第 1 章
数据结构与算法

本章概述

数据结构与算法是计算机科学的核心基础。本章主要介绍数据结构与算法的基本概念及相关知识，并介绍如何进行算法分析，概述数据结构与算法之间的关系。为后续学习算法知识打下坚实的基础。

知识导读

本章要点（已掌握的在方框中打钩）

- ☐ 数据结构的特性
- ☐ 数据结构的逻辑结构
- ☐ 数据结构的存储结构
- ☐ 算法的特性
- ☐ 算法的复杂度分析
- ☐ 数据结构与算法的关系

1.1 数据结构

数据结构（Data Structure）是程序的骨架，决定了数据的组织与操作效率。从数组到图论，它构建了数字世界的底层逻辑，是算法优化与系统设计的核心基础。掌握数据结构，方能驾驭计算之美。

1.1.1 什么是数据结构

数据结构是计算机存储、组织和管理数据的方式，它研究数据元素之间的逻辑关系、物理存储结构以及相关操作。作为计算机科学的基石，数据结构直接影响程序的运行效率、资源消耗和可维护性。

从逻辑角度看，数据结构可分为线性结构（如数组、链表以及衍生出来的栈、队列）和非

线性结构（如树、图）。物理实现上则涉及顺序存储（数组）和链式存储（链表）等方式。每种结构都有其特性和适用场景：数组适合快速访问，链表便于动态修改，树结构能高效处理层次关系，图结构擅长表达复杂关联。

优秀的数据结构设计能显著提升算法效率，实际应用中，操作系统使用队列管理进程，编译器借助栈处理函数调用，社交网络依赖图结构建立用户关系。

掌握数据结构是编写高效程序的关键，它架起了抽象问题与具体实现之间的桥梁，是算法设计和系统优化的核心基础。

1.1.2 数据结构的发展

数据结构的发展与计算机科学的进步密不可分，从早期的简单存储方式到现代高效的数据组织方法，其演变过程反映了计算需求的增长和算法的优化。以下是数据结构发展的主要阶段及关键里程碑：

1. 第一阶段：基础数据结构的形成（1940—1960 年）

这一阶段主要确立了最基本的数据结构形式。

（1）数组：最早出现的数据结构之一，用于顺序存储数据。

（2）链表：1955 年由 Allen Newell 等人提出，解决了数组固定大小的问题。

（3）栈和队列：在编译原理和系统程序设计中得到广泛应用。

2. 第二阶段：非线性结构的兴起（1960—1980 年）

随着计算需求的复杂化，更高级的数据结构应运而生。

（1）树结构：二叉搜索树（1960 年）、AVL 树（1962 年）、B 树（1972 年）。

（2）图结构：用于表示复杂关系，支持最短路径等算法。

（3）哈希表：实现快速查找，时间复杂度接近 O(1)。

3. 第三阶段：高效数据结构的优化（1980—2000 年）

这一阶段着重提升数据结构的性能。

（1）红黑树（1978 年）：平衡二叉树的优化实现。

（2）堆结构：支持高效的优先级队列操作。

（3）跳表（1989 年）：替代平衡树的简单高效结构。

4. 第四阶段：现代发展阶段（2000 年至今）

面对大数据和分布式计算的新挑战。

（1）布隆过滤器：空间效率极高的概率数据结构。

（2）LSM 树：支持高效写入的日志结构存储。

（3）分布式数据结构：如 CRDTs（无冲突复制数据类型）。

（4）持久化数据结构：支持版本控制。

5. 未来发展趋势

（1）量子数据结构：适应量子计算的新范式。

（2）学习型索引：利用机器学习优化数据访问。

（3）存算一体结构：突破冯·诺依曼架构限制。

6. 数据结构的发展

（1）从简单到复杂。

（2）从单一到多样。

（3）从集中式到分布式。

（4）从确定性到概率性。

（5）从静态到动态自适应。

理解数据结构的发展历程，有助于我们把握技术演进的内在逻辑，在面对新问题时能够选择或设计最合适的数据结构解决方案。

1.1.3　数据基本特性

数据是算法和数据结构处理的核心对象，理解数据的基本特性是设计高效算法和选择合适数据结构的基础。本节将详细讨论数据的四个关键特性，这些特性直接影响算法的时间复杂度、空间复杂度以及实现方式的选择。

