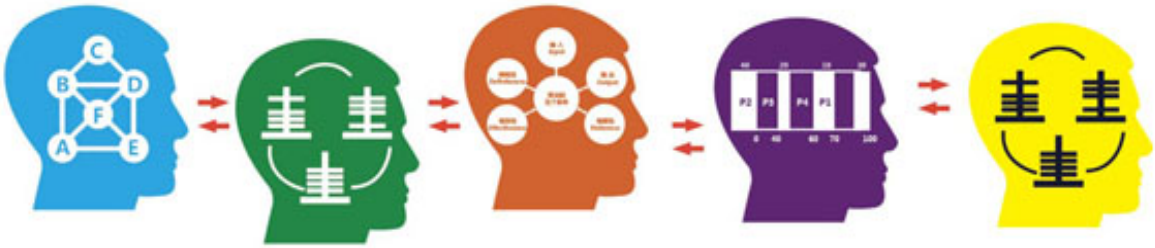
丰富的图解+范例应用
诠释算法的基本概念

计算思维与算法入门

赵 军 ◎等编著

■精选算法入门9堂课 ■计算思维训练
■算法巧思确保高质量编程



机械工业出版社
China Machine Press

计算思维与算法入门

第1章　程序设计与计算思维

计算机堪称20世纪以来人类最伟大的发明之一，对于人类的影响更甚于工业革命所带来的冲击。计算机是一种具备数据处理与计算功能的电子设备。自从人类发明并开始大量应用计算机之后，无论是政府、企业还是个人，对于数据和信息处理的效率都有了极大的提升，如图1-1所示。

图1-1　工厂生产线与大楼自动化安保管理

对于一个有志投身信息技术领域的人员来说，程序设计就是一门和计算机硬件与软件息息相关的学科，是计算机诞生以来一直蓬勃发展的一门新兴科学。之所以说它是新兴科学，是因为计算机的程序设计还在不断地发展和演变，只不过现在的方向是大数据和人工智能等领域。为了发挥计算机强大的运算能力，我们必须掌握程序设计的基本方法和了解它的基本概念。所谓程序，是由符合程序设计语言（Programming Language）语法规则的程序语句、程序代码或程序指令所组成的，而程序设计的目的是通过程序的编写与执行来满足计算机用户的需求。

提示

程序设计语言是一种人类用来和计算机沟通的语言，是由文字与记号所形成的程序语句、代码或指令的集合。程序设计语言主要的功能是将用户的需求使用程序指令表达出来，让计算机按照程序指令替我们完成诸多工作和任务，每种程序设计语言都有各自的文法规则，即语法（syntax），也就是它的使用规则。程序设计语言的语法一直朝着易于使用、易于调试、易于维护以及功能更强的目标持续发展和演变。

对于我们学习程序设计而言，目标无疑就是让我们设计的程序更有效率、可读性更高。我们知道，与计算机作为硬

件工具一样，程序设计语言也只是一种应用工具，因此没有最好的程序设计语言，只有是否适合的程序设计语言，各种程序设计语言都是实现目标的方法。例如，著名的积木式程序设计语言Scratch，其集成开发环境的界面如图1-2所示。

图1-2　积木式程序设计语言是指设计者能以拖曳积木的方式来组合出程序

图1-3　云计算加速了新一代人才必须具备程序设计能力时代的来临

随着信息技术与网络科技的发展，当前进入物联网（Internet of Things，IoT）、大数据、人工智能的云计算（Cloud Computing）时代。一个国家或地区的程序设计能力已经被看成是国力或者地区竞争力的象征。程序设计不再只是信息类学科的专业，而是新一代人才必备的基本能力，各个先进的国家或者地区纷纷将程序设计（或简称编程）列入学生的必修课程，发达地区的城市中小学都开设了编程的信息课程。通过学习程序设计的过程让学生获得解决问题的能力，只有将“创意”通过“设计过程”与计算机相结合，才能顺应这个快速发展和演变的物联网、大数据、人工智能的云计算时代，如图1-3所示。

提示

“云”泛指“网络”，这个名字的源头是工程师通常把网络架构图中不同的网络用“云朵”的形状来表示。云计算就是将网络连接的各种计算设备的运算能力提供出来作为一种服务，只要用户可以通过网络登录远程服务器进行操作，就可以使用这种计算资源。

“物联网”是近年来信息产业界的一个非常热门的议题，它是指将各种具有传感器或感测设备的物品（例如RFID、环境传感器、全球定位系统（GPS）等）与因特网结合起

来，并通过网络技术让各种实体对象自动彼此沟通和交换信息，也就是通过巨大的网络把所有东西都连接在一起。

1.1 认识计算思维

学习程序设计的目标绝对不是要将每个学习者都培养成专业的程序设计人员，而是要帮助每个人建立系统化的逻辑思维模式。以往程序设计的实践目标非常重视“计算”能力，近年来随着因特网的高速发展，计算能力的重要性早已不是唯一的目标，因而程序设计课程的目的特别着重于培养学生的“计算思维”（Computational Thinking，CT，或称为“运算思维”），也就是分析与分解问题的能力。

图1-4　要学好计算思维，通过程序设计来学是最快的途径

编写程序代码不过是程序设计整个过程中的一个阶段而已，在编写程序之前，还有需求分析与系统设计两大阶段。计算思维是用来培养系统化逻辑概念的基础，进而学习在面对问题时具有系统的分析与分解问题的能力，从中探索出可能的解决办法，并找出最有效的算法。我们可以这样说：“学习程序设计不等于学习计算思维，但要学好计算思维，通过程序设计来学绝对是最快的途径”，如图1-4所示。

计算思维是一种使用计算机的逻辑来解决问题的思维，前提是掌握程序设计的基本方法和了解它的基本概念，是一种能够将计算“抽象化”再“具体化”的能力，也是新一代人才都应该具备的素养。计算思维与计算机的应用和发展息息相关，程序设计相关知识和技能的学习与训练过程其实就是一种培养计算思维的过程。当前许多欧美国家从幼儿园开始就培养孩子的计算思维，让孩子从小就养成计算思维的习惯。培养计算思维的习惯可以从日常生活开始，并不限定于任何场所或工具，日常生活中任何牵涉到“解决问题”的议题，都可以应用计算思维来解决，通过边学边体会，逐渐建立起计算思维的逻辑能力。

假如你今天和朋友约在一个没有去过的知名旅游景点碰面，在出门前，你会先上网规划路线，看看哪些路线适合你的行程，以及选乘哪一种交通工具最好，接下来就可以按照计划出发。简单来说，这种计划与考虑过程就是计算思维，按照计划逐步执行就是一种算法（Algorithm），就如同我们把一件看似复杂的事情用容易理解的方式来解决，这样就具备了将问题程序化的能力。图1-5所示的范例是小华早上上学并买早餐的简单计算思维。

图1-5　学生买早餐的过程也是一种计算思维的应用

2006年，美国卡内基·梅隆大学Jeannette M.Wing教授首次提出了“计算思维”的概念，她提出计算思维是现代人的一种基本技能，所有人都应该积极学习。随后谷歌公司为教育者开发了一套计算思维课程，这套课程提到培养计算思维的4部分，分别是分解（Decomposition）、模式识别（Pattern Recognition）、模式概括与抽象（Pattern Generalization and Abstraction）以及算法（Algorithm）。虽然这并不是建立计算思维唯一的方法，不过通过这4部分我们可以更有效地进行思维能力的训练，不断使用计算方法与工具解决问题，进而逐渐养成我们的计算思维习惯。

在训练计算思维的过程中，其实就培养了学习者从不同角度以及现有资源解决问题的能力。正确地运用培养计算思维的这4部分，同时运用现有的知识或工具，找出解决困难问题的方法。学习程序设计就是对这4部分进行系统的学习与组合，并使用计算机来协助解决问题，如图1-6所示。

图1-6　计算思维的4部分示意图

1.1.1 分解

许多人在编写程序或解决问题时，对于问题的分解不知道从何处着手，将问题想得太庞大，如果一个问题不进行有效分解，就会很难处理。将一个复杂的问题分割成许多小问题，把这些小问题各个击破，小问题全部解决之后，原本的大问题也就解决了。

假如我们的一台计算机出现部件故障了，将整台计算机逐步分解成较小的部分，对每个部分内的各个硬件部件进行检查，就容易找出有问题的部件。再假如一位警察在思考如何破案时，也需要将复杂的问题细分成许多小问题，如图1-7所示。

图1-7　将复杂的问题分解为小问题

下面举一个例子来说明。假如我们要分解教小孩刷牙的问题，可以分解与细分成以下情况（见图1-8）：

·用哪种牙刷较好

·要刷多久

·如何刷

·哪种牙膏适合

·准备漱口杯

图1-8　教小孩刷牙的问题

在一些综艺节目中会出现所谓的终极密码游戏，主持人随机从1~100中取出一个彩球（见图1-9），让嘉宾猜彩球的数字，主持人只能针对嘉宾猜的数字回答“高了”或“低了”，这也是一种问题分解的具体应用。想想看，如何才能快速猜到这个数字呢？

图1-9　抽彩球游戏也是一种计算思维的训练

假如取出的彩球数字是“38”，那么我们可以将1~100的数字数列（sequence）先取中间的数字50来比较，38在1~50之间，所以只剩下数列前半段1~50，运用同样的方式取中间的数字再进行比较，数字数列又排除一半，只剩25~50，一直循环这个过程就能找到数字38。这个过程可以参考图1-10。

图1-10　猜测彩球数字的一种二分查找法

事实上，这样的解题分析过程就是训练和培养程序设计的计算思维过程。在上述分析过程中，虽然并没有提到任何艰深的程序设计语言，但是已经带入了程序设计的两个重要概念：“循环（loop）”和“二分查找法（binary search）”。

提示

循环会重复执行一个程序区块中的程序语句，直到满足特定的结束条件为止。例如，想要让计算机算出1+2+3+4+…+100的值，在程序语句中并不需要我们大费周章地从1累加到100，这时只需要使用循环结构就可以轻松实现这种累加。

二分查找法是将数据序列分割成两等份，再用要查找的键值与中间值进行比较，如果键值小于中间值，就可以确定要查找的数据在数据序列的前半段，否则就在后半段。

1.1.2 模式识别

在将一个复杂的问题分解之后，我们常常可以发现小问题中有共同的属性以及相似之处，在计算思维中，这些属性被称为“模式”（Pattern）。模式识别是指在一组数据中找出特征（Feature）或规则（Rule），用于对数据进行识别与分类，以作为决策判断的依据。在解决问题的过程中，找到模式是非常重要的，模式可以让问题的解决更简化。当问题具有相同的特征时，它们能够被更简单地解决，因为存在共同模式时，我们可以用相同的方法解决此类问题。

例如，当前常见的生物识别技术就是利用人体的形态、构造等生理特征（Physiological Characteristics）以及行为特征（Behavior Characteristics）作为依据，通过光学、声学、生物传感等高科技设备的密切结合对个人进行身份识别（Identification或Recognition）与身份验证（Verification）的技术。又例如，指纹识别（Fingerprint Recognition）系统以机器读取指纹样本，将样本存入数据库中，然后用提取的指纹特征与数据库中的指纹样本进行对比与验证（见图1-11），而脸部识别技术则是通过摄像头提取人脸部的特征（包括五官特征），再经过算法确认，就可以从复杂背景中判断出特定人物的脸孔特征。

图1-11　指纹识别系统的应用已经相当普遍

当我们发现越来越多的模式时，解决问题就会变得更加容易和迅速。在知道怎么描述一只狗之后，我们可以按照这种模式轻松地描述其他狗，例如狗都有眼睛、尾巴与4只脚，不一样的地方是每只狗都或多或少地有其独特之处（见图1-12），识别出这种模式之后，便可用这种解决办法来应对不同的问题。

图1-12　狗都有眼睛、尾巴与4只脚

因为我们知道所有的狗都有这类属性，当想要画狗的时候，便可将这些共同的属性加入，这样就可以很快地画出很多只狗。在平时，我们也能进行模式识别的思维训练，可以通过动手画图、识别图形、分辨颜色或对物体分类来进行训练。

1.1.3　模式概括与抽象

模式概括与抽象在于过滤以及忽略掉不必要的特征，让我们可以集中在重要的特征上，这样有助于将问题抽象化。通常这个过程开始会收集许多数据和资料，通过模式概括与抽象把无助于解决问题的特性和模式去掉，留下相关的以及重要的属性，直到我们确定一个通用的问题以及建立解决这个问题的规则。

“抽象”没有固定的模式，它会随着需要或实际情况而有所不同。例如，把一辆汽车抽象化，每个人都有各自的分解方式，像车行的业务员与修车技师对汽车抽象化的结果可能就会有差异，如图1-13所示。

图1-13　车行业务员和修车技师对汽车抽象化的结果会有差异

车行业务员：轮子、引擎、方向盘、刹车、底盘。

修车技师：引擎系统、底盘系统、传动系统、刹车系统、悬吊系统。

如何正确而快速地将现实世界的事物抽象化是一门学问，而计算思维着重于分析、分解与概括（或归纳）的能力，是练习抽象化非常有效的方法。在日常生活中也处处可见抽象化，例如我们将复杂、有地形背景的北京地铁运行图简化为如图1-14所示的纯线路图模式，以简单明了的方式标示出各个不同地铁线路的走向及各个站点。

图1-14　北京地铁线路图

计算思维可视为是运用信息科技有效解决问题的心智历程，通过模式概括与抽象的过程整理出有用的数据、资源以及限制条件，整个思维过程可以使用“思维导图（Mind Map）”来归纳和整理。思维导图是由英国的Tony Buzan于

20世纪70年代提出的一种辅助思考的工具，又称脑力激荡图、思维图，它是一种使用图像来帮助思考与表达思维的工具，可以刺激思维并帮助整合思想与信息。借助这种方式，我们可以更轻松地以图形方式来表达自己的想法。

思维导图的绘制其实非常简单，中心点通常表示一个核心主题，用一张纸把记忆记录下来，再把绘图者的思考、分析、规划、概括和归纳后的信息以笔记方式呈现，用关键词向外扩张延伸出分支来表达内容。通过主题的脑力激荡，把想到的字词、概念全部写下来，使用不同尺寸的文字、线条与图形来区别数据间的从属关系与重要程度。除了材料、内容、色彩等多方面的构思以外，同时也训练绘图者在思考过程的“流畅性”与“变通性”，刚开始的思维可能很发散，我们不用想太多，通通写下来就好了，等收集到一定程度，再慢慢筛选掉不适合的部分。假如我们要规划一个减轻体重的计划，首先必须制定一个核心目标，聚焦在“减重”这个主题上，再按序想出解决问题的方法来归纳整理。如图1-15所示是一个思维导图的范例。

图1-15　思维导图的范例

1.1.4 算法

算法是计算思维4个基石的最后一个，不但是人类使用计算机解决问题的技巧之一，也是程序设计中的精髓。算法常出现在规划和设计程序的第一步，因为算法本身就是一种计划，每一条指令与每一个步骤都是经过规划的，在这个规划中包含解决问题的每一个步骤和每一条指令。

在日常生活中有许多工作可以使用算法来描述，例如员工的工作报告、宠物的饲养过程、厨师准备美食的食谱、学生的课程表等。如今我们几乎每天都要使用的各种搜索引擎都必须借助不断更新的算法来运行，如图1-16所示。

特别是在算法与大数据的结合下，这门学科演化出“千奇百怪”的应用，例如当我们拨打某个银行信用卡客户服务中心的电话时，很可能会先经过后台算法的过滤，帮我们找出一名最“合我们胃口”的客服人员来与我们交谈。在因特网时代，通过大数据分析，网店可以进一步了解产品购买和需求产品的人群是哪类人，甚至一些知名IT企业在面试过程中也会测验候选者对于算法的了解程度，如图1-17所示。

图1-16　搜索引擎的背后是不断优化的搜索算法

图1-17　一些知名IT企业面试时也会测验候选者对算法的了解程度

提示

大数据（Big Data，又称为海量数据）由IBM公司于2010年提出，是指在一定时效（Velocity）内进行大量（Volume）、多样性（Variety）、低价值密度（Value）、真实性（Veracity）数据的获得、分析、处理、保存等操作。数据的来源有非常多的途径，大数据的

格式也越来越复杂，大数据解决了商业智能无法处理的非结构化与半结构化数据。

在韦氏辞典中，算法定义为：“在有限步骤内解决数学问题的程序。”如果运用在计算机领域中，我们也可以把算法定义成：“为了解决某项工作或某个问题，所需要的有限数量的机械性或重复性指令与计算步骤。”

1.2 算法的条件

在计算机中，算法更是不可或缺的一环。在认识了算法的定义之后，我们再来看看算法所必须符合的5个条件（可参考图1-18和表1-1）。

图1-18　算法必须符合的5个条件

表1-1　算法必须符合的5个条件

我们认识了算法的定义与条件后，接着要来思考：用什么方法表达算法最为适当呢？其实算法的主要目的在于让人们了解所执行的工作流程与步骤，只要能清楚地体现算法的5个条件即可。

常用的算法一般可以用中文、英文、数字等文字来描述，也就是使用文字或语言语句来说明算法的具体步骤，有些算法则是使用可读性高的高级程序设计语言（如Python、C、C++、Java等）或者伪语言（Pseudo-Language）来描述或说明的。例如，以下算法就是用Python语言来描述函数Pow（）的执行过程：计算所传入的两个数x、y的x^y值。

```
def Pow(x,y):
    p=1
    for i in range(1,y+1):
        p *=x
    return p

print(Pow(4,3))
```

公约数是指可以整除两个整数的整数，通过辗转相除法可以求两个整数的最大公约数，就是通过算法来求解的。下面我们使用while循环设计一个C语言程序，求所输入的两个整数的最大公约数（g.c.d）。辗转相除法的算法如下：

```
if (Num1 < Num2)
{
    TmpNum=Num1;
    Num1=Num2;
    Num2=TmpNum;          /* 找出两个整数中的较大值 */
}
while (Num2 != 0) /* 辗转相除法，直到除数为0就终止循环 */
{
    TmpNum=Num1 % Num2;    /* 求两个整数的余数 */
    Num1=Num2;
    Num2=TmpNum; /* 将本轮相除求得的余数作为下一轮的除数
*/
}
    printf("最大公约数(g.c.d)=%d\n",Num1);
```

提示

伪语言接近于高级程序设计语言，是一种不能直接放进计算机中执行的语言。一般都需要通过一种特定的预处理器（Preprocessor）或者通过人工编写转换成真正的计算机语言才能够加载到计算机中执行，目前较常使用的伪语言有SPARKS、PASCAL-LIKE等。

流程图（Flow Diagram）是一种相当通用的算法表示法，就是使用某些特定图形符号来表示算法的执行过程。为了让流程图具有更好的可读性和一致性，目前较为通用的是ANSI（美国国家标准协会）制定的统一图形符号。表1-2列出了流程图中一些常见的图形符号并附有简单的说明。

表1-2　流程图中常见的图形符号

假如我们要设计一个程序，让用户输入一个整数，而这个程序可以帮助用户判断输入的整数是奇数还是偶数，那么这个程序的流程图大致如图1-19所示。

为了让他人容易阅读，绘制流程图应注意以下几点：

（1）采用标准通用符号，符号内的文字尽量简明扼要。

（2）绘制方向应自上而下，从左到右。

（3）连接线的箭头方向要清楚，线条避免太长或交叉。

图1-19　用流程图描述算法的例子——判断奇数和偶数

提示

算法和过程是有区别的，过程不一定要满足算法有限性的要求，例如操作系统或计算机上运行的过程，除非宕机，否则永远在等待循环中（waiting loop），这就违反了算法五大条件中的“有限性”。

再来看一个算法的例子，我们知道计算机内部是以二进制数的方式来处理数据和信息的，而人类则是以十进制数的方式来处理日常运算的。在计算机中，有些数据也会采用八进制数或十六进制数来表示。十进制数转换成非十进制数要分为整数部分的转换和小数部分的转换，下面通过范例来说明相关的转换算法。

（1）十进制数转换成二进制数

$(63)_{10}=(111111)_2$（十进制整数转换成二进制整数，参考图1-20）

图1-20　十进制整数转成二进制整数

$(0.625)_{10}=(0.101)_2$（十进制小数转换成二进制小数，参考图1-21）

图1-21　十进制小数转换成二进制小数

$(12.75)_{10}=(12)_{10}+(0.75)_{10}$（十进制数转换成二进制数），参考图1-22。

图1-22　十进制数转换成二进制数

其中，$(12)_{10}=(1100)_2$，$(0.75)_{10}=(0.11)_2$

所以$(12.75)_{10}=(12)_{10}+(0.75)_{10}$

$=(1100)_2+(0.11)_2$

$=(1100.11)_2$

（2）十进制数转换成八进制数

$(63)_{10}=(77)_8$（参考图1-23）

图1-23　十进制整数转换成八进制整数

$(0.75)_{10}=(0.6)_8$（参考图1-24）

图1-24　十进制小数转换成八进制小数

（3）十进制转换成十六进制

$(63)_{10}=(3F)_{16}$（参考图1-25）

图1-25　十进制整数转换成十六进制整数

$(0.62890625)_{10}=(0.A1)_{16}$（参考图1-26）

图1-26　十进制小数转换成十六进制小数

$(120.5)_{10}=(120)_{10}+(0.5)_{10}$

其中，$(120)_{10}=(78)_{16}$，$(0.5)_{10}=(0.8)_{16}$，参考图1-27。

图1-27　十进制数转换成十六进制数

1.3 课后习题

1.下面两组词汇都有共同点，各自有一个不同，请找出不同的词汇，并说明差异之处。

（1）A．蛇

B．玫瑰

C．狗

D．老虎

（2）A．熊

B．兔子

C．老鹰

D．狼

E．狐狸

2.算法必须符合哪5项条件？

3.请找出以下序列的模式，并写出“？”处的值。

（1）151、242、333、424、？

（2）CEG、EHK、JN、？

（3）65536、256、16、？

4.试简述思维导图的由来。

5.把$(2004)_{10}$转换为十六进制数的结果是多少？

6.求二进制数$(11.1)_2$的平方，即$(11.1)_2 \times (11.1)_2$的值。

7.请问算法和过程有什么不同？

8.请简述云计算与物联网。

9.什么是“计算思维”？

10.谷歌公司为教育者开发了一套计算思维课程，这套课程提到培养计算思维有哪4部分？

11.什么是“模式”？什么是“模式识别”？

第2章　常用数据结构与算法

当初人们试图建造计算机的主要原因之一是用来存储和管理一些数字化的信息和数据，这也是最初数据结构概念的来源。当我们使用计算机解决问题时，必须以计算机能够了解的模式来描述问题，而数据结构是数据的表示法，也就是计算机中存储数据的基本结构，编写程序就像盖房子一样，要先规划出房子的结构图，如图2-1所示。

图2-1　编写程序就像盖房子一样，要先规划出房子的结构图

简单来说，数据结构所讲述的是一种辅助程序设计并进行优化的方法论，它不仅讨论数据的存储与处理的方法，同时也考虑数据彼此之间的关系与运算，目的是提高程序的执行效率与减少对内存空间的占用等。图书馆的书籍管理也是一种数据结构的应用，如图2-2所示。

图2-2　图书馆的书籍管理也是一种数据结构的应用

2.1 认识数据结构

在信息技术如此发达的今天，我们每天的生活已经和计算机密切不可分了。计算机与信息是息息相关的，因为计算机具有处理速度快和存储容量大两大特点，所以在数据处理中扮演着举足轻重的角色。所谓数据（Data），指的是一种未经处理的原始文字（Word）、数字（Number）、符号（Symbol）或图形（Graph）等，例如姓名或我们常看到的课表、通讯簿等都可称为一种“数据”。

当数据经过处理（Process），例如以特定的方式系统地进行整理、归纳，甚至进行分析后，就成为“信息”（Information）。这样处理的过程就称为“数据处理”，如图2-3所示。信息是使用大量的数据经过系统地整理、分析、筛选而提炼出来的，它具有参考价值，并可为决策提供所需的文字、数字、符号或图表。

图2-3　数据处理过程的示意图

数据结构加算法是指数据进入计算机内进行处理的一套完整逻辑，选定了数据结构就决定了在计算机中数据的存放顺序和位置。例如，在程序设计中需要存取某块内存的数据时，就可以直接使用变量（variable）名称num1与num2进行存取，如图2-4所示。

图2-4　变量在内存中的存取位置的示意图

使用数据结构，再通过程序设计语言所提供的数据类型、引用方法以及相应的操作就可以实现数据结构对应的算法。我们知道一个程序能否快速而有效地完成预定的任务取决于是否选对了数据结构，而程序是否能清楚而正确地把问题解决则取决于算法。因此，我们可以直接这么认为：“数据结构加上算法等于有效率的可执行程序”，如图2-5所示。

图2-5　数据结构加上算法等于可执行程序

程序设计人员必须选择各种数据结构来进行数据的添加、修改、删除、存储等操作。当数据存储在内存中时，根据数据的使用目的，对数据进行妥善的结构化，就可以提高使用效率，如果在选择数据结构时做了错误的决定，那么程序执行的速度将可能变得非常低，如果选错了数据类型，那么后果更是不堪设想。

以日常生活中的医院为例，医院会将事先设计好的个人病历表准备好，当有新的病人上门时，请他们填写好个人的基本信息，之后管理人员就可以按照某种次序（例如姓氏、年龄或电话号码）将病历表加以分类，然后用文件夹或档案柜加以收藏，如图2-6所示。

图2-6　病历表也是一种数据结构

日后当某位病人回诊时，只要询问病人的姓名或年龄，管理人员就可以快速地从文件夹或档案柜中找出病人的病历表，而这个档案柜中所存放的病历表也是一种数据结构的应用。

计算机化业务的增加带动了数字化数据的大量增长，如图2-7所示。数据结构用于表示数据在计算机内存中所存储的位置和方式，通常可以分为以下三种数据类型。

图2-7　计算机化业务的增加带动了数字化数据的大量增长

·基本数据类型（Primitive Data Type）

基本数据类型是不能以其他类型来定义的数据类型，或称为标量数据类型（Scalar Data Type）。几乎所有的程序设计语言都会为标量数据类型提供一组基本数据类型，例如

