DS-Lab 中文教程

Clazy Chen

本书目录

第 1 章 绪论

本书文本及配套代码的开源网站：https://github.com/ClazyChen/ds-lab。

作为一本新的《数据结构》教材和实验指导书，本书的目标是帮助读者掌握数据结构的基本概念和常用算法，以及如何用C++语言实现它们。本书的特点是：

1. 本书详细介绍了各种数据结构和算法的原理和设计思想，帮助读者理解数据结构和算法的本质，而不是仅仅学会使用它们。对于正在准备考试的读者，本书能够有效减少应对闭卷考试所需要的记忆量，是一本很好的复习资料。
2. 本书发扬了工程学科特色，设计了大量的数据结构实验，并为每个实验搭建了用户友好的实验环境。在经典教材中通过理论推导引出的结果，本书总是以实验的方式进行验证。读者可以很方便地根据自己的理解修改实验代码，在训练自己编程能力的同时加深对数据结构和算法的理解。
3. 本书提供了完整的、可运行的代码，读者可以使用 GCC、Clang 或 MSVC（推荐使用 Visual Studio 2022）直接编译代码。对编程不感兴趣、只想学习数据结构和算法的读者，可以直接跳过代码部分。观察、理解代码运行的结果，也足以对数据结构的理论学习起到很好的辅助作用。
4. 本书的所有数据结构和算法都是用 C++20 [1] 实现的，而且是用现代 C++的编程范式实现的。这些范式包括：面向对象编程、泛型编程、函数式编程、元编程等（考虑到 GCC 和 Clang 目前的支持程度，没有采用模块化编程）。这些范式是现代C++的核心特性，也是C++与其他编程语言的重要区别。对编程感兴趣的读者可以从本书中学到现代 C++的编程方法，提高自己的编程能力。

第 1.1 节 对象

本书采用现代C++的编程范式 [1] 。作为一门面向对象编程（Object Oriented Programming，OOP）的编程语言，C++程序总是从类和对象起步。我们从定义 `Object` 类作为本书中所有数据结构和算法的基类开始。

```
// Object.hpp
#pragma once
#include <string>
namespace dslab::framework {
    class Object {
    public:
        virtual std::string type_name() const {
            return "Undefined Object Name";
        }
    };
}
```

【C++学习】

本书中关于C++的部分，以灰色背景的方块展示，以方便读者将这些部分和数据结构知识作区分。本书将默认读者了解C++的基本知识 [2], [3] 。

本书采用和 Java 相似的命名规则。头文件后缀名采用 .hpp，每个公共类的实现放在与其同名的 .hpp 文件中，同一个命名空间的所有类放在同一个目录下，目录的路径为命名空间的名称。比如，上面的文件位于 dslab/framework/Object.hpp，其中 dslab 为母空间，framework 为子空间，Object 为类名。 第 1 章 里的所有头文件都会位于 dslab/framework 目录下，类似地，本章的实验都会位于 lab/framework 目录下。介绍实验时会用蓝色黑体字高亮标注其所在文件，读者可以按照自己的习惯进行实验。 在 Object 类中，我们定义了一个用于输出类名的虚函数。这个函数在后面的章节中会被重载，用于输出各个数据结构和算法的名称。

为了方便引入整个子命名空间内的类，在 dslab 文件夹下，我们加入了一个 framework.hpp 文件，用于导入 dslab::framework 命名空间。该文件的内容如下：

```
// framework.hpp
#pragma once
#include "framework/Object.hpp"
namespace dslab {
    using namespace framework;
}
```

实验 case.cpp。当一个包含了 framework.hpp 的文件需要使用上面定义的类 dslab::framework::Object 时，如果它在 dslab 命名空间下，就可以直接使用 Object 作为类名；如果它在 dslab 命名空间之外，则可以使用 dslab::Object 作为类名，或者 using namespace dslab 之后，使用 Object 作为类名。

第 1.2 节 数据结构

第 1.2.1 节 数据结构的定义

数据结构（data structure）是什么？这是一个很多初学者都不会思考的问题。简单期间，可以把“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 [4] 。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” [5] ——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 开始经常能造成数学上的间接性，避免一些公式出现突兀的“+1”余项 [4] ；但从 1 开始计数更符合自然习惯 [6] 。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。
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 [5] 。
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；但思想保守的学者可能还在用 father 和 son [7] ，进步主义者则可能会用 mother 和 daughter [8] 。
5. 计算机领域的大多数成果来自英文文献。在将英文翻译为中文时，不同译者可能采用不同的译法。如 robustness 有音译的**鲁棒性** [9] 和意译的**健壮性** [10] ，hash 有音译的**哈希** [6] 和意译的**散列** [4] 。
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”“大模型”等概念是这个现象的重灾区。

对于理解概念的专业人员来说，不同的定义方式总是能导出相同的工程实现。对于初学者而言，则需要更加简明、精准、切中要害的定义。因此，试图枚举所有的定义方式，是百害而无一利的教材写法；试图找出一个“正确的”定义，也是百害而无一利的学习方法。本书会将书中的定义和一些经典的教材进行比较 [4], [6] 。对使用其他参考书籍备考的读者，希望您能在理解概念的基础上，自行分析和比较不同教材的定义。

第 1.2.2 节　有限性和互异性

关于数据结构，我们需要认识到：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。在解题时，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，并非数据结构能够脱离具体的计算机而存在。 计算机中的数据和纸上的数据会有很多的不同，不同的计算机所支持的数据结构也有所差异。比如，计算机处理数据

时存在大小不一的高速缓存（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。

本书中提到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，所以讨论整数或浮点数组成的数据结构时，其元素的真正取值范围并不会是数学上的$\mathbb{Z}$或者$\mathbb{R}$。另一方面，数据结构的**规模**，即存放的元素个数，也是有限的。

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，编程者可能希望元素具有多个可能的类型（`std::variant`），甚至希望元素是任意类型的（`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。

【C++学习】

实验 `voa.cpp`。C++标准库提供的一些**容器**模板，实际上是一些数据结构的封装。这些容器模板的第一个模板参数通常就是数据结构中的元素类型，比如 `std::vector<int>`表示元素是 `int` 类型的一个向量（见 第 2 章 ）。如果用户希望表示任意类型的元素组成的向量，可以使用 `std::vector<std::any>`。在访问一个具体元素的时候，需要使用 `std::any_cast<T>`来将其转换成实际的元素类型 `T`。这种方法相比于 C 语言的 `void*`具有类型安全的优点。

但是，`std::vector<std::any>`也可以看成是 `std::any` 类型的元素组成的向量。从向量的角度看，`std::any` 和某个特定的数据类型（比如 `int`）没有不同，甚至不需要做模板**特化**（specialization）。因此在数据结构的视角下，不需要考虑元素具有多个可能类型的情况。

在计算机中不可能存在两个完全*相同*（注意不是相等）的数据，因为至少它们的*地址*不同。所以，数据结构中的元素互不相同，组成了一个**集合**。这种互异性是算法设计和实现中需要注意的。比如，如果将序列(a_1, b, c)修改为了(a_2, b, c)，其中a_1和a_2的值相同、地址不同，看起来好像修改前后这个序列没有什么区别，但实际上可能会引起内存泄漏等严重错误。

第 1.2.3 节　数据结构的实现

回到数据结构的实现中来：继承上一节中实现的 `Object`，设计一个数据结构的基类。作为数据结构这一概念的抽象，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，我们可以使用模板参数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。

【C++学习】

在 C++中，表示规模（size）的类型是 `std::size_t`，它通常是一个无符号整数类型，可能是 `uint16_t`、`uint32_t` 或 `uint64_t`。使用 `std::size_t` 可以语义明确地表示一个变量存储的是大小或长度，有助于提高代码的可读性和可维护性。

类似于熟知的后缀 `f` 表示 `float` 类型的浮点数，从 C++23 开始，可以在整数字面量后面加上 `uz` 来表示 `std::size_t` 类型的整数。比如，`auto a { 42uz };`会自动推导出一个 `std::size_t` 类型的变量 `a`，它的值是 42。

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

读者可能会认为，数据结构作为数据元素的集合，它理应支持增加元素（insert）、删除元素（remove）、查找元素（find）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。

第 1.3 节 算法

第 1.3.1 节 算法的定义和实现

和数据结构经常同时出现的另一个名词是**算法**（algorithm）。算法通常指接受某些**输入**（input），在**有限**（finite）步骤内可以产生**输出**（output）的计算机计算方法 [5] 。

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、输出和有限之外，还包括**可行**（effective）和**确定**（definite） [4] 。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）。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，所以一般不讨论它们。有限性则可以通

过白盒测试和黑盒测试评估。

【C++学习】

众所周知，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（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。

```
template <typename OutputType, typename... InputTypes>
class Algorithm;

template <typename OutputType, typename... InputTypes>
class Algorithm<OutputType(InputTypes...)> : public Object {
public:
    using Output = OutputType;
    template <std::size_t N>
    using Input = std::tuple_element_t<N, std::tuple<InputTypes...>>;
    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，使用可变参数模板（以`...`表示）以处理不同输入的算法。另一方面，借用了`std::function`的表示形式，要求用户使用`OutputType(InputTypes...)`的方式给出模板参数表。比如，一个输入两个整数、输出一个整数的算法可以继承于`Algorithm<int(int, int)>`。`Input<N>`用来表示第`N`个输入的类型，而`Output`用来表示输出类型。

像`Input`和`Output`这样可以反过来获得模板参数的技术，称为**类型萃取**（type traits）。它是模板元编程所使用的重要技术之一。类型萃取可以帮助开发人员在编译器就进行一些决策和判断，从而优化代码或者选择适当的算法。比如，可以通过`if constexpr (std::is_integral_v<A::Input<0>>) { ... }`来判断算法`A`的第一个输入参数是否是整数类型，并执行后面的操作。这个判断在编译期就会被执行，不会产生运行时的开销。

第 1.3.2 节 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。本节将简要介绍算法的几个典型的评价维度。

第一，**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)`、强制跳转（`goto`）和递归的算法是否必定会终止。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（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。

第二，**效率**。评价算法效率的标准可以简单地概括为多、快、好、省 [11]。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。如果需要的空间太多以至于超过了计算机的内存，则外存缓慢的读写也会对算法的时间效率造成重创。

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。

第三，**稳健性**（robustness，又译健壮性、鲁棒性；在看到英文之前，我曾一度认为“鲁棒性”这个词来源于山东小伙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。

第四，**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。对简单算法的正确性检验和复杂度分析，是数据结构的基本功之一。这两个问题留到后面的节里展开讨论。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，在下一小节会用一个实验展示它，后续不再赘述。

第 1.3.3 节 累加问题

算法和实现它的代码（code）或程序（program）有本质区别 [12] 。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。一些人员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，并遗留不计其数的错误或隐患；另一些人员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。参加过语文高考的同学可能会很有感受：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。如果专精算法设计而忽略工程实现，就会有类似的感觉。

上一小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）。在本小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。

实验 sum.cpp。本小节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$1+2+...+n$的和。

```
using Sum = Algorithm<int(int)>;
```

作为实验的一部分，您可以自己实现一个类，继承 Sum，并重载 operator()，和本书提供的示例程序做对比。如果您不熟悉编程，可以先再阅读后文的分析，再自己实现；如果您熟悉编程，可以先自己实现，再阅读后文的分析。当然，如果您不打算自己实现，也可以只阅读本书的理论部分。

在这个实验的示例程序里，设计了几个 Sum 的派生类来完成这个算法功能。最容易想到的办法，自然是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;
}
```

上面的这个算法显然称不上好。著名科学家高斯（Gauss）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（本书默认 `int` 为32位整数）。

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。一些典型值下三种实现的结果如 **表 1.1** 所示。

n		SumBasic	SumAP	SumAP2
10	普通值	√	√	√
0	边界值	√	√	√
50,000	临界值	√	×	√
100,000	溢出值	×	×	×
−10	非法值	√	×	×

表 1.1　求和算法的正确性检验

那么，上述的三个实现，哪些是正确的？

在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，您可以思考一个有趣的问题：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。

【C++学习】

一种可能的方案是，在溢出的时候返回 `std::numeric_limits<int>::max()`。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∞），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，题目要求n是正整数，负数是非法输入；但有时也会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负数输入和饱和的实现，您也可以独立尝试实现这个功能。

第 1.4 节 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**。本节将从数学归纳法的角度出发介绍递降法的原理。

第 1.4.1 节 数学归纳法

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，如 表 1.2 所示。

第一归纳法	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. $\forall n \geq 1, P(n) \to P(n+1)$ 则$\forall n, P(n)$。	令$P(n)$是一个关于正整数n的命题，若满足： 1. $P(1)$ 2. $\forall n > 1, (\forall k < n, P(k)) \to P(n)$ 则$\forall n, P(n)$。

表 1.2 数学归纳法的两种表述

其中，第一归纳法是皮亚诺（Piano）公理体系的一部分，第二归纳法是第一归纳法的直接推论。另一方面，第二归纳法的**归纳假设**$(\forall k \leq n, P(k))$，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。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“非负整数”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**：

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$A\left(k_j\right)$，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。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。

第 1.4.2 节　良序关系

为了将经典归纳法推广到更广的范围，需要提炼出$\mathbb{N}$使归纳法成立的性质：**良序性**（well-ordered）。如果集合上的一个关系$\preccurlyeq$满足：

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

那么称$\preccurlyeq$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“$\leq$”和小于“$<$”关系，对于关系$\preccurlyeq$，如果$x \preccurlyeq y$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“$\leq$”在$\mathbb{N}$上是良序的，而在$\mathbb{Z}$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$\mathbb{Z}$成为良序集（因此，$\mathbb{Z}$上的命题可以通过对绝对值归纳证明）。同时，熟知的小于等于关系“$\leq$”在正实数集$\mathbb{R}^+$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）。

和正整数集$\mathbb{N}$相比，一般的良序集具有下面的相似性质（**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$\forall j$成立。

您可以用反证法证明上述命题。据此，可以得到一般形式的递降法。

令S是一个良序集，如果S上的算法$A(n)$满足：

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

则$\forall x \in S, A(x)$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。在上述递降法的表述中，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，只是侧重点不同。在设计算法时，$A(x)$是一个待解决的算法问题，因此尝试将它拆解为有限个规模较小的子问题$A(y_j)$，递归地解决这些子问题，直到到达平凡情况（$\min S$）。而在分析算法时，首先证明平凡情况下算法是有限、正确的，然后再证明非平凡情况下，将$A(x)$化归为有限个$A(y_j)$的过程是有限、正确的。由于二者的相似性，后文将不再区分。

第 1.4.3 节　最大公因数

实验 **gcd.cpp**。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。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);
    }
}
```

【C++学习】

在《数据结构》中，通常会将上面这样的代码当做递归算法来分析。但在实际的C++编译器中，上面的代码会被自动优化为非递归形式，所以实际上不需要刻意去做这样的转换。这是C++编译器的一种**尾递归优化**（tail recursion optimization）。一些其他语言（比如Python）没有这样的优化，因此在这些语言中，递归算法的效率可能会比较低。

在这个算法中，递归边界是$(a,0)$，因此可以定义$f(a,b)=\min(a,b)$，将输入数据映射到熟知的良序集$\mathbb{N}$。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$a\geq b>0$，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。

第 1.4.4 节　数组求和

递降法的条件 2 允许在一个递归实例中，调用自身有限次。在上一节中，一个gcd只会调用一次自身；这一小节展示了一个多次调用自身的例子。

实验 asum.cpp。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$A_0+A_1+\ldots+A_{n-1}$的和。

```
using ArraySum = Algorithm<int(std::span<const int>)>;
```

【C++学习】

std::span 是 C++标准库提供的模板类，是对连续内存区域的一种视图抽象，可以安全、高效的操作连续的内存区域。它可以用来表示 std::array、std::vector、C 风格数组等连续内存区域的一个切片（slice）。这个切片可以使用迭代器、下标运算符等方法，像一个独立数组一样使用，而不需要实际将切片中的内存复制到一个新的独立数组。

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。

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

在 C++标准库中还存在另外两个函数可以用来求和，分别在 C++17 和 C++23 引入。这三种函数都可以用于归约（如使用乘、异或等代替求和中的加），不同编译器对于这三种函数的优化程度不同，其中一些编译器有可能应用了 SIMD 技术来提高执行效率，这大约会带来数倍的时间差距。

```
std::reduce(data.begin(), data.end(), 0); // C++17
std::ranges::fold_left(data, 0, std::plus{}); // C++23
```

下面回到递归话题来。一个基于经典递降法的朴素思路是：要求n个元素的和，可以先求前$n-1$个元素的和，然后将它和最后一个元素相加。这是一个递归算法，如下所示。

```
// ArraySumRnC
int operator()(std::span<const int> data) {
    if (data.size() == 0) {
        return 0;
    }
    return (*this)(data.first(data.size() - 1)) + data.back();
}
```

这种将待解决问题分拆为一个规模减小的问题+有限个平凡的问题的思想，被称为**减治**（reduce-and-conquer，或 decrease-and-conquer）[4]。在《数据结构》中使用减治思想，通常是将数据结构（这里是数组）分拆成几个部分，其中只有一个部分的规模和原数据结构的规模相关（减治项），其他部分的规模都是有界的（平凡项）。在上面的例子中，数组被分拆成了两个部分，前$n-1$个元素组成的部分是减治项，而最后一个元素是平凡项。

循环可以被看成是减治的一种特殊形式。我们可以认为循环的第一次执行是平凡项（起始处减治），此后的执行是减治项；或者，我们也可以认为循环的最后一次执行是平凡项（结尾处减治），此前的执行是减治项。在下面的循环代码的结尾处减治，就对应了刚刚介绍的减治算法。

```
// ArraySumIterative
int operator()(std::span<const int> data) override {
    int sum { 0 };
    for (int x : data) {
        sum += x;
    }
    return sum;
}
```

下面介绍另一种不同的思路。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）。这也是一个递归算法，如下所示。

```
// ArraySumDnC
int operator()(std::span<const int> data) override {
    if (auto sz { data.size() }; sz == 0) {
        return 0;
    } else if (sz == 1) {
        return data.front();
    } else {
        auto mid { sz / 2 };
```

```
        return (*this)(data.first(mid)) + (*this)(data.subspan(mid));
    }
}
```

上述算法设计中，我们将待解决问题拆分为多个规模减小的问题，这种思想称为**分治**（divide-and-conquer）。在《数据结构》中使用分治思想，通常是将数据结构（这里是数组）分拆为几个部分，每个部分的规模都和原数据的规模相关（分治项）。广义的分治也可以包含若干个平凡项。在上面的例子中，数组被分拆为了两个部分，每个部分的规模大约是原规模的一半。减治和分治是设计算法的重要思路，在本书中将广泛使用。这两种思想也能为您解决自己遇到的算法问题提供强大的助力。

n	迭代	accumulate	reduce	fold_left	减治递归	分治递归
10^4	0	0	0	0	0	0
10^6	0	0	0	0	栈溢出	0
10^7	3	3	3	3	栈溢出	12
10^8	33	31	28	31	栈溢出	169
10^9	315	316	267	308	栈溢出	2093

表 1.3　数组求和算法的时间

本书中的表格和图像都是通过实验得到的（GCC 13.2.0；单位为毫秒），其他编译器可能会使得实验结果和 表 1.3 有所不同，尤其是使用标准库函数的三列。请读者在自己的环境中进行实验，以得到更加准确的结果，并和本书中的结果对比分析。

分治算法和普通的迭代算法相比，只是修改了加法的运算次序；但测试结果显示，它竟然远远不如普通的迭代算法。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，将介绍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。

第 1.4.5 节　函数零点

实验 `zerop.cpp`。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。

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$l=-1,r=1,\varepsilon=10^{-6}$。

```
class ZeroPoint : public Algorithm<double(std::function<double(double)>)>
{
    static constexpr double limit_l { -1.0 };
    static constexpr double limit_r { 1.0 };
protected:
    static constexpr double eps { 1e-6 };
    using funcdd = std::function<double(double)>;
    virtual double apply(funcdd f, double l, double r) = 0;
public:
    double operator()(funcdd f) override {
        return apply(f, limit_l, limit_r);
    }
};
```

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**（bisect）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(funcdd 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;
}
```

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：$r-l<\varepsilon$。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$f(l,r)=\left\lfloor\frac{r-l}{\varepsilon}\right\rfloor$就可以将问题映射到$\mathbb{N}$，从而使用递降法进行正确性检验。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(funcdd f, double l, double r) override {
    if (r - l <= eps) {
        return l;
    } else {
        double mid { l + (r - l) / 2 };
        if (f(l) * f(mid) <= 0) {
            return apply(f, l, mid);
        } else {
            return apply(f, mid, r);
        }
    }
}
```

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，实际遇到问题不需要显式地将其改写为递归，只要在分析算法的正确性时，将循环视为递归即可。

第 1.5 节 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。

第 1.5.1 节 复杂度记号的定义

假设问题规模为n时，算法在某一计算机上执行的绝对时间单元数为$T(n)$。

1. 对充分大的n，如果$T(n) \le C \cdot f(n)$，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$C_1 \cdot f(n) \le T(n) \le C_2 \cdot f(n)$，其中$C_2 \ge C_1 > 0$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$C \cdot f(n) \le T(n)$，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**。类似可以定义空间复杂度。

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。

在“充分大的n”和“忽略常数”两个前提下，复杂度记号里的函数往往特别简单。比如，多项式$\sum_{i=0}^{k} a_i n^i$（$a_k > 0$）可以被记作$\Theta\left(n^k\right)$，因为充分大的n下，次数较低的项都可以被省略，忽略常数又使我们可以省略系数a_k。

一些教材为简略起见，只介绍了$O(\cdot)$一个符号，这非常容易引起理解错误或混淆 [6] 。在下一小节中会展示很多错误理解的例子。在本书中这三种复杂度记号都会使用。在只使用$O(\cdot)$的文献里，$O(\cdot)$常常用来实际上表达$\Theta(\cdot)$；而在严谨的文献里，除了$O(1)$和$\Theta(1)$没有区别外，其他情况下这两者都是被严格区分的。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。

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*。比如通常情况下，能正常工作的内存绝不可能需要多于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。

第 1.5.2 节 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 [5]。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。

第一，**不可交换**。设$n^3 = O(n^4)$，$n^2 = O(n^4)$，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。**（最基本的性质）
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。

第二，**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，例如$T(n) = n\left(\sin\frac{n\pi}{2} + 1\right)$，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$\Theta(\cdot)$。

第三，**只有$\Theta(\cdot)$才可以进行比较**。已知算法 A 的复杂度是$O(n)$，算法 B 的复杂度是$O(n^2)$，那么，算法 A 的复杂度是否一定低于算法 B？

这是最容易误解的一处，我们可以说$O(n)$的复杂度低于$O(n^2)$，但切不能想当然认为算法 A 的复杂度低于算法 B。这是因为，尽管已知条件告诉我们“算法 B 的复杂度是$O(n^2)$”，但已知条件并没有排除“算法 B 的复杂度同时也是$O(n)$甚至$O(1)$”的可能。类似地，$\Omega(\cdot)$也不可比较，只有表示同阶无穷大量的$\Theta(\cdot)$有比较的意义。考试中也可能遇到需要比较复杂度的情况，这是纯粹的数学问题，因此不做展开。

计算机工程师对数学严谨性并不敏感。在相当大规模的人群中，对$O(\cdot)$形式的复杂度进行比较已经成为了一种习惯。作为一门工程学科，当一种做法被人们普遍接受、成为共识的时候，通常就被认为是合理的，以至于人们忽视了，只有当$O(\cdot)$事实上在表达$\Theta(n)$的语义时，这样的比较才是成立的。如上一点所说，并非所有复杂度分析都能用$\Theta(\cdot)$形式的结果，只有了解了严谨的说法，才能避免在阅读文献时理解上出现混淆。

第四，**不满足对减法、除法的分配律**。不论是三种复杂度记号中的哪一种，都不服从对减法、除法的分配律。对加法和乘法，分配律是单向成立的（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，公式$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$是成立的，反过来则不成立。

第五，**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）”。

情况（case）表示和问题规模n无关的输入数据特征。比如说，我们想要在规模为n的整数数组A中，寻找第一个偶数（找不到时返回 0），可以使用如下算法。

```
int operator()(std::span<const int> data) override {
    for (auto a : data) {
        if (a % 2 == 0) {
            return a;
        }
    }
    return 0;
}
```

容易发现，除了问题规模n之外，这n个整数自身的特征也会影响找到第一个偶数的时间。如果A_0就是偶数，则只进行了一次奇偶判断，可以认为$T(n) = 1$；如果A中每一个元素都是奇数，则需要对每个元素都进行奇偶判断，可以认为$T(n) = n$。对于同样的n，不同情况的输入数据会得到不同的$T(n)$。

如果情况会对算法的性能造成影响，$T(n)$就不再是一个准确的值，变成了一个范围，它的下限对应了最好情况，上限对应了最坏情况。如果设$T(n)$在最好情况下为$g(n)$，最坏情况下为$h(n)$，那么对$g(n)$和$h(n)$可以分别做复杂度分析，得到它在最好情况和最坏情况下的复杂度。在这个复杂度分析的过程中，三种符号都是可以使用的。 也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。

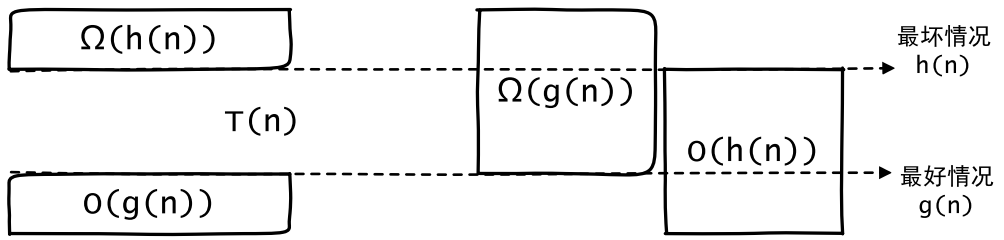

图 1.1　最好情况和最坏情况的评估

不同的符号表达的侧重点是不同的。如 图 1.1 所示，当我们使用$O(\cdot)$描述最好情况时，可以体现出“最好”究竟有多好，而$\Omega(\cdot)$描述最好情况则做不到这一点。相应地，$\Omega(\cdot)$才可以体现出“最坏”有多坏。因此，在无法用准确的$\Theta(n)$表达时，我们通常使用$O(\cdot)$描述最好，用$\Omega(\cdot)$描述最坏。

第 1.5.3 节 判断 2 的幂次

实验 **power2.cpp**。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。本节讨论判断一个正整数是否为 2 的幂次的算法。

```
using IsPower2 = 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 & (n - 1)) == 0;
}
```

【C++学习】

标准库提供了一个专门的函数 `std::has_single_bit`，用于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）。`<bit>` 中还包括很多其他常用的位操作函数，编译器通常会将这些函数优化为目标平台上的对应汇编指令。

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

上述两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$f(n)=\max\{d \mid n\%2^d=0\}$为满足$2^d$整除$n$的最大的$d$，使其映射到$\mathbb{N}$上的良序关系。当$n$为奇数的时候，$f(n)=0$到达边界值。但对这种简单的问题，也可以直接显式计算$T(n)$，即函数体的执行次数。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

$$T(n) = T\left(k \cdot 2^d\right) = 1 + T\left(k \cdot 2^{d-1}\right) = 2 + T\left(k \cdot 2^{d-2}\right)$$
$$= \ldots = d + T(k) = d + 1 = O(d) = O\left(\log\left(\frac{n}{k}\right)\right)$$
$$= O(\log n)$$

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。

同样，可以显式地计算下面这个算法的$T(n)$。

```
// IsPower2Iterative
bool operator()(int n) override {
    int m { 1 };
    while (m < n) {
        m *= 2;
    }
    return m == n;
}
```

上面的两个实现，并不是同一算法的迭代形式和递归形式。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“递增”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，它使得两种算法的时间复杂度有所不同。递归算法的时间复杂度为$O(\log n)$和$\Omega(1)$，而迭代算法为$\Theta(\log n)$。

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。

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。

第 1.5.4 节 快速幂

在上一小节，如果认为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；从而不能说最好$O(1)$和最坏$\Omega(\log n)$。但如果认为n的位宽$\log n$是“问题规模”，则可以这么说。这一现象反映了用位宽，也就是“输入规模”来表示问题规模的好处。经典教材 [4] 也建议这样表示问题规模，并使用了快速幂作为举例阐述这个问题。

实验 **power.cpp**。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。

```
using PowerProblem = Algorithm<int(int, int)>;
```

基本的求幂算法，和求和类似：

```
// PowerBasic
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$a^b = (a(a(a(a(a...(a \cdot a)...)))))$，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，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。这样，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$a^b = \left(a^2\right)^{\frac{b}{2}}$），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。

那么，这个算法的问题规模是什么？

最自然的想法是，它的问题规模是b。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（注意，此时不能认为两数相乘是$O(1)$的，因此复杂度的形式会有所不同）。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“输入规模派”可以做到这一点。

在大多数情况下，这两种学说并无分歧。大部分教材支持“输入规模派”[4]，而较早的书上没有讨论这个问题[6]，通常这个问题对解题也没有任何影响。

第 1.5.5 节　复杂度的积分计算

实验 `integral.cpp`。递归算法的复杂度分析在之后还会讨论更多技术。一类比较简单的复杂度问题是每层循环的迭代次数都很直观的多重循环问题。比如前面的 IsPower2Iterative，可以一眼就看出来迭代的次数是$\Theta(\log n)$。这种问题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。

```
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：

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

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：

$$\begin{aligned}T(n)&=\sum_{i=1}^{n}\sum_{j=1}^{i}j\cdot(1+\lfloor\log j\rfloor)=O\left(\int_0^n\int_0^x y(1+\log y)dydx\right)\\&=O\left(\int_0^n\int_0^x y\log ydydx\right)=O\left(\int_0^n x^2\log xdx\right)=O(n^3\log n)\end{aligned}$$

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。可以看出对于l的分析来说，可以直接使用更简单的$\log j$代替实际执行次数$1+\lfloor\log j\rfloor$。如 图 1.2 所示，该函数的实际执行时间确实和$n^3\log n$近似成线性关系。

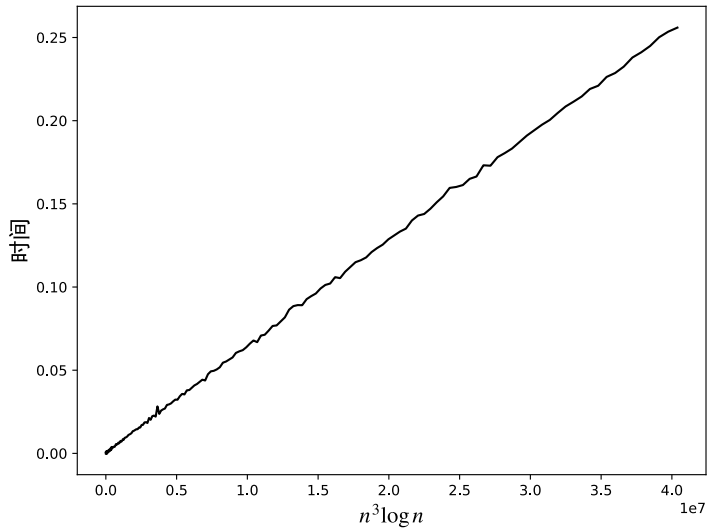

图 1.2 复杂度验证结果

如果想要估算常数，只需要：

$$T(n)=\sum_{i=1}^{n}\sum_{j=1}^{i}j\cdot(1+\lfloor\log j\rfloor)\sim\int_0^n\int_0^x y(1+\log y)dydx$$
$$\sim\int_0^n\int_0^x y\log ydydx\sim\int_0^n\frac{1}{2}x^2\log xdx\sim\frac{1}{6}n^3\log n$$

这里最后的$\frac{1}{6}$就是常系数。

第 1.5.6 节　多重循环复杂度的简单估算

上面这种积分的方法是计算此类循环算法的时间复杂度及其常数的一般方法。在熟练之后，可以用一些小技巧来处理。下面介绍一些小技巧；当然，如果时间充足，还是建议用积分方法去严格地进行计算。

在这类循环算法中，每一层循环主要有下面这几种典型的形式：

```
for (int i { 1 }; i <= n; ++i)    { /* (1) */ }
for (int i { 1 }; i <= n; i *= 2) { /* (2) */ }
for (int i { 2 }; i <= n; i *= i) { /* (3) */ }
```

1. 线性增长的循环，无论它出现在什么位置，都会为复杂度增加一个线性项n。
2. 指数增长的循环，如果它出现在最内层（或可以被交换到最内层，下同），那么会为复杂度增加一个对数项$\log n$；如果它不出现在最内层，通常不会影响复杂度。
3. 幂塔增长的循环，如果它出现在最内层，那么会为复杂度增加一个迭代对数项$\log^* n$（关于迭代对数，您不需要了解更多）；如果它不出现在最内层，通常不会影响复杂度。

当（2）（3）出现在非最内层时，通常不会影响复杂度。这一点看起来没有那么显然，甚至很容易被忘记。下面用一个典型的例子来说明上面的（2）。（3）的情况是类似的。

