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$A(k_j)$，其中$k_j < n$。

则$\forall n, A(n)$可在有限时间输出正确结果。

经典递降法的正确性由经典归纳法保证。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*。

```cpp
// 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$A_0+A_1+...+A_{n-1}$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。

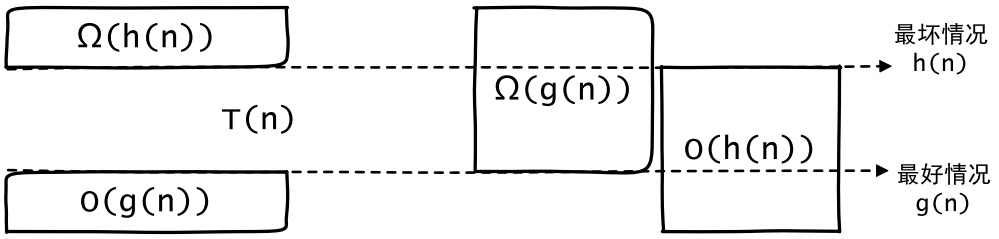

图 1.1　最好情况和最坏情况的评估

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。

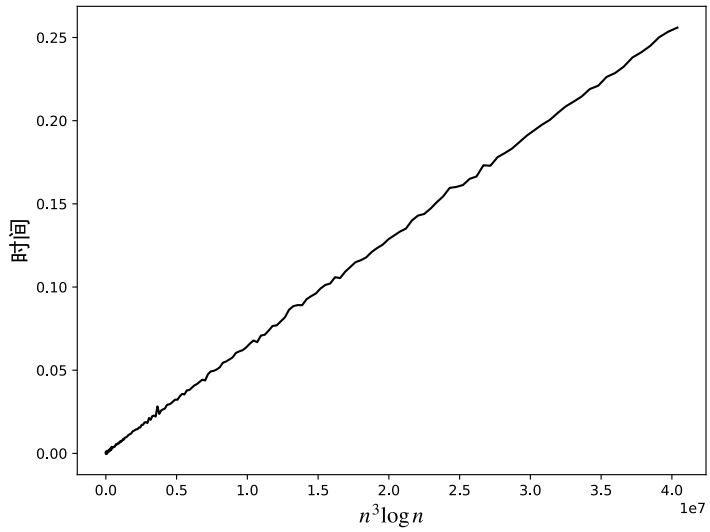

图 1.2 复杂度验证结果

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。

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

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

在 第 1.3.3 节 中：

1. SumBasic 的时间复杂度是多少？
2. SumAP 和 SumAP2 这两种实现，各自在n处于什么区间时可以输出正确的值？
3. 如果它们的输入输出范围不是 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. 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. 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时间复杂度和空间复杂度。
3. 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$O(m \log m)$。
4. 将中华更相减损术转换为递归算法，并分析其空间复杂度。

在 第 1.4.4 节 中：

1. 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. 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. 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. 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. 本节所述的二分算法属于分治还是减治？为什么？
2. 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。
3. 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？

在 第 1.5.1 节 中：

1. 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. 证明$\log n = O(n^c)$对一切$c > 0$成立。
3. 证明$n^c = O(a^n)$对一切$c > 0, a > 1$成立。
4. 在现代的通用个人计算机上，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. 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. 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. 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. 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. 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. `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. 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. 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. 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），又应该如何设计算法呢？

在 第 1.5.4 节 中：

1. 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. 将 PowerFastRecursive 转换为对应的迭代形式。
3. 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：

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。

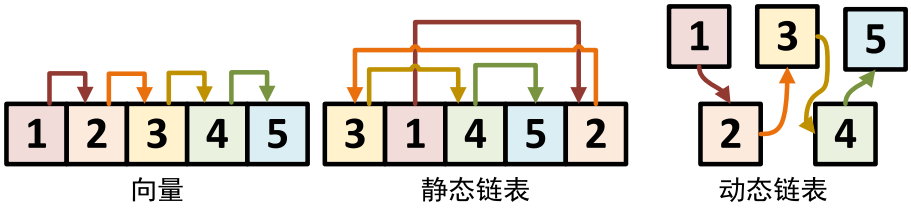

图 2.1　向量、静态链表和动态链表的对比

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。

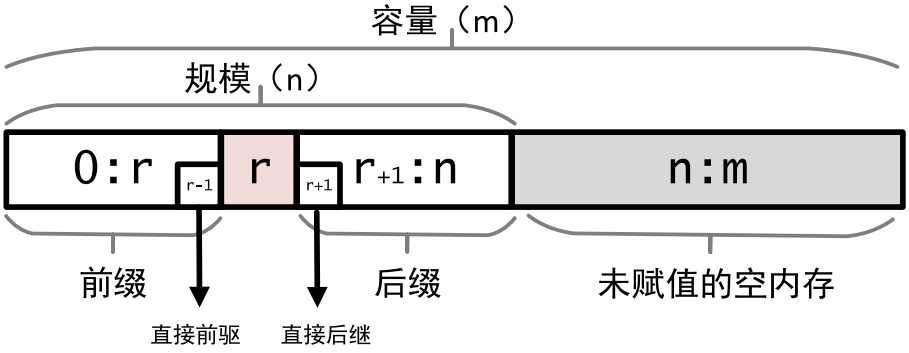

图 2.2　向量、静态链表和动态链表的对比

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）。它还缺少下面这些组件的实现：获取规模、容量和首地址的方法；修改规模和容量的方法；插入、删除、查找的方法。如果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。

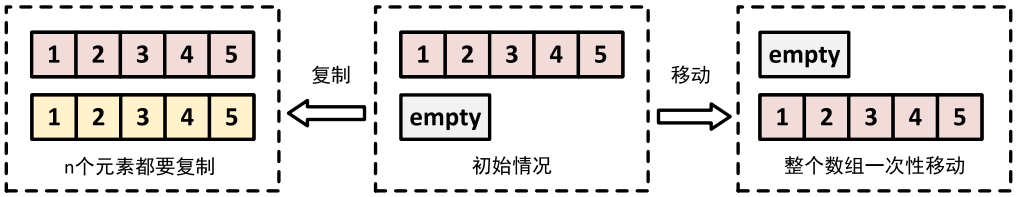

图 2.3 复制和移动的语义区别

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;
    }
}
```

