第一章　算　法

要利用计算机处理问题，需要编写出使计算机按人们意愿工作的计算机程序。所谓程序就是一组计算机指令。每一个指令使计算机执行特定的操作。因此，每一个学习计算机知识的人以及希望利用计算机进行某项工作的人，都应当学习如何进行程序设计。

为了有效地进行程序设计，应当至少具有两个方面的知识，即：（1）掌握一门高级语言的语法规则；（2）掌握解题的方法和步骤，也就是说，在拿到一个需要求解的问题后，怎么能将它分解成一连串的操作步骤。

计算机语言只是一种工具。光学习语言的规则还不够，最重要的是学会针对各种类型的问题，拟定出有效的解题方法和步骤。这就是本章所要讨论的问题——算法。有了正确而有效的算法，可以利用任何一种计算机高级语言编写程序，使计算机进行工作。因此，设计算法是程序设计的核心。希望读者对它给予足够的注意。

本书的叙述，是以算法为核心而展开的，目的不仅在于介绍一种计算机语言，而在于使读者能利用计算机语言编写出所需解决的问题的程序。

1.1 算法的概念

做任何事情都有一定的步骤。例如，你要看电影，就要先买票，然后按时到电影院，进场，找座位，坐下，看电影，退场，等等。你要考入大学，首先要填报名单，交报名费，拿到准考证，按时参加考试，得到录取通知书，到指定学校报到注册等。这些都是按一系列的顺序进行的步骤，缺一不可，次序错了也不行。因此，我们从事各种工作和活动，都必须事先想好进行的步骤，以免产生错乱。事实上，在日常生活中，由于已养成习惯，所以人们并不意识到需要事先设计“行动步骤”。例如吃饭、上学、打球、做作业等，事实上都是按照一定的规律进行的，只是人们不必每次都重复考虑它而已。

不要认为只有“计算”的问题才有算法。广义地说，为解决一个问题而采取的方法和步骤，称为“算法”（Algorithm）。例如，描述太极拳动作的图解，就是“太极拳的算法”。一首歌曲的乐谱，也可以称为该歌曲的算法，因为它指定了演奏该歌曲的每一个步骤，按照它的规定就能演奏出预定的曲子。

对同一个问题，可以有不同的解题方法和步骤。例如，求 1＋2＋3＋…＋100，即 $\sum_{n=1}^{100} n$，有的人可能先进行 1＋2，再加 3，再加 4，一直加到 100，而有的人采取这样的方法 $\sum_{n=1}^{100} n = 100 + (1+99) + (2+98) + \cdots + (49+51) + 50 = 100 + 50 + 49 \times 100 = 5050$。还可以有其它的方法。当然，方法有优劣之分。有的方法只需进行很少的步骤，而有些方法则需要较多的步骤。一般说，希望采用方法简单，运算步骤少的方法。因此，为了有效地进行解题，不仅需要保证算法正确，还要考虑算法的质量，选择合适的算法。

要完成一件工作，包括设计算法和实现算法两个部分。例如，作曲家创作一首曲谱就是设计一个算法，但它仅仅是一个乐谱，并未变成音乐，而作曲家的目的是希望使人们听到悦耳动人的音乐。由演奏家按照乐谱的规定进行演奏，就是“实现算法”。一个菜谱是一个算法，厨师炒菜就是在实现这个算法。设计算法的目的是为了实现算法。因此，我们不仅要考虑如何设计一个算法，也要考虑如何实现一个算法。

本书所关心的当然只限于计算机算法，即计算机能执行的算法。例如，让计算机算 1×2×3×4×5，或将 100 个学生的成绩按高低分次序排列，是可以做到的，而让计算机去执行“替我理发”或“煎一份牛排”，是不可能的（至少目前如此）。

计算机算法可分为两大类别：数值运算算法和非数值运算算法。数值运算的目的是求数值解，例如求方程的根、求一个函数的定积分等，都属于数值运算范围。非数值运算包括的面十分广泛，最常见的是用于事务管理领域，例如图书检索、人事管理、行车调度管理等。计算机在非数值运算方面的应用远远超过了在数值运算方面的应用。由于数值运算有现成的模型，可以运用数值分析方法，因此对数值运算的算法的研究比较深入，算法比较成熟。对各种数值运算都有比较成熟的算法可供选用。常常把这些算法汇编成册（写成程序形式），或者将这些程序存放在磁盘或磁带上，供用户调用。例如有的计算机系统提供“数学程序库”，使用起来十分方便。而非数值运算的种类繁多，要求各异，难以规范化，因此只对一些典型的非数值运算算法（例如排序算法）作比较深入的研究。其它的非数值运算问题，往往需要使用者参考已有的类似算法重新设计解决特定问题的专门算法。

本书不可能罗列所有算法，只是通过一些典型的算法的讨论，帮助读者了解如何设计一个算法，推动读者举一反三。希望读者通过这些例子了解怎样提出问题，怎样思考问题，怎样表示一个算法。

1.2 简单算法举例

【例 1.1】 求 1×2×3×4×5。

可以用最原始的方法进行。

步骤 1：先求 1×2，得到结果 2。

步骤 2：将步骤 1 得到的结果 2 再乘以 3，得到结果 6。

步骤 3：将 6 再乘以 4，得 24。

步骤 4：将 24 再乘以 5，得 120。这就是最后的结果。

这样的算法虽然是正确的，但太繁琐。如果要求 1×2×…×100，则要写 999 个步骤，显然是不可取的。而且每次都直接使用上一步骤的数值结果（如 2，6，24 等），也不方便。应当能找到一种通用的表示方法。

