



数据结构与算法

Data Structures and Algorithms

谢昊

xiehao@cuz.edu.cn

绪论

Introduction

大纲

1. 课程简介

2. 数据结构简介

何为数据结构

为何学数据结构

基本概念

抽象数据类型

3. 算法简介

何为算法

算法特点

算法分析

算法实例

4. 如何学数据结构与算法

推荐的参考资料

5. 小结

课程简介

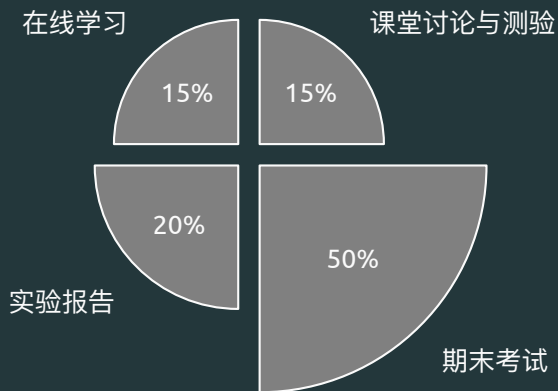


图 1: 总成绩构成

先修课程

- 编程语言：Python/C
- 数学：初等数学

后继课程

- 操作系统、计算机网络、数据库等
- 计算机图形学、图像与视频处理等
- 游戏开发技术、游戏引擎技术、移动应用开发等

数据结构简介

“In solving a problem with or without a computer it is necessary to choose an abstraction of reality, i.e., to define a set of data that is to represent the real situation. ”

—Niklaus E. Wirth
Algorithms and Data Structures, 1985, p11.

程序设计 = 数据结构 + 算法

载体 灵魂

实例：如何在书架上摆放图书

- 寻求解决问题的方法，与数据规模有关
- 大规模数据背景下，更关注解决问题的效率

实例：如何在书架上摆放图书

- 寻求解决问题的方法，与数据规模有关
- 大规模数据背景下，更关注解决问题的效率

需考虑的因素

- 插入：如何插入新书？
- 检索：如何查找旧书？

方法一：随便放

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法二：按书名拼音顺序

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法二：按书名拼音顺序

- 插入：先检索，再插入
- 检索：二分

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法二：按书名拼音顺序

- 插入：先检索，再插入
- 检索：二分

方法三：先分区，每区内按方法二处理

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法二：按书名拼音顺序

- 插入：先检索，再插入
- 检索：二分

方法三：先分区，每区内按方法二处理

- 插入：先检索，再插入
- 检索：先定区，再二分

方法一：随便放

- 插入：哪里有空放哪里
- 检索：崩溃!!