Python语言中的基本数据类型包括整数、浮点数、布尔值和字符等。

·结构数据类型（Structured Data Type）

结构数据类型也被称为虚拟数据类型（Virtual Data Type），是一种比基本数据类型更高一级的数据类型，例如字符串（String）、数组（Array）、指针（Pointer）、列表（List）、文件（File）等。

·抽象数据类型（Abstract Data Type，ADT）

我们可以将一种数据类型看成是一种值的集合，以及在这些值上所进行的运算和所代表的属性组成的集合。“抽象数据类型”比结构数据类型更高级，是指一个数学模型以及定义在此数学模型上的一组数学运算或操作。也就是说，抽象数据类型在计算机中体现了一种“信息隐藏”（Information Hiding）的程序设计思想以及表示了信息之间的某种特定的关系模式。例如，堆栈（Stack）就是一种典型的抽象数据类型，具有后进先出（Last In First Out，LIFO）的数据操作方式。

2.2 常见的数据结构

不同种类的数据结构适用于不同种类的程序应用，选择适当的数据结构是让算法发挥最大性能的主要因素，精心选择的数据结构可以给设计的程序带来更高效率的算法。然而，无论是哪种情况，数据结构的选择都是至关重要的。接下来我们将介绍一些常见的数据结构。

数组

“数组”结构其实就是一排紧密相邻的可数内存，并提供一个能够直接访问单个数据内容的计算方法。我们可以想象一下自家的信箱，每个信箱都有住址，其中路名就是名称，而信箱号码就是数组的下标（也称为“索引”），如图2-8所示。

图2-8　数组结构与邮递信箱系统类似

邮递员可以按照信件上的住址把信件直接投递到指定的信箱中，这就好比程序设计语言中数组的名称表示一块紧密相邻的内存的起始位置，而数组的下标（或索引）则用来表示从此内存起始位置开始后的第几个内存区块。

数组类型是一种典型的静态数据结构，它使用连续分配的内存空间（Contiguous Allocation）来存储有序表中的数据。静态数据结构是在编译时就给相关的变量分配好内存空间。在建立静态数据结构的初期，必须事先声明最大可能要占用的固定内存空间，因此容易造成内存的浪费。优点是设计时相当简单，而且读取与修改数组中任意一个元素的时间都是固定的，缺点则是删除或加入数据时，需要移动大量的数据。

数组是一组具有相同名称和数据类型的变量的集合，并且它们在内存中占有一块连续的内存空间。数组可以分为一维数组、二维数组与多维数组等，它们的基本工作原理都相同。如果想要存取数组中的数据，需要配合下标值

（index，或称为索引值）找到数组中指定位置的值。在图2-9中的Array_Name是一维数组，它是拥有5个相同数据类型数值的数组。通过名称Array_Name与下标值即可方便地存取这5个数据。

图2-9　数组中数据存储的示意图

二维数组（Two-Dimension Array）可视为一维数组的扩展，都是用于处理数据类型相同的数据，差别只在于维数的声明。例如，一个含有m×n个元素的二维数组A（1：m，1：n），m代表行数，n代表列数，A[4][4]数组中各个元素在直观平面上的具体排列方式可参考图2-10。

图2-10　4×4的数组在直观平面上的排列方式

三维数组的表示法和二维数组一样，都可视为是一维数组的扩展或延伸，如果数组为三维数组，可以看作一个立方体。将arr[2][3][4]三维数组想象成空间上的立方体，如图2-11所示。

图2-11　三维数组可以看成一个立方体

2.3　矩阵与深度学习

从数学的角度来看，对于m×n矩阵（Matrix）的形式，可以用计算机中的A（m，n）二维数组来描述，因此许多矩阵的相关运算与应用都是使用计算机中的数组结构来解决的。如图2-12所示的矩阵A，我们是否可以立即想到就是一个声明为A（1：3，1：3）的二维数组？

图2-12　用矩阵的方式来描述二维数组

在三维图形学中也经常使用矩阵，因为矩阵可以清楚地表示模型数据的投影、扩大、缩小、平移、偏斜与旋转等运算，如图2-13所示。

图2-13　矩阵平移是物体在三维世界向着某一个向量的方向移动

提示

在三维空间中，向量用（a，b，c）来表示，其中a、b、c分别表示向量在x、y、z轴的分量。在图2-14中的向量A是一个从原点出发指向三维空间中的一个点（a，b，c），也就是说，向量同时包含大小及方向两种特性。所谓单位向量（Unit Vector），指的是向量长度为1的向量。通常在向量计算时，为了降低计算上的复杂度，会以单位向量进行运算，所以使用向量表示法就可以指明某变量的大小与方向。

图2-14　向量A在三维空间中表示的大小与方向

深度学习（Deep Learning，DL）是目前最热门的话题之一，它是人工智能的一个分支，也可以看成是具有层次性的机器学习法（Machine Learning，ML），将人工智能推向类似人类学习模式的更深层次。在深度学习中，线性代

数是一个强大的数学工具，常常需要使用大量矩阵运算来提高深度学习的效率。

提示

机器学习是大数据与人工智能发展相当重要的一环，机器通过算法来分析数据，在大数据中找到规则。机器学习是大数据发展的下一个阶段，可以发掘出多种数据变动因素之间的关联性，充分利用大数据和算法来训练机器，其应用范围相当广泛，从健康监控、自动驾驶、机台自动控制、医疗成像诊断工具、工厂控制系统、检测用机器人到网络营销领域。

拥有超多核心的GPU（Graphics Processing Unit，图形处理器）问世之后，这种含有数千个微型且更高效率的运算单元的GPU就被用于并行计算（Parallel Computing），因而大幅地提高了计算机的运算性能。加上GPU内部本来就是以向量和矩阵运算为基础的，大量的矩阵运算可以分配给为数众多的内核同步进行处理，使得人工智能领域正式进入实用阶段，进而成为未来各个学科不可或缺的技术之一。

提示

人工智能的概念最早是由美国科学家John McCarthy于1955年提出的，目标是使计算机具有类似人类学习解决复杂问题与进行思考等能力，凡是模拟人类的听、说、读、写、看、动作等的计算机技术，都被归类为人工智能的可能范围。简单地说，人工智能就是由计算机仿真或执行的具有类似人类智慧或思考的行为，例如推理、规划、问题解决及学习等能力。

深度学习源自于类神经网络（Artificial Neural Network）模型，并且结合了神经网络架构与大量的运算资源，目的在于让机器建立模拟人脑进行学习的神经网络，以解读大数据中图像、声音和文字等多种数据或信息。最为人津津乐道的深度学习应用当属Google Deepmind开发的人工智能围棋程序AlphaGo，它接连打败欧洲和韩国的围棋棋

王。AlphaGo设计的核心思路是输入大量的棋谱数据，让AlphaGo通过深度学习掌握更抽象的概念来学习下围棋的方法，后来它创下了连胜60局的佳绩，并且还在不断反复跟自己的对弈中持续调整神经网络，即提高自己的棋艺。AlphaGo官方网站的首页如图2-15所示。

图2-15　AlphaGo官方网站的首页

提示

类神经网络是模仿生物神经网络的运行模式，取材于人类大脑的结构，基础研究的方向是：使用大量简单且相连的人工神经元来模拟生物神经细胞受到特定程度的刺激而反应刺激。通过神经网络模型建立出系统模型，便可用于推理、预测、评估、决策、诊断的相关应用。要使得类神经网络能正确地运行，必须通过训练的方式让类神经网络反复学习，经过一段时间学习获得经验值，才能有效学习到初步运行的模式。由于神经网络将权重存储在矩阵中，矩阵多半是多维模式，要考虑各种参数的组合，因此会牵涉到“矩阵”的大量运算。

类神经网络的原理也可以应用到计算机游戏中，如图2-16所示。

图2-16　类神经网络的原理也可以应用到计算机游戏中

2.3.1 稀疏矩阵

对于抽象数据类型而言，我们希望阐述的是在计算机应用中具备某种意义的特殊概念，例如稀疏矩阵（Sparse Matrix）就是一个很好的例子。什么是稀疏矩阵呢？简单地说，如果一个矩阵中的大部分元素为零，就被称为稀疏矩阵。图2-17所示的矩阵就是一种典型的稀疏矩阵。

图2-17　稀疏矩阵

对于稀疏矩阵而言，实际存储的数据项很少，如果在计算机中使用传统的二维数组方式来存储稀疏矩阵，就会十分浪费计算机的内存空间。特别是当矩阵很大时，例如存储一个1000×1000的稀疏矩阵，因为大部分元素都是零，所以这样的空间利用率确实不高。提高内存空间利用率的方法是使用三项式（3-Tuple）的数据结构。

我们把每一个非零项以（i，j，item-value）来表示，就是假如一个稀疏矩阵有n个非零项，那么可以使用一个A（0：n，1：3）的二维数组来存储这些非零项，我们把这样存储的矩阵叫压缩矩阵。

其中，A（0，1）存储这个稀疏矩阵的行数，A（0，2）存储这个稀疏矩阵的列数，A（0，3）存储此稀疏矩阵非零项的总数。另外，每一个非零项以（i，j，item-value）来表示。其中，i为这个矩阵非零项所在的行数，j为这个矩阵非零项所在的列数，item-value为这个矩阵非零的值。以图2-17所示的6×6稀疏矩阵为例，可以用图2-18所示的方式来表示。

图2-18　三项式表示稀疏矩阵的方式

A（0，1）表示此矩阵的行数，A（0，2）表示此矩阵的列数，A（0，3）表示此矩阵非零项的总数。

2.3.2　矩阵相加算法

矩阵的相加运算较为简单，前提是相加的两个矩阵对应的行数与列数都必须相等，而相加后矩阵的行数与列数也是相同的，例如$A_{m\times n}+B_{m\times n}=C_{m\times n}$。下面我们来实际看一个矩阵相加的例子，如图2-19所示。

图2-19　矩阵相加

2.3.3　矩阵相乘算法

两个矩阵A与B相乘受到某些条件的限制。首先，必须符合A为一个m×n的矩阵，B为一个n×p的矩阵，A×B的结果为一个m×p的矩阵C，如图2-20所示。

图2-20　矩阵相乘

$C_{11}=a_{11}\times b_{11}+a_{12}\times b_{21}+\ldots+a_{1n}\times b_{n1}$

…

$C_{1p}=a_{11}\times b_{1p}+a_{12}\times b_{2p}+\ldots+a_{1n}\times b_{np}$

…

$C_{mp}=a_{m1}\times b_{1p}+a_{m2}\times b_{2p}+\ldots+a_{mn}\times b_{np}$

2.3.4　转置矩阵

“转置矩阵”（A^t）就是把原矩阵的行坐标元素与列坐标元素相互调换。假设A^t为A的转置矩阵，则有A^t[j，i]=A[i，j]，如图2-21所示。

图2-21　矩阵的转置

2.4 链表

链表（Linked List）又称为动态数据结构，使用不连续内存空间来存储，是由许多相同数据类型的数据项按特定顺序排列而成的线性表。链表的特性是其各个数据项在计算机内存中的位置是不连续且随机（Random）存放的，其优点是数据的插入或删除都相当方便。

当有新数据加入链表后，就向系统申请一块内存空间，而在数据被删除后，就把这块内存空间还给系统，在链表中添加和删除数据都不需要移动大量的数据。链表的缺点是设计数据结构时较为麻烦，另外在查找数据时，也无法像静态数据（如数组）那样可以随机读取数据，必须按序查找到该数据为止。在日常生活中有许多链表抽象概念的运用，例如可以把链表想象成火车，有多少人就挂多少节车厢，当假日人多、需要较多车厢时就多挂些车厢，平日里人少时就把车厢的数量减少，这种做法非常有弹性，如图2-22所示。

图2-22　链表类似于火车及其挂接的车厢

我们最常使用的是“单向链表”（Single Linked List）。一个单向链表节点基本上由两个元素组成，即数据字段和指针，其中的指针指向链表中下一个节点在内存中的地址，如图2-23所示。

图2-23　单向链表的节点示意图

在单向链表中，第一个节点是“链表头指针”，指向最后一个节点的指针设为NULL，表示它是“链表尾”，不指向任何地方。例如列表A={a，b，c，d，x}，其单向链表的数据结构如图2-24所示。

图2-24　单向链表示意图

由于单向链表中所有节点都知道节点本身的下一个节点在哪里，但是对于前一个节点却没有办法知道，因此在单向链表的各种操作中，“链表头指针”就显得相当重要，只要存在链表头指针，就可以遍历整个链表，进行链表节点的加入和删除等操作。注意，除非必要，否则不要移动链表头指针。

2.4.1　单向链表的串接算法

对于两个或两个以上链表的串接（Concatenation，也称为级联），它的实现方法很容易：只要将链表的首尾相连即可，如图2-25所示。

图2-25　单向链表的链接

2.4.2　单向链表节点的删除算法

在单向链表类型的数据结构中，如果要在链表中删除一个节点，如同从一列火车中移走原有的某节车厢，这时根据所删除节点的位置会有三种不同的情况。

1．删除链表的第一个节点

只要把链表头指针指向第二个节点即可，如图2-26所示。

图2-26　删除链表的第一个节点

2．删除链表的最后一个节点

只要将指向最后一个节点ptr的指针直接指向NULL（空节点）即可，如图2-27所示。

图2-27　删除链表的最后一个节点

3．删除链表的中间节点

只要将删除节点的前一个节点的指针指向要删除节点的下一个节点即可，如图2-28所示。

图2-28　删除链表中间的一个节点

2.4.3　在单向链表中添加新节点

在单向链表中添加新节点如同在一列火车中加入新的车厢，有三种情况：加到第一个节点之前、加到最后一个节点之后以及加到此链表中间任一位置。接下来我们使用图解方式来说明。

1．将新节点加到第一个节点之前，即成为此链表的首节点

只需把新节点的指针指向链表原来的第一个节点，再把链表头指针指向新节点即可，如图2-29所示。

图2-29　将新节点加到第一个节点之前

2．将新节点加到最后一个节点之后

只需把链表的最后一个节点的指针指向新节点，新节点的指针再指向NULL即可，如图2-30所示。

图2-30　将新节点加到最后一个节点之后

3．将新节点加到链表中间的位置

例如要插入的节点在X与Y之间，只要将X节点的指针指向新节点，新节点的指针指向Y节点即可，如图2-31和图2-32所示。

图2-31　新节点的指针指向Y节点

图2-32　X节点的指针指向新节点

2.4.4　单向链表的反转

了解单向链表节点的删除和添加之后，大家会发现在这种具有方向性的链表结构中增删节点是相当容易的一件事。而要从头到尾输出整个单向链表也不难，但若要反转过来输出单向链表，则需要某些技巧。我们知道单向链表中的节点特性是知道下一个节点的位置，可是却无从得知它的上一个节点的位置。如果要将单向链表反转，就必须使用三个指针变量，如图2-33所示。

图2-33　单向链表的反转

在算法invert（X）中，我们使用了p、q、r三个指针变量，它的运算过程如下：

（1）执行while循环前，如图2-34所示。

图2-34　执行while循环前链表和各个指针变量的情况

（2）第一次执行while循环，如图2-35所示。

图2-35　第一次执行while循环后链表和各个指针变量的情况

（3）第二次执行while循环，如图3-36所示。

图2-36　第二次执行while循环后链表和各个指针变量的情况

当执行到p=NULL时，单向链表就整个反转过来了。

2.5　堆栈

堆栈（Stack）是一组相同数据类型的组合，所有的操作均在堆栈顶端进行，具有“后进先出”的特性。所谓后进先出，其实就如同自助餐中餐盘在桌面上一个一个往上叠放，在取用时先拿最上面的餐盘，这是典型的堆栈概念的应用，如图2-37所示。

图2-37　自助餐中餐盘存取就是一种堆栈的应用

堆栈是一种抽象数据结构（Abstract Data Type，ADT），具有下列特性：

（1）只能从堆栈的顶端存取数据。

（2）数据的存取符合“后进先出”的原则。

堆栈压入和弹出的操作过程如图2-38所示。

图2-38　堆栈压入和弹出的操作过程

堆栈的基本运算有表2-1所示的5种。

表2-1　堆栈的基本运算

堆栈压入和弹出操作示意图如图2-39所示。

图2-39　堆栈push（压入）和pop（弹出）操作示意图

堆栈结构在计算机中应用得相当广泛，常用于计算机程序的运行，例如递归调用、子程序的调用。在日常生活中的应用也随处可见，例如大楼的电梯（见图2-40）、货架上的商品等，其原理都类似于堆栈这样的数据结构。

图2-40　电梯搭乘方式就是一种堆栈的应用

2.6 队列

队列（Queue）和堆栈都是有序列表，也属于抽象数据类型，所有加入与删除的操作都发生在队列前后两端，即队首和队尾，并且符合“先进先出”（First In First Out，FIFO）的特性。队列的概念就好比乘火车时买票的队伍，先到的人当然可以优先买票，买完后就从队伍前端离去准备乘火车，而队伍的后端又陆续有新的乘客加入，如图2-41所示。

图2-41　买火车票的队伍就是队列原理的应用

队列在计算机领域应用得也相当广泛，例如计算机的模拟（Simulation）、CPU的作业调度（Job Scheduling）、外围设备联机并发处理系统（Spooling）的应用与图遍历的广度优先搜索法（Breadth-First Search，BFS）。堆栈只需一个top指针指向堆栈顶部，而队列则必须使用front和rear两个指针分别指向队列的前端和末尾，如图2-42所示。

图2-42　队列结构示意图

队列是一种抽象数据结构，有下列特性：

（1）具有先进先出的特性。

（2）拥有加入与删除两种基本操作，而且使用front与rear两个指针分别指向队列的前端与末尾。

队列的基本运算有表2-2所示的5种。

表2-2　队列的基本运算

2.6.1 双向队列

双向队列（Double Ended Queues，DEQue）是一种有序线性表，加入与删除可在队列的任意一端进行，如图2-43所示。

图2-43　双向队列的示意图

具体来说，双向队列就是允许队列两端都具备删除或加入功能，而且无论是左端还是右端的队列，队首与队尾指针都是朝队列中央移动的。通常，双向队列的应用可以分为两种：一种是数据只能从一端加入，但可以从两端取出；另一种是可从两端加入，但从一端取出。

2.6.2　优先队列

优先队列（Priority Queue）是一种不必遵守队列先进先出特性的有序线性表，其中的每一个元素都赋予一个优先级（Priority），加入元素时可任意加入，但有最高优先级者（Highest Priority Out First，HPOF）最先输出。

图2-44　急诊病人的安排就是优先队列的应用

像一般医院中的急诊室，当然以最严重的病患（如得SARS的病人）优先诊治，跟进入医院挂号的顺序无关，如图2-44所示。或者在计算机中CPU的作业调度中，优先级调度（Priority Scheduling，PS）就是一种按照进程优先级“调度算法”（Scheduling Algorithm）进行的调度，这种调度会用到优先队列，优先级高的用户比一般用户拥有较高的优先权利。

假设有4个进程：P1、P2、P3和P4，其在很短的时间内先后到达等待队列，每个进程所需的运行时间如表2-3所示。

表2-3　每个进程所需的运行时间

在此设置进程P1、P2、P3、P4的优先次序值分别为2、8、6、4（此处假设数值越小其优先级越低，数值越大其优先级越高）。按优先级调度进程绘出的甘特图如图2-45所示。

图2-45　优先级调度进程的示意图

在此特别提醒大家，当各个元素以输入的先后次序为优先级时，就是一般的队列，假如以输入先后次序的倒序作为优先级，此优先队列其实就是一个堆栈。

2.7　树结构

树结构是一种日常生活中应用相当广泛的非线性结构，包括企业内的组织结构、家族的族谱、篮球赛程以及计算机领域中的操作系统与数据库管理系统都是树结构的衍生应用。如图2-46所示的是Windows的文件资源管理器的示意图，它就是以树结构来组织和存储各种文件的。

图2-46　Windows的文件资源管理器就是以树结构来组织和存储各种文件的

例如，在年轻人喜爱的大型网络游戏中，需要获取某些物体所在的地形信息，如果程序依次从构成地形的模型三角面寻找，往往会耗费许多运行时间，非常低效。因此，程序员一般会使用树形结构中的二叉空间分割树（Binary Space Partitioning Tree，BSP Tree）、四叉树（Quadtree）、八叉树（Octree）等来代表分割场景的数据。四叉树示意图如图2-47所示。地形与四叉树的对应关系如图2-48所示。

图2-47　四叉树示意图

图2-48　地形与四叉树的对应关系

2.7.1 树的基本概念

“树”是由一个或一个以上的节点组成的，存在一个特殊的节点，称为树根（Root）。每个节点都是由一些数据和指针组合而成的记录。除了树根外，其余节点可分为n≥0个互斥的集合，即T_1，T_2，T_3，…，T_n，其中每一个子集合本身也是一个树结构，即此根节点的子树，如图2-49所示。

图2-49　树结构的示意图

一棵合法的树，节点间可以互相连接，但不能形成无出口的回路，例如图2-50所示就是一棵不合法的树。

图2-50　不合法的树（因为节点间形成了无出口的回路）

在树结构中，有许多常用的专有名词，这里将以图2-51中这棵合法的树来为大家详细介绍。

图2-51　一棵合法的树

·度数（Degree）：每个节点所有子树的个数。例如图2-51中，节点B的度数为2，D的度数为3，F、K、I、J等的度数为0。

·层数（Level）：树的层数，假设树根A为第一层，B、C、D节点的层数为2，E、F、G、H、I、J的层数为3。

·高度（Height）：树的最大层数。图2-51所示的树的高度为4。

·树叶或称终端节点（Terminal Nodes）：度数为零的节点就是树叶。图2-51中的K、L、F、G、M、I、J就是树叶。图2-52中有4个树叶节点，即E、C、H、I。

·父节点（Parent）：一个节点连接的上一层节点。如图2-51所示，F的父节点为B，而B的父节点为A，通常在绘制树形图时，我们会将父节点画在子节点的上方。

·子节点（Children）：一个节点连接的下一层节点。如图2-51所示，A的子节点为B、C、D，而B的子节点为E、F。

图2-52　有4个树叶节点的树

·祖先（Ancestor）和子孙（Descendent）：所谓祖先，是指从树根到该节点路径上所包含的节点；而子孙则是从该节点往下追溯，子树中的任一节点。在图2-51中，K的祖先为A、B、E节点，H的祖先为A、D节点，B的子孙为E、F、K、L节点。

·兄弟节点（Sibling）：有共同父节点的节点。在图2-51中，B、C、D为兄弟节点，H、I、J也为兄弟节点。

·非终端节点（Nonterminal Node）：树叶以外的节点，如图2-51中的A、B、C、D、E、H。

·同代（Generation）：在同一棵树中具有相同层数的节点，如图2-51中的E、F、G、H、I、J或B、C、D。

·森林（Forest）：n（n≥0）棵互斥树的集合。将一棵大树移去树根即为森林。例如图2-53所示为包含三棵树的森林。

图2-53　森林

2.7.2　二叉树

一般树结构在计算机内存中的存储方式以链表（Linked List）为主。对于n叉树来说，每个节点的度数都不相同，当n=2时，它的链接浪费率最低。为了避免树结构空间浪费的缺点，所以我们最常使用二叉树（Binary Tree）结构来取代其他的树结构。

二叉树（又称为Knuth树）是一个由有限节点所组成的集合，此集合可以为空集合，或由一个树根及其左右两棵子树组成。简单地说，二叉树最多只能有两个子节点，就是度数小于或等于2。二叉树每个节点在计算机中的数据结构如图2-54所示。

图2-54　二叉树节点在计算机中的数据结构示意图

二叉树和一般树的不同之处整理如下：

（1）树不可为空集合，而二叉树可以。

（2）树的度数为d≥0，而二叉树的节点度数为0≤d≤2。

（3）树的子树间没有次序关系，而二叉树有。

下面我们来看一棵实际的二叉树，如图2-55所示。

图2-55　二叉树

图2-55是以A为根节点的二叉树，包含以B、D为根节点的两棵互斥的左子树和右子树，如图2-56所示。

图2-56　两棵互斥的左子树和右子树

以上这两棵左右子树属于同一种树结构，不过却是两棵不同的二叉树，原因是二叉树必须考虑前后次序关系。这点大家要特别注意。

由于二叉树应用得相当广泛，因此衍生了许多特殊的二叉树结构。我们分别介绍如下：

1．满二叉树（Fully Binary Tree）

如果二叉树的高度为h（h≥0），树的节点数为2^h-1，我们就称此树为“满二叉树”，如图2-57所示：

图2-57　满二叉树

2．完全二叉树（Complete Binary Tree）

完全二叉树的高度为h，所含的节点数小于2^h-1，但其节点的编号方式如同高度为h的满二叉树一样，从左到右、从上到下的顺序一一对应，如图2-58所示。

图2-58　完全二叉树和非完全二叉树

3．斜二叉树（Skewed Binary Tree）

当一个二叉树完全没有右节点或左节点时，我们就把它称为左斜二叉树或右斜二叉树，如图2-59所示。

图2-59　左斜二叉树和右斜二叉树

4．严格二叉树（Strictly Binary Tree）

二叉树中的每一个非终端节点均有非空的左右子树，如图2-60所示。

图2-60　严格二叉树

2.7.3　树转化为二叉树的算法

对于将一般树结构转化为二叉树，使用的方法称为“Child-Sibling”（Leftmost-Child-Next-Right-Sibling，左儿子右兄弟表示法）法则。以下是其执行步骤：