可以设两个变量：一个变量代表被乘数，一个变量代表乘数。不另设变量存放乘积结果，而直接将每一步骤的乘积放在被乘数变量中。今设 T 为被乘数，I 为乘数。同时用循环来表示算法，可以将算法改写如下：

S1：使 T=1

S2：使 I=2

S3：使 T×I，乘积结果仍放在变量 T 中，可表示为：T×I⇒T

S4：使 I 的值加 1，即 I+1⇒I。

S5：如果 I 不大于 5，返回重新执行 S3，以及其后的步骤 S4，S5；否则，算法结束。最后得到 T 的值就是 5！的值。

上面的 S1，S2…代表步骤 1，步骤 2…。S 是 Step（步）的缩写。这是写算法的习惯用法。

请读者仔细分析这个算法，是否能得到预期的结果。显然这个算法比前面的一种形式的算法简练。

如果题目改为：求 1×3×5×7×9×11。

算法只需作很少的改动即可。

S1：1⇒T

S2：3⇒I

S3：T×I⇒T

S4：I+2⇒I

S5：若 I≤11，返回 S3。否则，结束。

可以看出用这种方法表示的算法具有通用性、灵活性。S3 到 S5 组成一个循环，在实现算法时要反复多次执行 S3，S4，S5 等步骤，直到某一时刻，执行 S5 步骤时经过判断，乘数 I 已超过规定的数而不返回 S3 步骤为止。此时算法结束，变量 T 的值就是所求结果。

由于计算机是高速进行运算的自动机器，实现循环是轻而易举的，所有计算机高级语言中都有实现循环的语句，因此，上述算法不仅是正确的，而且是计算机能实现的较好的算法。

请读者仔细分析循环结束的条件，即 S5 步骤。如果在求 1×2×…×11 时，将 S5 步骤写成：

S5：若 I<11，返回 S3。

这样会有什么问题？得到什么结果？

【例 1.2】 有 50 个学生，要求将他们之中成绩在 80 分以上者打印出来。

用 N 表示学生学号，N_1 代表第一个学生学号，N_i 代表第 i 个学生学号。用 G 代表学生成绩，G_i 代表第 i 个学生成绩，算法可表示如下。

S1：1⇒i

S2：如果 G_i≥80，则打印 N_i 和 G_i，否则不打印。

S3：i+1⇒i

S4：如果 i≤50，返回 S2，继续执行。否则，算法结束。

本例中，变量 i 作为下标，用它来控制序号（第几个学生，第几个成绩）。当 i 超过 50 时，表示已对 50 个学生的成绩处理完毕，算法结束。

【例 1.3】 将 2000～2500 年中每一年是否闰年打印出来。

闰年的条件是：(1) 能被 4 整除，但不能被 100 整除的年份都是闰年；(2) 能被 100 整除，又能被 400 整除的年份是闰年。如 1989，1990 年不是闰年，1992，2000 年是闰年。

设 Y 为年份，算法可表示如下：

S1：2000⇒Y

S2：若 Y 不能被 4 整除，则打印 Y“不是闰年”。然后转到 S5。

S3：若Y能被4整除，不能被100整除，则打印Y“是闰年”。然后转到S5。

S4：若Y能被100整除，又能被400整除，打印Y“是闰年”，否则打印“不是闰年”。

S5：Y+1⇒Y

S6：当Y≤2500时，转S2继续执行，如Y>2500，算法停止。

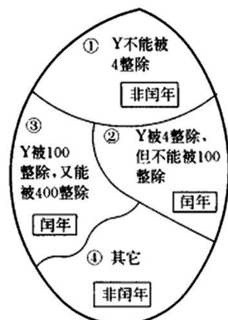


图 1.1

在这个算法中，采取了多次判断，先判断Y能否被4整除，如不能，则Y必然不是闰年。如Y能被4整除，并不能马上决定它是否闰年，还要看它能否被100整除。如不能被100整除，则肯定是闰年（例如1990年）。如能被100整除，还不能判断它是否闰年，还要被400整除，如果能被400整除，则它是闰年，否则不是闰年，在这个算法中，每做一步，都分别分离出一些年份（为闰年或非闰年），逐步缩小范围，使被判断的范围愈来愈小，直至最后，见图1.1示意。

从图1.1可以看出：“其它”这一部分，包括能被4整除，又能被100整除，而不能被400整除的那些年份（如1990）。

在考虑算法时，应当仔细分析所需判断的条件，如何一步一步缩小被判断的范围。有的问题，判断的先后次序是无所谓的，而有的问题，判断条件的先后次序是不能任意颠倒的，读者可根据具体问题决定其逻辑。

【例1.4】 求 $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}\cdots+\frac{1}{99}-\frac{1}{100}$

算法可以表示如下：

S1：sum=1

S2：deno=2

S3：sign=1

S4：sign=（-1）×sign

S5：term=sign×（1/deno）

S6：sum=sum+term

S7：deno=deno+1

S8：若deno≤100返回S4，否则算法结束。

本例中用有含义的单词作变量名，以使算法更易于理解。sum表示累加和，deno是分母denominator的缩写，sign代表数值的符号，term代表某一项，读者理解这个算法是不困难的。

