找到之后，`remove`的时间复杂度为 $\Theta(n-r)$。所以，每次循环的时间复杂度为 $\Theta(n)$。在极端情况下（比如我们设计的实验场景），有 $\Theta(n)$ 个元素要被删除，此时时间复杂度为 $\Theta(n^2)$。考虑到 C++ 的标准算法库在考试中手写代码时通常不能使用（阅卷者可能看不懂），所以如果您不熟练，非常建议您在理解这些标准算法之后把它改写成 C 语言的风格，直到您能熟练地在 STL 和 C 语言风格之间做转换为止。学习的时候使用 STL 的好处是精简代码，减少记忆量，容易抓

主要矛盾。

在设计算法的时候，题目不一定会给出要求的复杂度。这个时候，可以对比一下*相似问题*的复杂度。比如，之前介绍讨论批量插入（向量合并）的时候可以做到线性的时间复杂度，没有理由批量删除（按值删除）需要平方级的时间复杂度。因此，我们需要寻找提高时间效率的切入口。

为了降低时间复杂度，就要设法降低在算法中进行的*不必要工作*。在不必要工作中，有几种比较典型。一种是*不到位工作*，它代表了在算法中，本能够一步到位的计算被拆分成了若干个碎片化的步骤，而产生的时间浪费。比如在向量合并的逐个插入算法中，后缀 $V[r+1{:}n]$ 就接连移动了 n_1 次；我们通过合并这些移动实现了算法优化。

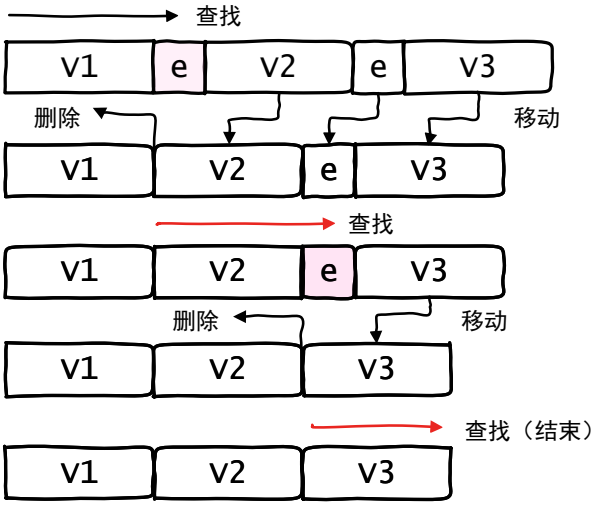

图 2.8　向量按值删除元素：一次查找、逐个删除

另一种是*重复工作*，它代表了在算法中，重复计算了同一算式造成的时间效率浪费。在上面的朴素的按值删除算法中，就有一项非常明显的重复工作。如图2.7所示，第一次检索了第一段 V_1，第二次检索了 V_1+V_2，第三次检索了 $V_1+V_2+V_3$；这里 V_1 被检索了 3 次，V_2 被检索了 2 次，而事实上只需要检索一次即可。在我们查找完毕的时候，可以记录下当前查找到的位置；下一次查找的时候从记录下的位置开始记录，如图2.8所示。借用 C++ 的 STL，我们对朴素算法做很小的改动就可以做到这一点，您可以对比两张图和两份代码。

```
1 // VectorRemoveImproved
2 void batchRemove(int e) override {
3    Rank r { 0 };
4    while (r = find(begin(V) + r, end(V), e) - begin(V), r < V.
        size()) {
5       V.remove(r);
6    }
7 }
```

然而您会发现，在最坏的情况下（所有元素都要被删除），光是remove就要花费 $\Theta(n^2)$ 的时间，上面这个算法的优化程度仍然不够。因此，下一步优化就要从remove入手，需要将remove的工作展开来，看看其中哪些是不必要的。

在remove中，主要消耗时间的是*元素移动*的操作。您可以发现，如果 $V[0{:}i]$ 中有 k 个元素要删除，那么最后一个元素 $V[i-1]$ 就要向前移动 k 次：依次移动到 $V[i-2] \to V[i-3] \to \cdots \to V[i-k-1]$ 的位置上。比如，在图2.8中，V_3 就移动了 2 次。您可以敏锐地发现者正是前面所讲述的*不到位*工作。这一系列的移动被拆成了 k 次，而实际上是可以一步到位，直接从 $V[i-1]$ 移动到目标位置 $V[i-k-1]$ 的。

为什么可以直接移动到目标位置呢?注意到,无论是上面的 Basic 还是 Improved 算法，当检索到 $V[i]$ 的时候，前缀 $V[0{:}i]$ 的所有元素都已经被检索过了，因此 k 的值已经确定了，并且前 $i-1$ 个元素已经移动到了正确的目标位置。所以您可以用归纳法的思路，证明直接移动的正确性。证明完成之后，剩下的就只有编码的工作了。*强烈建议您自己完成这个算法再阅读示例程序*。下面展示了一种实现。

```
// VectorRemoveFSP
void batchRemove(int e) override {
   Rank k { 0 };
   for (Rank r { 0 }; r < V.size(); ++r) {
      if (V[r] == e) {
         ++k;
      } else {
         V[r - k] = move(V[r]);
      }
   }
   V.resize(V.size() - k);
}
```

非常显然，现在时间复杂度被缩减到 $\Theta(n)$ 了。上面这个算法的思路可以被概括为*快慢指针*，这是线性表算法设计中非常典型的技巧。快指针即探测指针，指向 $V[r]$；慢指针即更新指针，指向 $V[r-k]$。快指针找到需要保留的元素，然后将它们移动到慢指针的位置处。如图2.9所示。

您可以通过示例程序的测试明显感知到这三种算法的性能差异。作为结束，上述算法在 C++ 的 STL 中有一个简单的记法：

```
// VectorRemoveErase
void batchRemove(int e) override {
   V.resize(remove(begin(V), end(V), e) - begin(V));
}
```

这里使用了 STL 提供的std::remove，它并不会真正地将 e 删除，而是将非 e 的元素都移动到向量的前半部分，并返回新的尾部迭代器。用户还需要进行一

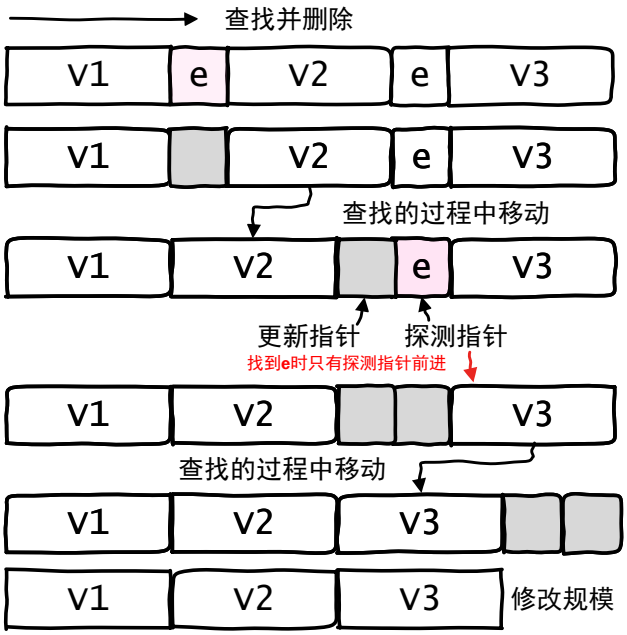

图 2.9　向量按值删除元素：快慢指针

次resize才能真正清除掉已经无效的后半部分，这个真正清除的过程被称为**擦除**（erase）；这个方法也被称为“删除-擦除”法。STL 中的向量容器std::vector提供了原生的erase方法，您可以直接使用它进行按值删除。

2.6　置乱和排序

一般数据结构重点讨论的只有插入、删除和查找三种基本操作，但向量作为一种非常基础的数据结构，经常被用来在考试中作为算法设计题的背景。下面这两个小节分别从*熵增*和*熵减*的角度出发，讨论**置乱**（shuffle）和**排序**（sort）的算法。