（1）将节点的所有兄弟节点用横线连接起来。

（2）只保留与最左子节点的连接，删掉其他所有与子节点间的连接。

（3）右子树都顺时针旋转45度。

按照下面的范例实践一次，就可以有更清楚的认识。转化前的多叉树如图2-61所示。

图2-61　将此多叉树转化为二叉树

步骤01　将树的各层兄弟用横线连接起来，如图2-62所示。

图2-62　将各层兄弟用横线连接起来

步骤02　只保留最左边的父子节点的连接，删掉其他所有子节点间的连接，如图2-63所示。

图2-63　只保留最左边的父子节点的连接

步骤03　右子树都顺时针旋转45度，如图2-64所示。

图2-64　顺时针旋转45度

2.7.4　二叉树转化为树的算法

既然树可以转化为二叉树，当然也可以将二叉树转化为树（多叉树），如图2-65所示。

图2-65　将此二叉树转化为多叉树

这其实就是树转化为二叉树的逆向步骤，方法也很简单。首先右子树都逆时针旋转45度，如图2-66所示。

图2-66　逆时针旋转45度

左子树（ABE）（DG）代表父子关系，而右子树（BCD）（EF）（GH）代表兄弟关系，按这种父子关系增加连接，同时删除兄弟节点间的连接，结果如图2-67所示。

图2-67　按层增加父子关系的连接，同时删除兄弟节点间的连接

2.8 图简介

树结构用于描述节点与节点之间“层次”的关系，而图结构却是讨论两个顶点之间“连通与否”的关系，在图中连接两个顶点的边，如果填上加权值（也可以称为成本），这类图就被称为“网络”。图在生活中的应用非常普遍，如图2-68所示。

图2-68　图的应用在生活中非常普遍

图除了应用在算法和数据结构的最短路径搜索、拓扑排序中之外，还能应用在系统分析中以时间为评审标准的性能评审技术（Performance Evaluation and Review Technique，PERT），或者像“IC电路设计”“交通网络规划”等应用。注意：在图论中，图的定义有特定的含义。

城市地铁路线的规划就是图的应用之一，如图2-69所示。

图2-69　城市地铁路线的规划就是图的应用之一

图的理论（简称图论）起源于1736年，是一位瑞士数学家欧拉（Euler）为了解决“哥尼斯堡”问题所想出来的一种理论，就是著名的“七桥问题”。简单来说，就是有7座横跨4个城市的大桥。欧拉所思考的问题是这样的，“是否有人在每一座桥梁只经过一次的情况下，能够把所有地方都走过一次且回到原点。”图2-70所示为“七桥问题”的示意图。

图2-70　“七桥问题”示意图

欧拉当时使用的方法是以图结构来进行分析。他先以顶点表示城市，以边表示桥梁，把连接每个顶点的边数称为该顶点的度数。我们将以如图2-71所示的简图来表示“七桥问题”。

图2-71　以图的抽象方式表示七桥问题——欧拉环

最后欧拉得出一个结论：“当所有顶点的度数都为偶数时，才能从某顶点出发，经过每条边一次，再回到起点。”也就是说，在图2-71中，每个顶点的度数都是奇数，所以欧拉思考的问题是不可能发生的。这个理论就是有名的“欧拉环”（Eulerian Cycle）理论。

但是，如果条件改成从某顶点出发，经过每条边一次，不一定要回到起点，即只允许其中两个顶点的度数是奇数，其余则必须全部为偶数，符合这样的结果就称为欧拉链（Eulerian Chain），如图2-72所示。

图2-72　欧拉链

图的定义

图是由“顶点”和“边”所组成的集合，通常用G=（V，E）来表示，其中V是所有顶点组成的集合，而E代表所有边组成的集合。图的种类有两种：一种是无向图，另一种是有向图。无向图以（V1，V2）表示其边，而有向图则以<V1，V2>表示其边。

1．无向图

无向图（Graph）是一种边没有方向的图，即同边的两个顶点没有次序关系，如图2-73所示。例如（V_1，V_2）与（V_2，V_1）代表的是相同的边。

图2-73　无向图

```
V={A,B,C,D,E}
E={(A,B),(A,E),(B,C),(B,D),(C,D),(C,E),(D,E)}
```

2．有向图

有向图（Digraph）是一种每一条边都可以使用有序对<V_1，V_2>来表示的图，并且<V_1，V_2>与<V_2，V_1>表示两个方向不同的边，<V_1，V_2>是指以V_1为尾端指向头部V_2的边，如图2-74所示。

图2-74　有向图

```
V={A,B,C,D,E}
E={<A,B>,<B,C>,<C,D>,<C,E>,<E,D>,<D,B>}
```

2.9 哈希表

哈希表是一种保存记录或数据的连续存储空间，通过使用哈希函数可以快速存取与查找数据，如图2-75所示。所谓哈希函数（Hashing Function），就是将记录或数据本身的键值经由特定数学函数的运算或使用其他的方法转换成对应记录或数据的存储地址。

图2-75　哈希表

下面我们来介绍哈希函数的相关名词。

·bucket（桶）：哈希表中存储数据的位置，每一个位置对应一个唯一的地址（bucket address）。桶就好比一个记录或数据。

·slot（槽）：每一个记录中可能包含好几个字段，而槽指的就是“桶”中的字段。

·collision（碰撞）：如果两个不同的数据经过哈希函数运算后对应相同的地址，就称为碰撞。

·溢出：如果数据经过哈希函数运算后，所对应的桶已满，就会使桶发生溢出。

·哈希表：保存记录的连续存储空间。哈希表是一种类似数据表的索引表，其中可分为n个桶，每个桶又可分为m个槽，如表2-4所示。

表2-4　索引表

·同义词（Synonym）：若两个标识符I_1和I_2经哈希函数运算后所得的数值相同，即f（I_1）=f（I_2），则称I_1与I_2对于哈希函数f是同义词。

· 加载密度（Loading Factor）：所谓加载密度，是指标识符的使用数目除以哈希表内槽的总数：

α值越大，表示哈希存储空间的使用率越高，碰撞或溢出的概率也会越高。

·完美哈希（perfect hashing）：指既没有碰撞又没有溢出的哈希函数。

在此建议大家，在设计哈希函数时应该遵循以下几个原则：

（1）降低碰撞和溢出的产生。

（2）哈希函数不宜过于复杂，越容易计算越佳。

（3）尽量把文字的键值转换成数字的键值，以利于哈希函数的运算。

（4）所设计的哈希函数计算得到的值尽可能均匀地分布在每一桶中，不要太过于集中在某些桶内，这样就可以降低碰撞并减少溢出的处理。

2.10 课后习题

1.请简单说明堆栈与队列的主要特性。

2.数据结构主要用于表示数据在计算机内存中存储的位置和模式，通常可以分为哪三种类型？

3.在单向链表类型的数据结构中，根据所删除节点的位置会有哪三种不同的情况？

4.什么是类神经网络？

5.请说明稀疏矩阵的定义，并举例说明。

6.请简单介绍GPU。

7.机器学习是什么？有哪些应用？

8.请解释下列哈希函数的相关名词。

（1）bucket（桶）

（2）同义词

（3）完美哈希

（4）碰撞

9.一般树结构在计算机内存中的存储方式以链表为主，对于n叉树来说，我们必须取n为链接个数的最大固定长度，请说明为了避免树结构存储空间浪费的缺点，我们最常使用二叉树结构来取代树结构。

10.请将下面的树转化为二叉树。

第3章　分治法

分治法（Divide and Conquer，也称为“分而治之法”）是一种很重要的算法，我们可以应用分治法来逐一拆解复杂的问题，核心思想是将一个难以直接解决的大问题按照相同的概念分割成两个或更多的子问题，以便各个击破，即“分而治之”。

下面我们以一个实际的例子来说明。如果有8幅很难画的图，我们可以分成2组各4幅画来完成，如果还是觉得太复杂，继续再分成4组，每组各两幅画来完成，采用相同模式反复分割问题（见图3-1），这就是最简单的分治法的核心思想。

图3-1　分治法解决问题的例子

3.1 分治法简介

其实任何一个可以用程序求解的问题所需的计算时间都与其规模与复杂度有关，问题的规模越小，越容易直接求解，因此可以使子问题的规模不断缩小，直到这些子问题简单到可以解决，最后将各个子问题的解合并，得到原问题的答案。举个例子来说，如果你被委托制作一个项目的计划书，这个计划书有8个章节的主题，只靠一个人独立完成，不但会花较长时间，而且计划书中有些主题的内容也有可能不是自己的专长，这个时候就可以按照这8个章节的特性分工给2位项目专员去完成。不过，为了让计划书更快完成，可以找到适合的分类，再分别将其分割成2章一组，并分配给更多不同的项目专员。如此一来，每位项目专员只需负责其中的2个章节。经过这样的分配，就可以将原先的大计划书简化为4个小项目，再委托给4位项目专员去完成。以此类推，上述问题的解决方案示意图如图3-2所示。

图3-2　用分治法将大项目分割为子项目来解决的例子

分治法也可以应用在数字的分类与排序上，例如要以人工的方式将散落在地上的打印稿从第1页开始整理并排序到第100页。我们有两种方法，一种方法是逐一捡起打印稿，并逐一按页码顺序插入正确的位置。但是，这样的方法有一个缺点，就是排序和整理的过程较为繁杂，而且较花时间。

另一种方法是应用分治法的原理，先将页码1到页码10放在一起，再将页码11到页码20放在一起，以此类推，最后将页码91到页码100放在一起。也就是说，将原先的100页分类为10个页码区间，然后我们分别对10堆页码进行整理，最后将页码从小到大的分组合并起来，就能较为轻松地恢复到原先的稿件顺序。通过分治法，我们可以让原先

复杂的问题变成规则更简单、数量更少且更容易解决的小问题。

3.2 递归法

递归是一种很特殊的算法，分治法和递归法很像一对孪生兄弟，都是将一个复杂的算法问题进行分解，让规模越来越小，最终使子问题容易求解。递归在早期人工智能所用的语言（如Lisp、Prolog）中，几乎是整个语言运行的核心，现在许多程序设计语言（包括C#、C、C++、Java、Python等）都具备递归功能。简单来说，对程序设计人员而言，“函数”（或称为子程序）不单纯是能够被其他函数调用（或引用）的程序单元，在某些程序设计语言中还提供了自己调用自己的功能，这种调用的功能就是所谓的“递归”。

从程序设计语言的角度，谈到递归的正式定义，我们可以这样描述：假如一个函数或子程序是由自身所定义或调用的，就称为递归。递归至少要定义两个条件：①包括一个可以反复执行的递归过程，②一个跳出递归过程的出口。

提示

“尾递归”（Tail Recursion）就是函数或子程序的最后一条语句为递归调用，因为每次调用后，再返回前一次调用处继续执行的第一条语句就是return语句，所以不需要再进行任何运算工作了。

阶乘函数是数学上很有名的函数，对递归法而言，也可以看成是很典型的范例，我们一般以符号“！”来代表阶乘。例如4的阶乘可写为4！，n！则表示为：

```
n！=n* (n-1) * (n-2) *…*1
```

我们可以逐步分解它的运算过程，以观察其规律性：

```
5! = (5 * 4!)
= 5 * (4 * 3!)
= 5 * 4 * (3 * 2!)
= 5 * 4 * 3 * (2 * 1)
```

```
= 5 * 4 * (3 * 2)
= 5 * (4 * 6)
= (5 * 24)
= 120
```

从上述过程中，我们可以发现阶乘问题非常适合以递归法与堆栈来解决。因为它满足了递归的两大特性：①反复执行的递归过程，②跳出递归过程的出口。

3.3　汉诺塔算法

法国数学家Lucas在1883年介绍了一个十分经典的汉诺塔（Tower of Hanoi）智力游戏，是使用递归法与堆栈概念来解决问题的典型范例，如图3-3所示。内容是说在古印度神庙中有三根木桩，天神希望和尚们把某些数量、大小不同的圆盘从第一个木桩全部移动到第三个木桩。

图3-3　用递归法与堆栈概念来解决汉诺塔问题

从更精确的角度来说，汉诺塔问题可以这样描述：假设有1号、2号、3号三个木桩和n个大小均不相同的圆盘（Disc），从小到大编号为1，2，3，…，n，编号越大，直径越大。开始的时候，n个圆盘都套在1号木桩上，现在希望能找到将1号木桩上的圆盘借助2号木桩当中间桥梁全部移到3号木桩上移动次数最少的方法。不过在搬动时必须遵守下列规则：

（1）直径较小的圆盘永远只能置于直径较大的圆盘上。

（2）圆盘可任意地从任何一个木桩移到其他的木桩上。

（3）每一次只能移动一个圆盘，而且只能从最上面的开始移动。

现在我们考虑n=1~3的情况，以图示示范求解汉诺塔问题的步骤：

1．n=1

当然是直接把圆盘从1号木桩移动到3号木桩，如图3-4所示。

图3-4　直接把圆盘从1号木桩移动到3号木桩

2．n=2

步骤01　将圆盘1从1号木桩移动到2号木桩，如图3-5所示。

图3-5　将圆盘1从1号木桩移动到2号木桩

步骤02　将圆盘2从1号木桩移动到3号木桩，如图3-6所示。

图3-6　将圆盘2从1号木桩移动到3号木桩

步骤03　将圆盘1从2号木桩移动到3号木桩，就完成了，如图3-7所示。

图3-7　将圆盘1从2号木桩移动到3号木桩

步骤04　移动完成的状态如图3-8所示。

图3-8　移动完成的状态

结论：移动了2^2-1=3次，圆盘移动的次序为1、2、1（圆盘次序）。

步骤：1→2，1→3，2→3（木桩次序）。

3．n=3

步骤01　将圆盘1从1号木桩移动到3号木桩，如图3-9所示。

图3-9　将圆盘1从1号木桩移动到3号木桩

步骤02　将圆盘2从1号木桩移动到2号木桩，如图3-10所示。

图3-10　将圆盘2从1号木桩移动到2号木桩

步骤03　将圆盘1从3号木桩移动到2号木桩，如图3-11所示。

图3-11　将圆盘1从3号木桩移动到2号木桩

步骤04　将圆盘3从1号木桩移动到3号木桩，如图3-12所示。

图3-12　将圆盘3从1号木桩移动到3号木桩

步骤05　将圆盘1从2号木桩移动到1号木桩，如图3-13所示。

图3-13　将圆盘1从2号木桩移动到1号木桩

步骤06　将圆盘2从2号木桩移动到3号木桩，如图3-14所示。

图3-14　将圆盘2从2号木桩移动到3号木桩

步骤07　将圆盘1从1号木桩移动到3号木桩，就完成了，如图3-15所示。

图3-15　将圆盘1从1号木桩移动到3号木桩

步骤08　移动完成的状态如图3-16所示。

图3-16　移动完成的状态

结论：移动了2^3-1=7次，圆盘移动的次序为1、2、1、3、1、2、1（圆盘次序）。

步骤：1→3，1→2，3→2，1→3，2→1，2→3，1→3（木桩次序）。

当有4个圆盘时，我们实际操作后（在此不用插图说明），圆盘移动的次序为1、2、1、3、1、2、1、4、1、2、1、3、1、2、1，而移动木桩的顺序为1→2、1→3、2→3、1→2、3→1、3→2、1→2、1→3、2→3、2→1、3→1、2→3、1→2、1→3、2→3，移动次数为2^4-1=15。

当n不大时，大家可以逐步用图解办法解决问题，当n的值较大时就十分伤脑筋了。事实上，我们可以得出一个结论，例如当有n个圆盘时，可将汉诺塔问题归纳成三个步骤，如图3-3所示。

步骤01　将n-1个圆盘从木桩1移动到木桩2。

步骤02　将第n个最大的圆盘从木桩1移动到木桩3。

步骤03　将n-1个圆盘从木桩2移动到木桩3。

3.4 快速排序法

排序（Sorting）算法是经常使用的一种算法，就是将一组数据按照某一个特定规则重新排列，使数据具有递增或递减的次序关系。按照特定规则用以排序的依据称为键（Key），它所含的值称为“键值（Key Value）”。排序的各种算法称得上是数据结构这门学科的精髓所在。每一种排序方法都有其适用的情况与数据类型。例如，常见的跑步比赛最终都会分出排名，如图3-17所示。

图3-17　跑步比赛最终都会分出排名

快速排序（Quick Sort）是由C.A.R.Hoare提出来的。快速排序法又称分割交换排序法，是目前公认最佳的排序法，也是使用“分而治之”的方式。这种算法会先在数据中找到一个虚拟的中间值，并按此中间值将所有打算排序的数据分为两部分，其中小于中间值的数据放在左边，而大于中间值的数据放在右边，再以同样的方式分别处理左右两边的数据，直到排序完为止。操作与分割步骤如下：

假设有n项记录R_1、R_2、R_3、…、Rn，其键值为K_1、K_2、K_3、…、K_n。

步骤01　先假设K的值为第一个键值。

步骤02　从左向右找出键值K_i，使得$K_i>K$。

步骤03　从右向左找出键值K_j，使得$K_j<K$。

步骤04　如果$i<j$，那么K_i与K_j互换，并回到步骤02。

步骤05　若$i\geq j$，则将K与K_j交换，并以j为基准点分割成左右部分。

步骤06　针对左右两边进行步骤01至步骤05的操作，直到左半边键值等于右半边键值为止。

下面示范用快速排序法对数据进行排序的过程，原始数据如图3-18所示。

图3-18　原始数据

步骤01　因为i<j，故交换K_i与K_j，如图3-19所示，然后继续比较。

图3-19　第一轮排序

步骤02　因为i<j，故交换K_i与K_j，如图3-20所示，然后继续比较。

图3-20　第二轮排序

步骤03　因为i≥j，故交换K与K_j，并以j为基准点分割成左右两半，如图3-21所示。

图3-21　第三轮排序

经过上述这几个步骤，大家可以将小于键值K的数据放在左半边；大于键值K的数据放在右半边，按照上述排序过程对左右两边再分别排序，过程如图3-22所示。

图3-22　左右两边分别排序

3.5 合并排序法

合并排序法（Merge Sort）的工作原理是针对已排序好的两个或两个以上的数列（或数据文件），通过合并的方式将其组合成一个大的且已排好序的数列（或数据文件）。其步骤如下：

步骤01　将N个长度为1的键值成对地合并成N/2个长度为2的键值组。

步骤02　将N/2个长度为2的键值组成对地合并成N/4个长度为4的键值组。

步骤03　将键值组不断地合并，直到合并成一组长度为N的键值组为止。

下面我们用38、16、41、72、52、98、63、25数列从小到大排序的过程来说明合并排序法的基本演算流程，如图3-23所示。

图3-23　合并排序法的演算流程

图3-23展示的合并排序法例子是一种最简单的合并排序，又称为2路（2-way）合并排序，主要是把原来的数列视作N个已排好序且长度为1的数列，再将这些长度为1的数列两两合并，结合成N/2个已排好序且长度为2的数列；同样的做法，再按序两两合并，合并成N/4个已排好序且长度为4的数列，以此类推，最后合并成一个已排好序且长度为N的数列。

现在将排序步骤整理如下：

步骤01　将N个长度为1的数列合并成N/2个已排序妥当且长度为2的数列。

步骤02　将N/2个长度为2的数列合并成N/4个已排序妥当且长度为4的数列。

步骤03　将N/4个长度为4的数列合并成N/8个已排序妥当且长度为8的数列。

步骤07　将$N/2^{i-1}$个长度为2^{i-1}的数列合并成$N/2^{i}$个已排序妥当且长度为2^{i}的数列。

3.6　二分查找法

在数据处理过程中，是否能在最短时间内查找到所需要的数据是信息从业人员最为关心的问题。所谓查找（Search，或搜索），指的是从数据文件中找出满足某些条件的记录。用以查找的条件称为“键值”，就如同排序所用的键值一样。例如，我们在电话簿中找某人的电话号码，这个人的姓名就成为在电话簿中查找电话号码的键值。我们在日常生活中每天也会查找许多目标物品，如图3-24所示。

图3-24　我们在日常生活中每天也会查找许多目标物品

如果要查找的数据已经事先排好序了，就可以使用二分查找法进行查找，这也是分治法的应用。二分查找法是将数据分割成两等份，再比较键值与中间值的大小，如果键值小于中间值，就可以确定要查找的数据在前半段，否则在后半段。如此分割数次，直到找到或确定不存在为止。

例如，已排序的数列为2、3、5、8、9、11、12、16、18，所要查找的值为11，步骤如下：

步骤01　首先与中间值（第5个数值）9比较，如图3-25所示。

图3-25　先和中间值比较

步骤02　因为11 > 9，所以和后半段的中间值12比较，如图3-26所示。

图3-26　再和后半段的中间值比较

步骤03　因为11 < 12，所以和后半段的前半段的中间值11比较，如图3-27所示。

图3-27　再和后半段的前半段的中间值比较

步骤04　因为11=11，所以查找完成（找到了）。若不相等，则表示找不到。

3.7 课后习题

1.试简述分治法的核心精神。

2.递归至少要定义哪两个条件？

3.请问使用二分查找法的前提条件是什么？

4.有关二分查找法，下列叙述哪一个是正确的？

A．文件必须事先排序

B．当排序的数据量非常少时，二分查找法的速度比顺序查找法的速度慢

C．排序的复杂度比顺序查找法高

D．以上都正确

5.试说明在汉诺塔问题中，移动n个圆盘所需的最小移动次数是多少。

6.待排序关键字的值如下，请使用合并排序法列出每一回合排序的结果：

11、8、14、7、6、8+、23、4

7.请简单介绍快速排序法。

第4章　贪心法

贪心法（Greed Method）又称为贪婪算法，方法是从某一起点开始，在每一个解决问题的步骤使用贪心原则，都采取在当前状态下最有利或最优化的选择，不断地改进该解答，持续在每一个步骤中选择最佳的方法，并且逐步逼近给定的目标，当达到某一个步骤不能再继续前进时，算法就停止，以尽可能快的方法求得更好的解。贪心法可以解决与优化有关的大部分问题。

贪心法的解题思路尽管是把求解的问题分成若干个子问题，不过有时还是不能保证求得的最后解是最佳的或最优化的解，因为贪心法容易过早做出决定，所以只能求出满足某些约束条件的解，而有时在某些问题上还是可以得到最优解的，例如求图结构的最小生成树、最短路径与哈夫曼编码、机器学习等方面。许多公共运输系统都会用到最短路径的理论，如图4-1所示。

图4-1　许多公共运输系统都会用到最短路径的理论

4.1 贪心法简介

我们来看一个简单的例子（后面的货币系统不是现实的情况，只是为了举例）。假设我们去便利商店购买几罐可乐（见图4-2），要价24元，我们付给售货员100元，希望找的钱不要太多硬币，即硬币的总数量最少，该如何找钱呢？假如目前的硬币有50元、10元、5元、1元4种，从贪心法的策略来说，应找的钱总数是76元，所以一开始选择50元的硬币一枚，接下来选择10元的硬币两枚，最后选择5元的硬币和1元的硬币各一枚，总共4枚硬币，这个结果也确实是最佳的解。

图4-2　购买可乐

贪心法也很适合用于旅游某些景点的判断，假如我们要从图4-3中的顶点5走到顶点3，最短的路径是什么呢？采用贪心法，当然是先走到顶点1，接着选择走到顶点2，最后从顶点2走到顶点5，这样的距离是28。可是从图4-3中我们发现直接从顶点5走到顶点3才是最短的距离，说明在这种情况下，没办法以贪心法的规则来找到最佳的解。

图4-3　用贪心法计算前往旅游景点的最短路径，但是不一定会找到最佳的解

4.2 最小生成树

生成树又称“花费树”“成本树”或“值树”，一个图的生成树（Spanning Tree）就是以最少的边来连通图中所有的顶点，且不造成回路（Cycle）的树结构。假设为树的边加上一个权重（Weight）值，这种图就成为一种“加权图（Weighted Graph）”。如果这个权重值代表两个顶点间的距离（Distance）或成本（Cost），这类图就被称为网络（Network），如图4-4所示。

图4-4　加权图也被称为网络

假如想知道从某个点到另一个点间的路径成本，例如从顶点1到顶点5有（1+2+3）、（1+6+4）和5三条路径成本，“最小成本生成树（Minimum Cost Spanning Tree）”就是路径成本为5的生成树，如图4-5最右边的图。

图4-5　最小成本生成树为最右边的图

在一个加权图中找到最小成本生成树是相当重要的，因为许多工作都可以用图来表示，例如从北京到上海的距离或花费等。接着将介绍两种以“贪婪法则”（Greedy Rule）为基础找出一个无向连通图的最小生成树的常见方法，分别是Prim算法和Kruskal算法。

4.2.1 Prim算法

Prim算法又称P氏法，对于一个加权图G=（V，E），假设V={1，2，…，n}、U={1}，也就是说，U和V是两个顶点的集合。从U-V所产生的集合中找出一个顶点x，该顶点x能与U集合中的某点形成最小成本的边，且不会造成回路。然后将顶点x加入U集合中，反复执行同样的步骤，一直到U集合等于V集合（U=V）为止。

接下来，我们将实际运用Prim算法求出图4-6的最小成本生成树。

图4-6　以此加权图为例使用Prim算法生成最小成本生成树

步骤01　从图4-6可得V={1，2，3，4，5，6}、U=1，从V-U所产生的集合{2，3，4，5，6}中找一个顶点与U顶点能形成最小成本的边，得到图4-7。

图4-7　V-U={2，3，4，6}，U={1，5}

步骤02　从V-U的顶点中找出与U顶点能形成最小成本的边，得到图4-8。

图4-8　U={1，5，6}，V-U={2，3，4}

步骤03　同理，找到顶点4，如图4-9所示。

图4-9　U={1，5，6，4}V-U={2，3}

步骤04　同理，找到顶点3，如图4-10所示。

图4-10　找到顶点3

步骤05　同理，找到顶点2，如图4-11所示。

图4-11　找到顶点2，最后得到了最小成本生成树

4.2.2 Kruskal算法

Kruskal算法是将各边按权值大小从小到大排列，接着从权值最低的边开始建立最小成本生成树，如果加入的边会造成回路，就舍弃不用，直到加入n-1个边为止。这个方法看起来似乎不难，我们直接来看看如何以Kruskal算法得到如图4-12所示的例图对应的最小成本生成树。

图4-12　以此加权图为例使用Kruskal算法来生成最小成本生成树

步骤01　把所有边的成本列出并从小到大排序，如表4-1所示。

表4-1　所有边的成本从小到大排序

步骤02　选择成本最低的一条边作为建立最小成本生成树的起点，如图4-13所示。

图4-13　选择最小成本生成树的起点

步骤03　以步骤1所建立的表格按序加入边，如图4-14所示。

图4-14　按序加入边

步骤04　加入CD边会形成回路，所以直接跳过，如图4-15所示。

图4-15　跳过CD边

完成图如图4-16所示。

图4-16　最后得到了最小成本生成树

4.3 图的最短路径法

在一个有向图G=（V，E）中，它的每一条边都有一个比例常数W（Weight，权重）与之对应，如果想求图G中某一个顶点V0到其他顶点的权重总和最小，这类问题被称为最短路径问题（The Shortest Path Problem）。由于交通运输工具和通信工具的便利与普及，因此两地之间发生货物运送或者进行信息传递时，最短路径问题随时都可能因为这种需求而产生，简单来说，就是找出两个顶点间可通行的快捷路径。