方法二：按书名拼音顺序

- 插入：先检索，再插入
- 检索：二分

方法三：先分区，每区内按方法二处理

- 插入：先检索，再插入
- 检索：先定区，再二分

问题

- 图书如何分类？区域大小如何确定？
- 二分法如何实现？

解决问题方法的效率与数据的组织方式有关

方法一：随便放



何为数据结构

方法一：随便放

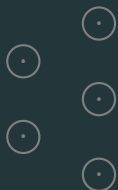


方法二：按序放



何为数据结构

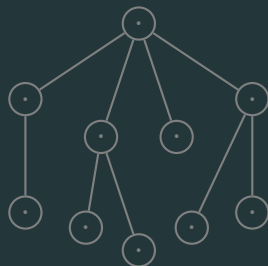
方法一：随便放



方法二：按序放



方法三：分类按序放



何为数据结构

方法一：随便放

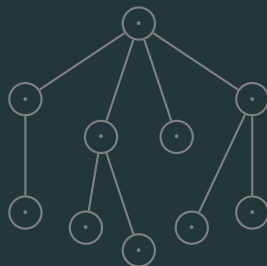


集合结构

方法二：按序放



方法三：分类按序放



何为数据结构

方法一：随便放



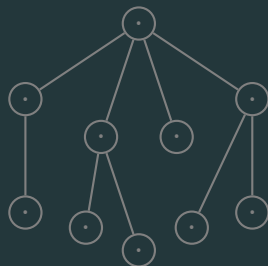
集合结构

方法二：按序放



线性结构

方法三：分类按序放



何为数据结构

方法一：随便放



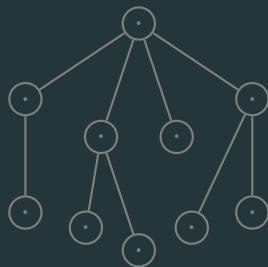
集合结构

方法二：按序放



线性结构

方法三：分类按序放



树形结构

何为数据结构

方法一：随便放



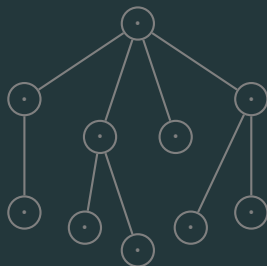
集合结构

方法二：按序放



线性结构

方法三：分类按序放



树形结构

由实际生活问题抽象出计算机可解的数据结构

数据的逻辑结构

- 从具体问题中抽象出的数学模型
- 对现实世界中某个特定领域知识或概念的抽象

何为数据结构

数据的逻辑结构

- 从具体问题中抽象出的数学模型
- 对现实世界中某个特定领域知识或概念的抽象

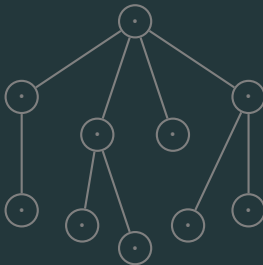
集合结构



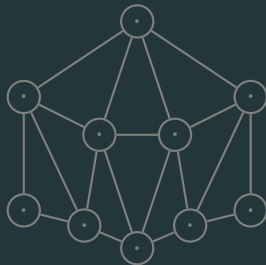
线性结构 1 : 1



树形结构 1 : n



图形结构 $m : n$



数据的物理结构

- 数据的逻辑结构在计算机中的存储形式
- 同一逻辑结构可采用多种不同的物理结构表达

数据的物理结构

- 数据的逻辑结构在计算机中的存储形式
- 同一逻辑结构可采用多种不同的物理结构表达

顺序存储结构



何为数据结构

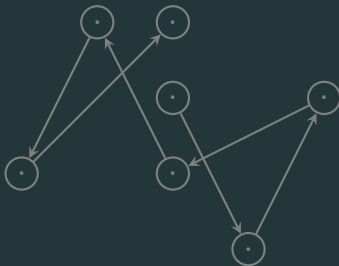
数据的物理结构

- 数据的逻辑结构在计算机中的存储形式
- 同一逻辑结构可采用多种不同的物理结构表达

顺序存储结构



链式存储结构



实例：编程实现按序打印正整数

- 输入：正整数个数 n
- 输出：按顺序打印从 1 至 n 的全部正整数

实例：编程实现按序打印正整数

- 输入：正整数个数 n
- 输出：按顺序打印从 1 至 n 的全部正整数
- 迭代实现

```
1 void print_n(int n) {  
2     for (int i = 1; i <= n; ++i) {  
3         printf("%d\n", i);  
4     }  
5 }
```

实例：编程实现按序打印正整数

- 输入：正整数个数 n
- 输出：按顺序打印从 1 至 n 的全部正整数
- 迭代实现

```
1 void print_n(int n) {  
2     for (int i = 1; i <= n; ++i) {  
3         printf("%d\n", i);  
4     }  
5 }
```

- 递归实现

```
1 void print_n(int n) {  
2     if (n) {  
3         print_n(n - 1);  
4         printf("%d\n", n);  
5     }  
6 }
```

实例：编程实现按序打印正整数（续）

- 测试

```
1  #include <stdio.h>
2
3  void print_n(int);
4
5  int main(void) {
6      int n;
7      scanf("%d", &n);
8      print_n(n);
9      return 0;
10 }
```

实例：编程实现按序打印正整数（续）

- 测试

```
1  #include <stdio.h>
2
3  void print_n(int);
4
5  int main(void) {
6      int n;
7      scanf("%d", &n);
8      print_n(n);
9      return 0;
10 }
```

- 当 $n = 10$ 时，二者均输出

```
1
2
3
4
5
6
7
8
9
10
```

实例：编程实现按序打印正整数（续）

- 当 $n = 1,000,000$ 时，迭代版输出

...

999992

999993

999994

999995

999996

999997

999998

999999

1000000

实例：编程实现按序打印正整数（续）

- 当 $n = 1,000,000$ 时，迭代版输出

```
...  
999992  
999993  
999994  
999995  
999996  
999997  
999998  
999999  
1000000
```

- 递归版输出 (GNU/Linux, GCC)

```
zsh: segmentation fault (core dumped) ...
```

实例：编程实现按序打印正整数（续）

- 当 $n = 1,000,000$ 时，迭代版输出

```
...  
999992  
999993  
999994  
999995  
999996  
999997  
999998  
999999  
1000000
```

- 递归版输出 (GNU/Linux, GCC)

```
zsh: segmentation fault (core dumped) ...
```

- 递归版无输出 (Windows, GCC)

解决问题方法的效率与空间利用率有关

实例：求多项式值

- 输入：多项式阶数 n 、系数序列 $\{a_k\}_{k=0}^n$ 、给定点 v
- 输出：多项式 $f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$ 在点 v 处的值 $f(v)$

实例：求多项式值

- 输入：多项式阶数 n 、系数序列 $\{a_k\}_{k=0}^n$ 、给定点 v
- 输出：多项式 $f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$ 在点 v 处的值 $f(v)$
- 常规实现

```
1 double f(int n, double *a, double x) {  
2     double p = a[0];  
3     for (int i = 1; i <= n; ++i) {  
4         p += a[i] * pow(x, i); // 采用标准库中的幂函数  
5     }  
6     return p;  
7 }
```

实例：求多项式值（续）

- 考虑如下转换：

$$\begin{aligned}f(x) &= a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n \\&= a_0 + x(a_1 + x(\cdots (a_{n-1} + x(a_n)) \cdots))\end{aligned}$$

实例：求多项式值（续）

- 考虑如下转换：

$$\begin{aligned}f(x) &= a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n \\ &= a_0 + x(a_1 + x(\cdots (a_{n-1} + x(a_n)) \cdots))\end{aligned}$$

