第 1 章　概　　论

算法与数据结构是计算机科学与技术、软件开发与应用、信息管理、电子商务、网络安全等专业的一门专业基础课。算法与数据结构课程的内容不仅是程序设计（特别是非数值计算的程序设计）的基础，而且是设计和实现编译程序、操作系统、数据库系统及其他系统程序的重要基础。无论读者从业于计算机行业，还是希望在计算机相关方面继续深造，该课程的学习都是必需的。

算法与数据结构研究解决非数值计算的现实问题中的数据在计算机中如何表示、快速存取和处理的方法。这里所说的数据是广义的概念，它不仅包括整型、实型、逻辑型等基本类型的数据，还包括带有一定结构的各种复杂的数据，如串、记录、向量、矩阵等，也包括各种表格、图形、音频和视频等非数值型的数据。当用计算机存储数据时不仅要存储这些数据的值，还要存储这些数据之间的相互关系。因此存储数据和这些数据之间的关系就出现了各种不同的存储方法。

学习算法与数据结构的目的是编写高质量的程序，主要包括以下三个方面：

① 能够分析研究计算机加工的对象的特性，选择合适的逻辑结构、存储结构并设计相应的算法；

② 提高处理复杂程序设计问题的能力，要求编写的程序结构正确、清晰、易读；

③ 掌握算法的时间复杂度和空间复杂度的分析技术。

本章学习目标：

- 理解数据结构的基本概念和术语；
- 掌握算法分析技术，学会计算算法的时间复杂度和空间复杂度。

1.1　什么是数据结构

美国计算机界最初出现信息结构这一名称是在 20 世纪 60 年代。1968 年，D. E. Knuth 教授开创了数据结构的最初体系，他所著的“*The Art of Computer Programming*”第一卷“*Fundamental Algorithms*”是第一本较系统地阐述数据的逻辑结构和存储结构及其操作的著作。20 世纪 70 年代初，数据结构作为一门独立的课程开始进入大学课堂。

数据结构起源于程序设计，它随着大型程序的出现而出现。随着计算机科学与技术的不断发展，计算机的应用领域已不再局限于科学计算，而更多地应用于控制、管理等非数值处理领域。据统计，当今处理非数值计算类问题占用了 90%以上的计算机时间。与此相应，计算机处理的数据也由纯粹的数值发展到字符、表格、图形、图像、声音等具有一定结构的数据，处理的数据量也越来越大，这就给程序设计带来一个问题：应如何组织待处理的数据，表示数据之间的关系及实现数据运算？

计算机解决一个具体问题时，大致需要经过下列几个步骤：首先要从具体问题中抽象出一个合适的数学模型，然后设计一个解此数学模型的算法，之后编出程序、进行测试/调整，直至得到最终解答。

寻求数学模型的实质是分析问题，从中提取操作的对象，并找出这些操作对象之间的关系，

然后用数学的语言加以描述。当人们用计算机处理“数值计算问题”时（如求解弹道轨迹），所用的数学模型是用数学方程描述的，所涉及的运算对象一般是整型、实型和逻辑型等基本类型的数据，因此程序设计者主要关注程序的设计技巧，而不是数据的存储和组织方式。然而，计算机应用的更多领域是“非数值计算问题”，它们的数学模型无法用数学方程描述，而需要用数据结构描述，解决此类问题的关键是设计出合适的数据结构。描述非数值型问题的数学模型通常是用线性表、树、图等结构来描述的。

【例 1.1】 学生信息管理系统

对学生信息表常用的操作有：查找某个学生的有关情况，统计班级人数，按某个字段排序，增加或删除学生信息（转专业）等，由此可以建立一张按学号顺序排列的学生信息表，如表 1.1 所示。诸如此类的表结构还有图书馆书目管理系统、人事档案管理系统、仓库管理系统等。

表 1.1 学生信息表

学号	姓名	性别	专业	住址
04180101	侯亮平	男	计算机科学与技术	北京
04180102	高小琴	女	计算机科学与技术	深圳
04180103	陆亦可	女	计算机科学与技术	珠海
04180104	陈海	男	计算机科学与技术	上海
04180105	李达康	男	计算机科学与技术	杭州
04180106	高育良	男	计算机科学与技术	南京
04180107	赵东来	男	计算机科学与技术	武汉
04180108	陈岩石	男	计算机科学与技术	重庆
04180109	沙瑞金	男	计算机科学与技术	珠海
04180110	蔡成功	男	计算机科学与技术	广州
……	……	……	……	……

在这类问题中：

① 计算机处理的对象是各种表；

② 元素之间的逻辑关系是线性关系；

③ 施加于对象上的操作有遍历、查找、插入、删除等。

【例 1.2】 人机博弈

计算机之所以能和人博弈，是因为事先已经将对弈的策略和评价规则等输入了计算机中。在人机对弈问题中，计算机操作的对象是对弈过程中可能出现的称为格局的棋盘状态。如图 1.1 所示是井字棋对弈树，包括了多个对弈的格局，格局之间的关系是由比赛规则决定的。通常，这个关系是非线性的，因为从一个棋盘格局可以派生出几个格局，而从每一个新的格局又可派生出多个可能的格局。因此，如果将从对弈开始到结束的过程中所有可能出现的格局都表示出来，就可以得到一棵“树”。其“树根”是对弈开始之前的棋盘格局，而所有的“叶子”就是可能出现的结局，对弈的过程就是从树根沿树枝到每个叶子的过程。诸如此类的树状结构还有家族的族谱、计算机的文件系统、单位的组织结构图等。

