第 1 章

进入算法的世界

1

计算机（Computer）是一种具备了数据计算与信息处理功能的电子设备。它可以接受人类所设计的指令或程序设计语言，经过运算处理后输出期待的结果。

对于有志于从事信息技术专业领域的人员来说，数据结构（Data Structure）是一门与计算机硬件和软件息息相关的学科，称得上是从计算机问世以来经久不衰的热门学科。这门学科研究的重点在计算机程序设计领域，即研究如何将计算机中相关数据或信息的组合以某种方式组织起来进行有效的加工和处理，其中包含算法（Algorithm）、数据存储的结构、排序、查找、树、图及哈希函数等。

随着信息与网络科技的高速发展，在目前这个物联网（Internet of Things，IoT）与云运算（Cloud Computing）的时代，程序设计能力已经被看成是国力的象征，有条件的中小学校都将程序设计（或称为“编程”）列入学生信息课的学习内容，在大专院校里，程序设计已不再只是信息技术相关科系的“专利”了。程序设计已经是接受全民义务制教育的学生们应该具备的基本能力，只有将“创意”通过“设计过程”与计算机相结合，才能让新一代人才轻松应对这个快速变迁的云计算时代（见图 1-1）。

图 1-1

“云”其实泛指“网络”，因为工程师在网络结构示意图中通常习惯用“云朵”图来代表不同的网络。云计算是指将网络中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，就能使用这种运算资源。

物联网是近年来信息产业中一个非常热门的话题，各种配备了传感器的物品，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、自动化设备彼此沟通与交换信息，也就是通过网络把所有东西都连接在一起。

对于一个有志于投身信息技术领域的人员来说，程序设计是人机沟通的重要桥梁与应用工具，

是从 20 世纪 50 年代之后逐渐兴起的学科。从发展的眼光来看，一个国家综合的程序设计能力已经被看成是一种国力的象征，将来人才的程序设计能力已经与人才应该具有的语文、数学、英语、艺术等能力一样，是人才必备的基础能力，它主要用于培养人才解决问题、分析、归纳、创新、勇于尝试错误等方面的能力，并为胜任未来数字时代的工作做好准备，让程序设计不再是信息相关科系的专业，而是全民的基本能力（见图 1-2）。

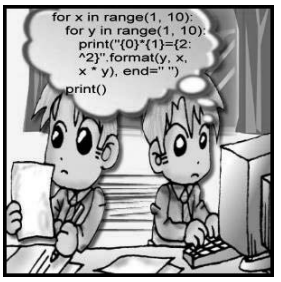


图 1-2

程序设计的本质是数学，而且是一门应用数学，过去对于程序设计的目标基本上就是为了数学的“计算”能力。随着信息与网络科技的高速发展，纯计算能力的重要性已慢慢降低，程序设计课程的目的更加着重于计算思维（Computational Thinking，CT）的训练。计算思维与当代计算机强大的执行效率相结合，让我们不断提升解决问题的能力与不断扩大解决问题的范围，因此在程序设计课程中引导学生建立计算思维（也就是分析与分解问题的能力）是为人工智能（Artificial Intelligence，AI）时代培养人才的必然。

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的能力。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技术。简单地说，人工智能就是由计算机所仿真或执行的具有类似人类智慧或思考的行为，如推理、规划、解决问题及学习等。

1.1　计 算 思 维

计算思维是一种使用计算机的逻辑来解决问题的思维，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。当前许多国家和地区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，并不限定于特定场所或工具，日常生活中任何涉及“解决问题”的议题，都可以应用计算思维来解决，通过边学边体会来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，接下来就可以按照计划出发了。简单来说，这种计划与考虑的过程就是计算思维，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简单的方式来解决，这样就具备了将问题程序化的能力。

我们可以这样来说：“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。”程序设计语言本来就只是工具，从来都不是重点，没有最好的程序设计语言，只有是否适合的程序设计语言，学习程序设计的目标不是把每个学习者都培养成专业的程序设计人员，而是帮助每一个人建立起系统化的逻辑思维模式和习惯。

2006 年，美国卡耐基梅隆大学（Carnegie Mellon University）的 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（Computational Thinking for Educators），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。虽然这并不是建立计算思维的唯一方法，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，通过不断使用计算方法与工具去解决问题，进而逐渐养成计算思维习惯。

在训练计算思维的过程中，培养学习者在现有资源上从不同角度去解决问题的能力，以及正确地运用培养计算思维的这 4 部分、运用现有的知识或工具找出解决问题的方法。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 1-3 所示。

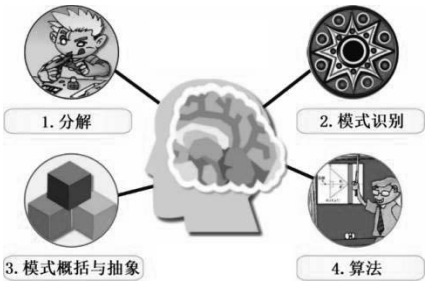


图 1-3

1.1.1　分解

许多人在编写程序或解决问题时将问题想得太复杂，如果不进行有效分解，就会很难处理。其实可以先将一个复杂的问题分割成许多小问题，再把这些小问题各个击破，之后原本的大问题也就迎刃而解了。

如果我们随身携带的智能手机出现故障了，就可以将整部手机拆解成较小的部分（部件），而后对各个部件进行检查，找出有问题的部件（见图 1-4）。

图 1-4

1.1.2　模式识别

在将一个复杂的问题分解之后，我们常常会发现这些分解后的小问题有一些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。假如我们想要画一只猫，首先就会想到猫咪通常有哪些特征，比如眼睛、尾巴、毛发、叫声、胡须等。当我们知道大部分的猫都有这些特征后，在想要画猫的时候便可以加入这些共有的特征，这样很快就可以画出很多五花八门的猫了（见图 1-5）。

图 1-5

知名的谷歌大脑（Google Brain）工具能够利用人工智能技术从庞大的猫图片库中自行识别出猫脸与人脸的不同（见图 1-6），其原理就是把所有图片内猫的“特征”提取出来，从训练数据中提取出数据的特征，同时进行“模式”分类，模拟识别特征中复杂的非线性关系来获得更好的识别能力。

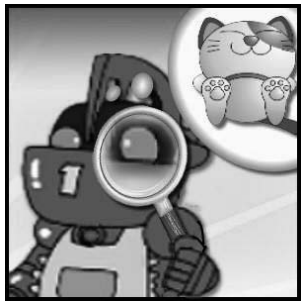


图 1-6

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，进而建立模型，目的是希望能够从原始特征数据集合中学习到问题的结构与本质。通常这个过程开始会收集许多数据，通过模式概括与抽象把无助于解决问题的特征和模式去掉，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，比如车行的业务员与修车技师对汽车抽象化的结果就会有所差异（见图 1-7）。

- 车行业务员：车轮、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。

图 1-7

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方法之一，也是程序设计的精髓。算法常出现在规划和程序设计的第一步，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。

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；通过大数据分析，网店还能进一步了解购买产品和需求产品的人群是哪类人。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 1-8 所示。

图 1-8

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。大数据是指无法使用普通的常用软件在可容忍时间内进行提取、管理及处理的大量数据，可以这么简单理解：大数据其实是巨大的数据库加上处理方法的一个总称，是一套有助于企业组织大量搜集、分析各种数据的解决方案。另外，数据的来源有非常多的途径，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。