- 换一种思路实现

```
1 double f(int n, double *a, double x) {  
2     double p = a[n];  
3     for (int i = n - 1; i >= 0; --i) {  
4         p = a[i] + x * p;  
5     }  
6     return p;  
7 }
```

实例：多项式求值（续）

- 测试运行时间¹

```
1 void time_f(int times) {
2     const int N = 9; // 多项式阶数
3     double a[] = {
4         0, 1, 2, 3, 4, 5, 6, 7, 8, 9
5     }; // 多项式系数
6     clock_t start = clock(); // 开始计时
7     for (int i = 0; i < times; ++i) {
8         f(N, a, 1.1); // 重复执行测试函数
9     }
10    clock_t end = clock(); // 结束计时
11    double duration = (double)(end - start)
12        / CLK_TCK / times; // 取平均值
13    printf("%6.2e\n", duration);
14 }
```

¹实现计时功能需包含<time.h>头文件

实例：多项式求值（续）

- 测试运行时间¹

```
1 void time_f(int times) {  
2     const int N = 9; // 多项式阶数  
3     double a[] = {  
4         0, 1, 2, 3, 4, 5, 6, 7, 8, 9  
5     }; // 多项式系数  
6     clock_t start = clock(); // 开始计时  
7     for (int i = 0; i < times; ++i) {  
8         f(N, a, 1.1); // 重复执行测试函数  
9     }  
10    clock_t end = clock(); // 结束计时  
11    double duration = (double)(end - start)  
12        / CLK_TCK / times; // 取平均值  
13    printf("%6.2e\n", duration);  
14 }
```

- 执行 10^7 次取平均，结果分别为

解决方案	运行时间（秒）
常规实现	4.55×10^{-7}
另一种思路	3.20×10^{-8}

¹实现计时功能需包含<time.h>头文件

解决问题方法的效率与**算法巧妙程度**有关

为何学数据结构

使人真正地会写程序

数据结构 (Data Structures)

- 一门研究非数值问题中的操作对象及其间的关系与操作等相关问题的学科

数据结构 (Data Structures)

- 一门研究非数值问题中的操作对象及其间的关系与操作等相关问题的学科

说明

- 数值问题可由数学方程解决

数据结构 (Data Structures)

- 一门研究非数值问题中的操作对象及其间的关系与操作等相关问题的学科

说明

- 数值问题可由数学方程解决
- 操作对象间的关系包括逻辑关系与物理关系

数据 (Data)

- 描述客观事物的符号
- 计算机中可**操作**的对象
- 可被计算机**识别**、并输入给计算机**处理**的符号集合

数据 (Data)

- 描述客观事物的符号
- 计算机中可操作的对象
- 可被计算机识别、并输入给计算机处理的符号集合

例

- 数值类型，如整型、浮点型等
- 非数值类型，如字符型、图形、图像、音频、视频等

数据元素 (Data Element)

- 组成数据的、有一定意义的**基本**单位
- 在计算机中通常作为**整体**处理与考虑
- 又名：元素 (element)、结点 (node)、顶点 (vertex)、记录 (record) 等

数据元素 (Data Element)

- 组成数据的、有一定意义的**基本**单位
- 在计算机中通常作为**整体**处理与考虑
- 又名：元素 (element)、结点 (node)、顶点 (vertex)、记录 (record) 等

例

- 在人类中的**人**
- 在畜类中的**牛、马、羊、猪、狗、鸡、鸭**等
- 在学籍管理系统中一个学生的**基本信息记录**

数据项 (Data Item)

- 一个数据元素可由多个数据项组成
- 数据不可分割的**最小**单位

数据项 (Data Item)

- 一个数据元素可由多个数据项组成
- 数据不可分割的**最小**单位

例

- 人可有**眼、耳、鼻、舌、身**等数据项
- 人亦可有**姓名、性别、年龄、电话**等数据项
- 学生的基本信息记录中可有**学号、姓名、性别**等数据项

数据对象 (Data Object)

- 性质相同²的数据元素的集合
- 数据的一类子集

²指所有数据元素具有相同数量与类型的数据项

数据对象 (Data Object)

- 性质相同²的数据元素的集合
- 数据的一类子集

例

- 包含若干名学生基本信息的**基本信息表**

²指所有数据元素具有相同数量与类型的数据项

逻辑层与物理层

- 逻辑层：只需停留在使用层面，了解接口 (interface)即可
- 物理层：尚需深入至更深层次，需掌握其实现 (implementation)

逻辑层与物理层

- 逻辑层：只需停留在使用层面，了解接口 (interface)即可
- 物理层：尚需深入至更深层次，需掌握其实现 (implementation)

表 1: 逻辑层与物理层的实例

实例	角色	行为	层次
汽车	司机	上下车、油门、刹车、方向盘等	逻辑层
	工程师	发动机、传动轴、刹车片等原理	物理层
电脑	文员	编辑文档、收发邮件、影音娱乐等	逻辑层
	科学家	数据结构、操作系统、体系结构、计算机组成等原理	物理层

数据类型 (Data Type)