【例1.5】 对一个大于或等于3的正整数，判断它是不是一个素数。

所谓素数，是指除1和该数本身之外，不能被其它任何整数整除的数。例如，13是素数。因为它不能被2，3，4，…，12整除。

判断一个数N（N≥3）是否素数的方法是很简单的：将N作为被除数，将2到（N-1）各个整数轮流作为除数，如果都不能被整除，则N为素数。

算法可以表示如下：

S1：输入 N 的值。

S2：I=2。

S3：N 被 I 除，得余数 R。

S4：如果 R=0，表示 N 能被 I 整除，则打印 N“不是素数”，算法结束。否则执行 S5。

S5：I+1⇒I

S6：如果 I⩽N-1，返回 S3。否则打印 N“是素数”。然后结束。

实际上，N 不必被 2 到（N-1）的整数除，只需被 2 到$\frac{N}{2}$间整数除即可，甚至只需被 2 到 $\sqrt{N}$之间的整数除即可。例如，判断 13 是否素数，只需将 13 被 2、3 除即可，如都除不尽，N 必为素数。S6 步骤可改为：

S6：如果 I⩽ $\sqrt{N}$，返回 S2，否则算法结束。

通过以上几个例子，可以初步了解怎样写一个算法。

1.3 算法的特性

一个算法应该具有以下特点：

1. 有穷性。一个算法应包含有限的操作步骤，而不能是无限的。例如上一节中例 1.4 的算法，如果将 S8 步骤改为：“若 deno>0 返回 S4”，则循环永远不会停止。这不是有穷的步骤。

事实上，“有穷性”往往指在合理的范围之内。如果让计算机执行一个历时 1000 年才结束的算法，这虽然是有穷的，但超过了合理的限度，人们也不把它视作有效算法。究竟什么算“合理限度”，并无严格标准，由人们的常识和需要而定。

2. 确定性。算法中的每一个步骤都应当是确定的，而不应当是含糊的，模棱两可的。例如，有一个健身操的动作要领，其中有一个动作：“手举过头顶”，这个步骤就是不确定的，含糊的。是双手都举过头？还是左手？或右手？举过头顶多少厘米不同的人可以有不同的理解。算法中的每一个步骤应当不致被解释成不同的含义，而应是十分明确无误的。如例 1.5 中的 S3 步骤如果写成“N 被一个整数除，得余数 R”，这也是“不确定”的，它没有说明 N 被哪一个整数除，因此难以执行。也就是说，算法的含义应当是唯一的，而不应当产生“歧义性”。所谓“歧义性”是指可以被理解为两种（或多种）的可能含义。

3. 有零个或多个输入。所谓输入是指在执行算法时需要从外界取得必要的信息。例如，在执行例 1.5 算法时，需要输入 N 的值，然后判断 N 是否素数。也可以有二个或多个输入，例如，求两个整数 m 和 n 的最大公约数，则需要输入 m 和 n 的值。一个算法也可以没有输入，例如，例 1.1 在执行算法时不需要输入任何信息，就能求出 5!。

不要把指定算法误认为是“输入”。输入是指在执行指定的算法时需要从外界获取的信息。

4. 有一个或多个输出。算法的目的是为了求解，“解”就是输出。如例 1.5 求素数的算法，最后打印出的“N 是素数”或“N 不是素数”就是输出的信息。但算法的输出不一定就是计算机的打印输出，一个算法得到的结果就是算法的输出。没有输出的算法是没有意义的。

5. 有效性。算法中的每一个步骤都应当能有效地执行，并得到确定的结果。例如，如果

试读结束：需要全本请在线购买： www.ertongbook.com

B=0，则执行 A/B 是无法有效执行的。

对于那些不熟悉计算机的人来说，他们可以只使用别人已设计好的现成算法，只需根据算法的要求给以必要的输入，就能得到输出的结果。对他们来说，算法如同一个“黑箱子”一样，他们可以不了解“黑箱子”中的结构，只是从外部特性上了解算法的作用，即可方便地使用算法。例如，对一个“输入三个数，求其中最大值”的算法，可以用图 1.2 表示，只要输入 a，b，c 三个数，执行算法后就能输出其中最大的数，但对于程序设计人员来说，必须会设计算法，并且根据算法编写程序。

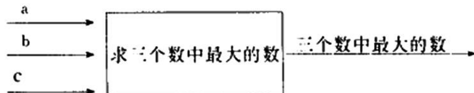


图 1.2

1.4 怎样表示一个算法

为了表示一个算法，可以用不同的方法。常用的有：自然语言；流程图；结构化流程图；伪代码；PAD 图等。

1.4.1 用自然语言表示算法

我们在 1.2 节中介绍的算法是用自然语言来表示的，自然语言就是人们日常使用的语言，可以是汉语或英语或其它文字。用自然语言表示通俗易懂，但文字冗长，容易出现“歧义性”。自然语言表示的含义往往不大严格，要根据上下文才能判断其正确含义。而且用自然语言来描述包含分支和循环的算法，不很方便（例 1.5 的算法）。因此，除了那些很简单的问题以外，一般不用自然语言描述算法。