2.6.1　实验：随机置乱

通常说的置乱都是指随机置乱。给定一个向量 V 和一个随机数发生器rand，随机打乱向量中的元素。在理论分析的时候，可认为随机数发生器是*理想的*，即每次调用能够随机生成一个*非负整数*。当然现实中的随机数发生器做不到理想，我们将在本小节的末尾讨论它们的区别。

置乱算法接收一个向量将它置乱。代码可以在 *Shuffle.cpp* 中找到。*如果您想到了解决方案，可以先自己实现它。*

```
1 class Shuffle : public Algorithm<void, Vector<int>&> {};
```

直接看这个“向量置乱”的问题，很容易没有头绪。不妨将这个问题迁移到比较熟悉的领域：比如洗牌。想必大家都非常熟悉洗牌。随机置乱的目的和洗牌是

一样的，但如果用洗牌的方法去做随机置乱，即抽出一沓牌、把这沓牌放到牌堆底部、再抽一沓牌，则会面临三个问题：

1. 您不知道重复多少次抽牌比较合理。
2. 在有限次抽牌之后，牌的 $n!$ 种随机次序并不是等概率的。
3. 每次抽牌都要伴随大量的元素移动，时间效率非常低下。

解决随机置乱问题可以从上面的第二个问题，也就是“随机次序等概率”入手。为了保证随机次序是等概率的，那么就要构造 $n!$ 种等可能的情况。根据乘法原理，可以很自然地想到，如果将每种次序表示为一个 n 元随机变量组 $(X_1, X_2, ..., X_n)$，其中 X_i 两两独立，并且 X_i 恰好有 i 个等可能的取值，那么这 $n!$ 种次序就是等可能的了。接下来，只需要建立在全排列和这样的 n 元组的一一对应的映射关系即可。当然，不能直接把全排列用上。全排列的两个元素不是相互独立的，它自身不是符合条件的 n 元组。

为了构造符合条件的映射，又可以采用递归的思想方法：

1. 如果 $n = 1$，全排列和 n 元组可以直接对应。
2. 对于 $n > 1$，考虑 $V[n-1]$ 在打乱后的秩，显然，它可以取 $0, 1, \cdots, n-1$ 这 n 个等可能的值，令这个数为 X_n，然后将 $V[n-1]$ 从打乱前后的向量中都删除，就化为了 $n-1$ 的情况。反复利用这个化归方法，最终可化归到 $n = 1$ 的情况。

以上就成功构造出了满足条件的一一映射关系，您可以在理解它的基础上自己设计相应的随机置乱算法。下面给出了一个示例实现。

```
void operator()(Vector<int>& V) override {
    for (auto i { V.size() }; i > 1; --i) {
        auto j { rand() % i };
        swap(V[i - 1], V[j]);
    }
}
```

显然上面这个算法是时间 $\Theta(n)$、空间 $O(1)$ 的。并且上面的分析表明，如果`rand`真的能随机生成一个非负整数（不是随机生成一个`unsigned int`！），那么这个算法就能将所有的 $n!$ 个排列等概率地输出。此外，C++11 在 STL 中也提供了置乱算法，需要包含`<random>`库使用。

```
class ShuffleStd : public Shuffle {
    default_random_engine m_engine;
public:
    void operator()(Vector<int>& V) override {
        shuffle(V.begin(), V.end(), m_engine);
    }
};
```

这里的默认随机数引擎可以被替换为其他用户定义的引擎，关于随机数引擎的问题和数据结构无关，不再赘述。默认的随机数引擎基于梅森旋转算法，产生随机数的速度较慢，但具有较好的均匀性。

下面回到随机生成器的问题上来，真实的`rand`会受到位宽的限制。如果每次随机生成一个随机的 32 位非负整数，那么 n 次随机一共只有 $2^{32n} = o(n!)$ 种可能的取值，所以在 n 充分大的时候，必然会有一些排列不可能被输出。

另一方面，即使忽略位宽的限制，也不可能做到等概率输出。因为当随机数生成器的返回值是在 $[0, 2^k - 1]$ 中随机生成的非负整数时，$n!$ 在 $n \geqslant 3$ 时不是 2^k 的因子（不论 k 有多大），所以这 $n!$ 个排列不可能是等概率的。不过，当 n 比较小时概率可以认为近似相等，您可以在示例程序的运行结果中看出这一点。

除此之外，现实中的`rand`是伪随机。对于同一个种子，生成的伪随机序列是相同的，所以并不能真正“随机”地打乱向量中的元素。当然，这是另一个话题了。当我们在后面的章节讨论散列的时候，再对伪随机问题进一步探讨。

2.6.2　偏序和全序

在讨论完置乱问题之后，接下来讨论排序问题。在具体介绍排序算法前，首先需要界定清除，**序**（order）是一个什么东西。在上一章定义过良序的概念，但要对一个向量做排序，并不一定要要求它的元素是某个定义了良序关系的类型。比如说，n 个实数同样可以关于熟知的“$\leqslant$”排序。因此，需要引入条件更松的序关系的定义。

将良序关系定义中的第 4 个条件（最小值）去掉，就变成了**全序**（total order）关系。如果集合 S 上的一个关系 $\preceq$ 满足：

1. **完全性**。$x \preceq y$ 和 $y \preceq x$ 至少有一个成立。
2. **传递性**。如果 $x \preceq y$ 且 $y \preceq z$，那么 $x \preceq z$。
3. **反对称性**。如果 $x \preceq y$ 和 $y \preceq x$ 均成立，那么 $x = y$。

那么称 $\preceq$ 是 S 上的一个**全序关系**。显然良序关系是全序关系的子集。

和良序关系相比，全序关系更加符合常规的认知。比如，实数集上熟知的“$\leqslant$”就是全序关系。由于完全性的存在，凡是具有全序关系的数据类型，都可以进行排序；反之，在《数据结构》里的“通常意义的排序”问题中，都假定数据结构中的元素数据类型具有“先验的全序关系”。

在 C++ 中，排序函数`std::sort`接受三个参数，其中第三个参数就表示“自定义的全序关系”。基本数据类型（如`int`和`double`），以及一些组合类型（如`std::tuple`）定义了内置的全序关系（即熟知的“$\leqslant$”），但也可以使用其他的全序关系进行排序。其他编程语言中的排序函数也有类似的设计。

除了全序关系之外，还有一种序关系在《数据结构》中也经常会提到：**偏序**（partial order）关系。

如果集合 S 上的一个关系 $\preceq$ 满足：

1. **自反性**。$x \preceq x$。
2. **传递性**。如果 $x \preceq y$ 且 $y \preceq z$，那么 $x \preceq z$。
3. **反对称性**。如果 $x \preceq y$ 和 $y \preceq x$ 均成立，那么 $x = y$。

那么称 $\preceq$ 是 S 上的一个**偏序关系**。偏序关系和全序关系相比，第 1 个条件（完全性）变成了更简单的自反性；也就是说，并不是 S 中的任意两个元素都能进行比较。比如，令 S 为“考生组成的集合”，$\preceq$ 定义为“考生 x 的每一门分数都小于等于考生 y”。您可以轻易验证，这个关系是偏序关系但不是全序关系。

在计算机编程中直接定义偏序关系是不方便的，因为 $\preceq$ 的返回值往往是`bool`类型，不存在`true`和`false`之外的第三个选项（*无法比较*）。并且，无法比较的情况不能随意地返回一个`true`或`false`的值，因为这可能导致传递性被破坏。所以，当在编程时需要定义一个偏序关系时，往往会将它扩展成一个全序关系。比如，给 S 中的所有元素做标号，当已有的偏序关系无法比较时，则根据标号的大小进行比较。扩展成全序关系之后，就可以进行排序了。由扩展成的全序关系的不同，可能会产生不同的排序结果。