```
int f(int n) {
    int result = 0;
    for (int i { 1 }; i <= n; i *= 2) {
        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;
}
```

上面的这个程序，和上一小节展示的例子相比，只有最外层i的循环，从线性递增改成了指数递增。以下通过积分方法解决此问题的方法。

$$T(n)=\sum_{i=1}^{\lfloor\log n\rfloor}\sum_{j=1}^{2^i}j\cdot(1+\lfloor\log j\rfloor)=O\left(\int_0^{\log n}\int_0^{2^x}y(1+\log y)dydx\right)$$
$$=O\left(\int_0^{\log n}\int_0^{2^x}y\log ydydx\right)=O\left(\int_0^{\log n}4^x\cdot xdx\right)$$
$$=O\left(\int_0^{n}t^2\cdot\log t\cdot\frac{dt}{t}\right)=O(n^2\log n)$$

可以发现，在最后一步进行换元$t=2^x$时，由于$dx=\frac{dt}{t\ln 2}$会引入一个分母（一次项），该分母和积分引入的、同样是一次的分子会抵消，抵消的结果就是这一层积分并不会升幂。换句话说，在上面这个程序中，指数递增的外层循环i是否存在，对时间复杂度没有影响。

第 1.6 节 本章习题

本书的习题按照小节排列，以黑体标注题目难度。简单 表示基础知识，中等 表示一般知识，较难 表示需要思考，挑战 是有一定难度的拓展题。

在 第 1.3.3 节 中：

1. 简单 SumBasic 的时间复杂度是多少？
2. 简单 SumAP 和 SumAP2 各自在n处于什么区间时可以输出正确的值？
3. 中等 如果输入输出范围不是 32 位的 `int`，而是 `W` 位的带符号整型，则这两种实现各自在n处于什么区间时可以输出正确的值？并分析 `W` $\to\infty$时的情况。

在 第 1.4.3 节 中：

1. 简单 分析朴素的最大公因数算法 GcdBasic（见 `gcd.cpp`）和欧几里得算法的时间复杂度和空间复杂度。
2. 中等 使用减治法推广欧几里得算法，使其能够计算多个数的最大公因数。
3. 简单 假设m位整数的乘法和带余数除法时间复杂度为$O(m\log m)$ [13], [14] ，分析在输入数据的位宽为m时，朴素的最大公因数算法和欧几里得算法的最坏时间复杂度。
4. 中等 对中国古典名著《九章算术》里介绍的中华更相减损术 [4] GcdCN（见 `gcd.cpp`）做正确性检验，并分析其在 `int` 和m位整数两种情形下的最坏时间复杂度。请注意除以 2 的操作在计算机中只需要移位就可以完成，因此其时间复杂度和移位一致，是$O(m)$而非$O(m\log m)$。
5. 中等 将中华更相减损术转换为递归算法，并分析其空间复杂度。

6. 较难 不考虑最坏情况而讨论一种特殊的情形。狄利克雷（Dirichlet）定理表明，如果 a，b 是随机选择的整数，则 a，b 的最大公因数是 1 的概率是 $\frac{6}{\pi^2}$，因此第 2 题设计的算法会在几次头部减治之后，退化为$a \gg b$的情况。在此情况下评估欧几里得算法和中华更相减损术算法的时间性能。
7. 较难 裴蜀（Bézout）定理指出对于任意两个整数 a，b，存在整数 x，y 使得$ax + by = \gcd(a, b)$。满足条件的x, y可以通过如下的扩展欧几里得算法求得。证明该算法的正确性。

```
std::pair<int, int> operator()(int a, int b) override {
    int x { 0 }, y { 1 }, u { 1 }, v { 0 };
    while (a != 0) {
        int q { b / a }, r { b % a };
        int m { x - u * q }, n { y - v * q };
        b = a; a = r; x = u; y = v; u = m; v = n;
    }
    return { b, x };
}
```

8. 挑战 将上述扩展欧几里得算法推广到欧几里得环$\mathbb{Z}\left[\sqrt{2}\right]$上 [15] 。

在 第 1.4.4 节 中：

1. 简单 对迭代算法 ArraySumIterative 做尾部减治，就对应了 asum.cpp 里展示的减治算法；请实现头部减治所对应的算法。
2. 简单 证明 asum.cpp 中的减治算法和分治算法的正确性，并分析它们的时间复杂度和空间复杂度。
3. 中等 在 asum.cpp 中的分治算法，将数组等分成了两个部分，然后递归地求解这两个部分的和。如果将数组等分成k个部分（$k \geq 2$是一个常数），然后递归地求解这k个部分的和，会得到什么样的算法？分析它的时间复杂度和空间复杂度。
4. 简单 采用本节介绍的减治和分治方法，分别设计求一个数组最大值的算法，并分析它们的时间复杂度和空间复杂度。

在 第 1.4.5 节 中：

1. 中等 本节所述的二分算法属于分治还是减治？为什么？
2. 中等 在二分算法的示例程序中，如果把$f(l) \cdot f(\text{mid}) \leq 0$中的小于等于改为严格小于，会造成什么结果？
3. 较难 实验 mid.cpp。在二分算法的示例程序中将 mid 赋值为了$l + \frac{r-l}{2}$。其他常见的写法还有$\frac{l+r}{2}$和$\frac{l}{2} + \frac{r}{2}$。数学上这两者是相同的，但由于计算机中的数据有位宽的限制，会导致这三种写法的实际效果有所不同。请在整型和浮点型两种场合下，分析这三种写法的优劣。

在 第 1.5.1 节 中：

1. 简单 证明$\Theta(\log_a n) = \Theta(\log_b n)$对一切$a, b > 1$成立。正是因为这个原因，在计算机领域通常省略底数，直接写作$\Theta(\log n)$。
2. 简单 证明$\log n = O(n^c)$对一切$c > 0$成立。
3. 简单 证明$n^c = O(a^n)$对一切$c > 0, a > 1$成立。
4. 简单 在现代通用个人计算机上，一秒大约可以完成10^9数量级的原子操作。假设关于该原子操作的常数为 1，那么一个$\Theta(n)$的算法只有在$n \leq 10^9$的条件下才能在一秒内完成。类似地，请讨论$\Theta(\log n), \Theta(n \log n), \Theta(n^2), \Theta(n^3), \Theta(2^n)$以及$\Theta(n!)$的算法分别在多大的$n$下才能在一秒内完成。
5. 简单 如果$T(n)$是等差数列，那么它的时间复杂度是什么？如果$T(n)$是等比数列，那么它的时间复杂度是什么？

在 第 1.5.2 节 中：

1. 中等 在$O(\log(n!)), \Theta(\log(n!)), \Omega(\log(n!)), O(n \log n), \Theta(n \log n), \Omega(n \log n)$之间，写出所有可以用等号连接的符号对。
2. 中等 正文中寻找第一个偶数的算法是从前向后遍历数组的，如果从后向前遍历数组，会得到什么样的结果？如何才能刻画这两种算法的效率差异？

在 第 1.5.3 节 中：

1. 简单 证明 IsPower2Iterative 的时间复杂度是$\Theta(\log n)$，空间复杂度是$O(1)$；而 IsPower2Recursive 的时间复杂度和空间复杂度均为$O(\log n)$和$\Omega(1)$。
2. 中等 将 IsPower2Iterative 转换为与其等效的递归算法，将 IsPower2Recursive 转换为与其等效的迭代算法。
3. 简单 `int` 溢出问题是否会对本节介绍的几种算法的稳健性产生影响？如果会，如何改正它？
4. 中等 如果按照“输入规模派”的观点，在判断 2 的幂次的问题中，问题规模应该如何定义？并在这个定义下，分析 IsPower2Basic、IsPower2Recursive 和 IsPower2Iterative 的时间复杂度和空间复杂度。
5. 较难 判断 2 的幂次等价于判断一个正整数的二进制表示是否只有一个 1。现在我们希望输出一个正整数的二进制表示中 1 的数量，在不借用`<bit>`库的情况下，分别利用迭代和递归设计相应的解决方案，并分析它们的时间复杂度和空间复杂度（问题规模采用输入规模定义）。
6. 较难 如果希望算法输出的是输入的正整数“按位颠倒”的结果（包括前导 0 但不包括符号位，如原数据为 00000001 11011001 10011111 11000001；则按位颠倒之后的结果为 01000001 11111100 11001101 11000000），又该如何设计算法？

在 第 1.5.4 节 中：

1. 中等 根据正文中的递推式，将 PowerBasic 转换为对应的减治递归形式。从这个快速幂的例子中也可以看出分治算法相对减治的好处。
2. 中等 将 PowerFastRecursive 转换为对应的迭代形式。
3. 中等 借助$a^b = \left(a^{\frac{b}{2}}\right)^2$，写出另一种快速幂的实现。

在 第 1.5.5 节 和 第 1.5.6 节 中：

简单 分析以下程序的时间复杂度（不考虑编译器优化）。定义最内层循环每次执行的时间是 1 个单位的情况下，估算这些时间复杂度的常数。

1. ```
   void F1(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < n; ++j);
   }
   ```
2. ```
   void F2(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < i; ++j);
   }
   ```
3. ```
   void F3(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 1 }; j < n; j *= 2);
   }
   ```
4. ```
   void F4(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < n; j += i);
   }
   ```
5. ```
   void F5(int n) {
       for (int i { 0 }, j { 0 }; i < n; i += j, j += 2);
   }
   ```
6. ```
   void F6(int n) {
       for (int i { 0 }, j { 1 }; i < n; i += j, j *= 2);
   }
   ```
7. ```
   void F7(int n) {
       for (int i { 0 }, j { 1 }; i < n; i += j, j *= j);
   }
   ```
8. ```
   void F8(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < n; ++j)
       for (int k { 0 }; k < n; k += j);
   }
   ```

```
9. void F9(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < n; ++j)
       for (int k { 2 }; k < j; k *= k);
   }
10. void F10(int n) {
       for (int i { 0 }; i < n; ++i)
       for (int j { 0 }; j < n; j += i)
       for (int k { 1 }; k < j; k *= 2);
   }
```

第 1.7 节 本章小结

本书的小结部分是不全面的，我只会介绍每一章的核心要点。如果您认为有必要，应当在自己的笔记上进行更加全面、更加适合您本人学习路径的总结。

本章的核心内容是**递归**的思想方法。

1. 您可以从超限归纳法和递降法的角度，理解递归思想的数学背景。
2. 您可以利用递归的思路分析迭代算法，将迭代终止条件和递归边界联系起来，认识到递归算法和迭代算法的等效性。
3. 您了解了利用递归的思想进行正确性检验和复杂度分析的技术。
4. 您了解了减治和分治这两种经典的递归设计方法。

第 2 章 向量

线性表是指相同类型的有限个数据组成的序列。本书按照 [4] 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 [6] 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**。本章介绍顺序结构的线性表，即向量。

第 2.1 节 线性表

对比线性表和一般的数据结构。一般的数据结构的定义中，数据被组织为“某种结构化的形式”，而在线性表这里它被具体化为了“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；每个元素有它在序列上的“上一个”和“下一个”元素。这是一个朴素的想法，从数学角度容易理解。但从计算机的角度，请您思考这样的问题：“元素”应该用什么表示？应该如何找到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（称为秩 [4] ，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。

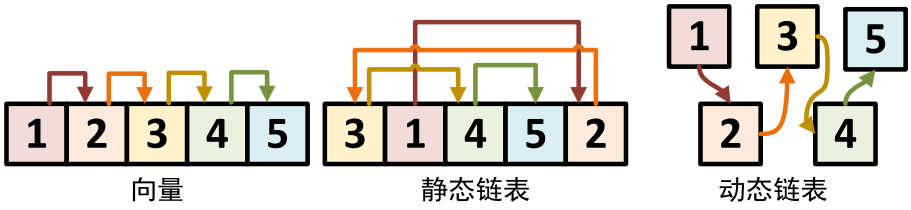

图 2.1　向量、静态链表和动态链表的对比

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。从这个例子中可以看到，静态链表是一个不实用的数据结构，本书将把重点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。我们通过传入不同的迭代器类型参数来控制这两种访问方式。

```
template <typename T, typename It, typename CIt>
class LinearList : public DataStructure<T> {
public:
    using value_type = T;
    using iterator = It;
    using const_iterator = CIt;

    virtual iterator insert(const_iterator p, const T& e) = 0;
    virtual iterator insert(const_iterator p, T&& e) = 0;
    virtual iterator erase(const_iterator p) = 0;
    virtual const_iterator find(const T& e) const = 0;
    virtual void clear() = 0;
};
```

【C++学习】

完整的一个 STL 容器（哪怕是 `std::array`）是非常复杂的，感兴趣的读者可自行阅读其源码。本书的示例代码是一个简化的版本。

右值引用（rvalue reference）被用于实现移动语义，即将一个对象的资源（比如内存）袋子*移动*到另一个对象的袋子里，而不是*复制*同样的一份资源。这样可以避免不必要的内存分配和释放，提高程序的性能。容易复制的类型，如基本类型和简单结构体，通常不需要使用右值引用。

传统的左值引用通常使用 `T&` 表示，而右值引用则使用 `T&&` 表示。以上面示例代码中的两个版本 `insert` 为例。假设 `L` 是一个元素类型为 `T` 的线性表对象：

```
T a {};
L.insert(L.end(), a); // insert(const_iterator, const T&)
L.insert(L.end(), std::move(a)); // insert(const_iterator, T&&)
L.insert(L.end(), T{}); // insert(const_iterator, T&&)
```

当 `e` 传入一个对象 `a` 时，`insert` 会调用左值引用的版本，因为在 `a` 被插入的线性表之后，我们希望线性表中的元素 `L.back()` 和原有的元素 `a` 是两个不同的元素，也就是说我们希望在 `a` 所使用的资源之外，再增加一份资源用于 `L` 中新加入的元素。而当 `e` 传入被移动的对象 `std::move(a)` 时，后续我们不会再通过 `a` 访问这个元素，因此我们可以把 `a` 的资源直接移动到 `L` 中，而不需要再分配一份新的资源；所以这个时候会调用右值引用的版本。最后，当 `e` 传入一个临时对象 `T{}` 时，`T{}` 是一个临时对象，它的资源不会被其他对象使用，因此我们可以直接移动它的资源到 `L` 中，也调用右值引用的版本。

迭代器（iterator）是 STL 广泛使用的一种设计模式。迭代器是一个对象，它可以指向容器中的一个元素，也可以通过一些操作来访问容器中的元素，类似于一个带封装的指针。迭代器的设计使得 STL 的算法可以独立于容器的具体实现。由于迭代器和《数据结构》中的知识点无关，本书中的数据结构会使用一个简化版本的迭代器，以使得一些 STL 算法以及 `range-based for` 可以被用于本书中的数据结构。在上面的示例代码中，`const_iterator` 是一个只读的迭代器，即被它指向的容器元素不能被修改，而 `iterator` 是一个可读可写的迭代器。使用迭代器，可以通过如下方式遍历一个 `std::vector` 对象 `V`（以下三种方式是等价的）：

```
for (auto i { 0uz }; i < L.size(); ++i) { visit(V[i]); } // visit by rank
for (auto i { V.begin() }; i != V.end(); ++i) { visit(*i); } // visit by
iterator
for (auto& e : V) { visit(e); } // visit by range-based for
```

第 2.2 节 向量的结构

向量（vector）是一个基于**数组**（array）的数据结构。和数组一样，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容器 `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`）。向量的时间和空间性能和数组相比都只有常数的差异，不会有复杂度的区别，且在大多数场景下二者的性能差异并不显著。因此，除非数组的大小具有确定的语义，否则总是可以使用 `std::vector` 代替数组。

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]`。众所周知，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。 第 2.1 节 中的旅行团例子也可以帮助理解规模和容量的关系：一个n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。

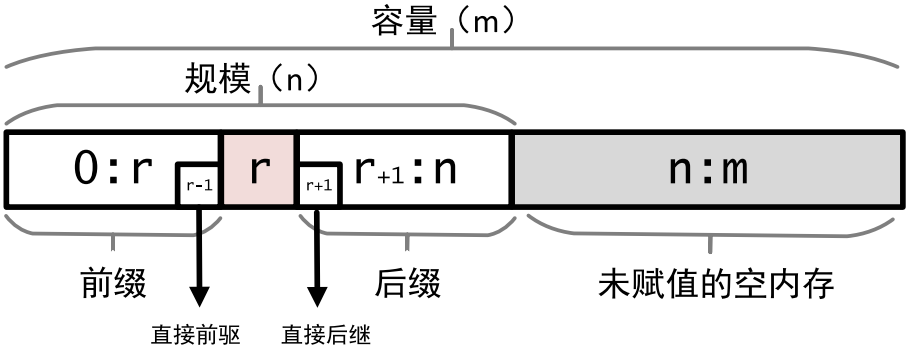

图 2.2　向量、静态链表和动态链表的对比

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],...,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。

第 2.3 节　循秩访问

向量的核心特征是**循秩访问**（access by rank）。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。因为下标（秩）总是非负的，所以我们用 `std::size_t` 类型来存储它。

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功能，通过重载 operator[]方法，像访问数组一样访问向量中的元素。

```
template <typename T>
class AbstractVector : public LinearList<T, /* iterators */> {
protected:
    virtual T* data() = 0;
public:
    virtual std::size_t capacity() const = 0;
    virtual void reserve(std::size_t n) = 0;
    virtual std::size_t size() const = 0;
    virtual void resize(std::size_t n) = 0;
    T& operator[](std::size_t r) { // definition (1)
        return data()[r];
    }
    const T& operator[](std::size_t r) const { // definition (2)
        return data()[r];
    }
};
```

【C++学习】

在类的成员函数之后加上 const 关键字，表示这个成员函数是一个只读的成员函数，它不会修改对象的状态。这样的成员函数可以被 const 对象调用，也可以被非 const 对象调用。但是，const 对象不能调用非 const 成员函数，因为这些函数可能修改对象的状态。

这会带来一个问题，以 operator[]为例，当向量 V 是 const 对象时，按照上述定义（1）的 operator[]无法被调用，也就是说我们无法通过 V[r]的方式去访问向量中的元素。反之，如果只采用定义（2），那么 const 对象固然可以调用 operator[]了，但是返回的 V[r]是一个可修改的元素，这就违背了 const 的语义。

为了解决这个问题，C++引入了 const 成员函数的重载。我们可以引入一个 const 版本的 operator[]，它返回的是一个只读的元素。这样，const 对象可以通过 V[r]的方式去访问向量中的元素，而且返回的元素是只读的。当然这样还有一个编程效率上的小问题，就是我们需要写两个完全相同的函数体。这个问题在 C++23 引入“this 捕获器”特性之后得以解决；本书目前使用的编译器暂时不支持该特性。

```
template <typename Self>
auto&& operator[](this Self&& self, std::size_t r) {
    return std::forward<Self>(self).data()[r];
}
```

最后，操作底层内存的时候因为不能获得所有权，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 `protected`，这样只有子类可以访问它。

现在，我们已经拥有了一个抽象类 `AbstractVector`，它被称为**抽象数据类型**（Abstract Data Type，ADT）。ADT 可以认为是数据结构的接口，它定义了数据结构的行为，但独立于数据结构的具体实现。如果读者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；读者也可以继承本书提供的示例向量类，重写其中的部分方法。以下将展示本书的示例实现。

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

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

【C++学习】

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。它的主要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。本书中若无特殊情况，将总是使用智能指针来表示所有权，避免在任何地方使用 `delete` 关键字释放内存。

第 2.4 节 向量的容量和规模

第 2.4.1 节 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，应当实现相应的构造函数。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，所以只能将 `data` 赋值为 `nullptr`，就像在 第 2.3 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(std::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。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& other) : Vector(other.m_size) {
    std::copy_n(other.m_data.get(), other.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。如前所述，**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所有权，我们永远不会从被移动后的对象里访问那些数据。

```
Vector(Vector&& other) noexcept : m_data { std::move(other.m_data) }
                                , m_capacity { other.m_capacity }
                                , m_size { other.m_size } {
        other.m_capacity = 0;
        other.m_size = 0;
    }
```

【C++学习】

移动构造函数和移动赋值运算符通常被声明为 `noexcept`，因为如果一个对象在移动过程中抛出异常，那么该对象可能处于一个无效的状态。而由于移动语义的设计，源对象在移动操作后可能不再保留其原有状态，这使得异常处理变得困难。此外，应当避免抛出异常的析构函数也通常被声明为 `noexcept`。

通过对智能指针调用 `std::move`，我们可以在常数时间内，将被移动对象的数据转移到新对象里。被移动之后，`other` 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。

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。

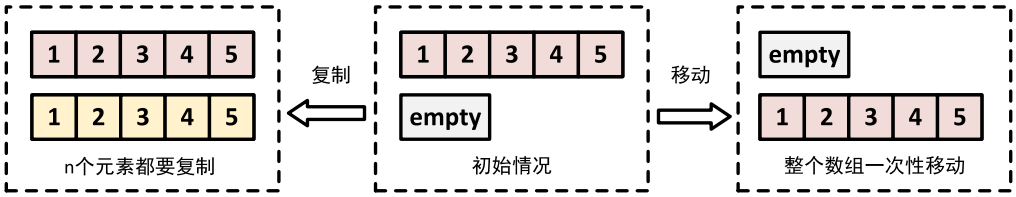

图 2.3　复制和移动的语义区别

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（不破坏源），又定义了一个插入 T&& 类型的方法用于移动（破坏源）。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，在读者的日常编程中需要注意，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，而不应该只实现复制。

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），编译器就不会自动生成其他的函数。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，rule of zero）。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，rule of five）。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容器，可以直接使用 STL 方法移动。

```
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（正如 std::vector 一样）。

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

第 2.4.2 节　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）。

刚开始时，装填因子一定在$[0, 1]$之间。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）。相反，如果向量的规模n变得很小，那么m可能会远大于n，这时就需要令m减小，这一操作称为**缩容**（shrink）。扩容和缩容的操作都是需要花费时间的，因此我们希望尽量减少它们的发生次数。在对时间性能要求非常严格的场景下（如算法竞赛），有可能禁止扩容和缩容，转为预先分配足够大的内存，这样可以避免动态分配内存的时间开销。

在一般场景下，扩容是非常常见的操作，但现实中很少进行缩容。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实现缩容：

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*手动缩容*的备用方案。即，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，本书的示例代码中没有实现缩容。

第 2.4.3 节　扩容

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。

```
void reserve(std::size_t n) override {
    if (n > m_capacity) {
        auto tmp { std::make_unique<T[]>(n) };
        std::move(m_data.get(), m_data.get() + m_size, tmp.get());
        m_data = std::move(tmp);
        m_capacity = n;
    }
}
```

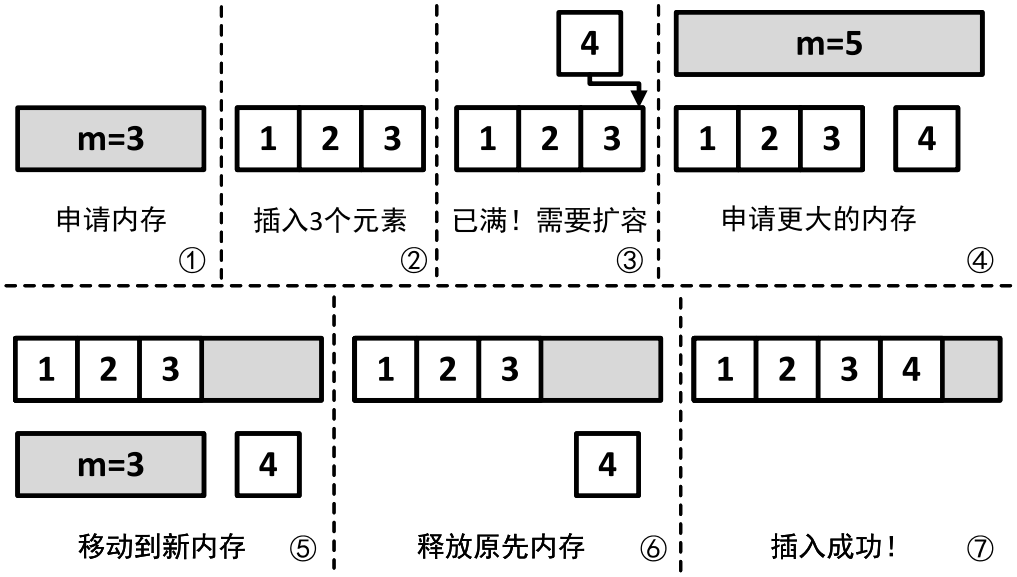

图 2.4 扩容的过程

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。

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 第 2.4.5 节 里将会讨论一些扩容策略。

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

如上所述，我们还设计了一个方法 `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。*

第 2.4.4 节 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$n \to n+1 \to n+2 \to ...$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 进行手动扩容。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自动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 VectorAllocator :
    public Algorithm<std::size_t(std::size_t, std::size_t)> {
protected:
    virtual std::size_t expand(std::size_t capacity
                               , std::size_t size) const = 0;
    virtual std::size_t shrink(std::size_t capacity
                               , std::size_t size) const {
        return capacity;
    }
public:
    std::size_t operator()(std::size_t capacity
                           , std::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容量改变的策略。在 第 2.4.5 节，将继承这个类并重写 expand 方法，以实现不同策略的扩容。缩容方法则直接返回 capacity（永不缩容）；如果需要实现缩容的时候，也可以使用这个模板，重写 shrink 方法。

第 2.4.5 节 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$n \to n+d \to n+2d \to ...$的次序扩容。

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

【C++学习】

`requires` 表示模板参数必须要满足后面的条件，否则无法通过编译。当 `D` 为 0 时，等差数列扩容的公差是 0，这是一个无意义的操作，因此我们禁止这种情况的发生。使用 `requires` 代替传统的 `std::enable_if` 有助于简化在模板元编程中限制，有助于更好地实现 SFINAE（Substitution Failure Is Not An Error）原则。

SFINAE 原则是模板元编程中的重要原则。当使用了 SFINAE 实现对于同一个函数模板的多个重载时，对于函数模板的每一个实例，编译器会尝试将实参代入到模板参数中，如果代入失败，编译器会尝试下一个重载，而不是报错。比如，我们可以同时定义带有 `requires std::is_same_v<T, int>`以及 `requires std::is_same_v<T, std::string>`的两个重载，当传入一个 `int` 时，编译器会选择第一个重载，而传入一个 `std::string` 时，编译器会选择第二个重载。

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$n \to qn \to q^2n \to ...$的次序扩容。

```
template <typename Q> requires (Q::num > Q::den)
class VectorAllocatorGP : public VectorAllocator {
protected:
    std::size_t expand(std::size_t capacity
                       , std::size_t size) const override {
        std::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$0\to1\to2\to3\to4\to6\to9\to...$，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。

第 2.4.6 节 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$O_1,O_2,...,O_n$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**（amortized complexity），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$d,2d,3d,...,n$，共进行$\frac{n}{d}$次扩容。因此，分摊复杂度为：

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

另一方面，从空间效率的角度，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%。

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

$$T(n)=\frac{q+q^2+q^3+...+n}{n}=\frac{q}{q-1}=O(1)$$

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$\frac{3}{2}$。GCC、Clang 和 [4] 选择的公比是 2；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。

第 2.5 节 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。

第 2.5.1 节 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]`。

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 图 2.5 所示的算法，其时间复杂度为$\Theta(n-r)$，和插入位置相关，这类算法被称为**输入敏感**（input-sensitive）的。

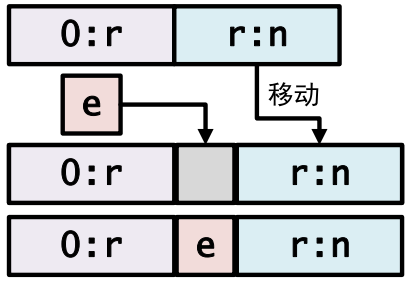

图 2.5 往向量里插入一个元素

```
iterator insert(iterator p, const T& e) override {
    if (m_size == m_capacity) {
        reserve(m_allocator(m_capacity, m_size));
    }
    std::move_backward(p, end(), end() + 1);
    *p = e;
    ++m_size;
    return p;
}
```

【C++学习】

为了让我们在 第 2.4.4 节 定义的扩容策略被嵌入到向量里来，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 A = VectorAllocatorGP<std::ratio<3, 2>>
    requires std::is_base_of_v<VectorAllocator, A>
class Vector : public AbstractVector<T> { /* ... */ };
```

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 `A` 必须是我们上面实现的 `VectorAllocator` 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）；也可以使用自定义的扩容策略。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。

第 2.5.2 节　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（如在算法比赛中，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、帕累托（Pareto）分布和泊松（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。就分摊时间复杂度一样，平均时间复杂度的计算也经常是非常复杂且困难的，只需要了解其基本的技术即可。

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。
4. 分摊时间复杂度依赖于一系列连续操作，但不同的操作序列可能导致不同的分摊复杂度。因此，分摊时间复杂度可以叠加“最坏”、“最好”和“平均”的修饰词，而平均时间复杂度是和最坏、最好并列且互斥的。

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，但它超出了本书和一般计算机工程师需要掌握的范围。

现在回到插入的算法，它的时间复杂度是$\Theta(n-r)$（不考虑扩容）。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$\Theta(n)$。

第 2.5.3 节 查找一个元素

查找需要返回被查找元素所在的位置。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所在的位置。对于更加复杂的查找类型，比如按区间查找（给定被查找元素所在区间）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。

```
iterator find(const T& e) const override {
    for (auto it { begin() }; it != end(); ++it) {
        if (*it == e) {
            return it;
        }
    }
    return end();
}
```

【C++学习】

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迭代器，除了返回序列尾部溢出的 `end()`之外，返回序列头部溢出的`--begin()`也是常见的选择。此外，也可以将返回值的类型改为 `std::optional<iterator>`，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。

第 2.5.4 节　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`。

```
iterator erase(iterator p) override {
    std::move(p + 1, end(), p);
    --m_size;
    return p;
}
```

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。如果您自己完成了向量的设计，可以使用 **`vector.cpp`** 进行简单的功能测试，并在实验中将 `dslab::Vector` 替换为自己的向量类。

第 2.5.5 节 插入连续的元素

实验 `vins.cpp`。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 第 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实验结果的影响，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。

n	$d=64$	$d=4096$	$q=\frac{3}{2}$	$q=2$	$q=4$
200K	444	8	1	1	0
400K	2241	36	1	1	1
600K	3649	59	1	2	0
800K	5474	74	3	0	1
1M	6388	96	2	2	2

表 2.1 数组求和算法的时间

如 表 2.1 所示，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。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，因此我们可以判断出，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。

第 2.5.6 节 向量合并

实验 `vcat.cpp`。在 第 2.5.5 节 中讨论的是连续插入元素情况，实际上不应该存在。一方面，这种情况不应该使用向量的自动扩容策略，而应该预先调用 `reserve` 函数进行预分配。这样可以避免不必要的扩容操作，提高效率。另一方面，因为要插入的不是单个元素，而是多个元素，插入操作的处理方式也可以有所不同。本节讨论一次性插入多个元素的问题，亦即向量合并问题。假设我们有两个向量 `V[0:n]`和 `W[0:m]`，我们希望将整个 `W` 插入到 `V` 的位置 `r` 处，实现向量合并。

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

```
Vector<T>& operator()(std::size_t r) override {
    auto pos { V.begin() + r };
    for (auto&& e : W) {
        V.insert(pos++, std::move(e));
    }
```

```
    return V;
}
```

读者可以敏锐地发现，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 W 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]，W，V[r:n]。其中，V[r:n]被转移到了 V[r+m:n+m]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。

```
Vector<T>& operator()(std::size_t r) override {
    V.resize(V.size() + W.size());
    std::move_backward(V.begin() + r, V.end() - W.size(), V.end());
    std::move(W.begin(), W.end(), V.begin() + r);
    return V;
}
```

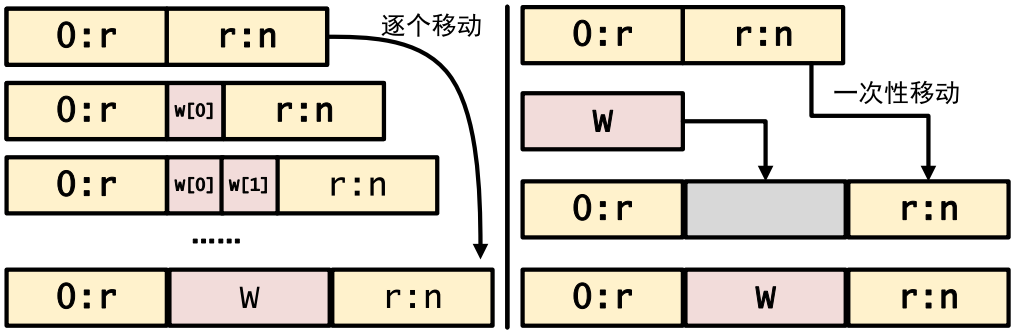

图 2.6　向量合并的两种方法

在示例程序中已经预先 reserve，所以这里 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$m = O(1)$，而大量可以认为$m = \Theta(n)$。