我们在4.2节中所介绍的最小成本生成树就是计算连通网络中每一个顶点所需的最少花费，但是连通树中任意两个顶点之间的路径不一定是一条花费最少的路径，这也是本节将研究最短路径问题的主要理由。一般使用的算法有两种：Dijkstra算法和A*算法。

4.3.1 Dijkstra算法

一个顶点到多个顶点的最短路径通常使用Dijkstra算法求得，Dijkstra算法如下：

假设S={V_i|V_i∈V}，且V_i在已发现的最短路径中，其中V_0∈S是起点。

假设w∉S，定义Dist（w）是从V_0到w的最短路径，这条路径除了w外必属于S，且有下列几点特性：

（1）若u是当前所找到最短路径的下一个节点，则u必属于V-S集合中最小成本的边。

（2）若u被选中，将u加入S集合中，则会产生当前的从V_0到u的最短路径，对于w∉S，DIST（w）被改变成DIST（w）——Min{DIST（w），DIST（u）+COST（u，w）}。

从上述算法中，我们可以推演出如下步骤：

步骤01

```
G = (V, E)
D[k] = A[F, k]，其中k从1到N
S = {F}
V = {1, 2, …… N}
```

D为一个N维数组，用来存放某一顶点到其他顶点的最短距离。

F表示起始顶点。

A[F，I]为顶点F到I的距离。

V是网络中所有顶点的集合。

E是网络中所有边的组合。

S也是顶点的集合，其初始值是S={F}。

步骤02　从V-S集合中找到一个顶点x，使D（x）的值为最小值，并把x放入S集合中。

步骤03　按照公式D[I]=min（D[I]，D[x]+A[x，I]），其中（x，I）∈E，用于调整D数组的值，I是指x的相邻各顶点。

步骤04　重复执行步骤02，一直到V-S是空集合为止。

现在直接看一个例子，在图4-17中找出顶点5到各顶点间的最短路径。

图4-17　以此图为例找出顶点5到各顶点的最短路径

做法相当简单，首先从顶点5开始，找出顶点5到各顶点间的最短距离，到达不了的以∞表示。步骤如下：

步骤01　D[0]=∞，D[1]=12，D[2]=∞，D[3]=20，D[4]=14。在其中找出值最小的顶点并加入S集合中：D[1]。

步骤02　D[0]=∞，D[1]=12，D[2]=18，D[3]=20，D[4]=14。D[4]最小，加入S集合中。

步骤03　D[0]=26，D[1]=12，D[2]=18，D[3]=20，D[4]=14。D[2]最小，加入S集合中。

步骤04　D[0]=26，D[1]=12，D[2]=18，D[3]=20，D[4]=14。D[3]最小，加入S集合中。

步骤05　加入最后一个顶点即可得到表4-2。

表4-2　顶点S到各顶点间的最短距离

从顶点5到其他各顶点的最短距离如下。

顶点5-顶点0：26。

顶点5-顶点1：12。

顶点5-顶点2：18。

顶点5-顶点3：20。

顶点5-顶点4：14。

4.3.2 A*算法

前面所介绍的Dijkstra算法在寻找最短路径的过程中是一个效率不高的算法，这是因为这个算法在寻找起点到各个顶点的距离的过程中，无论哪一个顶点都要实际计算起点与各个顶点间的距离，以便获得最后的一个判断：到底哪一个顶点距离与起点最近。

也就是说，Dijkstra算法在带有权重值（Weight Value或Cost Value（成本值））的有向图中，寻找最短路径的方式只是简单地使用广度优先进行查找，完全忽略了许多有用的信息，这种查找算法会消耗许多系统资源，包括CPU的运算时间与内存的存储空间。如果能有更好的方式帮助我们预估从各个顶点到终点的距离，善加利用这些信息就可以预先判断图上有哪些顶点离终点的距离较远，以便直接略过对这些顶点的查找，这种更有效率的查找算法有助于程序以更快的方式找到最短路径。

在这种需求的考虑下，A*算法可以说是Dijkstra算法的改进版，它结合了在路径查找过程中从起点到各个顶点的“实际权重”和各个顶点预估到达终点的“推测权重”（Heuristic Cost，或称为试探权重）两个因素，可以有效地减少不必要的查找操作，从而提高查找最短路径的效率，如图4-18所示。

图4-18　两种算法的对比

因此，A*算法也是一种最短路径算法，与Dijkstra算法不同的是，A*算法会预先设置一个“推测权重”，并在查找最短路径的过程中将“推测权重”一并纳入决定最短路径的考虑因素。所谓“推测权重”，就是根据事先知道的信息给定一个预估值，结合这个预估值，A*算法可以更有效地查找最短路径。

例如，在一个已知“起点位置”与“终点位置”的迷宫中寻找最短路径，因为事先知道迷宫的终点位置，所以可以采用顶点和终点的欧氏几何平面直线距离（Euclidean Distance，即数学定义中的平面内两点间的距离）作为该顶点的推测权重。

提示

有哪些常见的距离评估函数？

在A*算法中，用来计算推测权重的距离评估函数除了上面提到的欧氏几何平面直线距离外，还有许多距离评估函数可供选择，例如曼哈顿距离（Manhattan Distance）和切比雪夫距离（Chebysev Distance）等。对于二维平面上的两个点（x1，y1）和（x2，y2），这三种距离的计算方式如下：

·曼哈顿距离

D=|x1-x2|+|y1-y2|

·切比雪夫距离

D=max（|x1-x2|，|y1-y2|）

·欧氏几何平面直线距离

A*算法并不像Dijkstra算法只考虑从起点到下一个顶点的实际权重（就是实际距离）来决定下一步要尝试的顶点。不同的是，A*算法在计算从起点到各个顶点的权重时，会同步考虑从起点到下一个顶点的实际权重，再加上该顶点到终点的推测权重，以推算出该顶点从起点到终点的权重。再从中选出一个权重最小的顶点，并将该顶点标示为已查找完毕。接着计算从查找完毕的顶点出发到各个顶点的权重，并从中选出一个权重最小的顶点，遵循前面同样的做法，将该顶点标示为已查找完毕的顶点，以此类推，反复进行同样的步骤，一直到抵达终点，才结束查找的工作，最终得到最短路径的最优解。

做一个简单的总结，实现A*算法的主要步骤如下：

步骤01　首先确定各个顶点到终点的“推测权重”。“推测权重”的计算方法可以采用各个顶点和终点之间的直线距离（四舍五入后的值），直线距离的计算从上述三种距离计算方式中选择一种即可。

步骤02　分别计算从起点可抵达的各个顶点的权重，计算方法是由起点到该顶点的“实际权重”加上该顶点抵达终点的“推测权重”。计算完毕后，选出权重最小的点，并标示为查找完毕的点。

步骤03　接着计算从查找完毕的顶点出发到各个顶点的权重，并从中选出一个权重最小的顶点，将其标示为查找完毕的顶点。以此类推，反复进行同样的计算过程，一直到抵达最后的终点。

A*算法适用于可以事先获得或预估各个顶点到终点距离的情况，但是万一无法获得各个顶点到目的地终点的距离信息，就无法使用A*算法。虽然A*算法是Dijkstra算法的改进版，但并不是在任何情况下A*算法的效率都优于Dijkstra算法。例如，当“推测权重”的距离和实际两个顶点间的距离相差甚大时，A*算法的查找效率可能比Dijkstra算法更差，甚至还会误导方向，从而造成无法得到最短路径的最终解。

但是，如果推测权重所设置的距离和实际两个顶点间的真实距离误差不大时，A*算法的查找效率就远大于Dijkstra算法。因此，A*算法常被应用于游戏软件中玩家与怪物两种角色间的追逐行为，或者引导玩家以最有效率的路径及最便捷的方式快速突破游戏关卡，如图4-19所示。

图4-19　A*算法常被应用于游戏中角色追逐与快速突破关卡的设计

4.3.3 Floyd算法

由于Dijkstra算法只能求出某一点到其他顶点的最短距离，如果想求出图中任意两点甚至所有顶点间最短的距离，就必须使用Floyd算法。

Floyd算法定义如下：

（1）$A^k[i][j]=\min\{A^{k-1}[i][j]，A^{k-1}[i][k]+A^{k-1}[k][j]\}$（$k\geq1$），k表示经过的顶点，$A^k[i][j]$为从顶点i到顶点j经由k顶点的最短路径。

（2）$A^0[i][j]=COST[i][j]$（A^0等于COST），A^0为顶点i到顶点j之间的直线距离。

（3）$A^n[i，j]$代表顶点i到顶点j的最短距离，A^n便是我们所要求取的最短路径成本矩阵。

这样看来，Floyd算法似乎相当复杂难懂。下面直接以实例来说明它的算法。试着以Floyd算法求出图4-20中各个顶点间的最短路径。

图4-20　以此图为例使用Floyd算法求出各个顶点间的最短路径

步骤01　找到$A^0[i][j]=COST[i][j]$，A^0为不经任何顶点的成本矩阵。若没有路径，则以∞（无穷大）表示，如图4-21所示。

图4-21　求得A0矩阵

步骤02　找出$A^1[i][j]$从顶点i到顶点j经由顶点1的最短距离，并填入矩阵。

$A^1[1][2]=\min\{A^0[1][2]，A^0[1][1]+A^0[1][2]\}=\min\{4，0+4\}=4$

$A^1[1][3]=\min\{A^0[1][3]，A^0[1][1]+A^0[1][3]\}=\min\{11，0+11\}=11$

$A^1[2][1]=\min\{A^0[2][1]，A^0[2][1]+A^0[1][1]\}=\min\{6，6+0\}=6$

$A^1[2][3]=\min\{A^0[2][3]，A^0[2][1]+A^0[1][3]\}=\min\{2，6+11\}=2$

$A^1[3][1]=\min\{A^0[3][1]，A^0[3][1]+A^0[1][1]\}=\min\{3，3+0\}=3$

$A^1[3][2]=\min\{A^0[3][2]，A^0[3][1]+A^0[1][2]\}=\min\{\infty，3+4\}=7$

按序求出各顶点的值后可以得到A^1矩阵，如图4-22所示。

图4-22　得到A^1矩阵

步骤03　求出$A^2[i][j]$经由顶点2的最短距离。

$A^2[1][2]=\min\{A^1[1][2]，A^1[1][2]+A^1[2][2]\}=\min\{4，4+0\}=4$

$A^2[1][3]=\min\{A^1[1][3]，A^1[1][2]+A^1[2][3]\}=\min\{11，4+2\}=6$

按序求其他各顶点的值可得到A^2矩阵，如图4-23所示。

图4-23　得到A^2矩阵

步骤04　求出$A^3[i][j]$经由顶点3的最短距离。

$A^3[1][2]=\min\{A^2[1][2]，A^2[1][3]+A^2[3][2]\}=\min\{4，6+7\}=4$

$A^3[1][3]=\min\{A^2[1][3]，A^2[1][3]+A^2[3][3]\}=\min\{6，6+0\}=6$

按序求其他各顶点的值可得到A^3矩阵，如图4-24所示。

图4-24　得到A^3矩阵

所有顶点间的最短路径如矩阵A^3所示。

从上例可知，一个加权图若有n个顶点，则此方法必须执行n次循环，逐一产生A^1，A^2，A^3，…，A^k个矩阵。但因Floyd算法较为复杂，读者可以用4.3.1小节所讨论的Dijkstra算法，按序以各顶点为起始顶点，可以得到同样结果。

4.4 课后习题

1.试简述贪心法的主要概念。

2.什么是生成树？生成树应该包含哪些特点？

3.请简述用Prim算法求解一个无向连通图的最小生成树的主要过程。

4.请简述用Kruskal算法求解一个无向连通图的最小生成树的主要过程。

5.请简述A*算法的优点。

6.请用K氏法求出下图中的最小成本生成树。

7.在注有各地距离的图上（各地之间都是单行道），求各地之间的最短距离。分别求解以下两道题。

（1）写出下图的邻接矩阵。

（2）写出下图各个顶点之间最短距离的矩阵，表示出所求得的各个顶点之间的最短距离，即各地间最短的距离。

第5章　动态规划法

动态规划法（Dynamic Programming Algorithm，DPA）类似于分治法，在20世纪50年代初由美国数学家R.E.Bellman发明，用于研究多阶段决策过程的优化过程与求得一个问题的最优解。动态规划法主要的做法是：如果一个问题的答案与子问题相关的话，就能将大问题拆解成各个小问题，其中与分治法最大的不同是可以让每一个子问题的答案被存储起来，以供下次求解时直接取用。这样的做法不但可以减少再次计算的时间，而且可以将这些解组合成大问题的解，故而使用动态规划可以解决重复计算的问题。

5.1　动态规划法简介

动态规划法是分治法的延伸。当用递归法分割出来的问题“一而再，再而三”出现时，就可以运用记忆（Memorization）法来存储这些问题。与分治法不同的地方在于，动态规划法增加了记忆机制的使用，将处理过的子问题的答案记录下来，避免重复计算。

下面我们来看著名的斐波那契数列（Fibonacci Sequence）的求解。斐波那契数列的基本定义如下：

简单来说，这个数列的第0项是0、第1项是1，之后的各项的值是由其前面两项的值相加的结果（后面的每项值都是其前两项值之和）。

前面斐波那契数列是用类似分治法的递归法，如果改用动态规划法，那么已计算过的数据就不必重复计算了，也不会再往下递归，进而达到提高性能的目的。例如我们想求斐波那契数列的第4项数Fib（4），它的递归过程可以用图5-1表示。

图5-1　计算斐波那契数列的递归执行路径图

从上面的执行路径图中，我们可以看到递归调用了9次，而加法运算执行了4次，Fib（1）执行了3次，重复计算影响了执行性能。我们根据动态规划法的算法思路可以绘制出如图5-2所示的执行示意图。

图5-2　采用动态规划法计算斐波那契数列的执行路径图

前面提过动态规划法的精神是已计算过的数据不必重复计算。为了达到这个目的，我们可以先设置一个用来记录该斐波那契数列中的项是否已计算过的数组output，该数组中的每一个元素用来分别记录已被计算过的斐波那契数列

中的各项。不过在计算之前，该output数组的初值全部设置为空值（在C语言中，空值为NULL，在Python语言中，空值为None），当该斐波那契数列中的项被计算过后，就必须将该项计算得到的值存储到output数组中。举例来说，我们可以将F（0）记录到output[0]，F（1）记录到output[2]，以此类推。

每当要计算斐波那契数列中的项时，就先从output数组中判断，如果是空值，就进行计算，再将计算得到的斐波那契数列项存储到对应的output数组中（数组下标对应数列的项次），这样就可以确保斐波那契数列的每一项只被计算过一次。算法的执行过程如下：

（1）第一次计算F（0），按照斐波那契数列的定义，得到数值为0，将此值存入用来记录已计算斐波那契数列的数组中，即output[0]=0。

（2）第一次计算F（1），按照斐波那契数列的定义，得到数值为1，将此值存入用来记录已计算斐波那契数列的数组中，即output[1]=1。

（3）第一次计算F（2），依照斐波那契数列的定义，得到数值为F（1）+F（0），因为这两个数值都已计算过，因此可以直接计算output[1]+output[0]=1+0=1，将此值存入用来记录已计算斐波那契数列的数组中，即output[2]=1。

（4）第一次计算F（3），按照斐波那契数列的定义，得到数值为F（2）+F（1），因为这两个数值都已计算过，因此可以直接计算output[2]+output[1]=1+1=2，将此值存入用来记录已计算斐波那契数列的数组中，即output[3]=2。

（5）第一次计算F（4），按照斐波那契数列的定义，得到数值为F（3）+F（2），因为这两个数值都已计算过，因此可以直接计算output[3]+output[2]=2+1=3，将此值存入用来记录已计算斐波那契数列的数组中，即output[4]=3。

（6）第一次计算F（5），按照斐波那契数列的定义，得到数值为F（4）+F（3），因为这两个数值都已计算过，

所以可以直接计算output[4]+output[3]=3+2=5，将此值存入用来记录已计算斐波那契数列的数组中，即output[5]=5，后续各项以此类推。

5.2　字符串对比功能

字符串对比就是对比两个字符串的内容是否完全相同。比较方法是使用循环从头开始逐一比较字符串中的每个字符，按照它们在ASCII码中编码的数字大小来排列。解决字符串对比问题的传统方法是使用动态规划算法填写一个大小为（m+1）×（n+1）的矩阵，直到出现字符不相同或终止符（'\0'）为止。

字符串对比的算法相当简单，使用循环从头开始逐一比较每一个字符，只要有一个字符不相等即跳离循环，相等则继续比较下一个字符，直到比较到结尾字符为止，如图5-3所示。

图5-3　字符串对比

5.3 AOV网络与拓扑排序

网络图主要用来协助规划大型项目，首先我们将复杂的大型项目细分成很多工作项，而每一个工作项代表网络的一个顶点，由于每一项工作可能有完成的先后顺序，有些可以同时进行，有些则不行，因此可以用网络图来表示其先后完成的顺序。这种以顶点代表工作项的网络称为顶点活动网络（Activity On Vertex Network，简称AOV网络），如图5-4所示。

图5-4 AOV网络

更清楚地说，AOV网络就是在一个有向图G中，每一个顶点（或节点）代表一项工作或行为，边则代表工作之间存在的优先关系，即<V_i，V_j>表示$V_i \rightarrow V_j$的工作，其中顶点V_i的工作必须先完成后，才能进行顶点V_j的工作，称V_i为V_j的“先行者”，而V_j为V_i的“后继者”。

拓扑排序简介

如果在AOV网络中具有部分次序（Partial Order）的关系（有某几个顶点为先行者），拓扑排序的功能就是将这些部分次序的关系转换成线性次序（Linear Order）的关系。例如，i是j的先行者，在线性次序中，i仍排在j的前面，具有这种特性的线性次序就称为拓扑排序（Topological Order）。排序的步骤如下：

（1）寻找图中任何一个没有先行者的顶点。

（2）输出此顶点，并将此顶点的所有边全部删除。

（3）重复以上两个步骤处理所有的顶点。

下面我们试着求出图5-5的拓扑排序，拓扑排序所输出的结果不一定是唯一的，如果同时有两个以上的顶点没有先

行者，结果就不是唯一的。

图5-5　以此图为例来示范拓扑排序

步骤01　首先输出V_1，因为V_1没有先行者，所以删除<V_1，V_2>、<V_1，V_3>、<V_1，V_4>，结果如图5-6所示。

图5-6　删除<V_1，V_2>、<V_1，V_3>、<V_1，V_4>的结果

步骤02　可输出V_2、V_3或V_4，这里我们选择输出V_4，结果如图5-7所示。

图5-7　输出V_4的结果

步骤03　输出V_3，结果如图5-8所示。

图5-8　输出V_3的结果

步骤04　输出V_6，结果如图5-9所示。

图5-9　输出V_6的结果

步骤05　输出V_2、V_5，拓扑排序如图5-10所示。

图5-10　求得拓扑排序的结果

我们再来看一个例子，请大家试着写出图5-11的拓扑排序。

图5-11　求此图的拓扑排序

拓扑排序结果：A，B，E，G，C，F，H，D，I，J，K。

5.4 AOE网络

之前所谈的AOV网络是指在有向图中的顶点表示一项工作，而边表示顶点之间的先后关系。下面介绍一个新名词AOE（Activity On Edge，用边表示的活动网络）。所谓AOE，是指事件（Event）的行动（Action）在边上的有向图。

其中的顶点作为各“进入边事件”（Incident In Edge）的汇集点，当所有“进入边事件”的行动全部完成后，才可以开始“外出边事件”（Incident Out Edge）的行动。在AOE网络上会有一个源头顶点和目的顶点。从源头顶点开始计时执行各边上事件的行动，到目的顶点完成为止，所需的时间为所有事件完成的时间总花费。

关键路径

AOE完成所需的时间是由一条或数条关键路径（Critical Path）所控制的。所谓关键路径，就是AOE有向图从源头顶点到目的顶点之间所需花费时间最长的一条有方向性的路径。当有一条以上的路径花费时间相等，而且都是最长，这些路径都称为此AOE有向图的关键路径。也就是说，想缩短整个AOE完成的花费时间，必须设法缩短关键路径各边行动所需花费的时间。

关键路径可用来判定一个项目至少需要多少时间才能完成，即在AOE有向图中，从源头顶点到目的顶点间最长的路径长度。

在图5-12中，边线和顶点分别代表12个action（a_1，a_2，a_3，a_4…，a_{12}）和10个event（V_1，V_2，V_3…V_{10}），我们先来看一些重要的定义。

图5-12　AOE网络

①最早时间（Earliest Time）

AOE网络中顶点的最早时间为该顶点最早可以开始其“外出边事件”的时间，它必须由最慢完成的“进入边事件”所控制，我们用TE来表示。

②最晚时间（Latest Time）

AOE网络中顶点的最晚时间为该顶点最慢可以开始其“外出边事件”而不会影响整个AOE网络完成的时间。它是由“外出边事件”中最早要求开始者所控制的。我们用TL来表示。

TE和TL的计算原则如下。

·TE：从前往后（从源头到目的地的正方向），若第i项工作前面几项工作有好几个完成时段，取其中最大值。

·TL：从后往前（从目的地到源头的反方向），若第i项工作后面几项工作有好几个完成时段，取其中最小值。

③关键顶点（Critical Vertex）

AOE网络中顶点的TE=TL，我们就称它为关键顶点。从源头顶点到目的顶点的各个关键顶点可以构成一条或数条有向关键路径。只要控制好关键路径所花费的时间，就不会拖延项目的进度。如果集中火力缩短关键路径所需花费的时间，就可以加速整个项目完成的速度。我们以图5-13为例来简单说明如何确定关键路径。

图5-13　以此图为例来说明如何确定关键路径

从图5-13得知，V_1、V_4、V_6、V_8、V_9、V_{10}为关键顶点，可以求得如图5-14所示的关键路径。

图5-14　根据关键顶点求得的关键路径

5.5　青蛙跳台阶算法

青蛙跳台阶算法也是动态规划法的一种，情况是一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶，求该青蛙跳上n级台阶总共有多少种跳法。说明如下：

1级台阶的跳法如图5-15所示。

图5-15　1级台阶的跳法

2级台阶的跳法如图5-16所示。

图5-16　2级台阶的跳法

3级台阶时可以分解为下面两种情况：

（1）可以从最后只跳一级台阶得出，此时和JumpStep[2]情况相同。

（2）也可以从最后跳两级台阶得出，此时和JumpStep[1]情况相同。

也就是说：

JumpStep[3]=JumpStep[3-1]+JumpStep[3-2]

=JumpStep[2]+JumpStep[1]

3级台阶的跳法如图5-17所示。

图5-17　3级台阶的跳法

最后可以得到结论：对于一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶，求该青蛙跳上n级台阶总共有多少种跳法。其解题的通式为JumpStep[n]=JumpStep[n-1]+JumpStep[n-2]。

5.6　课后习题

1.简述动态规划法与分治法的差异。

2.简述拓扑排序的步骤。

3.求下图的拓扑排序。

4.求下图的拓扑排序。

5.什么是关键路径？

6.什么是顶点活动网络？

第6章　安全性算法

网络已成为我们日常生活中不可或缺的一部分，使用计算机上网也越来越频繁，信息可通过网络来互通共享，部分信息可公开，而部分信息则属于机密。网络设计的目的是提供信息、数据和文件的自由交换，不过网络交易确实存在很多风险，因为因特网的成功远远超过了设计者的预期，它除了带给人们许多便利外，也带来了许多安全上的问题，如图6-1所示。

图6-1　网络安全示意图

对于信息安全而言，很难有一个十分严谨而明确的定义或标准。例如，就个人用户而言，只是代表在因特网上浏览时个人数据或信息不被窃取或破坏，不过对于企业或组织而言，可能就代表着进行电子交易时的安全考虑与不法黑客的入侵等。简单来说，信息安全（Information Security）必须具备如图6-2所示的4个特性。

图6-2　信息安全必须具备的4个特性

·保密性（confidentiality）：表示交易相关信息或数据必须保密，当信息或数据传输时，除了被授权的人外，要确保信息或数据在网络上不会遭到拦截、偷窥而泄露信息或数据的内容，损害其保密性。

·完整性（integrity）：表示当信息或数据送达时，必须保证该信息或数据没有被篡改，如果遭篡改，那么这条信息或数据就会无效。例如由甲端传至乙端的信息或数据，乙端在收到时，立刻就会知道这条信息或数据是否完整无误。

·认证性（authentication）：表示当传送方送出信息或数据时，支付系统必须能确认传送者的身份是否为冒名。例如

传送方无法冒名传送信息或数据，持卡人、商家、发卡行、收单行和支付网关都必须申请数字证书进行身份识别。

·不可否认性（non-repudiation）：表示保证用户无法否认他所实施过的信息或数据传送行为的一种机制，必须不易被复制和修改，就是无法否认其传送、接收信息或数据的行为。例如收到金钱不能说没收到，同样，下单购物了不能否认其购买过。

国际标准制定机构英国标准协会（British Standards Institution，BSI）曾经于1995年提出了BS 7799信息安全管理系统，最近的一次修订已于2005年完成，并经国际标准化组织（International Standards Organization，ISO）正式通过，成为ISO 27001信息安全管理系统要求标准，为目前国际公认最完整的信息安全管理标准，可以帮助企业与机构在高度网络化的开放服务环境中鉴别、管理和减少信息所面临的各种风险。

6.1 数据加密

未经加密处理的商业数据或文字资料在网络上进行传输时，任何“有心人士”都能够随手取得，并且一览无遗。因此，在网络上，对于有价值的数据在传送前必须先将原始的数据内容以事先定义好的算法、表达式或编码方法转换成不具任何意义或者不能直接辨读的代码，这个处理过程就是“加密”（Encrypt）。数据在加密前称为“明文”（Plaintext），经过加密后则称为“密文”（Ciphertext）。