在 C++20 中定义了各种序关系，用于作为航天飞机运算符`operator<=>`的返回值。比如，偏序关系被定义为`std::partial_ordering`，这个类型包括大于、小于、等价以及无法比较四种比较结果。除此之外，还提供了包括大于、小于和等于的强序关系`std::strong_ordering`和包括大于、小于和等价的弱序关系`std::weak_ordering`。这两者都可以解释为全序关系，区别在于“等于”和“等价”，或者说“相同”和“相等”。强序关系不允许两个不同的元素相等（这是非常强的条件），而弱序关系允许。

2.6.3　归并排序

现在回到向量排序的问题。对于一个线性表，如果它的数据类型是全序的；且对其中的任意一个元素 x，和 x 的*后缀*中的任意一个元素 y，总是有 $x \preceq y$，则称它是**有序的**（ordered）。对于无序线性表，通过移动元素位置使其变为有序的过程，称为**排序**（sort）。在计算机领域所说的有序，一般都是指*升序*。所以在上面的定义中使用的是*后缀*。如果您想要讨论降序或者其他的顺序（比如按最小素因子排序），只需要重新定义全序关系 $\preceq$，即可以回归为升序的情况进行处理。

```
1 template <typename T, template<typename> typename Linear,
    typename Comparator = std::less<T>>
```

```
    requires std::is_base_of_v<AbstractLinearList<T, typename
      Linear<T>::position_type>, Linear<T>>
class AbstractSort : public Algorithm<void, Linear<T>&> {};
```

在上面的程序中，我们规定了一个排序的抽象模板类，它接收一个线性表（通过概念限制）对其进行排序。因为排序的结果一定是一个序列，所以我们只考虑线性表。三个模板参数分别为元素类型、数据结构类型和规定的序关系。约定俗成地，我们通常使用严格的序关系，比如 $<$，而不是不严格的 $\leqslant$。C++ 提供了模板`std::less`来表示使用类型`T`定义的小于运算符，用户也可以自定义仿函数传入这个模板中。

排序是计算机领域最重要的算法之一。在计算机出现至今，人们提出了各种各样的排序算法，并且仍然有不少研究者在从事着排序算法的研究。在《数据结构》中，将专门有一章讨论各种排序算法。在本节，先介绍一种最基本、最经典的排序方法：**归并排序**（merge sort）。归并排序的发明人是大名鼎鼎的冯·诺依曼，这位“计算机之父”在 1945 年设计并实现了该算法。

归并排序的设计采用的仍然是递归的思想：

1. 规模 $n \leqslant 1$ 的向量总是天然有序的。
2. 对于规模 $n > 1$ 的向量，可以将其分成前后两部分，长度分别为 $\frac{n}{2}$ 和 $n - \frac{n}{2}$（在上一章您见过这个分法），从而将规模为 n 的问题化归为两个规模较小的子问题。这些子问题可以继续递归下去直到化为 1。解决子问题之后，V 的前半部分和后半部分分别有序，只需要将这 2 个有序序列合并为 1 个有序序列，就可以解决原问题了。这一合并的过程就称为**归并**（merge）。

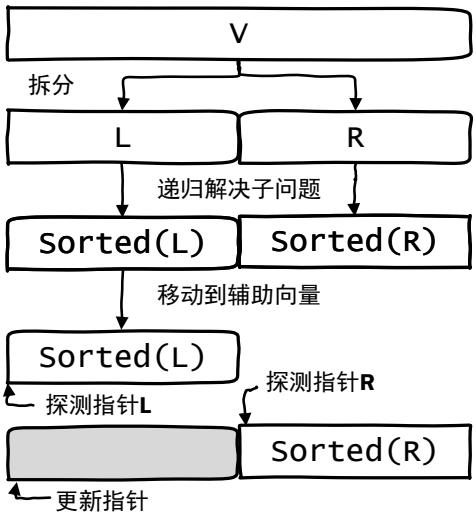

图 2.10　向量的归并排序

您可以根据上面的思想，自己实现一个归并排序的算法（一个建议是，将归

并的过程提取为单独的函数以提高可读性），然后和下面的示例算法进行比较。这个代码比较长。最好自己先写一份代码，因为直接读示例代码很难记住。注意，在《数据结构》部分，代码的记忆既不是重点也没有必要。归并排序这个知识点的核心是上面的这一段文字，即归并的思想。

下面的示例代码既可以用于向量，又可以用于后面的章节介绍的列表。

```
template <typename T, template<typename> typename Linear,
   typename Comparator = std::less<T>>
class MergeSort : public AbstractSort<T, Linear, Comparator> {
   Vector<T> W;
   Comparator cmp;
   using Iterator = typename Linear<T>::iterator;
   void merge(Iterator lo, Iterator mi, Iterator hi, size_t
      size) {
      W.resize(size / 2);
      std::move(lo, mi, W.begin());
      auto i { W.begin() };
      auto j { mi }, k { lo };
      while (i != W.end() && j != hi) {
         if (cmp(*j, *i)) {
            *k++ = std::move(*j++);
         } else {
            *k++ = std::move(*i++);
         }
      }
      std::move(i, W.end(), k);
   }
   void mergeSort(Iterator lo, Iterator hi, size_t size) {
      if (size < 2) return;
      auto mi { lo + size / 2 };
      mergeSort(lo, mi, size / 2);
      mergeSort(mi, hi, size - size / 2);
      merge(lo, mi, hi, size);
   }
public:
   void operator()(Linear<T>& L) override {
      mergeSort(std::begin(L), std::end(L), L.size());
   }
};
```

C++ 提供了`std::inplace_merge`函数（传入 lo、mi 和 hi 的迭代器）来进行就地归并，您可以用这个函数代替上述手写的`merge`方法。归并排序的算法中有很多细节可以挖掘，下面列出了其中的一部分。在阅读并理解上述归并排序的算法的基础上，您可以尝试回答以下问题（其中，假定比较函数`cmp`的时间、空间复杂度都是 $O(1)$ 的），然后再查看后面的分析。

在归并的一开始，将前半部分移动到了辅助向量中。为什么前半部分需要移动出去，而后半部分不需要？

在归并的过程中，事实上也采用了快慢指针的思想。快指针（探测指针）是 j，慢指针（更新指针）是 k。还有一个探测指针是 i，不过它工作在辅助向量 W 上。如果前半部分不移动的话，i 也会工作在原向量上，它和 k 不构成快慢关系。于是，一旦后半部分比前半部分的元素小，就会把前半部分的元素覆盖掉；而对后半部分而言，除非前半部分已经全部加入到原向量中，否则 j 永远能够在 k 前面。而当前半部分全部加入之后，$i < \mathrm{mi} - \mathrm{lo}$ 不再成立，循环结束。

在归并的最后，我们将前半部分（此时在辅助向量里）多余的元素移动回原向量。为什么后半部分多余的元素不需要？

后半部分的数据没有移动出去，如果前半部分的元素已经全加入到原向量了，则后半部分剩余的元素已经在它们应该在的位置上，不需要再移动了。

辅助向量 W 的长度至少是多少？并由此确定归并排序的空间复杂度。

辅助向量 W 的长度至少为最大的 $\mathrm{mi} - \mathrm{lo}$，也就是 $\frac{n}{2}$，因此空间复杂度为 $\Theta(n)$。这里需要注意，递归产生的 $\Theta(\log n)$，相比于辅助数组的 $\Theta(n)$ 来说可以忽略；但在解答题的场合，应当书写在解题过程中表示考虑到了此种情况。

归并排序的时间复杂度是多少？如果划分的时候，左半部分不取 $\frac{n}{2}$ 而是取 kn（其中 $0 < k < 1$），时间复杂度又会变成多少？如果取作 $\max\left(\frac{n}{2}, C\right)$，其中 C 是一个给定的常数，那么时间复杂度又会变成多少？