- 一组性质相同的值的集合、及在该集合上定义的一些操作的总称

数据类型 (Data Type)

- 一组性质相同的值的集合、及在该集合上定义的一些操作的总称

抽象 (Abstract)

- 抽取出事物具有的普遍性的本质

抽象数据类型

数据类型 (Data Type)

- 一组性质相同的值的集合、及在该集合上定义的一些操作的总称

抽象 (Abstract)

- 抽取出事物具有的普遍性的本质

抽象数据类型 (Abstract Data Type, ADT)

- 对已有数据类型的抽象
- 只描述数据类型是什么，而非如何实现
- 仅取决于逻辑特性，与其在计算机内部的表示与实现³无关

³包括：数据存储的硬件环境、物理结构，实现操作的算法、编程语言等

抽象数据类型实现了数据的逻辑层与物理层的分离

抽象数据类型实现了数据的逻辑层与物理层的分离

表 2: 抽象数据类型作用举例

	电动车	燃油车
能源与动力（物理层）	电能驱动	汽油燃烧供能驱动
驾驶方式（逻辑层）	基本类似	

抽象数据类型

描述 ADT 的一般形式

抽象数据类型名称 {

数据:

数据对象及其间逻辑关系的定义

操作:

操作 1 名称

初始条件描述

操作结果描述

...

操作 2 名称

...

...

操作 n 名称

...

}

⁴与具体实现无关，可以数组、链表或其他任何合理结构实现

抽象数据类型

描述 ADT 的一般形式

抽象数据类型名称 {

数据:

数据对象及其间逻辑关系的定义

操作:

操作 1 名称

初始条件描述

操作结果描述

...

操作 2 名称

...

...

操作 n 名称

...

}

例: 矩阵 ADT

Matrix {

数据:

三元组{a,i,j}, $A = (a_{i,j})_{i \in [1,m], j \in [1,n]}$ ⁴

操作:

Matrix create(int m, int n):

根据输入创建并返回 $m \times n$ 阶空矩阵

int rows(Matrix A):

获取并返回矩阵A的行数

int columns(Matrix A):

获取并返回矩阵A的列数

ElementType get_entry(int i, int j):

根据输入获取并返回矩阵元素 $a_{i,j}$

...

}

⁴与具体实现无关，可以数组、链表或其他任何合理结构实现

算法简介

算法 (Algorithm)

- 解决特定问题求解步骤的描述，表现为指令的有限序列每条指令表示一个或多个操作

算法的伪码描述

- 与硬件环境、物理结构、编程语言等无关的一种指令集描述

何为算法

算法的伪码描述

- 与硬件环境、物理结构、编程语言等无关的一种指令集描述

例：Euclid 辗转相除法求最大公约数

算法 2 Euclid 算法

1: **procedure** Euclid(a, b)

▷ a 与 b 的最大公约数

2: $r \leftarrow a \bmod b$

3: **while** $r \neq 0$ **do**

▷ 若 $r = 0$ 则可跳出循环返回答案

4: $a \leftarrow b$

5: $b \leftarrow r$

6: $r \leftarrow a \bmod b$

7: **end while**

8: **return** b

▷ 最大公约数为 b

9: **end procedure**

算法五大特性

- 输入 (Input): 有 n 个输入, 其中 $n \in \mathbb{Z}^+ \cup \{0\}$
- 输出 (Output): 有 m 个输出, 其中 $m \in \mathbb{Z}^+$
- 有穷性 (Finiteness): 在执行**有限步**后、且在可接受时间内结束, 而非陷入无限循环
- 确定性 (Definiteness): 每个步骤均有**明确含义**, 避免出现歧义
- 可行性 (Feasibility): 可被现有计算机**实现**, 而非仅存于理论研究中

算法设计的要求

- 正确性 (Correctness): 无歧义, 能正确反映问题要求, 并得到正确答案
- 可读性 (Readability): 便于人类理解与交流
- 健壮性 (Robustness): 能正确处理不合法输入, 而非产生异常或莫名其妙结果
- 高效性 (Efficiency): 占用尽可能少的时间与空间

衡量算法效率的指标

- 时间复杂度 $T(n)$: 当问题规模为 n 时, 算法中核心语句的总执行次数 $T(n)$
- 空间复杂度 $S(n)$: 当问题规模为 n 时, 执行算法核心步骤所占用的存储空间 $S(n)$

衡量算法效率的指标

- 时间复杂度 $T(n)$ ：当问题规模为 n 时，算法中核心语句的总执行次数 $T(n)$
- 空间复杂度 $S(n)$ ：当问题规模为 n 时，执行算法核心步骤所占用的存储空间 $S(n)$

说明

- 二者均只与问题规模 n 有关，与硬件等其他条件无关
- 二者均越小越好，但往往不可得兼

算法的渐进分析 (Asymptotic Analysis)