实验场景	逐个插入	批量插入
插入少量元素到开头	2	0
插入大量元素到开头	1257	0
插入少量元素到结尾	0	0
插入大量元素到结尾	0	0

表 2.2　向量合并算法的时间

在 表 2.2 中，$n = 2 \times 10^5$，少量元素$= 10^2$，大量元素$= 10^5$。可以看出，当插入元素到结尾时，两种方法的时间性能接近，而插入元素到开头时，二者有着巨大的差异，尤其是在插入大量元素时。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问题中被广泛使用。

第 2.5.7 节　按值删除元素

实验 vrm.cpp。在 第 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。

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。

【C++学习】

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，读者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。和 第 2.5.6 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。

```
void remove(const T& e) override {
    while (true) {
        if (auto r { std::find(V.begin(), V.end(), e) }; r != V.end()) {
            V.remove(r);
        } else {
            break;
        }
    }
}
```

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 第 2.5.3 节 介绍的查找一样，时间复杂度为$\Theta(r)$。而另一方面，按照 第 2.5.4 节 的介绍，在找到之后 erase 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$\Theta(n^2)$。

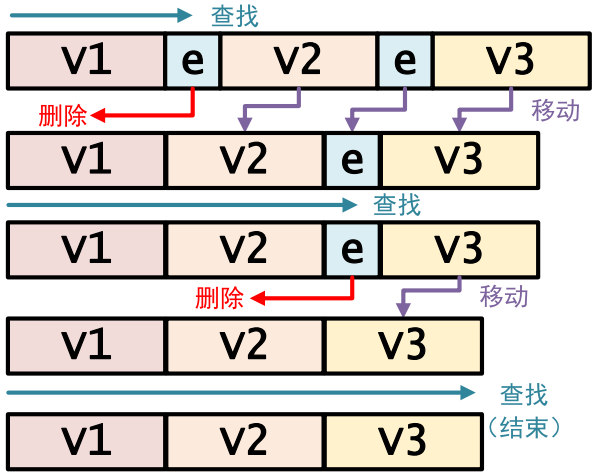

图 2.7　向量按值删除元素：逐个查找、逐个删除

在设计算法的时候，不一定能直接想到最佳的复杂度。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`m`次；我们通过合并这些移动实现了算法优化。

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`V1`，第二次检索了`V1+V2`，第三次检索了`V1+V2+V3`；这里`V1`被检索了 3 次，`V2`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。

```
void remove(const T& e) override {
    auto r { V.begin() };
    while (r = std::find(r, V.end(), e), r != V.end()) {
        V.erase(r);
    }
}
```

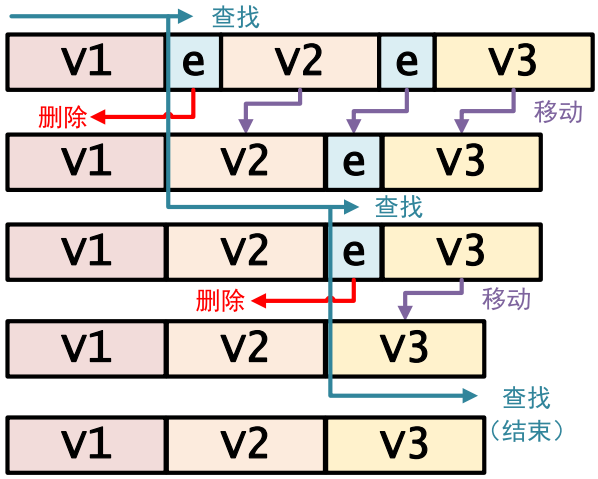

图 2.8 向量按值删除元素：全局查找、逐个删除

我们发现上述算法的优化仍然无法提高最坏时间复杂度。在最坏的情况下（所有元素都要被删除），光是 erase 就要花费$\Theta(n^2)$的时间，所以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 erase 入手，需要将 erase 的工作展开来，看看其中哪些是不必要的。

在 erase 中，主要消耗时间的是*元素移动*的操作。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] → V[i-3] → ... → V[i-k-1] 的位置上。比如，在 图 2.8 中，V3 就移动了 2 次。您可以敏锐地发现这正是前面所讲述的*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，无论是上面的哪一种算法，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。因此，V[i] 的移动是可以直接到位的。这样，我们就可以设计出一个更加高效的算法。

```
void remove(const T& e) override {
    auto fp { V.begin() }, sp { V.end() };
    while (fp != V.end()) {
        if (*fp != e) {
            *sp++ = std::move(*fp);
        }
        ++fp;
    }
    V.resize(sp - V.begin());
}
```

【C++学习】

利用 STL，上述算法有一个简单的写法：

```
void remove(const T& e) override {
    V.resize(std::remove(V.begin(), V.end(), e) - V.begin());
}
```

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。

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。

图 2.9　向量按值删除元素：快慢指针

如 表 2.3 所示，逐个查找的时间效率是最低的，而全局查找只有常数级优化。

n	逐个查找	全局查找	快慢指针
10^3	0	0	0
10^4	13	9	0
10^5	1425	879	0

表 2.3　向量按值删除算法的时间

第 2.6 节 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，经常被用来承担一些辅助功能。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。

第 2.6.1 节 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。

置乱算法接收一个向量将它置乱。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。想必读者非常熟悉洗牌。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,...,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。

```
void operator()(Vector<T>& V) override {
    for (auto i { V.size() - 1 }; i > 0; --i) {
        auto j { Random::get(i) };
        std::swap(V[i], 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 std::size_t!），那么这个算法就能将所有的 n!个排列等概率地输出。

【C++学习】

C++11 在 STL 中也提供了置乱算法，需要包含<random>库使用。

```
void operator()(Vector<T>& V) override {
    std::shuffle(V.begin(), V.end(), Random::engine());
}
```

这里的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。图 2.10 展示了 4 个元素随机置乱10^6次的结果，全部的 24 种可能情况出现的频率非常接近，平均相对误差只有大约 0.4%。

图 2.10　4 个元素随机置乱的频率分布

下面回到随机数生成器的问题上来，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 $n \geq 3$ 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 通常是伪随机。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。

第 2.6.2 节 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，首先需要界定清楚，**序**（order）是一个什么东西。在 第 1.4.2 节 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“≤”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$\preccurlyeq$满足：

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

那么称$\preccurlyeq$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“≤”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》里通常讨论排序问题时，也总是假定数据结构中的元素数据类型具有全序关系。

【C++学习】

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 `operator<`），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。

传统上对于自定义类型通常通过重载 `operator<`实现。C++20 引入了航天飞机运算符 `operator<=>`，定义该运算符之后会自动生成 `operator<`、`operator>`、`operator<=`和 `operator>=`，这样就可以方便地定义全序关系了。对于现实中遇到的大多数情况，都可以通过下面的方式定义全序关系：

```
auto operator<=>(const T& other) const = default;
```

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$\preccurlyeq$满足：

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

那么称$\preccurlyeq$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”，$\preccurlyeq$定义为“考生 x 的每一门分数都小于等于考生 y”。显然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$\preccurlyeq$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。

【C++学习】

在 C++20 中定义了各种序关系，用于作为 `operator<=>`的返回值类型。比如，偏序关系被定义为 `std::partial_ordering`，包括大于、小于、等价以及无法比较四个值。此外，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`。这两者的区别在于“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。一般而言，全序关系即对应强序关系，而经过一个映射的全序关系（如二维坐标只比较一个维度） 则对应弱序关系。

第 2.6.3 节　自上而下的归并排序

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$x \preccurlyeq y$，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$\preccurlyeq$，即可以回归为升序的情况进行处理。

```
template <typename T, template<typename> typename L = DefaultVector>
    requires std::is_base_of_v<LinearList<T
                , typename L<T>::iterator
                , typename L<T>::const_iterator>, L<T>>
class AbstractSort : public Algorithm<void(L<T>&)> {
protected:
    std::function<bool(const T&, const T&)> cmp { std::less<T>() };
    virtual void sort(L<T>& L) = 0;
public:
    template <typename Comparator>
    void operator()(L<T>& L, Comparator&& cmp) {
        this->cmp = cmp;
        sort(L);
```

```
    }
    void operator()(L<T>& L) override {
        sort(L);
    }
};
```

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）[16]。归并排序的发明人是大名鼎鼎的冯·诺依曼（von Neumann），这位“计算机之父”在 1945 年设计并实现了该算法。

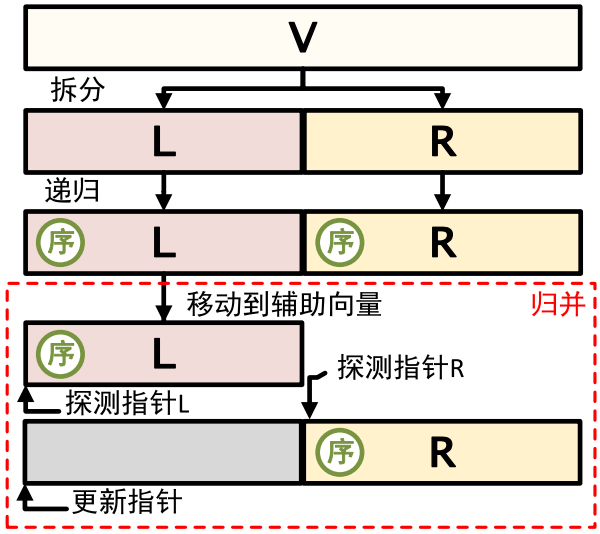

图 2.11　向量的归并排序

归并排序的设计采用的仍然是递归的思想：

1. 规模$n \leq 1$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$n - \frac{n}{2}$（在第 1.4.4 节 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）。

```
void merge(iterator lo, iterator mi, iterator hi, std::size_t ls) {
    static Vector<T> W {};
    W.resize(ls);
    std::move(lo, mi, W.begin());
    auto i { W.begin() }, 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, std::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 / 2);
    }
    void sort(L<T>& V) override {
        mergeSort(V.begin(), V.end(), V.size());
    }
```

【C++学习】

C++的 STL 提供了 `std::inplace_merge` 函数（传入 `lo`、`mi` 和 `hi` 的迭代器）来进行归并，这个函数可以用来替代上述实现中的 `merge`。对于向量来说，左半部分的长度 `ls` 可以通过 `std::distance(lo, mi)`得到，所以也可以不用传入 `ls`。

我们注意到在归并的一开始，将排序好的前半部分移动到了辅助向量中。那么，为什么前半部分需要移动出去，而后半部分不需要呢？这是因为在归并的过程中，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，快慢关系是始终能保持的，不会有元素被错误地覆盖掉。而当前半部分全部加入之后，`i` 到达了辅助向量 `W` 的末尾，循环结束。

另一方面，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呢？这是因为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` 的长度至少为最大的 `ls`，也就是$\frac{n}{2}$，因此空间复杂度为$\Theta(n)$。递归产生的$\Theta(\log n)$，相比于辅助数组的$\Theta(n)$来说可以忽略。归并排序的时间复杂度则是一个非常典型的问题。容易发现 `merge` 的时间复杂度为$\Theta(n)$（这里假定 `cmp`

是$O(1)$的），因而可以得到递归式$T(n)=2T\left(\frac{n}{2}\right)+\Theta(n)$。此类递归式的求解通常可以用**主定理**（master theorem）[17] 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 [15]，这里只陈述结论。

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

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

根据主定理可以得到归并排序的时间复杂度为$\Theta(n\log n)$。

第 2.6.4 节　自下而上的归并排序

上一节中展示的归并排序是自上而下的。这里的“自上而下”指的是，我们首先调用整个向量的归并排序，在这个函数中递归地调用子向量的归并排序。以此类推，直到“最下方”的递归实例，也就是递归边界（只有一个元素的向量）。

自上而下的归并排序是归并排序的经典实现，下面介绍一种自下而上的方法。我们分析归并函数 `merge` 的调用次序会发现，由于归并发生在递归调用之后，所以归并的次序反而是自下而上的：首先对“最下方”的子向量做归并，得到长度为 2 的有序子向量。一个子向量可以被归并，当且仅当它的左半和右半的子向量已经被归并完成。

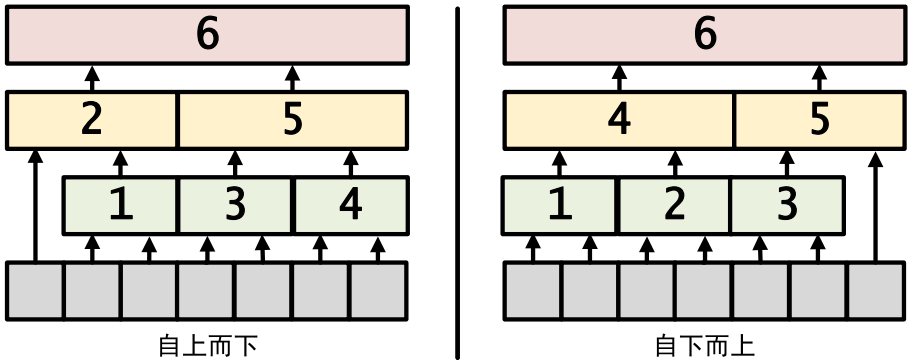

图 2.12　自上而下和自下而上的归并次序

如 图 2.12 中左图所示，在自上而下的归并排序中，只要左半和右半的子向量都已经被归并完成，就会引发这两个子向量的归并。对于 7 个元素组成的向量，一共会触发 6 次归并，归并的次序在图中用 1 至 6 表示。由于一个向量的归并，只需要保证它左半和右半归并完成之后进行，而并不要求在左半和右半归并完成之后立即进行，所以我们可以调整这 6 次归并的次序。我们首先将向量视作长度为 1 的段两两合并，然后将它视作长度为 2 的段（每个段内已经有序）两两合并，

再视作长度为 4 的有序段两两合并，以此类推，直到归并整个向量为止。上面的每一步称为一**趟**（run）。这种方法同样保证了一个子向量晚于它的左半和右半被归并，因而也能保证正确性。

如 图 2.12 所示，因为向量规模不一定是 2 的幂次，每一趟的末尾处需要特殊处理。从图中还可以看出，自下而上的归并排序对于子向量的分拆方式，是有可能和自上而下不同的。下面展示了自下而上的归并排序的一个实现，它复用了自上而下版本的 `merge` 函数，只是在调用 `merge` 的次序上和自上而下的版本有所区别。

```
void sort(L<T>& V) override {
    auto n { V.size() };
    for (auto w { 1uz }; w < n; w *= 2) {
        auto lo { V.begin() };
        auto i { 2 * w };
        while (i < n) {
            auto mi { lo + w };
            auto hi { mi + w };
            merge(lo, mi, hi, w);
            lo = hi;
            i += 2 * w;
        }
        if (n + w > i) {
            auto mi { lo + w };
            auto hi { V.end() };
            merge(lo, mi, hi, w);
        }
    }
}
```

外层循环进行的次数（也就是趟数）很显然是$\lceil \log n \rceil$，每一趟的时间复杂度为$\Theta(n)$。因此，自下而上的归并排序时间复杂度同样是$\Theta(n \log n)$。

自下而上的归并排序的时间消耗随n的增长是不平滑的。当n从2^k增长到$2^k + 1$时，自下而上的归并排序需要增加整整一趟；因此，它在n略小于或等于一个 2 的幂次的时候表现更好，而在略大于一个 2 的幂次时表现较差。

第 2.6.5 节　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：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*。使用信息论的思路，有助于在算法设计的过程中辅助自己判断是否达到了最优的时间复杂度，也有助于记忆知识点。

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，而在排序算法结束之后，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：

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。

第 2.6.6 节　信息熵

基于 第 2.6.5 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 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：

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

现在联系 第 2.6.5 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。

第 2.6.7 节　有序性和逆序对

第 2.6.5 节 和 第 2.6.6 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。

第 2.6.8 节　先验条件下的归并排序

实验 `vmergesort.cpp`。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，以下展示了一种可能的实现。

```
void sort(L<T>& V) override {
    auto mi { --V.end() };
    while (mi != V.begin() && cmp(*(mi - 1), *mi)) {
        --mi;
    }
    auto max_left { *std::max_element(V.begin(), mi) };
    auto left { std::lower_bound(mi, V.end(), max_left, cmp) };
```

```
    mergeSort(V.begin(), left, std::distance(V.begin(), left));
}
```

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：

```
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；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。

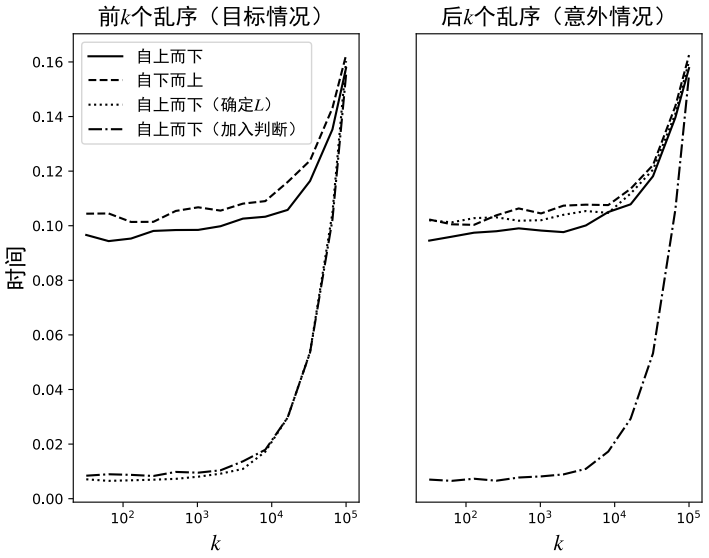

图 2.13 归并排序在基本有序的情况下的性能比较

如 图 2.13 所示，在上述两种方法中，从递归方程形式入手的方法具有更好的泛用性。右图测试了一个对偶的问题：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。从图中还可以看出，即使是没有做针对性优化的版本，由于有序段内不需要移动后半段，在“整体有序”的情况下性能也可以得到一定的提升。

第 2.6.9 节　原地归并排序

此前所展示的归并排序都是需要辅助空间的，在归并的过程中，需要将前半部分的元素移动到辅助空间中。这样的归并排序称为**非原地**（non-in-place）归并排序。在实际应用中，非原地归并排序的空间消耗是一个潜在的问题。在这一节中，我们介绍一种**原地**（in-place），即空间复杂度$O(1)$的归并排序的方法。

为了降低空间复杂度，我们有两种思路：

1. 直接降低**归并**的空间复杂度，即在合并两个长度为 n 的有序序列时，不再需要$\Theta(n)$的辅助空间。这一思路是相当困难的 [16] 。
2. 不降低**归并**的空间复杂度，即在合并有序序列的时候仍然使用辅助空间，但是让这个辅助空间“羊毛出在羊身上”，利用原向量中的空间作为辅助空间使用。这一思路的基础是：当我们将前半部分的元素移动到辅助空间后，原有的空间是空出来的；而当我们完成归并之后，辅助空间又被空了出来。也就是说，辅助空间并不一定需要是实际新开辟出来的一块空闲的空间，它可以原先存放一些数据。当我们将前半部分的元素移动到辅助空间时，辅助空间原先存放的数据也被交换到了前半部分。而随着归并的进行，这些数据又被移回了辅助空间里。

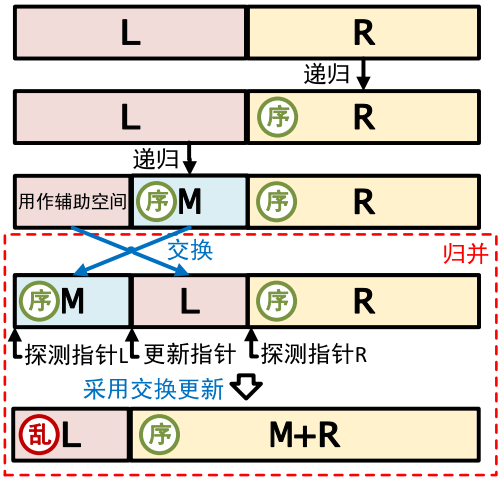

图 2.14　原地归并排序

在经典的自上而下的归并排序中，最后一趟会归并两个长度为$\frac{n}{2}$的向量，需要长度为$\frac{n}{2}$的辅助空间，但这个时候原向量没有可以“薅羊毛”的空间了，因此无法使用上面的思想原地归并。但容易想到的是，归并所消耗的空间是**不对称**的，前半部分越长，所需的辅助空间越大，而后半部分的增长并不会影响对辅助空间的需求量。为此，我们可以改变分治的方式，不再采用二等分的形式，而是采用**不对称**的分法，如 图 2.14 所示。我们先取$\frac{n}{2}$长的向量作为“后半部分”递归排序，再取$\frac{n}{4}$长的向量作为“前半部分”递归排序；此时，剩余的$\frac{n}{4}$长的未排序部分，恰好能作为足以容纳前半部分的辅助空间。于是，我们可以顺利进行归并，得到$\frac{n}{4}$长的未排序部分，以及$\frac{3}{4}n$长的已排序部分。接下来，再从未排序部分中取出一半$\frac{n}{8}$递归排序并作为“前半部分”，和已排序的$\frac{3}{4}n$归并，此时剩余的$\frac{n}{8}$又可以被当做辅助空间使用，以此类推。当分治算法的各部分对时间或空间的使用量不对称时，顺应这个不对称性，采用不对称的分法，是优化分治算法的重要思路。

```
void merge(iterator lo, iterator mi, iterator hi, iterator tmp) {
    std::swap_ranges(lo, mi, tmp);
    auto i { tmp }, j { mi }, k { lo };
    auto te { tmp + std::distance(lo, mi) };
    while (i != te && j != hi) {
        if (cmp(*j, *i)) {
            std::iter_swap(k++, j++);
        } else {
            std::iter_swap(k++, i++);
        }
    }
    while (i != te) {
        std::iter_swap(k++, i++);
    }
    while (j != hi) {
        std::iter_swap(k++, j++);
    }
}
void mergesort(iterator lo, iterator hi) {
    if (std::distance(lo, hi) < 2) return;
    auto mi2 { hi - std::distance(lo, hi) / 2 };
    mergesort(mi2, hi);
    while (std::distance(lo, mi2) > 1) {
        auto mi1 { mi2 - std::distance(lo, mi2) / 2 };
        mergesort(mi1, mi2);
        merge(mi1, mi2, hi, lo);
        mi2 = mi1;
    }
```

```
    merge(lo, mi2, hi, W.begin());
}
```

如上所述，只有最后一次 merge 的时候，因为两个待归并向量已经填满了整个向量，没有多余的辅助空间了，所以需要使用 W 作为辅助空间。但是我们的减半策略使得最后一次 merge 的时候，前半部分的长度必定为 1，因此只需要$O(1)$的辅助空间。这样，我们就实现了原地归并。

以上述方法实现的原地归并排序保持了原有归并排序的时间复杂度，但是存在两个方面的问题。首先，如 图 2.14 所示，在原地归并的过程中，用作辅助空间的区域 L 有可能会被直接交换到左侧（M 中元素小的情形），也有可能先被交换到右侧若干次（R 中元素小的情形），再交换到左侧。这种不确定性导致原地归并之后的 L 和归并前的 L 不再是相同的，而是会被打乱次序。当然，如果向量中的元素两两不相等，那么排序的最终结果总是正确的；问题在于，如果向量中存在两个相等的元素a_1, a_2，且a_1位于a_2之前，那么以此法排序之后，a_1是有可能位于a_2之后的。这种不能保持相等元素的相对次序的排序称为**不稳定**（unstable）排序，相应地，如果相等元素的相对次序不变，那么称为**稳定**（stable）排序。此前介绍的自上而下或自下而上的归并排序都属于稳定排序，而从上面的分析可以知道原地归并排序是不稳定的。其次，原地归并排序的空间复杂度虽然是$O(1)$，但是归并排序中的移动（move）操作变成了交换（swap），时间常数大幅提高。由于这两方面的原因，原地归并排序的实际应用范围有限。

第 2.7 节 有序向量上的算法

第 2.7.1 节 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 L = DefaultVector>
    requires std::is_base_of_v<LinearList<T
                , typename L<T>::iterator
                , typename L<T>::const_iterator>, L<T>>
class AbstractSearch : public Algorithm<
                  typename L<T>::iterator(const L<T>&, const T&)> {
protected:
    using iterator = typename L<T>::iterator;
    std::function<bool(const T&, const T&)> cmp { std::less<T>() };
    virtual iterator search(const L<T>& V, const T& e) = 0;
public:
    template <typename Comparator>
```

```
    iterator operator()(const L<T>& V, const T& e, Comparator&& cmp) {
        this->cmp = cmp;
        return search(V, e);
    }
    iterator operator()(const L<T>& V, const T& e) override {
        return search(V, e);
    }
};
```

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] <= 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。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。

```
iterator search(const Vector<T>& V, const T& e
              , iterator lo, iterator hi) const {
    if (std::distance(lo, hi) <= 1) {
        if (lo != V.end() && cmp(*lo, e)) {
            return hi;
        } else {
            return lo;
        }
    }
    auto mi { lo + std::distance(lo, hi) / 2 };
    if (cmp(e, *mi)) {
        return search(V, e, lo, mi);
    } else {
        return search(V, e, mi, hi);
    }
}
```

上面的算法，时间复杂度和空间复杂度均为$\Theta(\log n)$。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 第 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 `V.end()`。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 < *mi` 就只保留前半段（`lo` 到 `mi` 的部分），所以我们在任何时刻都可以保证，`e < *hi`（可认为初始值`*V.end() = +∞`）。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`*lo <= e`；注意由于 `lo` 的初始值是有意义的 `V.begin()`，不像 `hi` 的初始值是无意义的 `V.end()`，所以后面这个式子只有 `*lo != V.begin()`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，从 `lo` 到 `mi` 之间有且只有一个元素`*lo`。此时，可以分成小于、等于、大于三种情况讨论。

1. 如果`*lo < e`，那么我们可以定位到`*lo < e < *hi`，此时返回 `hi`，就是大于 `e` 的第一个元素。
2. 如果`*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 < *hi`，所以如果有多个等于 `e` 的元素，则返回的是最大的一个。
3. 如果`*lo > e`，根据前面的分析可知，这种情况只可能发生在 `lo = V.begin()` 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。

第 2.7.2 节　消除尾递归

查找`V[0:n]`中某个元素 `e` 的位置，这个问题在计算前有$n+1$种（包括不存在情形的 `V.end()`）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）。在实际的编程过程中如果开启了编译器优化选项，则尾递归在通常会被编译器自动优化。

本小节只介绍对于尾递归的消除方法，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尾递归。

```
T recursive(Args... args) override {
    if (is_base_case(args...)) {
        return base_case(args...);
    } else {
        next_args(args...);
```

```
        return recursive(args...);
    }
}
```

上面的递归算法可以改写成如下的迭代算法：

```
T iterative(Args... args) override {
    while (!is_base_case(args...)) {
        next_args(args...);
    }
    return base_case(args...);
}
```

上面这种形式是非常通用的，is_base_case 是判断是否到达递归边界的函数，base_case 是递归边界的处理函数，next_args 是递归函数的参数更新函数。下面给出了折半查找的迭代形式。

第 2.7.3 节　迭代形式的折半查找

实验 vsearch.cpp。将折半查找的递归形式进行拆解，分解出 is_base_case、next_args 和 base_case 三个函数，然后代入到上面的迭代模板中，就可以将其改写为迭代形式。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。

```
iterator search(const Vector<T>& V, const T& e) override {
    auto lo { V.begin() }, hi { V.end() };
    while (std::distance(lo, hi) > 1) {
        auto mi { lo + std::distance(lo, hi) / 2 };
        if (cmp(e, *mi)) {
            hi = mi;
        } else {
            lo = mi;
        }
    }
    if (lo != V.end() && cmp(*lo, e)) {
        return hi;
    } else {
        return 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10^7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，由于进行了一些抽象，使用模板的方法

性能会低于直接递归或迭代的性能。但即使是模板，它作为对数复杂度的算法，效率仍然远远高于在 第 2.5.3 节 中讨论的顺序查找。

n	递归	迭代	递归（模板）	迭代（模板）
10^4	317	169	479	440
10^5	401	186	499	500
10^6	511	226	548	578

表 2.4　向量按值删除算法的时间

现代 C++编译器如果开启了优化选项，通常可以在编译的过程中自动消除尾递归，所以在实际上机编程时，通常不需要刻意将尾递归改写成循环形式。但是如 表 2.4 所示，递归版本的性能仍然可能和迭代版本有一定差异（和编译器相关），所以简单的尾递归改写成循环形式仍然是有益的。此外，一些语言（比如 Python）拒绝提供尾递归优化 [18]，在 C++中运行良好的代码移植到这些语言上可能会发生问题，因此手动将尾递归改写成迭代形式仍然是非常重要的技能。

第 2.7.4 节　向量唯一化

实验 **vunique.cpp**。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（迭代器记录的位置）来区分。

一个简单但有效的解法是：从左到右考察向量 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。

```
void operator()(Vector<T>& V) override {
    for (auto i { V.begin() }; i != V.end(); ++i) {
        auto j { i };
        while (j = std::find(i + 1, V.end(), *i), j != V.end()) {
            V.erase(j);
        }
    }
}
```

上面的方法是比较基本的逐个删除。利用了我们在 第 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，可以对上面的算法做出简化。

```
void operator()(Vector<T>& V) override {
    for (auto i { V.begin() }; i != V.end(); ++i) {
        V.resize(std::remove(i + 1, V.end(), *i) - V.begin());
    }
}
```

在最好情况下（所有元素都相等），“删除-擦除”法可以达到$\Theta(n)$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 需要从 0 到 n 地遍历整个向量，所以在最坏情况下需要进行$\Theta(n^2)$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，外层循环的迭代器是 `V.begin()`到 `V.end()`；然而，`std::distance(V.begin(), V.end())`并不保持是一个常量 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。

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）。

```
void operator()(Vector<T>& V) override {
    auto sp { V.begin() }, fp { V.begin() };
    while (++fp != V.end()) {
        if (*sp != *fp) {
            *++sp = std::move(*fp);
        }
    }
    V.resize(++sp - V.begin());
}
```

【C++学习】

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`。

这个算法的时间复杂度是$\Theta(n)$的。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。它的时间复杂度为$\Theta(n \log n)$，优于前面的逐个“删除-擦除”的做法；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;
        std::size_t 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.clear();
        std::transform(V.begin(), V.end(), std::back_inserter(W),
            [](const T& e) { return Item { e, 0 }; });
    }
    void moveFromW(Vector<T>& V) {
        V.clear();
        std::transform(W.begin(), W.end(), std::back_inserter(V),
            [](const Item& item) { return item.value; });
    }
public:
    void operator()(Vector<T>& V) override {
        moveToW(V);
        std::sort(W.begin(), W.end());
        W.resize(std::unique(W.begin(), W.end()) - W.begin());
        std::sort(W.begin(), W.end(),
            [](const Item& lhs, const Item& rhs) {
            return lhs.rank < rhs.rank;
        });
        moveFromW(V);
    }
};
```

如 图 2.15 所示，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。

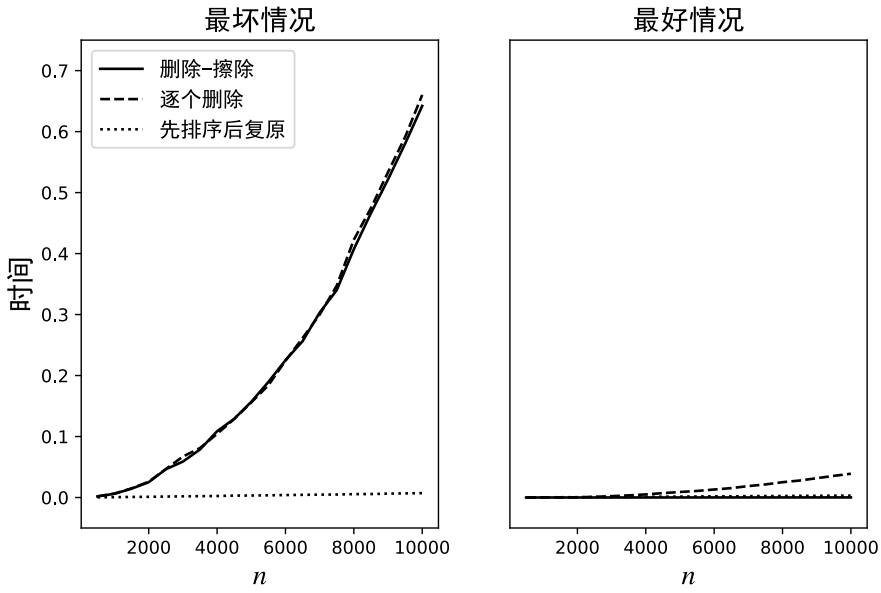

图 2.15　最坏和最好情况下的向量唯一化性能

第 2.8 节　循环位移

实验 `vrotate.cpp`。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。

给定向量 `V[0:n]`和位移量 `k`，则将原有的向量 `V[0],V[1],...,V[n-1]`，变换为 `V[k],V[k+1],...,V[n-1],V[0],V[1],...,V[k-1]`，称为**循环左移**。相应地，变换为 `V[n-k],V[n-k+1],...,V[n-1],V[0],V[1],...,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，右移的情况是对称的。

循环左移的过程可以表示为`(V[0:k],V[k:n])`→`(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) {
    auto tmp { std::move(a) };
    a = std::move(b);
    b = std::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 2.16 所示。