这个问题是归并排序相关的一个经典问题，考试中也可能出现。对于原始的归并排序（折半二分），您可以列出 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ 的方程，递降计算出 $T(n) = \Theta(n \log n)$。当取 kn（定比二分）时做法类似，时间复杂度不会变，但常数会增加，您可以自行计算。如果左半部分的长度存在上限 C（定长二分），则在 n 充分大时，$T(n) = T(n - C) + T(C) + \Theta(n) = \Theta(n^2)$。因此两部分的划分必须按比例取，而不能受到某个固定值 C 的限制。

此类问题通常可以用**主定理**（master theorem）解决。主定理是用来处理分治算法得到的递归关系式的“神兵利器”。它的证明太过复杂，这里只陈述结论。

定理 2.1 (主定理)： 设 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$，则：

1. 若 $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$，其中 $\varepsilon > 0$，则 $T(n) = \Theta\left(n^{\log_b a}\right)$。
2. 若 $f(n) = \Theta\left(n^{\log_b a}\right)$，则 $T(n) = \Theta\left(n^{\log_b a} \log n\right)$。
3. 若 $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$，其中 $\varepsilon > 0$，且 $\overline{\lim_{n \to \infty}} \frac{f\left(\frac{n}{b}\right)}{f(n)} < 1$，则 $T(n) = \Theta\left(f(n)\right)$。

2.6.4　基于比较的排序的时间复杂度

归并排序是一种**基于比较**（comparison-based）的排序。所谓基于比较，就是在算法进行过程的每一步，都依赖于元素的比较（即调用`cmp`）进行。基于比较的

排序是针对全序关系设计的。大多数的排序算法都是基于比较的。还有一些不基于比较的排序，它们不是针对待排序数据类型的全序性设计的，而是针对待排序数据类型的其他性质设计的，因而应用范围会更小。在后文中会介绍一些不基于比较的排序。

下面将说明一个重要结论：

定理 2.2： 基于比较的排序在最坏情况下的时间复杂度为 $\Omega(n \log n)$。

这是本书中第一次使用信息论方法，讨论时间复杂度的最优性。信息论方法在考试中通常不会直接考到，但用信息论的思路，有助于在算法设计题中判断自己是否能得到时间复杂度层面上的满分。此外，信息论方法也对记忆知识点有帮助。

1. 因为是最坏情况，不妨假设所有元素互不相等。在排序算法开始之前，这 n 个元素可能的顺序关系有 $n!$ 种，而在排序算法开始之后，这 n 个元素可能的顺序关系只有 1 种（因为已经找到了它们的顺序）。
2. 另一方面，每次比较都有两种结果（`if`分支和`else`分支）。剩下的可能的顺序关系被分为 2 个部分，根据比较结果，只保留其中的 1 个部分。在最坏情况下，每次保留的都是元素较多的部分，从而每次比较至多排除一半的可能。

综合以上两点，至少需要进行 $\log_2(n!) = \Theta(n \log n)$ 次比较，于是就证明了上述的定理。最后一步的结论基于一个重要的公式：

定理 2.3 (斯特林公式)： 在 $n \to \infty$ 时，$n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$。

这一公式的证明是纯数学的话题。感兴趣的话可自行在网上查找，这里不再叙述。在《数据结构》的学习中需要记住的公式并不多，斯特林（Stirling）公式是必须记住的公式之一。或者，您也可以只记住 $\log(n!) = \Theta(n \log n)$，因为这是它的主要应用。

由此可见，归并排序在基于比较的排序中，已经达到了最优的时间复杂度。当然，空间复杂度不是最优的，它需要 $\Theta(n)$ 的额外空间。关于排序的更多性质，在后面的专门章节中将继续分析。从这个时间复杂度下界也可以看出，排序需要付出的努力总是比置乱要高，这也符合我们对信息的一般认知：排序是熵减的过程，而置乱是熵增的过程，熵减总是要付出更多的努力。

2.6.5　信息熵 *

基于上一小节的讨论，本小节对香农（Shannon）提出的**信息熵**（information entropy）概念进行简要的介绍。在使用信息论判断复杂度问题时，并不一定需要使用信息熵去定量计算，因此如果您不感兴趣，也可以跳过本节。

在信息熵提出之前，人们很难定量地衡量一份信息所包含的信息量。香农创造性地引入概率论和热力学中的熵的概念，对信息的多少进行了定量描述。对于

一个信息来说，在我们识别之前，会对它有一些先验的了解。如果我们已经先验地确切知道这个信息的内容，那么这个信息就完全是无效信息，所蕴含的信息量是 0；反过来，我们对这个信息的先验了解越少、越模糊，这个信息所蕴含的信息量越丰富。

假设一个信息在我们已知的先验了解下，共有 n 种可能发生的情况。则对于每个情况，设其发生的概率为 p，我们定义该情况的不确定性为 $-\log p$。对于一个信息整体，我们考虑它所有可能发生的情况的*平均不确定性*（即数学期望），定义其为该信息的信息熵，即

$$H = \mathrm{E}(-\log p) = -\sum_{i=1}^{n} p_i \log p_i$$

现在联系上一小节讨论的场景。在没有其他先验信息的情况下，一个乱序序列出现 $n!$ 种排列的可能性是相等的，所以它的信息熵为：

$$H = -\log\left(\frac{1}{n!}\right) = \Theta(n \log n)$$

在排序结束时，序列只有唯一的可能性，所以信息熵为 0。在最坏情况下，我们进行 1 次比较-判断可以最多消除一半的不确定性，由于取了对数，所以一次判断能造成的熵减是 $O(1)$ 的。综上所述，基于比较的排序的最坏时间复杂度是 $\Omega(n \log n)$ 的。

需要注意的是，即使信息熵的初值和终值都为 0，也不代表可以在 $O(1)$ 的时间内完成算法。比如，对于完全倒序的序列来说，它的信息熵也为 0，但是将其进行排序需要 $\Theta(n)$ 的时间对序列进行倒置。所以，信息熵方法通常只能求出一个理论边界，并不代表这个边界是可以达到的。

2.6.6　有序性和逆序对

前面两个小节的讨论显示，对于没有任何先验信息的乱序序列来说，它的信息熵是 $\Theta(n \log n)$ 的，因此基于比较的排序在最坏情况下，永远不可能突破这一时间复杂度限制。但是，如果我们事先了解到了一些先验信息，初始状态的信息熵就会下降，从而在理论上可以以更低的时间复杂度进行排序。

一个比较常见的情况是“基本有序”的条件。对于基本有序，通常有两种理解方式。

1. 认为基本有序就是信息熵很低的状态。我们知道，信息熵对应了信息的不确定性，也就是“无序性”，信息熵比较高的序列无序性也比较高。因此，反过

来也可以认为，信息熵比较低的序列基本有序。

2. 认为基本有序指的是**逆序对**很少的状态。对于一个序列 $A[0{:}n]$，如果 $i < j$ 但 $A[i] \succ A[j]$，则称 (i, j) 是一个逆序对。采用逆序对对序列有序性进行刻画，和信息论方法是分离的（因为一次交换可以最多消除 $\Theta(n)$ 个逆序对），但可以对顺序和倒序进行区分，在分析算法时常常可以起到重要的作用。

信息熵和逆序对都是我们分析和解决排序问题的方法。信息熵的视角更加宏观，我们很难计算每一步操作削减了多少信息熵，因此它往往用于复杂度层面上的分析；而逆序对的视角更加微观，很容易计算每一步操作消除了几个逆序对，所以可以用于具体的算法性质分析。下面我们从逆序对的角度回顾归并的过程。每次向更新指针的位置移动一个元素：

1. 如果被移动的元素来自于前半部分的探测指针，那么不会对逆序对的数量产生影响。
2. 如果被移动的元素来自于后半部分的探测指针，那么我们消除了它和前半部分剩余元素之间的逆序对。也就是说，我们消除的逆序对数量等于前半部分的剩余元素数量。