在这类问题中：

① 计算机处理的对象是树状结构；

② 元素间的关系是一种一对多的层次关系；

③ 施加于对象上的操作有遍历、查找、插入、删除等。

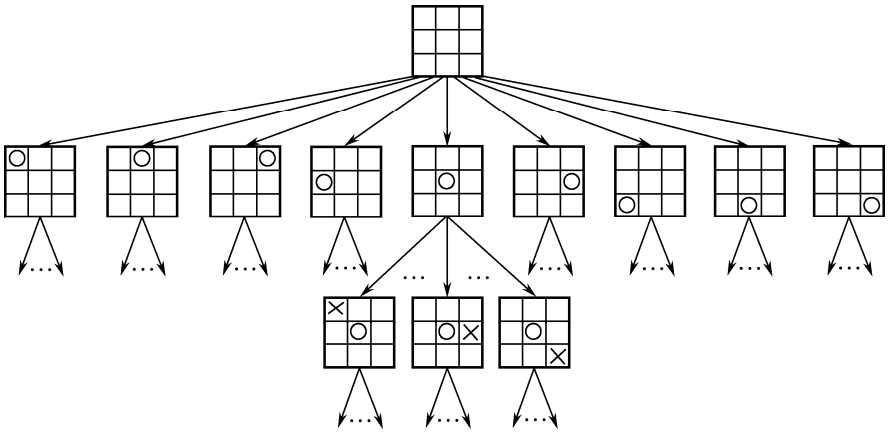


图 1.1　井字棋对弈树

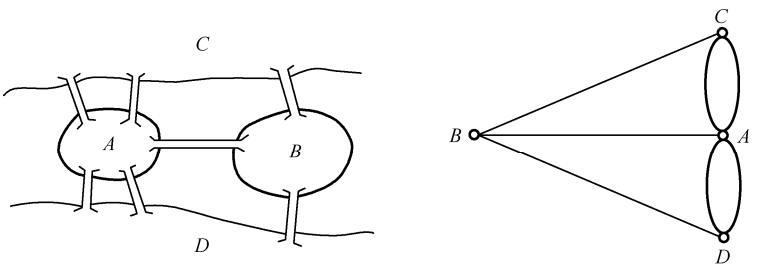
【例 1.3】　哥尼斯堡七桥问题

在哥尼斯堡的一个公园里，有 7 座桥将普雷格尔河中两个岛与两边的河岸相互连接起来，如图 1.2（a）所示。问是否可能从这 4 块陆地中任意一块出发，恰好通过每座桥一次，再回到起点？这一问题很久没有人找到答案。

1736 年，数学家欧拉把这 4 块陆地抽象为 4 个点 *A*、*B*、*C*、*D*，每座桥抽象为连接两个点的一条边，如图 1.2（b）所示。这样哥尼斯堡七桥问题被抽象为：从某一点出发，寻找经过每条边一次且仅一次，最后回到出发点的路径，这也被称为无向图的欧拉回路问题。欧拉回路存在的充分必要条件是：

① 图是连通的；

② 图中与每个顶点相连的边数（即顶点的度）必须是偶数。



（a）哥尼斯堡七桥　　（b）哥尼斯堡七桥的图表示

图 1.2　哥尼斯堡七桥问题

因为哥尼斯堡七桥问题不满足上述条件，所以它是无解的。欧拉由哥尼斯堡七桥问题所引发的对图的研究论文是图论的开篇之作，因此欧拉也被称为图论之父。从哥尼斯堡七桥问题可以看到，要利用图来解决问题，关键的第一步是找到现实问题的实体与图的点和边的对应关系。诸如此类的结构还有城市网络交通图、网络工程图等。

在这类问题中：

① 计算机处理的对象是各种图；

② 元素间的关系是复杂的图或网状关系，是一种多对多的关系；

③ 施加于对象上的操作有遍历、查找、插入、删除等。

上述三个例子表明，描述这类非数值计算问题的数学模型不再是数学方程式，而是诸如表、

树和图之类的数据结构。因此，简单地说，数据结构是一门研究非数值计算的程序设计问题中计算机的操作对象以及它们之间的关系和操作的学科。

在程序的设计中，数据结构的选择是一个基本的设计考虑因素。一些大型系统的构造经验表明，系统实现的困难程度和系统构造的质量都严重依赖于是否选择了最优的数据结构。通常，确定了数据结构后，算法就容易得到了。有些时候，事情也会反过来，我们要根据特定算法来选择数据结构与之适应。不论哪种情况，选择合适的数据结构都是非常重要的。

至今，数据结构课程已经成为计算机程序设计的重要理论基础，是计算机学科中一门综合性的专业基础课,也是非计算机专业的主要选修课程，同时还是一门考研课程，数据结构前承高级语言程序设计和离散数学，后接操作系统、编译原理、数据库原理等专业课程，为研制开发各种系统与应用软件奠定理论和实践基础。该课程学习的效果不仅关系到后续课程的学习，而且直接关系到软件设计水平的提高和专业素质的培养，在计算机学科教育中有非常重要的作用。

1.2 基本概念和术语

1．数据（Data）

数据是信息的载体，是描述客观事物的数字、字符，以及所有能输入计算机中的、被计算机程序识别和处理的符号的集合。

例如，数学计算中所用到的整数和实数、文本编辑所用到的字符串等都是数据。随着计算机软、硬件技术的发展，计算机能够处理的对象范围也在扩大，相应地，数据的含义也被拓宽了。文字、图像、图形、声音、视频等非数值型数据也都是计算机可以处理的数据。

2．数据元素（Data Element）

数据元素是数据中的一个“个体”，是数据的基本单位。在有些情况下，数据元素也称为元素、结点、顶点、记录等。数据元素用于完整地描述一个对象。

例如，一个学生的记录、棋盘中的一个格局、图中的一个顶点等都是一个数据元素。

3．数据项（Data Item）

数据项是组成数据元素的有特定意义的不可分割的最小单位。如构成一个数据元素的字段、域、属性等都可称之为数据项。数据元素是数据项的集合。

例如，学生信息表中的学号、姓名、性别、专业等即为数据项。

4．数据对象（Data Object）

数据对象是具有相同性质的数据元素的集合，是数据的一个子集。

例如，非负整数数据对象是自然数集合 N={0,1,2,…}，英文小写字母数据对象是字符集合 $C=\{'a','b',\cdots,'z'\}$ 等。

5．数据结构（Data Structure）

数据结构通过抽象的方法研究一组有特定关系的数据的存储与处理。数据结构主要研究三个方面的内容，如图 1.3 所示。

① 数据之间的逻辑关系，即数据的逻辑结构；

② 数据及其逻辑关系如何在计算机中存储与实现，即数据的存储结构；

③ 在某种存储模式下，对数据施加的操作是如何实现的，即运算实现。

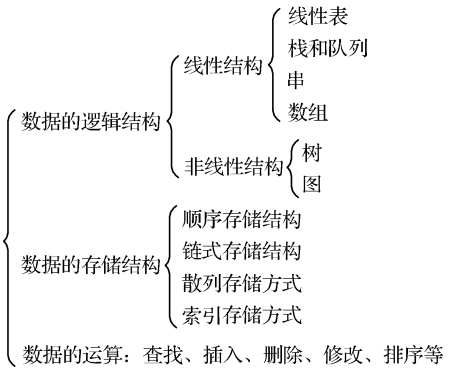


图 1.3　数据结构的三要素

6．数据的逻辑结构

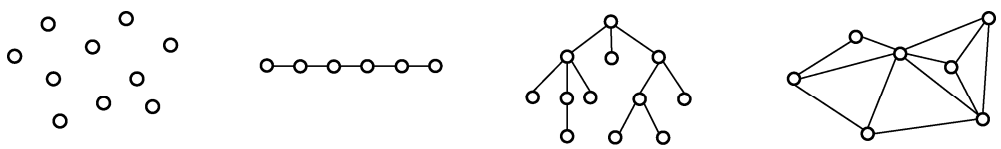
数据的逻辑结构讨论的是数据元素间的逻辑关系，与存储实现无关，是独立于计算机的。常见的逻辑结构如下。