```
void operator()(Vector<T>& V, std::size_t k) override {
    Vector<T> W(k);
    std::copy(V.begin(), V.begin() + k, W.begin());
    std::move(V.begin() + k, V.end(), V.begin());
    std::move(W.begin(), W.end(), V.end() - 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$k=O(n)$，这一时间复杂度也可以简化为$\Theta(n)$。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。

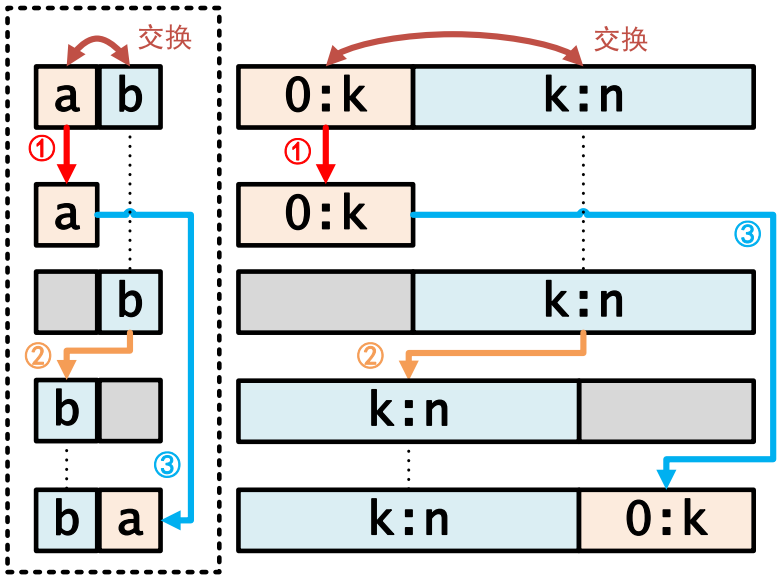

图 2.16　仿照交换元素，用三次移动实现循环左移

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。最后一次赋值，将辅助空间里的`V[i]`拿出来赋给`V[i-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趟“轮转交换”，各趟共享同一个辅助空间。图 2.17 展示了`n=12,k=3`的轮转交换例子。

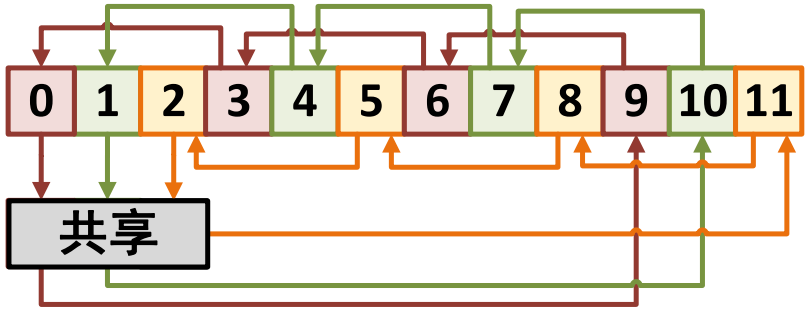

图 2.17 用轮转交换进行循环位移

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

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)，都成立$(i_1 + j_1k)\%n \neq (i_2 + j_2k)\%n$。

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$k = k_1d, n = n_1d$，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。

因此遍历顺序为：依次从 0,1,...,d-1 出发，以 k 为步长遍历$\frac{n}{d}$次回到起点。

```
void operator()(Vector<T>& V, std::size_t k) override {
    auto d { gcd(V.size(), k) };
    for (auto i { 0uz }; i < d; ++i) {
        auto tmp { std::move(V[i]) };
        auto cur { i }, next { (cur + k) % V.size() };
        while (next != i) {
            V[cur] = std::move(V[next]);
            cur = next;
            next = (cur + k) % V.size();
        }
        V[cur] = std::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$O(1)$。

在教材 [4] 上介绍了另一种解法：三次**反转**（reverse）。基于反转的原理是，为了实现(V[0:k],V[k:n])→(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`W[n:0]`（这里`W`表示反转后的向量`V`），我们可以将它拆分为`(W[n:k],W[k:0])`，然后再将这两段分别反转即可，如图 2.18 所示。

```
void operator()(Vector<T>& V, std::size_t k) override {
    std::reverse(V.begin(), V.end());
    std::reverse(V.begin(), V.begin() + k);
    std::reverse(V.begin() + k, V.end());
}
```

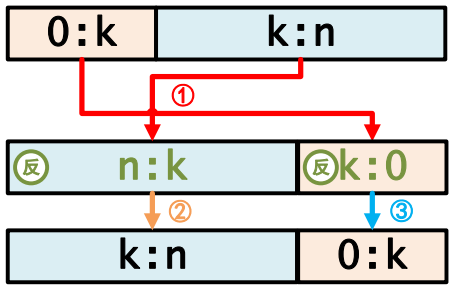

图 2.18　用三次反转进行循环位移

三次反转的时间复杂度同样是$\Theta(n)$，而空间复杂度是$O(1)$。从理论上分析：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）和释放本身需要时间，所以它在`k`比较大的时候反而会慢于三次反转。

【C++学习】

在C++中可以使用`std::rotate`直接进行循环位移。在这个例子中，编译器的标准库实现和优化策略会对三种方法的性能产生显著影响，您可以使用MSVC进行编译，然后和本书中展示的GCC编译结果进行对比。

`k`（$n=10^8$）	`std::rotate`	三次移动	轮转交换	三次反转
1	68	85	388	130
10000	88	89	605	114
10007（素数）	81	81	1129	117
5×10^7	61	221	208	116
9×10^7	101	321	211	117

表 2.5　向量按值删除算法的时间

如 表 2.5 所示，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），并且一些情况下消耗的时间会远多于三次移动和三次反转。这是算法的局部性导致的。三次反转虽然不总是速度最快的算法，但对于各种情况的 k 性能相当稳定。

第 2.9 节 本章习题

在 第 2.4 节 中：

1. 中等 假设我们预先分配了大小为N的内存空间给 m_data，并且不允许重新申请内存空间，那么此时向量的扩容上限为N。在这种场景下应该如何设计 reserve 函数进行扩容？
2. 较难 假设我们预先分配了大小为N的内存空间作为内存池，在扩容的时候，m_data 不从操作系统申请新的内存，而是从内存池中未被占用的空间里申请新的内存。开始时，m_data 指向内存池的起始地址，m_capacity 为 0，随着向量的规模增大不断扩容。在这个场景下讨论等比扩容q的选择问题。
3. 中等 自动扩容可以在装填因子达到 1 并且还需要增加规模时进行；自动缩容则应该在装填因子小于某个阈值θ的情况下进行。采用分摊分析的方法，分析等差缩容和等比缩容的时间复杂度。
4. 较难 同时采用公比为q_1的等比扩容和公比为q_2的等比缩容，分析这种策略的最坏分摊时间复杂度。
5. 较难 一个k位的二进制计数器 [15] 使用 bit 的向量存储，计数器的初值为 0。假设每次操作是将计数器加 1（当计数器的值达到2^k时，计数器的值被重置为 0），在这种情况下，计数器的分摊复杂度是多少？假设每次操作是将计数器加 1 或减 1（当计数器的值达到2^k或小于 0 时，计数器的值被重置为 0 或2^k），在这种情况下，计数器的分摊复杂度又是多少？

在 第 2.5.1 节 中：

1. 简单 为什么插入中采用了 std::move_backward，而删除中采用了 std::move？
2. 中等 在示例代码中使用了 第 2.4.4 节 中定义的扩容策略来实施扩容，那么扩容的时候，需要先申请一片内存，将原来的数据复制到新的内存中；随后再将 V[r:n]向后移动 1 个单位。因此在示例代码中，V[r:n]被移动了两次。请设计一个插入时的自动扩容方法，使得 V[r:n]只需要移动一次。

在 第 2.5.2 节 中：

1. 中等 如果分摊分析中的一系列操作都是无后效的，那么分摊复杂度是否和平均复杂度相同？
2. 简单 对一个等比（$q = 2$）扩容的空向量连续插入一系列元素，每次被插入的元素插入在每个位置是等可能的，在此条件下计算插入的平均分摊时间复杂度。

3. 较难 对一个等比（$q=2$）扩容的空向量连续插入一系列元素，每次被插入的元素必定插入在上一次插入位置的前面或后面（前面的概率为p，后面的概率为q，$p+q=1$），在此条件下计算插入的平均分摊时间复杂度。

在 第 2.5.3 节 中：

1. 简单 如果向量中存在多个等于 e 的元素，那么示例代码中会返回第一个等于 e 的元素的位置。请设计一个查找方法，返回最后一个等于 e 的元素的位置。并分析该算法的时间复杂度。
2. 简单 网络流量通常满足 Zipf 分布。当使用向量存储路由表时，如果将热门站点放在路由表的前面，就可以加速路由查找。假设 e 在向量中的位置服从 Zipf 分布，即$f(x)=\frac{x^{-\alpha}}{\sum_{k=1}^{n}k^{-\alpha}}$，讨论查找成功时的平均时间复杂度。

在 第 2.5.6 节 中：

1. 简单 分析一次性移动做法下合并向量的时间复杂度，并在 r 取等可能的情况下计算平均时间复杂度。
2. 中等 向量合并的逆操作是向量拆分。如果希望将向量中的某一段 V[r:r+m]拆分出来成为一个新的向量，其他部分保持不变，设计一个拆分的算法，并分析时间复杂度。如果不需要拆分，只是将 V[r:r+m]从原向量中删除，那么时间复杂度会有什么变化？
3. 中等 如果内存没有被预先分配，则向量合并还会涉及一次扩容。类似 第 2.5.1 节 的第 2 题，设计一个合并时的自动扩容方法，使得 V[r:n]只需要移动一次。

在 第 2.5.7 节 中：

1. 简单 将删除每一个等于 e 的元素改为删除第一个（或最后一个）等于 e 的元素。设计算法并分析时间复杂度。
2. 中等 设计一个算法，在向量中查找一个连续的子向量，使得子向量的和最大。分析算法的时间复杂度。
3. 中等 设计一个算法，在向量中查找一个连续的子向量，使得子向量的和是大于某个给定值 k 的所有和中最小的那个。分析算法的时间复杂度。
4. 中等 设计一个算法，在向量中查找一个数对(a, b)，使得 a+b 是小于某个给定值 k 的所有和中最大的那个。分析算法的时间复杂度。
5. 中等 设计一个算法，在向量中查找一个数对(a, b, c)，使得 a+b+c 等于给定的值。分析算法的时间复杂度。
6. 中等 设计一个算法，在向量中查找一个数对(a, b)，使得 b-a 的绝对值最大。分析算法的时间复杂度。
7. 中等 设计一个算法，在向量中统计满足 b-a 的绝对值等于给定值的数对(a, b)的个数。分析算法的时间复杂度。

在 第 2.6.3 节 中：

1. 简单 在归并排序中，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），时间复杂度会变成多少？如果左半部分取作$\max\left(\frac{n}{2}, C\right)$，其中 C 是一个给定的常数，那么时间复杂度又会变成多少？
2. 中等 设计一个算法，对 k 个有序序列（k>2）进行归并。
3. 中等 设计一个算法，对两个严格有序的序列进行归并，但两个序列中的重复项只保留一个。
4. 较难 使用 merge 函数合并两个有序序列，其长度分别为 m 和 n。该合并的结果共有C_{m+n}^{m}种（相当于 m 个 0 和 n 个 1 组成的 0-1 串数量），假设发生每种合并结果是等可能的，那么合并过程中 cmp 平均被调用多少次？
5. 较难 记最坏情况下合并两个长度为 m 和 n 的有序序列的最小比较次数是$m@n$，证明：当$|m-n| \leq 1$时，$m@n = m+n-1$。
6. 中等 证明：$(m_1+m_2)@n \leq m_1@n + m_2@n$。
7. 较难 证明：$m@n \leq m@\left\lfloor\frac{n}{2}\right\rfloor + m$。

在 第 2.6.4 节 中：

1. 简单 证明自下而上的归并排序中，每一趟的时间复杂度为$\Theta(n)$。
2. 中等 使用递归的技术，将正文中的自下而上的归并排序的实现改为自上而下，使其调用 merge 的次序和正文中的自下而上的实现相同。

在 第 2.6.8 节 中：

1. 中等 证明确定L的示例代码正确性，并分析此法实现归并排序的时间复杂度。
2. 较难 用本节讨论的两种策略改写自下而上的归并排序，使得在基本有序的情况下时间复杂度更低。分析改进后的时间复杂度。

在 第 2.6.9 节 中：

1. 简单 为什么经典归并中，在归并的最后只需要将前半部分多余的元素移动回原向量，而在原地归并中，后半部分多余的元素也需要处理？
2. 简单 证明原地归并排序的时间复杂度为$\Theta(n \log n)$。
3. 简单 举例说明：在原地归并的过程中，用作辅助空间的区域 L 中的元素有可能被直接交换到左侧，也有可能先被交换到右侧若干次再被交换到左侧。
4. 中等 为什么说普通的归并排序是稳定的？

在 第 2.7.1 节 中：

1. 简单 证明折半查找的时间复杂度为$\Theta(\log n)$。
2. 简单 假设待查找元素在向量中的位置服从均匀分布，分析顺序查找和折半查找的平均时间复杂度。
3. 中等 假设待查找元素在向量中的位置服从几何分布，分析顺序查找和折半查找的平均时间复杂度。什么情况下，顺序查找的性能会超过折半查找？

4. 较难 假设一个$m \times n$的矩阵的每行和每列都是严格有序的。给定一个元素 x，设计算法在矩阵中查找 x，并分析时间复杂度。
5. 挑战 假设一个$m \times n$的矩阵的每行和每列都是严格有序的。给定一个元素 x，证明：为了判定 x 是否存在于该矩阵中，最坏情况下最少需要$m@n$（定义见 第 2.6.3 节 的第 5 题）次比较。

在 第 2.7.4 节 中：

1. 简单 什么情况下，“删除-擦除”法会导致$\Theta(n^2)$的时间复杂度？
2. 简单 分析采用 第 2.5.7 节 中的其他按值删除算法替换“删除-擦除”法的条件下，唯一化的时间复杂度。
3. 中等 比较“删除-擦除”法和逐个删除法的性能差异。
4. 较难 利用信息熵方法证明，唯一化的时间复杂度有下界$\Omega(n \log n)$。

第 2.10 节 本章小结

向量（顺序表）是程序设计中最经常使用的数据结构，因此本章具有较大的容量。向量的顺序结构是简单的，向量的循秩访问特性是自然的，无论是学习还是考试，这一章的重点都在算法设计上。本章通过较多的实验展示了一些算法设计的典型技巧。其中，对于排序和查找两个重点内容，我们还将在后续的算法章节再次讨论。

本章的主要学习目标如下：

1. 您有了对算法设计中的不必要工作的优化意识。
2. 您学会了对顺序表的整块进行操作，而不是对单个元素进行操作。
3. 您学会了使用快慢指针进行探测和更新。
4. 您学会了利用分支的不对称性改变划分方式以优化分治算法。
5. 您了解到可以从信息的观点分析时间复杂度问题。
6. 您了解到排序预处理可以为后续问题的解决提供帮助。
7. 您了解了分析分摊复杂度和平均复杂度的意义。
8. 您深化了对递归-迭代关系的认识，掌握了消除尾递归的方法。

向量本身是很简单的数据结构，记忆它的插入、删除、查找和各种变形，也是相对简单的；重要的地方在于希望您能理解并掌握书中这些典型算法的设计思路、优化思路。希望本书的内容能为您解决基于向量的算法设计问题提供帮助。

第 3 章 列表

本章介绍另一种线性表：**列表**（list）。列表和向量的区别在于，列表不要求在内存中占据的空间是连续的。这一特点赋予了列表更大的灵活性，使列表的一些操作比向量效率更高；但同时，这一特点也让列表无法通过秩定位到内存地址，丧失了循秩访问的能力，从而在另一些操作上效率不如向量。本章将详细介绍列表的特质，请读者在阅读的同时和向量进行对比，以便更好地理解列表的特点。

第 3.1 节 列表的结构

列表不要求在内存中占据的空间是连续的，因此不能循秩访问，只能循位置访问：通过指向列表中某个元素的指针（也就是该元素的地址）来访问它。那么，如何获得一个元素的地址呢？首先，把所有元素的地址汇总在一张表看起来是可行的，但如果这么做的话，这张表本身如何存储就变成了一个新的问题。如果这个表使用向量存储，那么我们就放弃了列表的不连续的灵活性；如果这个表使用列表存储，那么就成为了一个嵌套的问题。所以，我们不能把所有元素的地址汇总在一张表，也就是说，我们不能采用*集中式*的地址存储，需要采用*分布式*的地址存储。

所谓分布式的地址存储，就是我们在每个元素处，同时存储它前一个和后一个元素的地址。这样，我们无论是从前往后还是从后往前，都能遍历整个列表。很明显，这样带来了一个坏处，就是我们只能“逐个”访问元素，而不能像向量那样根据任意的秩访问元素。这就是列表选择更大灵活性的代价。除此之外，列表还有一些其他的代价，比如每个元素处除了它本身的空间，还需要存两个地址，因而有可能需要付出比向量更大的空间（注意这里是有可能，因为向量的装填因子很低时，所造成的的空间浪费更大）。为了降低列表的空间浪费，有时我们会放弃存储前一个元素的地址，只存储后一个元素的地址。这种情况称为**单向列表**（forward list）或单链表，相对应地，同时存储前一个和后一个元素地址的情况称为**双向列表**（bidirectional list）或双链表。本书中如无特别说明，列表均指双向列表。

第 3.1.1 节 单向列表的节点

为了设计列表的抽象类，我们需要首先将列表中的每个元素抽象出来。列表中每个数据元素及其附加属性（即，前后元素的地址）组成了一个数据单元，或者称**节点**（node）。《计算机科学技术名词》[19] 认为“节点”是数据结构中数据元素的连接点或端点，而“结点”是计算机网络中的网络拓扑设备，并注明二者互为别称。本书不做区分。本着从简到繁的精神，我们从单向列表的节点开始。

```
template <typename T>
class ForwardListNode {
    T m_data;
    std::unique_ptr<ForwardListNode<T>> m_next {};
};
```

【C++学习】

在列表中，每个节点只有唯一的直接前驱，因此可以用唯一控制所有权的智能指针 `std::unique_ptr` 来存储。一个节点的所有权归属于它的直接前驱，第一个节点的所有权归属于列表对象。和裸指针或 `std::shared_ptr` 相比，这样做可以做到列表作为线性结构的排他性，即不会出现形如$a \to b \leftarrow c$这样，两个节点指向同一个节点的列表结构。一个极端的例子是，如果采用裸指针，列表甚至有可能形成$a \to b \to c \to d \to b$这种形式。在这个例子中，因为$b \to c \to d \to b$形成了一个环，所以会导致在列表上的访问无限循环。

使用智能指针 `std::unique_ptr` 来持有下一个节点的所有权时，由于下一个节点又持有了再下一个节点的所有权，所以以此类推，一个节点实际上持有了其所有后继节点的所有权。根据智能指针的特性，一个节点被释放时，它所持有的下一个节点也会被释放，以此类推，它所有后继节点都会被释放。那么，只要释放一个列表的第一个节点，整个列表就会被递归地释放。

递归释放（即析构函数声明为 `default`）这个想法看起来很美好，实际上却会出现问题。

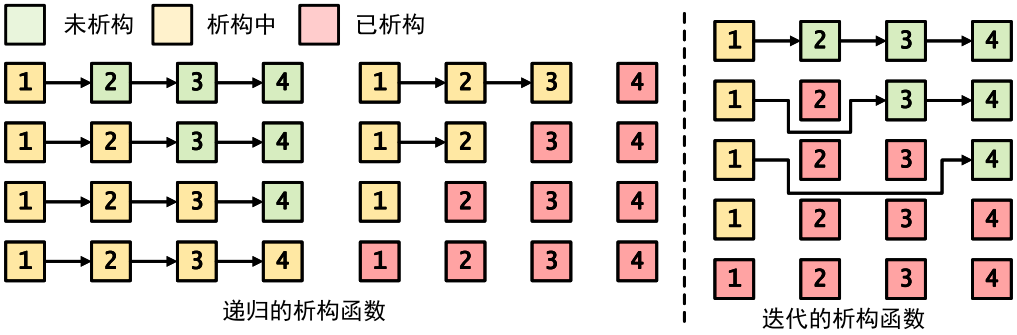

图 3.1　列表节点的递归析构和迭代析构

如果我们清空一个规模为 n 的列表 L，则如 图 3.1 中的左图所示，黄色节点代表析构中的节点，可以看出在最后一个节点被析构的时候，前面所有的节点都在等待。因此，会触发 n 层析构函数的递归，当 n 非常大时，会引发栈溢出（stack overflow）错误。因此，我们需要手动设计列表节点的析构函数，将析构函数的递归改为迭代，而不能依赖于 `std::unique_ptr` 的自动释放空间。下面展示了一

个迭代版本的析构函数，如 图 3.1 的右图所示，它不会造成大量节点在析构中状态等待的情况。

```
virtual ~ForwardListNode() {
    auto p { std::move(m_next) };
    while (p != nullptr) {
        p = std::move(p->m_next);
    }
}
```

【C++学习】

另一方面，对列表节点进行复制或者移动操作是没有意义的。首先，如果复制一个列表节点，那么就需要面临，复制出的列表节点和原节点的后继是否指向同一个节点的问题。然而因为 std::unique_ptr 的独占性，指向同一个节点是不可能的，而如果递归地复制所有后继，那么这样的操作等同于复制列表而不是复制列表的节点。其次，如果移动一个列表节点，似乎原节点的后继的所有权可以被移交给新的节点，然而，原节点的前驱并不知道这个移动操作，它持有的仍然是移动前的节点的所有权，从而造成列表被切断（断链）。退一步说，即使我们通过某种方式，让一个新节点替换了原节点在其所在列表中的位置，这样做的意义也不大，因为完全可以直接移动数据字段 m_data，不需要移动节点。

从上面的分析可以看出，我们不需要为列表节点提供复制和移动操作。因此，我们可以将复制和移动操作声明为删除函数，以防止用户误操作。基于 第 2.4.1 节 所介绍的“5 原则”，我们需要声明以下 5 个函数（默认构造函数不再可以省略）。

```
ForwardListNode() = default;
ForwardListNode(const ForwardListNode&) = delete;
ForwardListNode& operator=(const ForwardListNode&) = delete;
ForwardListNode(ForwardListNode&&) = delete;
ForwardListNode& operator=(ForwardListNode&&) = delete;
```

第 3.1.2 节　双向列表的节点

【C++学习】

当我们试图扩展单向列表到双向列表的时候，我们有两种选择。

1. 让前向指针和后向指针共享节点的所有权，也就是两个指针都声明为 std::shared_ptr。这种做法过于浪费空间，因为 std::shared_ptr 除了指针本身之外，还包括一个引用计数。
2. 仍然保留单向列表中，只有后向指针拥有节点的所有权的设计，即后向指针仍然是 std::unique_ptr，而前向指针则使用不涉及所有权的裸指针。这种做法节约了空间，但会破坏前向指针和后向指针的对称性，提高算法设计难度。

在本书中，采用后向指针单独拥有所有权，而前向指针使用裸指针的设计，因为对于基础数据结构而言，增加两个引用计数的成本是难以接受的。需要注意的是，前向指针的裸指针永远不会被释放。作为一个良好的编程习惯，应当避免在使用智能指针的工程中使用 `new` 或者 `delete` 关键字操作裸指针，应当始终注意裸指针是一个表示无所有权的指针。

当然，即使用户保证自己不会操作裸指针，这种做法仍然是存在安全隐患的。用户可能将裸指针传递给了其他对象，随后通过智能指针将节点释放。这样，裸指针就变成了悬空指针，并且没有办法检测到。因此，我们应当在接口上尽可能使用迭代器操作列表，将裸指针操作封装在类内部，避免用户将裸指针传递给外部的其他对象。

```
template <typename T>
class ListNode {
    T m_data;
    ListNode<T>* m_prev {};
    std::unique_ptr<ListNode<T>> m_next {};
};
```

第 3.1.3 节 哨兵节点

在单向列表中，我们只能从前向后访问节点，所以，我们只需要知道指向第一个节点的指针，就可以从前向后依次访问所有节点。而在双向列表中，我们可以从两个方向访问节点，因此我们需要知道指向第一个节点和最后一个节点的指针，才可以从两个方向依次访问所有节点。

在 [4] 中，每个列表存在两个不存储实际数据的虚拟节点，称为**头哨兵节点**和**尾哨兵节点**，或者简称为**头节点**（head）和**尾节点**（tail），用来标志列表的开始和结束。即使列表是空列表，这两个节点也存在。哨兵节点的引入使得许多实现得到了简化（simplify）和统一化（unify），比如前插（在一个节点之前插入新元素）不需要对第一节点进行特殊判定。在 [6] 中介绍的列表则是没有哨兵节点的，转而使用指向第一个节点和最后一个节点的裸指针来对头尾进行定位，称为**头指针**和**尾指针**。这不会影响到列表的功能，节约了 2 个节点的空间，但有些功能的实现会略显复杂。本章采用 [4] 的设计，使用哨兵节点。

```
template <typename T>
class AbstractList : public LinearList<T, /* iterators */> {
protected:
    virtual ListNode<T>* head() = 0;
    virtual ListNode<T>* tail() = 0;
public:
    virtual iterator insertAsNext(iterator p, const T& e) = 0;
```

```
    virtual iterator insertAsPrev(iterator p, const T& e) = 0;
};
```

在列表类中，我们将插入操作区分为了前插（插入为 p 的直接前驱）和后插（插入为 p 的直接后继）。因为后向指针和前向指针不是对称的，所以前插和后插也有一些区别。

【C++学习】

单向列表的抽象类是相似的。注意，在基类 LinearList 中存在一些方法是单向列表无法实现的，比如，我们使用下面的方法来实现 back（取最后一个元素）：

```
virtual T& back() { return *--end(); }
```

由于单向列表节点中没有储存指向直接前驱的指针，单向列表的迭代器是无法实现--操作的。因此，我们无法实现上述的 back 方法。对这样的问题，有两种处理模式：

1. 重写 back 方法，让单向列表中第一个元素开始遍历，直到最后一个元素。这样的实现会使得 back 方法的时间复杂度变为$\Theta(n)$，而不是$O(1)$。但是，用户并不一定能够敏锐地发现这个问题，用户可能仍然以为这个方法是$O(1)$的。一旦这样自以为是的用户在自己的程序中大量使用了 back 方法，其程序的性能将会受到严重影响。这一问题是有先例的，在 C++11 强制规定前，STL 的 std::list::size()就是$\Theta(n)$的，而许多用户并不知情，导致他们的程序有很大一部分时间被浪费在了无意义的遍历求列表规模上。
2. 隐藏 back 方法。通过将 back 方法重载为一个 private 方法，防止用户调用。这样，用户在使用 back 方法时，会得到一个编译错误，从而引起他们的注意。如果用户真的想要获取最后一个元素，他们可以通过*(begin() + (size() - 1))等方式进行。这样的实现虽然不够优雅，但是可以避免用户的误用。

本书的示例代码采用了隐藏的处理方式。由于 back 方法会返回一个 T&，而我们不希望使用这个方法，它没有有意义的返回值，所以可以用下面的方式抛出一个异常而不返回值：

```
[[noreturn]] T& back() override {
    throw std::logic_error("/* error message */");
}
```

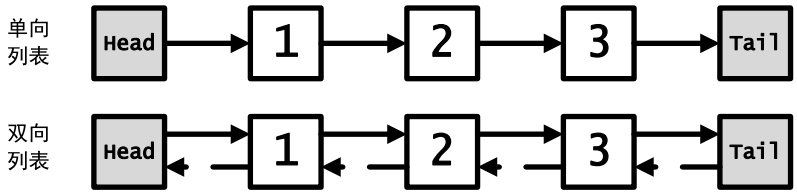

图 3.2 单向列表和双向列表

如 图 3.2 所示，我们在单向列表和双向列表的两头各添加一个哨兵节点，代表所有权的智能指针用实线箭头表示，不涉及所有权的裸指针则用虚线箭头表示。由于哨兵节点的存在，用户总是可以知道列表的头和尾的位置。因此，即使在单向列表中，用户无法直接获得尾部（back）元素中的数据（需要从头节点向后依次遍历来寻找尾部），也无法直接删除尾部（pop_back）的元素，但用户可以直接在尾部之后插入（push_back）。如果没有尾节点的存在，则我们无法直接进行 push_back 操作。

【C++学习】

在 C++的标准库中，std::forward_list 就是这样设计的：它包含一个最简单、最节约空间的抽象，没有尾节点也不支持 push_back。

需要特别指出，哨兵节点仅仅用于列表的实现方法，而非列表的组成部分，因此在 图 3.2 我们将它们用灰色表示。当我们具体讨论到某个列表 L[0]→L[1] → ... →L[n-1]的时候，第一个节点 L[0]和最后一个节点 L[n-1]都是实际存储元素的节点，而非头节点或尾节点。在列表中，L[0]没有前驱；而在列表的具体实现中，我们可以认为头节点是它的直接前驱以方便设计代码。

第 3.2 节 插入、查找和删除

第 3.2.1 节 后插一个元素

我们首先讨论后插。给定被插节点的位置 p 和待插入元素 e，将元素 e 插入到列表中，使其对应的节点成为 p 的直接后继，这种类型的插入称为后插。假设 p 原有的直接后继是 q，则二者的指针连接形成$p \to q$的形式。我们希望将 e 插入到二者之间，形成$p \to e \to q$的形式。这里可以看出引入哨兵的好处。在引入哨兵节点之后，对于任何一个被插节点，它总是有直接后继 q；当被插节点是最后一个节点时，q 就是尾哨兵节点，而不需要进行特判。

从宏观层面上来说，对于双向列表，我们需要对 p 的后向指针、q 的前向指针以及新生成的 e 的双向指针进行赋值，然而，这四个指针的赋值顺序需要仔细斟酌。这是设计列表算法的易错点，因为“拆散-重组”的过程不是一次性发生的，必定存在先后顺序。允许的顺序有许多种，但如果顺序错误，就有可能在拆散的过程中丢失了信息，导致无法进行重组。在设计完列表节点算法之后，应当检查“拆散-重组”过程是否有遗漏，是否有重复，是否有错误。

比如说，如果我们直接断开 p 到 q 的后向指针，将 e 接入，形成$p \to e \mid q$的形式，那么 q 会因为失去所有权而被智能指针自动回收（这种情况可以被生动地形容为“断链”）。因此，我们可以通过 std::swap 交换智能指针的所有权，然后再将 q 连接到 e 的后向指针处。建立好$p \to e \to q$之后，再将前向指针补上。

```
iterator insertAsNext(iterator p, const T& e) override {
    auto node { std::make_unique<ListNode<T>>(e) };
    auto& q { p.node()->next() };
    std::swap(q, node);
    q->next() = std::move(node);
    q->prev() = p.node();
    q->next()->prev() = q.get();
    ++m_size;
    return ++p;
}
```

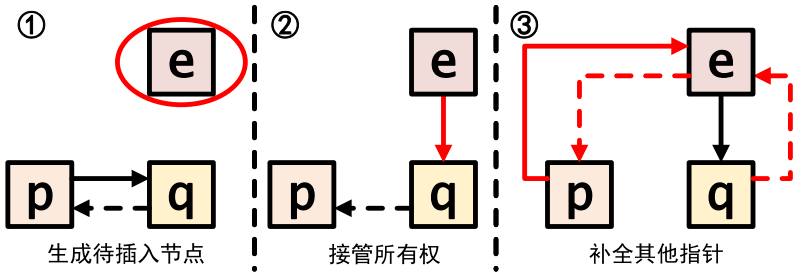

图 3.3 双向列表的后插

因为使用了交换语义，上面的这种做法产生了比预期更多的赋值次数。我们看到在上面的算法中，对于 node 的重复赋值是不到位工作，它经历了 e 变化为 q、再变化为空的过程，而实际上可以让它一步到位地变化为空。于是，我们不应该从$p \to e \mid q$开始，可以从另一个后向指针，也就是$p \mid e \to q$开始。这样就能减少一次赋值。

从 图 3.3 中可以看出，在后插的过程中，应当首先让 e 的后向指针接管 q 的所有权；而其他三项的赋值次序是可以随意的。当然，尽管次序可以随意，但在书写的时候仍然需要当心。比如，在 e 的后向指针接管 q 的所有权之后，我们不再能通过 p 的后向指针来找到 q，只能通过 e 的后向指针来找到它。这些细小的注意点有时熟练的程序员也会忽略，因此借助图形和手工演算进行检查是有意义的。