1. 数据规模（Data Scale）

数据规模指数据元素的数量大小，通常用 n 表示。在算法分析中，数据规模直接影响以下三个方面。

（1）时间复杂度：不同算法对数据规模的敏感度不同。

（2）空间复杂度：大规模数据需要考虑内存限制。

（3）算法选择：小规模数据可能适用简单算法，大规模数据需要更高效的算法。

2. 数据分布（Data Distribution）

数据分布描述数据元素的排列特征和统计特性，包括以下几种。

（1）均匀分布：数据元素均匀分布。

（2）偏态分布：数据集中在某一范围。

（3）有序 / 无序：数据是否已排序。

（4）随机性：数据是否随机生成。

分布特性会影响搜索算法的效率、排序算法的选择和哈希函数的设计。

3. 数据类型（Data Type）

数据类型决定数据的存储方式和操作规则，数据类型包括以下几种。

（1）基本类型：整型、浮点型、字符型等。

（2）复合类型：结构体、对象等。

（3）特殊类型：字符串、多维数据等。

类型特性将会影响内存占用、操作复杂度和算法实现方式。

4. 数据关系（Data Relationship）

数据元素间的相互关系包括以下几种。

（1）线性关系：顺序、前驱后继。

（2）层次关系：父子、包含。

（3）网状关系：图结构。

（4）无明确关系：集合。

关系特性影响数据结构的选择、遍历方式、操作复杂度和特性综合应用。

理解数据的基本特性是算法设计和性能优化的第一步，后续章节将基于这些特性展开对各种数据结构和算法的详细讨论。

1.1.4 数据的逻辑结构和存储结构

1. 逻辑结构

数据的逻辑结构描述了数据元素之间的抽象关系，与计算机存储方式无关，是算法设计的理论基础。以下是四种核心逻辑结构及其 C++ 实现示例：

1）集合（Set）

特点：数据元素属于同一整体，但无明确关系。

C++ 示例：实现数学中的集合 {1，2，3}，代码如下所示。

```
#include <unordered_set>                    // 引入无序集合的标准库头文件
unordered_set<int> s = {1, 2, 3};           // 定义并初始化集合
```

2）线性结构（Linear）

特点：元素之间存在一对一顺序关系。

常见类型：数组、链表、栈、队列。

C++ 示例：双向链表，代码如下所示。

```
#include <list>                             // 引入 STL 双向链表容器
list<int> linear = {1, 2, 3};               // 双向链表
```

3）树结构

特点：元素间存在一对多层关系。

常见类型：二叉树、B 树、堆。

C++ 示例：二叉树结点，代码如下所示。

```
struct TreeNode {
    int val;                                // 结点存储的整数值
    TreeNode* left;                         // 左子结点指针
    TreeNode* right;                        // 右子结点指针
    TreeNode(int x)                         // 构造函数，初始化结点
        : val(x),                           // 设置结点值
          left(nullptr),                    // 左子结点初始化为空指针
          right(nullptr) {}                 // 右子结点初始化为空指针
};
```

4）图结构（Graph）

特点：元素间存在多对多任意关系。

常见类型：有向图、无向图、带权图。

C++ 示例：邻接表表示，代码如下所示。

```
vector< vector<int>> graph = {
    {1, 2},                                 // 结点 0 的邻居
```

```
    {0, 3},                          // 结点 1 的邻居
    {0}                              // 结点 2 的邻居
};
```

2. 存储结构

数据的存储结构（也称为物理结构）是指数据在计算机内存中的实际存储方式，它决定了数据如何被访问、修改和存储。以下是常见的数据存储结构及其特点，结合 C++ 示例说明。