1.4.2 用流程图表示算法

流程图是用一些图框来表示各种类型的操作。用图形表示算法，直观形象，易于理解。

美国国家标准化协会 ANSI（American National Standard Institute）规定了一些常用的流程图符号（见图 1.3），已为世界各国程序工作者普遍采用。图 1.3 中菱形框的作用是对一个给定的条件进行判断，根据给定的条件是否成立决定如何执行其后的操作。它有一个入口，二个出口。见图 1.4 示意。连接点（小圆圈）是用于将画在不同地方的流程线连接起来。如图 1.5 中有两个以 1 标志的连接点（在连接点圈中写上“1”），它表示这两个点是连接在一起的，相当于一个点一样。用连接点，可以避免流程线的交叉或过长，使流程图清晰。注释框不是流程图中必要的部分，不反

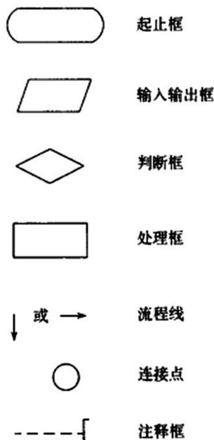


图 1.3

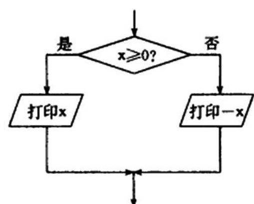


图 1.4

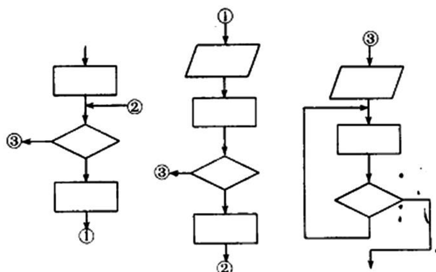


图 1.5

映流程和操作，只是为了对流程图中某些框的操作做必要的补充说明，以帮助阅读流程图的人更好地理解流程图的作用。

我们对 1.2 中所举的几个算法例子，改用流程图表示。

【例 1.6】 将例 1.1 求 5! 的算法用流程图表示，流程图见图 1.6。

如果需要将最后结果打印出来，可以在菱形框的下面再加一个输出框，见图 1.7。

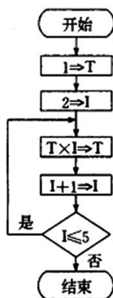


图 1.6

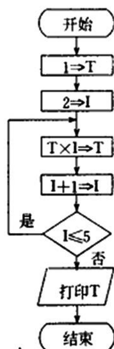


图 1.7

【例 1.7】 将例 1.2 的算法用流程图表示。将 50 名学生中成绩在 80 分以上者的学号和成绩打印出来。

见图 1.8，菱形框两侧的“Y”和“N”代表“是”（yes）和“否”（no）。在此算法中没有包括输入 50 个学生数据的部分。如果包括这个输入数据的部分，流程图如图 1.9 所示。

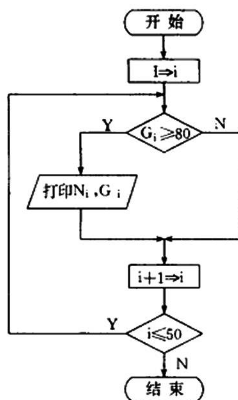


图 1.8

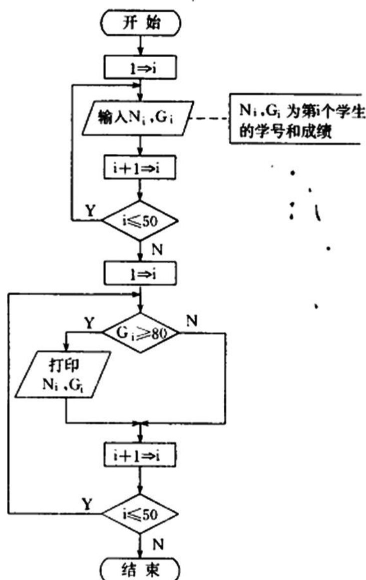


图 1.9

【例 1.8】 将例 1.3 判定闰年的算法用流程图表示。

见图 1.10，显然，用图 1.10 表示算法要比用文字描述算法逻辑清晰、易于理解。

请读者考虑，如果例 1.3 所表示的算法中，S2 步骤内没有最后“转到 S5”这一句话，而只是：

S2：若 Y 不能被 4 整除，则打印 Y“不是闰年”。

这样就意味着执行完 S2 步骤后应执行 S3 步骤，而不论 S2 的执行情况如何。请读者画出相应的流程图。请思考这样的算法在逻辑上有什么错误？从流程图上，是很容易发现逻辑上的错误的。

【例 1.9】 将例 1.4 的算法用流程图表示。即：求 $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}\cdots+\frac{1}{99}-\frac{1}{100}$

见图 1.11。

【例 1.10】 将例 1.5 判断素数的算法用流程图表示。

见图 1.12。

通过以上几个例子，可以看出流程图是表示算法的较好的工具。一个流程图包括以下几部分：(1) 表示相应操作的框；(2) 带箭头的流程线；(3) 框内外必要的文字说明。需要提醒的是：流程线不要忘记画箭头，因为它是反映流程的执行先后次序的，如不画出箭头就难以判定各框的执行次序了。