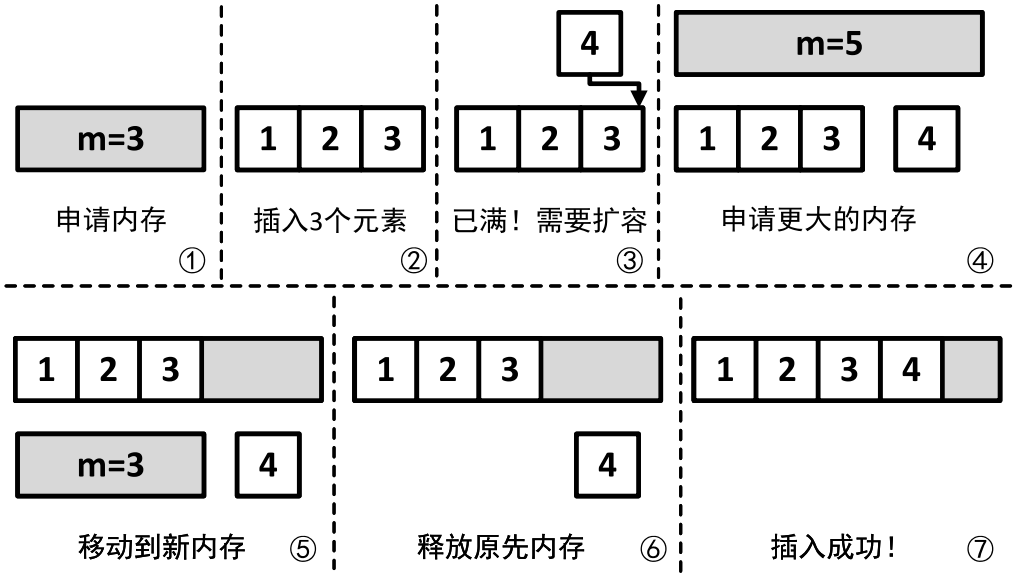

图 2.4 扩容的过程

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。

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.