1）顺序存储结构（Sequential Storage）

（1）数据元素存储在连续的内存单元中。

（2）通过下标 / 偏移量直接访问（随机访问高效）。

（3）插入 / 删除可能需要移动大量元素。

C++ 示例，代码如下所示。

```
// 使用数组或 vector 实现
int arr[5] = {1, 2, 3, 4, 5};     // 静态数组
vector<int> vec = {1, 2, 3};      // 动态数组
```

2）链式存储结构（Linked Storage）

（1）数据元素存储在非连续的内存块中，通过指针链接。

（2）插入 / 删除高效（O(1)），但访问需遍历（O(n)）。

（3）每个结点需额外空间存储指针。

C++ 示例，代码如下所示。

```
// 单向链表结点
struct ListNode {
    int val;                // 存储结点的值
    ListNode* next;         // 指向下一个结点的指针
    ListNode(int x)         // 构造函数，初始化结点
        : val(x),           // 设置结点值
          next(nullptr) {}  // 初始化 next 为空指针
};
// 双向链表结点
struct DoublyListNode {
    int val;                // 存储结点的值
    DoublyListNode *prev;   // 指向前一个结点的指针
    DoublyListNode *next;   // 指向下一个结点的指针
};
```

1.1.5　数据结构的研究对象

数据结构的研究对象主要包括以下几个方面：

1. 数据元素间的逻辑关系

（1）线性结构：数据元素之间存在一对一的关系，如数组、链表、栈、队列等。

（2）非线性结构：数据元素之间存在一对多或多对多的关系，如树（层次结构）、图（网状结构）等。

2. 数据的存储结构（物理结构）

（1）顺序存储：数据元素存储在连续的存储单元中（如数组），支持随机访问但插入 / 删除效率低。

（2）链式存储：通过指针或引用链接数据元素（如链表），插入 / 删除高效但访问需遍历。

3. 数据的操作与算法

（1）基本操作：增删改查（CRUD），如链表的插入。

（2）高级操作：排序、遍历等。

（3）效率分析：通过时间复杂度和空间复杂度衡量算法性能。

4. 特殊数据结构的应用场景

（1）树结构：层次化数据管理（如文件系统、数据库索引）。

（2）图结构：建模网络关系（如社交网络、路径规划）。

1.2 算法

算法（Algorithm）是解决特定问题的一系列明确、有限、可执行的步骤或指令集合。它是计算机科学的核心基础，用于描述如何通过计算或数据处理完成任务。

1.2.1 什么是算法

算法是解决特定问题的一系列明确、有限的步骤，是计算机科学的核心基础。它具有以下特点：明确性（每一步清晰无歧义）、有限性（在有限步骤内完成）、输入（接收初始数据）、输出（产生结果）和有效性（步骤可行且能在有限时间内完成）。算法可以用于各种任务，如排序、搜索、路径规划等，其设计目标包括高效性（时间与空间复杂度低）、正确性（准确解决问题）和通用性（适用于多种场景）。常见的算法类型包括分治法、动态规划、贪心算法等。通过算法分析（如时间复杂度、空间复杂度），可以评估其性能并优化设计。算法不仅是编程的基础，也是解决实际问题的关键工具。

1.2.2 算法的特性

算法是计算机科学的核心基础，其设计必须满足以下五大基本特性，以确保其正确性和有效性：

1. 有穷性（Finiteness）

（1）定义：算法必须在“有限步骤”后终止，不能无限循环或永不停止。

（2）重要性：避免程序陷入死循环，确保问题可解。

代码示例如下所示。

```
// 有限步骤：计算 1 到 n 的和
int sum = 0;
for (int i = 1; i <= n; i++) {  // 循环 n 次后终止
    sum += i;
}
```

2. 确定性（Definiteness）

（1）定义：算法的每个步骤必须明确无歧义，不允许出现“可能”“大概”等模糊描述。

（2）重要性：确保不同人实现同一算法结果一致。

代码示例如下所示。

```
明确：若 x > 0，则 y = 1；否则 y = 0
模糊：若 x 较大，则 y 增加一些（" 较大 " 和 " 一些 " 无明确标准）
```