1.2　计算思维的脑力大赛

国际计算思维挑战赛（简称 Bebras）是一项信息学领域为推动计算思维教育创办的国际赛事，由非营利性的国际组织主办。接下来我们根据这一赛事历年出题的重点设计一些生动有趣、富有挑战的计算思维模拟试题，希望通过本节能让读者清楚计算思维的训练重点，同时在进入算法学习之前让自己的大脑进行各种计算思维解题的脑力热身训练。

1.2.1　三分球比赛灯记录器

在一项高中杯篮球的三分球比赛中，看谁能在指定时间内投入 15 个三分球，当选手投入 15 个三分球后停止投球，并拿到神射手的头衔。所有选手投入的三分球个数介于 0~15 之间。为了展现三分球投入的总数，主办单位使用特殊的灯来显示当前的得分情况，灯的显示规则说明如下：

最下方的灯亮了就代表投入 1 个三分球，由下往上数的第 2 个灯亮了就代表投入 2 个三分球，由下往上数的第 3 个灯亮了就代表投入 4 个三分球，最上方的灯亮了就代表投入 8 个三分球，如图 1-9 所示。

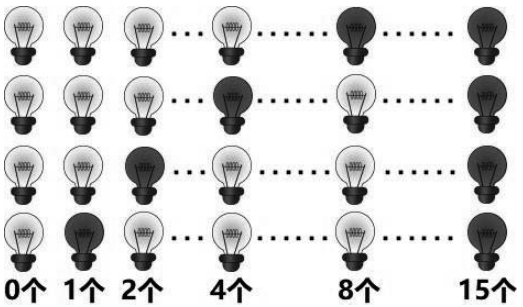


图 1-9

请问图 1-10 中的哪一组灯代表投入 13 个三分球？

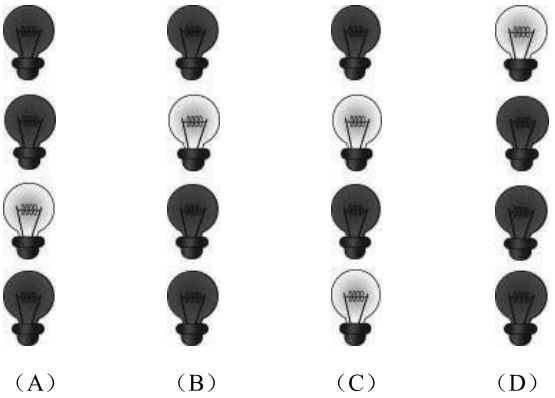
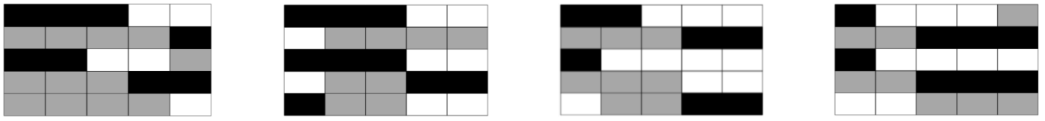


图 1-10

答案：A。

1.2.2　图像字符串编码

假设图像由许多小方格组成，并且每个小方格中只有一种颜色，整个图像只有三种颜色：黑色（Black）、白色（White）和灰色（Gray）。当图像经过编码后会形成一串英文字母与数字交互组成的字符串（String），对于每一个由英文字母与数字所组成的单元，其中的数字代表该颜色连续的次数，例如 B3 表示 3 个连续的黑色，W2 表示 2 个连续的白色，G5 表示 5 个连续的灰色。请问图 1-11 中的哪一张图像的编码字符串为“B3W2G4B3W2G4B2G4W1”？



（A）　（B）　（C）　（D）

图 1-11

答案：A。

1.2.3　计算机绘图指令实践

阿灿从计算机绘图课中学到了 7 条指令，每条指令的功能如下：

- BT——画出大三角形。
- ST——画出小三角形。
- BC——画出大圆形。
- SC——画出小圆形。
- BR——画出大矩形。
- SR——画出小矩形。
- Repeat (a1 a2 a3 …) b ——重复括号内所有指令 b 次，例如 Repeat (SC) 2 表示连续画出两个小圆形。

绘图软件会根据指令自动配色，每画出一个图形后自动换行，也就是说，一行中不会出现两个以上的图形。例如执行以下指令：

```
BC ST Repeat(SC SR)2 BT
```

该绘图软件在随机配色后会画出如图 1-12 所示的图形。

阿灿在练习时画出了如图 1-13 所示的图形，请问他使用了哪条指令？

图 1-12

图 1-13

（A）BT Repeat (BC SR)2 BR BC　　（B）BT Repeat (BC SR)2 BC BR
（C）BR Repeat (BC SR)2 BC BR　　（D）BC Repeat (BC SR)2 BC BT

答案：B。

1.2.4　炸弹超人游戏

在一款《新无敌炸弹超人》游戏中有 4 个玩家在不同的位置，其周围放置了炸弹（见图 1-14），请问哪一个玩家引爆炸弹的概率最高？试说明原因。

（A）第 2 行第 2 列的男玩家　　（B）第 2 行第 4 列的女玩家
（C）第 4 行第 2 列的女玩家　　（D）第 5 行第 5 列的男玩家

答案：D。

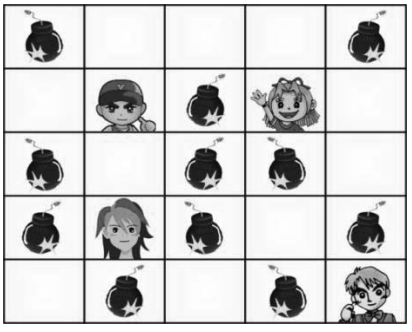


图 1-14

各选项中玩家周围的炸弹数量分别如下：

- A 选项的男玩家周围的炸弹数量为 4 个，概率为 4/8。
- B 选项的女玩家周围的炸弹数量为 4 个，概率为 4/8。
- C 选项的女玩家周围的炸弹数量为 5 个，概率为 5/8。
- D 选项的男玩家周围的炸弹数量为 2 个，概率为 2/3。

1.3　生活中处处都存在算法

算法是计算机科学中程序设计领域的核心理论之一，每个人每天都会用到一些算法。算法也是人类使用计算机解决问题的技巧之一，不但可用于计算机领域，而且在数学、物理甚至是每天的生活中都应用广泛。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，还有我们每天都在使用的各种搜索引擎也必须借助不断更新的算法来运行，如图 1-15 所示。

图 1-15

在韦氏辞典中算法定义为：A procedure for solving a mathematical problem in a finite number of steps，即“在有限步骤内解决数学问题的过程。”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，所需要有限数量的机械性或重复性指令与计算步骤。”

1.3.1　算法的条件

在计算机系统中算法更是不可或缺的一环，有一个著名的公式“计算机程序 = 算法 + 数据结构”，它从另一个角度阐述算法的概念与定义，也表述了算法、数据结构和计算机程序之间的关系。在了解了算法的定义之后，下面来说明一下算法所必须符合的 5 个条件，如图 1-16 和表 1-1 所示。