- 小规模问题往往无法反映算法的真实情况
- 着眼长远、更注重时间复杂度总体变化趋势与增长速度的策略与方法

渐进上界：大 O 记号

- 若存在常数 $c > 0$ 、正整数 N 与函数 $f(n)$ ，使得对任意满足 $n > N$ 的 n 均有

$$T(n) \leq cf(n),$$

则称 $f(n)$ 给出 $T(n)$ 增长速度的一个渐进上界，记作 $T(n) = O(f(n))$

渐进上界的性质

- 对任意常数 $c > 0$, 有

$$O(f(n)) = O(cf(n))$$

- 对任意常数 $a > b > 0$, 有

$$O(n^a + n^b) = O(n^a)$$

- 若 $T_f(n) = O(f(n))$ 且 $T_g(n) = O(g(n))$, 则有

$$T_f(n) + T_g(n) = \max\{O(f(n)), O(g(n))\}$$

$$T_f(n) \cdot T_g(n) = O(f(n) \cdot g(n))$$

表 3: 常见渐进上界

渐进上界	输入规模					
	1	2	4	8	16	32
$O(1)$	1	1	1	1	1	1
$O(\log_2 n)$	0	1	2	3	4	5
$O(n)$	1	2	4	8	16	32
$O(n \log_2 n)$	0	2	8	24	64	160
$O(n^2)$	1	4	16	64	256	1,024
$O(n^3)$	1	8	64	512	4,096	32,768
$O(2^n)$	2	4	16	256	65,536	4,294,967,296
$O(n!)$	1	2	24	40,326	2,092,278,988,000	2.6313×10^{37}

实例：求最大子序列和

- 给定整数序列 $\{a_k\}_{k=0}^{n-1}$ ，求其中连续子序列和

$$s(i, j) = \sum_{k=i}^j a_k$$

的最大值 $s^* = \max_{0 \leq i \leq j < n} s(i, j)$

实例：求最大子序列和

- 给定整数序列 $\{a_k\}_{k=0}^{n-1}$ ，求其中连续子序列和

$$s(i, j) = \sum_{k=i}^j a_k$$

的最大值 $s^* = \max_{0 \leq i \leq j < n} s(i, j)$

示范样例

- 输入：4, -3, 5, -2, -1, 2, 6, -2
- 输出⁵：11

⁵注意到 $4 - 3 + 5 - 2 - 1 + 2 + 6 = 11$

解决方案甲：直接解法

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      for (int i = 0; i < n; ++i) { // 子列左端  
4          for (int j = i; j < n; ++j) { // 子列右端  
5              int this_sum = 0;  
6              for (int k = i; k <= j; ++k) { // 遍历子列  
7                  this_sum += a[k];  
8              }  
9              if (this_sum > max_sum) {  
10                 max_sum = this_sum; // 更新最值  
11             }  
12         }  
13     }  
14     return max_sum;  
15 }
```

解决方案甲：直接解法

```
1 int max_subsequence_sum(int *a, int n) {  
2     int max_sum = 0;  
3     for (int i = 0; i < n; ++i) { // 子列左端  
4         for (int j = i; j < n; ++j) { // 子列右端  
5             int this_sum = 0;  
6             for (int k = i; k <= j; ++k) { // 遍历子列  
7                 this_sum += a[k];  
8             }  
9             if (this_sum > max_sum) {  
10                 max_sum = this_sum; // 更新最值  
11             }  
12         }  
13     }  
14     return max_sum;  
15 }
```

问题与改进

- 三重循环
- $T(n) = O(n^3)$

解决方案甲：直接解法

```
1 int max_subsequence_sum(int *a, int n) {  
2     int max_sum = 0;  
3     for (int i = 0; i < n; ++i) { // 子列左端  
4         for (int j = i; j < n; ++j) { // 子列右端  
5             int this_sum = 0;  
6             for (int k = i; k <= j; ++k) { // 遍历子列  
7                 this_sum += a[k];  
8             }  
9             if (this_sum > max_sum) {  
10                 max_sum = this_sum; // 更新最值  
11             }  
12         }  
13     }  
14     return max_sum;  
15 }
```

问题与改进

- 三重循环
- $T(n) = O(n^3)$
- 内层循环冗余
- 可与中层循环合并

解决方案乙：一种改进

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      for (int i = 0; i < n; ++i) { // 子列左端  
4          int this_sum = 0;  
5          for (int j = i; j < n; ++j) { // 遍历子列  
6              this_sum += a[j];  
7              if (this_sum > max_sum) {  
8                  max_sum = this_sum; // 更新最值  
9              }  
10         }  
11     }  
12     return max_sum;  
13 }
```

解决方案乙：一种改进

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      for (int i = 0; i < n; ++i) { // 子列左端  
4          int this_sum = 0;  
5          for (int j = i; j < n; ++j) { // 遍历子列  
6              this_sum += a[j];  
7              if (this_sum > max_sum) {  
8                  max_sum = this_sum; // 更新最值  
9              }  
10         }  
11     }  
12     return max_sum;  
13 }
```

问题与改进

- 二重循环
- $T(n) = O(n^2)$

解决方案乙：一种改进

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      for (int i = 0; i < n; ++i) { // 子列左端  
4          int this_sum = 0;  
5          for (int j = i; j < n; ++j) { // 遍历子列  
6              this_sum += a[j];  
7              if (this_sum > max_sum) {  
8                  max_sum = this_sum; // 更新最值  
9              }  
10         }  
11     }  
12     return max_sum;  
13 }
```