用流程图表示算法直观形象，比较清楚地显示出各个框之间的逻辑关系。前一时期国内外计算机书刊都广泛使用这种流程图表示算法。但是，这种流程图占用篇幅较多，尤其当算

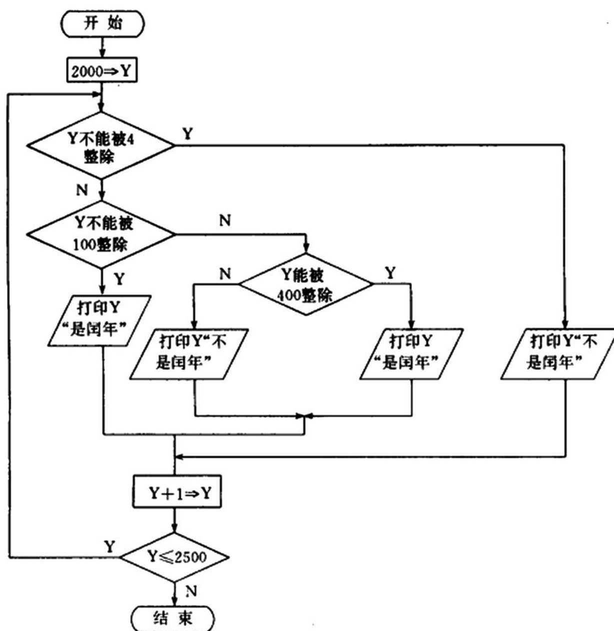


图 1.10

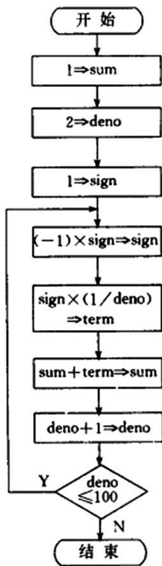


图 1.11

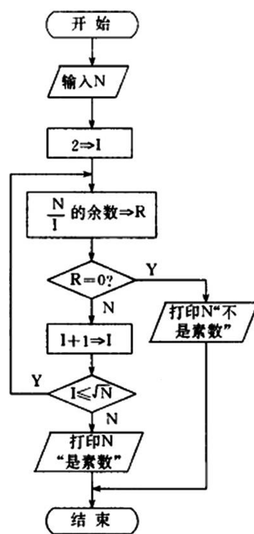


图 1.12

法比较复杂时，画流程图既费时又不方便。在结构化程序设计方法推广之后，许多书刊已用N-S结构化流程图代替这种传统的流程图。但是每一个程序编制人员都应当熟练掌握传统流程图，会看会画。

1.4.3 三种基本结构和改进的流程图

一、传统流程图的弊端

传统的流程图用流程线指出各框的执行顺序，对流程线的使用没有严格限制。因此，使用者可以毫不受限制地使流程随意地转来转去，使流程图变得毫无规律，阅读者要花很大精力去追踪流程，使人难以理解算法的逻辑。这种情况如图 1.13 所示。这种如同乱麻一样的算法称为 BS 型算法，意为一碗面条（A Bowl of Spaghetti），乱无头绪。

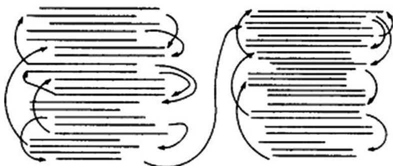


图 1.13

这种算法并不好，难以阅读，也难以修改，从而使算法的可靠性和可维护性难以保证。如果我们写出的算法能限制流程的无规律任意转向，而像一本书那样，由各章各节顺序组成，那样，阅读起来就很方便，不会有任何困难，只需从头到尾顺序地看下去即可。而如果一本书不是由各章节顺序组成，而毫无规律地乱排（例如第一章从 36 页开始到 47 页，第二章从 98 页到 107 页，第三章从 19 页到 35 页……），各章内各节也是毫无规律地乱排，阅读这种书是不会愉快的。

为了提高算法的质量，使算法的设计和阅读方便，必须限制箭头的滥用，即不允许无规律地使流程乱转向，只能顺序地进行下去。但是，算法上难免会包含一些分支和循环，而不可能全部由一个一个框顺序组成。例如图 1.6 到图 1.12 都不是由各框顺序进行的，都包含一些流程的向前或向后的非顺序转移。为了解决这个问题，人们设想，如果规定出几种基本结构，然后由这些基本结构按一定规律组成一个算法结构（如同用一些基本预制构件来搭成房屋一样），整个算法的结构是由上而下地将各个基本结构顺序排列起来的。如果能做到这一点，算法的质量就能得到保证和提高。

二、三种基本结构

1966 年，Bohra 和 Jacopini 提出了以下三种基本结构，用这三种基本结构作为表示一个良好算法的基本单元。

1. 顺序结构

如图 1.14 所示，虚线框内是一个顺序结构。其中 A 和 B 两个框是顺序执行的。顺序结构是最简单的一种基本结构。

2. 选择结构

或称选取结构，如图 1.15 所示。虚线框内是一个选择结构。此结构中必包含一个判断框。根据给定的条件 p 是否成立而选择执行 A 和 B。例如 p 条件可以是“$x\geqslant 0$”或“$x>y$”，“$x+4<t+y$”等，详见第四章。

注意，无论 p 条件是否成立，只能执行 A 或 B 之一，不可能既执行 A 又执行 B。无论走哪一条路径，在执行完 A 或 B 之后，经过 b 点，然后脱离本选择结构。A 或 B 两个框中可以