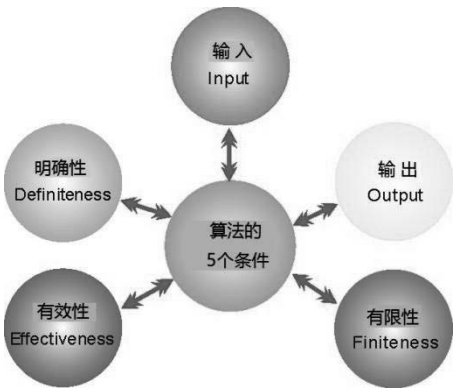


图 1-16

表1-1　算法必须符合的5个条件

算法的特性	内容与说明
输入（Input）	0 个或多个输入数据，这些输入必须有清楚的描述或定义
输出（Output）	至少会有一个输出结果，不能没有输出结果
明确性（Definiteness）	每一个指令或步骤必须是简洁明确的
有限性（Finiteness）	在有限步骤后一定会结束，不会产生无限循环
有效性（Effectiveness）	步骤清晰且可行，能让用户用纸笔计算而求出答案

我们了解了算法的定义与条件后，接着要思考一下用什么方法来表达算法比较合适。其实算法的主要目的在于让人们了解所执行工作的流程与步骤，只要清楚地体现出算法的 5 个条件即可。

常用的算法一般可以用中文、英文、数字等文字方式来描述，也就是用自然语言来描述算法的具体步骤。例如，图 1-17 所示就是小华早上去上学并买早餐的简单文字算法。

图 1-17

常用的算法也可以用可读性高的高级程序设计语言或伪语言（Pseudo-Language）来描述或者表达。以下算法是用 C 语言描述的，给 Pow()函数传入两个数 x、y，求 x 的 y 次方的值，即求 x^y 的值：

```
float Pow( float x, int y )
{
    float p = 1;
    int i;
    for( i = 1; i <= y; i++ )
        p *= x;

    return p;
}

int main(void)
{
    float x;
    int y;

    printf( "请输入次方运算（ex.2^3）：" );
    scanf( "%f^%d", &x, &y );
    printf( "次方运算结果：%.4f\n", Pow(x, y) );
    /* 调用 Pow()函数，并输出计算结果 */
}
```

伪语言是接近高级程序设计的语言，也是一种不能直接放入计算机中执行的语言。一般需要一种特定的预处理器（Preprocessor），或者用人工编写转换成真正的计算机语言，经常使用的有 SPARKS、PASCAL-LIKE 等。

流程图（Flow Diagram）是一种以图形符号来表示算法的通用方法。例如，输入一个数值，并判断是奇数还是偶数，如图 1-18 所示。

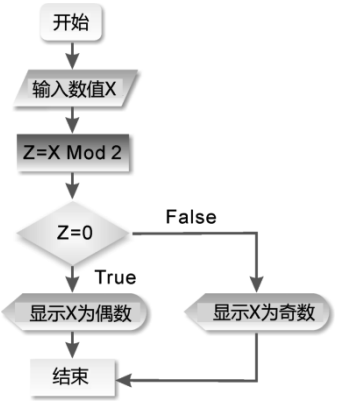


图 1-18

算法和过程（Procedure）有何不同？与流程图又有什么关系？

算法和过程是有所区别的，因为过程不一定要满足有限性的要求，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）。这也违反了算法 5 个条件中的“有限性”。另外，只要是算法，就都能够使用流程图来表示，但是由于过程流程图可包含无限循环，因此无法使用算法来表达。

以图形方式也可以表示算法，如数组、树形图、矩阵图等。图 1-19 就是用图形描述算法的一个例子。

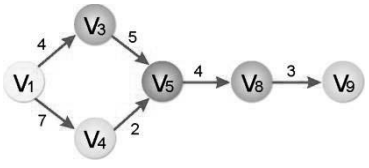


图 1-19

1.3.2　时间复杂度 $O(f(n))$

读者可能会想，应该怎么评估一个算法的好坏呢？例如，可以把某个算法执行步骤的计数来作为衡量运行时间的标准，程序语句如下：

```
a = a + 1
```

与

```
a = a + 0.3 / 0.7 * 10005
```

由于涉及变量存储类型与表达式的复杂度，因此绝对精确的运行时间一定不相同。不过如此大费周章地去考虑程序的运行时间往往会寸步难行，而且毫无意义，此时可以利用一种“概量”的概念来衡量运行时间，我们称之为时间复杂度（Time Complexity）。其详细定义如下：

在一个完全理想状态下的计算机中，我们定义 $T(n)$来表示程序执行所要花费的时间，其中 n 代表数据输入量。当然程序的运行时间（Worse Case Executing Time）或最大运行时间是时间复杂度的衡量标准，一般以 Big-Oh 表示。

在分析算法的时间复杂度时，往往用函数来表示它的成长率（Rate of Growth），其实时间复杂度是一种渐近表示法（Asymptotic Notation）。

$O(f(n))$可视为某算法在计算机中所需运行时间不会超过某一常数倍的 $f(n)$。也就是说，当某算法的运行时间 $T(n)$的时间复杂度为 $O(f(n))$（读成 Big-oh of $f(n)$或 order is $f(n)$）时，意思是存在两个常数 c 与 n_0，若 $n \geqslant n_0$，则 $T(n) \leqslant cf(n)$。$f(n)$又称为运行时间的成长率。由于在估算算法复杂度时采取“宁可高估不要低估”的原则，因此估计出来的复杂度是算法真正所需运行时间的上限。参看以下范例，以了解时间复杂度的意义。

范例▶ 假如运行时间 $T(n)=3n^3+2n^2+5n$，求时间复杂度。

解答▶ 首先找出常数 c 与 n_0。当 $n_0=0$、$c=10$ 时，若 $n \geqslant n_0$，则 $3n^3+2n^2+5n \leqslant 10n^3$，因此得知时间复杂度为 $O(n^3)$。

事实上，时间复杂度只是执行次数的一个概略的量度，并非真实的执行次数。而 Big-Oh 则是一种用来表示最坏运行时间的表现方式，也是最常用于描述时间复杂度的渐近式表示法。常见的 Big-Oh 可参考表 1-2 和图 1-20。

表 1-2　常见的 Big-Oh

Big-Oh	特色与说明
$O(1)$	称为常数时间（Constant Time），表示算法的运行时间是一个常数
$O(n)$	称为线性时间（Linear Time），表示执行的时间会随着数据集合的大小而线性增长
$O(\log_2 n)$	称为次线性时间（Sub-Linear Time），成长速度比线性时间还慢，而比常数时间还快
$O(n^2)$	称为平方时间（Quadratic Time），算法的运行时间会成二次方的增长
$O(n^3)$	称为立方时间（Cubic Time），算法的运行时间会成三次方的增长
$O(2^n)$	称为指数时间（Exponential Time），算法的运行时间会成 2 的 n 次方增长。例如，解决 Nonpolynomial Problem（非多项问题）算法的时间复杂度为 $O(2^n)$
$O(n\log_2 n)$	称为线性乘对数时间，介于线性和二次方增长的中间模式

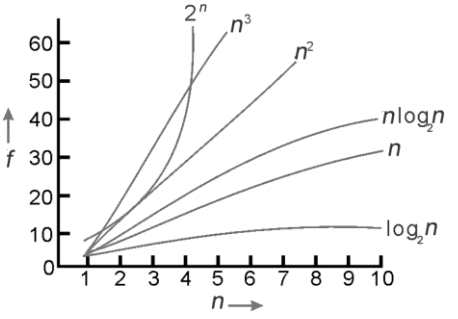


图 1-20

当 $n \geqslant 16$ 时，时间复杂度的优劣比较关系如下：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$$