根据这一性质，我们可以在归并排序的过程中统计逆序对的数量。因为在前半和后半之一的元素被用尽之后，后半元素不需要移动，而前半元素需要从辅助空间中移回；所以，上面讨论的情况（2）会比情况（1）有数量更多的移动。因此我们可以看出，归并排序在处理完全倒序的序列时，尽管它的信息熵为 0，但排序算法并不能发现这一先验信息，而是会进行更多的移动。

2.6.7　实验：先验条件下的归并排序

下面讨论一个基本有序的场景：如果向量已经基本有序，只有开头的长度为 L（未知）的一小段前缀是乱序的（即前缀外全部有序，且前缀中的元素都比前缀外的元素小），如何改进我们的归并排序，让它可以有更高的时间效率？改进之后的时间复杂度是多少？

这个问题的示例代码可以在 *VectorMergeSort.cpp* 中找到，您可以根据自己的理解改写归并排序，并和示例代码进行比较。

这个问题也是一个排序经典问题。我们可以用信息熵和逆序对两种方法对这个场景进行分析。我们可以发现在这个场景中，乱序前缀之外的所有元素既不会提供信息熵也不会提供逆序对，因此在给定的先验条件下，序列的信息熵为 $\Theta(L \log L)$，逆序对数量为 $O(L^2)$。而原有的归并排序算法在最好情况下，时间复杂度也是 $\Theta(n \log n)$ 的。所以必须要改进。改进的时候，可以针对已知的方程

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ 做优化。

最直接的想法是从递归方程的目标入手，也就是将 $T(n)$ 替换为 $T(L)$。从信息熵和逆序对的角度我们都可以发现，问题中的特殊场景相当于把排序问题的规模从 n 降低到了 L。我们可以从后向前遍历整个向量，以确定 L 的值。这种方法看起来非常直观，但确定 L 并没有那么简单。示例代码中给出了一个样例，它通过 $\Theta(n)$ 的时间找到 L，您可以自己寻找解决方法并和它对比。

```
1  void operator()(Linear<T>& L) override {
2     Rank mid { L.size() - 1 };
3     while (this->cmp(L[mid - 1], L[mid])) {
4        if (--mid == 0) return;
5     }
6     auto max_left { *max_element(begin(L), begin(L) + mid) };
7     auto left { lower_bound(begin(L) + mid, end(L), max_left,
         this->cmp) };
8     this->mergeSort(begin(L), left, left - begin(L));
9  }
```

另一个思路是从递归方程的形式入手。注意到，在这个方程中，递归项 $2T\left(\frac{n}{2}\right)$ 只要不改动递归方式，就是没法做优化的；而余项 $\Theta(n)$ 是有机会被优化的。需要在“比较好的情况”（即题中给出的“基本有序”的情况）下，让 $\Theta(n)$ 变得更小。归并排序在“将前半部分移动到辅助空间”的时候，就已经需要付出 $\Theta(n)$ 的时间。所以，我们需要在归并的最开始进行一次判断，判断是否可以不将前半部分移动到辅助空间：只需要判断前半部分的结尾是否小于后半部分的开头就可以。在我们之前实现的归并算法中，只需要加入下面的一行代码：

```
1  if (cmp(*(mi - 1), *mi)) return;
```

这样，如果归并前的序列已经有序，就不需要进行归并。于是，对于不需要归并的部分，递归方程就变化为 $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$，也就是 $\Theta(n)$。而需要归并的部分由题意，长度不超过 L，在这部分利用前面获得的结论，就可以得到时间复杂度为 $\Theta(L \log L)$。因此，改进后的时间复杂度为 $\Theta(n + L \log L)$。

由于归并排序的实际时间性能还和很多其他的因素相关，所以本书提供的实验代码只能大致地进行定性分析。如果您想要得到更加精准的分析结果，应当多次随机取平均值。我们可以从实验结果中看到，在题目给定的条件（前缀乱序）下，两种改进策略的时间性能差不多；考虑到递归的消耗，如果 L 不大，从递归目标入手的方法会快一些。同时，在 L 不大的时候，两种改进策略的性能都显著高于未改进的版本。当然，如果没有先验信息即 $L = n$ 的时候，改进策略有些时候会因为额外的计算，性能反而低于未改进的版本。

相对来说，从递归方程入手的方法虽然在原题中性能不一定能和直接定位 L

的方法相比，但具有更好的泛用性。我们可以测试一个对偶问题：有且只有后缀是乱序的问题。在这个对偶问题上，从递归方程入手的方法仍然可以做到高效率，而从递归目标入手的方法则因为无法识别先验信息，和未改进的版本性能相若。

2.7　有序向量上的算法

2.7.1　折半查找

排序之后得到的有序向量，在查找时有额外的优越性。排序之后要执行查找操作，就不再需要一个一个元素看是否相等了。类似排序，我们首先建立针对有序线性表的抽象查找类。

```
template <typename T, template <typename> typename Linear =
   DefaultVector, typename Comparator = std::less<T>>
   requires std::is_base_of_v<AbstractLinearList<T, typename
      Linear<T>::position_type>, Linear<T>>
class AbstractSearch : public Algorithm<typename Linear<T>::
   position_type, const Linear<T>&, const T&> {};
```

这里可以使用刚才介绍的，基于比较的算法思路。将被查找的元素 e 和向量中的某个元素 $V[i]$ 比较，比较结果有 2 种：

1. 如果 $V[i] > e$，那么只需要保留 $V[0{:}i]$ 作为新的查找区间。
2. 如果 $V[i] \leqslant e$，那么只需要保留 $V[i{:}n]$ 作为新的查找区间。

当取 $i = \frac{n}{2}$（**折半**）时，可以保证新的查找区间长度大约是原来的一半，从而在 $\Theta(\log n)$ 的时间里完成查找。所以这个思路称为**折半查找**。当然，也存在其他二分的方法（i 取其他值），参见后面的《查找》一章。

您可以借助上面的设计或者自己的理解，设计折半查找的算法。下面给出了一个使用递归的示例代码，它实现了刚才的设计。

```
template <typename T, typename Comparator = std::less<T>>
class BinarySearchRecursive : public AbstractSearch<T,
   DefaultVector, Comparator> {
   Comparator cmp;
   Rank search(const Vector<T>& V, const T& e, Rank lo, Rank hi
      ) const {
      if (hi - lo <= 1) {
         return lo < V.size() && cmp(V[lo], e) ? hi : lo;
      }
      Rank mi { lo + (hi - lo) / 2 };
      if (cmp(e, V[mi])) {
         return search(V, e, lo, mi);
      } else {
         return search(V, e, mi, hi);
      }
   }
```

```
15 public:
16    Rank operator()(const Vector<T>& V, const T& e) override {
17       return search(V, e, 0, V.size());
18    }
19 };
```

上面的算法，时间复杂度和空间复杂度均为 $\Theta(\log n)$，您可以自己证明。

查找是算法设计的重点。在设计的时候，需要尤其注意多个相等元素的时候是返回秩最大、秩最小还是任意一个，以及查找失败的时候返回何种特殊值。如果是无序向量，正如2.5.3节那样，那么在查找失败的时候很自然地会返回一个无效值，比如说 -1 或者 V.size()。但是在有序向量的情况下，即使查找失败，我们也可以返回一些有意义的值，向调用者传递一些额外的信息。下面以前面的折半查找算法为例，分析查找成功和查找失败的情况下返回值的设计。

因为如果 $e < V[\text{mi}]$ 就只保留前半段 V[lo:mi]，所以我们在任何时刻都可以保证，$V[\text{hi}] > e$（可认为初始值 $V[V.\text{size}()] = +\infty$）。另一方面，因为另一边的比较是不严格的，所以我们保证的是 $V[\text{lo}] \leqslant e$；注意，这个式子只有 $\text{lo} \neq 0$ 也就是 lo 被修改过一次之后才成立。