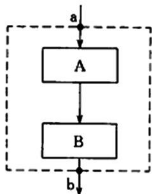


图 1.14

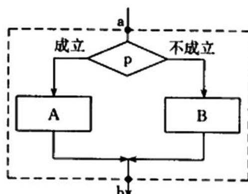


图 1.15

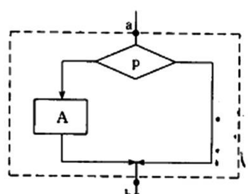


图 1.16

有一个是空的，即不执行任何操作，如图 1.16 所示。

3. 循环结构

它又称重复结构，即反复执行某一部分的操作。有两类循环结构：

(1) 当型（While 型）循环结构

见图 1.17 (a)。它的功能是：当给定的条件 p_1 成立时，执行 A 框操作，执行完 A 后，再判断 p_1 条件是否成立，如果仍然成立，再执行 A 框，如此反复执行 A 框，直到某一次 p_1 条件不成立为止，此时不执行 A 框，而从 b 点脱离循环结构。

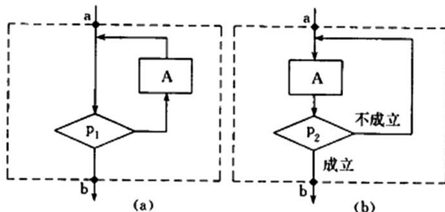


图 1.17

(2) 直到型（Until 型）循环

见图 1.17 (b)。它的功能是：先执行 A 框，然后判断给定的 p_2 条件是否成立，如果 p_2 条件不成立，则再执行 A，然后再对 p_2 条件作判断，如果 p_2 条件仍然不成立，又执行 A，…如此反复执行 A，直到给定的 p_2 条件成立为止，此时不再执行 A，从 b 点脱离本循环结构。

图 1.18 是当型循环的应用例子，图 1.19 是直到型循环的应用例子。

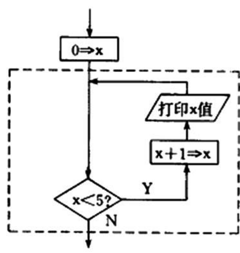


图 1.18

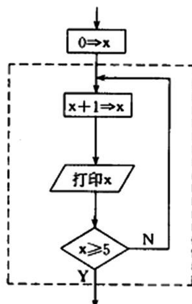


图 1.19

图 1.18 和图 1.19 的作用都是打印五个数 1,2,3,4,5。可以看到：对同一个问题既可以用当型循环来处理，也可以用直到型循环来处理。

请读者注意这两种循环结构的异同。(1) 两种循环结构都能处理需要重复执行的操作。(2) 当型循环是“先判断（条件是否成立），后执行（A 框）”。而直到型循环则是“先执行（A 框），后判断（条件）”。(3) 当型循环是当给定条件成立满足时执行 A 框，而直到型循环则是在给定条件不成立时执行 A 框。

试读结束：需要全本请在线购买： www.ertongbook.com

(4) 对同一个问题，如果分别用当型循环结构和直到型循环结构来处理的话，则两者结构中的判断框内的判断条件 p_1 和 p_2 恰为互逆条件。例如图 1.18 中 p_1 条件是“x<5”，而图 1.19 中 p_2 条件为“x≥5”。二者恰好相反，这是很重要的一个规律。请读者深入地掌握以上特点，不要弄错。

当型循环和直到型循环是可以互相转换的。凡用当型循环能处理的问题，用直到型循环亦可解决，反之亦然。请读者自己练习一下。

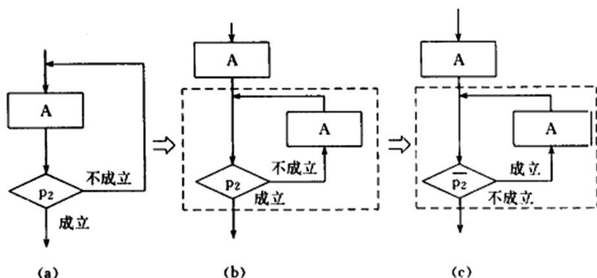


图 1.20

从图 1.20 可以看出怎样把一个直到型循环转换成一个当型循环结构。图 1.20 (a) 是一个直到型循环，可以用图 1.20 (b) 代替 (a)，二者的作用完全一样。因为直到型循环不论条件是否成立都要先执行一次 A 框，而当型循环则有可能一次也不执行 A 框，因此在 (b) 图最上方增加一个 A 框，以使 (a) 与 (b) 图的作用等价。(b) 图中的虚线框还不是一个当型循环结构，因为当型循环要求在给定的条件成立时反复执行 A 框，而 (b) 图中却是当 p_2 条件不成立时才执行 A 框。因此，还要再将 (b) 图变换成 (c) 图。将 (b) 图中的 p_2 条件变换成其逆条件，以 $\bar{p}_2$ 表示。这样 (c) 图中当 $\bar{p}_2$ 条件成立时，执行 A 框。例如，若 p_2 条件是“x≥5”，则 $\bar{p}_2$ 为“x<5”。假设 x=3，则对“x≥5”条件而言，该条件显然不成立。按 (b) 图应再执行 A 框。而对“x<5”条件而言，该条件却是成立的，按 (c) 图也应再执行 A 框。因此，可以看出 (b) 图与 (c) 图的作用完全相同。(c) 图虚线框内正是一个当型循环结构。因此可知，一个直到型循环等于一个 A 框加上一个当型循环（需要注意的是将给定的判断条件“取反”）。见图 1.21 示意。