问题与改进

- 二重循环
- $T(n) = O(n^2)$
- 能否更快

解决方案丙：分而治之

- 将原问题逐步**分解**为规模更小且更易解决的同类型子问题
- 将已解决的若干小问题逐步反向**合并**为更简单的原问题
- 代码详见教材

解决方案丙的时间复杂度推导

- 若问题规模为 n ，且分解方式采用二分，单次合并操作耗时为 c ，则有

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + nc \\&= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}c\right) + nc \\&\dots \\&\leq 2^k T\left(\frac{n}{2^k}\right) + knc,\end{aligned}$$

解决方案丙的时间复杂度推导

- 若问题规模为 n ，且分解方式采用二分，单次合并操作耗时为 c ，则有

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + nc \\&= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}c\right) + nc \\&\dots \\&\leq 2^k T\left(\frac{n}{2^k}\right) + knc,\end{aligned}$$

- 若 $T(1) = O(1)$ ，则当 $k = \log_2 n$ 时，有

$$T(n) = O(n \log_2 n)$$

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :								
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4							
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3						
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5					
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2				
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1			
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2		
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :								
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4							
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5					
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5			+2		
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5			+2	+6	
4 :								
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5			+2	+6	
4 :	+6							
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5			+2	+6	
4 :	+6					+8		
8 :								

图 2: 解决方案丙示例演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
1 :	+4	-3	+5	-2	-1	+2	+6	-2
2 :	+4		+5			+2	+6	
4 :	+6					+8		
8 :	+11							

图 2: 解决方案丙示例演示过程

解决方案丁：在线处理

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      int this_sum = 0;  
4      for (int j = 0; j < n; ++j) { // 遍历子列  
5          this_sum += a[j];  
6          if (this_sum > max_sum) {  
7              max_sum = this_sum; // 更新最值  
8          } else if (this_sum < 0) {  
9              this_sum = 0; // 丢弃负子序列和  
10         }  
11     }  
12     return max_sum;  
13 }
```

解决方案丁：在线处理

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      int this_sum = 0;  
4      for (int j = 0; j < n; ++j) { // 遍历子列  
5          this_sum += a[j];  
6          if (this_sum > max_sum) {  
7              max_sum = this_sum; // 更新最值  
8          } else if (this_sum < 0) {  
9              this_sum = 0; // 丢弃负子序列和  
10         }  
11     }  
12     return max_sum;  
13 }
```

问题与改进

- 一重循环
- $T(n) = O(n)$

解决方案丁：在线处理

```
1  int max_subsequence_sum(int *a, int n) {  
2      int max_sum = 0;  
3      int this_sum = 0;  
4      for (int j = 0; j < n; ++j) { // 遍历子列  
5          this_sum += a[j];  
6          if (this_sum > max_sum) {  
7              max_sum = this_sum; // 更新最大值  
8          } else if (this_sum < 0) {  
9              this_sum = 0; // 丢弃负子序列和  
10         }  
11     }  
12     return max_sum;  
13 }
```