而当进入递归边界的时候，V[lo:hi] 有且只有一个元素 V[lo]。此时，可以分成小于、等于、大于三种情况讨论。

1. 如果 $V[\text{lo}] < e$，那么我们可以定位到 $V[\text{lo}] < e < V[\text{hi}]$，此时返回 hi，就是大于 e 的第一个元素。
2. 如果 $V[\text{lo}] = e$，那么我们直接返回 lo，符合查找算法的期待；并且，由于 $e < V[\text{hi}]$，所以如果有多个等于 e 的元素，则返回的是最大的秩。
3. 如果 $V[\text{lo}] > e$，根据前面的分析可知，这种情况只可能发生在 $\text{lo} = 0$ 的情况。于是，e 小于向量中的所有元素，此时返回 lo，也是大于 e 的第一个元素。

综上所述，我们验证了上述折半查找算法的正确性，并且在查找成功时，返回的是最大的秩，在查找失败时，返回的是大于 e 的第一个元素的秩。二分算法在设计的时候非常容易出错；当您自己设计二分算法的时候，也可以使用上面的思考流程来分析自己的算法。

2.7.2　消除简单尾递归

查找 V[0:n] 中某个元素 e 的下标，这个问题在计算前有 $n+1$ 种（包括 -1）可能的结果，计算后有 1 种确定的答案，因此从信息论的角度讲，最坏时间复杂度一定是 $\Omega(\log n)$ 的。但空间复杂度并不一定要是 $\Omega(\log n)$。在这一小节，将介绍一种叫做**消除尾递归**的技术，使用这个技术，可以将上面的折半查找算法的空间复杂度降为 $O(1)$。

如果一个递归函数只在返回（return）前调用自身，则称其为**尾递归**（tail recursion）。特别地，如果在返回前只调用自身至多一次，则称为**简单尾递归**。在实际的编程过程中，简单尾递归在通常会被编译器自动优化。

本小节只介绍对于简单尾递归的消除方法，其他类型的递归消除将在后文中讨论。对于简单尾递归，只需要将递归函数的参数作为循环变量，就可以将其改写为不含递归的形式，从而降低空间复杂度。下面的模板是典型的简单尾递归。

```
R operator()(Args... args) override {
   if ((*pred)(args...)) {
      return (*bound)(args...);
   }
   return apply(*this, (*next)(args...));
}
```

示例代码可以在 *VectorSearch.cpp* 中找到，这里只提取出关键部分。如果您对模板元编程的技巧不感兴趣，下面的说明略读即可。其中，需要传入三个仿函数指针`pred`（用于判断是否进入递归边界的谓词）、`bound`（用于在递归边界上生成返回值）和`next`（用于在非递归边界上生成下一个递归调用的参数）。`std::apply`用于将`next`返回的元组再次传递到仿函数中进行调用。此外，递归参数往往是值而不是引用。引用总是作为整个递归过程中共享的参数，如折半查找中的向量 V 和待查找元素 e，而不是每次递归调用时传递的参数。因此，我们可以直接用值传递`args...`，而不需要使用`std::forward`进行完美转发。上面的递归算法可以改写成如下的迭代算法，其中`std::tie`表示元组的绑定赋值。

```
R operator()(Args... args) override {
   while (!(*pred)(args...)) {
      tie(args...) = (*next)(args...);
   }
   return (*bound)(args...);
}
```

2.7.3　实验：迭代形式的折半查找

您可以利用上面的简单尾递归模板，将折半查找递归形式进行拆解，分解出`pred`、`bound`和`next`，然后代入到上面的迭代模板中，*将其改写为迭代形式*。示例代码仍然在 *VectorSearch.cpp* 中。比如，`next`可以实现为：

```
tuple<Rank, Rank> operator()(Rank lo, Rank hi) override {
   Rank mi { lo + (hi - lo) / 2 };
   if (cmp(e, V[mi])) {
      return { lo, mi };
   } else {
      return { mi, hi };
   }
```

```
}
```

下面是迭代形式的一个示例程序，它和最初的递归形式是完全等价的。

```
Rank operator()(const Vector<T>& V, const T& e) override {
   Rank lo { 0 }, hi { V.size() };
   while (hi - lo > 1) {
      Rank mi { lo + (hi - lo) / 2 };
      if (cmp(e, V[mi])) {
         hi = mi;
      } else {
         lo = mi;
      }
   }
   return lo < V.size() && cmp(V[lo], e) ? hi : lo;
}
```

在示例的测试程序中，我们构造了长度为 n 的向量，将其随机赋值为某个区间的数并排序，然后重复 10^4 次查找（这是因为单次查找的效率太高，无法正确反映各个算法的性能差异）。我们会发现，由于进行了很多抽象，使用模板的方法性能会显著低于直接递归或迭代的性能；而递归和迭代之间相差无几（编译器对递归的版本进行了自动优化）。但即使成模板，它作为对数复杂度的算法，效率仍然远远高于在2.5.3节中讨论的顺序查找，*您可以自己实现一个基于顺序查找的算法参与对比。*

在本节的最后再次强调，现代编译器通常可以在编译的过程中自动消除简单尾递归，所以在实际上机编程时，不需要刻意将简单尾递归改写成循环形式。但在《数据结构》中分析算法的时候，不应该考虑编译器做的优化，所以在算法设计题中，如果出现了未被改写的简单尾递归，就会有被扣空间分的风险。

2.7.4　实验：向量唯一化

下面讨论**唯一化**（unique）问题。给定一个向量，我们希望删除它中间*相等*的重复元素，只保留秩最小的那一个。这里再次看到了*相等*和*相同*的区别，数据结构中是不可能有相同的元素的，而相等的元素我们可以用它的地址（位置、秩）来区分。代码可以在 *VectorUnique.cpp* 中找到。

```
template <typename T>
class VectorUnique : public Algorithm<void, Vector<T>&> {};
```

一个简单但有效的解法是：从左到右考察 V 中的每个元素，删除这个元素的后缀中，所有和它相等的元素。*在看下面的代码前，您可以自己实现这个算法。*

```
// VectorUniqueBasic
void operator()(Vector<T>& V) override {
   for (Rank r { 0 }; r < V.size(); ++r) {
```

```
4         V.resize(remove(begin(V) + r + 1, end(V), V[r]) - begin(
              V));
5     }
6 }
```

在上面的算法里，利用了我们在2.5.7节中使用的“删除-擦除”法一次性地删除后缀里的所有重复元素；因此，最好情况下（所有元素都相等），可以达到 $\Theta(n)$ 的时间复杂度。虽然按值删除达到了 $\Theta(n-r)$ 的时间复杂度，但是因为 r 需要遍历整个向量，所以在最坏情况下（即所有元素都互不相同）需要进行 $\Theta(n^2)$ 次比较。

值得一提的是，如果按值删除的时候采用的是最坏情况 $\Theta(n^2)$ 的朴素（逐个删除）算法，那么唯一化在最坏情况下时间复杂度仍然是 $\Theta(n^2)$，并不会来到 $\Theta(n^3)$。这是初学者容易出现的错误，即直接把循环内外的时间复杂度相乘。这种天真的想法可能是来自于公式 $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$。这个公式本身没有问题，但我们联系一下概率论里的 $P(A)P(B) = P(AB)$ 的条件就能发现端倪：如果内层循环和外层循环是有联系的（不独立的），那么就不能直接相乘。比如，在上面的代码中，外层循环的 r 的区间是 0 到 V.size()；然而，V.size() 并不是一个常量 n，在内层循环中它会发生变化。在使用朴素算法进行按值删除的时候，删除的元素越多，花费的时间越长，但同时缩小的向量规模也就越多，减少的外层循环的轮数也就越多。您可以在这个基础上，完成对朴素按值删除情况下，上述唯一化算法的时间复杂度分析。