经过加密的数据在送抵目的端之后，必须经过“解密”（Decrypt）的过程，才能将数据还原成原来的内容，在这个过程中用于加密和解密的“密码”称为“密钥”（Key）。

数据加密和解密的流程如图6-3所示。

图6-3　数据加密和解密的流程

6.1.1 对称密钥加密系统

“对称密钥加密”（Symmetrical Key Encryption）又称为“单密钥加密”（Single Key Encryption）。这种加密方法的工作方式是发送端与接收端拥有共同的加密和解密的钥匙，这个共同的钥匙被称为密钥（Secret Key）。这种加解密系统的工作方式是：发送端使用密钥将明文加密成密文，使文件看上去像一堆“乱码”，再将密文进行传送，而接收端在收到这个经过加密的密文后，再使用同一把密钥将密文还原成明文。因此，使用对称加密法不但可以为文件加密，而且能达到验证发送者身份的作用。因为如果用户B能用这一组密码解开文件，就能确定这份文件是由用户A加密后传送过来的。对称密钥加密系统进行加密和解密的过程如图6-4所示。

图6-4　对称密钥加密系统进行加密和解密的过程

这种加密系统的工作方式较为直截了当，因此在加密和解密上的处理速度都相当快。常见的对称密钥加密系统算法有DES（Data Encryption Standard，数据加密标准）、Triple DES、IDEA（International Data Encryption Algorithm，国际数据加密算法）等。

6.1.2　非对称密钥加密系统与RSA算法

“非对称密钥加密”是目前较为普遍，也是金融界应用上最安全的加密方法，也被称为“双密钥加密”（Double Key Encryption），又称为公钥（Public Key）加密。这种加密系统主要的工作方式是使用两把不同的密钥——“公钥”（Public key）与“私钥”（Private Key）进行加解密。“公钥”可在网络上自由公开用于加密过程，但必须使用“私钥”才能解密，“私钥”必须由私人妥善保管。例如，用户A要传送一份新的文件给用户B，用户A会使用用户B的公钥来加密，并将密文发送给用户B；当用户B收到密文后，会使用自己的私钥来解密，过程如图6-5所示。

图6-5　非对称密钥加密系统进行加密和解密的过程

RSA（Rivest-Shamir-Adleman）加密算法是一种非对称加密算法，在RSA算法之前，加密算法基本都是对称的。非对称加密算法使用了两把不同的密钥，一把叫公钥，另一把叫私钥，它是在1977年由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）一起提出的，RSA就是由他们三人姓氏的开头字母所组成的。

RSA加解密速度比“对称密钥加解密”速度要慢，方法是随机选出超大的两个质数p和q，使用这两个质数作为加密与解密的一对密钥，密钥的长度一般为40比特到1024比特之间。当然，为了提高加密的强度，现在有的系统使用的RSA密钥的长度高达4096比特，甚至更高。在加密的应用中，这对密钥中的公钥用来加密，私钥用来解密，而且只有私钥可以用来解密。在进行数字签名的应用中，则是用私钥进行签名。要破解以RSA加密的数据，在一定时间内几乎是不可能的，因此这是一种十分安全的加解密算法，特别是在电子商务交易市场被广泛使用。例如，著名的信用卡公司VISA和MasterCard在1996年共同制定并发表

了“安全电子交易协议”（Secure Electronic Transaction，SET），陆续获得IBM、Microsoft、HP及Compaq等软硬件大公司的支持，SET安全机制采用非对称密钥加密系统的编码方式，即采用著名的RSA加密算法。

6.1.3　认证

在数据传输过程中，为了避免用户A发送数据后却否认，或者有人冒用用户A的名义传送数据而用户A本人不知道，面对这类情况，我们需要对数据进行认证。后来又衍生出了第三种加密方式，它是结合对称加密和非对称加密。首先以用户B的公钥加密，接着使用用户A的私钥做第二次加密，当用户B收到密文后，先以A的公钥进行解密，此举可确认信息是由A发送的，再使用B的私钥进行解密，如果能解密成功，就可确保信息传递的保密性，这就是所谓的“认证”，整个过程如图6-6所示。认证的机制看似完美，但是使用非对称密钥进行加解密运算时，计算量非常大，对于大数据量的传输工作而言是个沉重的负担。

图6-6　认证过程

6.1.4　数字签名

在日常生活中，签名或盖章往往是个人或机构对某些承诺或文件承担法律责任的一种署名，而在网络世界中，“数字签名”（Digital Signature）是属于个人或机构的一种“数字身份证”，可以用来对数据发送者的身份进行鉴别。

“数字签名”的工作方式是以公钥和哈希函数互相搭配使用，用户A先将明文的M以哈希函数计算出哈希值H，再用自己的私钥对哈希值H加密，加密后的内容即为“数字签名”。最后将明文与数字签名一起发送给用户B。由于这个数字签名是以A的私钥加密的，且该私钥只有A才有，因此该数字签名可以代表A的身份。由于数字签名机制具有发送者不可否认的特性，因此能够用来确认文件发送者的身份，使其他人无法伪造发送者的身份。数字签名的过程如图6-7所示。

图6-7　数字签名的过程

提示

哈希函数（Hash Function）是一种保护数据完整性的方法，它对要保护的数据进行运算，得到一个“哈希值”，接着将要保护的数据与它的哈希值一同传送。

想要使用数字签名，当然必须先向认证中心（Certification Authority，CA）申请数字证书（Digital Certificate），它可以用来认证公钥为某人所有以及信息发送者的不可否认性，而认证中心所签发的数字签名则包含在数字证书上。通常每一家认证中心的申请过程都不完全相同，只要用户按照网页上的指引步骤操作，即可顺利完成申请。

提示

认证中心为一个具有公信力的第三者，主要负责证书的申请和注册、证书的签发和废止等管理服务。中国国内知名

的证书管理中心如下：

国家电子政务外网数字证书中心：
http://www.stateca.gov.cn/。

北京数字认证股份有限公司：http://www.bjca.org.cn/。

6.2 哈希算法

哈希算法是使用哈希函数计算一个键值所对应的地址，进而建立哈希表，并依靠哈希函数来查找各键值存放在哈希表中的地址，查找的速度与数据多少无关，在没有碰撞和溢出的情况下，一次即可查找成功，这种方法还具有保密性高的优点，因为事先不知道哈希函数就无法查找。

选择哈希函数时，要特别注意不宜过于复杂，设计原则上至少必须符合计算速度快和碰撞频率尽量小两个特点。常见的哈希算法有除留余数法、平方取中法、折叠法和数字分析法。

6.2.1 除留余数法

最简单的哈希函数是将数据除以某一个常数后，取余数作为索引。例如，在一个有13个位置的数组中，只使用到7个地址，值分别是12、65、70、99、33、67、48。我们可以把数组内的值除以13，并以其余数作为数组的下标（索引）。可以用以下式子表示：

```
h (key) =key mod B
```

在这个例子中，我们所使用的B=13。一般而言，建议大家在选择B时，B最好是质数。而上例所建立出来的哈希表如表6-1所示。

表6-1　所建立的哈希表

下面我们将用除留余数法作为哈希函数，将下列数字存储在11个空间：323、458、25、340、28、969、77。请问其哈希表外观如何？

令哈希函数为h（key）=key mod B，其中B=11为一个质数，这个函数的计算结果介于0~10之间（包括0和10这两个数），则h（323）=4、h（458）=7、h（25）=3、h（340）=10、h（28）=6、h（969）=1、h（77）=0。所建立的哈希表如表6-2所示。

表6-2　所建立的哈希表

6.2.2 平方取中法

平方取中法和除留余数法相当类似，就是先计算数据的平方，之后取中间的某段数字作为索引。在下例中，我们用平方取中法计算，将数据存放在100个地址空间中，其操作步骤如下：

将12、65、70、99、33、67、51取平方后如下：

144、4225、4900、9801、1089、4489、2601

再取百位数和十位数作为键值，分别如下：

14、22、90、80、08、48、60

上述7个数字的数列对应于原先的7个数12、65、70、99、33、67、51存放在100个地址空间的索引键值，即

f（14）=12

f（22）=65

f（90）=70

f（80）=99

f（8）=33

f（48）=67

f（60）=51

若实际空间介于0~9之间（10个空间），取百位数和十位数的值介于0～99之间（共有100个空间），则我们必须将平方取中法第一次所求得的键值再压缩1/10，才可以将100个可能产生的值对应到10个空间，即将每一个键值除以10取整数（我们以DIV运算符作为取整数的除法），可以得到下列对应关系：

6.2.3 折叠法

折叠法是将数据转换成一串数字后，先将这串数字拆成几部分，再把它们加起来，就可以计算出这个键值的Bucket Address（桶地址）。例如，有一个数据转换成数字后为2365479125443，若以每4个数字为一个部分，则可拆为2365、4791、2544、3。将这4组数字加起来后即为索引值。

在折叠法中有两种做法，一种是如上例直接将每一部分相加所得的值作为其桶地址，这种做法我们称为“移动折叠法”。哈希法的设计原则之一是降低碰撞，如果希望降低碰撞，就可以将上述每一部分数字中的奇数或偶数反转，再相加，以取得其桶地址，这种改进后的做法我们称为“边界折叠法（Folding At The Boundaries）”。

还是以上面的数字为例，请看下面的说明。

情况一：将偶数反转

情况二：将奇数反转

6.2.4　数字分析法

数字分析法适用于数据不会更改且为数字类型的静态表。在决定哈希函数时，先逐一检查数据的相对位置和分布情况，将重复性高的部分删除。例如，图6-8左图这个电话号码表是相当有规则性的，除了区码全部是080外（注意：此区号仅用于举例，表中的电话号码也不是实际的），中间三个数字的变化不大，假设地址空间的大小m=999，我们必须从这些数字中提取适当的数字，即数字不要太集中，分布范围较为平均（或称随机度高），最后决定提取最后4个数字的末尾三个。故最后得到的哈希表如图6-8右图所示。

图6-8　电话号码表（左图）和所得的哈希表（右图）

看完上面几种哈希函数之后，相信大家可以发现哈希函数并没有一定的规则可寻，可能使用其中的某一种方法，也可能同时使用好几种方法，所以哈希函数常常被用来处理数据的加密和压缩。

6.3 碰撞与溢出处理

在哈希法中，当键对应的值（或标识符）要放入哈希表的某个Bucket（桶）中时，若该Bucket已经满了，则会发生溢出（Overflow）。哈希法的理想情况是所有数据经过哈希函数运算后都得到不同的值，不过现实情况是，即使要存入哈希表的记录中的所有关键字段的值都不相同，经过哈希函数的计算还是可能得到相同的地址，于是就发生了碰撞（Collision）问题。因此，如何在碰撞后处理溢出的问题就显得相当重要。下面介绍常见的处理算法。

6.3.1　线性探测法

线性探测法是当发生碰撞时，如果该索引对应的存储位置已有数据，就以线性的方式往后寻找空的存储位置，一找到空的位置，就把数据放进去。线性探测法通常把哈希的位置视为环状结构，如此一来，如果后面的位置已被填满而前面还有位置时，就可以将数据放到前面，如图6-9所示。

图6-9　线性探测法

6.3.2 平方探测法

线性探测法有一个缺点，就是相当类似的键值经常会聚集在一起，因此可以考虑以平方探测法来加以改善。在平方探测中，当溢出发生时，下一次查找的地址是（f（x）+i^2）mod B与（f（x）-i^2）mod B，即让数据值加或减i的平方，例如数据值为key，哈希函数为f：

第一次寻找：f（key）

第二次寻找：（f（key）+1^2）%B

第三次寻找：（f（key）-1^2）%B

第四次寻找：（f（key）+2^2）%B

第五次寻找：（f（key）-2^2）%B

……

第n次寻找：（f（key）±（（B-1）/2）^{2}）%B

其中，B必须为4j+3型的质数，且1≤i≤（B-1）/2。

6.3.3 再哈希法

再哈希就是一开始先设置一系列哈希函数，如果使用第一种哈希函数出现溢出，就改用第二种，如果第二种也出现溢出，就改用第三种，一直到没有发生溢出为止。例如，h1为key%11，h2为key*key，h3为key*key%11，等等。

下面使用再哈希处理数据碰撞的问题：

```
681，467，633，511，100，164，472，438，445，366，118；
```

其中哈希函数如下（此处的m=13）：

```
f1=h (key) =key MOD m
f2=h (key) = (key+2) MOD m
f3=h (key) = (key+4) MOD m
```

说明如下：

（1）使用第一种哈希函数h（key）=key MOD 13，所得的哈希地址如下：

```
681 —> 5
467 —> 12
633 —> 9
511 —> 4
100 —> 9
164 —> 8
472 —> 4
438 —> 9
445 —> 3
366 —> 2
118 —> 1
```

（2）其中，100、472、438都发生碰撞，再使用第二种哈希函数h（value+2）=（value+2）MOD 13，进行数据的地址安排：

```
100 —> h(100+2)=102 mod 13=11
472 —> h(472+2)=474 mod 13=6
438 —> h(438+2)=440 mod 13=11
```

（3）438仍发生碰撞问题，故接着使用第三种哈希函数h（value+4）=（438+4）MOD 13，重新进行438地址的安排：

```
438 —> h(438+4)=442 mod 13=0
```

经过三次再哈希后，数据的地址安排如表6-3所示。

表6-3　数据的地址安排

6.4 课后习题

1.信息安全必须具备哪4种特性，请简要说明。

2.请简述“加密”与“解密”。

3.请说明“对称密钥加密”与“非对称密钥加密”二者的差异。

4.请简要介绍RSA算法。

5.试简要说明数字签名。

6.用哈希法将下列7个数字存在0、1、…、6七个位置：101、186、16、315、202、572、463。若要存入1000开始的11个位置，应该如何存放？

7.什么是哈希函数？试以除留余数法和折叠法并以7位电话号码作为数据进行说明。

8.试简述哈希查找与一般查找技巧有何不同。

9.什么是完美哈希？在哪种情况下可以使用？

10.采用哪一种哈希函数可以把整数集合：{74，53，66，12，90，31，18，77，85，29}存入数组空间为10的哈希表而不会发生碰撞？

第7章　树结构的算法

树结构（或称为树形结构）是一种日常生活中应用相当广泛的非线性结构（例子见图7-1），树结构的算法在程序中大多使用链表来处理，因为链表的指针用来处理树结构相当方便，修改树节点的指向只需改变指针。当然，也可以使用数组这样的连续内存表示二叉树，使用数组或链表表示树结构各有利弊。本章将为大家介绍与树结构相关的常见算法。

图7-1　社团的组织结构就是树结构的一种应用

7.1　二叉树的遍历

我们知道线性数组或链表只能单向从头至尾遍历或反向遍历。所谓二叉树的遍历（Binary Tree Traversal），最简单的说法是“访问树中所有的节点各一次”，并且在遍历后，将树中的数据转化为线性关系。如图7-2所示是一个简单的二叉树节点，每个节点都可分为左右两个分支。

图7-2　以简单的二叉树作为遍历的例子

所以可以有ABC、ACB、BAC、BCA、CAB、CBA六种遍历方法。如果按照二叉树的特性一律从左向右遍历，就只剩下三种遍历方式，分别是BAC、ABC和BCA。这三种方式的命名与规则如下。

（1）中序遍历（Inorder Traversal，BAC）：左子树→树根→右子树。

（2）前序遍历（Preorder Traversal，ABC）：树根→左子树→右子树。

（3）后序遍历（Postorder Traversal，BCA）：左子树→右子树→树根。

对于这三种遍历方式，大家只需要记得树根的位置，就不会把前序、中序和后序搞混。例如，中序法是树根在中间，前序法是树根在前面，后序法则是树根在后面。遍历方式也一定是先左子树，后右子树。下面针对这三种方式进行更详尽的介绍。

1．中序遍历

中序遍历的遍历顺序是“左中右”，也就是从树的左侧逐步向下方移动，直到无法移动，再访问此节点，并向右移动一个节点。如果无法再向右移动，可以返回上层的父节点，并重复左、中、右的遍历顺序，步骤如下：

（1）遍历左子树。

（2）遍历（或访问）树根。

（3）遍历右子树。

如图7-3所示的二叉树的遍历顺序为FDHGIBEAC。

图7-3　中序遍历二叉树

2.后序遍历

后序遍历的遍历顺序是“左右中”，就是先遍历左子树，再遍历右子树，最后遍历（或访问）根节点，反复执行此遍历顺序，步骤如下：

（1）遍历左子树。

（2）遍历右子树。

（3）遍历树根。

如图7-4所示的二叉树的后序遍历顺序为FHIGDEBCA。

图7-4　后序遍历二叉树

3.前序遍历

前序遍历的遍历顺序是“中左右”，也就是先从根节点遍历，再往左方移动，当无法继续时，向右方移动，接着重复执行此遍历顺序，步骤如下：

（1）遍历（或访问）树根。

（2）遍历左子树。

（3）遍历右子树。

如图7-5所示的二叉树的前序遍历顺序为ABDFGHIEC。

图7-5　前序遍历二叉树

7.2　二叉查找树

如果一个二叉树符合“每一个节点的数据大于左子节点且小于右子节点”，这棵树便称为二分树。二分树便于排序和搜索，二叉排序树和二叉查找树都是二分树的一种。当建立一棵二叉排序树之后，我们要清楚如何在该排序树中查找一个数据。事实上，二叉查找树和二叉排序树可以说是一体两面，没有差别。

二叉查找树具有以下特点：

·可以是空集合，但若不是空集合，则节点上一定要有一个值。

·每一个树根的值需大于左子树的值。

·每一个树根的值需小于右子树的值。

·左右子树也是二叉查找树。

·树的每个节点值都不相同。

基本上，只要懂二叉树的排序就可以理解二叉树的查找。只需在二叉树中比较树根及要查找的值，再按左子树树根右子树的原则遍历二叉树，就可以找到要查找的值。

我们先来讨论如何在所建立的二叉查找树中查找单个节点数据。基本上，二叉查找树在建立的过程中，是按照左子树树根右子树的原则建立的，因此只需从树根出发比较节点的值，如果要查找的值比树根的值大，就前往右子树查找，否则前往左子树查找，直到值相等，表示找到了要查找的值，如果无法再前进（到树叶了），就代表查找不到此值。

二叉树节点插入的情况和查找相似，重点是插入后仍要保持二叉查找树的特性。如果插入的节点已经在二叉树中，就没有插入的必要。而查找失败之处就是准备插入节点的位置。

二叉查找树节点的删除算法则稍为复杂些，可分为以下三种情况。

1.删除的节点为树叶

只要将其相连的父节点指向NULL（空节点）即可。

2.删除的节点只有一棵子树

例如在图7-6中删除节点1，就将节点1的右指针赋值给它的父节点的左指针。

图7-6　删除只有一棵子树的节点，例如节点1

3.删除的节点有两棵子树

例如在图7-7中删除节点4，有两种方式，虽然结果不同，但都符合二叉查找树的特性。

（1）找出中序立即先行者（Inorder Immediate Predecessor），即将要删除节点的左子树中最大者向上提，在此即为图7-7中的节点2。简单来说，就是从该节点的左子树往右寻找，直到右指针为NULL，这个节点就是中序立即先行者。

图7-7　删除有两棵子树的节点，例如节点4

（2）找出中序立即后继者（Inorder Immediate Successor），即把要删除节点的右子树中的最小者向上提，在此即为图7-8中的节点5。简单来说，就是从该节点的右子树往左寻找，直到左指针为NULL，这个节点就是中序立即后继者。

请将32、24、57、28、10、43、72、62按中序方式存入可放10个节点的数组内，试绘图并说明节点在数组中的相关位置。如果插入的数据为30，试绘图并写出其相关操作与节点位置的变化。接下来，如果再删除数据32，试绘图并写出其相关操作与节点位置的变化。

建立10个节点的二叉树如图7-8所示。

图7-8　建立10个节点的二叉树

建立好10个节点的二叉树后，该树中各个节点的数据字段和指针字段的值如表7-1所示。

表7-1　该树中各个节点的数据字段和指针字段的值

插入数据30后的二叉树如图7-9所示。

图7-9　把数据30插入二叉树中

将数据30插入二叉树后，该树中各个节点的数据字段和指针字段的值如表7-2所示。

表7-2　插入数据30后，该树中各个节点的数据字段和指针字段的值

删除数据32后的二叉树如图7-10所示。

图7-10　删除数据32后的二叉树

删除二叉树中的数据32之后，该树中各个节点数据字段和指针字段的值如表7-3所示。

表7-3　删除数据32之后，该树中各个节点数据字段和指针字段的值

7.3 优化二叉查找树

在前文中我们介绍过，如果一个二叉树符合“每一个节点的数据大于左子节点且小于右子节点”，这棵树便具有二叉查找树的特性。所谓优化二叉查找树，简单地说，就是在所有可能的二叉查找树中有最小查找成本的二叉树。

7.3.1 扩充二叉树

什么叫作最小查找成本呢？我们先从扩充二叉树（Extension Binary Tree）谈起。任何一个二叉树中，若具有n个节点，则有n-1个非空链接和n+1个空链接。如果在每一个空链接加上一个特定节点，就称为外节点，其余的节点称为内节点，因而定义这种树为“扩充二叉树”。另外定义：外径长等于所有外节点到树根距离的总和，内径长等于所有内节点到树根距离的总和。我们将以图7-11的图（a）和图（b）来说明它们的扩充二叉树的绘制过程。

图7-11　以图（a）和图（b）为例说明扩充二叉树的绘制

图7-11的图（a）的扩充二叉树如图7-12所示。

图7-12　图（a）的扩充二叉树（代表外部节点）

外径长：2+2+4+4+3+2=17　内径长：1+1+2+3=7

图7-12的图（b）的扩充二叉树如图7-13所示。

图7-13　图（b）的扩充二叉树

外径长：（2+2+3+3+3+3）=16　内径长：（1+1+2+2）=6

以图7-11的图（a）和图（b）为例，若每个外部节点有加权值（例如查找概率等），则外径长必须考虑相关加权值，或称为加权外径长。下面将讨论图（a）和图（b）的加权外径长。

对图（a）来说，2×3+4×3+5×2+15×1=43。具有加权值的图（a）的扩充二叉树如图7-14所示。

对图（b）来说，2×2+4×2+5×2+15×2=52。具有加权值的图（b）的扩充二叉树如图7-15所示。

图7-14　具有加权值的图（a）的扩充二叉树

图7-15　具有加权值的图（b）的扩充二叉树

7.3.2 哈夫曼树

哈夫曼树经常应用于数据的压缩，是可以根据数据出现的频率来构建的二叉树。例如，数据的存储和传输是数据处理的两个重要领域，两者都和数据量的大小息息相关，而哈夫曼树正好可以用于数据的压缩。

简单来说，如果有n个权值（q_1，q_2，…，q_n），且构成一个有n个节点的二叉树，每个节点的外部节点的权值为q_i，则加权外径长最小的就称为“优化二叉树”或“哈夫曼树”（Huffman Tree）。对于7.3.1小节中图7-11的图（a）和图（b）的二叉树而言，图（a）就是二者的优化二叉树。下来我们来说明对于一个含权值的链表，如何求其优化二叉树，步骤如下：

步骤01　产生两个节点，对数据中出现过的每一个元素各自产生一个树叶节点，并赋予树叶节点该元素的出现频率。

步骤02　令N为T_1和T_2的父节点，T_1和T_2是T中出现频率最低的两个节点，令N节点的出现频率等于T_1和T_2出现频率的总和。

步骤03　将步骤02中的T_1和T_3节点插入N，再重复步骤1。

我们将采用以上步骤来实现哈夫曼树，假设现在有5个字母BDACE，其出现频率分别为0.09、0.12、0.19、0.21和0.39，哈夫曼树的构建过程如下：

步骤01　取出最小的0.09和0.12，合并成一棵新的二叉树，其根节点的频率为0.21，如图7-16所示。

图7-16　第一次合并

步骤02　再取出0.19，与0.21为根的二叉树合并后，得到0.40为根的新二叉树，如图7-17所示。

图7-17　第二次合并

步骤03　再取出0.21和0.39的节点，产生频率为0.6的新节点，得到右边的新二叉树，如图7-18所示。

图7-18　第三次合并

步骤04　最后取出0.40和0.60两个二叉树的根节点，将它们合并成频率为1.0的节点，至此二叉树就完成了。

7.4　平衡树（AVL树）

由于二叉查找树的缺点是无法永远保持在最佳状态。在加入的数据部分已排序的情况下，极有可能产生斜二叉树，因而使树的高度增加，导致查找效率降低。因此，一般的二叉查找树不适用于数据经常变动（加入或删除）的情况。为了能够尽量减少查找所需要的时间，在查找的时候能够很快找到所要的键值，我们必须让树的高度越小越好。

平衡树的定义

平衡树（Balanced Binary Tree）又称为AVL树（是由Adelson-Velskii和Landis两人发明的），它本身也是一棵二叉查找树。在AVL树中，每次在插入数据和删除数据后，必要的时候会对二叉树做一些高度的调整，而这些调整就是让二叉查找树的高度随时维持平衡。T是一棵非空的二叉树，T_l和T_r分别是它的左右子树，若符合下列两个条件，则称T是一棵高度平衡树。

（1）T_1和T_r都是高度平衡树。

（2）$|h_1-h_r|\leq 1$，h_1和h_r分别为T_1和T_r的高度，也就是所有内部节点的左右子树的高度相差必定小于或等于1。

AVL树与非AVL树的例子如图7-19所示。

图7-19　AVL树（a）与非AVL树（b）

至于如何调整一棵二叉搜索树成为一棵平衡树，最重要的是找出“不平衡点”，再按照以下4种不同的旋转方式重新调整其左右子树的长度。首先，令新插入的节点为N，且其最近的一个具有±2的平衡因子节点为A，下一层为B，再下一层为C，分别介绍如下。