反之，一个当型循环转换成直到型循环的情况如图 1.22 所示。

图 1.22 (a) 是一个当型循环结构。先将它变换成 (b) 图所示的流程图。(b) 图中虚线框还不是一个直到型循环结构。再将其变换为 (c) 图。(c) 图中的虚线框内是一个直到型循环。(c) 图中的判断条件 $\bar{p}_1$ 是对 (b) 图中 p_1 条件的“取反”。可知，一个当型循环相当于一个直到型循环结构出现在一个选择结构的一侧，见图 1.23。同样，直到型循环结构中的判断条件应为当型循环中判断条件 p_1 的“取反” $\bar{p}_1$。

对于一个具体的问题，究竟用当型循环还是用直到型循环由使用者自定。由于当型循环执行 A 框（称“循环体”）可以是零次到多次，而直到型循环则至少执行一次循环体，所以，当事先不能确定是否至少执行一次循环体的情况下，宁可用当型循环。

以上三种基本结构，有以下共同特点：

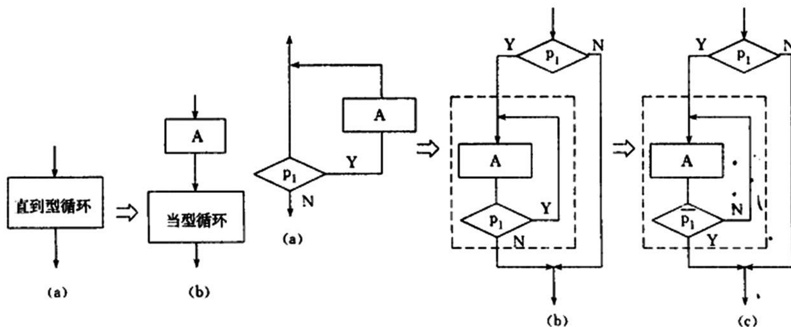


图 1.21　　　　　　　　　　　　　　图 1.22

(1) 只有一个入口。图 1.14～图 1.17 中的 a 点为入口点。

(2) 只有一个出口。图 1.14～图 1.17 中的 b 点为出口点。请注意，一个菱形判断框有两个出口，而一个选择结构只有一个出口。不要将菱形框的出口和选择结构的出口混淆。

(3) 结构内的每一部分都有机会被执行到。也就是说，对每一个框来说，都应当有一条从入口到出口的路径通过它。图 1.24 中没有一条从入口到出口的路径通过 A 框。

(4) 结构内不存在“死循环”(无终止的循环)。图 1.25 就是一个死循环。

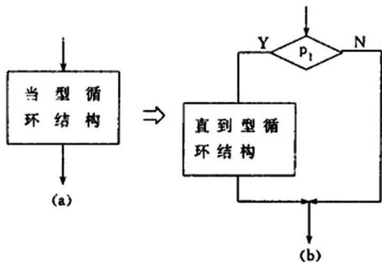


图 1.23

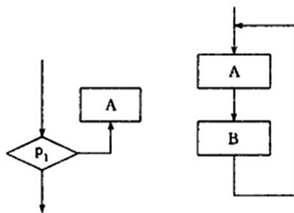


图 1.24　　　　图 1.25

已经证明，由以上三种基本结构顺序组成的算法结构，可以解决任何复杂的问题。由基本结构所构成的算法属于“结构化”的算法，它不存在无规律的转移，只在本基本结构内才存在分支和向前或向后的跳转。

其实，基本结构并不一定只限于上面三种，只要具有上述四个特点的都可以作为基本结构。人们可以自己定义基本结构，并由这些基本结构组成结构化程序。例如，我们也可以将图 1.26 和图 1.27 这样的结构定义为基本结构。图 1.27 所示的是一个多分支选择结构，根据给定的表达式的值决定执行哪一个框。图 1.26 和图 1.27 虚线框内的结构也是一个入口一个出口，并且有上述全部的四个特点。由它们构成的算法结构也是结构化的算法。但是，可以认为像图 1.26 和图 1.27 那样的结构是由三种基本结构所派生出来的。因此，人们都普遍认为最基本的是本节介绍的三种基本结构。

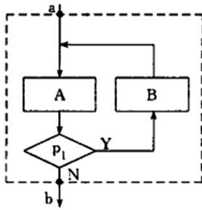


图 1.26

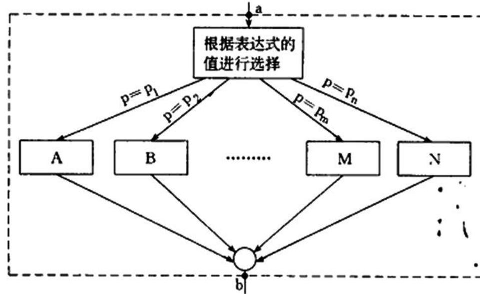


图 1.27

三、改进的流程图

ANSI 规定的流程图符号虽然应用较普遍，但随着结构化程序设计方法的出现，它也暴露出一些弱点，如它没有表示循环结构的符号，使用带箭头的流程线，允许流程的随意转移，这是不宜提倡的。

国际标准化组织 ISO（International Standard Organization）在 ANSI 提出的流程图符号的基础上，作了改进，提出了 ISO 流程图符号。除了图 1.3 所示的符号外，主要作了以下两点补充和改进：