① 集合结构：数据元素间的次序是任意的。数据元素之间除“属于同一集合”的联系外，没有其他的逻辑关系，如图 1.4（a）所示。

② 线性结构：数据元素为有序序列。数据元素之间存在着一种一对一的关系。这种结构的特征是，若结构是非空集，那么，除第一个和最后一个数据元素外，其余数据元素都有且只有一个直接前驱（元素）和一个直接后继（元素），如图 1.4（b）所示。

③ 树结构：数据元素之间存在着一种一对多的关系。在这种结构中，除一个特殊的结点（根结点）外，其他所有结点都有且仅有一个前驱（结点）和零至多个后继（结点），如图 1.4（c）所示。

④ 图结构：数据元素之间存在着一种多对多的关系。在这种结构中，所有结点均可以有多个前驱（结点）和多个后继（结点），如图 1.4（d）所示。图结构也称为网状结构。



（a）集合结构　（b）线性结构　（c）树结构　（d）图结构

图 1.4　数据的逻辑结构示意图

数据（逻辑）结构可形式化定义为： Data _Structure＝(D, R) 。在数据结构的二元组中，D 是数据元素的有限集合，R 是 D 上的关系的有限集合。其中，每个关系都是从 D 到 D 的关系。在表示每个关系时，用尖括号表示有向关系，如<a,b>表示存在结点 a 到结点 b 之间的关系；用圆括号表示无向关系，如(a,b)表示既存在结点 a 到结点 b 之间的关系，又存在结点 b 到结点 a 之间的关系。

7．数据的存储结构

讨论数据结构的目的是在计算机中存储数据并实现对它的操作，因此还需要研究数据及其逻辑关系如何在计算机中存储与实现，即数据的存储结构（也称物理结构）。常用的存储结构说明如下。

① 顺序存储结构：借助数据元素在存储器中的相对位置来表示数据元素之间的关系，通常用数组实现。

② 链式存储结构：借助表示数据元素存储地址的指针显式地指出数据元素之间的逻辑关

系，逻辑上相邻的数据元素其物理位置不要求相邻，通常用指针实现。采用链式存储结构，除存储数据元素本身之外还要存储指示数据元素间关系的指针。

③ 散列（哈希）存储方式：是专用于集合结构的数据存储方式。在散列存储方式中，用一个散列（哈希）函数将数据元素按关键字和一个唯一的存储位置关联起来。方法是，根据设定好的散列函数 f(key)和处理冲突的规则将一组关键字映象到一个有限的连续的地址集（区间）上。

④ 索引存储方式：数据被排成一个序列 $d_1, d_2, \cdots, d_n$，每个结点 d_i 在序列里都有对应的位序 $i(1 \leqslant i \leqslant n)$，位序可以作为结点的索引存储在索引表中。检索时利用结点的顺序号 i 来确定结点的存储地址。

【例 1.4】 有一个数据结构为：$G=(D, R)$，$D=\{d_1, d_2, d_3, d_4, d_5\}$，$R=\{\langle d_1, d_2\rangle, \langle d_2, d_3\rangle, \langle d_3, d_4\rangle, \langle d_4, d_5\rangle\}$，集合 D 中的数据元素具有线性关系，假定每个数据元素占 4 个存储单元，则顺序存储结构如图 1.5 所示（结点 d_1 放在 2000H 号单元中），链式存储结构如图 1.6 所示（头指针 head 指向 2040H 号单元）。

2000H	d_1
2004H	d_2
2008H	d_3
200CH	d_4
2010H	d_5
2014H	
2018H	
201CH	
2020H	

图 1.5 顺序存储结构

头指针

head 2040

	Data	Next
2000H	d_5	NULL
2008H		
2010H	d_3	2018
2018H	d_4	2000
2020H		
2028H		
2030H	d_2	2010
2038H		
2040H	d_1	2030

图 1.6 链式存储结构

从图 1.5 可以看出，在顺序存储结构中，数据元素占用连续的存储空间，它是一种紧凑结构。而在链式存储结构中，一部分存储空间中存放的是表示数据关系的附加信息（即指针），因此这是一种非紧凑结构。

存储密度：数据本身所占的存储量和整个结构所占的存储量之比，即：

$$d=\frac{\text{数据本身所占的存储量}}{\text{整个结构所占的存储量}}$$

由此可见，紧凑结构的存储密度可以达到 1，非紧凑结构的存储密度小于 1。存储密度越大，存储空间的利用率越高。但是，非紧凑结构中存储的附加信息会给某些运算带来极大的方便，例如，在进行插入、删除等运算时，链式存储结构比顺序存储结构就方便得多。非紧凑结构牺牲了存储空间，换取了时间。

同一个逻辑结构可以采用不同的存储结构实现，如何选择合适的存储结构要根据实际的需求和具体应用而确定。

8．数据的运算

① 创建：创建某种数据结构。

② 清除：删除数据结构。

③ 插入：在数据结构指定的位置插入一个新数据元素。

④ 删除：将数据结构中的某个数据元素删去。

⑤ 搜索：在数据结构中搜索满足特定条件的数据元素。

⑥ 更新：修改数据结构中某个数据元素的值。

⑦ 访问：访问数据结构中的某个数据元素。

⑧ 遍历：按照某种顺序访问数据结构中的每个数据元素，使每个数据元素恰好被访问一次。

自测题 1．在数据结构中，从逻辑上可以将之分为（　　）。

A．动态结构和静态结构　　B．紧凑结构和非紧凑结构

C．内部结构和外部结构　　D．线性结构和非线性结构

【2005 年中南大学】

【参考答案】D

自测题 2．已知表头元素为 c 的单链表在内存中的存储状态如下表所示。

地址	元素	链接地址
1000H	a	1010H
1004H	b	100CH
1008H	c	1000H
100CH	d	NULL
1010H	e	1004H
1014H		