（1）左左型（LL型，如图7-20所示）

图7-20　LL型

（2）左右型（LR型，如图7-21所示）

图7-21　LR型

（3）右右型（RR型，如图7-22所示）

图7-22　RR型

（4）右左型（RL型，如图7-23所示）

图7-23　RL型

现在我们来实现一个范例。例如图7-24所示的二叉树原来是平衡的，加入节点12后不平衡了，请重新调整成平衡树，但不可破坏原有的次序结构。

图7-24　加入节点12之前的平衡树

调整结果如图7-25所示。

图7-25　加入节点12之后再调整为平衡树

7.5　博弈树——八枚金币问题

符合博弈法则的决策树（Decision Tree）被称为博弈树（Game Tree），这是因为游戏中的人工智能经常以博弈树的数据结构来实现。对数据结构而言，博弈树本身是人工智能中的一个重要概念。在信息管理系统（Management Information System，MIS）中，决策树是决策支持系统（Decision Support System，DSS）执行的基础。

简单来说，博弈树使用树结构的方法来讨论一个问题的各种可能性。下面用典型的“八枚金币”问题来阐述博弈树的概念。假设有8枚金币a、b、c、d、e、f、g、h，其中有一枚金币是伪造的，伪造金币的特征是重量稍轻或偏重。如何使用博弈树的方法来找出这枚伪造的金币？以L表示伪造的金币轻于真品，以H表示伪造的金币重于真品。第一次比较时，从8枚金币中任意挑选6枚：a、b、c、d、e、f，分成2组来比较重量，则会出现下列三种情况：

```
(a+b+c) > (d+e+f)
(a+b+c) = (d+e+f)
(a+b+c) < (d+e+f)
```

我们可以按照以上步骤画出如图7-26所示的博弈树。

如果我们要设计的游戏属于“棋类”或“纸牌类”，那么所采用的技巧在于进行游戏时电脑“决策”的能力，简单地说，就是该下哪一步棋或者该出哪一张牌。因为游戏时可能发生的情况很多，例如象棋游戏的人工智能必须在所有可能的情况中选择一步对自己最有利的棋，想想看，如果开发此类游戏，我们要怎么做呢？这时博弈树就可以派上用场了。

图7-26　“八枚金币”问题的博弈树

通常此类游戏人工智能的实现技巧是先找出所有可走的棋（或可出的牌），然后逐一判断走这步棋（或出这张牌）的优劣程度如何，或者替这步棋打个分数，然后选择走得分最高的那步棋。

一个常被用来讨论博弈型人工智能的简单例子是“井”字棋游戏，因为它可能发生的情况不多，我们大概只要花十分钟便能分析完所有可能的情况，并且找出最佳的玩法，例如图7-27表示在某种情况下X方的博弈树。

图7-27　“井”字棋游戏的部分博弈树

图7-27是“井”字棋游戏的部分博弈树，下一步是X方下棋，很明显X方绝对不能选择第二层的第二种下法，因为X方必败无疑。我们可以看出这个博弈决策形成树结构，所以称为“博弈树”，而树结构正是数据结构所讨论的范围，这说明数据结构也是人工智能的基础，博弈决策形成人工智能的基础是查找，在所有可能的情况下，找出可能获胜的下法。

7.6　堆积排序法

堆积排序法是选择排序法的改进版，它可以减少在选择排序法中的比较次数，进而减少排序时间。堆积排序法用到了二叉树的技巧，它是使用堆积树来完成排序的。堆积树是一种特殊的二叉树，可分为最大堆积树和最小堆积树两种。最大堆积树满足以下3个条件：

（1）它是一棵完全二叉树。

（2）所有节点的值都大于或等于其左右子节点的值。

（3）树根是堆积树中最大的。

而最小堆积树则具备以下3个条件：

（1）它是一棵完全二叉树。

（2）所有节点的值都小于或等于其左右子节点的值。

（3）树根是堆积树中最小的。

下面我们将使用堆积排序法对34、19、40、14、57、17、4、43进行排序，排序的过程示范如下：

步骤01　按图7-28上图的数字顺序建立完全二叉树（见图7-28下图）。

图7-28　建立完全二叉树

步骤02　建立堆积树，结果如图7-29所示。

图7-29　建立堆积树

步骤03　将57从树根删除，再重新建立堆积树，结果如图7-30所示。

图7-30　删除57，再重新建立堆积树

步骤04　将43从树根删除，再重新建立堆积树，结果如图7-31所示。

图7-31　删除43，再重新建立堆积树

步骤05　将40从树根删除，再重新建立堆积树，结果如图7-32所示。

图7-32　删除40，再重新建立堆积树

步骤06　将34从树根删除，再重新建立堆积树，结果如图7-33所示。

图7-33　删除34，再重新建立堆积树

步骤07　将19从树根删除，再重新建立堆积树，结果如图7-34所示。

图7-34　删除19，再重新建立堆积树

步骤08　将17从树根删除，再重新建立堆积树，结果如图7-35所示。

图7-35　删除17，再重新建立堆积树

步骤09　将14从树根删除，再重新建立堆积树，结果如图7-36所示。

图7-36　删除14，再重新建立堆积树

最后将4从树根删除，得到的排序结果如下：

57、43、40、34、19、17、14、4

7.7 斐波那契查找法

斐波那契查找法（Fibonacci Search）又称为斐氏查找法，此查找法和二分法一样都是以分割范围来进行查找的，不同的是斐波那契查找法不以对半分割，而是以斐波那契级数的方式来分割的。

斐波那契级数F（n）的定义如下：

$$F_0 = 0, F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2}, \quad i \geq 2$$

斐波那契级数：0、1、1、2、3、5、8、13、21、34、55、89、…。也就是除了第0个和第1个元素外，级数中的每个值都是前两个值的和。

斐波那契查找法的好处是只用到加减运算而不需要用到乘除运算，这从计算机运算的过程来看效率会高于前两种查找法。在尚未介绍斐波那契查找法之前，我们先来认识斐波那契查找树。所谓斐波那契查找树，是以斐波那契级数的特性来建立的二叉树，其建立的原则如下：

（1）斐波那契树的左右子树均为斐波那契树。

（2）当数据个数n确定时，若想确定斐波那契树的层数k值是多少，我们必须找到一个最小的k值，使得斐波那契层数的Fib（k+1）≥n+1。

（3）斐波那契树的树根一定是一个斐波那契数，且子节点与父节点差值的绝对值为斐波那契数。

（4）当k≥2时，斐波那契树的树根为Fib（k），左子树为（k-1）层斐波那契树（其树根为Fib（k-1）），右子树为（k-2）层斐波那契树（其树根为Fib（k）+Fib（k-2））。

（5）若n+1值不为斐波那契树的值，则可以找出存在一个m，使得Fib（k+1）-m=n+1，m=Fib（k+1）-（n+1），再

按斐波那契树的建立原则完成斐波那契树的建立，最后斐波那契树的各节点再减去差值m即可，并把小于1的节点去掉。

斐波那契树建立过程的示意图如图7-37所示。

图7-37　斐波那契树建立过程的示意图

也就是说，当数据个数为n，且我们找到一个最小的斐波那契数Fib（k+1）使得Fib（k+1）>n+1时，Fib（k）就是这棵斐波那契树的树根，而Fib（k-2）则是与左右子树开始的差值，左子树用减的，右子树用加的。例如，我们来实际求取n=33的斐波那契树。

由于n=33，且n+1=34为一棵斐波那契树，我们知道斐波那契数列的三个特性：

```
Fib(0)=0
Fib(1)=1
Fib(k)=Fib(k-1)+Fib(k-2)
```

得知Fib（0）=0、Fib（1）=1、Fib（2）=1、Fib（3）=2、Fib（4）=3、Fib（5）=5、Fib（6）=8、Fib（7）=13、Fib（8）=21、Fib（9）=34。

从上式可得知，Fib（k+1）=34k=8，建立二叉树的树根为Fib（8）=21。

左子树的树根为Fib（8-1）=Fib（7）=13。

右子树的树根为Fib（8）+Fib（8-2）=21+8=29。

按此原则，我们可以建立如图7-38所示的斐波那契树。

图7-38　斐波那契树

斐波那契查找法是以斐波那契树来查找数据，如果数据的个数为n，n比某一个斐波那契数小，且满足如下表达式：

Fib (k+1) ≥n+1

此时Fib（k）就是这棵斐波那契树的树根，而Fib（k-2）则是与左右子树开始的差值，若我们要查找的键值为key，首先比较数组下标Fib（k）和键值key，此时有下列三种情况：

（1）当key值比较小时，表示所查找的键值key落在1到Fib（k）-1之间，故继续查找1到Fib（k）-1之间的数据。

（2）如果键值与数组下标Fib（k）的值相等，就表示成功查找到所需要的数据。

（3）当key值比较大时，表示所找的键值key落在Fib（k）+1到Fib（k+1）-1之间，故继续查找Fib（k）+1到Fib（k+1）-1之间的数据。

7.8 课后习题

1.请问以下二叉树的中序法、后序法以及前序法表达式分别是什么？

2.关于二叉查找树的叙述，哪一个是错的？

A．二叉查找树是一棵完整二叉树

B．可以是歪斜树

C．一节点最多只有两个子节点

D．一节点的左子节点的键值不会大于右节点的键值

3.形成8层的平衡树最少需要几个节点？

4.请将下图的树转换化二叉树。

5.在下图的平衡二叉树中，加入节点11后，重新调整后的平衡树是什么？

6.有一棵二叉查找树：

（1）键值key平均分配在[1，100]之间，求在该查找树查找平均要比较几次。

（2）假设k=1时，其概率为0.5，k=4时，其概率为0.3，k=9时，其概率为0.103，其余97个数的概率为0.001，求在该查找树查找平均要比较几次。

（3）假设各key的概率如（2），是否能将此查找树重新安排？

（4）以得到的最小平均比较次数绘出重新调整后的查找树。

7.请建立一棵最小堆积树，必须写出建立此堆积树的每一个步骤。

8.试以数列26、73、15、42、39、7、92、84说明堆积排序的过程。

9.什么是博弈树？试举例说明。

10.请简要介绍哈夫曼树。

第8章　改变程序功力的经典算法

我们可以这样说，算法是用计算机来实现数学思想的一种学问，学习算法就是了解它们如何演算，以及它们如何在各层面影响我们的日常生活。对于程序设计人员来说，“排序”（Sorting）在计算机相关领域非常重要且使用得非常普遍，熟悉各种经典算法与相关的排序原理，往往是程序设计能否顺利运行甚至是成败的关键。接下来我们介绍成为一名专业程序设计人员必学的算法，涉及的内容包括这些算法的相关知识、特性及其工作原理。

提示

在排序的过程中，数据的移动方式可分为“直接移动”和“逻辑移动”两种。“直接移动”是直接交换存储数据的位置，而“逻辑移动”并不会移动存储数据的位置，仅改变指向这些数据的辅助指针，如图8-1和图8-2所示。

图8-1　直接移动排序

图8-2　逻辑移动排序

8.1 迭代法

迭代法（Iterative Method）是指无法使用公式一次求解，而需要使用重复结构重复执行一段程序代码来得到答案。“重复结构”即所谓的循环（Loop），对于程序中需要重复执行的程序语句，都可以交由循环来完成。循环主要由下面的两个基本要素组成：

（1）循环的执行主体（简称循环体），由程序语句或复合语句组成。

（2）循环的条件判断表达式，决定循环体何时停止执行。

例如，想要让计算机算出1+2+3+4+…+100的值，在程序中并不需要我们大费周章地从1累加到100，这时只需要使用重复结构即可。如果是相同的算法，能够直接使用循环，而不是使用递归算法的堆栈运算，那么都能以较高的效率完成工作，我们之前讲述的求解n！的值也不例外。

8.1.1　帕斯卡三角算法

帕斯卡（Pascal）三角算法基本上就是计算出三角形每一个位置的数值。在帕斯卡三角上的每一个数字各对应一个${}_rC_n$，其中r代表行（row），而n代表列（column），r和n都从数字0开始。帕斯卡三角如下：

帕斯卡三角对应的数据如图8-3所示。

图8-3　帕斯卡三角对应的数据

至于如何计算帕斯卡三角中的rCn，我们可以使用以下式子：

```
rC0=1
rCn=rCn-1* (r-n+1) /n
```

上面的两个式子所代表的意义是每一行的第0列的值一定为1。例如，${}_0C_0=1$、${}_1C_0=1$、${}_2C_0=1$、${}_3C_0=1$，以此类推。

一旦每一行第0列元素的值为数字1确立后，该行的每一列元素值都可以从同一行前一列的值根据下面的公式计算得到：

```
rCn=rCn-1* (r-n+1) /n
```

下面举例来说明。

（1）第0行帕斯卡三角的求值过程

当r=0、n=0时，即第0行（row=0）、第0列（column=0）所对应的数字为0时，帕斯卡三角外观如下：

（2）第1行帕斯卡三角的求值过程

当r=1、n=0时，代表第1行第0列所对应的数字$_1C_0=1$。

当r=1、n=1时，求第1行（row=1）、第1列（column=1）所对应的数字$_1C_1$。请代入公式$_rC_n=_rC_{n-1}$*（r-n+1）/n（其中r=1、n=1），可以推导出下面的式子：

$_1C_1=_1C_0$* (1-1+1) /1=1*1=1

得到的结果是$_1C_1=1$。此时的帕斯卡三角外观如下：

（3）第2行帕斯卡三角的求值过程

按照上面计算每一行中各个元素值的求值过程，可以推导出$_2C_0=1$、$_2C_1=2$、$_2C_2=1$。此时的帕斯卡三角外观如下：

（4）第3行帕斯卡三角的求值过程

按照上面计算每一行中各个元素值的求值过程，可以推导出$_3C_0=1$、$_3C_1=3$、$_3C_2=3$、$_3C_3=1$。此时的帕斯卡三角外观如下：

同理，可以陆续推导出第4行、第5行、第6行……所有帕斯卡三角中各行的元素。

8.1.2　插入排序法

插入排序法（Insert Sort）是将数组中的元素逐一与已排序好的数据进行比较，前两个元素先排好，再将第三个元素插入适当的位置，所以这三个元素仍然是已排好序的，接着将第四个元素加入，重复此步骤，直到排序完成为止。可以看作在一串有序的记录R_1、R_2、…、R_i中，要插入新记录R，就要使得i+1个记录排序妥当。

下面我们仍然用55、23、87、62、16数列的从小到大排序过程来说明插入排序法的演算流程。在图8-4中，在步骤二，以23为基准与其他元素比较后，放到适当位置（55的前面），步骤三则用87与其他两个元素比较，步骤四用62与前面三个数比较后，插入87的前面，如此反复，直到将最后一个元素比较完后即完成排序。

图8-4　插入排序过程示例

8.1.3　希尔排序法

在原始记录的键值大部分已排好序的情况下，插入排序法会非常高效，因为它不需要执行太多的数据搬移操作。“希尔排序法”是D.L.Shell在1959年7月提出的一种排序法，可以减少插入排序法中数据搬移的次数，以加快排序的过程。排序的原则是将数据区分成特定间隔的几个小区块，以插入排序法排完区块内的数据后，再渐渐减少间隔的距离。

下面我们仍然用63、92、27、36、45、71、58、7这个数列的从小到大排序过程来说明希尔排序法的演算流程，原始数据如图8-5所示。

图8-5　原始数据

步骤01　将所有数据分成Y：（8div 2），即Y=4，称为划分的区块数。注意，划分的区块数不一定要是2，质数最好。但为了算法方便，我们习惯选择2。因而一开始的间隔设置为8/2，如图8-6所示。

图8-6　间隔设置为4

步骤02　如此一来，可得到4个区块，它们分别是（63，45）（92，71）（27，58）（36，7），再分别用插入排序法排序成（45，63）（71，92）（27，58）（7，36）。在整个数列中，数据的排列如图8-7所示。

图8-7　排序后的结果

步骤03　接着缩小间隔为（8/2）/2，如图8-8所示。

图8-8　间隔设置为2

步骤04　（45，27，63，58）（71，7，92，36）再分别用插入排序法排序成（27，45，58，63）（7，36，71，92），得到如图8-9所示的结果。

图8-9　排序后的结果

步骤05　再以（（8/2）/2）/2的间距进行插入排序，也就是每一个元素进行排序，于是得到最后的结果，如图8-10所示。

图8-10　间隔设置为1排序后的结果

8.1.4 基数排序法

基数排序法和我们之前所讨论的排序法不太一样，它并不需要进行元素间的比较操作，而是属于一种分配模式的排序方式。基数排序法按比较的方向可分为最高位优先（Most Significant Digit First，MSD）和最低位优先（Least Significant Digit First，LSD）两种。MSD是从最左边的位数开始比较，而LSD则是从最右边的位数开始比较。

在下面的范例中，我们以LSD将三位数的整数数据加以排序，是按照个位数、十位数、百位数进行排序的。请直接看以下LSD例子的说明，便可清楚地知道它的工作原理。

原始数据如下：

步骤01　把每个整数按其个位数字放到列表中：

合并后成为：

步骤02　按其十位数字，按序放到列表中：

合并后成为：

步骤03　按其百位数字，按序放到列表中：

最后合并即完成排序：

8.2　枚举法

枚举法又称为穷举法，是一种常见的数学方法，也是日常中用到最多的一种算法，它的核心思想是：列举所有的可能。根据问题的要求逐一列举问题的解答，或者为了便于解决问题，把问题分为不重复、不遗漏的有限种情况，逐一列举各种情况，并加以解决，最终达到解决整个问题的目的。枚举法这种分析问题、解决问题的方法得到的结果总是正确的，枚举算法的缺点是速度太慢。

例如，我们想将A与B两个字符串连接起来，就是将B字符串接到A字符串的后面，具体做法是将B字符串的每一个字符从第一个字符开始逐步连接到A字符串的最后一个字符，如图8-11所示。

图8-11　将B字符串连接到A字符串后面

再来看一个例子，当1000依次减去1、2、3、…直到哪一个数时，相减的结果开始为负数？这是很纯粹的枚举法的应用，只要按序减去1、2、3、4、5、6、7、8、…即可：

```
1000-1-2-3-4-5-6-…-? < 0
```

以枚举法求解这个问题，算法过程如下：

简单来说，枚举法的核心思路是将要分析的项目在不遗漏的情况下逐一列举出来，再从所列举的项目中找到自己所需要的目标对象。

8.2.1　3个小球放入盒子

接下来所举的例子很有趣，我们把3个相同的小球放入A、B、C三个小盒中，请问共有多少种不同的放法？分析枚举法的关键是分类，本题分类的方法有很多，例如可以分成这样三类：3个球放在一个盒子里；2个球放在一个盒子里，剩余的一个球放在一个盒子里；3个球分3个盒子放。

第一类：3个球放在一个盒子里，会有三种可能的情况，如图8-12~图8-14所示。

图8-12　第一类放置小球的方法——情况1

图8-13　第一类放置小球的方法——情况2

图8-14　第一类放置小球的方法——情况2

第二类：2个球放在一个盒子里，剩余的一个球放在一个盒子里，会有6种可能的情况，如图8-15~图8-20所示。

图8-15　第二类放置小球的方法——情况1

图8-16　第二类放置小球的方法——情况2

图8-17　第二类放置小球的方法——情况3

图8-18　第二类放置小球的方法——情况4

图8-19　第二类放置小球的方法——情况5

图8-20　第二类放置小球的方法——情况6

第三类：3个球分3个盒子放，只有一种可能的情况，如图8-21所示。

图8-21　第三类放置小球的方法，只有一种可能的情况

根据枚举法的思路找出上述10种放置小球的方式。

8.2.2　质数求解算法

所谓质数，就是大于1并且除了自身之外无法被其他整数整除的数，例如2、3、5、7、11、13、17、19、23等，如图8-22所示。如何快速找出质数呢？在此特别推荐埃拉托色尼筛选法（Eratosthenes），即求质数的方法。首先假设要检查的数是N，接着参照下列步骤就可以判断数字N是否为质数。在求质数的过程中，可以适时运用一些技巧以减少循环检查的次数，以便加速对质数的判断工作。

图8-22　质数

除了判断一个数是否为质数外，另一个衍生的问题是如何求出小于N的所有质数？在此一并说明。

求质数很简单，这个问题可以使用循环将数字N除以所有小于它的正整数，如果可以整除，就不是质数。进一步检查发现，其实只要检查到N的开平方根取整的正整数就可以了，这是因为N=A*B，如果A大于N的平方根，那么因为A和B乘积对称的关系，相当于B已经被检查过了。由于开平方根常会碰到浮点数精确度的问题，因此为了让循环检查的速度加快，可以使用整数i和$i \times i \leq N$的条件判断表达式判定检查到哪一个整数时停止。

举例来说，要判断89是否为质数，只需判断到数字i=8即可。过程如下：

89mod 2=1（不能整除）

89mod 3=2（不能整除）

89mod 4=1（不能整除）

89mod 5=4（不能整除）

89mod 6=5（不能整除）

89mod 7=5（不能整除）

89mod 8=1（不能整除）

结论：计算到i×i≤N（N=89），推算出i=8，循环到此还找不到一个整数可以整除89，就可以判定89为质数。

再来看另一个例子，要判断91是否为质数，只需判断到数字i=8即可。过程如下：

91mod 2=1（不能整除）

91mod 3=1（不能整除）

91mod 4=3（不能整除）

91mod 5=1（不能整除）

91mod 6=1（不能整除）

91mod 7=0（可以整除）

结论：还没计算到i×i≤N（N=91，i=8），循环到此已找到数字7可以整除91，可以判定91不是质数。

8.2.3　顺序查找法

顺序查找法又称线性查找法，是一种最简单的查找法。顺序查找法是将数据一项一项地按顺序逐个查找，所以无论数据顺序如何，都得从头到尾遍历一次。顺序查找法的优点是文件在查找前不需要进行任何的处理与排序，缺点是查找速度较慢。如果数据没有重复，找到数据就可以中止查找的话，最差的情况是未找到数据，需进行n次比较，最好的情况则是一次就找到数据，只需1次比较。

现在以一个例子来说明。假设已有数列74、53、61、28、99、46、88，若要查找28，则需要比较4次；若要查找74，则仅需比较1次；若要查找88，则需比较7次，表示当查找的数列长度n很大时，用顺序查找是不太适合的，它是一种适用于小数据文件的查找方法。在日常生活中，我们经常会使用这种查找法，例如我们想在衣柜中找衣服，通常会从柜子最上方的抽屉逐层寻找，如图8-23所示。

图8-23　顺序查找法在现实生活中的应用

8.2.4 冒泡排序法

冒泡排序法又称为交换排序法，是从观察水中气泡变化构思而成的，原理是从第一个元素开始比较相邻元素的大小，若大小顺序有误，则对调后再进行下一个元素的比较，就仿佛气泡从水底逐渐冒升到水面上一样。如此扫描过一次之后就可以确保最后一个元素位于正确的顺序。接着逐步进行第二次扫描，直到完成所有元素的排序为止。

下面使用55、23、87、62、16这个数列来演示排序过程，这样大家可以清楚地知道冒泡排序法的具体流程。图8-24所示为原始值，按从小到大排序的过程如下：

图8-24　排序前的原始值

步骤01　第一次扫描会先用第一个元素55和第二个元素23进行比较，如果第二个元素小于第一个元素，就进行互换。接着用55和87进行比较，就这样一直比较并互换，到第4次比较完后，即可确定最大值在数组的最后面，如图8-25所示。

图8-25　冒泡排序的第一次扫描

步骤02　第二次扫描也是从头比较，但因为最后一个元素在第一次扫描时就已确定是数组中的最大值，故只需比较3次即可把数组中剩余元素的最大值排到剩余数组的最后面，如图8-26所示。

图8-26　冒泡排序的第二次扫描

步骤03　第三次扫描完，完成三个值的排序，如图8-27所示。

图8-27　冒泡排序的第三次扫描

步骤04　第四次扫描完，即可完成所有排序，如图8-28所示。

图8-28　冒泡排序的第四次扫描

由此可知，5个元素的冒泡排序法必须执行5-1=4次扫描，第一次扫描需比较5-1=4次，一共要比较4+3+2+1=10次。

8.2.5　选择排序法

选择排序法（Selection Sort）是枚举法的另一类应用，这种排序算法的思路是反复从未排序的数列中取出最小的元素，加入另一个数列中，把未排序数列中的所有元素取完，另一个数列的结果即为已排序的数列。选择排序法可使用两种方式排序：一种是在所有的数据中，当从大到小排序时，将最大值放入第一个位置；另一种是从小到大排序，将最大值放入最后一个位置。例如，一开始在所有的数据中挑选一个最小项放在第一个位置（假设是从小到大排序），再从第二项开始挑选一个最小项放在第2个位置，以此重复，直到完成排序为止。

下面我们仍然用55、23、87、62、16这个数列从小到大排序的过程来说明选择排序法的演算流程。原始数据如图8-29所示。

图8-29　待排序的数列

步骤01　首先找到此数列中的最小值，与数列中的第一个元素交换，如图8-30所示。

图8-30　选择排序的第一次扫描

步骤02　从第二个值开始找，找到此数列中（不包含第一个）的最小值，再和第二个值交换，如图8-31所示。

图8-31　选择排序的第二次扫描

步骤03　从第三个值开始找，找到此数列中（不包含第一、二个）的最小值，再和第三个值交换，如图8-32所示。

图8-32　选择排序的第三次扫描

步骤04　从第四个值开始找，找到此数列中（不包含第一、二、三个）的最小值，再和第四个值交换，此排序完成，如图8-33所示。

图8-33　选择排序的第四次扫描

8.3 回溯法

回溯法（Backtracking）也是枚举法的一种，对于某些问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，同时避免枚举不正确的数值。一旦发现不正确的数值，就不再递归到下一层，而是回溯到上一层，以节省时间，是一种走不通就退回再走的方式。它的特点主要是在搜索过程中寻找问题的解，当发现不满足求解条件时，就回溯（返回），尝试别的路径，避免无效搜索。

8.3.1　老鼠走迷宫

例如，老鼠走迷宫就是一种回溯法的应用。老鼠走迷宫问题的描述是：假设把一只老鼠放在一个没有盖子的大迷宫盒的入口处，盒中有许多墙，使得大部分的路径都被挡住而无法前进。老鼠可以采用尝试错误的方法找到出口。不过，这只老鼠必须在走错路时就退回来并把走过的路记下来，避免下次走重复的路，就这样直到找到出口为止。简单来说，老鼠行进时，必须遵守以下三个原则：