在有序向量的情况下，问题可以得到进一步的简化。相等的元素总是排在连续的位置的。所以，可以通过快慢指针的方法。快指针一次经过一片相等的元素，而慢指针保留这些元素中的第一个（这个算法的前提条件并不是有序，只要求相等的元素排在一起即可，但通常建立这一条件的方法是排序）。您已经见过好几次快慢指针的方法，应该可以自己写出来。下面是一个示例程序。

```
1  // VectorUniqueFSP
2  void operator()(Vector<T>& V) override {
3     Rank r { 0 }, s { 0 };
4     while (++s < V.size()) {
5        if (V[r] != V[s]) {
6           V[++r] = move(V[s]);
7        }
8     }
9     V.resize(++r);
10 }
```

这个算法的时间复杂度是 $\Theta(n)$ 的。在 STL 中，提供了`std::unique`来进行唯一化操作，它同样要求相等的元素排在一起。`std::unique`和`std::remove`一样不提供擦除功能，需要再进行`resize`。

如果定义了全序关系（即可排序），那么无序向量的唯一化可以化归到有序向量的情况进行处理：先进行一次 $\Theta(n \log n)$ 的排序，再用有序向量唯一化。但在排序的时候，会损失“元素原先的位置”这一信息，所以需要开辟一个额外的 $\Theta(n)$ 的空间保存这一信息，以在唯一化之后能够顺利还原。您可以尝试实现这样的算法，并和给出的示例算法相比较。它的时间复杂度为 $\Theta(n \log n)$，优于前面的 VectorUniqueBasic；但需要引入辅助向量，空间复杂度为 $\Theta(n)$。

```
template <typename T>
class VectorUniqueSort : public VectorUnique<T> {
   struct Item {
      T value;
      Rank rank;
      bool operator==(const Item& rhs) const {
         return value == rhs.value;
      }
      auto operator<=>(const Item& rhs) const {
         return value <=> rhs.value;
      }
   };
   Vector<Item> W;
   void moveToW(Vector<T>& V) {
      W.resize(V.size());
      for (Rank r { 0 }; r < V.size(); ++r) {
         W[r].value = move(V[r]);
         W[r].rank = r;
      }
   }
   void moveToV(Vector<T>& V) {
      V.resize(W.size());
      for (Rank r { 0 }; r < W.size(); ++r) {
         V[r] = move(W[r].value);
      }
   }
public:
   void operator()(Vector<T>& V) override {
      moveToW(V);
      stable_sort(begin(W), end(W));
      W.resize(unique(begin(W), end(W)) - begin(W));
      sort(begin(W), end(W), [](const Item& lhs, const Item&
         rhs) {
         return lhs.rank < rhs.rank;
         });
      moveToV(V);
   }
};
```

通过实验我们看到，在最坏情况下（所有元素互不相同），直接对无序向量进行处理，无论采用“删除-擦除”法还是逐个删除法都非常慢，而先排序再复原则快得多。而在最好情况下（所有元素都相同），“删除-擦除”法时间复杂度仅为 $\Theta(n)$

最快，先排序后复原需要 $\Theta(n \log n)$ 较慢，而逐个删除的方法仍然需要 $\Theta(n^2)$ 最慢。

2.8　实验：循环位移

通常我们遍历向量都采用从前向后、一个一个元素遍历的形式。在本章的最后，用**循环位移**（cyclic shift）作为例子，讨论一下非常规的遍历方法。代码可以在 *CyclicShift.cpp* 中找到。

给定向量 $V[0{:}n]$ 和位移量 k，则将原有的向量 $V[0], V[1], \cdots, V[n-1]$，变换为 $V[k], V[k+1], \cdots, V[n-1], V[0], V[1], \cdots, V[k-1]$，称为**循环左移**。相应地，变换为 $V[n-k], V[n-k+1], \cdots, V[n-1], V[0], V[1], \cdots, V[n-k-1]$，称为**循环右移**。循环左移和循环右移的实现方法大同小异，这一小节只讨论循环左移，右移的情况请您自己完成。建议您先自己设计解决这个问题的算法并实现。

循环左移的过程可以表示为 $(V[0{:}k], V[k{:}n]) \to (V[k{:}n], V[0{:}k])$。这个形式非常类似于交换。相信您一定知道最经典的交换函数的实现：

```
template <typename T>
void swap(T& a, T& b) {
   T tmp { move(a) };
   a = move(b);
   b = move(tmp);
}
```

一个最朴素的想法，就是用类似的辅助空间，暂存 $V[0{:}k]$ 中的元素，然后通过 3 次移动来完成循环左移，如图2.11所示。

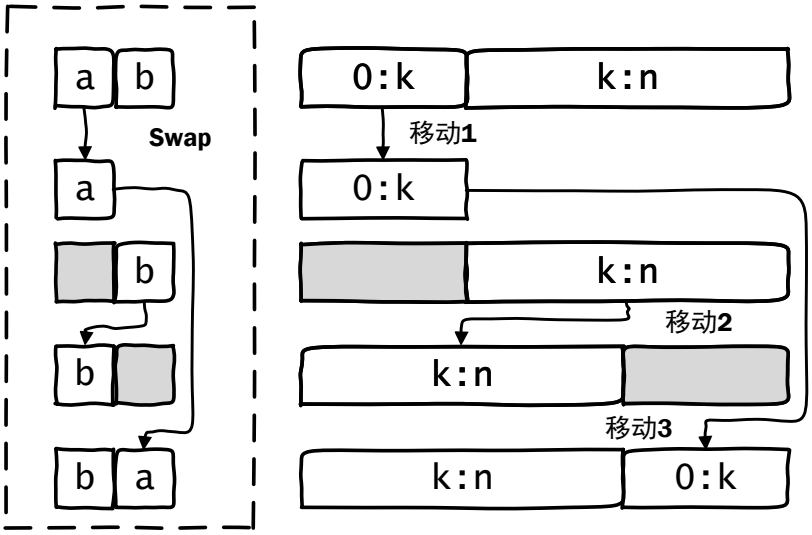

图 2.11　用三次移动实现循环左移，与交换元素的对比

```cpp
// CyclicShiftMove
void operator()(Vector<T>& V, size_t k) override {
   Vector<T> W(k);
   move(begin(V), begin(V) + k, begin(W));
   move(begin(V) + k, end(V), begin(V));
   move(begin(W), end(W), end(V) - k);
}
```

这一算法的时间复杂度是 $\Theta(k)+\Theta(n-k)+\Theta(k)=\Theta(n+k)$。考虑到 $k=O(n)$，也可以简化为 $\Theta(n)$。空间复杂度则为 $\Theta(k)$。

下面的目标则是将空间复杂度降到 $O(1)$。为了保持时间复杂度仍然为 $\Theta(n)$ 不变，需要尽可能一步到位地移动元素。当我们让 $V[i+k]$ 移动到 $V[i]$ 的位置上时，需要暂存 $V[i]$ 到辅助空间去。下一步，如果继续将 $V[i+k+1]$ 移动到 $V[i+1]$ 的位置，那么需要的辅助空间就会增大。为了防止辅助空间增大，则需要考虑“不用暂存”的元素：也就是已经被移动的 $V[i+k]$。下一步将 $V[i+2k]$ 移动到 $V[i+k]$ 的位置，这就是不需要新的辅助空间的。

因此可以得到一个算法：将 $V[i+k]$ 移动到 $V[i]$，再将 $V[i+2k]$ 移动到 $V[i+k]$，以此类推。由于向量中的元素是有限的，您可以证明，存在 j，使得 $(i+jk)\%n=i$，也就是经过 j 次移动后回到了 $V[i]$。最后一次赋值，将辅助空间里的 $V[i]$ 拿出来赋给 $V[i-k]$ 也就是 $V[i+(j-1)k]$ 即可。这样实现了一个“轮转交换”的功能。

需要注意的是，这样一轮并不一定能经过 V 中所有的元素。比如在 $n=6,k=2,i=0$ 时，只轮转交换了 $V[0],V[2],V[4]$ 这 3 个元素，而对另外 3 个元素则没有移动。我们可能需要进行多轮“轮转交换”，各轮共享同一个辅助空间。图2.12展示了 $n=12,k=3$ 的轮转交换例子。