现将 f 放于 1014H 处并插入单链表中，若 f 在逻辑上位于 a 和 e 之间，则 a，e，f 的链接地址依次是（　　）。

A．1010H, 1014H, 1004H　　B．1010H, 1004H, 1014H

C．1014H, 1010H, 1004H　　D．1014H, 1004H, 1010H

【2012 年全国统一考试】

【参考答案】D

1.3　算法和算法分析

Pascal 之父、结构化程序设计的先驱 Niklaus Wirth 教授于 1984 年凭借一本专著获得了图灵奖，这本专著的名字是“*Algorithms + Data Structures = Programs*”（算法+数据结构=程序），这里的数据结构指的是数据的逻辑结构和存储结构，而算法则是对数据运算的描述。由此可见，算法与数据结构关系紧密，选择的数据结构是否恰当将直接影响算法的效率，而数据结构的优劣由算法的执行来体现。程序设计的实质是，对要处理的实际问题选择一种合适的数据结构，再设计一个好的算法。

1.3.1　算法的定义及特性

算法（Algorithm）是对特定问题求解步骤的一种描述，是指令的有限序列。其中每条指令表示一个或多个操作。简单地说，算法就是解决特定问题的方法。描述一个算法可以采用文字叙述，也可以采用传统流程图、N-S 图或 PAD 图等。本书采用 C++语言描述。

算法与数据结构的关系紧密，在进行算法设计时首先要确定相应的数据结构。如果在 100 个杂乱无章的数中查找一个给定的数，则只能用顺序查找的方法，效率较低。但是，如果这 100 个数已经按照从小到大的顺序排列好，则可以采用折半查找的方法，这显然比顺序查找的效率要高得多。

一个算法具有下列重要特性。

（1）有穷性：算法只执行有限步，并且每步应该在有限的时间内完成。这里“有限”的概念不是纯数学的，而是指在实际应用中合理的和可接受的。例如，一个简单的算法程序不应该在一个月的计算时间内还不能完成。

以下算法不符合有穷性。

[代码 1.1]

```
void loopforever {
    while(1)
        cout<<"do nothing";
}
```

（2）确定性：算法中的每条指令必须有确切的含义，无二义性。在任何条件下，算法只有唯一的一条执行路径，即对于相同的输入只能得出相同的输出。

（3）可行性：算法中描述的操作都必须足够基本，也就是说，可以通过已经实现的基本运算执行有限次来实现。

例如，“把两个变量交换”和“将变量 *a* 的值增 1”的算法，就是足够基本的，是可行的；而“把两个变量 *a* 和 *b* 的最大公因子 *s* 送给变量 *c*”的算法就不是足够基本的，是不可行的。

（4）输入：算法具有零个或多个输入，也就是说，算法必须有加工的对象。输入取自特定的数据对象的集合。输入的形式可以是显式的，也可以是隐式的。有时，输入可能被嵌入在算法中。

（5）输出：算法具有一个或多个输出。这些输出与输入之间有某种确定的关系。这种确定关系就是算法的功能。举例如下。

[代码 1.2]

```
int getsum(int num) {
    int  sum = 0;
    for (i = 1;i <= num; i++)
        sum += i;
    return sum;
}
```

无输出的算法没有任何意义。

算法和程序十分类似，但是也有区别：

① 在执行时间上，算法所描述的步骤一定是有限的，而程序可以无限地执行下去。因此程序并不需要满足上述的第一个条件（有穷性）。例如，操作系统程序是一个在无限循环中执行的程序，因而不是一个算法。

② 在语言描述上，程序必须采用规定的程序设计语言来书写，而算法没有这种限制。

1.3.2 算法的设计要求

要设计一个好的算法通常要考虑达到以下目标。

（1）正确性（Correctness）。算法的执行结果应当满足预先规定的功能和性能要求。正确性是设计和评价一个算法的首要条件。如果一个算法不正确，则不能完成所要求的任务，其他方面也就无从谈起。一个正确的算法，是指在合理的数据输入下，能够在有限的运行时间内得出正确的结果。采用各种典型的输入数据，上机反复调试算法，使得算法中的每段代码都被测试过，若发现错误则及时纠正，最终可以验证出算法的正确性。当然，要从理论上验证一个算法的正确性，并不是一件容易的事，也不属于本课程所研究的范围，故不进行讨论。

（2）可读性（Readability）。是指算法描述的可读性。算法描述主要是为了方便人的阅读与交流，其次才是让计算机执行。因此算法描述应该思路清晰、层次分明、简捷明了、易读易懂，必要的地方加以注释。此外，晦涩难懂的算法描述易于隐藏较多的错误而难以调试和修改。可读性还要求对算法描述中出现的各种自定义变量和类型做到“见名知义”，即读者一看到每个变量（或类型名）就知道其功用。总之，算法描述的可读性不仅能让读者理解算法的设计思想，同时也可以方便算法的维护。

（3）健壮性（Robustness）。这是指一个算法对不合理（又称不正确、非法、错误等）的数据输入的反应和处理能力。一个好的算法应该能够识别错误数据并进行相应的处理。对错误数据的处理一般包括打印错误信息、调用错误处理程序、返回标识错误的特定信息、终止程序运行等。对错误输入数据进行适当处理，使得不至于引起严重后果。例如，数组都是有上下界的，当访问的下标越过该界限时，系统应该给出保护性错误提示，如直接返回一个错误指示，以避免“数组越界的错误”。

（4）高效性（Efficiency）。算法应有效地使用存储空间并且有较高的时间效率。两者都与问题的规模有关。

1.3.3 算法效率的衡量方法

解决同一个问题，可能存在多个算法，而最重要的计算资源是时间和空间，因此，评价一个算法优劣的重要依据是：程序所用算法运行时花费的时间代价和程序中使用的数据结构占有的空间代价。算法设计者需要在两者之间折中。进行算法性能分析的目的是寻找高效的算法来解决问题，提高工作效率。那么，如何评估各算法的好坏？或者据此设计出更好的算法呢？

衡量算法效率的方法主要有两大类：算法的事后统计方法（后期测试）和算法的事前分析估算方法。

（1）事后统计方法：利用计算机的时钟进行算法执行时间的统计。在算法中的某些部位插入时间函数 time ( )来测定算法完成某一功能所花费的时间。这种方法有非常明显的缺陷：首先，必须把算法转变为程序来执行；其次，时间统计依赖于硬件和软件环境，这容易掩盖算法本身的优劣。

（2）事前分析估算方法：用高级语言编写的程序，其运行时间主要取决于以下因素。

① 算法选用的策略。

② 问题的规模。随着问题涉及的数据量增大，处理起来会越来越困难、复杂。我们用问题规模来描述数据量增大程度。规模越大，消耗时间越多。例如，求 100 以内的质数和求 10000 以内的质数，其执行时间显然是不同的。