1.4　课 后 习 题

1. 以下 C 程序片段是否相当严谨地表达出算法的含义？

```
count＝0;
while(count < > 3)
```

2. 以下程序的 Big-Oh 是什么？

```
total=0;
for(i=1; i<=n ; i++)
   total=total+i*i;
```

3. 算法必须符合哪 5 个条件？
4. 在下列程序的循环部分中，实际执行的次数与时间复杂度是什么？

```
for i=1 to n
   for j=i to n
      for k =j to n
         { end of k Loop }
   { end of j Loop }
{ end of i Loop }
```

5. 试证明 $f(n) = a_m n^m + \cdots + a_1 n + a_0$，则 $f(n) = O(n^m)$。
6. 下面的程序片段执行后，其中程序语句 sum=sum+1 被执行的次数是多少？

```
sum=0;
for(i=-5;i<=100;i=i+7)
   sum=sum+1;
```

7. 请问计算思维课程包含哪几个部分？

第 2 章

经典算法介绍

2

算法可以说就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如何在各层面影响我们的日常生活。善用算法是培养程序设计逻辑很重要的步骤，许多实际的问题都可以用多个可行的算法来解决，但要从中找出最佳的解决算法则是一项挑战。本章将介绍一些近年来相当知名的算法，帮助读者了解不同算法的概念与技巧，以便可以分析各种算法的优劣。

2.1 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，核心思想就是将一个难以直接解决的大问题依照相同的概念分割成两个或更多的子问题，以便各个击破。

下面以一个实际的例子来进行说明：假设有 8 幅很难画的画，我们可以分成两组（每组各 4 幅画）来完成；如果还是觉得复杂，就再分成 4 组（每组各 2 幅画）来完成，即采用相同模式反复分割问题，这就是分治法的核心思想，如图 2-1 所示。

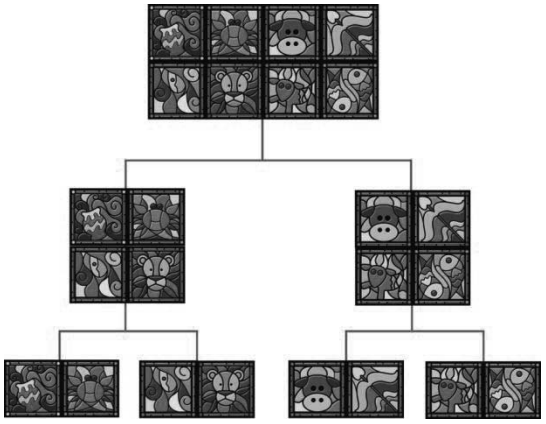


图 2-1

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和复杂度有关，问题的规模越

小，越容易直接求解。因此，可以不断分解问题，使子问题规模不断缩小，让这些子问题简单到可以直接解决，再将各个子问题的解合并，最后得到原问题的解答。再举个例子，规划一个有 8 个章节主题的项目，如果只靠一个人独立完成，不但时间比较长，而且有些规划的内容可能不是那个人的专长，这时就可以按照这 8 个章节的特性分给 2 个项目负责人去完成。为了让这个规划更快完成，并能找到适合的分类，可以再分别分割成 2 部分，并分派给不同的项目成员，如此一来，每个成员只需负责其中 2 个章节，经过这样的分配就可以将原先的大项目简化成 4 个小项目，并委派给 4 个成员去完成。根据分治法的核心思想，还可以将其分割成 8 个小主题，委派给 8 个成员去分别完成，因为参与人员较多，所以所需时间缩减为原先一个人独立完成的 1/8。这个例子的分治法解决方案的示意图如图 2-2 所示。

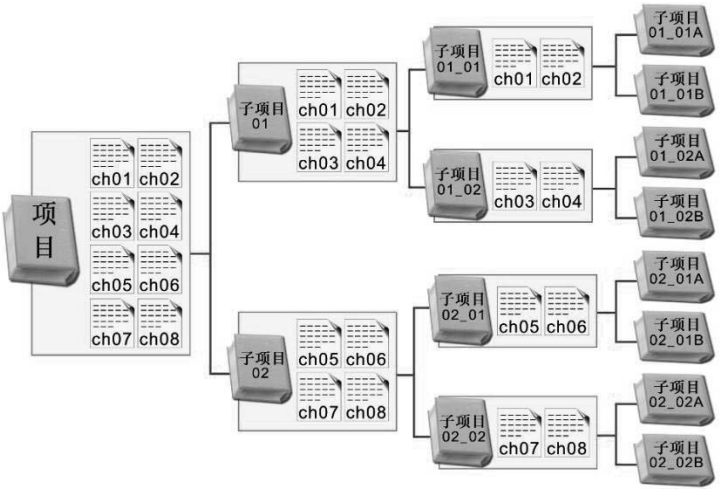


图 2-2

分治法也可以应用在数字的分类与排序上，如果要以人工的方式将散落在地上的打印稿按从第 1 页整理并排序到第 100 页，可以有两种做法，一种方法是逐一捡起打印稿，并逐一按页码顺序插入到正确的位置。但这样的方法的缺点是排序和整理的过程较为繁杂，且比较浪费时间。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，页码 11 到页码 20 放在一起，以此类推，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，就可以轻松恢复到原先的稿件顺序。通过分治法可以让原先复杂的问题变成规则更简单、数量更少、速度更快且更容易解决的小问题。

2.2　递　归　法

递归法（Recursive Method）和分治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。递归法是一种很特殊的算法，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，在某些程序设计语言中，函数或子程序不只是能够被其他函数调用或引用，还可以自己调用自己，这种调用的功能就是所谓的“递归”。

从程序设计语言的角度来说，可以这样描述递归：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。它至少要定义两个条件，一个可以反复执行的递归过程与一个跳出执行过程的出口。

尾递归（Tail Recursion）是指函数或子程序的最后一条语句为递归调用，因为每次调用后再回到前一次调用的第一条语句是 return 语句，所以不需要再进行任何运算了。

对递归法而言，阶乘是很典型的范例，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：

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

下面逐步分解它的运算过程，以观察其规律。

```
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
```

用 C 语言编写的 *n*!递归函数算法如下，请注意其中所应用的递归基本条件：一个反复的过程；一个递归终止的条件，确保有跳出递归过程的出口。

```
int factorial(int i)
{
    int sum;
    if(i == 0)  /* 递归终止的条件，跳出递归过程的出口 */
        return(1);
    else
        sum = i * factorial(i-1);  /* sum=n*(n-1)!，反复执行的递归过程 */
    return sum;
}
```

以上是用阶乘函数的范例来说明递归的运行方式，在系统中具体实现递归时，则要用到堆栈的数据结构。所谓堆栈（Stack），就是一组相同数据类型的集合，所有的操作均在这个结构的顶端进行，具有后进先出（Last In First Out，LIFO）的特性。有关堆栈的详细功能说明与实现，请参考后面章节。

我们再来看著名的斐波那契数列（Fibonacci Polynomial）的递归法求解。斐波那契数列的基本定义为：

$$F_n = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F_{n-1}+F_{n-2} & n=2,3,4,5,6\cdots（n\text{ 为正整数}） \end{cases}$$

简单来说，这个数列的第 0 项是 0，第 1 项是 1，之后各项的值是由其前面两项值相加的结果

（后面每项的值都是其前两项值的和）。根据斐波那契数列的定义，可以尝试把它设计成递归形式：

```
int fib(int n)
{
   if(n==0) return 0;
   if(n==1)
      return 1;
   else
      return fib(n-1) + fib(n-2);/*递归调用自身 2 次*/
}
```