3. 可行性（Effectiveness）

（1）定义：每个步骤必须能通过基本操作（如加减、比较、赋值）在有限时间内完成。

（2）重要性：排除理论上可行但实际无法实现的操作。

代码示例如下所示。

```
a = b + c;  // 基本加法操作，可行
```

4. 输入（Input）

（1）定义：算法有零个或多个输入，这些输入是问题的初始数据。

（2）重要性：明确算法的处理对象。

（3）示例：

排序算法的输入：待排序的数组（如 [3,1,2]）。

计算圆周率的算法：可能无输入（直接输出 π 的近似值）。

5. 输出（Output）

（1）定义：算法必须产生至少一个输出，作为问题的解。

（2）重要性：验证算法是否解决问题。

（3）示例：

排序算法的输出：有序数组（如 [1,2,3]）。

搜索算法的输出：目标值的下标或“未找到”。

1.2.3　算法的时间复杂度和空间复杂度

1. 时间复杂度

算法时间复杂度是衡量算法执行效率的重要指标，它描述了算法运行时间随输入规模增长的变化趋势。时间复杂度表示算法执行所需时间与输入规模 n 之间的关系，通常用大 O 符号（如 O(n)）表示，描述最坏情况下的增长趋势。

常见的时间复杂度类型如表 1-1 所示。

表 1-1　常见时间复杂度类型

复杂度类型	示例场景	性能对比（效率从高到低）
O(1)	数组随机访问	最优，执行时间与 n 无关
O(log n)	二分查找	次优，常见于分治算法
O(n)	线性遍历数组	线性增长，如 MetaQ 算法优化案例
$O(n^2)$	冒泡排序	效率较低，n 较大时性能骤降
$O(2^n)$	穷举密码破解	极低效，如 8 字符密码需 62 年破解

在实际应用中，可以通过以下几种方式来优化时间复杂度。

（1）空间换时间：使用哈希表减少查找时间；

（2）分治策略：将问题分解为更小的子问题；

（3）动态规划：存储中间结果避免重复计算；

（4）贪心算法：局部最优解可能导向全局最优；

（5）剪枝策略：提前终止不必要的计算。

2. 空间复杂度

算法的存储空间需求是指算法在运行过程中所需占用的内存空间，通常用空间复杂度来衡量。空间复杂度描述了算法所需存储空间与问题规模之间的关系，通常用大 O 符号（如 O(n)）表示。

算法存储空间主要包括以下部分：

（1）输入数据所占空间：存储算法输入数据所需的内存空间；

（2）程序本身所占空间：存储算法代码和指令所需的内存空间；

（3）辅助变量所占空间：算法运行过程中临时变量、递归栈等所需的内存空间。

1.2.4　算法的描述方法

算法的描述方法是指如何清晰地表达算法的步骤和逻辑，以便于理解和实现。常见的描述方法包括自然语言描述、流程图描述、伪代码描述和程序代码描述。

1. 自然语言描述

用日常语言（如中文、英文）描述算法的步骤和逻辑。

2. 流程图描述

用图形化的方式描述算法的流程。流程图通过标准化的图形符号（如开始 / 结束框、处理框、判断框等）直观地展示算法的逻辑结构和执行顺序，适合可视化分析。

3. 伪代码描述

用介于自然语言和编程语言之间的形式描述算法。伪代码结合了语言的简洁性和结构的清晰性，避免了编程语言的语法细节，适合算法设计阶段使用。

4. 程序代码描述

用具体的编程语言（如 C++、Python）描述算法。这种方法可以直接运行和测试，但可能会受限于特定语言的语法和特性，不易于跨语言理解。

1.3　算法分析

算法分析是对一个算法需要多少计算时间和存储空间作定量的分析。算法（Algorithm）是解题的步骤，可以把算法定义成某一确定类问题的任意一种特殊的方法。在计算机科学中，算法要用计算机算法语言描述，算法代表用计算机解一类问题的精确、有效的方法。