```
iterator insertAsNext(iterator p, const T& e) override {
    auto node { std::make_unique<ListNode<T>>(e) };
    node->next() = std::move(p.node()->next());
    node->next()->prev() = node.get();
    node->prev() = p.node();
    p->next() = std::move(node);
    ++m_size;
    return ++p;
}
```

单向列表的情况大同小异，只是减少了对前向指针赋值的操作。可以看出，在列表上做后插的时间复杂度和空间复杂度均为为$O(1)$。

第 3.2.2 节　前插一个元素

对于双向节点，前插的过程和后插大同小异。因为头哨兵节点的存在，我们总是可以找到 p 的直接前驱 q，于是我们只需要在$q \to p$之间插入 e，和前面的$p \to q$只有字母上的区别。当然，也可以直接用后插实现前插。

```
iterator insertAsPrev(iterator p, const T& e) override {
    return insertAsNext(--p, e);
}
```

对于单向列表，情况变得有些令人沮丧。因为单向列表中，我们无法使用前向指针，也就无法定位到 p 的直接前驱。那么，我们是否无法进行前插呢？是，也不是。我们可以用两种方法实现前插：

1. 从头节点开始，沿着后向指针访问整个列表，寻找 p 的直接前驱，然后再采用和上面一样的方式进行前插。显而易见，如果 p 在列表中的秩是 r，则该方法的时间复杂度高达线性的$\Theta(r)$。我们放弃循秩访问而使用列表，为的是允许操作不连续内存，从而在插入和删除的时候达到$O(1)$的常数复杂度；因此，这种方法虽然万无一失，但和我们的初衷背道而驰。此外如 第 3.1.3 节 中所述，对外接口的时间复杂度也会带来隐患。
2. 使用后插代替前插。我们首先找到 p 的直接后继 q 进行一次后插，形成$p \to e \to q$的形式，然后我们交换 p 和 e 节点的值，形成$e \to p \to q$，如 图 3.4 所示。这样在值上实现了前插，但是在位置上并没有实现前插。比如，如果原先外部有一个迭代器指向 p，我们执行前插之后，这个迭代器并不知道 p 已经被 e“偷梁换柱”了，它可能还以为自己指向的是 p，从而产生不可估计的错误。因此，这种方法事实上损失了安全性。但是，毕竟它是一个$O(1)$的方法，我们也只能接受它，并期待用户正确地使用这个方法。

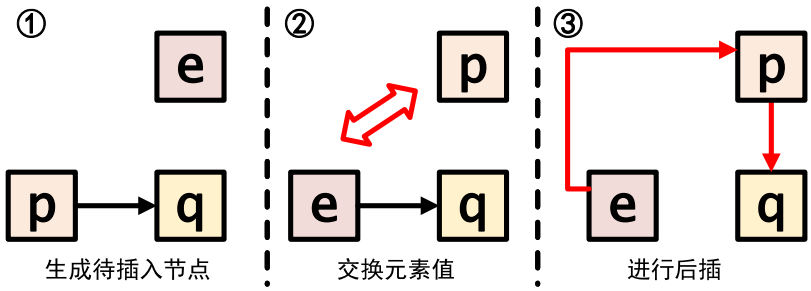

图 3.4　单向列表的前插

【C++学习】

很显然，两种前插都存在一些问题，因此 `std::forward_list` 就不允许进行前插。在现实中设计程序时，应当避免这种高风险的设计。

回到使用后插代替前插的技术上。和 第 3.2.1 节 一样，我们注意到采用交换语义会引入不到位的赋值。在消除交换语义之后，一个可能的实现如下。

```
iterator insertAsPrev(iterator p, const T& e) override {
    auto node { std::make_unique<ForwardListNode<T>>(std::move(*p)) };
    *p = e;
    node->next() = std::move(p.node()->next());
    if (p == end()) {
        m_tail = node.get();
    }
    p.node()->next() = std::move(node);
    ++m_size;
    return p;
}
```

前面讨论过，如果原先外部有一个指针指向 `p`，我们执行前插之后，这个指针并不知道 `p` 已经被替换了。我们无法保证用户使用的“外部指针”的安全性，但我们需要保证数据结构内部的指针的安全性。这个可能丧失安全性的指针就是尾节点指针 `m_tail` 当我们要在尾节点处进行前插（也就是 `push_back`）的时候，上述方法事实上进行了后插，并将原先的尾节点赋值为 `e`。在这种情况下，我们需要将尾节点指针指向新的尾节点处。

另外，请注意这里必须对 `p` 中的数据使用移动语义，而不能使用复制语义，否则当 `p` 中的数据非常大（比如，一个被嵌套在列表里的向量）时，会引入大量的（非常数的）性能损失。

第 3.2.3 节　查找一个元素

无论是单向列表还是双向列表，查找元素的过程和无序向量都是一样的，只需要从前向后一个一个查找即可。因为不支持循秩访问的缘故，即使是有序列表，也不支持折半查找。下面是一个示例的实现。

```
iterator find(const T& e) const override {
    auto p { begin() };
    while (p != end() && *p != e) {
        ++p;
    }
    return p;
}
```

第 3.2.4 节　删除一个元素

列表删除是插入的逆操作。对于双向列表，假设被删除的节点为 p，由于头节点和尾节点的存在，我们可以保证它存在直接前驱 q 和直接后继 r。因此，我们需要将$q \to p \to r$变形为$q \to r$。显然，我们只需要对 q 的后向指针和 r 的前向指针各进行一次赋值。顺序仍然是重要的，但难度比插入降低了不少（因为 2 次赋值的顺序一共只有 2 种）。下面是一个可行的实现。

```
iterator erase(iterator p) override {
    auto node { p.node() }, q { p + 1 };
    node->next()->prev() = node->prev();
    node->prev()->next() = std::move(node->next());
    --m_size;
    return q;
}
```

类似地，上面的做法也不适用于单向列表的删除。对于单向列表，我们需要采用和前插相似的技术，首先找到 p 的直接后继 r，以及 r 的后继 s。我们在$p \to r \to s$上交换 p 和 r 节点上的值，从而在值的角度，变成了$r \to p \to s$，然后重新赋值 r 的后向指针，变为$r \to s$即可。双向列表和单向列表在删除节点上的区别如图 3.5 所示，请注意在双向列表的场合，当移交所有权之后，节点 p 会被智能指针自动释放（用灰色表示）。

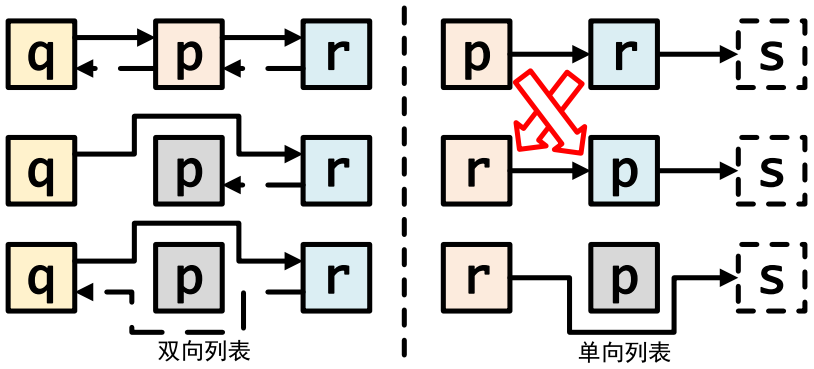

图 3.5　在列表中删除一个元素

图 3.5 中，s 是虚线节点，这是因为即使在存在尾节点的列表中，s 仍然可能为空（nullptr）。如果 s 为空，那么 r 就是尾节点（在有哨兵的场合），交换元素之后我们会把原先的尾节点删除。所以，我们需要进行一次特判，在上述情况下让新的 r 成为新的尾节点。在消除了交换语义带来的不到位赋值之后，得到的一种示例程序如下所示。

```
iterator erase(iterator p) override {
    auto q { p + 1 };
    *p = std::move(*q);
    p.node()->next() = std::move(q.node()->next());
    if (q == end()) {
        m_tail = p.node();
    }
    --m_size;
    return p;
}
```

和 第 3.2.2 节 一样，这里的尾节点指针 `m_tail` 也是需要维护的“外部的指向节点的指针”。我们需要对这个指针重新赋值以避免安全性问题。

【C++学习】

同样地，在 STL 的 `std::forward_list` 中，为了安全性考虑，只提供“删除后继”的方法 `erase_after` 而不提供直接删除的方法。

从上面对插入、查找和删除的分析中可以看出，列表的三种基本操作完全不涉及规模。所以，如果不需要在$O(1)$时间里获取规模，那么列表是可以不记录这个属性的。不记录规模可以使列表在插入和删除时减少一次赋值，从而略微提高列表的性能。至此，列表的基本操作已经讨论完毕，可以在 **`list.cpp`** 和 **`flist.cpp`** 中分别对双向列表和单向列表进行测试。

第 3.2.5 节　在头部和尾部连续插入元素

实验 **`lins.cpp`**。随着三种基本操作分析完毕，我们可以开始对列表展开实验。这一节讨论在列表的头部和尾部连续插入元素的情况。

在示例代码中，采用示例实现的向量、双向列表和单向列表类作为实验对象。我们考虑两种环境。第一种是连续在头部插入元素，也就是每次插入的元素会成为线性表的第一个元素；另一种是连续在尾部插入元素，也就是每次插入的元素会成为线性表的最后一个元素。

这个实验表明，当在尾部连续插入元素时，向量的效率显著高于列表，但是单向列表并没有和双向列表之间有很大的差异。根据您的运行环境不同，向量和列表大致呈现一个数倍的性能差距，总体来说是常数的。需要特别指出的是，尽管在尾部连续插入元素的测试中向量的性能高于列表，但对于单次插入元素的过程则不一定。触发扩容时，向量单次插入元素的时间复杂度可达$\Theta(n)$，而列表是稳定的$O(1)$。这一点是列表的独特优势，在对响应时间稳定性有要求的场合下，

向量的扩容变得不能接受，而列表的插入时间是可以保证很低的。

【C++学习】

当在头部插入元素时，可以显著地看到，向量$\Theta(n^2)$的时间复杂度远高于列表$\Theta(n)$的时间复杂度，当$n = 10^6$的时候向量的情况已经难以接受；我们可以直接做一个判断，在这种情况下抛出一个异常拒绝执行，如下所示。

```
if constexpr (std::is_base_of_v<Vector<std::size_t>, L<std::size_t>>) {
    if (n >= 1000'000) {
        throw std::runtime_error { "/* error message */" };
    }
}
```

因为向量的耗时增长很快，我们在$n = 10^5$量级已经可以看出它的性能，因此不需要花费大量的时间等待它在更大的规模上完成实验。这里采用了 C++17 引入的 `if constexpr` 语法，从而实现在编译期判断。上面的代码对于以向量为参数的模板实例，会将 `if constexpr` 内的判断编译进去；而对于以列表为参数的模板实例，则不会将 `if constexpr` 内的判断编译进去；这个向量和列表的区分是编译期完成的，运行期不会再做判断。这一语法简洁清晰，可以用来替代 C 语言风格很容易被滥用的 `#ifdef` 等条件编译命令。

插入位置	n	向量	双向列表	单向列表
尾部	10^4	0	0	0
尾部	10^5	1	2	3
尾部	10^6	9	32	27
头部	10^4	26	0	0
头部	10^5	2774	2	2
头部	10^6	非常大	31	27

表 3.1　不同数据结构连续插入元素的耗时对比

如 表 3.1 所示，双向列表和单向列表在时间上的性能差异实际上非常小。双向列表虽然节点多了一个字段，但消耗的时间和单向列表相比，只有非常有限的增加，与此同时，它获得了从后向前访问的能力、更强的灵活性以及更好的安全性（关于安全性，在 第 3.2.2 节 和 第 3.2.4 节 已经讨论过）。因此，除非性能要求苛刻或空间不足，否则都建议使用双向列表。

第 3.2.6 节　顺序删除和随机删除

当我们连续地在列表中插入节点时，这些节点也会存在一定程度上的空间连续性，因为它们的空间是操作系统连续分配的。这会导致我们的实验并不能真实地反映出现实情况下的列表所包含的节点地址弱局部性这一特征。在这个实验中

我们将会认识到，如果列表所包含的节点地址局部性很差，会对列表的性能造成很大的负面影响。

实验 lrm.cpp。我们考虑下面的连续删除场景。

```
template <typename T>
class ContinuousPop : public Algorithm<void()> {
protected:
    List<T> L;
    Vector<typename List<T>::iterator> V;
public:
    virtual void initialize(std::size_t n) = 0;
    void operator()() override {
        for (auto p : V) {
            L.erase(p);
        }
    }
};
```

在这个框架中，我们使用向量 V 来存储列表 L 的每一个节点的位置。通过改变初始化函数，我们让向量 V 成为顺序的或者随机的，然后，按照向量 V 中存储的节点次序依次删除列表中的元素。对于顺序存储的情况，每次向 L 加入元素后储存在 V 中即可。对于随机存储的情况，可以在顺序存储之后增加一个 `std::shuffle`（见 第 2.6.1 节 ）。

n	顺序删除	随机删除
10^5	2	5
10^6	22	96
10^7	246	1561

表 3.2　顺序删除和随机删除的耗时对比

如 表 3.2 所示，从这个例子中可以看到，当 n 比较小的时候，顺序删除和随机删除的性能基本一致；但当 n 比较大的时候，二者就会产生一个明显的差异。这个性能差异就是随机删除的局部性丧失引起的。

在现实中，我们生成的列表可能不是以连续插入的形式分配内存的。因此，它可能天然就具有一个很差的局部性。这就意味着，现实中的列表在较坏的情况下有可能会更接近于本实验中随机访问的性能。结合 第 3.2.5 节 的实验结果，我们意识到，列表和向量之间有着巨大的性能差距，甚至在数据结构规模较小的情况下，列表$O(1)$插入、删除的优势体现不出来，从时间消耗的角度上来说还不如$\Theta(n)$的向量操作。因此对于小型线性表，无论我们是否需要循秩访问、折半查找等向量特性，都应该优先使用向量而不是列表。

第 3.2.7 节　倒置列表

当在向量上设计算法的时候，由于向量循秩访问的特性，我们可以对秩进行运算，从而快速、精准地找到某个元素。一个典型的例子就是折半查找。循秩访问使得向量中的元素地位比较“平等”；而在列表的场合，循位置访问使得列表中的元素地位比较“不平等”，比较“任人唯亲”。比如，从头节点出发访问列表中的一个元素时，所需要消耗的时间会和被访问元素的位置相关；越接近头部的元素，访问需要的时间越短。

我们可以站在更高的层次，理解 列表只能直接访问首尾两端、以及循位置访问“任人唯亲”的特性：列表是**线性递归定义**的。具体来说，列表可以被这样定义：

1. ∅是列表（空列表，没有元素）。
2. 设 x 是一个节点，L 是一个列表，则 x→L 是列表（x 的后向指针指向 L 的第一个节点）。

这个定义具有和 第 1.4 节 所述递降法相似的形式，并且它意味着列表适合使用**减治**思想设计算法。首先，我们研究递归边界（空列表）的情况；其次，对于规模为 n 的列表，我们将第一个节点提取出来，递归地研究后 n-1 个节点组成的子列表，再和第一个节点进行合并。对称地，我们可以想到列表具有另一个定义：

1. ∅是列表。
2. 设 x 是一个节点，L 是一个列表，则 L→x 是列表。

这一定义同样可以引出一种减治算法的设计。它和前一种定义的区别在于，平凡项 x 在减治项 L 的头部还是尾部。对于双向列表来说，头部减治和尾部减治是等价的，提取出平凡项 x 都只需要$O(1)$的时间。当然，我们需要注意，减治递归算法应用在列表上时，递归深度高达$\Theta(n)$，这不但导致了较高的空间复杂度，同时在 n 较大时很可能引发栈溢出错误；因此在设计完成减治算法之后，应当尽可能将其修改为迭代算法。

单向列表的头部和尾部不具有对称性，因此情况会有所不同。考虑单向列表 L[0]→L[1]→L[2]→ ... →L[n-1]。当在头部进行减治时，只需要$O(1)$的时间就可以找到 L[0]；而在尾部进行减治时，需要$\Theta(n)$的时间才能找到 L[n-1]。看起来单向列表应当总是在头部进行减治，然而事实却不一定是如此。

实验 **lreverse.cpp**。本节将以倒置列表为例子，对两种减治方法进行比较。首先，对于双向列表的倒置，我们可以直接从头尾两个方向遍历，交换对称的两个节点的数据域。这种做法和向量上的做法一致，也可以使用 std::reverse。

```
void operator()(List<T>& L) override {
    auto l { L.begin() }, r { --L.end() };
    for (auto n { L.size() }; n >= 2; n -= 2) {
        std::iter_swap(l++, r--);
```

```
    }
}
```

下面讨论单向列表（为了方便比较仍然使用 `List<T>`作为参数，但不允许使用前向指针），首先考虑在头部进行减治的情况。按照减治的思想，在将 `L[0]`摘出列表之后，把从 `L[1]`开始的剩余列表反转，然后再进行合并：把被摘出的 `L[0]`接到反转后的剩余列表的尾部，如 图 3.6 所示。

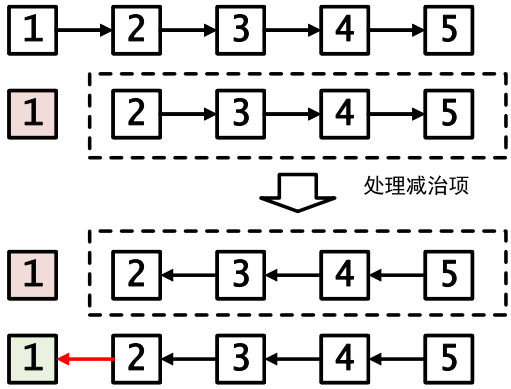

图 3.6 使用头部减治法倒置单向列表

```
void reverse(List<T>& L) {
    if (L.size() <= 1) {
        return;
    }
    auto first { std::move(*L.begin()) };
    L.pop_front();
    reverse(L);
    L.push_back(first);
}
```

它的时间复杂度和空间复杂度均为$\Theta(n)$。这个算法并不是尾递归，这意味着我们将它改写成迭代的时候会遇到困难。这一困难来自于我们在做递归调用的时候，需要保存当前递归实例摘出的元素 `first`。当我们进行到最后一层递归调用的时候，需要保存之前每一次摘出的元素，这就意味着不可避免地需要$\Theta(n)$的空间。这也就意味着，将此方法改写为迭代之后，几乎和下面的“极端”做法等价。

```
void operator()(List<T>& L) override {
    Vector<T> V(L.size());
    std::move(L.begin(), L.end(), V.begin());
    std::reverse(V.begin(), V.end());
    std::move(V.begin(), V.end(), L.begin());
}
```

这个做法的时间复杂度和空间复杂度同样都是$\Theta(n)$。它将所有元素移动到一个辅助向量里，在向量里面倒置，再移动回列表里。我们很难接受这样的算法。从刚才的分析中已经可以发现，影响空间效率的主要原因是，单向列表只有后向指针。当在头部进行减治的时候，被提取出来的`L[0]`彻底和`L[1:n]`断开，无法通过`L[1:n]`上的指针找到`L[0]`，从而必须要额外的空间来存储它。于是就可以想到，是否可以利用单向列表的不对称性，采用在尾部进行减治的策略设计算法？答案是肯定的。当在尾部进行减治的时候，我们可以先处理`L[0:n-1]`，然后利用`L[n-1]`的后向指针直接找到`L[n]`。下面展示了一个示例代码（已隐藏前向指针的相关赋值操作），其原理如 图 3.7 所示。

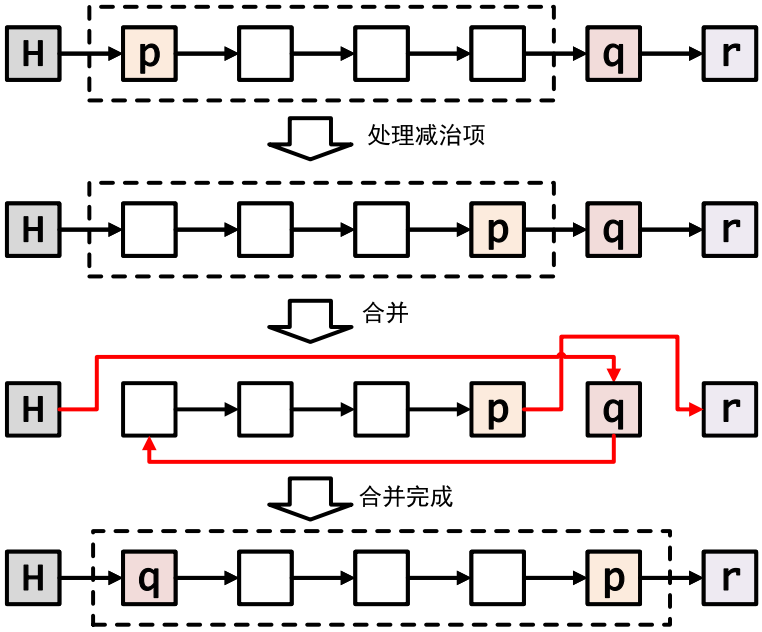

图 3.7 使用尾部减治法倒置单向列表

```
iterator reverse(iterator head, std::size_t sz) {
    if (sz <= 1) {
        return ++head;
    }
    auto p { reverse(head, sz - 1) };
    auto q { p + 1 }, r { q + 1 };
    auto tmp { std::move(p.node()->next()) };
    p.node()->next() = std::move(q.node()->next());
    q.node()->next() = std::move(head.node()->next());
    head.node()->next() = std::move(tmp);
    return p;
}
```

```
void operator()(List<T>& L) override {
    reverse(--L.begin(), L.size());
}
```

上面的示例代码虽然使用了`--L.begin()`，但这个迭代器在单向列表的场合中也可以通过头哨兵指针得到，并没有和不使用前向指针的约束相违背。这份代码虽然不是尾递归，但它很容易被改写为迭代。这是因为，如果将减治的过程分成“处理减治项”和“合并”两个阶段，则上述算法中，合并所用到的信息都可以在处理完减治项之后得到，而不需要再处理减治项之前预先保存。因此，我们可以使用固定的变量存储每次递归调用的参数和返回值，从而将减治的过程变为“平凡项→合并 1→合并 2→ ...”的迭代过程。一个示例的迭代写法如下所示。

```
void operator()(List<T>& L) override {
    auto head { --L.begin() }, p { head + 1 };
    for (auto i { 1uz }; i < L.size(); ++i) {
        auto q { p + 1 }, r { q + 1 };
        auto tmp { std::move(p.node()->next()) };
        p.node()->next() = std::move(q.node()->next());
        q.node()->next() = std::move(head.node()->next());
        head.node()->next() = std::move(tmp);
    }
}
```

上述算法的时间复杂度为$\Theta(n)$，空间复杂度为$O(1)$。此外，这是一个针对指针域进行操作的算法，在整个算法过程中，只有各个后向指针的指向发生了变化，而每个节点内部的数据域没有变化，也没有创建或删除节点。

n	数据域 直接交换	数据域 转移到向量	指针域 头部减治	指针域 尾部减治
10^5	0	1	栈溢出	0
10^6	8	29	栈溢出	14
10^7	84	263	栈溢出	136

表 3.3　顺序删除和随机删除的耗时对比

另一方面，利用`std::reverse`逐个交换双向列表中对称的节点上的数据，则是针对数据域进行操作的算法。在整个算法过程中，只有各个节点内部的数据域发生了变化，而没有改变每个节点的指针指向关系，也没有创建或删除节点。

针对指针域和**针对数据域**，是设计列表算法时的两种不同路径。针对指针域的算法将列表上的节点当做一个整体，通过修改指针来改变节点在列表中的顺序。针对数据域的算法则将列表当做一个访问受限（只能访问已知位置的直接前驱和直接后继）的向量处理。如果一个向量上的算法没有涉及循秩访问，那就可

以原封不动地迁移到列表上。涉及循秩访问的算法如折半查找则不适用。这种做法显得非常投机取巧，但效率却不一定低，在现实程序开发中不失为一种选择。

第 3.3 节 列表的归并排序

本节讨论在列表上做归并排序，这是一个比较复杂的列表操作。一种基本的思路是针对数据域的归并排序。从迭代器的观点看，向量和列表作为线性表具有高度的相似性，所以无论是双向列表还是单向列表，都可以使用和 第 2.6.3 节 或 第 2.6.4 节 一样的方法进行归并排序。唯一的区别在于向量的迭代器支持随机访问，而列表无法支持这一点。因此，在我们想要找到中点的时候，向量的版本可以直接使用左边界和右边界的平均值，而列表的版本只能传入一个规模并手动向前查找；幸运的是，迭代的时间是$\Theta(n)$的，和归并的时间相同，因此不会引入更高的时间复杂度。

第 3.3.1 节 针对指针域的归并排序

显然，上面这种归并排序没能发挥列表的特点。在归并的时候，没有必要使用快慢指针，而是可以充分利用列表的灵活性，达到不需要辅助空间的效果。比如，原先的列表可以分成 L 和 R 两部分，为了对列表 L→R 进行排序，我们可以首先将它切断成两个独立的列表 L 和 R，递归地分别排序之后，再根据 L 和 R 的首节点大小关系，将它们重新连接起来，如 图 3.8 所示。

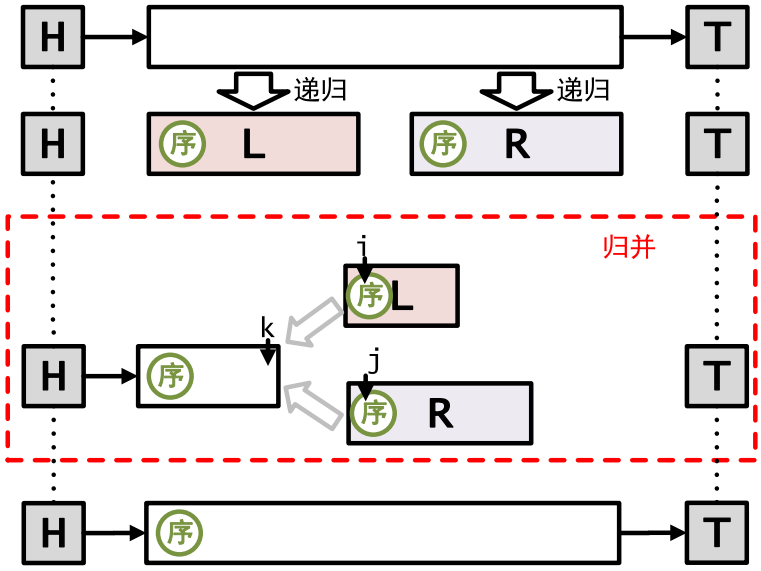

图 3.8 针对指针域的归并排序

针对数据域的归并排序在整个排序过程中没有修改任何节点的相对位置，只是改动了每个节点里的数据元素的值。因此这个版本的实现和向量相同，因为向

量也不能改变数据单元的相对位置，向量中相对位置和绝对位置（地址）是统一的。而对于针对指针域的归并排序，在整个排序过程中，每个节点的值都没有被修改，被改动的只有节点的前向和后向指针，从而改变节点的相对位置。这是在 [4] 中介绍的版本，原理仍然是归并排序，但归并的实现和向量的版本完全不同。

哨兵节点在这个版本的归并排序里会产生麻烦，因为它们会破坏拆分后的列表结构。如 图 3.8 所示，如果在拆分列表前不分离哨兵节点，则拆分后的前后两段，一个只有头节点，一个只有尾节点，需要各自单独判断；如果先分离哨兵节点再拆分，则两段都既没有头节点也没有尾节点，不需要单独判断。当数据区域被排序完成之后，再将头尾节点接入即可。值得一提的是， 图 3.8 中的 H 和 T 并不代表整个列表的哨兵节点，当递归地对列表的一部分进行排序时，H 和 T 是这一部分的第一个节点的直接前驱、最后一个节点的直接后继，或者说这一部分的哨兵节点。

```
using node_ptr = std::unique_ptr<typename L<T>::node_type>;
constexpr node_ptr&& ptr(iterator pos) {
    return std::move((--pos).node()->next());
}
void connect(iterator& prev, node_ptr& next) {
    next->prev() = prev.node();
    prev.node()->next() = std::move(next);
    next = std::move((++prev).node()->next());
}
iterator merge(iterator lo, iterator mi, iterator hi) {
    auto plo { ptr(lo) }, pmi { ptr(mi) }, tail { ptr(hi) };
    auto head { lo - 1 }, last { head };
    while (plo && pmi) {
        if (cmp(pmi->data(), plo->data())) {
            connect(last, pmi);
        } else {
            connect(last, plo);
        }
    }
    while (plo) {
        connect(last, plo);
    }
    while (pmi) {
        connect(last, pmi);
    }
    tail->prev() = last.node();
    last.node()->next() = std::move(tail);
    return ++head;
```

```
}
iterator mergeSort(iterator lo, iterator hi, std::size_t sz) {
    if (sz <= 1) return;
    auto mi { lo + sz / 2 };
    lo = mergeSort(lo, mi, sz / 2);
    mi = mergeSort(mi, hi, sz - sz / 2);
    return merge(lo, mi, hi);
}
void sort(L<T>& l) override {
    mergeSort(l.begin(), l.end(), l.size());
}
```

【C++学习】

C++11 开始允许程序员定义 constexpr 修饰的函数，并在每个版本的 C++中不断扩大允许 constexpr 函数修饰的范围。constexpr 函数被用于在编译期计算常量表达式，从而提高程序在运行期的性能。极端情况下，程序员可以在编译期计算出能够为问题范围内所有输入提供输出的一张表，从而在运行期直接查表得到结果，即使用编译时间和编译出的可执行文件大小来换取运行时间的降低。

constexpr 函数现在被用于替换几乎所有的 C 语言风格宏（macro）。宏是一种在预处理阶段被展开的文本替换，它不是真正的函数，因此不具有函数的类型检查、作用域等特性。相比之下，constexpr 函数则是真正的函数，它可以被编译器检查、优化，也可以被调试器调试。因此，现在的 C++程序员应当尽量避免使用宏，而使用 constexpr 函数。

一般的宏主要用于下列 4 个场景：

1. 定义一些常量。此类宏应当被改写为 constexpr 常量，如：

```
#define PI 3.14159265358979323846             // not recommended
constexpr auto pi { 3.14159265358979323846 }; // recommended
```

2. 定义一个枚举类型。此类宏应当被改写为 enum 类型，如：

```
#define COLOR_RED 0
#define COLOR_GREEN 1
#define COLOR_BLUE 2                    // not recommended
enum class Color { Red, Green, Blue }; // recommended
```

3. 定义一些简单的函数。此类宏应当被改写为 constexpr 函数，如：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b)) // not recommended
constexpr auto max(auto a, auto b) {      // recommended
    return a > b ? a : b;
}
```

4. 条件编译。函数体内的条件编译可改为 if constexpr，但全局的条件编译目前仍然没有替代方案。

第 3.3.2 节 归并排序的性能差异

实验 `lmergesort.cpp`。针对指针域的归并排序将空间复杂度从$\Theta(n)$降低到了$\Theta(\log n)$（一定要注意递归本身的空间复杂度，不能误以为是$O(1)$），那么显然它会在时间上有所损失。为了评估归并排序的性能差异，我们设计了一个实验。

n	向量	向量	列表	列表	列表	列表	列表	列表
方向	↓	↑	↓	↑	↓	↓	↑	↓
字段	数据	数据	数据	数据	指针	数据	数据	指针
随机化	-	-	否	否	否	是	是	是
10^4	0	0	1	1	1	1	1	1
10^5	11	11	19	19	23	50	121	37
10^6	133	132	277	483	596	1882	3663	1274

表 3.4 归并排序的耗时对比

如 表 3.4 所示，我们对比三个归并排序，分别为针对数据域的方法实现的归并排序（自上而下和自下而上版本），以及针对指针域的归并排序。实验结果表明，和向量的情况不同，在列表的场合，自下而上的归并排序消耗的时间会显著高于自上而下的归并排序。这是因为，自上而下版本的一些归并段的左右端点来自递归调用的参数，只有中点需要自己找；自下而上版本因为没有递归，所以在已知左端点的情况下，需要自己找中点和右端点。在循秩访问的向量中，这只是多了一次加法，时间消耗可以忽略不计；而在循位置访问的列表中，找端点需要循着后向指针一个一个找，时间消耗就明显增加了。自上而下的版本用时明显比向量归并排序多，一定程度上也来自于这个原因。