【范例程序：CH02_01.cpp】

下面设计一个计算第 *n* 项斐波那契数列的递归程序。

```
/*
[示范] 斐波那契数列的递归程序
*/
#include<iostream>
using namespace std;

int fib(int);              //fib()函数的原型声明

int main()
{
   int i,n;
   cout<<"请输入要计算到第几项斐波那契数列：";
   cin>>n;
   for(i=0;i<=n;i++)       //计算前 n 项斐波那契数列
        cout<<"fib("<<i<<")="<<fib(i)<<endl;
   return 0;
}

int fib(int n)             //定义函数 fib()
{

   if (n==0)
      return 0;            //如果 n=0，则返回 0
   else if(n==1 || n==2)   //如果 n=1 或 n=2，则返回 1
      return 1;
   else                    //否则返回 fib(n-1)+fib(n-2)
      return (fib(n-1)+fib(n-2));
}
```

【执行结果】参考图 2-3。

```
请输入要计算到第几项斐波那契数列：10
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
fib(10)=55

--------------------------------
Process exited after 2.763 seconds with return value 0
请按任意键继续. . .
```

图 2-3

2.3 贪　心　法

贪心法（Greed Method）又称为贪婪算法，该算法是从某一起点开始，在每一个解决问题的步骤中采用贪心原则，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，算法停止，以尽可能快的方法求得更好的解。

图 2-4

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。贪心法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（Huffman Coding）、机器学习等方面。许多公共运输系统也会用到最短路径的理论，如图 2-4 所示。

哈夫曼编码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。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，哈夫曼树正好可以解决数据的大小问题。

我们来看一个简单的例子（例子中的货币系统不是现实的情况，只是为了举例）。假设我们去超市购买几罐可乐（见图 2-5），要价 24 元，我们付给售货员 100 元，希望不要找太多纸币，即纸币的总数量最少，该如何找钱呢？假设目前的纸币金额有 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 元的纸币一张，接下来选择 10 元的纸币两张，最后选择 5 元的纸币和 1 元的纸币各一张，总共 5 张纸币，这个结果也确实是最优的解。

贪心法也适合用于某些旅游景点路线的判断，假如我们要从图 2-6 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，接着走到顶点 2，最后从顶点 2 走到顶点 3，这样的距离是 28。可是从图 2-6 中我们发现直接从顶点 5 走到顶点 3 才是最短的距离（距离为 20），

说明在这种情况下，没有办法以贪心法的规则来找到最优的解。

图 2-5

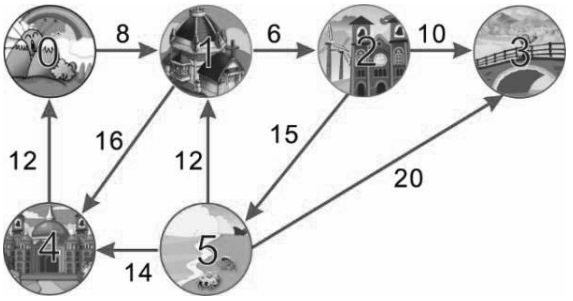


图 2-6

2.4　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，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将每一个子问题的答案存储起来，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，故而可以解决重复计算的问题。

例如，斐波那契数列采用的是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，这样就可以提高性能。若想求斐波那契数列的第 4 项数 Fib(4)，则它的递归过程可以用图 2-7 表示。

从图中可知递归调用了 9 次，而加法运算了 4 次，Fib(1)执行了 3 次，Fib(0)执行了 2 次，重复计算则影响了执行性能。根据动态规划法的算法思路可以绘制出如图 2-8 所示的执行示意图。

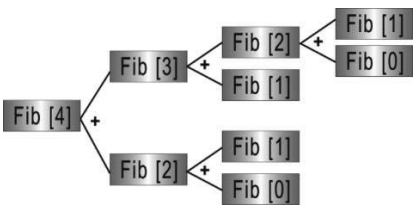


图 2-7

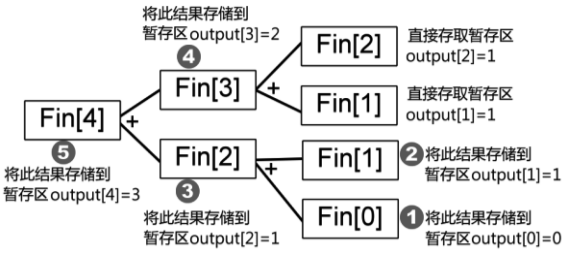


图 2-8

前面提到动态规划法的优点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为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（设置为 0 即可），当该斐波那契数列中的项被计算后，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 Fib(0)记录到 output[0]中，Fib(1)记录到 output[1]中，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 Fib(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 Fib(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 Fib(2)，按照斐波那契数列的定义，得到数值为 Fib(1)+ Fib(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 Fib(3)，按照斐波那契数列的定义，得到数值为 Fib(2)+ Fib(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 Fib(4)，按照斐波那契数列的定义，得到数值为 Fib(3)+ Fib(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 Fib(5)，按照斐波那契数列的定义，得到数值为 Fib(4)+ Fib(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。

根据上面动态规划法改进斐波那契数列的递归算法，用 C++语言实现改进的算法，程序代码如下：

```
int fib(int);         // fib()函数的原型声明
int output[1000];     // Fibonacci 的暂存区
int main()
{
   int n;
   cout<<"请输入所要计算第几个斐式数列：";
   cin>>n;
   cout<<"fib("<<n<<")="<<fib(n);
   return 0;
}

int fib(int n) // 以动态规划法定义函数
{
   int i;
   if(n==0)
      return 0;
   if(n==1)
      return 1;
   else
   {
       output[0]=0;
```

```
            output[1]=1;
            for(i=2;i<=n;i++)
                 output[i]=output[i-1]+output[i-2];
            return output[n];
        }
    }
```

2.5　迭　代　法

迭代法（Iterative Method）无法使用公式一次求解，而需要使用迭代。

【范例程序：CH02_02.cpp】

下面以 C++用 for 循环设计一个计算 1!~*n*!的阶乘程序。

```
01  // 计算 10! 的值
02  #include <iostream>
03  #include <cstdlib>
04  using namespace std;
05
06  int main()
07  {
08      int i, sum=1;
09      for (i=1;i<=10;i++)              // 定义 for 循环
10      {
11          sum*=i;                      // sum=sum+i
12      }
13      cout<<i-1<<"!="<<sum<<endl;      // 打印出 i 和 sum 的值
14      return 0;
15  }
```

【执行结果】参考图 2-9。

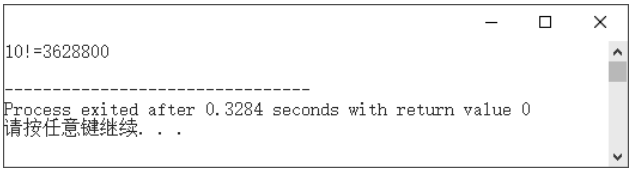


图 2-9

上述例子采用的是一种固定执行次数的迭代法，当遇到问题，无法用公式一次求解，又无法确定要执行多少次，此时就可以使用 while 循环。

while 循环必须加入控制变量的起始值及递增或递减表达式，并且在编写循环过程时必须检查离开循环体的条件是否存在，如果条件不存在，则会让循环体一直执行而无法停止，导致“无限循环”（即死循环）。循环结构通常需要具备以下 3 个条件：