1.3.1　算法的复杂度分析

算法复杂度主要分为时间复杂度和空间复杂度。一个好算法的评判标准是高效率和低存储，高效率为时间复杂率小，低存储为空间复杂度小。

1. 时间复杂度

时间复杂度是衡量算法执行时间随规模 n 的增长趋势。

下面计算嵌套循环的时间复杂度，代码如下所示。

```
for (int i = 0; i < n; i++) {              // 外层循环 n 次
    for (int j = 0; j < n; j++) {          // 内层循环 n 次
        cout << i << " " << j;             // 基本操作 1 次
    }
}
```

在上述代码中，“外层循环”的执行次数为：i 从 0 到 n−1，共 n 次，时间复杂度为 O(n)；“内层循环”的执行次数为：对于每一个 i，j 从 0 到 n−1，共 n 次，时间复杂度为 O(n)；“基本操作”是常数时间操作 O(1)。

总的时间复杂度计算：外层执行 n 次，每次外层循环触发一个 O(n) 的内层循环，内层循环的 O(1) 操作执行 n×n 次。总执行次数 =n(外层)×n(内层)×1(基本操作)=n^2，即最终时间复杂度为 O(n^2)。

2. 空间复杂度

衡量算法运行所需的额外存储空间随 n 的增长趋势。算法占用的存储空间为输入输出数据、算法本身和额外需要的存储空间。

下面计算递归斐波那契数列的空间复杂度，代码如下所示。

```
int fib(int n) {
    if (n <= 1) return n;                  // 递归终止条件
    return fib(n-1) + fib(n-2);            // 递归调用
}
```

在上述代码中，每次调用会分裂两个子调用 fib(n−1) 和 fib(n−2)，递归深度为 n，形成二叉树结构，总调用次数为二叉树结点数 =2^n，则最终的空间复杂度为 O(2^n)。

1.3.2 常用的算法设计方法

算法设计是计算机科学的核心内容和解决问题的系统化策略，不同的方法适用于不同的问题类型，合理选择算法设计方法可以显著提高问题求解效率。以下是常见的算法设计方法及其典型应用：

1. 分治法

（1）核心思想：分治法是将大规模的问题分解为多个互相独立的子问题，递归求解后合并结果。

（2）特点：子问题与原问题结构相同，子问题相互独立；必须有递归终止条件。

（3）典型应用：归并排序、快速排序。

2. 动态规划

（1）核心思想：将问题分解为重叠子问题，存储中间结果避免重复计算。

（2）特点：最优子结构；重叠子问题；自底向上或记忆化递归。

（3）典型应用：背包问题、最长公共子序列、最短路径问题等。

3. 贪心算法

（1）核心思想：每一步选择当前局部最优解，期望最终得到全局最优解。

（2）特点：局部最优选择；不能回退；需要证明正确性。

（3）典型应用：哈夫曼编码、最小生成树（Prim/Kruskal）、任务调度等。

4. 回溯法

（1）核心思想：通过深度优先搜索尝试所有可能的解，并在发现不满足条件时回退（剪枝）。

（2）特点：系统性搜索；能回退；通常用递归实现。

（3）典型应用：N 皇后问题、数独求解、组合问题等。

5. 分支限界法

（1）核心思想：在回溯法基础上，通过优先级队列和界限函数减少搜索空间，常用于优化问题。

（2）特点：使用优先级队列管理结点；估算界限值进行剪枝；适合优化问题。

（3）典型应用：旅行商问题、整数规划、任务分配等。

6. 随机化算法

（1）核心思想：引入随机因素提高算法效率或简化实现。

（2）典型应用：快速排序、素数测试、近似算法等。

1.3.3 算法分析在竞赛中的应用

算法分析在竞赛中的应用主要体现在时间复杂度优化和空间复杂度控制两方面，通过合理评估算法效率，指导选手选择最优解法。以下是具体应用场景及典型案例：