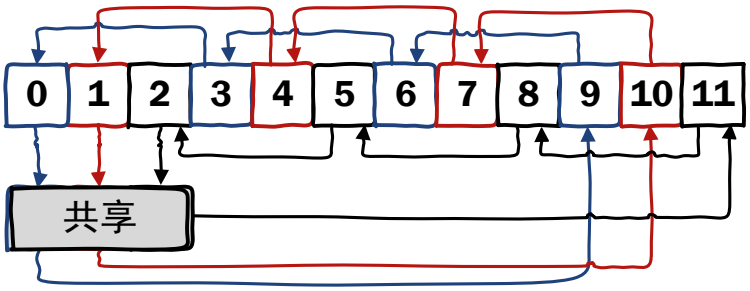

图 2.12　用轮转交换进行循环位移

下面证明：对任意的秩 $0\leqslant r<n$，都存在唯一的 $0\leqslant i<d$ 和 $0\leqslant j<\frac{n}{d}$，使得 $r=(i+jk)\%n$。其中，$d=\gcd(n,k)$ 是 n 和 k 的最大公因数。

证明　因为 r 和数对 (i,j) 的取值范围都是 n 元集，所以只要证明 (i,j) 到 r 是单射，就

蕴含了它同时是满射。因此，只需要证明对于不等的 (i_1, j_1) 和 (i_2, j_2)，$(i_1 + j_1 k)\%n \neq (i_2 + j_2 k)\%n$。

假设存在整数 q，使得 $(i_1 - i_2) + (j_1 - j_2)k + qn = 0$。由于 $(j_1 - j_2)k + qn$ 必定是 d 的倍数，而 $|i_1 - i_2| < d$，所以只能有 $i_1 = i_2$。

设 $k = k_1 d, n = n_1 d$，那么 $(j_1 - j_2)k_1 + qn_1 = 0$。因为 $(k_1, n_1) = 1$，所以 $j_1 - j_2$ 必定是 n_1 的倍数，但 $|j_1 - j_2| < \frac{n}{d} = n_1$，所以只能有 $j_1 = j_2$。这和 (i_1, j_1) 与 (i_2, j_2) 不等矛盾，故由反证法得到单射成立。 ∎

于是，遍历顺序应当是：依次从 $0, 1, \cdots, d-1$ 出发，以 k 为步长遍历 $\frac{n}{d}$ 次回到起点。这个算法的书写有一定挑战性，您可以效仿前面的`swap`去实现轮转交换。

```
// CyclicShiftSwap
void operator()(Vector<T>& V, size_t k) override {
   size_t d = gcd(V.size(), k);
   for (Rank i { 0 }; i < d; ++i) {
      T tmp { move(V[i]) };
      Rank cur { i }, next { (cur + k) % V.size() };
      while (next != i) {
         V[cur] = move(V[next]);
         cur = next;
         next = (cur + k) % V.size();
      }
      V[cur] = move(tmp);
   }
}
```

这一算法的时间复杂度是 $\Theta(d) \cdot \Theta\left(\frac{n}{d}\right) = \Theta(n)$，空间复杂度降低到了 $O(1)$。

在很多教材上会介绍另一种解法：三次**反转**（reverse）。基于反转的原理是，为了实现 $(V[0{:}k], V[k{:}n]) \to (V[k{:}n], V[0{:}k])$，本质上是需要让后半段移动到前半段，前半段移动到后半段。而反转恰好能实现这个功能，且不需要移动算法那样的额外空间。在第一次反转后，$V[0{:}n]$ 变为了 $\overline{V}[n{:}0]$（这里用上划线表示反转），我们可以将它拆分为 $(\overline{V}[n{:}k], \overline{V}[k{:}0])$，然后再将这两段分别反转即可，如图2.13所示。

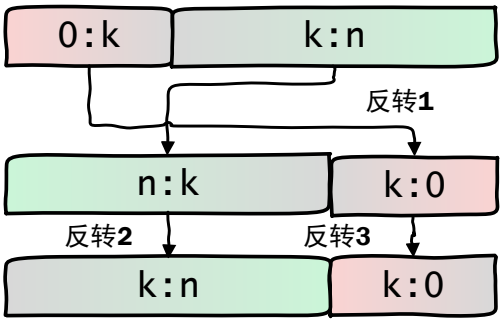

图 2.13　用三次反转进行循环位移

```
// CyclicShiftReverse
```

```
2 void operator()(Vector<T>& V, size_t k) override {
3     reverse(begin(V), end(V));
4     reverse(begin(V), end(V) - k);
5     reverse(end(V) - k, end(V));
6 }
```

三次反转的时间复杂度同样是 $\Theta(n)$, 而空间复杂度是 $O(1)$。下面从理论层面来比较以上三种做法的时间效率。三次移动的做法, 写入内存的次数为 $k+(n-k)+k = n+k$。轮转交换的做法，写入次数为 n（每个元素都一步到位地写入了目标位置）。三次反转的做法，写入次数为 $n+(n-k)+k = 2n$。显然，轮转交换的写入次数最少，三次移动次之，三次反转最多。但实际上，由于三次移动中开辟 $\Theta(k)$ 的空间（并自动做零初始化）本身需要时间，所以它在 k 比较大的时候反而会慢于三次反转。您可以从实验结果中看到这一点。此外，您还能从实验结果中看到，虽然理论上轮转交换的写入次数是 n，但它的时间消耗上下浮动很大（并不像三次移动那样随 k 变大而变大），有些时候时间消耗还会大于三次移动和三次反转。这是算法的局部性导致的，具体机制参见《组成原理》。

附录 A　C++ 的安装和配置

在这篇附录里介绍 Windows 操作系统下 C++ 的安装和配置方法。采用 MacOS 或 Linux 操作系统的读者应当有能力自己完成配置。

A.1　使用 MSVC 编译 C++

Microsoft Visual C++（MSVC）是笔者在编写此书是使用的编译器，也是本书推荐的编译器；本书采用的模块接口文件后缀名（.ixx）就是 MSVC 的标准。得益于强大的 IDE Microsoft Visual Studio，您可以很方便地运行 C++20 编写的程序。

1. 在搜索引擎上输入 Visual Studio，请注意忽略可能在最前方展示位的广告。或者直接访问Microsoft Visual Studio 中文官方网站。
2. 选择下载 Visual Studio，选择 Community 2022（免费的社区版）。如果您是在校学生并且学校购买了专业版，也可以从学校的下载站中获取。请注意不要下载旧版本的 Visual Studio，因为可能不支持 C++20。
3. 安装 Visual Studio，点选“使用 C++ 的桌面开发”，其他可以不选。
4. 创建项目之后，选择项目 → 属性。在“常规”中将“C++ 语言标准”改为 ISO C++20 标准。另外，在 C/C++→ 优化中，将“优化”改为“最大优化（优选速度）（/O2）”。

A.2　使用 GCC 编译 C++

在 Windows 操作系统中使用 GCC，通常需要使用 MinGW。目前 MinGW 上似乎还不支持 format 库，需要对相应的部分进行修改。目前 VS Code 的 C/C++ 插件似乎在 C++20 项目构建会有一些问题，建议采用命令行构建。

1. MinGW 的下载方式有很多，您可以在搜索引擎中找到。笔者使用的 MinGW 来自WinLibs 下载站。
2. 选择最新版本的 MinGW，请注意需要选择 64 位的版本。下载之后解压，需要自行配置环境路径，将 g++.exe 的所在路径配置到 includePath 中。
3. 在控制台（终端）使用`g++ -fmodules-ts -std=c++20 -x c++ -c` 文件名.ixx 来编译模块接口文件（.ixx），它会生成一个模块缓存文件（.gcm）和一个输出文件（.o）；使用同样的命令编译源文件（.cpp）生成输出文件（.o）。
4. 最后使用`g++ a.o b.o main.o -o main.exe`这样的命令进行连接。