③ 编写程序的语言。对于同一个算法，实现语言的级别越高，其执行效率就越低。

④ 编译程序所产生的目标代码的质量。对于代码优化较好的编译程序，其所生成的程序质量较高。

⑤ 机器执行指令的速度。

显然，上述因素中后面三个与算法设计是无关的。也就是说，同一个算法用不同的语言实现，或者用不同的编译程序进行编译，又或者在不同的计算机上运行，其效率均不相同。这表明，使用绝对的时间单位来衡量算法的效率是不合适的。去掉这些与计算机硬件、软件有关的因素，可以认为，一个特定算法的“运行工作量”的大小，只依赖于问题的规模（通常用整数 n 来表示），或者说：它是问题规模的函数。

1.3.4 算法的时间复杂度

算法的时间复杂度（Time Complexity）：$T(n)$是该算法的时间耗费，是其所求解问题规模 n 的函数。当问题规模 n 趋向无穷大时，不考虑具体的运行时间函数，只考虑运行时间函数的数量级（阶），这称为算法的渐进时间复杂度（Asymptotic Time Complexity）。

渐进表示法的常用记法如下。

① 大 O 表示法 $T(n)=\mathrm{O}(f(n))$

说明：存在常量 $c>0$ 和正整数 $N_0 \geqslant 1$，当 $n \geqslant n_0$ 时有 $T(n) \leqslant cf(n)$，即给出了时间复杂度的上界，不可能比 $cf(n)$ 更大。

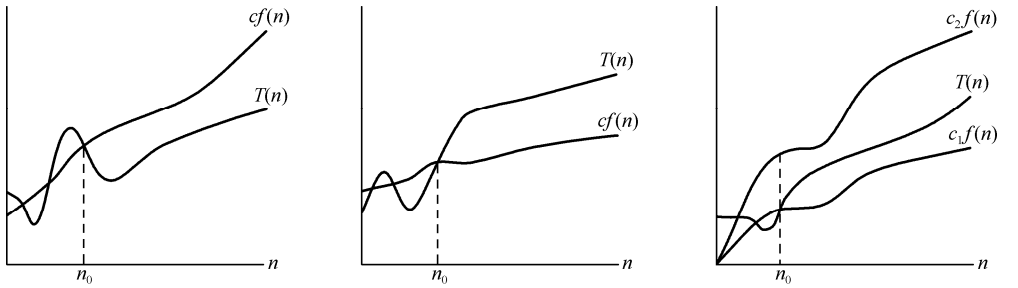
② 大 Ω 表示法 $T(n)=\Omega(f(n))$

说明：存在常量 $c>0$ 和正整数 $n_0 \geqslant 1$，当 $n \geqslant n_0$ 时有 $T(n) \geqslant cf(n)$，即给出了时间复杂度的下界，不可能比 $cf(n)$ 更小。

③ 大 Θ 表示法 $T(n)=\Theta(f(n))$

说明：存在常量 $c>0$ 和正整数 $n_0 \geqslant 1$，当 $n \geqslant n_0$ 时有 $T(n)=cf(n)$，即 $T(n)=\mathrm{O}(f(n))$ 与 $T(n)=\Omega(f(n))$ 都成立，给出了时间复杂度的上界和下界。

上述渐进表示法表示随问题规模 n 的增大，算法执行时间的增长率和数量级函数 $f(n)$ 的增长率是相同的。在进行具体算法分析时，往往对算法的时间复杂度和渐进时间复杂度不予区分，而将渐进时间复杂度大 O 表示法 $T(n)=\mathrm{O}(f(n))$ 简称为时间复杂度。图 1.7 给出了大 O 表示法、大 Ω 表示法及大 Θ 表示法的图示。



（a）$T(n)$=O($f(n)$)　（b）$T(n)$=Ω($f(n)$)　（c）$T(n)$=Θ($f(N)$)

图 1.7　渐进表示法的图示

计算时间复杂度的基本原则如下。

① 在程序中找出最复杂、运行时间最长的程序段，计算它的时间复杂度，也就是整个程序的时间复杂度。

② 数量级函数 $f(n)$的选择：通常选择比较简单的函数形式，并忽略低次项和系数。

常见的时间复杂度及其关系如下：

$$\mathrm{O}(1)<\mathrm{O}(\log n)^{①}<\mathrm{O}(n)<\mathrm{O}(n\log n)<\mathrm{O}(n^2)<\mathrm{O}(n^3)\cdots<\mathrm{O}(n^k)<\mathrm{O}(2^n)<\mathrm{O}(3^n)<\mathrm{O}(n!)\cdots$$

【例 1.5】　时间复杂度的计算。假设某算法的时间函数为 $T(n)=(n+1)^2=n^2+2n+1$，求该算法的时间复杂度。

因为当 $N_0=1$ 及 $c=4$ 时，$T(n) \leqslant cn^2$ 成立，所以 $T(n)=\mathrm{O}(n^2)$。

① 为简单表示，本书中用 log n 表示以 2 为底的对数 $\log_2 n$，以下不再说明。

可见，大 O 表示法取运行时间函数的主项，并忽略低次项和系数。

【例 1.6】 假定有两个算法 A 和 B 求解同一个问题，它们的时间函数分别是 $TA(n)=100n^2$，$TB(n)=5n^3$，它们的时间复杂度分别记为 $O(n^2)$ 和 $O(n^3)$，那么算法 A 一定比算法 B 慢吗？

分析：

① 当输入量较小（如 $n<20$）时，有 $TA(n)>TB(n)$，算法 B 比算法 A 花费的时间少。

② 随着问题规模 n 的增大，两个算法的时间开销之比 $5n^3/(100n^2)=n/20$ 也随着增大。也就是说，当问题规模较大时，算法 A 比算法 B 要有效得多。

算法主要由程序的控制结构（顺序，分支，循环）和原操作构成，算法的运行时间主要取决于两者。具体而言：从算法中选取一种对于所研究的问题来说是基本操作的原操作，然后以该基本操作在算法中重复执行的次数作为算法运行时间的衡量准则。在多数情况下，基本操作就是最深层循环内的原操作，它的执行次数和包含它的语句的频度相同。语句的频度（Frequency Count）指的是该语句重复执行的次数。