问题与改进

- 一重循环
- $T(n) = O(n)$
- 能否更快

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2

this_sum	0
max_sum	0

图 3: 解决方案丁示例 A 演示过程

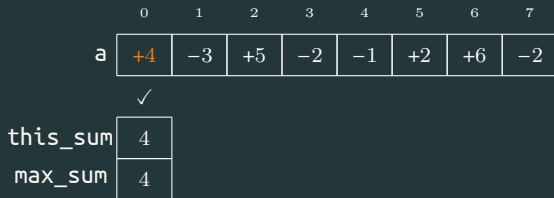


图 3: 解决方案丁示例 A 演示过程

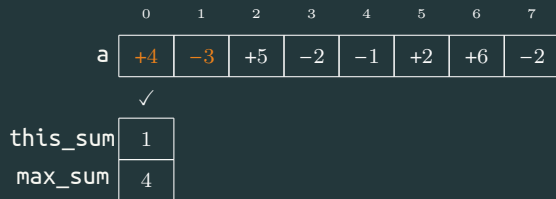


图 3: 解决方案丁示例 A 演示过程

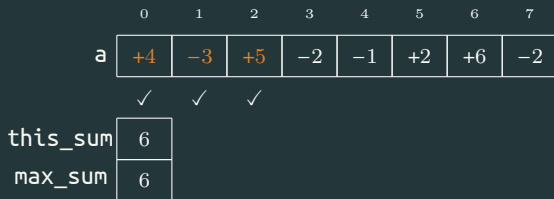


图 3: 解决方案丁示例 A 演示过程

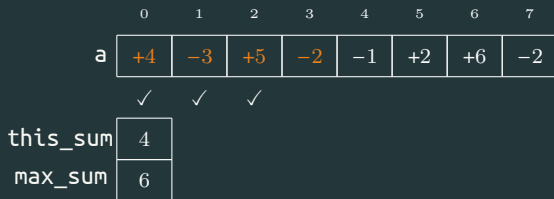


图 3: 解决方案丁示例 A 演示过程

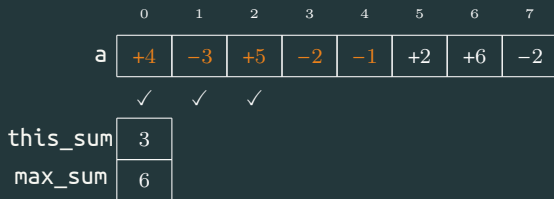


图 3: 解决方案丁示例 A 演示过程

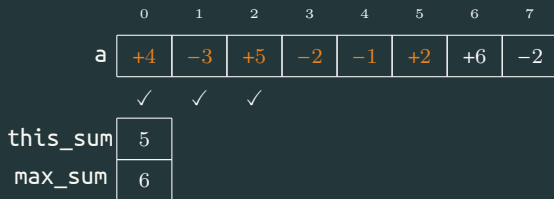


图 3: 解决方案丁示例 A 演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
	✓	✓	✓	✓	✓	✓	✓	
this_sum	11							
max_sum	11							

图 3: 解决方案丁示例 A 演示过程

	0	1	2	3	4	5	6	7
a	+4	-3	+5	-2	-1	+2	+6	-2
	✓	✓	✓	✓	✓	✓	✓	
this_sum	9							
max_sum	11							

图 3: 解决方案丁示例 A 演示过程

	0	1	2	3	4	5	6	7
a	-1	+3	-2	+4	-6	+1	+6	-1

this_sum	0
max_sum	0

图 4: 解决方案丁示例 B 演示过程

	0	1	2	3	4	5	6	7
a	-1	+3	-2	+4	-6	+1	+6	-1

this_sum	0
max_sum	0

图 4: 解决方案丁示例 B 演示过程

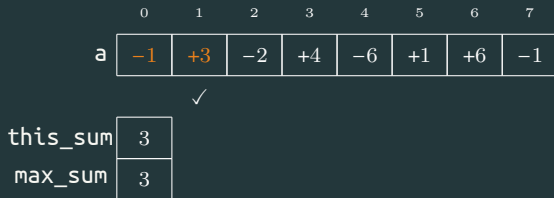


图 4: 解决方案丁示例 B 演示过程

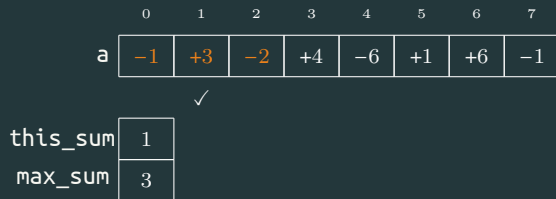


图 4: 解决方案丁示例 B 演示过程

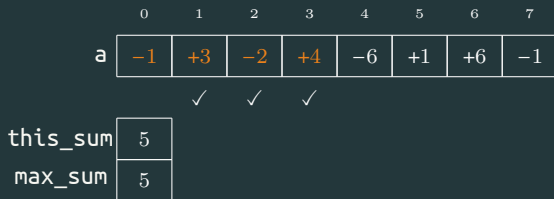


图 4: 解决方案丁示例 B 演示过程

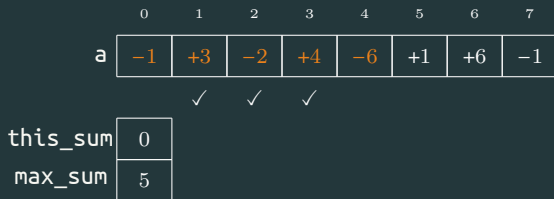


图 4: 解决方案丁示例 B 演示过程

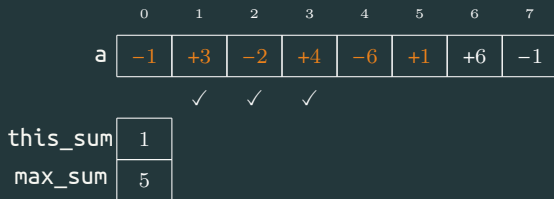


图 4: 解决方案丁示例 B 演示过程

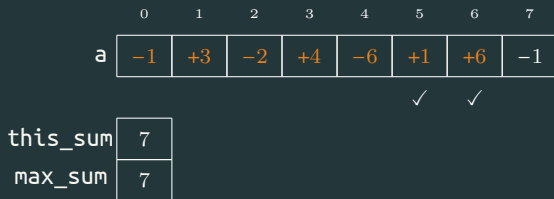


图 4: 解决方案丁示例 B 演示过程

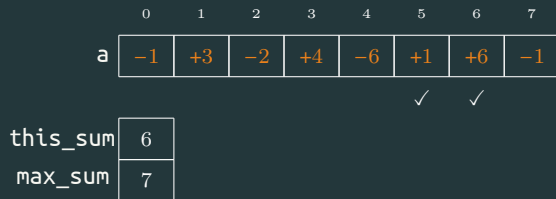


图 4: 解决方案丁示例 B 演示过程

解决方案丁的说明

- 只需遍历一次序列即可求出结果
- 在任意时刻算法均可对截至当前子序列给出正确解

表 4: 各种解决方案运行时间比较 (单位: 秒)