（1）循环控制变量的初始值。

（2）循环条件判断表达式。

（3）调整循环控制变量的增减值。

程序如下：

```
int i=0,sum=0;
while(i<10)
{
    i++;                /* 执行循环一次则加 1，控制循环的条件变量 */
    sum=i+sum;
}
cout<<i<<"!"<<"="<<sum;
```

当 i 小于 10 时会执行 while 循环体内的语句，所以 i 会加 1，直到 i 等于 10，这时条件判断表达式为 false 了，就会终止循环。

帕斯卡三角形算法

帕斯卡（Pascal）三角形算法基本上就是计算出三角形每一个位置的数值。在帕斯卡三角形上的每一个数字都对应一个 ${}_rC_n$，其中 r 代表行（row），而 n 代表列（column），r 和 n 都是从数字 0 开始的。帕斯卡三角形算法的定义如下：

$$
\begin{array}{c}
{}_0C_0 \\
{}_1C_0\ {}_1C_1 \\
{}_2C_0\ {}_2C_1\ {}_2C_2 \\
{}_3C_0\ {}_3C_1\ {}_3C_2\ {}_3C_3 \\
{}_4C_0\ {}_4C_1\ {}_4C_2\ {}_4C_3\ {}_4C_4
\end{array}
$$

帕斯卡三角形对应的数据如图 2-10 所示。

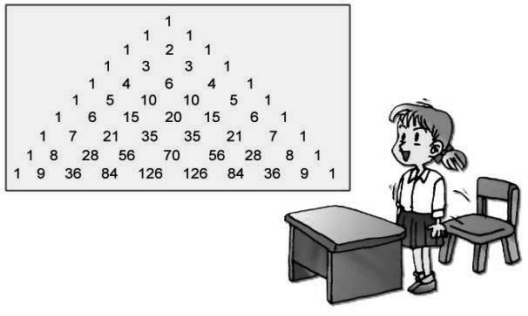


图 2-10

帕斯卡三角形的 ${}_rC_n$ 计算公式如下：

$$
\begin{aligned}
{}_rC_0 &= 1 \\
{}_rC_n &= {}_rC_{n-1} \times (r - n + 1) / n
\end{aligned}
$$

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。一旦每一行的第 0 列元素的值为数字 1 确定后，该行每一列的元素值就都可以

从同一行前一列的值根据下面的公式计算得到：

$$_rC_n = {}_rC_{n-1} \times (r - n + 1) / n$$

举例来说：

（1）第 0 行帕斯卡三角形的求值过程：当 $r = 0$、$n = 0$ 时，即第 0 行第 0 列所对应的数字为 1。

此时的帕斯卡三角形外观如下：

```
1
```

（2）第 1 行帕斯卡三角形的求值过程：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，即第 1 行第 1 列所对应的数字为 $_1C_1$，代入公式 $_rC_n = {}_rC_{n-1} \times (r - n + 1) / n$（其中 $r = 1$，$n = 1$），可以推导出 $_1C_1 = {}_1C_0 \times (1 - 1 + 1) / 1 = 1 \times 1 = 1$。得到的结果是 $_1C_1 = 1$。

此时的帕斯卡三角形外观如下：

```
   1
  1 1
```

（3）第 2 行帕斯卡三角形的求值过程：按照上面每一行中各个元素值的求值过程可以推导得出 $_2C_0$ =1、$_2C_1$ =2、$_2C_2$=1。

此时的帕斯卡三角形外观如下：

```
    1
   1 1
  1 2 1
```

（4）第 3 行帕斯卡三角形的求值过程：按照上面每一行中各个元素值的求值过程可以推导得出 $_3C_0$ =1、$_3C_1$ =3、$_3C_2$=3、$_3C_3$ =1。

此时的帕斯卡三角形外观如下：

```
    1
   1 1
  1 2 1
 1 3 3 1
```

同理，可以推导出第 4 行、第 5 行、第 6 行等所有帕斯卡三角形中各行的元素值。

2.6　枚　举　法

枚举法（又称为穷举法）是一种常见的数学方法，是我们在日常工作中使用比较多的一种算法，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、不遗漏的有限几种情况，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。像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，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 字符串中的每一个字符从第一个字符开始逐步连接到 A 字符串中的最后一个字符，如图 2-11 所示。

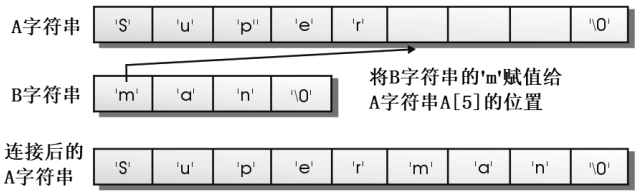


图 2-11

【范例程序：CH02_03.cpp】

下面的 C++范例程序声明两个字符串，再把它们串接起来。

```
#include <iostream>

using namespace std;

int main()
{
    char Str_1[40];
    char Str_2[40];
    char Str_3[80];
    int count, s_record;

    cout<<"字符串 Str_1 的内容：";
    cin>>Str_1;
    cout<<"字符串 Str_2 的内容：";
    cin>>Str_2;

    s_record=0;
    // 把整数变量 s_record 归 0，用来记录 Str_3 所指向的数组元素

    for (count=0; Str_1[count] != '\0'; count++, s_record++)
        // 将 Str_1 字符串复制到 Str_3
        Str_3[s_record]=Str_1[count];

    for (count=0; Str_2[count] != '\0'; count++, s_record++)
        // 将 Str_2 字符串复制到 Str_3
        Str_3[s_record]=Str_2[count];

    Str_3[s_record]='\0';
    // 字符串最后要加上 NULL 字符

    cout<<"串接后的字符串 Str_3："<<Str_3<<endl;
    // 显示字符串串接后的结果

    return 0;
}
```

【执行结果】参考图 2-12。

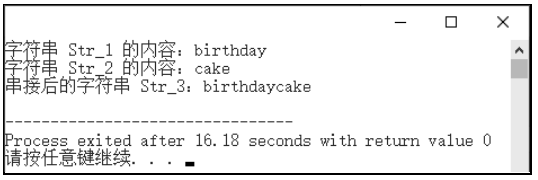


图 2-12

再来看一个例子，计算 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？这是很典型的枚举法应用，只要按序减去 1，2，3，4，5，6……即可。

$$1000-1-2-3-4-5-6-\cdots-? < 0$$

以枚举法来求解这个问题，算法过程如下：

1000–1 = 999
999–2 = 997
997–3 = 994
994–4 = 990
⋮　　⋮　　⋮
139–42 = 97
97–43 = 54
54–44 = 10
10–45 = –35

开始产生负数，根据枚举法得知，一直减到数字 45，相减的结果开始为负数。

简单来说，枚举法的核心概念就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需要的目标对象。

【范例程序：CH02_04.cpp】

下面的 C++范例程序以 while 循环来计算 1000 依次减去 1，2，3，……直到哪一个数时，相减的结果为负数。

```
01  #include <iostream>
02  #include <cstdlib>
03
04  using namespace std;
05
06  int main()
07  {
08      int x=1, sum=1000;
09
10      while(sum>0) // while 循环
11      {
12          sum-=x;
13          x++;
14      }
15      cout<<x-1<<endl;
16
```

```
17      return 0;
18   }
```