【例 1.7】 分析下述程序段的时间复杂度。

[代码 1.3]

```
{ ++x; s=0; }
```

本例选取“++x;”为基本操作，语句频度为 1，则时间复杂度为 $O(1)$，即常量阶。

【例 1.8】 分析下述程序段的时间复杂度。

[代码 1.4]

```
for(j=1; j<=10000; ++j){
    ++x;  s+=x;
}
```

本例选取“++x;”为基本操作，语句频度为 10000，则时间复杂度为 $O(1)$，即常量阶。

【例 1.9】 分析下述程序段的时间复杂度。

[代码 1.5]

```
s=0;
for(j=1; j<=n; j*=2)
    ++x;
```

本例选取“++x;”为基本操作，语句频度为 $\log n$，则时间复杂度为 $O(\log n)$，即对数阶。

【例 1.10】 分析下述程序段的时间复杂度。

[代码 1.6]

```
for(i=1; i<=2*n; ++i) {
    ++x; s+=x;
}
```

本例选取“++x;”为基本操作，语句频度为 $2\times n$，则时间复杂度为 $O(n)$，即线性阶。

【例 1.11】 分析下述程序段的时间复杂度。

[代码 1.7]

```
for(j=1; j<=n; ++j){
    for(k=1; k<=n/4; ++k) {
        ++x; s+=x;
    }
}
```

本例选取“++x;”为基本操作，语句频度为 $n\times n/4$，则时间复杂度为 $O(n^2)$，即平方阶。

【例 1.12】　分析下述程序段的时间复杂度。

[代码 1.8]

```
s=0;
for(j=1; j<=n; j++)
    for(k=1; k<=j; ++k)
        ++x;
```

本例选取“++x;”为基本操作，语句频度为 $(1+n)\times n/2$，则时间复杂度为 $O(n^2)$，即平方阶。

但是，并非所有的双重循环的时间复杂度都是 $O(n^2)$ 的，下面举例说明。

【例 1.13】　分析下述程序段的时间复杂度。

[代码 1.9]

```
for(j=1; j<n; j*=2){
    for(k=1; k<=n; ++k) {
        ++x; s+=x;
    }
}
```

本例选取“++x;”为基本操作，外层循环每循环一次 j 就乘以 2，直至 $j<n$ 条件不满足为止，所以外层循环的循环体共执行 $\log n$ 次，而内层循环的循环体的执行次数总是 n 次。因此“++x;”的语句频度为：$\sum_{j=1}^{\log n} n = n\log n$，则时间复杂度为 $O(n\log n)$，即线性对数阶。

【例 1.14】　矩阵相乘。

[代码 1.10]

```
for(i = 1; i <= n; i++){
    for(j = 1; j <= n; j++){
        c[i][j] = 0;
        for(k = 1; k <= n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

选取第 3 层循环中的循环体“c[i][j] = c[i][j] + a[i][k] * b[k][j];”为基本操作，语句频度为 n^3，所以该算法的时间复杂度是 $O(n^3)$，即立方阶。

最坏情况下的时间复杂度称为最坏时间复杂度。在不做特别说明时，我们讨论的时间复杂度是最坏情况下的时间复杂度。

【例 1.15】　冒泡排序。

[代码 1.11]

```
template <class RecType>
void BubbleSort (RecType a[ ],int size){
    RecType temp;
    int i,pass,flag=1;
    for(pass=1; pass<size && flag; pass++){       // 控制比较的趟数
        flag=0;                                   // 假设本趟没有交换发生
        for(i=0; i<size-pass; i++)                // 一趟中两两比较的次数
            if(a[i]>a[i+1]){                      // 若为逆序，则交换
                temp=a[i];
                a[i]=a[i+1];
```

```
                a[i+1]=temp;
                flag=1;                                    // 若有交换，则 flag 置为 1
            }
        }
    }
```

最好情况：如果待排序序列为正序，则只需要进行一趟冒泡，比较次数为 $n-1$ 次，交换次数为 0 次，最好情况下的时间复杂度为 $O(n)$。

最坏情况：如果待排序序列为逆序，则需要进行 $n-1$ 趟冒泡，比较次数为 $\sum_{i=n}^{2}(i-1)=n(n-1)/2$；交换次数为 $n(n-1)/2$，即进行 $3n(n-1)/2$ 次赋值。因此，冒泡排序算法的时间复杂度取最坏情况下的时间复杂度为 $O(n^2)$。

【例 1.16】 求数组中最大连续子序列之和，如果所有子序列之和都是负数，则返回 0。

数组 A={-2,11,-4,13,-5,-2}，其最大连续子序列之和为 20，即 11+(-4+13)=20。

数组 B={1,3,-2,4,-5}，其最大连续子序列之和为 6，即 1+3+(-2)+4=6。

数组 C={-1,-2,-3,-5,-4}，其最大连续子序列之和为 0。

方法 1：求出所有连续子序列之和，然后从中选取最大值。因为最大连续子序列之和只可能从数组下标 0～length-1 中某个位置开始，所以我们按照如图 1.8 所示的下标范围分别求各子序列之和。

[0, 0], [0, 1], …, [0, length]

[1, 1], [1, 2], …, [1, length]

…

[length-1, length-1]

图 1.8　各子序列的下标范围

[代码 1.12]

```
int maxSubSum1(int array[] ,int length) {
    int i,maxSum = 0,start,end;
    // start 为待求和的子序列的起始下标，end 为结束下标
    for(start = 0; start < length; start++) {
        for(end = start; end < length; end++) {
            int thisSum = 0;
            // thisSum 统计当前子序列 array[start]～array[end]之和
            for(i = start; i <= end; i++) {
                thisSum += array[i];
            }
            // 若 thisSum 大于之前得到的最大子序列之和，则更新 maxSum
            if(thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

方法 1 的时间复杂度为 $O(n^3)$。

方法 2：算法设计的一个重要原则就是“不要重复做事”。在方法 1 中，对 array[start]～array[end]子序列求和，可以由上一次求和 array[start]～array[end-1]的结果加上 array[end]得到，从而不用

从头开始计算。

[代码 1.13]

```
int maxSubSum2(int array[],int length) {
    int maxSum = 0,start,end;
    for(start = 0; start < length; start++) {
        int thisSum = 0;
        for(end = start; end < length; end++) {
            // 已求得的 array[start]～array[end-1]子序列之和加上 array[end]
            // 得到 array[start]～array[end]子序列之和
            thisSum += array[end];
            if(thisSum > maxSum)
                maxSum = thisSum;
        }
    }
    return maxSum;
}
```

方法 2 的时间复杂度是 $O(n^2)$。

方法 3：因为最大连续子序列之和只可能以 0～length-1 中某个下标结尾，所以当遍历到下标是 i 的数据元素时，若 array[i]和前面的连续子序列之和 thisSum 相加的结果小于 0，则 thisSum 不满足题目要求，应舍弃，将 thisSum 重置为 0。若 array[i]和前面的连续子序列之和 thisSum 相加的结果大于等于 0，则保留，并用 thisSum 和 maxSum 进行比较。