另一个影响归并排序效率的重要因素来自于数据的局部性。实验中提供了一个 `shuffle` 方法，用于将列表中的节点随机打乱。是否调用该方法和不调用会大幅影响实验结果。在不调用 `shuffle` 的情况下，自下而上的版本就会和自上而下版本有一定的差距，而针对指针域的版本在 `n` 较大的情况下会更慢。这是因为不随机打乱的时候，列表中的节点是连续生成的，因此针对数据域的排序能利用到局部性，而针对指针域的排序在排序过程中会将节点次序打乱，从而丧失局部性。

在调用了 `shuffle` 的情况下，情况会发生变化。所有版本的列表归并排序都会因为损失局部性而性能降低。由于循着后向指针一个一个找的过程没有了局部性，自下而上版本和自上而下版本的差距会进一步拉大。针对指针域的归并排序因为移动指针本来就会破坏局部性，受到的影响比较小。因此，如果列表的节点是随机生成的或者已经经过了长期维护，无法保证局部性，那么针对指针域的归并排序会变成一个比较好的选择。

第 3.4 节 循环列表

循环列表（circular list）基于循环链表，后者是普通的、线性的链表的一个变体。简单地说，循环列表就是舍弃了头节点和尾节点，让第一个节点 `L[0]`的前向指针指向最后一个节点`L[n-1]`，让最后一个节点`L[n-1]`的后向指针指向第一个节点 `L[0]`，首尾相连，形成的环状结构。循环列表同样可以分为单向循环列表和双向循环列表。循环列表的三种基本操作和 第 3.2 节 介绍的普通线性列表几乎完全一致，所以不再赘述。

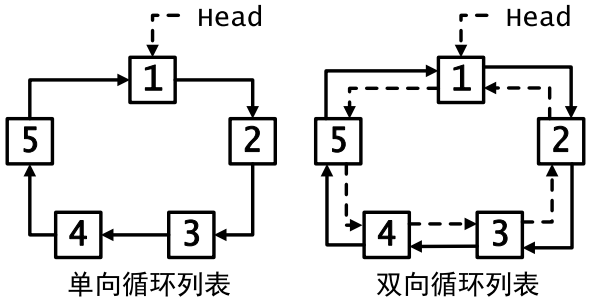

图 3.9 循环列表的结构

图 3.9 展示了单向循环列表和双向循环列表的结构。由于需要保证列表的节点首尾相连，所以无法在循环列表的环上嵌入哨兵节点。 图 3.9 中的 `head` 称为**头指针**，它是一个指向列表中第一个元素的裸指针；类似地，也可以（可选地）用一个**尾指针**指向列表中的最后一个元素。

注意到，如果循环列表中只有一个元素，那么它的后向指针（双向循环列表中，也包括前向指针）将指向自己，因此它持有自身的所有权。当使用 第 3.2.4 节 介绍的普通线性列表的删除方法去删除循环列表中唯一的一个元素时，会无法释放这个唯一元素的空间。因此，对最后一个元素进行删除的时候需要进行特判。同理，在对一个空的循环列表插入第一个元素时，也需要进行特判。此外，由于没有哨兵节点的存在，当循环列表在头部进行操作的时候，需要特别地对头指针进行维护；如果启用了尾指针，那么在尾部进行操作的时候，也需要特别地对尾指针进行维护；这些维护的行为都会给代码设计造成更大的负担，同时也使得程序运行过程中增加了一定数量的判断，影响列表的性能。测试程序位于 **`clist.cpp`**。

循环列表的主要特征是：循环列表上的指针沿着前向指针或后向指针走下去，就可以在列表上无限轮询每个节点。**`clistit.cpp`** 展示了一个例子。这种无限轮询的特点使它在《网络原理》和《操作系统》中的一些“持续性维护”的情景中有用武之地，但在《数据结构》中不常用。

第 3.5 节 静态列表

静态列表（static list）指的是基于数组实现的列表（或者也可以基于向量实现，从而允许扩容）。列表在建立的时候，申请一块连续的内存，所有的节点都存在这部分内存中。与向量有所区别的是，节点的相对位置可以和它们在数组里的位置不同。在静态列表中，前向指针和后向指针往往用所基于的数组（向量）中的下标代替，因此，静态列表虽然本质上是一种循位置访问的结构（列表），但它的位置是用秩表示的，因为列表中的节点存在数组或向量里。

和动态分配内存的动态列表相比，由于装填因子的问题，静态列表需要消耗更多的空间，所以它非常不常用（仅出现在不提供指针/引用语法的古代编程语言中）。但在小型程序的设计中（如算法竞赛），由于动态分配内存可能会引起时间消耗上的不确定性，并且此时程序可以使用的空间几乎总是绰绰有余的，所以静态列表仍然有发挥的空间。

单链/双链、线性/循环、动态/静态，这三对关系是相互独立的，可以组成 8 种不同结构的列表实现。在本节将以双链、线性为例，介绍静态列表的实现方法。为了支持扩容，这里基于向量实现。由于静态列表和动态列表非常相似，这里只介绍静态列表的特有特点。

第 3.5.1 节 静态列表上的节点

和动态列表不同，静态列表用来存储前向指针和后向指针的类型变成了秩。

```
template <typename T>
class StaticListNode {
    T m_data;
    std::size_t m_prev { 0 };
    std::size_t m_next { 0 };
};
```

静态列表节点中的前向指针和后向指针，只是指示其直接前驱和直接后继的节点在静态列表用于存储所有节点的向量中的位置，而不实际对这些节点进行内存管理。因此，静态列表中的节点不需要显式写出构造函数，并且其复制和移动语义都可以是默认的。对于秩而言，我们需要指定特殊值用来对应指针情形中的空指针。在这里，我们认为秩为 0 表示指向空；因此在实现静态列表的时候，我们不能使用秩为 0 的数据单元来存放数据，否则会产生混淆。当然，也可以规定一个不可能出现的值（比如-1），或者使用 `std::optional<std::size_t>`代替裸的 `std::size_t` 来表示秩。除了秩为 0 的单元外，我们还需要规定固定的单元用来存放头节点和尾节点。所以，对于有哨兵节点的静态列表，需要至少 3 个固定的、和数据单元无关的单元用来存放节点。这类单元被称为**元数据**（metadata），和普通的数据（data）单元相对。

```
template <typename T>
class StaticList : public AbstractStaticList<T> {
    Vector<node_type> m_nodes {};
    static constexpr std::size_t NIL { 0 };
    static constexpr std::size_t HEAD { 1 };
    static constexpr std::size_t TAIL { 2 };
    node_type& node(std::size_t index) override {
        return m_nodes[index];
    }
};
```

这里，我们采用了一个 node 方法在用于表示位置的秩和实际节点之间建立联系。静态列表的操作和动态列表相比大同小异，因为不需要考虑智能指针的所有权问题，静态列表可以更加随意一些，比如，下面是静态列表的后插实现。

```
iterator insertAsNext(iterator p, const T& e) override {
    iterator q { this, m_nodes.size() };
    auto r { p + 1 };
    m_nodes.push_back(node_type { e, p.index(), r.index() });
    p.node().next() = q.index();
    r.node().prev() = q.index();
    return q;
}
```

在上述后插实现中，我们先构造了一个节点 q，将其前向和后向指针接在 p 和 r 之间，然后再将 p 的后向指针和 r 的前向指针接在 q 上。

第 3.5.2 节　在静态列表上删除一个元素

如果我们按照动态列表的方法在静态列表上删除，可以得到下面的算法。

```
iterator erase(iterator p) override {
    auto q { p - 1 }, r { p + 1 };
    q.node().next() = r.index();
    r.node().prev() = q.index();
}
```

上述算法存在一个致命的问题：被删除的节点的内存无法被释放。设列表用来存储所有节点的向量为 V，则因为被删除的节点在向量 V 中、由向量进行内存管理，所以不能直接释放内存。所以，需要将被删除的节点从向量 V 中删除，以释放 V 中的空间，使其可以被分配给其他节点。然而，直接调用向量的删除元素方法也是不行的。因为删除 V[0:n] 中的一个节点 V[r]，会导致它的后缀 V[r+1:n] 整体前移。V[r+1:n] 的前移会导致这些节点的秩发生变化，如果其他节点的前向或后向指针指向了它们，则需要更新这些指针。所以，如果用从 V 中删除节点的

方式来清理内存，则删除一个节点的时间复杂度高达$\Theta(n-r)=O(n)$，这是列表所不能允许的。如果用这种方法来删除列表上的节点，那还不如直接使用向量来存储。

为了将删除节点的影响降到最低，我们希望 r 越大越好。当 r=n-1（被删除的节点在向量末尾）时，$\Theta(n-r)=O(1)$，这是列表所理想的结果。因为列表的元素次序和 V 中的元素次序无关，所以您可以很自然地想到，只需要将 V[r]交换到 V[n-1]，就可以顺利在$O(1)$的时间内删除了。

```
iterator erase(iterator p) override {
    auto q { p - 1 }, r { p + 1 };
    q.node().next() = r.index();
    r.node().prev() = q.index();
    if (iterator s { this, m_node.size() - 1 }; p != s) {
        p.node() = std::move(s.node());
        (p - 1).node().next() = p.index();
        (p + 1).node().prev() = p.index();
    }
    m_nodes.pop_back();
    return r;
}
```

对于元素次序不重要的向量结构，在执行删除时，可以将被删除的元素移动到向量末尾再删除，可以将删除操作的时间复杂度从$\Theta(n-r)$降低为$O(1)$，这是一个常用的技巧。在后面的章节中还会再次出现。测试程序位于 **slist.cpp**。

根据我们在 第 3.2.4 节 中的经验，这种对数据域的直接操作是不安全的。如果有一个迭代器指向 V[n-1]，那么当我们删除了前面的一个节点之后，这个迭代器就会失效，而该迭代器的持有者并不能了解到这一点。我们希望克服这个困难。为了保证迭代器不失效，我们就不能针对数据域做操作，只能针对指针域做操作。因此，在删除一个节点 V[r]的时候，被释放的空间只能是 V[r]本身，并且为了保证时间复杂度，我们不能让 V[r+1:n]整体前移，需要用一个其他机制来标识 V[r]已经被释放。

一个自然的想法是，我们将“已经被释放”的节点的秩全都记录下来。当插入一个新元素时，如果有“已经被释放”的节点，就从中取出一个节点来使用，而不是直接在向量末尾插入。这样，我们就可以保证向量中的节点是连续的，而且不会有空洞。这种方法被称为**对象池**（object pool）。需要注意，在使用对象池的时候，不再能够用 V.size()减去元数据单元数量来获得列表的规模，而需要额外维护一个规模变量。使用对象池创建节点对象时，采用下面的方法。

```
Vector<std::size_t> m_free {};
iterator create() {
```

```
    ++m_size;
    if (m_free.empty()) {
        m_nodes.push_back(node_type {});
        return iterator { this, m_nodes.size() - 1 };
    } else {
        auto index { m_free.back() };
        m_free.pop_back();
        return iterator { this, index };
    }
}
```

这样，我们在删除一个节点的时候，只需要将它的秩加入到 `m_free` 中即可，既不会导致迭代器失效，也不会导致时间复杂度增加。这种方法的时间复杂度是$O(1)$，但是空间复杂度会增加$O(n)$。

第 3.6 节 本章习题

在 第 3.1 节 中：

1. 简单 假设元素被访问的概率均等，则列表的循秩访问平均时间复杂度是多少？
2. 简单 为什么没有尾节点的单向列表无法支持 `push_back` 操作？
3. 中等 如果采用裸指针而不是智能指针来实现单向列表节点的 `next` 字段，则单向列表中有可能会出现环。给定单向列表的头节点，设计一个算法判断是否存在环，并分析其时间复杂度。
4. 中等 裸指针也有可能造成两个节点指向同一个节点的情况。已知节点 `a` 和节点 `b` 有某个公共后继 `c`，即 `a`... →`c`← ...`b`。设计一个算法找到 `c`，并分析其时间复杂度。
5. 中等 双向列表中的每个数据域需要对应 2 个指针域。修改双向列表使得每个数据域只需要对应 1 个指针域，从而节约空间。

在 第 3.2 节 中：

1. 简单 在没有哨兵节点的情况下，后插一个元素需要进行哪些特判？
2. 简单 写出采用裸指针的单向列表的后插、前插和删除算法。
3. 简单 写出单向列表的 `erase_after` 方法，删除在所给迭代器的直接后继。
4. 简单 如何在有序列表上做唯一化？和向量有什么不同？
5. 中等 双向列表可以通过 `insertAsPred(begin(), e)`实现在头部插入元素。单向列表如果使用该方法在头部插入元素，会存在什么问题？如何解决这个问题？
6. 简单 设计一个算法，将一个列表整体插入到另一个列表的指定位置（前插）。
7. 中等 设计一个算法，让同一列表中的节点 `p` 和节点 `q` 的位置互换，要求时间复杂度为$O(1)$。请注意是节点互换，而不是节点数据互换。

8. 中等 由于 CPU 执行指令是有次序的，因此软件算法通常假定两个语句的执行时刻不同，比如 `a = b, b = a` 这样的语句会使得 `a` 被赋值为 `b`，而 `b` 并不会被赋值为（原先的）`a`。
   但是，基于触发器的硬件语句并不遵循这个规则，比如在某个时钟上升沿，触发 `a = b, b = a`，则寄存器 `a` 和 `b` 的值会交换。现在希望模拟硬件上同时触发的行为，给定一个向量 `V[0:n]`和 `k` 个形如 `V[i] = V[j]`的更新语句，假定这些语句在同一时刻触发，将触发后的结果更新在向量 `V` 上。已知 `k` 远小于 `n`，设计一个算法，将这些语句的触发行为模拟出来，并分析其时间复杂度。

在 第 3.2.7 节 中：

1. 简单 写出双向列表通过针对指针域的操作实现的倒置算法。
2. 中等 头部减治算法的示例程序涉及到了创建和删除节点，修改该程序，让它也成为一个针对指针域进行操作的算法。
3. 简单 如果列表没有 `size` 方法，修改尾部减治算法，使其不需要通过预先进行一次遍历来获取列表的规模。
4. 中等 如果没有哨兵节点，是否仍然可以使用尾部减治策略，在单向列表上进行倒置操作？请给出一个算法。

在 第 3.3 节 中：

1. 简单 针对指针域的归并排序是否可以用来改写自下而上的归并排序算法？
2. 中等 在针对指针域的归并排序中，指针的所有权管理会增加问题的难度。如果使用裸指针代替 `std::unique_ptr`，则不需要考虑断链引起节点被释放的问题，该算法可以得到一定的简化。请给出一个使用裸指针的归并排序算法。
3. 较难 在针对指针域的归并排序中，示例代码会使用到节点的前向指针，从而无法直接应用在单向列表上。修改该算法使得它可以在单向列表上应用。

在 第 3.4 节 中：

1. 简单 如果循环列表上有哨兵节点，会造成哪些不利影响？
2. 简单 如何向一个空的循环列表插入第一个元素？
3. 简单 如何从一个单元素的循环列表删除最后一个元素？
4. 简单 什么情况下，循环列表的头指针需要更新？如果启用了尾指针，那么什么情况下，循环列表的尾指针需要更新？
5. 中等 设计一个算法反转单向的循环列表。
6. 中等 约瑟夫（Josephus）问题：有 `n` 个人围成一圈，从第一个人开始报数，报到 `k` 的人出列，然后从出列的下一个人开始重新报数，直到所有人都出列。求最后一个出列的人的编号。请使用循环列表实现该算法，并分析其时间复杂度。
7. 较难 约瑟夫问题实际上不需要模拟，可以通过减治的方法解决，从而得到 $\Theta(n)$甚至$O(k \log n)$的算法，设计一个这样的算法，并分析其时间复杂度。

在 第 3.5 节 中：

1. 简单 双向列表的后插操作总是包括了 4 个指针的赋值。在静态列表和动态列表的后插操作中，这 4 个指针的赋值分别需要满足什么顺序条件？
2. 中等 采用 `m_free` 实现的对象池需要$O(n)$的额外空间。有一个方法可以避免空间上的浪费：由于 `m_nodes` 中“已经被释放”的节点仍然带有 `prev` 和 `next` 两个指针，我们可以借助这两个指针将对象池里的节点也连接成一个列表，从而只需要$O(1)$的额外空间来存这个列表的第一个节点的秩即可。请实现这个方法并分析它和 `m_free` 的优劣。

第 3.7 节 本章小结

和向量相比，列表的链式结构显得不那么直观，在列表上进行插入、删除等操作时对指针的多次赋值，对初学者而言颇有难度。本书采用智能指针实现列表，为读者提供了一个不同于经典教材的视角：所有权视角。每个节点被且仅被一个智能指针持有所有权，我希望这一特性使得读者能够在设计链表操作的时候能够有迹可循（尽管在实际程序设计中您可能使用的是裸指针）。下面列出了本章的一些学习目标。

1. 您学会了绘制列表的指针关系图，通过图形设计算法。
2. 您了解到哨兵节点在列表中的作用。
3. 您学会了采用后向智能指针控制列表节点所有权，并能从所有权的视角触发设计指针赋值操作序列，注意避免“断链”和“迭代器失效”的问题。
4. 您了解到列表作为线性递归定义的数据结构，适合使用减治策略设计算法。
5. 您了解到设计列表上的算法时，可以有针对数据域和针对指针域两种路径。

	向量	双向列表	单向列表
存储连续	√	×	×
性能稳定	×	√	√
增删灵活	×	√	〇
位置安全	〇	√	×
节约空间	√	×	〇

表 3.5 归并排序的耗时对比

向量和列表的区别也是一个重要的问题。本书从插入、删除和归并排序等几种典型场景出发，对向量、双向列表和单向列表的区别作出了分析。在 表 3.5 中提供了几个视角，读者可以根据自己的理解，列出更加全面的对比表格。

第 4 章 栈

线性表是一个非常“开放的”数据结构，用户可以读、写、插入和删除线性表上的任何一个数据单元。但在实际的计算模型中，并不是所有的数据结构都允许这样做。通过对允许访问的数据进行限制，我们可以过滤掉多余的信息，简化模型的复杂程度，从而更快更好地解决问题。

栈（stack）、**队列**（queue）以及**双端队列**（deque）就是典型的“访问受限制”的数据结构，它们的实现都是以线性表作为基础的；如果说线性表是“容器”，那么栈和队列这些结构更接近于“接口”，在 C++ STL 中称它们为容器**适配器**（adapter）。另一种经典的限制访问的数据结构是**优先队列**（priority queue），它需要以树作为基础，因此放到较后的章节讨论。栈和队列限制了用户访问元素的范围，这使得它们能够防止用户做出对进程、系统、计算机或网络有害的“非法”行为，在《操作系统》和《网络原理》中都能看到它们的应用。本章将首先讨论栈及其应用。

第 4.1 节 栈的性质

第 4.1.1 节 栈的定义

栈（stack）是一种特殊的线性表，又称**下推表**。对于栈 `S[0:n]`，它的插入、访问（因为只能读一个元素，不存在“查找”）、删除操作均只能对**栈顶**元素（top 或 peek，即栈的最后一个元素）`S[n-1]`进行。如 图 4.1 所示，当栈顶为 x 时，三种操作都只能在 x 处进行。

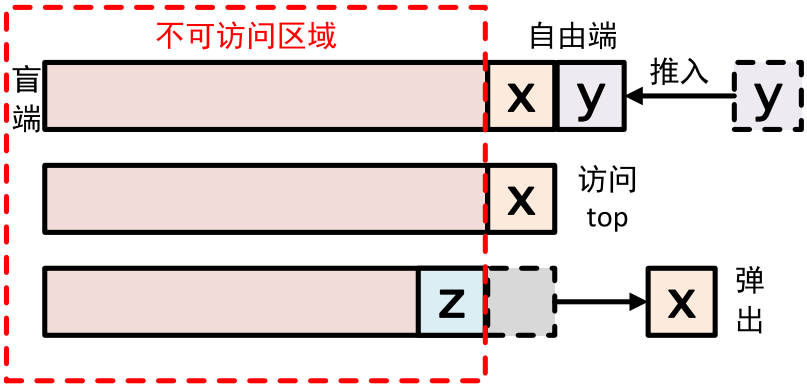

图 4.1 栈的推入、访问和弹出

1. 栈的插入就是在 `S[n-1]`后面插入新的元素（称为**入栈**或**推入**，push）。
2. 栈的访问就是取 `S[n-1]`。
3. 栈的删除就是将 `S[n-1]`从栈中删除（称为**出栈**或**弹出**，pop）。

```
template <typename T>
class AbstractStack {
public:
    virtual void push(const T& e) = 0;
    virtual T pop() = 0;
    virtual T& top() = 0;
};
```

第 4.1.2 节 栈的实现

由于栈实质上是对线性表的访问权限作出限定，所以很容易在线性表的基础上建立栈。以向量（顺序表）实现的栈称为顺序栈，以列表（链表）实现的栈称为链栈。测试程序位于 **stack.cpp**。

```
template <typename T, template<typename> typename L = DefaultVector>
class Stack : public AbstractStack<T> {
protected:
    L<T> V;
public:
    void push(const T& e) override {
        V.push_back(e);
    }
    T pop() override {
        return V.pop_back();
    }
    T& top() override {
        return V.back();
    }
    std::size_t size() const override {
        return V.size();
    }
};
```

这里需要注意的是，如果以向量（常规方式）实现栈，则我们总是使用末尾元素代表栈顶，就像在上一节所定义的那样。向量在头部做插入和删除操作的效率非常低，所以向量只能以尾部作为栈顶。但如果以单向列表实现栈，末尾元素变得不可访问，所以应当反过来以起始元素代表栈顶。双向列表因为是对称的，所以在末尾或起始都可以作为栈顶。

显然无论是顺序栈还是链栈，取顶的时间复杂度是$O(1)$，入栈和出栈的时间复杂度在分摊意义下也是$O(1)$。根据 第 3 章 的分析可知，用列表实现的栈，时间效率和空间效率都不如向量，但稳定性稍好。在绝大多数应用场景下，使用向量实现栈都比使用列表实现栈更优，很少会出现使用链栈的场景。

第 4.1.3 节 后入先出

因为栈只能从尾部进行操作的特性，栈可以被写作$[S_0, S_1, S_2, ..., S_{n-1}\rangle$的形式，左侧的中括号表示不可操作的一端（**盲端**），右侧的尖括号表示可操作的一端（**自由端**）。更常见的一种表示方法是把它竖过来：

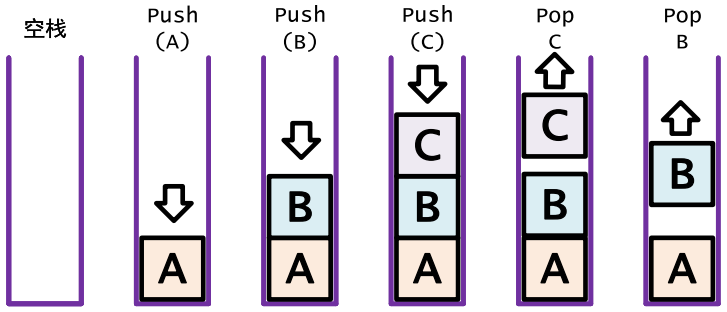

图 4.2 栈的桶式表示

如 图 4.2 所示，可以把栈想成一个桶，盲端是桶的底部，自由端（栈顶）是桶中最上面的物品。于是，push 操作就是往桶里放东西，后放的东西总是放在先放的东西的上面，像上图中，B 放在 A 上面，C 放在 B 上面。而要取东西（pop）的时候，每次只能取桶里最上面的东西，也就是栈顶元素。在 图 4.2 里，为了把 B 取出来，必须先把 B 上面的 C 取出来。

就像列表相关问题经常用链式图帮助思考一样，处理和分析栈的相关问题经常要用到上面这种桶式图。从 图 4.2 中不难发现，栈的最重要的性质是：*先入栈的元素后出栈，后入栈的元素先出栈*，简称为“**后入先出**”（**LIFO**, Last In First Out）。生活中有不少 LIFO 的例子，比如，桌子上一大摞书需要先搬开上面的才能拿下面的。这些现象被抽象化到计算机中处理就对应栈这种数据结构。比如，在 第 3.5.2 节 中实现对象池的辅助向量就是一个栈。严格来说，栈也应该采用 LIFO 进行定义，只要能够实现 LIFO 功能的数据结构都可以认为是栈，而不需要限制在 第 4.1.1 节 中这种带盲端和自由端的线性表。不过由于这种单端自由的线性表实现非常简单，通常默认栈是用这种方法实现的。在《队列》和《优先队列》章中，读者将看到其他适配器结构会有多种不同的实现方法。

第 4.2 节 出栈序列

第 4.2.1 节 出栈序列的定义

出栈序列（又称**栈混洗序列**）是关于栈的一个重点问题。下面首先给出出栈序列的定义。

给定一个序列$A=(a_1,a_2,...,a_n)$，如果对于序列$B=(b_1,b_2,...,b_n)$，存在对空栈 S 的入、出栈操作各 n 次的操作序列$O=(o_1,o_2,...,o_{2n})$，使得当序列 O 中的入栈操作为依次将$a_1,a_2,...,a_n$入栈时，序列 O 中的出栈操作出栈的元素恰好依次为$b_1,b_2,...,b_n$，则称序列 B 为序列 A 的一个**出栈序列**。

操作序列中的每个元素是 push 或者 pop，为简化叙述，本书将使用∨和∧分别表示它们。如果用桶式图想象栈的形象，那么这两个符号是直观的。

一个显而易见的结论是：出栈序列是全排列的子集。以入栈序列为(1,2,3)为例，则(3,2,1)是它的一个出栈序列，对应的操作序列为∨ ∨ ∨ ∧ ∧ ∧。容易验证，(2,3,1)、(2,1,3)、(1,3,2)以及(1,2,3)自身，都是它的一个出栈序列；而(3,1,2)则无法成为一个出栈序列。事实上，如果入栈序列是$(1,2,...,n)$，则$B=(b_1,b_2,...,b_n)$是出栈序列，当且仅当不存在“312”模式。即：不存在$i<j<k$，使得$b_j<b_k<b_i$（证明留作习题）。

根据出栈序列的定义，可以立刻得到，在给定 A 的情况下，从 O 到 B 是一个满射。那就自然地会引发猜想，从 O 到 B 也应该是一个单射。这个问题可以用递归法分析。设$O=(o_1,o_2,...,o_{2n})$中的最后一个∨是o_j，则下一个操作o_{j+1}一定是∧，且o_j入栈的和o_{j+1}出栈的元素都是a_n。那么，将o_j、o_{j+1}和a_n从序列里删去，就递归到了 n-1 的情形。递归边界 n=1 结论显然。这样，每个出栈序列就唯一对应了一个操作序列。

第 4.2.2 节 出栈序列的计数

从 第 4.2.1 节 的分析可知，给定入栈序列的情况下，出栈序列的数量等于操作序列的数量，而操作序列$O=(o_1,o_2,...,o_{2n})$的数量和入栈序列的内容无关，只和入栈序列的长度 n 相关。

在这一小节中，设长度为 2n 的操作序列 O 的数量是$f(n)$。我们希望求出$f(n)$。为了求解这个函数，需要确定操作序列 O 需要满足的条件。

1. 包括 n 个∨和 n 个∧。
2. 对于操作序列的任意一个前缀，前缀中∨的数量一定不小于∧的数量。否则，在这个前缀的操作结束之后，栈的规模会变成负数，这是不可能的。

容易验证满足上述两个条件的序列，也一定是合法的、可以生成对应出栈序列的操作序列。下面利用这两个条件，去推导$f(n)$满足的递归式。

由于条件（2），o_1必定是∨。设o_1入栈的a_1在o_k时出栈，其中$1<k\leq 2n$。又由于a_1在o_1的时候被压入了栈的底部，所以o_k之后，栈变成了空栈。因此，$(o_1,o_2,...,o_k)$和$(o_{k+1},o_{k+2},...,o_{2n})$各自都是一个比较短的、符合条件的操作序列。因而$k$必须是偶数，设$k=2i$，其中$1\leq i\leq n$。

在这两个操作序列中，o_1和o_k已经被确定了，而$(o_2, o_3, ..., o_{k-1})$有$f(i-1)$种可能性，$(o_{k+1}, o_{k+2}, ..., o_{2n})$有$f(n-i)$种可能性。于是：

$$f(n) = \sum_{i=1}^{n} f(i-1)f(n-i)$$

上述递归方程可以被改写为：

$$f(n+1) = \sum_{k=0}^{n} f(k)f(n-k)$$

下面介绍一种基于生成函数的解法[15]，这一方法可以用于求解许多类似的递归方程问题。

设$H(x) = \sum_{n=0}^{\infty} h_n x^n$，其中$h_n$为待求解的函数$f(n)$。于是，

$$\begin{aligned} H^2(x) &= \sum_{n=0}^{\infty} h_n x^n \sum_{k=0}^{\infty} h_k x^k = \sum_{n=0}^{\infty}\sum_{k=0}^{\infty} h_n h_k x^{n+k} \\ &= \sum_{n=0}^{\infty} x^n \sum_{k=0}^{n} h_k h_{n-k} = \sum_{n=0}^{\infty} h_{n+1} x^n \\ &= \frac{H(x) - h_0}{x} \end{aligned}$$

由$H(0) = h_0 = 1$，解得$H(x) = \frac{1-\sqrt{1-4x}}{2x}$。将分子上的二次根式泰勒（Taylor）展开，即可得到：

$$H(x) = \sum_{n=0}^{\infty} \frac{C_{2n}^n}{n+1} x^n$$

于是，使用生成函数法可以得到，这个递归方程可以解出显式的通项公式：

$$f(n) = \frac{C_{2n}^n}{n+1} = \frac{(2n)!}{n!(n+1)!}$$

这个数被称为**卡特兰**（Catalan）**数**，记为Catalan(n)。

除了生成函数法之外，还有一种基于一一映射的计数方法[16]，这种方法相对生成函数法更加巧妙。在这种方法中，我们分析出栈序列需要满足的两个条件。

1. 为了满足条件（1），需要在长度为 2n 的序列中安排 n 个∨和 n 个∧，可能的情况有C_{2n}^n种。
2. 如果满足条件（1）而不满足条件（2），则必定存在一个最小的 k，使得序列的前 k 项中，∨比∧少 1 个；后 n-k 项中，∨比∧多 1 个。那么，保持后 n-k 项不变，令前 k 项的∨变为∧、∧变为∨，则得到了一个新的序列，这个序列中有

n+1 个∨和 n-1 个∧。可以证明这是一个一一映射。因此，不满足条件的情况有C_{2n}^{n+1}种。

因此，出栈序列的数量为

$$C_{2n}^{n}-C_{2n}^{n+1}=\frac{C_{2n}^{n}}{n+1}$$

递归方程（即使用生成函数求解递归方程）法和一一映射法，是计数问题的两种基本方法。相对来说，递归方程法比较常规，很容易列出递归方程，但计算比较复杂；一一映射法则需要较高的构造技巧，而计算比较简单。

第 4.2.3 节 随机出栈序列

这一小节从扩展卡特兰数的角度出发，讨论如何生成一个随机的栈操作序列[16]。

考虑包含 p 个∨和 q 个∧的序列，其中$0\le p\le q$。如果在这个序列前添加 q-p 个∨可以得到合法的栈操作序列，则下文称其为 p,q 的*后缀操作序列*（显然，一个合法操作序列的任一后缀都是后缀操作序列）。设 p,q 的后缀操作序列有$C_{p,q}$个，则可以得到$C_{n,n}=\mathrm{Catalan}(n)$。

直接对栈的操作序列做递归比较困难。而定义了后缀栈操作序列之后，就可以研究生成过程中的中间结果。我们的总体目标是生成 n+n 的后缀操作序列。如果从前到后生成，在某一步处已经生成了 n-p 个∨和 n-q 个∧，那么剩余部分是一个 p,q 的后缀操作序列，有$C_{p,q}$种等概率的可能性。从而，这一步生成∨的概率是$\frac{C_{p-1,q}}{C_{p,q}}$，生成∧的概率是$\frac{C_{p,q-1}}{C_{p,q}}$。

上述结论表明，只要计算出$C_{p,q}$的通项公式，就可以从前向后随机地逐个生成操作序列上的操作。使用 第 4.2.2 节 介绍的方法构造一一映射，容易证明：

$$C_{p,q}=C_{p+q}^{q}-C_{p+q}^{q+1}=\frac{q-p+1}{q+1}C_{p+q}^{q}$$

因此，生成∨的概率为：

$$\frac{C_{p-1,q}}{C_{p,q}}=\frac{q-p+2}{q-p+1}\cdot\frac{p}{p+q}$$

这意味着不需要实际计算组合数，就可以得到每次生成∨和∧的概率。从而可以得到时间复杂度为$\Theta(n)$的算法（假设随机数生成器的时间复杂度为$O(1)$）生成一个长度为 2n 的合法操作序列。