【执行结果】参考图 2-13。

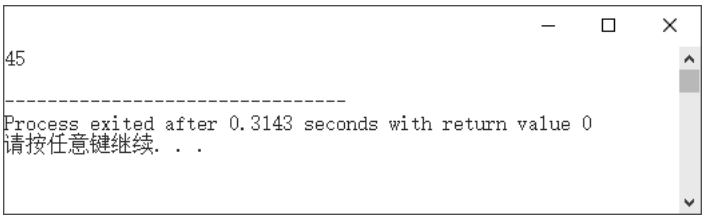
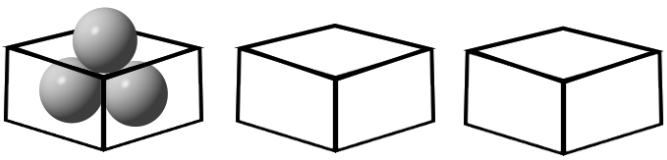


图 2-13

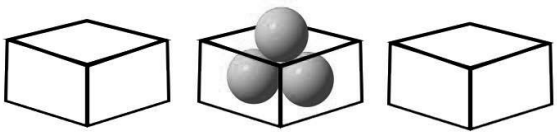
下面我们把 3 个相同的小球放入 A、B、C 三个盒子中，试问共有多少种不同的方法？分析枚举法的关键是分类，本题分类的方法有很多种，例如可以分成这样三类：第一类是 3 个球放在一个盒子里；第二类是两个球放在一个盒子里，剩余的 1 个球放在一个盒子里；第三类是将 3 个球分 3 个盒子放。

第一类：3 个球放在一个盒子里，会有 3 种可能的情况，如图 2-14~图 2-16 所示。



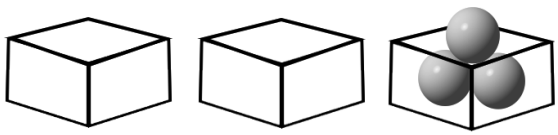
盒子 A　　盒子 B　　盒子 C

图 2-14



盒子 A　　盒子 B　　盒子 C

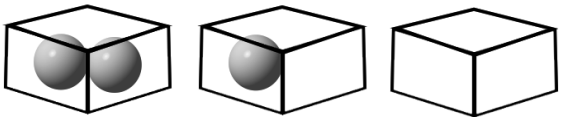
图 2-15



盒子 A　　盒子 B　　盒子 C

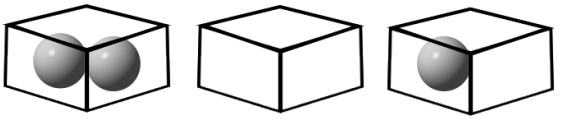
图 2-16

第二类：两个球放在一个盒子里，剩余的 1 个球放在一个盒子里，会有 6 种可能的情况，如图 2-17~图 2-22 所示。



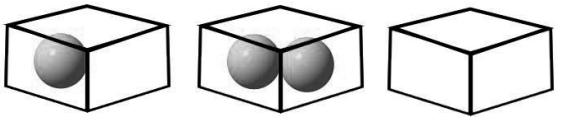
盒子 A　　盒子 B　　盒子 C

图 2-17



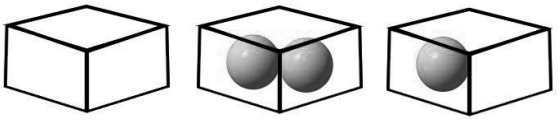
盒子 A　　盒子 B　　盒子 C

图 2-18



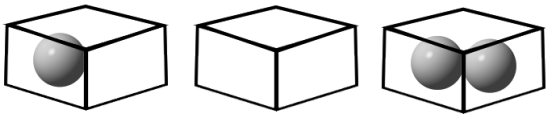
盒子 A　　盒子 B　　盒子 C

图 2-19



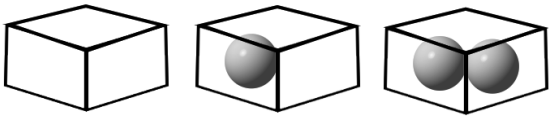
盒子 A　　盒子 B　　盒子 C

图 2-20



盒子 A　　盒子 B　　盒子 C

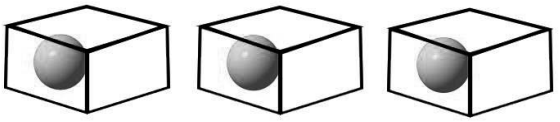
图 2-21



盒子 A　　盒子 B　　盒子 C

图 2-22

第三类：3 个球分 3 个盒子放，只有一种可能的情况，如图 2-23 所示。



盒子 A　　盒子 B　　盒子 C

图 2-23

根据枚举法的思路找出上述 10 种放置小球的方式。

质数求解算法

所谓质数（也称为素数）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 2-24 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为 N，接着参照下列步骤判断数字 N 是否为质数。在求质数的过程中，可以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。

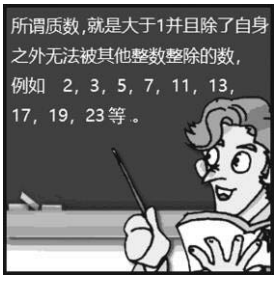


图 2-24

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？在此也一并说明。

求质数很简单，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。进一步检查会发现，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 $N = A \times B$，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 $i \times i \leq N$ 的条件判断表达式来判定要检查到哪一个整数后即可停止。

【范例程序：CH02_05.cpp】

下面的 C++范例程序通过求解来判断输入的数字 N 是否为质数。

```
01   #include <iostream>
02   #include <math.h>
03
04   using namespace std;
05
06   bool is_prime(int n)
07   {
08       int i=2;
```

```
09      while (i<=sqrt(n))
10      {
11          if(n % i == 0) // 如果可以整除，就表示 i 是 n 的因子，则返回 false
12              return false;
13          i=i+1;
14      }
15      return true;
16  }
17  int main()
18  {
19      int n;
20      cout<<"请输入一个大于或等于 2 的数字：";
21      cin>>n;
22      cout<<endl;
23      if(n==2)
24      {
25          cout<<n<<"是质数";
26          return 0;
27      }
28      if(is_prime(n))
29          cout<<n<<"是质数。";
30       else
31          cout<<n<<"不是质数。";
32
33      return 0;
34   }
```

【执行结果】参考图 2-25。

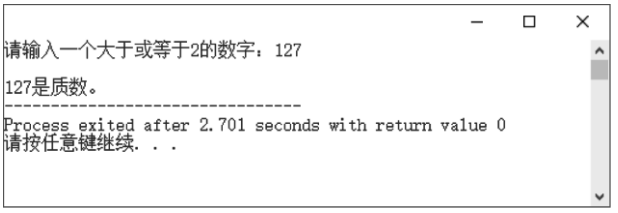


图 2-25

2.7　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，回溯法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），并尝试别的路径，避免无效搜索。

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 3 个原则：

（1）一次只能走一格。
（2）遇到墙无法往前走，则退回一步找找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——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。

图 2-26 是一个使用 10×12 二维数组表示的仿真迷宫地图。假设老鼠从左上角的 MAZE[1][1]进入，从右下角的 MAZE[8][10]出来，老鼠的当前位置用 MAZE[x][y]表示，那么老鼠可能移动的方向如图 2-27 所示。由图中可知，老鼠可以选择的方向共有 4 个，分别为东、西、南、北，但是并非每个位置都有 4 个方向可以选择，必须视情况而定。例如，T 字形的路口就只有东、西、南 3 个方向可以选择。