[代码 1.14]

```
int maxSubSum3(int array[],int length){
    int maxSum = 0, thisSum = 0 , i;
    for(i = 0; i < length; i++) {
        thisSum += array[i];
        if (thisSum < 0)        // 如果 thisSum<0，则表明前几个连续数之和是小于 0 的
            thisSum = 0;        // 需要计算新的 thisSum，因此 thisSum = 0
        else if(thisSum > maxSum)
            maxSum = thisSum;
    }
    return maxSum;
}
```

方法 3 的时间复杂度是 $O(n)$。

在子序列求和问题的三个解法中，方法 1 的时间复杂度是立方阶，方法 2 的时间复杂度是平方阶，方法 3 的时间复杂度是线性阶，由此可见，算法的设计非常重要。

自测题 3．设 n 是描述问题规模的非负整数，下面程序片段的时间复杂度是（　　）。

```
x=2;
while(x<n/2)
    x=2*x;
```

A．$O(\log_2 n)$[①]　　B．$O(n)$

C．$O(n\log_2 n)$　　D．$O(n^2)$

【2011 年全国统一考试】

① 为与真题一致，这里保留 $\log_2 n$ 的形式，以下不再说明。

【参考答案】A

自测题 4．求整数 $n(n\geq 0)$的阶乘的算法如下，其时间复杂度是（　　）。

```
int fact(int n){
    if (n<=1)    return 1;
    return     n*fact(n-1);
}
```

A．$O(\log_2 n)$　　B．$O(n)$

C．$O(n\log_2 n)$　　D．$O(n^2)$

【2012 年全国统一考试】

【参考答案】B

自测题 5．下列程序段的时间复杂度是（　　）。

```
count=0;
for(k=1; k<=n; k*=2)
    for(j=1; j<=n; j+=1)
        count++;
```

A．$O(\log_2 n)$　　B．$O(n)$

C．$O(n\log_2 n)$　　D．$O(n^2)$

【2014 年全国统一考试】

【参考答案】C

自测题 6．下列函数的时间复杂度是（　　）。

```
int func(int n)
{
    int i=0,sum=0;
    while( sum<n ) sum += ++i;
    return i;
}
```

A．$O(\log_2 n)$　　B．$O(n^{1/2})$

C．$O(n)$　　D．$O(n\log_2 n)$

【2017 年全国统一考试】

【参考答案】B

1.3.5　算法的空间复杂度

空间复杂度（Space Complexity），或称为空间复杂性，是指解决问题的算法在运行时所占用的存储空间。这也是衡量算法有效性的一个指标，记为：

$$S(n)=O(g(n))$$

式中，n 为问题的规模（或大小），表示随着问题规模 n 的增大，算法运行所需的存储空间的增长率与函数 $g(n)$ 的增长率相同。

算法的存储空间包括三个部分：程序本身所占的存储空间、输入数据所占的空间以及辅助变量所占的存储空间。再具体一些，也就是进行程序设计时，程序的存储空间、变量占用空间、系统堆栈的使用空间等。也正由此，空间复杂度的度量分为两个部分：固定部分和可变部分。存储空间的固定部分包括程序指令代码所占空间，常数、简单变量、定长成分（如数组元素、结构成分、对象的数据成员等）变量所占空间等。可变部分包括与实例特性有关的成分变量所占空间、引用变量所占空间、递归栈所占空间、通过 new 运算动态使用的空间等。

如果输入数据所占空间只取决于问题本身，和算法无关，则在讨论算法的空间复杂度时只需分析除输入和程序之外的辅助变量所占的额外空间即可。如果所需额外空间相对于输入数据量来说只是一个常数，则称此算法为“原地工作”，此时的空间复杂度为O(1)。如果算法所需的存储空间与特定的输入有关，那么同时间复杂度一样，也要按照最坏的情况进行考虑。

【例 1.17】 斐波那契数列（Fibonacci Sequence）又称黄金分割数列，指的是这样一种数列：1,1,2,3,5,8,13,21,34,…，这个数列从第 3 项开始，每项都等于前两项之和，编写算法输出斐波那契数列的前 n 项。

说明：使用基本整型 int（4B）表示斐波那契数列时，由于取值范围限制，因此 n 的值不应超过 47。

[代码 1.15]

```
int  i, n, pre, next ;
cin >> n ;
pre = next = 1 ;
cout << pre << '\t' << next << '\t';
for ( i = 3; i < n ; i +=2 )
{    pre= pre + next ;
     next = next + pre ;
     cout << pre << '\t' << next << '\t' ;
}
if ( n%2 == 1 )
     cout << pre+next << endl ;
```

[代码 1.16]

```
int  i, n,* fib;
cin >> n;
fib = new int[n];
fib[0] = fib[1] = 1;
cout << fib[0] << '\t' << fib[1] << '\t';
for( i=2 ; i<n ; i++ ) {
    fib[i] = fib[i-2] + fib[i-1] ;
    cout<< fib[i] << '\t' ;
}
delete []fib;
```

上述两个算法的时间复杂度均为O(n)，而空间复杂度不同：代码 1.15 的空间复杂度为O(1)；代码 1.16 由于申请了一个大小为 n 的动态数组，因此其空间复杂度为O(n)。

对于一个算法，其时间复杂度和空间复杂度往往是相互影响的，当追求一个较好的时间复杂度时，可能会使空间性能变差，即可能导致占用较多的存储空间；反之，当追求一个较好的空间复杂度时，可能会使时间性能变差，即可能导致占用较长的运行时间。另外，算法的所有性能之间或多或少都存在着相互影响。因此，当设计一个算法（特别是大型算法）时，需要综合考虑算法的各项性能、算法的使用频率、算法处理的数据量大小、算法描述语言的特性、算法运行的机器系统环境等诸多因素，通过权衡利弊才能够设计出理想的算法。

1.4 抽象数据类型

当我们使用计算机来解决一个具体问题时，一般需要分析、研究计算机加工的对象的特性，

获得其逻辑结构，根据需求选择合适的存储结构，并设计相应的算法。为了形象直观地描述数据结构，这里引入抽象数据类型（Abstract Data Type，ADT）的概念。

抽象数据类型和高级语言中的数据类型实质上是一个概念，是指一个数学模型以及定义在该模型上的一组操作。例如，各个计算机系统都拥有的“整数”类型其实也是一个抽象数据类型，因为尽管它们在不同的处理器中实现的方法可能不同，但由于其定义的数学特性相同，在用户看来都是相同的，因此，“抽象”的意义在于数据类型的数学抽象特性。