算法	$T(n)$	输入规模 n				
		10	10^2	10^3	10^4	10^5
甲	$O(n^3)$	0.00103	0.47015	448.77	∞	∞
乙	$O(n^2)$	0.00045	0.01112	1.1233	111.13	∞
丙	$O(n \log_2 n)$	0.00066	0.00486	0.05843	0.68631	8.0113
丁	$O(n)$	0.00034	0.00063	0.00333	0.03042	0.29823

如何学数据结构与算法

掌握原理，多加练习



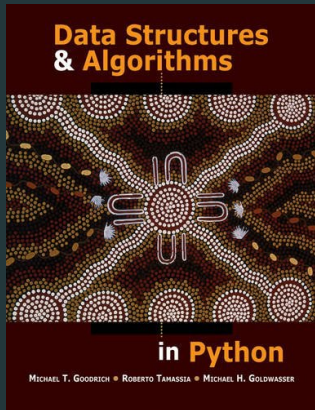
图 5: 大话数据结构 [程 20]

- 通俗易懂
- 适合入门



图 6: 数据结构 (C++ 版) [邓 13]

- 脉络清晰
- 适合入门与进阶
- 可免费获取几乎所有资源



- 概念解释较为透彻
- 适合入门
- 翻译欠佳，建议原版

图 7: 数据结构与算法 (Python 版) [GTG13]

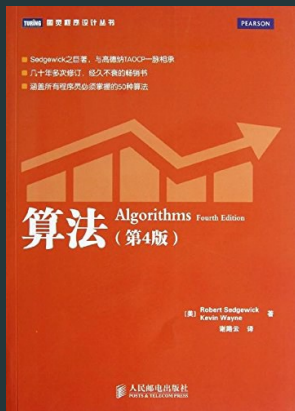


图 8: 算法 [SW11]

- 循序渐进
- 适合入门
- 可配合教学视频

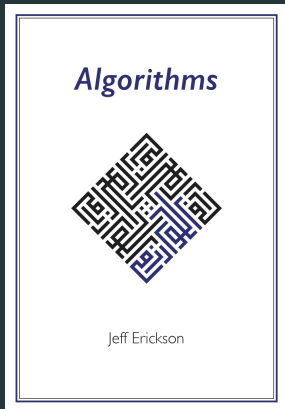


图 9: 算法 [Eri19]

- 可读性极高
- 算法进阶之作
- 开源免费

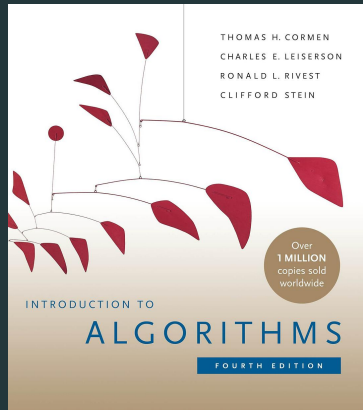


图 10: 算法导论 [CLRS22]

- 算法介绍全面深刻
- 与编程语言无关
- 领域权威

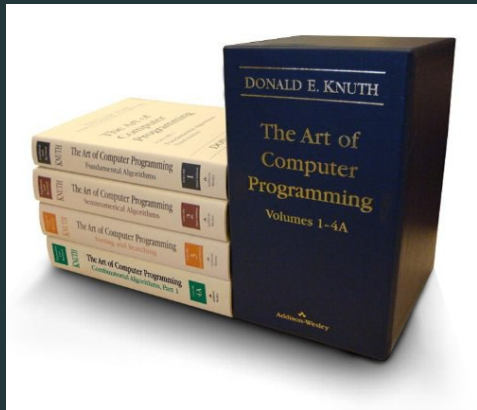


图 11: 计算机程序设计艺术 [Knu11]

- 计算机科学领域圣经级著作
- 高德纳 (Donald E. Knuth) 毕生力作
- 数据结构开山之作


小结

小结

- 数据结构为程序设计的**载体**，算法是程序设计的**灵魂**
- 解决问题方法的效率与**数据的组织方式**、**空间利用率**以及 **算法的巧妙程度**有关
- 数据结构研究**非数值**问题中对象的操作及其间的关系
- 数据结构有**逻辑结构**与**物理结构**两大分类方式
- 抽象数据类型实现了逻辑层与物理层的分离
- 算法描述解决特定问题步骤，表现为指令的有限序列，每条指令表示一个或多个操作
- 算法五大特性：输入、输出、有穷、确定、可行
- 衡量算法效率的指标包括**空间**与**时间**复杂度
- 算法的渐进分析着眼长远、注重时间复杂度的**总体变化趋势**与增长速度的策略与方法

问与答


附录

 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein.
Introduction to algorithms.

MIT press, 4th edition, 2022.

 Jeff Erickson.
Algorithms.

2019.

 Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser.
Data structures and algorithms in Python.

John Wiley & Sons Ltd, 2013.

 Donald E Knuth.
The Art of Computer Programming, Volumes 1-4A.

Addison-Wesley Professional, 1997-2011.



Robert SedgeWick and Kevin Wayne.

Algorithms.

Pearson Education, 4th edition, 2011.



程杰.

大话数据结构（溢彩加强版）.

清华大学出版社, 2020.



邓俊辉.

数据结构: C++ 语言版.

清华大学出版社, 3rd edition, 2013.