（1）一次只能走一格。

（2）遇到墙无法往前走时，就退回一步找找看是否有其他的路可以走。

（3）走过的路不会再走第二次。

在编写走迷宫程序之前，我们先来了解如何在计算机中描述一个仿真迷宫的地图。这时可以使用二维数组MAZE[row][col]并符合以下规则：

MAZE[i][j]=1表示[i][j]处有墙，无法通过。

MAZE[i][j]=0表示[i][j]处无墙，可通行。

MAZE[1][1]是入口，MAZE[m][n]是出口。

如图8-34就是一个使用10×12二维数组的仿真迷宫地图。

图8-34　10×12二维数组的仿真迷宫地图

假设老鼠从左上角的MAZE[1][1]进入，从右下角的MAZE[8][10]出来，老鼠当前位置用MAZE[x][y]表示，那么老鼠可能移动的方向如图8-35所示。

图8-35　老鼠可能移动的方向

如图8-35所示，老鼠可以选择的方向共有4个，分别为东、西、南、北。但并非每个位置都有4个方向可以选择，必须视情况来决定，例如T字形的路口就只有东、西、南三个方向可以选择。

我们可以使用链表来记录走过的位置，并且将走过的位置所对应的数组元素内容标记为2，然后将这个位置放入堆栈，再进行下一个方向或路的选择。如果走到死胡同并且还没有抵达终点，就退回上一个位置，直到退回到上一个岔路后再选择其他的路。由于每次新加入的位置必定会在堆栈的顶端，因此堆栈顶端指针所指向的方格编号便是当前搜索迷宫出口的老鼠所在的位置。如此重复这些动作，直到走到迷宫出口为止。在图8-36和图8-37中以小球代表迷宫中的老鼠。

图8-36　在迷宫中寻找出口

图8-37　终于找到迷宫出口

8.3.2　八皇后算法

八皇后问题也是常见的回溯法应用的一个实例。在国际象棋中，皇后可以对棋盘中的其他棋子竖吃、横吃和对角斜吃（左斜吃或右斜吃都可以）。现在要放入多个皇后到棋盘上，相互之间不能吃到对方。后放入的新皇后在放入前必须考虑所放位置的垂直方向、横线方向或对角线方向是否已被放置了旧皇后，否则会被先放入的旧皇后吃掉。

参照这种规则，我们可以将其应用在4×4的棋盘上，称为4-皇后问题；应用在8×8的棋盘上，称为8-皇后问题。应用在N×N的棋盘上，就称为N-皇后问题。要解决N-皇后问题（在此我们以8-皇后为例），首先在棋盘中放入一个新皇后，且这个位置不会被先前放置的皇后吃掉，然后将这个新皇后的位置压入堆栈。

但是，如果当放置新皇后的该行（或该列）的8个位置都没有办法放置新皇后时（放入任何一个位置，都会被先前放置的旧皇后吃掉），就必须从堆栈中弹出前一个皇后的位置，并在该行（或该列）中重新寻找另一个新的位置来放，再将该位置压入堆栈中，这种方式就是一种回溯算法的应用。

N-皇后问题的解答是结合堆栈和回溯两种数据结构，以逐行（或逐列）寻找新皇后合适位置（如果找不到，就回溯到前一行寻找前一个皇后的另一个新位置，以此类推）的方式来寻找N-皇后问题的其中一组解答。

下面分别是4-皇后和8-皇后在堆栈存放的内容以及对应棋盘的其中一组解，如图8-38和图8-39所示。

图8-38　4-皇后问题其中的一组解

图8-39　8-皇后问题其中的一组解

8.4 课后习题

1.什么是迭代法？

2.枚举法的核心概念是什么？

3.回溯法的核心概念是什么？

4.请简述基数排序法的主要特点。

5.下列叙述正确与否？请说明原因。

（1）无论输入什么数据，插入排序的元素比较总次数比冒泡排序的元素比较总次数要少。

（2）输入数据已排序完成，再使用堆积排序时，只需O（n）（n为元素个数）时间即可完成排序。

6.待排序的关键字的值如下，请使用冒泡排序法列出每个回合排序的结果：

26、5、37、1、61

7.待排序的关键字的值如下，请使用选择排序法列出每个回合排序的结果：

26、5、37、1、61

第9章　游戏设计中的算法

移动设备成就了智能手机发展的新趋势，同时带动了手机游戏的快速崛起，不少人在上班途中或等人的时候都会拿出智能手机来玩游戏。游戏已经渐渐成为人们生活的一部分，就像电影一样，成为一种休闲方式，进而成为家庭休闲娱乐的最新选择之一。

从广义的角度来看，游戏设计是一门集现代科学与计算机理论之大全的学科，其中综合了数学、物理（例子见图9-1）、光学、二维（2D）与三维（3D）图形学，例如碰撞处理、图的遍历、平移与远景三维坐标转换等算法。这些理论和算法无论是应用在软件工程上还是游戏程序的开发上，从程序设计的角度来看，都占据了举足轻重的位置。将真实世界中的自然现象在游戏中呈现，对于游戏设计来说至关重要，对数学和物理相关知识的熟悉度往往成为游戏程序设计能否顺利甚至成败的关键。

图9-1　物理学原理也是由算法来实现的

9.1 游戏中的数学与物理算法

有许多程序设计人员在程序设计语言的运用方面“功力”十足，但是对游戏中的数学和物理原理的理解稍显不足，因而设计出来的游戏常有一些不自然的动作，这些缺失往往会影响游戏玩家对游戏的认同。在本节中，我们将整理出各种经常被应用于游戏制作的数学和物理算法，我们将用最容易理解的方式引导大家熟悉这些知识。

9.1.1　两点距离的算法

在二维系统中，定义两个点A和B，坐标分别为（x1，y1）与（x2，y2），A、B两点之间的距离为：x轴方向的坐标差的平方，加上y轴方向的坐标差的平方，再开平方根，公式如下（见图9-2）：

x=x2-x1

y=y2-y1

两点之间的距离

图9-2　求两点之间的距离

通常求两点之间的距离会使用到平方根的计算，这会花费计算机极大的运算资源，为了加快程序的执行，就要避免平方根的运算。例如，在游戏中球体间碰撞的测试，由于只要判断是否发生碰撞，并不一定要精确地计算出碰撞的范围大小，因此可以省略平方根计算的繁复程序。又如，两点之间的距离也可以应用在射击游戏中，通过射程距离远近的判断来决定所有子弹大小呈现的外观。另外，在类似高尔夫球游戏的制作过程中，也常会使用到距离的运算，通过两点之间距离的计算可以精确地求出球与洞口的距离。

同理，在三维系统中，定义两个点A和B，坐标分别为（x1，y1，z1）与（x2，y2，z2），两点之间的距离为：x轴方向的坐标差的平方，加上y轴方向的坐标差的平方，再加上z轴方向的坐标差的平方，然后开平方根，计算公式如下：

x=x2-x1

y=y2-y1

z=z2-z1

两点之间的距离

9.1.2　匀速运动

所谓“速度”，是指单位时间内所改变距离的量。物体会移动，那么这个物体一定具有“速度”，速度是物体在各个方向上“速度分量”的合成。例如，描述一个人跑步每小时10公里，我们就称这个人的跑步速度为时速70公里。在游戏中，要表现速度时，只要在物体坐标位置上加上一个速度常量，这个物体就会在游戏中朝指定的方向匀速移动。

以一个在二维平面上移动的物体为例，假设它的移动速度为V，x轴方向上的速度分量为V_x，y轴方向上的速度分量为V_y，那么V与V_x、V_y间的关系如图9-3所示。

图9-3　合成速度V的分量

匀速运动是指物体在每一个时刻的速度都相同，即V_x与V_y都保持不变。在设计二维平面上物体匀速运动时，总会在每次画面更新时，使用物体速度分量V_x与V_y的值来计算下次物体出现的位置，以产生物体移动的效果，计算公式可表示如下：

下次x轴位置=现在x轴位置+x轴上的速度分量

下次y轴位置=现在y轴位置+y轴上的速度分量

我们所设计的程序中小球以匀速运动的结果如图9-4所示。

图9-4　匀速运动的小球

9.1.3 加速运动

从物理学的角度来说，凡是物体移动时，其运动的速度或方向会随着时间而改变，那么该物体的运动便属于加速运动。加速度是指单位时间内速度改变的速率，平均加速度则为单位时间内物体速度的变化量，单位为m/s^2。例如，当我们踩下车子的刹车时，车速会递减，直到车子到静止的状态。加速运动不同于匀速运动，它是一种变量，当物体在空间中移动的速度越来越快或越来越慢时，我们只有靠加速度这个变量来计算或测量。例如，高铁进站和出站时都会受到加速度的影响，如图9-5的示。

图9-5　高铁进站和出站时都会受到加速度的影响

加速度通常被应用于设计二维游戏的物理移动，一般物体的移动速度或者方向改变时，都是受到加速度的影响。加速度与速度的关系如下：

$$V=V_o+A\times t$$

在上面的公式中，A表示每一时间间隔加速度的量，t表示物体运动从开始到要计算时间点为止所经过的时间间隔，V_o为物体原来所具有的速度，而V则是由以上公式计算出来的某一时刻物体的运动速度。

作用于物体上的加速度是各个方向上“加速度分量”的合成，加速度作用于物体上时，可根据上面的公式来计算如何影响物体原有的移动速度。而在二维平面上运动的物体，根据上面的公式，考虑X、Y轴上加速度分量对于速度分量的改变，其下一时刻（前一时刻与下一时刻的时间间隔t=1）X、Y轴上的速度分量V_{x1}与V_{y1}的计算方式如下：

$$V_{x1}=V_{xo}+A_x$$

$$V_{y1}=V_{yo}+A_y$$

在上面的公式中，V_{xo}与V_{yo}为物体前一时刻在X、Y轴上的运动速度，A_x与A_y为在X、Y轴上的加速度。在求出物体下一时刻的移动速度后，便可依此推算出加入加速度后，物体下一时刻所在的位置：

$$S_{x1}=S_{xo}+V_{x1}$$

$$S_{y1}=S_{yo}+V_{y1}$$

在上面的公式中，S_{xo}与S_{yo}分别表示物体前一时刻在X、Y轴的坐标位置，V_{x1}与V_{y1}是加入加速度后下一时刻物体的移动速度，如此求出的S_{x1}与S_{y1}便是下一刻物体的位置。

9.2 图的遍历算法

在一个图G=（V，E）中，存在某一顶点v-V，我们希望从v开始，通过此节点相邻的节点而去访问图G中的其他节点，这就被称为“图的遍历”。也就是从顶点V_1开始，可以经过V_1到达某个顶点，接着访问下一个顶点，直到全部的顶点访问完毕为止。在遍历的过程中，可能会重复经过某些顶点和边。通过图的遍历可以判断该图是否连通，并找出图的连通分支和路径。在本节中，我们要介绍三种图遍历的方法：“路径算法”“深度优先遍历”和“广度优先遍历”。

9.2.1 路径算法

路径算法是图的应用中的一种，在当前的游戏设计中占有相当重要的地位。例如角色扮演游戏（Role Playing Games，RPG）、模拟类游戏（Simulation Game，SLG）、益智类型游戏都会用到路径算法。事实上，在游戏地图中，路径算法大多以4个方向的移动为主，即上、下、左、右4个方向，也就是不能直接按斜角方向移动，可参照图9-6左上角的4个方向移动。但是，如果需要，也可以设计成8个方向移动和6个方向移动，如图9-6右上角的图和下图所示。

游戏中所用的路径算法有许多种，下面要介绍的逼近法是最简单的算法，用来计算直接从当前的坐标渐渐移向目的地坐标，这种算法常用于游戏地图中没有任何障碍物移动的情况，例如在空气中、在水中等。

图9-6　路径算法中可以选择实现的移动方向

在图9-7中，玩家要从A点到B点，一共有三种路径计算方式，以路径1来说，先逼近Y轴，再逼近X轴，就可以得到路径1的行走路线；而路径2是先逼近X轴的结果。至于路径3的计算方式，则是比较X与Y的距离比例，最先逼近差异值最高的轴，在逼近过程中，因为X和Y的差异比例会发生变化，于是就会出现我们在图9-7中所看到的路径3。

图9-7　从A点到B点的路径计算一共有三种方式

三种路径的行走距离一样长，路径3的走法是因为在逼近过程中X和Y的差异比例发生变化而造成的，看起来逼近法似乎已经很不错了。不过，在图9-8中，我们可以看到在第14步时逼近法就已经失效了，无论是X轴还是Y轴都没有办法按照原来的算法进行计算。

图9-8　路径算法中逼近法失效的情况

由此可见，逼近法只是计算路径的简单工具，没有办法应付复杂的地形，需要借助其他的算法来解决。这里只是介绍了有关路径算法的基本概念。

9.2.2　深度优先查找算法

深度优先查找（Depth-First Search，DFS）算法也称为深度优先搜索算法，有点类似于前序遍历法，因为查找的过程也是遍历的过程，所以也习惯叫深度优先遍历。从图的某一顶点开始遍历，被访问过的顶点就做上已访问的记号，接着遍历此顶点的所有相邻且未访问过的顶点中的任意一个顶点，并做上已访问的记号，再以该点为新的起点继续进行深度优先搜索。

这种图的遍历方法结合了递归和堆栈两种数据结构的技巧，由于此方法会造成无限循环，因此必须加入一个变量，判断该点是否已经遍历完毕。下面我们以图9-9来看看这个方法的遍历过程。

图9-9　以此无向图为例演示深度优先遍历的步骤

步骤01　以顶点1为起点，将相邻的顶点2和顶点5压入堆栈，如图9-10所示。

图9-10　第一次遍历

步骤02　弹出顶点2，将与顶点2相邻且未访问过的顶点3和顶点4压入堆栈，如图9-11所示。

图9-11　第二次遍历

步骤03　弹出顶点3，将与顶点3相邻且未访问过的顶点4和顶点5压入堆栈，如图9-12所示。

图9-12　第三次遍历

步骤04　弹出顶点4，将与顶点4相邻且未访问过的顶点5压入堆栈，如图9-13所示。

图9-13　第四次遍历

步骤05　弹出顶点5，将与顶点5相邻且未访问过的顶点压入堆栈，大家可以发现与顶点5相邻的顶点全部被访问过了，所以无须再压入堆栈，如图9-14所示。

图9-14　第五次遍历

步骤06　将堆栈内的值弹出并判断是否已经遍历过了，直到堆栈内无节点可遍历为止，如图9-15所示。

图9-15　无节点可遍历

故深度优先遍历的顺序为：顶点1、顶点2、顶点3、顶点4、顶点5。

9.2.3　广度优先查找算法

之前所谈到的深度优先查找算法（或深度优先遍历）是使用堆栈和递归的技巧来遍历图的，而广度优先查找（Breadth-First Search，BFS）算法则是使用队列和递归的技巧来查找的，是回溯法的变形，称为分支限界法（Branch and Bound Method）。同理，这个算法也被称为广度优先搜索算法，因为查找的过程也是遍历的过程，于是也习惯叫广度优先遍历。

提示

分支限界法是一种用途十分广泛的算法，类似于回溯法，但是求解目标却不相同，回溯法可以找出满足条件的所有解，分支限界法通常用于求出一个最优解，思路为将树节点不断分支，但随时以问题的条件限制分支的持续，这样的做法可以大幅减少所需查找的路径。

广度优先查找算法（或广度优先遍历）是从图的某一顶点开始遍历，被访问过的顶点就做上已访问的记号，接着遍历此顶点的所有相邻且未访问过的顶点中的任意一个顶点，并做上已访问的记号，再以该点为新的起点继续进行广度优先遍历。下面我们以图9-16来看看广度优先的遍历过程。

图9-16　以此无向图为例演示广度优先遍历的步骤

步骤01　以顶点1为起点，与顶点1相邻且未访问过的顶点2和顶点5加入队列，如图9-17所示。

图9-17　第一次遍历

步骤02　取出顶点2，将与顶点2相邻且未访问过的顶点3和顶点4加入队列，如图9-18所示。

图9-18　第二次遍历

步骤03　取出顶点5，将与顶点5相邻且未访问过的顶点3和顶点4加入队列，如图9-19所示。

图9-19　第三次遍历

步骤04　取出顶点3，将与顶点3相邻且未访问过的顶点4加入队列，如图9-20所示。

图9-20　第四次遍历

步骤05　取出顶点4，将与顶点4相邻且未访问过的顶点加入队列中，大家可以发现与顶点4相邻的顶点全部被访问过了，所以无须再加入队列中，如图9-21所示。

图9-21　第五次遍历

步骤06　将队列内的值取出并判断是否已经遍历过了，直到队列内无节点可遍历为止，如图9-22所示。

图9-22　无节点可遍历

所以，广度优先的遍历顺序为：顶点1、顶点2、顶点5、顶点3、顶点4。

9.3　碰撞处理算法

在游戏设计中，游戏对象的碰撞处理也是一种相当常见的算法，其算法又可以分成好几种，如人物与敌人的碰撞、飞机与子弹的碰撞、为了某些特殊事件而产生的碰撞等。日常生活中的车祸事件就是一种碰撞，如图9-23所示。

图9-23　日常生活中的车祸事件就是一种碰撞

由于在游戏中碰撞检测的方式不止一种，有的碰撞检测处理是按范围来检测的，有的碰撞检测是以颜色来检测的，有的碰撞检测则是以行进路线是否交叉来检测的，等等。

9.3.1　以行进路线来检测

卫星云图中的台风以行进路线判断是否会波及中国台湾地区甚至是中国大陆，如图9-24所示。

图9-24　卫星云图中的台风以行进路线判断是否会波及中国台湾地区甚至是中国大陆

以行进路线来检测游戏中的物体是否碰撞相当简单，主要是检测两个移动的物体或者移动物体与平面是否发生碰撞，如图9-25所示。

图9-25　两球行进路线交叉可能发生碰撞（左图），球行进路线与平面交叉也可能发生碰撞（右图）

我们可以看到在图9-25中，无论是两个球的行进方向还是平面，它们各自都加上了一个箭头，表示为向量。下面将以向量来判断一个具有速度值的小球是否会与斜面（非水平面与垂直面）发生碰撞，在这里先假设小球当前与下一个时刻的圆心位置分别为P_3与P_4，而斜面的起点与终点为P_1与P_2，原点为O，若小球与平面发生碰撞，则碰撞点为C，其碰撞示意图如图9-26所示。

图9-26　小球与斜面发生碰撞的示意图

从图9-26可推导出如下式子：

OP_1C中：$OC=OP_1+P_1C=OP_1+mP_1P_2$

OP_3C中：$OC=OP_3+P_3C=OP_3+nP_3P_4$

$=>OP_1+mP_1P_2=OP3+nP_3P_4$

若交点C在两向量之间，则在上面的式子中，m与n的值会介于0～1之间，其值代表球与斜面是否发生碰撞。

假设斜面的起点坐标P_1为（a，b），而向量P_2-P_1可得（L_x，L_y），小球圆心当前的坐标为（c，d），其速度向量为（V_x，V_y），代入上式为：

（a，b）+m（L_x，L_y）=（c，d）+n（V_x，V_y）

→x轴方向的向量：$a+mL_x=c+nV_x$

y轴方向的向量：$b+mL_y=d+nV_y$

→$m=[V_x（b-d）+V_y（c-a）]/（L_xV_y-L_yV_x）$

$n=[L_x（d-b）+L_y（a-c）]/（V_xL_y-V_yL_x）$

我们导出了m与n的结果之后，在程序中，若要判断小球与斜面是否会发生碰撞，只要将其移动的路径和斜面的向量与起点坐标代入上面的方程式中，然后判断m与n是否都介于0～1之间，就可以得知是否会发生碰撞。

9.3.2　范围检测

按范围检测碰撞的方法其实是最简单且快速的，不过在制作游戏程序时，遇到不规则形状图形的情况相当多，在允许的情况下，还是希望能够按范围检测的方式来检测物体是否发生碰撞，如此将会节省许多计算的时间。按范围检测碰撞的方式适用于具有规则形状且可取得其范围大小的几何图形，如图9-27所示的图形。

图9-27　矩形有交集表示发生碰撞（左图），两圆有交集表示发生碰撞（右图）

基本上，以矩形范围来检测，首要的条件是必须取得矩形的左上角坐标与右下角坐标，如图9-28所示。

图9-28　矩形范围的碰撞检测

这时，如果要判断一个变量坐标是否碰撞到矩形，只要判断这个变量坐标的X值是否在X1、X2之间以及Y值是否在Y1、Y2之间，随后就可以知道这个变量坐标是否碰撞到矩形了，如图9-29所示。

图9-29　判断一个变量坐标是否碰撞到矩形

例如，当两辆不规则形状的车辆在同样的高度上移动时，要检测这两辆车是否碰撞，只需判断这两辆车的图片在水平方向是否有交集，如图9-30所示。

图9-30　两辆车图片的水平方向有交集表示两车发生碰撞

如果两车在不同高度上移动，就必须使用图片矩形的长与高来检测是否发生碰撞，但是这种检测碰撞的方式会产生细微的误差，如图9-31所示。

图9-31　未真正碰撞但已检测到碰撞，这就是误差

在二维游戏中，矩形的碰撞判断属于较简单但不精确的方式，这种方式的指令执行周期比较短，速度较快，而且程序代码比较简单。在下面的程序中，笔者用了三张图片，分别为两张车辆的图案以及发生碰撞时所要显示的图案，将车辆的移动设置在等高的位置上，并使用车辆的两个图片矩形，以这两个图片矩形在水平方向上是否有交集来判断是否发生碰撞。程序的执行结果如图9-32和图9-33所示。

图9-32　未发生碰撞

图9-33　发生碰撞

另外，还有一种球面范围检测，它不像矩形可以使用4个角的坐标来判断变量坐标是否在圆内，如果坚持要使用4个角坐标来判断，那么将会产生如图9-34所示的情况。

图9-34　出现误差

在这个时候，就可以使用数学中表示“圆的方程”来求出变量坐标在球面的哪一个地方，表示圆的方程如下：

若圆心为（h，k），半径为r，则圆的方程为$(x-h)^2+(y-k)^2=r^2$。

例如，有一个圆的圆心为（2，1），半径为2，这个圆的方程如下：

$(x-2)^2+(y-1)^2=2^2\rightarrow(x-2)^2+(y-1)^2=4$

9.3.3 颜色检测

在9.3.2小节中，我们介绍了一些数学公式来作为判断碰撞的条件。不过，在游戏中，无论是主角、敌人还是游戏中的宝物，它们都是没有固定角度的图形（不规则的图形），如图9-35所示。

图9-35　没有固定角度的图形

如果想要更精确地判断不规则形状的物体是否发生碰撞，最常使用的方法是使用颜色来判断。以颜色检测碰撞的方式一般比较麻烦，不过这种碰撞检测方式可以很精确地判断出两个不规则形状的物体是否真的发生了碰撞，假设会发生碰撞的情况如图9-36所示。

图9-36　会发生碰撞的情况

检测车辆是否进入树林中的方法是检测车辆是否与树林发生碰撞，那么该如何使用颜色来判断车辆是否与树林发生碰撞呢？当然，从图9-36中看不出任何蛛丝马迹，因为它无法以任何颜色为基准点来计算出是否发生了碰撞，换一张图来试试看，如图9-37所示。

图9-37　将树林改成黑色的

在图9-37中，将树林改成黑色的，之所以要将树林改成黑色的，是为了把它当作一张“屏蔽图”（也称为暗图），以便能够使用颜色来判断是否发生碰撞。当车辆与树林发生碰撞时，在车头的部分会被黑色所遮蔽。

这时如果要判断是否发生碰撞，关键就在于黑色部分与车辆是否会产生交集，但是又要如何判断车辆与黑色部分有交集呢？方法很简单，因为黑色与任何颜色进行“AND”（与）运算的结果还是黑色，所以如果以车辆颜

色与当前位置上的“屏蔽图”进行AND运算后，就会产生黑色的结果，如此一来，便表示车辆与树林发生了碰撞，前提是车辆图案不可以是纯黑色的。

接下来以颜色判断的方式来检测是否产生了碰撞，思路如图9-38所示。

图9-38　判断思路

在以下设计中，每次当车辆移动时，必须重新进行“透空”以及取得图片中所有像素的颜色值，并进行运算，然后判断是否产生了碰撞。执行结果如图9-39和图9-40所示。

图9-39　车辆在树林中

图9-40　车辆走出树林

9.4 遗传算法

游戏设计者都知道太简单的游戏可能吸引不了玩家，而太复杂的游戏会让受挫的玩家很快就放弃。一些人工智能算法在游戏中的表现能够随着游戏玩家的操控而自行适应，遗传算法（Genetic Algorithm）就符合这样的精神。遗传算法是仿真生物演化与遗传过程的搜索与优化算法，它的理论根基源于John Holland在1975年提出的“进化论”，他借用的是生物学家达尔文（Charles Darwin）所提出的进化论“物竞天择，适者生存”的概念，再进一步发展，就是一种解决优化问题的工具与不断改进群体适应的算法。

在真实世界中，物种的演化（Evolution）是为了更适应大自然的环境，而在演化过程中，某个遗传基因的改变也能让下一代来继承，以繁衍更优越的下一代，这些物种也是最适宜继续存活下去的生物。演化是使用选择步骤建立新对象的过程，例如设计团队要制作出游戏动画中人物行走的画面，通常需要事先仔细描述每个画面的细节，如果运用遗传算法，把重力和人物（或角色）的肌肉结构都做好关联后，再来指引剧中人物的走路情况，可针对游戏AI的角色或游戏场景、敌人等进行相应的设计，将遗传算法的抽象概念转化为具体的呈现，也就是游戏世界中的元素，也可以演化并适应不同的环境。

在游戏设计中，玩家可以挑选自己喜欢的角色来扮演，不同的角色有不同的特质与挑战性，与生物在自然界中的生存模式相似，游戏设计人员无法事先预知或了解玩家打算扮演的角色。这时为了响应不同的情况，就可以将可能的场景指定给某个“染色体”，例如在角色扮演类游戏中，玩家可以挑选自己喜欢的角色来扮演，记录游戏玩家的攻击行为和游戏AI本身的应对方式，通过“适者生存”的原则，不同的角色都有各自不同的特质与挑战性。