抽象数据类型包含一般数据类型的概念，但其含义比一般数据类型更广、更抽象。一般数据类型通常由具体语言系统内部定义，直接提供给用户定义数据并进行相应的运算，因此也称它们为系统预定义数据类型。抽象数据类型通常由用户根据已有数据类型定义，包括定义其所包含的数据（数据结构）和在这些数据上所进行的操作。在定义抽象数据类型时，就是定义其数据的逻辑结构和操作说明，而不必考虑数据的存储结构和操作的具体实现（即具体操作代码），使得抽象数据类型具有很好的通用性和可移植性，便于用任何一种语言，特别是面向对象的语言实现。

使用抽象数据类型可以更容易地描述现实世界。例如，用线性表抽象数据类型描述学生成绩表，用树或图抽象数据类型描述遗传关系以及城市道路交通图等。抽象数据类型的特征是，使用与实现相分离，实行封装和信息隐蔽。就是说，在抽象数据类型设计时，把类型的定义与其实现分离开来。

和数据结构的形式定义相对应，抽象数据类型可用以下三元组表示：

$$(D, R, P)$$

其中，D 是数据对象，即具有相同特性的数据元素（以下简称元素）的集合，R 是 D 上的关系集合，P 是对 D 的基本操作集合。

抽象数据类型的伪代码定义格式如下：

```
ADT 抽象数据类型名 {
    数据对象 D:〈数据对象的定义〉
    数据关系 R:〈数据关系的定义〉
    基本操作 P:〈基本操作的定义〉
}ADT 抽象数据类型名
```

线性表的抽象数据类型用伪代码描述如下：

```
ADT List {
    数据对象:D={ai | ai∈ElemSet, i=1, 2, …, n, n≥0}
    数据关系:R={<ai-1, ai> | ai, ai-1∈D, i=2, …, n}
    基本操作:
    清空线性表:clear();
    判空线性表:empty();
    求线性表的长度:size();
    遍历线性表:traverse();
    逆置线性表:inverse();
    插入:insert(p, value);
    删除:remove(p);
    定位查找:search(value);
    取表元素:visit(i);
}ADT List
```

随着程序设计的发展，数据结构的发展经历了三个阶段：无结构阶段、结构化阶段和面向对象阶段。本书采用 C++语言描述抽象数据类型和算法，C++语言作为一种面向对象的程序设计

语言，是在吸收结构化程序设计语言的优点的基础上发展起来的。面向对象的程序设计方法，其本质是把数据和处理数据的过程抽象成一个具有特定身份和某些属性的自包含实体，即对象。

计算机科学家 N. Wirth 教授曾经提出一个著名的公式“算法+数据结构=程序”，这个公式在软件开发的进程中产生了深远的影响，但是，它并没有强调数据结构与解决问题的算法是一个整体，因此在面向对象程序设计阶段，有人主张将它修改为：

程序=对象+对象+……

对象=数据结构+算法

在后续章节中，将采用 C++语言中的“抽象类”来描述数据结构的抽象数据类型。例如，线性表的抽象数据类型用抽象类描述如下：

```
template<class T>
class List{
public:
    virtual void clear()=0;                    // 清空线性表
    virtual bool empty()const=0;               // 判空，表空返回 true，非空返回 false
    virtual int size()const=0;                 // 求线性表的长度
    virtual void insert(int i,const T &value)=0;// 在位序 i 处插入元素，值为 value
    virtual void remove(int i)=0;              // 在位序 i 处删除元素
    virtual int search(const T&value)const=0; // 查找值为 value 的元素第一次出现的位序
    virtual T visit(int i)const=0;             // 查找位序为 i 的元素并返回其值
    virtual void traverse()const=0;            // 遍历线性表
    virtual void inverse()=0;                  // 逆置线性表
    virtual ~List(){};
};
```

习题

一、选择题

1．数据结构通常是研究数据的（　　）及它们之间的相互联系。

A．存储结构和逻辑结构　　B．存储和抽象

C．联系和抽象　　D．联系与逻辑

2．数据元素之间没有任何逻辑关系的是（　　）。

A．图结构　　B．线性结构

C．树结构　　D．集合

3．非线性结构中的每个结点（　　）。

A．无直接前驱结点

B．无直接后继结点

C．只有一个直接前驱和一个直接后继结点

D．可能有多个直接前驱和多个直接后继结点

4．链式存储结构所占存储空间（　　）。

A．分为两部分，一部分存放结点的值，另一部分存放表示结点间关系的指针

B．只有一部分存放结点的值

C．只有一部分存储表示结点间关系的指针

5．算法的计算量大小称为算法的（　　）。

A．效率　　B．难度

C．时间复杂度　　D．空间复杂度

6．算法分析的目的是（　　）。

A．找出数据结构的合理性　　B．研究算法中的输入和输出的关系

C．分析算法的效率以求改进　　D．分析算法的可读性和文档特点

二、填空题

1．常见的数据结构有集合、线性结构、_____结构、_____结构。

2．算法的 5 个重要特性是有穷性、_____、可行性、输入、_____。

3．评价算法的优劣，主要考虑算法的_____和_____这两方面。

4．线性结构中元素之间存在_____关系，树结构中元素之间存在_____关系，图结构中元素之间存在_____关系。

三、判断题

1．程序和算法没有区别。

2．算法可以没有输出。

3．数据的逻辑结构与数据元素本身的内容和形式无关。

4．抽象数据类型与计算机内部表示和实现无关。

四、应用题

1．数据存储结构包括哪几种类型？数据逻辑结构包括哪几种类型？

2．算法的 5 个重要特征是什么？

3．在编制管理通讯录的程序时，什么样的数据结构合适？为什么？

4．有实现同一功能的两个算法 A1 和 A2，其中 A1 的时间复杂度为 $T1=O(2^n)$，A2 的时间复杂度为 $T2=O(n^2)$，仅就时间复杂度而言，请具体分析这两个算法哪一个更好。

五、算法设计题

1．已知输入 x，y，z 三个不相等的整数，设计一个“高效”算法，使得这三个数按从小到大的顺序输出。“高效”的含义是，元素比较次数、元素移动次数和输出次数最少。

2．在数组 A[n]中查找值为 k 的元素，若找到则输出其位置 i(i>=1 且 i<=n)，否则输出 0 作为标志。设计算法求解此问题，并分析其时间复杂度。

3．公元前 5 世纪，我国古代数学家张丘建在《算经》一书中提出了“百鸡问题”：鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？请设计一个“高效”的算法求解。