第 4.3 节 括号序列

第 4.3.1 节 合法括号序列的计数

卡特兰数不仅仅是出栈序列（合法操作序列）的数量，同时也是其他很多问题的答案。这些问题的解决，也普遍具有两条路径：

1. 从模型角度入手，将其变换为等价的出栈序列或合法操作序列问题，然后套用卡特兰数的通项公式。（一一映射法）
2. 从计算角度入手，列出递归方程，发现和卡特兰数的递归方程的相似性（不需要使用生成函数实际求解）后，利用已知的卡特兰数通项公式求解。（递归方程法）

一个典型的例子是合法括号序列问题。考虑左括号和右括号组成的合法括号序列，可以用以下的递推定义：

1. 空串是合法括号序列。
2. 如果 S 和 T 是合法括号序列，那么(S)T 也是合法括号序列。

条件（2）的一个等价形式拆分成两个条件：如果 S 是合法括号序列，则(S)是合法括号序列；以及，如果 S 和 T 是合法括号序列，那么 S T 也是合法括号序列。这种两个条件的版本更符合直观认知。

可以注意到，从(S)T 递归到长度更短的 S 和 T，这个递归方法和前面推导合法操作序列数量时使用的递归方法如出一辙，因此，用∨代替左括号，用∧代替右括号，就可以建立合法括号序列到合法操作序列的一个映射。容易证明这是一个双射，从而化归到出栈序列的问题上来。另一个方向也是类似的，容易通过条件（2）得到和之前完全一样的递归方程，从而解出卡特兰数的通项公式。

在这个问题上，从模型角度入手建立一一对应是显然的。但有一些问题，模型上可能一时看不出来，但通过计算角度发现结果是卡特兰数之后，就可以自然地联想到从模型上可以建立一一对应。后面还会遇到的一个非常重要的卡特兰数应用是树和二叉树的计数问题，我们将在对应章节进行分析讨论。

第 4.3.2 节 判断括号序列是否合法

回到合法括号序列的问题上来。发现合法括号序列的数量和出栈序列相同之后，在讨论合法括号序列的问题时，就可以自然地联想到从栈的角度入手。*数量上相同只是初步的结论，更重要的是建立了一一映射*。比如，借助这个一一映射，就可以用栈来判断一个括号序列是否合法。

实验 paren.cpp。在这个实验中，我们将同时讨论小括号、中括号和大括号，因此我们首先建立括号直接的对应关系。在存在多种括号的场合，只需要修改合法括号序列中的条件（2），将递推定义的合法括号序列包括[S]T 和{S}T。

```
class ParenMatch : public Algorithm<bool(const std::string&)> {
protected:
    char left(char c) {
        switch (c) {
            case '(': case ')': return '(';
            case '[': case ']': return '[';
            case '{': case '}': return '{';
            default: return 0;
        }
    }
};
```

从上面的分析中我们可以看到，左括号和右括号在建立一一映射的时候分别被映射为了∨和∧。因此，我们从左到右扫描括号序列的时候，每次扫描到左括号，就进行一次入栈操作；每次扫描到右括号，就进行一次出栈操作。由于我们考虑了三种括号，所以我们需要保证左括号和右括号是同一种。因此，每次扫描到左括号，就将它入栈（因此栈里只有左括号）；每次扫描到右括号，就从栈里弹出一个左括号，判断是否和右括号匹配。示例程序如下所示。

```
bool operator()(const std::string& expr) {
    Stack<char> S;
    for (auto c : expr) {
        if (auto l { left(c) }; l == c) {
            S.push(c);
        } else if (l) {
            if (S.empty() || S.top() != l) return false;
            S.pop();
        }
    }
    return S.empty();
}
```

如果只有一种括号，那么栈中的所有元素都是相等的。在这种情况下，栈也变得不必要，因为我们事实上只需要知道栈的规模，用一个整数来表示即可；从而将空间复杂度降低到了$O(1)$。

第 4.4 节 栈与表达式

第 4.4.1 节 表达式的定义

括号主要的用处，就是给表达式规定计算顺序。所以，自然地就能想到，栈也可以被用来计算表达式。

回顾 第 4.3 节，括号序列和栈的操作序列相对应，而一个出栈序列，是由入栈序列和操作序列共同决定的。从信息的角度看，如果要在出栈序列和表达式之间建立联系，且已知操作序列和括号序列之间有联系，那么顺理成章地，就会想到，入栈序列应当和表达式中的其他部分——也就是**操作数**（operand）和**运算符**（operator）建立联系。

那么接下来，就需要推导这个联系了。为了更清晰地展示推导过程，以下采用表达式$1+2*(3-4)$作为例子。

1. 建立完整的括号序列。

   这样做的目的是，让括号序列能够完全地和运算次序形成一一对应。原有的表达式中，有一些括号被省略了，这一步的目的是将它补全。在例子中，$1+2*(3-4)$被变换为$((1)+((2)*((3)-(4))))$。

   这里给每一个操作数也加上了一堆括号；这是因为，我们的目标是将“入栈序列”和“操作数与运算符”之间建立联系。在表达式中，操作数和运算符一共有 7 个（$1,2,3,4,+,*,-$），因此入栈序列的长度为 7，相应地，操作序列的长度应该为 14，也就是说括号序列的长度应该为 14（7 对括号）。

   在这 7 对括号中，有 3 对是用来描述运算符的计算次序的，而另外 4 对则用来描述操作数的位置。这样才能达成一一对应。

2. 构造入栈序列和出栈序列。

   利用第 1 步中建立的“括号对”和“操作数与运算符”的一一对应关系，决定每一对括号对应的入栈或出栈的元素。如果某一对括号对应的运算符或者操作数为c，那么将左括号变换为$\vee(c)$，右括号变换为$\wedge$（出栈元素恰好为c），就得到了一个带参数的操作序列。

   在例子中，第 1 步加入了 7 个括号，其中 3 对用来描述 3 个运算符的运算次序，4 对用来描述 4 个操作数的位置，也就是说，每一对括号都对应了一个运算符或者操作数。它对应的带参数操作序列是：$\vee(+)\vee(1)\wedge\vee(*)\vee(2)\wedge\vee(-)\vee(3)\wedge\vee(4)\wedge\wedge\wedge\wedge$。对应的入栈序列是：`+ 1 * 2 - 3 4`；对应的出栈序列是：`1 2 3 4 - * +`。

定义这个入栈序列为**前缀表达式**（又称**波兰式**），出栈序列为**后缀表达式**（又称**逆波兰式**）。在前缀表达式和后缀表达式中，由于两个数字可能连在一起，所以为了区分边界，通常使用空格或其他分隔符隔开相邻的两个元素。熟知的数学的表达式形式，即$1+2*(3-4)$，被称为**中缀表达式**。这三个概念来自于运算符相对于操作数的位置。在前缀表达式中，运算符出现在它所对应的操作数之前；后缀表达式放在之后，而中缀表达式的运算符出现在中间。后缀表达式和前缀表达式的性质大多是对偶的，而后缀表达式更适合使用计算机计算。

第 4.4.2 节 表达式中的元素

为简单考虑，本书中只讨论操作数为整数的表达式，讨论的运算符则包括 7 种：加、减、乘、除、取余、乘方、阶乘。通过对负号和阶乘的支持，我们要求表达式处理中允许单目运算符的存在，且单目运算符可以放在操作数的左侧（负号）或者右侧（阶乘）。

需要指出的是，表达式的实现在各个教材中会有很大的、基础性的差异。比如，针对表达式中的元素的抽象，有三种解决方案。

1. 分离方案 [4] 。操作数和运算符被分离，使用不同的类型的数据单元进行存储。这种方案的优点在于不需要自己写类对数据单元进行封装（可以使用基本数据类型，如 `char` 和 `int`），缺点是在处理表达式时往往需要两个栈分别处理操作数和运算符。而在表达式中，操作数和运算符本质上都是表达式的元素，分离方案没有显示出这一点。
2. 合取方案。在每个数据单元中，存储一个运算符和一个操作数。对于输入的中缀表达式，将每个运算符和紧随其后的一个操作数合并在一个数据单元存储。通过在整个表达式的开头和结尾增加一对括号，我们可以保证在中缀表达式中每个操作数都紧跟在某个运算符之后。这种方案的优点在于处理中缀表达式时非常方便，缺点是将操作数和它前面的运算符合并，这一操作并不自然。
3. 析取方案。在每个数据单元中，存储一个运算符或一个操作数。这种方法可以很好地表达操作数和运算符都是表达式元素的本质。缺点是为了区分运算符和操作数，需要额外的标记，从而增加了存储空间的开销。

本书将采用析取方案，我们将从表达式元素的基本功能开始，逐步往其中添加功能，使其可以支持表达式转换和计算。表达式的每个元素可以定义如下。

```
template <typename T>
class ExpressionElement {
    std::variant<char, T> m_element;
};
```

【C++学习】

`std::variant` 是 C++17 引入的新特性，它是一种类型安全的联合体，可以存储多种类型的值，但在任意时刻只能存储其中一种。可以通过 `std::get` 来访问存储在其中的值并转换为目标类型，也可以通过 `std::holds_alternative` 来判断当前存储的是哪种类型。示例代码如下：

```
char getOperator() const { return std::get<char>(m_element); }
bool isOperator() const {
    return std::holds_alternative<char>(m_element);
}
```

std::variant 通常被用于替代 C 语言中的 union。和使用 constexpr 代替常规宏的做法不同，采用 std::variant 替代 union 不是无损的。二者的比较如下：

1. std::variant 是类型安全的，能够提供运行时的类型检查和动态分派；而 union 不是，需要程序员自己保证类型安全性，如果程序员错误地访问了 union 中的非活跃成员，那么程序的行为是未定义的。
2. std::variant 需要存储额外的类型信息，因此会占用更多的内存；而 union 不需要额外的类型信息，只占用存储在其中的最大类型的内存。
3. std::variant 是一个类模板，容易通过模板元编程的方法和其他模板类一起使用，支持通过模板特化和继承的方法扩展。相应不可避免地，std::variant 也会带来更长的编译时间（不过在小型程序中通常都是可以忽略的）。
4. union 允许程序员直接控制内存布局，在一些低级编程任务（比如和硬件交互）中，这种控制是必要的。

我们使用 apply 方法表示运算，当元素是运算符时，它接收至多两个操作数，返回运算的结果；当元素是操作数时，它直接返回操作数本身。operandPosition 方法则表示操作数的位置，当元素是运算符时，两个返回值各表示左侧和右侧是否可以接受操作数（比如阶乘，就只允许左侧有操作数）；当元素是操作数时，当然两边都不能有操作数。

```
T apply(const T& lhs, const T& rhs) const override {
    if (isOperator()) {
        switch (getOperator()) {
        case '+': return lhs + rhs;
        case '-': return lhs - rhs;
        case '*': return lhs * rhs;
        case '/': return lhs / rhs;
        case '%': return lhs % rhs;
        case '^': return m_power(lhs, rhs);
        case '!': return m_factorial(lhs);
        }
    }
    return getOperand();
}

std::pair<bool, bool> operandPosition() const {
    if (isOperator()) {
        switch (getOperator()) {
        case '(': return { false, true };
        case ')': case '!': return { true, false };
        case '+': case '-':
        case '*': case '/':
```

```
        case '%': case '^': return { true, true };
        }
    }
    return { false, false };
}
```

这里乘方计算可以通过 第 1.5.4 节 所介绍的快速幂算法得到，对于阶乘，我们可以使用递归的方法进行计算。

```
constexpr T factorial(T n) const {
    return n == 0 ? T { 1 } : n * factorial(n - 1);
}
```

【C++学习】

这里使用了 constexpr 关键字，当我们正常调用一个 factorial(n)时，这个关键字并不会产生效果，但如果这个函数的参数是一个常量表达式，那么编译器会在编译时计算这个函数的返回值，而不是在运行时计算。比如，factorial(3)就会在编译时被计算为 6。这样可以提高程序的性能。

第 4.4.3 节 将字符串解析为中缀表达式

本节讨论将字符串解析为中缀表达式。我们将表达式看成是析取元素组成的线性表，每一个元素存储一个操作数或一个运算符。在将字符串解析为中缀表达式的这个过程中，我们需要进行两件事：

1. 将操作数提取出来。每个运算符都是一个字符，而操作数可能不止有一个字符。因此，当扫描到数字时，不能立刻将它加入到表达式中，需要继续向下扫描，直到扫描到运算符或字符串尾，才能确认这个数字结束。
2. 将表达式中的负号和减号区分开来。减号需要两个操作数进行运算，而负号只是将它后面的操作数取反，二者具有不同的语义。通过分析可以发现，运算符-表示负号，当且仅当它在字符串起始位置，或前一个元素是左括号。

一种可能的实现如下所示。可以看到在字符串的左边界和右边界都有可能需要特判，如果不想特判也可以在字符串两端加上括号，然后再解析。

```
Expression(std::string_view expr, std::size_t base = 10) {
    auto start { 0uz };
    T temp {};
    auto add { [&](std::size_t end) {
        if (end > start) {
            std::from_chars(expr.data() + start,
                            expr.data() + end, temp, base);
            push_back(std::move(temp));
        }
    } };
```

```
    for (auto i { 0uz }; i < expr.size(); i++) {
        if (auto c { expr[i] }; !std::isdigit(c)) {
            add(i);
            if (c == '-' && (empty() || back() == '(')) {
                start = i;
            } else {
                start = i + 1;
                push_back(c);
            }
        }
    }
    add(expr.size());
}
```

【C++学习】

上面的代码中，我们使用了 `std::from_chars` 函数将字符串转换为整数。这个函数是 C++17 引入的新特性，它可以将字符串转换为整数。借助这个函数，我们可以使用一个函数调用来完成字符串到整数的转换，而不需要根据用户传入的整数类型选择不同的转换函数（如 `std::stoi`、`std::stol` 等）。当用户定义了自己的整数类型时，只需要重载 `std::from_chars` 函数即可。扩展成其他类型的转换也是类似的。

第 4.4.4 节　中缀表达式转换为后缀表达式

在 第 4.4.1 节 中，我们已经介绍了将中缀表达式转换为后缀表达式的方法。对于短表达式，也可以采用以下的快速手工方法，如 图 4.3 所示。

1. 补全运算符对应的括号，而不补操作数对应的括号。
2. 对于每一对括号，设它对应的运算符为 o，则删掉左括号，将右括号替换为 o；如此就得到了后缀表达式。

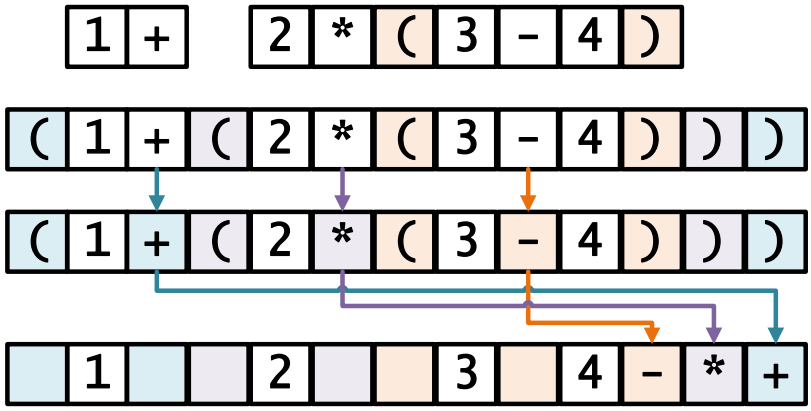

图 4.3　手工将中缀表达式转换为后缀表达式

如果要得到前缀表达式，类似地，只需要删掉右括号，将左括号替换为 o 即可。前缀表达式的运算符位于入栈的位置上，而后缀表达式的运算符位于出栈的位置上，这都是非常自然的；反而，我们熟悉的中缀表达式，运算符的位置是不自然的。所以，为了计算中缀表达式，可以先将它转换为前缀表达式或后缀表达式。此处以后缀表达式为例，您可以自行完成前缀表达式的情况。

在手工方法中，操作数仍然在它原本的位置上，而运算符则被移动到了它出栈的位置（必定后于运算符出现的位置）。因此，我们可以对中缀表达式从前到后扫描，当扫描到操作数的时候将其直接加入后缀表达式，而当扫描到运算符的时候将其入栈，在运算符出栈的时候，再将它加入到后缀表达式中。但是，当在计算机中处理中缀表达式的时候，并不是每一个运算符都有对应的一对括号，因此，我们需要配合运算符的优先级进行处理。

我们的目标是寻找每个运算符应当在哪个位置出栈。我们考虑不为左右括号的运算符 o，因为左右括号不会被加入到后缀表达式中。考虑表达式 `( A o B )`，其中，左右的一对括号是 o 对应的括号（在中缀表达式中有可能实际存在，也有可能实际上不存在），A 和 B 都是表达式。

1. 如果这对括号实际存在，那么我们分析表达式 B 的情况。如果 B 是操作数或者被一对括号包裹，则它确实参与了 o 的运算。如果 B 是其他情况，则可以设 B 为 `C p D`，其中 p 是 B 在最后一步进行的运算，C 和 D 都是表达式。那么，只有 o 的优先级小于 p 时，B 作为整体才会参与 o 的运算。否则，在 `A o C p D` 中，会先计算 `A o C`，o 对应的括号应当不包括 `p D` 这一段。
2. 反过来，如果这对括号实际不存在，那么我们需要定位到它的位置。根据上面的分析可以知道，我们从 o 向后可以继续扫描，直到扫描到优先级不大于 o 的运算符 p 为止（不包括括号里的运算符）。o 对应的括号应当出现在 p 之前；所以在扫描到 p 的时候将 o 出栈。当然还有一种特别的情况就是检查到表达式的末尾。如果在 第 4.4.3 节 中对中缀表达式的开始和结尾加上了一对括号，那么此时就不需要特殊处理。

综上所述，我们可以得到这样的算法框架：

1. 当扫描到操作数时，直接加入后缀表达式。
2. 当扫描到非括号的运算符 o 时，首先将栈内优先级不小于 o 的运算符依次弹出并加入后缀表达式，然后将 o 入栈。

现在考虑括号的情况。我们会发现，左括号作为运算符，很难确定它的优先级。一方面，当扫描到左括号的时候，它应当具有最高的优先级，左括号内部永远应当先于当前的栈顶运算符进行计算。另一方面，当左括号在栈内时，它应当具有（除了右括号外）最低的优先级，因为必须计算完括号才能脱括号，所以在找到其对应的右括号之前，任何运算符都不应该让左括号出栈。

这就意味着我们不能用单一的优先级来描述运算符，而是需要将其拆分为栈内优先级和栈外优先级。当我们扫描到运算符 o 时，将栈顶的栈内优先级，和 o 的栈外优先级进行比较。如果栈顶的栈内优先级不小于 o 的栈外优先级，则将栈顶弹出。左括号出栈后不进入后缀表达式，右括号则不必入栈，因为右括号对应的栈操作是∧，扫描到右括号的时候恰好会弹出它对应的左括号。

除了左括号外，其他运算符的栈内和栈外优先级相同；右括号的栈内优先级和栈外优先级都是最低。左括号具有最高的栈外优先级，以及除右括号之外最低的栈内优先级。因此，左括号入栈必定不会引起其他运算符出栈（括号内总是先于括号外计算），且仅当扫描到右括号的时候左括号可以出栈。

只考虑左结合运算的情况下，可以将运算符优先级实现如下：

```
inline static const std::unordered_map<char, int> priority_left {
    {'(', 0}, {')', 0}, {'^', 3}, {'!', 4},
    {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}, {'%', 2}
};
inline static const std::unordered_map<char, int> priority_right {
    {'(', 6}, {')', 0}, {'^', 3}, {'!', 4},
    {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}, {'%', 2}
};

bool prior(char next) const override {
    if (isOperator()) {
        auto prior_left { priority_left.find(getOperator()) };
        auto prior_right { priority_right.find(next) };
        if (prior_left != priority_left.end() &&
            prior_right != priority_right.end()) {
            return prior_left->second >= prior_right->second;
        }
    }
    return false;
}
```

上面虽然进行了很多分析，但目的只是为了让读者更好地理解我们为什么要这样设计算法。算法的实际实现非常简单，如下所示。

```
void infix2suffix() {
    Stack<ExpressionElement<T>> S;
    Expression<T> suffix;
    auto process { [&](const ExpressionElement<T>& e) {
        while (!S.empty() && S.top().prior(e.getOperator())) {
            if (auto op { S.pop() }; op != '(') {
                suffix.push_back(std::move(op));
            } else {
```

```
                break;
            }
        }
    } };
    for (auto& e : *this) {
        if (e.isOperand()) {
            suffix.push_back(std::move(e));
        } else {
            process(e);
            if (e != ')') {
                S.push(std::move(e));
            }
        }
    }
    process(')');
    *this = std::move(suffix);
}
```

第 4.4.5 节　后缀表达式转换为中缀表达式

现在考虑上述问题的反问题：如何从后缀表达式转换为中缀表达式？

下面从信息的角度分析，“中缀转后缀”和“后缀转中缀”有什么不同。注意到，中缀表达式是由“语义元素”（操作数和非括号的运算符）以及“括号序列”共同组成的。“语义元素”是参与运算的元素，对应了入、出栈过程中操作的数据；而“括号序列”是进行运算的次序，对应了入、出栈过程中的操作序列。所以知道中缀表达式，也就知道了操作序列的每一步入、出栈了什么元素，从而得到入栈序列（前缀表达式）和出栈序列（后缀表达式）。从信息的角度看，中缀表达式中包含的信息，包括了前缀表达式和后缀表达式所需的信息。

但是，后缀表达式仅仅是由“语义元素”组成的，它本身只是一个出栈序列。从前面的小节中您已经知道，给定出栈序列而不给定操作序列，则有 `Catalan(n)` 种可能的入栈序列。所以，如果仅仅依赖后缀表达式的文本，而不借助其他先验信息，那么从后缀表达式反推中缀（前缀）表达式是不可能的、是缺少信息的。要能够实现后缀表达式到中缀表达式的转换，必须要知道文本以外的额外信息。

这个先验的额外信息就是：*每个运算符需要的操作数数量*。在常见的运算符中，加、减、乘、除、乘方，操作数数量都是 2；阶乘的操作数数量是 1。需要特别注意的是，当“减号”和“负号”都被作为运算符处理，且采用同一符号 `-` 时，它的操作数数量是不确定的，这一不确定性会导致后缀表达式出现歧义。中缀表达式可以通过判断 `-` 之前是否为操作数来进行区分，而后缀表达式无法进行区分。比如：对于后缀表达式 `1 2 - 3 - -`，它的三个 `-` 中有一个代表负号，另外两个代表减

号。因此，这个后缀表达式可能对应 3 种不同的中缀表达式：`1-((-2)-3)`、`(1-2)-(-3)`和`-((1-2)-3)`。 第 4.4.3 节 中我们处理中缀表达式的时候，在解析阶段就已经将减号和负号进行了分离。负号被认为是操作数的一部分，而所有出现在表达式中的运算符`-`都是减号；在后缀表达式的处理中我们也将延续这一约定以避免上述歧义。

下面解释这个先验的额外信息是如何帮助后缀表达式转换成中缀表达式的。

假设某个运算符 `o` 的操作数数量是 2。在补全了所有括号的中缀表达式中，这个操作数出现在`((A) o (B))`这个片段里，其中 `A` 和 `B` 都是中缀表达式。按照 第 4.4.4 节 的方法，这个片段化为后缀表达式是 `A' B' o`，其中 `A'`和 `B'`是 `A` 和 `B` 对应的后缀表达式。于是得到：对于后缀表达式 `E` 中的每个运算符 `o`，都存在 `E` 中的一个连续的子序列 `E(o)`，`E(o)`是以 `o` 结尾的后缀表达式。

在之前讨论一般的出栈序列的时候，`A'`和`B'`的划分是任意的，所以会得到卡特兰数的递归方程。但在讨论后缀表达式的时候，需要受到操作数数量的限制，因此`A'`和`B'`的划分不是任意的。事实上，只有一种划分是合法的。我们知道，后缀表达式的开头必定是操作数，末尾必定是运算符。对于语义元素组成的序列$C = (c_1, c_2, ..., c_n)$，定义：

$$f(c_i) = \begin{cases} 1, & c_i \text{ 是操作数} \\ 1 - \mathrm{Cnt}(c_i), & c_i \text{ 是运算符} \end{cases}$$

其中$\mathrm{Cnt}(c_i)$表示运算符c_i的操作数数量。令$f(C) = \sum f(c_i)$，则对于合法的后缀表达式 `E`，可以证明$f(E) = 1$：使用数学归纳法，对E中含有的运算符数量归纳。

1. 对于单个操作数，$f(E) = 1$。
2. 对于以 `k` 元运算符结尾的表达式，设 `E` 为$A_1 A_2 ... A_k c$，其中A_i是后缀表达式。从而$f(E) = \sum f(A_i) + f(c) = k + (1 - k) = 1$。

进一步地，用数学归纳法可以证明，对于合法后缀表达式E的任意前缀$E_j = (e_1, e_2, ..., e_j), j \le n$，始终有$f(E_j) \ge 1$。

1. 由于e_1是操作数，所以$f(E_1) = 1$。
2. 如果e_j是操作数，则$f(E_j) = f(E_{j-1}) + 1 \ge 1 + 1 > 1$。
3. 如果e_j是运算符，根据前面的证明的结果，必定存在某个$(e_t, e_{t+1}, ..., e_j), 1 \le t < j$是以$e_j$结尾的后缀表达式。于是我们知道$f(e_t, e_{t+1}, ..., e_j) = 1$，从而$f(E_j) = f(E_{t-1}) + f(e_t, e_{t+1}, ..., e_j) \ge f(e_t, e_{t+1}, ..., e_j) = 1$。

在上述结论的基础上，我们就可以使用反证法证明最开始提出的那个命题：对于以 `k` 元运算符 `c` 结尾的后缀表达式 `E`，只有唯一的合法划分方式使 `E` 被划分为$A_1 A_2 ... A_k c$。

如果有不止一种合法的划分方式，设两种划分方式中$(A_1, A_2, ..., A_k)$序号最大的一个不同的项是A_j和$A_{j'}$。不妨设A_j的长度大于$A_{j'}$，那么A_j则可以被拆分为两段$PA_{j'}$。因为A_j和$A_{j'}$都是合法的后缀表达式，所以根据此前的结论，$f(P) = f(A_j) - f(A_{j'}) = 1 - 1 = 0$。但由于$P$是$A_j$的前缀，于是$f(P) \geq 1$，矛盾。

因此，递归地进行唯一合法的划分，就可以得到最终的中缀表达式。类似可以得到前缀表达式转换为中缀表达式的手段。

需要说明的是，上面的转换方式并不实用，本书在此处也没有给出相应的示例代码，仅仅从理论的角度进行介绍。实际转换的时候，通常利用**表达式树**作为中介进行转换，表达式树的内容将会在后面的章节介绍。在本章介绍表达式，是希望读者在思维中巩固“栈”和“表达式”之间的联系。

第 4.4.6 节　后缀表达式转换为前缀表达式

在 第 4.4.5 节 中使用了一个$f(\cdot)$函数用来辅助证明，这个函数并不是凭空产生的，而是另一个栈的产物。我们采用下面的方法构造一个操作序列。

1. 将后缀表达式中的操作数 i，用$\vee(i)$替代。
2. 将后缀表达式中的运算符 o，用$\wedge^k \vee (o)$替代，其中$\wedge^k$表示 k 个连续的$\wedge$，k 是这个运算符的运算所需操作数数量。
3. 在整个序列的结尾处增加一个$\wedge$。

根据 第 4.4.5 节 的结论，我们可以确定，这样得到的操作序列是合法的。

我们发现，在这个操作序列中，对应的入栈序列恰好是后缀表达式。自然地，我们想到要分析它的出栈序列。仍然以前面的 `1 2 3 4 - * +`作为例子，它对应$\vee(1) \vee (2) \vee (3) \vee (4) \wedge \wedge \vee (-) \wedge \wedge \vee (*) \wedge \wedge \vee (+) \wedge$，从而可以得到出栈序列是 `4 3 - 2 * 1 +`。这恰好是倒序的前缀表达式。

第 4.4.7 节　后缀表达式的计算

后缀表达式的计算是重要的基本功之一。要计算后缀表达式，只需要对 第 4.4.6 节 中的构造操作序列作出一点点修改。

1. 将后缀表达式中的操作数 i，用$\vee(i)$替代。
2. 将后缀表达式中的运算符 o，用$\wedge^k \vee (r)$替代，其中$\wedge^k$表示 k 个连续的$\wedge$，k 是这个运算符的运算所需操作数数量。r 是本次运算的结果。设第i个$\wedge$出栈的数为A_i，则$r = c(A_1, A_2, ..., A_k)$。
3. 在整个序列的结尾处增加一个$\wedge$。这次出栈的数就是后缀表达式的计算结果。

```
T calSuffix() const {
    Stack<T> S;
    for (auto& e : *this) {
        if (e.isOperand()) {
```

```
            S.push(e.getOperand());
        } else {
            auto [l, r] { e.operandPosition() };
            T rhs { r ? S.pop() : 0 };
            T lhs { l ? S.pop() : 0 };
            S.push(e.apply(lhs, rhs));
        }
    }
    return S.pop();
}
```

实际出现的后缀表达式计算问题，可以使用上述算法手工计算。不过和之前一样，我更推荐使用表达式树而不是栈进行计算。诚然，表达式树的做法比栈要复杂一些；但表达式树的方法更加清晰，更加适合答题结束后的检查，更加不容易出错。知道后缀表达式如何计算之后，由于之前已经介绍过各种表达式之间互相转换的方法，所以也就能够得到计算前缀表达式和中缀表达式的算法了。

第 4.4.8 节　中缀表达式的计算

实验 **expr.cpp**。在这个实验中将实现中缀表达式的计算，从而实现一个类似计算器的功能。根据前面的结论，我们可以先将中缀表达式转换为后缀表达式，再使用后缀表达式计算。

```
T operator()(std::string_view expr) override {
    Expression<T> e { expr };
    e.infix2suffix();
    return e.calSuffix();
}
```

因为无论是中缀转后缀，还是后缀求值，都是从左到右依次进行，所以我们也可以将中缀转后缀和后缀求值的过程合并在一起。如果在将中缀转换为后缀的过程里，将向后缀表达式新加入一个元素e_j的过程记为$\vee\left(e_j\right)$；在计算后缀表达式的过程里，遍历后缀表达式处理元素e_j的过程记为$\wedge\left(e_j\right)$。由于$\vee\left(e_1\right),\vee\left(e_2\right),\ldots$和$\wedge\left(e_1\right),\wedge\left(e_2\right),\ldots$都是按顺序依次进行的，所以只需要将中缀转换为后缀的过程里的行为$\vee\left(e_j\right)$替换为$\wedge\left(e_j\right)$，就可以跳过中间的后缀表达式的构造，直接计算中缀表达式的值。

具体而言，需要使用两个栈，将运算符和表达式拆开。存储运算符的栈对应第 4.4.4 节 中的辅助栈，用于分析中缀表达式的计算次序。在将中缀表达式转换为后缀表达式的过程中，需要用这个栈来对运算符重新排序，因为后缀表达式的计算次序就是每个运算符的出现次序。在直接计算中缀表达式的过程中，只需要用这个栈来储存暂时不能计算的运算符即可，当栈内的运算符被弹出时，直接用它进行计算即可，不需要把它加入到后缀表达式里。另一个需要使用到的栈则来

自 第 4.4.7 节 中的计算后缀表达式的过程。这个栈用于存储操作数，用于计算运算符的结果。以下展示了直接计算中缀表达式的过程，它基本上是 第 4.4.4 节 和 第 4.4.7 节 中展示的程序的融合。

```
T calInfix() const {
    Stack<T> Sr;
    Stack<ExpressionElement<T>> So;
    auto process { [&](const ExpressionElement<T>& e) {
        while (!So.empty() && So.top().prior(e.getOperator())) {
            if (auto op { So.pop() }; op != '(') {
                auto [l, r] { op.operandPosition() };
                T rhs { r ? Sr.pop() : 0 };
                T lhs { l ? Sr.pop() : 0 };
                Sr.push(op.apply(lhs, rhs));
            } else {
                break;
            }
        }
    } };
    for (auto& e : *this) {
        if (e.isOperand()) {
            Sr.push(e.getOperand());
        } else {
            process(e);
            if (e != ')') {
                So.push(e);
            }
        }
    }
    process(')');
    return Sr.pop();
}
```

在实验中提供了一些简单的例子来测试它的正确性，并使用大量的 1 相加来对算法的性能进行简单的测定。直接计算的常数会比通过后缀表达式间接计算低一些，但是差距非常小，几乎可以忽略不计。

第 4.5 节 栈与递归

栈和递归的关系非常紧密，因为调用递归函数本质上相当于使用了系统栈。具体的原理参见《操作系统》。系统栈的空间是有限的，如果递归层次过多，就会发生栈溢出（stack overflow）错误。为了避免这种情况发生，我们可以通过手写栈将递归改写为迭代形式。本节将介绍使用栈消除递归的方法。