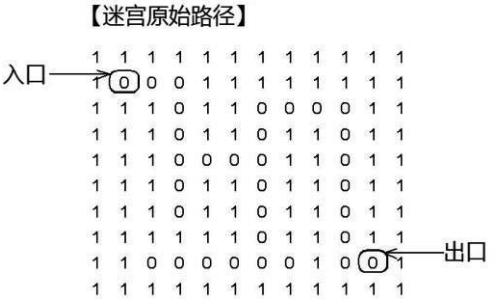


图 2-26

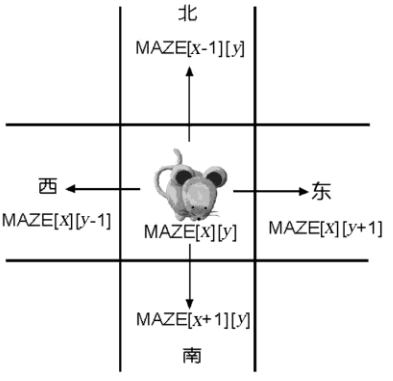


图 2-27

可以先使用链表记录走过的位置，并且将走过的位置所对应的数组元素内容标记为 2，然后将这个位置压入堆栈，再进行下一个方向或路的选择。如果走到死胡同并且没有抵达终点，就退回到上一个位置，直至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，直至走到迷宫出口为止。在图 2-28 和图 2-29 中以小球代表迷宫中的老鼠。

图 2-28

图 2-29

上面这样一个迷宫搜索的过程可以使用如下 C++语言算法来描述。

```
if(上一格可走)
{
    把方格编号压入堆栈;
    往上走;
    判断是否为出口;
}
else if(下一格可走)
{
    把方格编号压入堆栈;
    往下走;
    判断是否为出口;
}
else if(左一格可走)
{
    把方格编号压入堆栈;
    往左走;
    判断是否为出口;
}
else if(右一格可走)
{
    把方格编号压入堆栈;
    往右走;
    判断是否为出口;
}
else
{
    从堆栈删除一个方格编号;
    从堆栈中取出一个方格编号;
    往回走;
}
```

上面的算法是每次进行移动时所执行的操作，其主要是判断当前所在位置的上、下、左、右是

否有可以前进的方格，若找到可前进的方格，则将该方格的编号加入记录移动路径的堆栈中并向该方格移动；而当四周没有可走的方格（第 25 行程序语句），也就是当前所在的方格无法走出迷宫，则必须退回到前一格重新检查是否有其他可走的路径。所以在上面算法中的第 27 行会将当前所在位置的方格编号从堆栈中删除，之后第 28 行从堆栈再弹出的就是前一次所走过的方格编号。

以下是用 C++语言编写的走迷宫程序。

【范例程序：CH02_06.cpp】

设计一个 C++程序，使用链表堆栈来找出老鼠走迷宫的路线，1 表示该处有墙无法通过，0 表示[i][j]处无墙可通行，并且将走过的位置对应的数组元素内容标记为 2。

```
01  #include <iostream>
02  #define EAST  MAZE[x][y+1]   //定义东方的相对位置
03  #define WEST  MAZE[x][y-1]   //定义西方的相对位置
04  #define SOUTH MAZE[x+1][y]     //定义南方的相对位置
05  #define NORTH MAZE[x-1][y]     //定义北方的相对位置
06  using namespace std;
07  const int ExitX = 8;   //定义出口的 X 坐标在第 8 行
08  const int ExitY = 10; //定义出口的 Y 坐标在第 10 列
09  struct list
10  {
11     int x,y;
12     struct list* next;
13  };
14  typedef struct list node;
15  typedef node* link;
16  int MAZE[10][12] = {1,1,1,1,1,1,1,1,1,1,1,1, //声明迷宫数组
17                      1,0,0,0,1,1,1,1,1,1,1,1,
18                      1,1,1,0,1,1,0,0,0,0,1,1,
19                      1,1,1,0,1,1,0,1,1,0,1,1,
20                      1,1,1,0,0,0,0,1,1,0,1,1,
21                      1,1,1,0,1,1,0,1,1,0,1,1,
22                      1,1,1,0,1,1,0,1,1,0,1,1,
23                      1,1,1,1,1,1,0,1,1,0,1,1,
24                      1,1,0,0,0,0,0,0,1,0,0,1,
25                      1,1,1,1,1,1,1,1,1,1,1,1};
26  link push(link stack,int x,int y);
27  link pop(link stack,int* x,int* y);
28  int chkExit(int ,int ,int,int);
29  int main(void)
30  {
31     int i,j;
32     link path = NULL;
33     int x=1;  //入口的 X 坐标
34     int y=1;    //入口的 Y 坐标
35     cout<<"[迷宫的路径(0 标记的部分)]\n"<<endl; //打印出迷宫的路径图
36     for(i=0;i<10;i++)
37     {
38        for(j=0;j<12;j++)
39           cout<<MAZE[i][j]<<" ";
40        cout<<endl;
41     }
42     while(x<=ExitX&&y<=ExitY)
```

```
43      {
44          MAZE[x][y]=2;
45          if(NORTH==0)
46          {
47              x -= 1;
48              path=push(path,x,y);
49          }
50          else if(SOUTH==0)
51          {
52              x+=1;
53              path=push(path,x,y);
54          }
55          else if(WEST==0)
56          {
57              y-=1;
58              path=push(path,x,y);
59          }
60          else if(EAST==0)
61          {
62              y+=1;
63              path=push(path,x,y);
64          }
65          else if(chkExit(x,y,ExitX,ExitY)==1) // 检查是否走到出口了
66              break;
67          else
68          {
69              MAZE[x][y]=2;
70              path=pop(path,&x,&y);
71          }
72      }
73      cout<<"[老鼠走过的路径(2 标记的部分)]"<<endl; // 打印出老鼠走完迷宫后的路径图
74      for(i=0;i<10;i++)
75      {
76          for(j=0;j<12;j++)
77              cout<<MAZE[i][j]<<" ";
78          cout<<endl;
79      }
80
81      return 0;
82  }
83  link push(link stack,int x,int y)
84  {
85      link newnode;
86      newnode = new node;
87      if(!newnode)
88      {
89          cout<<"Error! 内存分配失败! "<<endl;
90          return NULL;
91      }
92      newnode->x=x;
93      newnode->y=y;
94      newnode->next=stack;
95      stack=newnode;
96      return stack;
97  }
```

```
link pop(link stack,int* x,int* y)
{
    link top;
    if(stack!=NULL)
    {
        top=stack;
        stack=stack->next;
        *x=top->x;
        *y=top->y;
        delete top;
        return stack;
    }
    else
        *x=-1;
    return stack;
}
int chkExit(int x,int y,int ex,int ey)
{
    if(x==ex&&y==ey)
    {
        if(NORTH==1||SOUTH==1||WEST==1||EAST==2)
            return 1;
        if(NORTH==1||SOUTH==1||WEST==2||EAST==1)
            return 1;
        if(NORTH==1||SOUTH==2||WEST==1||EAST==1)
            return 1;
        if(NORTH==2||SOUTH==1||WEST==1||EAST==1)
            return 1;
    }
    return 0;
}
```

【执行结果】参考图 2-30。

图 2-30

2.8　课 后 习 题

1. 试简述分治法的核心思想。
2. 递归至少要定义哪两个条件？
3. 试简述贪心法的主要核心概念。
4. 简述动态规划法与分治法的差异。
5. 什么是迭代法？试简述。
6. 试简述枚举法的核心概念。
7. 试简述回溯法的核心概念。