在遗传算法中，实际上是一个优化过程，试着找出适应力最强的一组特征，每个自变量都可以叫一种染色体，主要

的结构有遗传基因编码方式、适应度函数（Fitness Function）、选择机制（Selection Mechanism）、交叉变异机制（Crossover and Mutation Mechanism），编码是把染色体存储在计算机某个数据结构中的过程，遗传算法之所以能自我进化，关键在于以适应度函数自动调整优胜劣汰的遗传基因组合，经由繁殖（Reproduction）、交叉（Crossover）、变异（Mutation）的演化，使用不同的染色体来存储每种情况的演化。

和所有遗传算法一样，首先必须建立第一代，在生物世界中，通常是父母双亲贡献染色体给后代。在游戏开发中，开发人员不必考虑生物世界的局限性，双亲如何配对可以变通，以一个遗传基因代表一个实数或一个整数，甚至可以只是一个位（bit），每一条染色体即为一组解，而每一条染色体是由许多遗传基因组成的，留下好的染色体，不好的个体则会被淘汰，因而绝种。使用不同的染色体来存储每种情况的演化，像是运用遗传算法，把重力和游戏角色的肌肉结构都做好关联后，就可让角色走得非常顺畅，或者当遗传基因发生突变时，人物的战斗力变强，如图9-41和图9-42所示。

图9-41　把重力和角色的肌肉结构都做好关联后，就可以让游戏角色走得非常顺畅

图9-42　当遗传基因发生突变时，人物的战斗力变强

John Holland提出的遗传算法其实就是模仿大自然界“物竞天择”法则和遗传基因交叉的法则。对于以往传统人工智能方法无法有效解决的计算问题，它都可以快速地找出答案。遗传算法是一种特殊的搜索技巧，适合处理多变量与非线性的问题，我们可以使用图9-43来表示演化过程。

图9-43　遗传算法的演化过程

9.5　课后习题

1.碰撞处理中的范围检测是什么？

2.什么是遗传算法？试举例说明在游戏中的应用。

3.根据下图求解以下问题：

（1）使用深度优先算法求出生成树。

（2）使用广度优先算法求出生成树。

4.什么是加速运动？

5.什么是分支限界法？

附录　课后习题与参考答案

第1章　课后习题与参考答案

1.下面两组词汇都有共同点，各自有一个不同，请找出不同的词汇，并说明差异之处。

（1）A.蛇

B.玫瑰

C.狗

D.老虎

（2）A.熊

B.兔子

C.老鹰

D.狼

E.狐狸

答：（1）B（玫瑰是植物）；（2）B（兔子是食草动物）。

2.算法必须符合哪5项条件？

答：

3.请找出以下序列的模式，并写出“？”处的值。

（1）151、242、333、424、？

（2）CEG、EHK、JN、？

（3）65536、256、16、？

答：515、R、4。

4.试简述思维导图的由来。

答：思维导图是由英国的Tony Buzan于20世纪70年代提出的一种辅助思考的工具，又称脑力激荡图、思维图，它是一种使用图形来帮助思考与表达思维的工具，可以刺激思维并帮助整合思想与信息。借助这种方式，我们可以更轻松地以图形来表达自己的想法。

5.把（2004）$_{10}$转换为十六进制数的结果是多少？

答：（07D4）$_{16}$。

6.求二进制数（11.1）$_2$的平方，即（11.1）$_2$×（11.1）$_2$的值。

答：（1100.01）$_2$。

7.请问算法和过程有什么不同？

答：算法和过程是有区别的，过程不一定要满足算法有限性的要求，例如操作系统或计算机上运行的过程，除非宕机，否则永远在等待循环中，这就违反了算法五大条件中的“有限性”。

8.请简述云计算与物联网。

答：“云”泛指“网络”，这个名字的源头是工程师通常把网络架构图中不同的网络用“云朵”的形状来表示。云计算就是将网络连接的各种计算设备的运算能力提供出来作为一种服务，只要用户可以通过网络登录远程服务器进行操作，就可以使用这种计算资源。

“物联网”是近年来信息产业界的一个非常热门的议题，它是指将各种具有传感器或感测设备的物品（例如RFID、环境传感器、全球定位系统等）与因特网结合起来，并通过网络技术让各种实体对象自动彼此沟通和交换信息，也就是通过巨大的网络把所有东西都连接在一起。

9.什么是“计算思维”？

答：计算思维是一种使用计算机的逻辑来解决问题的思维，前提是具有程序设计的基本概念，是一种能够将计

算“抽象化”再“具体化”的能力，也是新一代人才都应该具备的素养。

10.谷歌公司为教育者开发了一套计算思维课程，这套课程提到培养计算思维有哪4部分？

答：这套课程提到培养计算思维的4部分，分别是分解、模式识别、模式概括与抽象、算法。

11.什么是“模式”？什么是“模式识别”？

答：在将一个复杂的问题分解之后，我们常常可以发现小问题中有共同的属性以及相似之处，在计算思维中，这些属性被称为“模式”。模式识别是指在一组数据中找出特征或规则，用于对数据进行识别与分类，以作为决策判断的依据。

第2章　课后习题与参考答案

1.请简单说明堆栈与队列的主要特性。

答：堆栈是一组相同数据类型的数据的组合，所有的操作均在顶端进行，具有“后进先出”的特性。队列和堆栈一样，都是一种有序列表，也属于抽象数据类型，它所有加入与删除的操作都发生在队列的两端，并且符合“先进先出”的特性。

2.数据结构主要用于表示数据在计算机内存中存储的位置和模式，通常可以分为哪三种类型？

答：（1）基本数据类型

（2）结构数据类型

（3）抽象数据类型

3.在单向链表类型的数据结构中，根据所删除节点的位置会有哪三种不同的情况？

答：根据所删除节点的位置会有以下三种不同的情况。

（1）删除链表的第一个节点：只要把链表头指针指向第二个节点即可。

（2）删除链表后的最后一个节点：只要指向最后一个节点的ptr指针直接指向None即可。

（3）删除链表内的中间节点：只要将删除节点的前一个节点的指针指向要删除节点的下一个节点即可。

4.什么是类神经网络？

答：类神经网络是模仿生物神经网络的运行模式，取材于人类大脑的结构，基础研究的方向是：使用大量简单且相连的人工神经元来模拟生物神经细胞受到特定程度的刺激而反应刺激。由于类神经网络具有高速运算、记忆、学习与容错等能力，因而可以使用一组范例，通过神经网络模型建立系统模型，用于推理、预测、评估、决策、诊断的相关应用。要使得类神经网络能正确地运行，必须通过训练的方式让类神经网络反复学习，经过一段时间学习获得经验值，才能有效学习到初步运行的模式。

5.请说明稀疏矩阵的定义，并举例说明。

答：一个矩阵中大部分元素为0，即可称为“稀疏矩阵”。例如下图的矩阵就是典型的稀疏矩阵。

6.请简单介绍GPU。

答：GPU是近年来科学计算领域的最大变革，是指以图形处理单元（GPU）搭配微处理器的新型计算方式。因为GPU含有数千个微型且更高效率的运算单元，可以有效进行并行计算，所以大幅提高了运算性能。GPU的运用加速了科学、分析、工程、消费和企业的应用，另外，GPU的应用更因为人工智能的快速发展开始发生截然不同的新转变。

7.机器学习是什么？有哪些应用？

答：机器学习是大数据与人工智能发展相当重要的一环，是人工智能的一个分支，机器通过算法来分析数据，在大数据中找到规则。机器学习是大数据发展的下一个阶段，

可以发掘出多种数据变动因素之间的关联性，充分利用大数据和算法来训练机器，让它学习如何执行任务，其应用范围相当广泛，从健康监控、自动驾驶、机台自动控制、医疗成像诊断工具、工厂控制系统、检测用机器人到网络营销领域。

8.请解释下列哈希函数的相关名词。

（1）bucket（桶）

（2）同义词

（3）完美哈希

（4）碰撞

答：

（1）bucket（桶）：哈希表中存储数据的位置，每一个位置对应唯一的一个地址。桶就好比存在一个记录的位置。

（2）同义词：当两个标识符I_1和I_2经哈希函数运算后所得的数值相同时，即$f（I_1）=f（I_2）$，就称I_1与I_2对于f这个哈希函数是同义词。

（3）完美哈希：指既没有碰撞又没有溢出的哈希函数。

（4）碰撞：如果两项不同的数据经过哈希函数运算后，对应相同的地址，就称为碰撞。

9.一般树结构在计算机内存中的存储方式以链表为主，对于n叉树来说，我们必须取n为链接个数的最大固定长度，请说明为了避免树结构存储空间浪费的缺点，我们最常使用二叉树结构来取代树结构。

答：假设此n叉树有m个节点，那么此树共用了n*m个链接字段。另外，因为除了树根外，每一个非空链接都指向一个节点，所以得知空链接个数为n*m-（m-1）=m*（n-1）+1，而n叉树的链接浪费率为。因此，我们可以得到以下结论：

n=2时，2叉树的链接浪费率约为1/2。

n=3时，3叉树的链接浪费率约为2/3。

n=4时，4叉树的链接浪费率约为3/4。

……

故而当n=2时，它的链接浪费率最低。

10.请将下面的树转化为二叉树。

答：

第3章　课后习题与参考答案

1.试简述分治法的核心精神。

答：分治法的核心思想在于将一个难以直接解决的大问题按照不同的分类分割成两个或更多的子问题，以便各个击破，分而治之。

2.递归至少要定义哪两个条件？

答：递归至少要定义2个条件：①可以反复执行的递归过程；②跳出递归过程的出口。

3.请问使用二分查找法的前提条件是什么？

答：必须存放在可以直接存取且已排好序的文件中。

4.有关二分查找法，下列叙述哪一个是正确的？

A.文件必须事先排序

B.当排序的数据量非常少时，二分查找法的速度比顺序查找法的速度慢

C.排序的复杂度比顺序查找法高

D.以上都正确

答：D。

5.试说明在汉诺塔问题中，移动n个圆盘所需的最小移动次数是多少。

答：本书中提过当有n个圆盘时，可将汉诺塔问题求解的过程归纳为三个步骤，其中a_n为移动n个圆盘所需的最少移动次数，a_{n-1}为移动n-1个圆盘所需的最少移动次数，$a_1=1$为只剩一个圆盘时的移动次数，因此可得如下式子：

由此可知，要移动n个盘子所需的最小移动次数为2^n-1次。

6.待排序关键字的值如下，请使用合并排序法列出每一回合排序的结果：

11、8、14、7、6、8+、23、4

答：

7.请简单介绍快速排序法。

答：快速排序是由C.A.R.Hoare提出来的，这种排序法又被称为分割交换排序法，是目前公认最佳的排序法，也是使用“分而治之”的方式。排序开始后，会先在数据中找到一个虚拟的中间值，并按此中间值将所有要排序的数据分为两部分。其中小于中间值的数据放在中间值的左边，而大于中间值的数据放在中间值的右边，再以同样的方式分别处理左右两边的数据，直到排序完成为止。

第4章　课后习题与参考答案

1.试简述贪心法的主要概念。

答：贪心法又称为贪婪算法，方法是从某一起点开始，在每一个解决问题的步骤中使用贪心原则，即采取在当前状态下最有利或最优化的选择，不断地改进该解答，持续在每一个步骤中选择最佳的方法，并且逐步逼近给定的目标，当达到某一个步骤不能再继续前进时，算法就停止，就是尽可能快地求得更好的解。

2.什么是生成树？生成树应该包含哪些特点？

答：一个图的生成树是以最少的边来连接图中所有的顶点，且不造成回路的树结构。由于生成树是由所有顶点和遍历过程经过的边所组成的，令S=（V，T）为图G中的生成树，该生成树具有以下几个特点：

（1）E=T+B。

（2）将集合B中的任意一边加入集合T中，就会造成回路。

（3）V中任意两个顶点Vi和Vj，在生成树S中存在唯一的一条简单路径。

3.请简述用Prim算法求解一个无向连通图的最小生成树的主要过程。

答：Prim算法又称为P氏法。针对一个加权图G=（V，E），设V={1，2，…，n}，假设U={1}，也就是说，U和V是两个顶点的集合。从V-U所产生的集合中找出一个顶点x，该顶点x能与U集合中的某个顶点形成最小成本的边，且不会造成回路。然后将顶点x加入U集合中，反复执行同样的步骤，一直到U集合等于V集合（U=V）为止。

4.请简述用Kruskal算法求解一个无向连通图的最小生成树的主要过程。

答：Kruskal算法是将各边按权值大小从小到大排列，接着从权值最低的边开始建立最小成本生成树，如果加入的边会造成回路，就舍弃不用，直到加入n-1条边为止。

5.请简述A*算法的优点。

答：A*算法是Dijkstra算法的改进版，它结合了在路径查找过程中从起点到各个顶点的“实际权重”和各个顶点预估到达终点的“推测权重”两个因素，这个算法可以有效地减少不必要的查找操作，从而提高查找最短路径的效率。

6.请用K氏法求出下图中的最小成本生成树。

答：

7.在注有各地距离的图上（各地之间都是单行道），求各地之间的最短距离。分别求解以下两道题。

（1）写出下图的邻接矩阵。

（2）写出下图各个顶点之间最短距离的矩阵，表示出所求得的各个顶点之间的最短距离，即各地间最短的距离。

答：

（1）

（2）

第5章　课后习题与参考答案

1.简述动态规划法与分治法的差异。

答：动态规划法主要的做法是：如果一个问题的答案与子问题相关，就能将大问题拆解成各个小问题，其中与分治法最大的不同之处是可以让每一个子问题的答案被存储起来，以供下次求解时直接取用。这样的做法不但能减少再次计算的时间，而且可以将这些解组合成大问题的解，故而使用动态规划可以解决重复计算的问题。

2.简述拓扑排序的步骤。

答：拓扑排序的步骤如下：

步骤01　寻找图中任何一个没有先行者的顶点。

步骤02　输出此顶点，并将此顶点的所有边删除。

步骤03　重复以上两个步骤依次处理所有的顶点。

3.求下图的拓扑排序。

答：拓扑排序为A→B→C→D→E或B→A→C→D→E。

4.求下图的拓扑排序。

答：7，1，4，3，6，2，5。

5.什么是关键路径？

答：所谓关键路径，就是AOE有向图从源头顶点到目的顶点之间所需花费时间最长的一条有方向性的路径。当有一条以上的路径花费时间相等而且都是最长时，这些路径都称为此AOE有向图的关键路径。

6.什么顶点活动网络？

答：网络图主要用来协助规划大型项目，首先我们将复杂的大型项目细分成很多工作项，而每一个工作项代表网络的一个顶点，由于每一项工作可能有完成的先后顺序，有些可以同时进行，有些则不行。因此，可用网络图来表示其先后完成的顺序。这种以顶点代表工作项的网络称为顶点活动网络，简称AOV网络。

第6章　课后习题与参考答案

1.信息安全必须具备哪4种特性，请简要说明。

答：

·保密性：表示交易相关信息或数据必须保密，当信息或数据传输时，除了被授权的人外，要确保信息或数据在网络上不会遭到拦截、偷窥而泄露信息或数据的内容，损害其保密性。

·完整性：表示当信息或数据送达时，必须保证该信息或数据没有被篡改，如果遭篡改，那么这条信息或数据就会无效。

·认证性：表示当传送方送出信息或数据时，支付系统必须能确认传送者的身份是否为冒名。

·不可否认性：表示保证用户无法否认他所实施过的信息或数据传送行为的一种机制，必须不易被复制及修改，即无法否认其传送或接收信息或数据的行为。

2.请简述“加密”与“解密”。

答：“加密”就是将数据通过特殊算法，把源文件中的内容转换为无法读取的密文（看上去像乱码）。而当加密后的数据传送到目的地后，将密文还原成明文的过程就称为“解密”。

3.请说明“对称密钥加密”与“非对称密钥加密”二者的差异。

答：“对称密钥加密”的工作方式是：发送端与接收端用于加密和解密的密钥是同一把。“非对称密钥加密”的工作方式是：使用两把不同的密钥进行加密和解密，一把“公钥”和一把“私钥”。

4.请简要介绍RSA算法。

答：RSA算法是一种非对称加密算法，在RSA算法之前，加密算法基本都是对称的。非对称加密算法使用两把不同的密钥，一把叫公钥，另一把叫私钥，它是在1977年由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）一起提出的，RSA就是由他们三人姓氏开头字母所组成的。RSA加解密速度比“对称密钥加解密”速度要慢，方法是随机选出两个超大的质数p和q，使用这两个质数作为加密与解密的一对密钥，密钥的长度一般为40比特到1024比特之间。当然，为了提高加密的强度，现在有的系统使用的RSA密钥的长度高达4096比特，甚至更高。在这对密钥中，公钥用来加密，私钥用来解密，而且只有私钥可以用来解密。要破解以RSA加密的数据，在一定时间内几乎是不可能的，因此这是一种十分安全的加解密算法，特别是在电子商务交易市场被广泛使用。

5.试简要说明数字签名。

答：“数字签名”的工作方式是以公钥和哈希函数互相搭配使用的，用户A先将明文的M以哈希函数计算出哈希值H，再用自己的私钥对哈希值H加密，加密后的内容即为“数字签名”。

6.用哈希法将下列7个数字存在0、1、…、6七个位置：101、186、16、315、202、572、463。若要存入1000开始的11个位置，应该如何存放？

答：

f（X）=X mod 7

f（101）=3

f（186）=4

f（16）=2

f（315）=0

f（202）=6

f（572）=5

f（463）=1

同理取：

f（X）=（X mod 11）+1000

f（101）=1002

f（186）=1010

f（16）=1005

f（315）=1007

f（202）=1004

f（572）=1000

f（463）=1001

7.什么是哈希函数？试以除留余数法和折叠法并以7位电话号码作为数据进行说明。

答：

以下列6组电话号码为例：

（1）9847585

（2）9315776

（3）3635251

（4）2860322

（5）2621780

（6）8921644

·除留余数法

利用f_D（X）=X mod M，假设M=10。

f_D（9847585）=9847585mod 10=5

f_D（9315776）=9315776mod 10=6

f_D（3635251）=3635251mod 10=1

f_D（2860322）=2830322mod 10=2

f_D（2621780）=2621780mod 10=0

f_D（8921644）=8921644mod 10=4

·折叠法

将数据分成几段，除最后一段外，每段长度都相同，再把每段值相加。

f（9847585）=984+758+5=1747

f（9315776）=931+577+6=1514

f（3635251）=363+525+1=889

f（2860322）=286+032+2=320

f（2621780）=262+178+0=440

f（8921644）=892+164+4=1060

8.试简述哈希查找与一般查找技巧有何不同。

答：一般而言，一个查找法的好坏主要由其比较次数和查找时间来决定。一般的查找技巧主要是通过各种不同的比较方式来查找所需要的数据项，反观哈希，则是直接通过数学函数来取得对应的地址，因此可以快速找到所要的数据。也就是说，在没有发生任何碰撞的情况下，其比较时间只需O（1）的时间复杂度。除此之外，它不仅可以用来进行查找的工作，还可以很方便地使用哈希函数来进行创建、插入、删除与更新等操作。重要的是，通过哈希函数进行查找的文件事先不需要排序，这也是它和一般的查找差异比较大的地方。

9.什么是完美哈希？在哪种情况下可以使用？

答：所谓完美哈希，是指该哈希函数在存入与读取的过程中，不会发生碰撞或溢出，一般而言，只有在静态表中才可以使用。

10.采用哪一种哈希函数可以把整数集合：{74，53，66，12，90，31，18，77，85，29}存入数组空间为10的哈希表不会发生碰撞？

答：采用数字分析法，并取出键值的个位数作为其存放地址。

第7章　课后习题与参考答案

1.请问以下二叉树的中序法、后序法以及前序法表达式分别是什么？

答：中序：A/B**C+D*E-A*C

后序：ABC**/DE*+AC*-

前序：-+/A**BC*DE*AC

2.关于二叉查找树的叙述，哪一个是错的？

A．二叉查找树是一棵完整二叉树

B．可以是歪斜树

C．一个节点最多只有两个子节点

D．一个节点的左子节点的键值不会大于右节点的键值

答：A。

3.形成8层的平衡树最少需要几个节点？

答：因为条件是形成最少节点的平衡树，不但要最少，而且要符合平衡树的定义。在此我们逐一讨论：

（1）一层的最少节点的平衡树：

（2）二层的最少节点的平衡树：

（3）三层的最少节点的平衡树：

（4）四层的最少节点的平衡树：

（5）五层的最少节点的平衡树：

由以上讨论得知：

所以第8层最少节点的平衡树有54个节点。

4.请将下图的树转换化为二叉树。

答：

5.在下图的平衡二叉树中，加入节点11后，重新调整后的平衡树是什么？

答：

6.有一棵二叉查找树：

（1）键值key平均分配在[1，100]之间，求在该查找树查找平均要比较几次。

（2）假设k=1时，其概率为0.5，k=4时，其概率为0.3，k=9时，其概率为0.103，其余97个数的概率为0.001，求在该查找树查找平均要比较几次。

（3）假设各key的概率如（2），是否能将此查找树重新安排？

（4）以得到的最小平均比较次数绘出重新调整后的查找树。

答：

（1）2.97次。

（2）2.997次。

（3）可以重新安排此查找树。

（4）

7.请建立一棵最小堆积树，必须写出建立此堆积树的每一个步骤。

答：

根据最小堆积树的定义：

（1）是一棵完全二叉树。

（2）每一个节点的键值都小于其子节点的值。

（3）树根的键值是此堆积树中最小的。

建立好的最小堆积树为：

8.试以数列26、73、15、42、39、7、92、84说明堆积排序的过程。

答：请参考本章介绍的方法，输出顺序为7、15、26、39、42、73、84、92。

9.什么是博弈树？试举例说明。

答：博弈树使用树结构的方法来讨论一个问题的各种可能性。下面用最典型的“八枚金币”问题来阐述博弈树的概念，内容是假设有8枚金币a、b、c、d、e、f、g、h，其中有一枚金币是伪造的，伪造金币的特征是重量稍轻或偏重。如何使用博弈树的方法来找出这枚伪造的金币？以L表示伪造的金币轻于真品，H表示伪造的金币重于真品。

第一次比较时，从8枚金币中任挑6枚a、b、c、d、e、f，分成2组来比较重量，会出现下列三种情况：

（a+b+c）＞（d+e+f)

（a+b+c）＝（d+e+f)

（a+b+c）＜（d+e+f)

我们可以按照以上步骤画出博弈树：

10.请简要介绍哈夫曼树。

答：哈夫曼树经常应用于处理数据压缩，可以根据数据出现的频率来构建二叉树。例如，数据的存储和传输是数据处理的两个重要领域，两者都和数据量的大小息息相关，

而哈夫曼树正好可以用于数据压缩的算法。简单来说，如果有n个权值（q_1，q_2，…，q_n），且构成一个有n个节点的二叉树，每个节点的外部节点的权值为q_i，则加权外径长最小的就称为“优化二叉树”或“哈夫曼树”。

第8章　课后习题与参考答案

1.什么是迭代法？

答：迭代法是指无法使用公式一次求解，而需要使用迭代，例如用循环重复执行程序代码的某些部分来得到答案。

2.枚举法的核心概念是什么？

答：枚举法的核心思想是：列举所有的可能。根据问题要求逐一列举问题的解答，或者为了便于解决问题，把问题分为不重复、不遗漏的有限种情况，逐一列举各种情况，并加以解决，最终达到解决整个问题的目的。

3.回溯法的核心概念是什么？

答：回溯法也是枚举法的一种，对于某些问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，同时避免枚举不正确的数值。一旦发现不正确的数值，就不再递归到下一层，而是回溯到上一层，以节省时间，是一种走不通就退回再走的方式。

4.请简述基数排序法的主要特点。

答：基数排序法并不需要进行元素之间的直接比较操作，它属于一种分配模式的排序方式。基数排序法按比较的方向可分为最高位优先和最低位优先两种。MSD是从最左边的位数开始比较，而LSD则是从最右边的位数开始比较。

5.下列叙述正确与否？请说明原因。

（1）无论输入什么数据，插入排序的元素比较总次数比冒泡排序的元素比较总次数要少。

（2）输入数据已排序完成，再使用堆积排序时，只需O（n）（n为元素个数）时间即可完成排序。

答：

（1）错。提示：当有n个已排好序的输入数据时，两种方法的比较次数相同。

（2）错。在输入数据已排好序的情况下，需要O（nlogn）。

6.待排序的关键字的值如下，请使用冒泡排序法列出每个回合排序的结果：

26、5、37、1、61

答：

7.待排序的关键字的值如下，请使用选择排序法列出每个回合排序的结果：

26、5、37、1、61

答：

第9章　课后习题与参考答案

1.碰撞处理中的范围检测是什么？

答：按范围检测碰撞的方法其实是最简单且快速的，不过在制作游戏程序时，碰到不规则形状图形的情况相当多，在允许的情况下，还是希望能够按范围检测的方式来检测物体是否发生碰撞，如此将会省掉许多计算的时间。

2.什么是遗传算法？试举例说明在游戏中的应用。

答：遗传算法是仿真生物演化与遗传过程的搜索与优化算法，它的理论根基源于John Holland在1975年提出的“进化论”，他是借用生物学家达尔文（Charles Darwin）所提出的进化论“物竞天择，适者生存”的概念进一步发展而来

的，就是一个解决优化问题的工具与不断改进群体适应的算法。在真实世界中，物种的演化是为了更适应大自然的环境，而在演化过程中，某个基因的改变也能让下一代来继承。例如，游戏动画中人物行走的画面通常需要事先仔细描述每个画面的细节，如果运用遗传算法把重力和人物的肌肉结构都做好关联，就可以让人物走得非常顺畅。

3.求解以下问题：

（1）使用深度优先算法求出生成树。

（2）使用广度优先算法求出生成树。

答：

（1）深度优先算法

（2）广度优先算法

4.什么是加速运动？

答：凡是物体移动时，其运动的速度或方向会随着时间而改变，那么该物体的运动便是加速运动。加速度与速度的关系如下：

$V=V_o+A\times t$

5.什么是分支限界法？

答：分支限界法是一个用途十分广泛的算法，类似于回溯法，但是求解目标却不相同，回溯法可以找出满足条件的所有解，分支限界法通常用于求出一个最优解，思路为将树节点不断分支，但随时以问题的条件限制分支的持续，这样的做法可以大幅减少所需查找的路径。