第 4.5.1 节 消除多个实例的尾递归

在 第 2.7.2 节 我们介绍了消除尾递归的方法。尾递归因为每个递归实例只会在尾部调用一次自身，所以不需要使用栈。在本节中，我们将介绍一种尾递归的扩展形式：每个递归实例会在尾部调用不止一次自身。这样的递归函数通常没有返回值。

实验 **parenout.cpp**。在本节，我们选取一个简单的问题：输出所有合法的括号序列。输入 n，那么 n 对括号组成的序列一共有Catalan(n)个。这个问题的递归解法是这样的：在一个合法的括号序列中，左括号的数量大于等于右括号的数量，也就是说，剩余的左括号的数量小于等于剩余右括号的数量。作为递归参数，我们可以用两个变量 l 和 r 来表示剩余的左括号和右括号的数量。在递归的过程中，如果 l 大于 0，则可以加入一个左括号；如果 r 大于 l，则可以加入一个右括号。当 l 和 r 都等于 0 时，就得到了一个合法的括号序列。此算法可以以下列方法实现：

```
Vector<std::string> ans;
void search(std::string str, int l, int r) {
    if (l == 0 && r == 0) {
        ans.push_back(str);
        return;
    }
    if (l > 0) {
        search(str + '(', l - 1, r);
    }
    if (r > l) {
        search(str + ')', l, r - 1);
    }
}
```

这不是一个严格的多实例尾递归，因为在两次递归之间还包括了 if 的判断。对于这种情况，我们可以*将递归边界和递归调用分离*。比如，对于 search(str + '(', l - 1, r)这个递归调用，它的条件是 l > 0。如果我们不加条件判断直接进行递归调用，那么不满足 l > 0 的 l，再减 1 之后就变为了负值。所以，对于这个递归调用，我们只需要：

1. 将该调用的条件 l > 0 去掉，改为直接调用；
2. 增加一个递归边界的判断，在 l < 0 时参数不合法，直接返回。

用同一技术去处理另一个递归调用，就可以得到一个多实例尾递归的形式。如下所示：

```
void search(std::string str, int l, int r) {
    if (l < 0 || l > r) return;
```

```
    if (r == 0) {
        ans.push_back(str);
    }
    search(str + '(', l - 1, r);
    search(str + ')', l, r - 1);
}
```

我们调用 search("", n, n)来进行搜索。由于其满足多实例尾递归的形式，在函数的尾部只进行 2 次递归调用而不包含其他操作。下面说明这种性质带给我们的好处。多实例尾递归可以将每次调用的过程简化为“操作—调用 1—调用 2”。那么，在操作结束之后，我们可以立刻计算出调用 1 和调用 2 的参数，只需要存下来这两个调用的参数，当前调用就没有包括其他信息了，其占用的空间可以被释放掉。但如果不是多实例尾递归，比如“操作 1—调用 1—操作 2—调用 2”，那么在执行调用 1 的时候，就必须保留下来当前调用的信息，因为调用 1 完成之后还需要用这些信息来执行操作 2，需要更多的空间。这种一般形式的递归我们在下一节介绍。

回到“操作—调用 1—调用 2”的多实例尾递归问题上。在操作结束之后，当前调用可以被释放，但调用 1 和调用 2 的参数需要保存下来。这就和 第 2.7.2 节 中讨论的单实例尾递归不同。在单实例尾递归中，因为只有一个调用，那么新调用的参数可以直接存储在当前调用的参数位置上，从而不需要额外的空间，可以保证空间复杂度为$O(1)$。在多实例尾递归中，至多只有一个调用能存储在当前调用的参数位置上，而其他调用的参数需要额外的空间。假设一共有 m 次递归调用，则每递归深入一层，就要多存储 m-1 次调用的参数。所以，多实例尾递归的空间复杂度是$\Theta(h)$，其中h是递归的深度（在本节的例子中，空间复杂度的主要来源是存储返回值的 ans，但另一些情况下递归可能会变成空间复杂度的主要来源）。注意到，调用 2 必须等到调用 1 及其所有子调用结束之后才能进行，所以我们使用“后进先出”的栈来存储调用参数，需要先存储调用 2，再存储调用 1，也就是说使用栈的迭代形式入栈的次序，和递归形式的调用次序是相反的。

```
void search(std::size_t n) {
    Stack<std::pair<std::string, std::pair<int, int>>> S;
    S.push({"", {n, n}});
    while (!S.empty()) {
        auto [str, p] = S.pop();
        auto [l, r] = p;
        if (l < 0 || l > r) continue;
        if (r == 0) {
            ans.push_back(str);
        }
        S.push({str + ')', {l, r - 1}});
```

```
            S.push({str + '(', {l - 1, r}});
        }
    }
```

对于一般的多实例尾递归，其模板形式如下：

```
void recursive(Args... args) {
    if (is_base_case(args...)) {
        base_case(args...);
    } else {
        for (auto&& next : next_args(args...)) {
            std::apply(recursive, next);
        }
    }
}
```

对应地，其迭代形式如下：

```
void iterative(Args... args) {
    Stack<std::tuple<Args...>> S { { args... } };
    while (!S.empty()) {
        std::tie(args...) = S.pop();
        if (is_base_case(args...)) {
            base_case(args...);
        } else {
            for (auto&& next : next_args(args...)
                             | std::ranges::views::reverse) {
                S.push(std::move(next));
            }
        }
    }
}
```

【C++学习】

`std::tuple` 和 `std::pair` 非常相似，它可以用来存储三个或更多数量的元素，并使用 `std::get<N>`方法来访问第 `N` 个元素。这个方法可读性很差，平时应当更多地使用结构体或者类来代替 `std::tuple`。下面这种写法是极其不推荐的：

```
using MyStructure = std::tuple<int, double, std::string>;
constexpr auto field1 = 0, field2 = 1, field3 = 2;
```

应该用下面的方法代替：

```
struct MyStructure {
    int field1;
    double field2;
    std::string field3;
};
```

但是，`std::tuple` 在模板元编程中有更多的使用空间。假设我们调用了一个函数 `f(a, b, c)`。一方面，它可以被看成一个三元运算符 `f`，其三个操作数分别是 `a`, `b`, `c`。另一方面，它也可以被看成是一个二元运算符 `apply`（函数调用），其两个操作数分别是 `f`（函数名）和`(a, b, c)`（操作数的元组）。后者是更加通用的观点，其好处有两点：（1）将函数调用抽象出来作为 `apply`，而不依赖用户自己定义的标识符 `f`；（2）保证函数调用是二元运算符，而不是不确定的多元运算符。在一些能够直接操作语法树（如 Julia）的编程语言中往往采用这种方案。

在 C++中，我们可以使用 `std::tuple` 来处理任意数量参数的情况。比如，可以通过调用 `std::apply(f, std::tuple{a, b, c})`来替代 `f(a, b, c)`。`std::apply` 的第一个参数可以是普通函数、lambda 函数、仿函数对象等等，当调用非静态成员函数时，`std::tuple` 的第一个元素是调用的对象的引用（即`*this`）。此外，如果需要调用构造函数 `MyStructure(a, b, c)`来构造一个对象，则需要改为使用 `std::make_from_tuple<MyStructure>(std::tuple{a, b, c})`。

`std::tie` 是 `std::tuple` 的一个特殊用法，它可以用来解包元组，赋值给多个变量。比如，`std::tie(a, b, c) = std::tuple{1, 2, 3}`可以将 `1, 2, 3` 分别赋值给 `a, b, c`（类似 Python 的绑定赋值）。如果需要忽略元组中的某些元素，可以使用 `std::tie(std::ignore, b, c) = std::tuple{1, 2, 3}`可以将 `2, 3` 分别赋值给 `b, c`，而 `1` 被忽略掉了。

最后，`std::ranges::views::reverse`（需要`<ranges>`）可以反方向遍历一个范围。在本节的例子中，我们使用它来保证栈的迭代形式和递归形式的调用次序是相反的。关于范围（ranges）的技术本书涉及不多，感兴趣的读者可以参考 [1]。

需要再次指出的是，由于现代编译器的优化，一般情况下自己实现的基于栈的迭代形式，性能上比递归形式要低，二者大致相差一个常数。并且，递归形式通常具有更好的可读性。

第 4.5.2 节　消除多个带返回值的尾递归

实验 **comb.cpp**。对 第 4.5.1 节 介绍的尾递归扩展形式稍作改动，就可以用来计算一些有返回值的递归函数。这些函数在计算完每个递归调用的结果之后，对这些结果进行一个简单的处理，然后返回。本节以组合数计算为例介绍这种方法。

众所周知，组合数满足递归公式（杨辉三角）：

$$C_n^k = \begin{cases} 1, & k = 0 \\ 1, & k = n \\ C_{n-1}^k + C_{n-1}^{k-1}, & \text{otherwise} \end{cases}$$

其中$k=0$和$k=n$都是递归边界，其他情况下，将问题转化为两个子问题，即递归调用了两次，然后将这两次调用的结果相加。

```
int comb(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    } else {
        return comb(n - 1, k - 1) + comb(n - 1, k);
    }
}
```

这个函数非常直观。很明显这个函数不符合上一小节提到的多实例尾递归的形式，因为在递归调用之后还有一个加法操作，将两次递归调用的结果相加。但好在，加法操作是一个满足结合律的操作，所以我们可以将这个函数改写为多实例尾递归的形式，即：

```
int ans { 0 };
void comb(int n, int k) {
    if (k == 0 || k == n) {
        ans++;
        return;
    }
    comb(n - 1, k - 1);
    comb(n - 1, k);
}
```

这样，我们就可以使用上一小节的方法，对这个函数进行迭代形式改写。这一技术可以被应用到其他操作比较简单的递归函数上。但是我们可以很容易地发现，这个函数的时间复杂度高达$\Theta\left(C_n^k\right)$，可以用斯特林公式估计这个组合数的值，很容易发现它的增长速率非常高。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*，容易发现，造成这种现象的原因是其中包含了大量的*重复工作*。比如，$C_n^k=C_{n-1}^{k-1}+C_{n-1}^k=\left(C_{n-2}^{k-2}+C_{n-2}^{k-1}\right)+\left(C_{n-2}^{k-1}+C_{n-2}^k\right)$，可以看到，$C_{n-2}^{k-1}$被计算了两次。为了解决这个问题，我们可以在第一次计算出C_{n-2}^{k-1}之后，将它存储下来，下次需要的时候直接使用。这种方法称为**记忆化搜索**（memory search）。记忆化搜索的实现方法是，使用全局变量 `mem` 记录已经计算出的结果。在递归函数中，首先检查 `mem` 在对应的参数上是否已经有了这个结果，如果有，直接返回；如果没有，就计算出这个结果，并将它存入 `mem`。这样就可以避免重复计算。`mem` 通常是一个**字典**（dictionary）。具体而言，在本节的例子中是一个二维数组，`mem[n][k]`表示C_n^k的值。

```
std::vector<std::vector<int>> mem;
int comb(int n, int k) {
    if (mem[n][k] == 0) {
```

```
            mem[n][k] = comb(n - 1, k - 1) + comb(n - 1, k);
        }
        return mem[n][k];
    }
    int operator()(int n, int k) override {
        while (n >= mem.size()) {
            mem.push_back(std::vector<int>(mem.size() + 1, 0));
            mem.back()[0] = mem.back().back() = 1;
        }
        return comb(n, k);
    }
```

上述算法的空间复杂度和 mem 的定义方式有关。如果 mem 是一个$(n+1)\times(n+1)$的二维数组，则空间复杂度为$\Theta(n^2)$，而如果声明为一个$(n+1)\times(k+1)$的二维数组，则空间复杂度可以降低到$\Theta(nk)$。在不考虑 mem 初始化的情况下，时间复杂度为$\Theta(k(n-k))$。注意，像求组合数这样的函数通常会被反复调用，而 mem 是可以复用的。在一次计算组合数的时候得到的中间结果，可以被之后计算另一个组合数时复用。

如果忽略反复调用的潜力，我们可以想办法让 mem 的空间复杂度进一步降低。观察递归公式$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$可以发现，mem[n][]这一行的值，只依赖于 mem[n-1][]这一行，而不需要此前的行。所以我们可以用下面的方法去计算 mem[n][]：

```
std::vector<int> comb(int n) {
    if (n == 0) {
        return std::vector { 1 };
    } else {
        auto prev { comb(n - 1) };
        std::vector<int> ans(n + 1);
        ans[0] = ans[n] = 1;
        for (int i = 1; i < n; i++) {
            ans[i] = prev[i - 1] + prev[i];
        }
        return ans;
    }
}
```

这个方法中，整个 mem 数组被替换为一个一维数组 prev。在计算 mem[n][]的时候，我们只需要 prev 数组的内容，而不需要 mem[n-1][]之前的内容。这样，空间复杂度可以降低到$\Theta(n)$。这种方法称为**滚动数组**（rolling array）。

以上所有方法都是二维的，它在杨辉三角上沿着两个方向进行计算。但是，组合数这个问题除了递归定义之外还有直接定义，我们知道：

$$C_n^k = \frac{n!}{k!(n-k)!}$$

根据这个公式计算，在不考虑乘除法时间复杂度的情况下，时间复杂度为$\Theta(n)$。但我们可以注意到，在$n!$中包含了$(n-k)!$这一项，所以只需要计算剩余k项即可。即，利用下面的一维递归公式：

$$C_n^k = \frac{n-k+1}{k} \cdot C_n^{k-1}$$

使用这个公式可以在计算C_n^k的时候顺便计算出 `mem[n][0:k]`，其时间复杂度为$\Theta(k)$。其递归形式为：

```
int comb(int n, int k) {
    if (k == 0) {
        return 1;
    }
    return comb(n, k - 1) * static_cast<long long>(n - k + 1) / k;
}
```

【C++学习】

注意这里的计算结果是 `int` 类型，而乘法的中间结果可能会超出 `int` 的范围。所以我们在乘法之前将 `n-k+1` 转换为 `long long` 类型，以避免溢出。如果计算结果本身就是 `long long` 类型，那么乘法的中间结果完全有可能超过标准整数类型的位宽，需要使用 `boost::multiprecision` 或其他大数库来处理。

第 4.5.3 节 消除一般的递归

在前面两小节中，我们讨论了将递归边界和递归调用分离，从而将递归函数转化为尾部多次调用自身的形式。当然，仍然存在一些情况，无论如何都无法将递归函数转化为多实例尾递归。这种情况下，我们必须在最后一次调用完成之前都保留当前调用中的参数。下面展示了调用 2 次的一般递归形式：

```
void recursive(Args... args) {
    if (is_base_case(args...)) {
        return base_case(args...);
    } else {
        operation_1(args...);
        recursive(next_args_1(args...));
        operation_2(args...);
        recursive(next_args_2(args...));
        operation_3(args...);
    }
}
```

这种形式的递归无法被直接转换为多实例尾递归，但我们可以通过引入新变量**状态**（state）来间接将它转换为多实例尾递归。上述递归形式中，我们可以引入一个状态变量 `state`，用来表示当前递归调用的进度。在每次递归调用中，我们首先检查 `state`，然后根据 `state` 的值进行相应的操作，然后将 `state` 更新为下一个状态。这样，我们可以将递归函数转化为多实例尾递归的形式。下面是一个例子：

```
void recursive(int state, Args... args) {
    if (is_base_case(args...)) {
        return base_case(args...);
    } else {
        switch (state) {
            case 1: operation_1(args...); break;
            case 2: operation_2(args...); break;
            case 3: operation_3(args...); break;
        }
        if (state != 0) return;
        recursive(1, args...);
        recursive(0, next_args_1(args...));
        recursive(2, args...);
        recursive(0, next_args_2(args...));
        recursive(3, args...);
    }
}
```

这样，就将一般形式的递归转换为了多实例尾递归，从而可以将其很容易地转换为迭代形式。这种方法的缺点是，原先一个函数只会进行 2 次递归调用，修改之后变成了 5 次递归调用，导致空间消耗显著增加。一种解决方案是，状态切换之后的递归调用 `recursive(1|2|3, args...)`不在栈上开辟额外的空间，而是直接基于原先的递归调用 `recursive(0, args...)`上进行操作，操作结束之后转入下一个状态。这种方法称为**状态机**（state machine）。例如，上述例子可以转换为：

```
void iterative(Args... args) {
    Stack<std::tuple<int, Args...>> S;
    S.push({1, args...});
    while (!S.empty()) {
        auto& [state, args...] = S.top();
        if (is_base_case(args...)) {
            base_case(args...);
        } else {
            switch (state) {
                case 1: operation_1(args...); state = 2;
                        S.push({1, next_args_1(args...)}); break;
```

```
                case 2: operation_2(args...); state = 3;
                        S.push({1, next_args_2(args...)}); break;
                case 3: operation_3(args...);
                        S.pop(); break;
            }
        }
    }
}
```

上面的例子是一个只有 2 次递归调用的 `void` 函数。对于一般的情况，递归调用的返回值，以及函数中用到的一些局部变量，都需要存储在栈中，称为该次调用的**栈帧**（stack frame）。事实上，这也是递归写法实际被编译后的调用方法，只不过我们使用了私有栈代替了递归版本调用时使用的系统栈。

第 4.6 节 共享栈

当我们使用顺序栈的时候，我们为栈申请了一片连续内存作为存储空间。注意到，我们的栈只使用了前半部分的空间，而没有使用后半部分的空间。有一种高效利用空间的方法称为**共享栈**（shared stack），它由两个共享同一片连续内存的栈组成。其中一个栈使用前半部分，以秩为 0 的元素为栈底，以秩最大的元素为栈顶；另一个栈使用后半部分，以秩为 `n-1` 的元素为栈底，以秩最小的元素为栈顶。两个栈的栈顶“相向而行”。在大多数的软件开发过程中，由于空间相对来说是充足的，所以共享栈并不常见。但是在一些特殊的场合，共享栈可以发挥出它的优势。比如在一些嵌入式系统中，使用一个栈作为系统栈，另一个栈作为用户栈，可以有效地隔离系统和用户的内存空间。

测试程序位于 **`sstack.cpp`**。我们使用向量 `V`（而不是数组）作为实现共享栈的基础，以提供变长特性。使用两个变量 `topf` 和 `topb` 分别表示前向栈（栈底位于向量头部，栈顶向向量尾部移动）和后向栈（栈底位于向量尾部，栈顶向向量头部移动）的栈顶位置。预先分配好向量的空间，当向栈中插入元素时，只需要为栈顶的元素赋值并移动栈顶，而删除元素时直接移动栈顶即可。下面展示了后向栈的实现（其中，栈顶指针代表“下一个被插入元素在向量中的秩”），前向栈与其大体相同。

```
class BackwardStack : public AbstractStack<T> {
    SharedStack& S;
public:
    BackwardStack(SharedStack& s) : S(s) {}
    void push(const T& e) override {
        S.V[S.m_topb--] = e;
    }
    T& top() override {
```

```
        return S.V[S.m_topb + 1];
    }
    T pop() override {
        return std::move(S.V[++S.m_topb]);
    }
    std::size_t size() const override {
        return S.V.size() - 1 - S.m_topb;
    }
};
```

当两个栈顶相遇时，说明共享栈已满。在这种情况下，我们需要对向量进行扩容，扩容之后的新空间位于向量的尾部，无法被共享栈直接利用。此时，需要将后向栈整体移动到向量的尾部，把新空间置换到向量中部来，并更新后向栈的栈顶。

```
bool full() const {
    return V.size() == 0 || m_topb + 1 == m_topf;
}
void expand() {
    std::size_t oldsize { V.size() };
    V.resize(m_allocator(V.capacity(), V.size()));
    std::move_backward(V.begin() + m_topb + 1,
                       V.begin() + oldsize, V.end());
    m_topb += V.size() - oldsize;
}
```

在单个的基于向量的栈中，空闲的元素位于内存的尾部，从而可以利用向量本身的扩容机制进行扩容；而共享栈中，空闲的元素位于内存的中部，因此不使用向量的扩容机制。在共享栈里，两个指针 `topf` 和 `topb` 实现了和向量中的 `size` 同样的内存管理功能。

第 4.7 节 最小栈

栈对于用户可以访问的位置做出了很强的限制，有时我们希望略微放开这些限制，使得用户可以访问到一些更多的信息。**最小栈**（minimum stack）是其中的一个典型的例子。在最小栈中，除了栈本身所支持的三种基本操作之外，还支持 `min` 操作，返回当前栈内元素的最小值。要求在栈的三种基本操作的分摊复杂度保持$O(1)$的前提下，新增的 `min` 操作的分摊复杂度也为$O(1)$。和之前一样，这里仍然假定比较两个元素的操作时间复杂度为$O(1)$。

一个基本的思想是维护一个变量 `m` 来存储栈中的最小值。当插入的元素 `e < m` 的时候更新 `m` 没有问题，但当删除的元素 `e = m` 的时候，我们既不知道这个 `e` 是否是当前栈中唯一的最小值 `m`，又不知道当唯一最小值被删除之后，新的最小

值是多少。这就导致如果我们进行连续的删除操作，则需要反复地遍历整个栈来寻找新的最小值。这在两个方面不能达到我们的要求。

1. 时间复杂度上会存在问题。弹出元素的时候，时间复杂度高达$\Theta(n)$，而无法满足我们需求的$O(1)$。
2. 栈的性质上会存在问题。我们不应当访问栈顶以外的其他元素，即使在栈的实现内部也是如此。

为了在弹出元素的时候，能够立刻找到新的最小值，我们想到用空间换时间的方法。我们定义一个辅助线性表`Sm`，规定`Sm[k] = min(S[0], S[1], ..., S[k])`。因为栈`S`只能在栈顶进行操作，而`Sm[k] = min(Sm[k-1], S[k])`，所以`Sm`也只会在尾部进行操作，它同样是一个栈。

```
template <typename T>
class MinStack : public Stack<T> {
protected:
    Stack<T> Sm;
    void update() {
        if (Sm.empty()) {
            Sm.push(this->top());
        } else if (auto t { Sm.top() }; t < this->top()) {
            Sm.push(t);
        } else {
            Sm.push(this->top());
        }
    }
public:
    void push(const T& e) override {
        Stack<T>::push(e);
        update();
    }
    T pop() override {
        Sm.pop();
        return Stack<T>::pop();
    }
    const T& min() const {
        return Sm.top();
    }
};
```

上面的实现方式需要$\Theta(n)$的额外空间。我们可以发现，在辅助栈中存在大量重复的元素。比如，在栈`S=[3, 4, 5, 1, 2>`时，辅助栈`Sm = [3, 3, 3, 1, 1>`，其中 3 被重复了 3 次，1 被重复了 2 次。因此，直觉上看存在空间复杂度并非最优的可能性。为了判断是否已经取得了最优的空间复杂度，我们需要对辅助栈进

行分析。考虑一个略简单的模型，即栈内的元素互不相等的情况：设栈中的元素为 1, 2, ..., n。我们对辅助栈和栈操作序列建立一个一一对应。

1. 补充定义 Sm[-1] = n+1。对于 S[j]，如果 S[j]≠ S[j-1]，则将 S[j]替换为$\vee^k \wedge$，其中 k = S[j-1] - S[j]。
2. 如果 S[j] = S[j-1]，则将 S[j]替换为∧。

这样，我们就得到了一个操作序列，因此，辅助栈的所有可能情况数量为Catalan(n)。从信息论的角度看，我们需要至少$\log(\text{Catalan}(n)) = \Theta(n)$的空间才能表示这些情况（在允许元素相等的时候，情况只会更多）。因此，无论采用如何的方式存储辅助栈，在最坏情况下需要的额外空间复杂度总是$\Theta(n)$。

第 4.8 节 本章习题

在 第 4.1 节 中：

1. 简单 仿照 第 3.7 节 列出一个表格，比较顺序栈和链栈的优缺点。

在 第 4.2 节 中：

1. 简单 设入栈序列为 1234，则可能的出栈序列有哪些？
2. 较难 证明：如果入栈序列是$(1, 2, ..., n)$，则$B = (b_1, b_2, ..., b_n)$是出栈序列，当且仅当不存在“312”模式。即：不存在$i < j < k$，使得$b_j < b_k < b_i$。
3. 中等 如果令$\vee = 0, \wedge = 1$，那么就可以将一个操作序列转换为为 2n 位的二进制数。借助二进制数的大小关系，我们可以比较两个操作序列，这种比较方法称为**字典序**。给定一个合法操作序列，设计一个算法，求字典序中的下一个合法操作序列。
4. 中等 给定一个合法操作序列，设计一个算法，求它在字典序中的排名。

在 第 4.3 节 中：

1. 简单 括号匹配算法是减治算法还是分治算法？
2. 简单 设计一个算法，解决只有一种括号的匹配问题。
3. 中等 正则文法是指产生式满足下列形式的文法：$A \to a$或$A \to aB$，其中A, B是非终结符，a是终结符。满足正则文法的字符串可以用正则表达式（regex）或有限状态机（FSM）判别。合法的括号序列是否是正则文法？如果是，给出相应的产生式。
4. 较难 如果限定括号序列中的嵌套层数不超过 k，其中 k 是一个常数，那么括号序列是否是正则文法？如果是，给出相应的产生式。

在 第 4.4 节 中：

1. 中等 证明 第 4.4.6 节 中的方法可以得到倒序的前缀表达式。
2. 简单 区分负号和减号的另一种思路是，在第一次遍历字符串的时候将负号替换为另一个符号，比如~。如何实现这一思路？

3. 中等 有一些运算符的字符数可能不止一个。在本书的代码基础上，增加对Python风格的乘方运算符**的支持。
4. 中等 有一些运算符的字符数可能是不确定的。定义双阶乘$n!! = n(n-2)(n-4)...$，三阶乘$n!!! = n(n-3)(n-6)...$，以此类推。在本书的代码基础上，增加对这种运算符的支持。
5. 简单 左操作数和右操作数的区分是中缀表达式的一个特点，而在后缀表达式中并不关心操作数的位置。修改本书代码中对于`operandPosition`的处理，使我们只需要关心操作数的数量即可。
6. 较难 对于最多二元的运算符来说，如果使用合取方法来定义表达式中的元素，则 第 4.4.8 节 中的计算算法可以得到大幅度简化：因为表达式里的每个元素都是运算符，而操作数是附带在前方最近的运算符上的。写出这种简化后的计算算法。
7. 较难 在本书的代码基础上，增加对三元运算符`?:`的支持（`A ? B : C`表示，如果`A`不为0则返回`B`的计算结果，如果`A`为0则返回`C`的计算结果；在一些陈旧的编译器上，`?:`运算符不会触发分支预测，从而性能比`if-else`低，现代编译器已基本无影响）。
8. 较难 常规的运算符几乎都是左结合的，但我们也可以加入一些右结合运算符。一个比较常见的右结合运算符是↑，即高德纳（Knuth）箭头，定义$a \uparrow b = a^b$，同时$a \uparrow b \uparrow c = a \uparrow (b \uparrow c)$；因此，我们让本书中的乘方运算符^改为右结合是非常合理的做法。在本书的代码基础上，增加对右结合运算符的支持。

在 第 4.5.1 节 中：

1. 简单 给定`n`个操作数和`n-1`个二元运算符，设计一个算法，输出所有合法的后缀表达式。
2. 中等 设计一个算法，给定`n`个操作数和目标值`t`，输出用这个`n`个操作数和四则运算符计算出`t`的所有后缀表达式。

在 第 4.5.2 节 中：

1. 简单 证明二维递归算法的时间复杂度是$\Theta\left(C_n^k\right)$。
2. 简单 证明二维记忆化搜索算法的时间复杂度是$\Theta(k(n-k))$。
3. 简单 修改滚动数组的实现，使得它不需要`prev`和`ans`两个数组，而只需要一个数组。然后将其改写为迭代形式。
4. 简单 在一维组合数算法中，即使$k = n$也不会直接返回1，而是继续递归，这会造成相对低效。修改算法使得时间复杂度被降低到$\Theta(\min(k, n-k))$。

在 第 4.6 节 中：

1. 简单 示例实现中栈顶指针是下一个被插入元素在向量中的位置。修改共享栈的实现，使得栈顶指针表示栈顶元素在向量中的位置。

2. 较难 如果在一个向量中维护多于 2 个的栈，则它们无法像共享栈那样很方便地，有固定的栈底和移动的栈顶。假设我们有一个容量充分大的向量 `V` 存储 `n` 个栈$S_1, S_2, ..., S_n$，所有的栈紧挨着存储。初始时所有的栈顶都位于 `V[0]`。当向栈S_i插入一个元素时，它的栈顶向后移动一个位置，同时让$S_{i+1}, S_{i+2}, ..., S_n$之后的栈的所有元素及其栈顶都向后移动一个位置。这当然是一个很低效的方法。假设对于每次插入，插入到每个栈的概率相同，求从起始状态开始，连续插入 `m` 个元素造成的平均移动次数。
3. 挑战 在第 2 题的基础上，如果插入到每个栈的概率不同，分别为$p_1, p_2, ..., p_n$，则插入 `m` 个元素造成的平均移动次数是多少？利用不对称性改进算法，改变栈的顺序，使得平均移动次数最小。

在 第 4.7 节 中：

1. 简单 证明辅助栈的可能性和栈操作序列确实满足正文中构造的一一对应。
2. 简单 证明$\log(\text{Catalan}(n)) = \Theta(n)$。
3. 较难 尽管看上去辅助栈的做法需要的额外空间为$\Theta(n)$，但如果栈中的元素占用的空间较大（比如是一个结构体），辅助栈需要的空间也会相应增加。这是由于辅助栈中存储了和原栈中的重复元素。如果辅助栈中存储的是原栈中的指针而不是元素本身，在这种情况下辅助栈的空间可以得到减小，然而这样做会使得空间复杂度变为$\Theta(n \log n)$。设计一种最小栈，使得辅助空间的复杂度严格保持在$\Theta(n)$。

第 4.9 节 本章小结

作为适配器的栈本身非常简单，无论采用顺序栈还是链栈进行实现都没有难度。栈能做到的事情线性表（向量或列表）都能做到，所以初学者很容易无法理解，为什么需要给自己设置限制使用栈去处理问题，而不使用向量或列表来处理问题。事实上，这和 OOP 中将一些方法设置为私有的原因是一样的。通过对使用者的访问权限加以限制，我们的目光被聚焦了，更容易抓住解决问题的要点。本质上这是一种增加额外条件从而简化问题的策略。

栈的操作非常简单，这也就意味着，将其他复杂的问题（比如括号匹配、表达式计算）转化为栈的问题，就可以让问题得到大幅度的简化，从而更容易解决。这种思路在计算机科学中非常常见，比如在编译原理中，将复杂的语法分析问题转化为栈的问题，从而使用栈的特性来解决问题。本章的主要学习目标如下：

1. 您理解了栈操作序列的性质，在遇到相似性质的场景时可以联想到用栈解决。
2. 您了解到出栈序列的计数是卡特兰数，在遇到相似的递归方程时可以联想到用栈解决。
3. 您掌握了将一般性的递归转化为多实例尾递归，从而转化为迭代形式的方法。

参考文献

[1] M. Gregoire, *Professional C++*. John Wiley & Sons, 2021.

[2] 吴文虎, 徐明星, and 邬晓钧, 程序设计基础（第 *4* 版）. 清华大学出版社, 2017.

[3] 郑莉 and 董渊, *C++*语言程序设计（第 *5* 版）. 清华大学出版社, 2020.

[4] 邓俊辉, 数据结构*: C++* 语言版. 清华大学出版社, 2013.

[5] D. E. Knuth, "The art of computer programming, vol 1: Fundamental," *Algorithms. Reading, MA: Addison-Wesley*, 1968.

[6] 严蔚敏 and 吴伟民, 数据结构*: C* 语言版. 清华大学出版社, 1997.

[7] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Transactions on software Engineering*, no. 2, pp. 223–228, 1981.

[8] P. Blackburn and W. Meyer-Viol, "Linguistics, logic and finite trees," *Logic Journal of the IGPL*, vol. 2, no. 1, pp. 3–29, 1994.

[9] 王红梅, 胡明, and 王涛, 数据结构 *(C++* 版*)*. 清华大学出版社, 2005.

[10] 彭波, 数据结构. 清华大学出版社, 2002.

[11] 鞠文飞, "多快好省, 改造 SOHO 路由器," 网管员世界, no. 10, pp. 108–110, 2007.

[12] R. Kowalski, "Algorithm= logic+ control," *Communications of the ACM*, vol. 22, no. 7, pp. 424–436, 1979.

[13] D. E. Knuth, "The art of computer programming, Vol 2: Seminumerical Algorithms." Addison-Wesley Professional, 1997.

[14] D. Harvey and J. van der Hoeven, "Integer multiplication in time $O(n \log n)$", *Annals of Mathematics*, vol. 193, pp. 563–617, 2021.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[16] D. E. Knuth, *The art of computer programming*, vol. 3. Pearson Education, 1997.

[17] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980.

[18] G. van Rossum, "Tail Recursion Elimination," *Neopythonic. blogspot. be. Retrieved*, vol. 3, 2012.

[19] 全国科学技术名词审定委员会审定, 计算机科学技术名词（第三版）. 科学出版社, 2018.