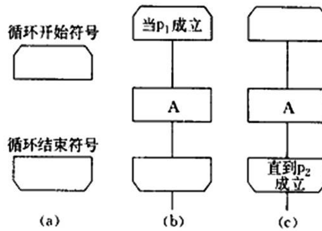


图 1.28

（1）增加了表示循环的符号，见图 1.28（a）。

用它可以表示当型循环（见图 1.28（b））和直到型循环（见图 1.28（c））。在循环开始和循环结束的框内写明循环继续的条件（当型循环）或循环约束的条件（直到型循环）。

（2）规定当流程由上到下和由左到右时，流程线可不带箭头（因为这是最常用的，符合顺序执行的特点的），如图 1.29 示意。

若流程是从下向上或从右向左，则流程线应加箭头。如图 1.30 所示。

这样做的目的是便于结构化的设计，减少箭头，使流程清晰，同时允许在必要时改变顺

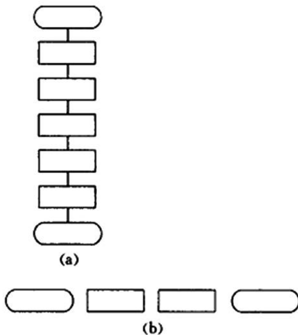


图 1.29

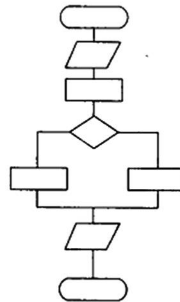
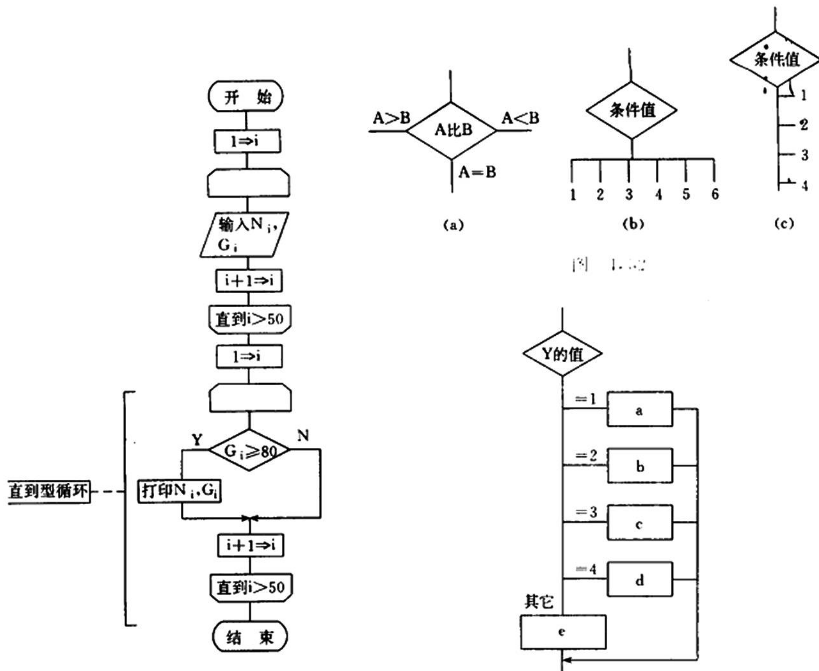


图 1.30

序执行的流程。

图 1.9 的流程图可以表示为图 1.31。

(3) 多出口的判断可以用图 1.32 所示的方法来表示。有三个出口的，可以用图 1.32 (a)，三个以上出口的用图 1.32 (b) 或 (c)。图 1.33 是当 Y 的值分别为 1，2，3，4 和“其它”时执行 a，b，c，d，e 操作的表示方法。



(a) (b) (c)

图 1.32

图 1.31　　图 1.33

我国已接受 ISO 标准提出的程序流程图符号，以此为基础制定出我国的标准，将于近期内公布推广。新标准规定的流程图符号见图 1.34。

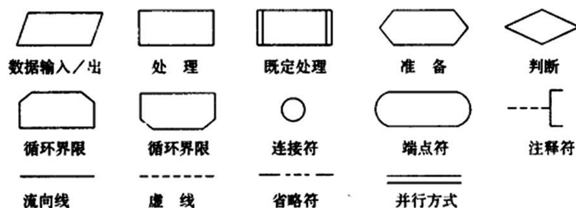


图 1.34

1.4.4 用N-S流程图表示算法

既然用基本结构的顺序组合可以表示任何复杂的算法结构，那么，基本结构之间的流程线就属多余的了。

1973 年美国学者 I. Nassi 和 B. Shneiderman 提出了一种新的流程图形式。在这种流程图中，完全去掉了带箭头的流程线。全部算法写在一个矩形框内，在该框内还可以包含其它的从属于它的框，或者说，由一些基本的框组成一个大的框。这种流程图又称 N-S 结构化流程图（N 和 S 是两位美国学者的英文姓名的第一个字母）。这种流程图适于结构化程序设计，因而很受欢迎。

N-S 流程图用以下的流程图符号：

（1）顺序结构

用图 1.35 形式表示。A 和 B 两个框组成一个顺序结构。

（2）选择结构

用图 1.36 表示。它与图 1.15 相应。当 p 条件成立时执行 A 操作，p 不成立则执行 B 操作。请注意图 1.36 是一个整体，代表一个基本结构。

（3）循环结构

当型循环结构用图 1.37 形式表示。

图 1.37 表示当 p_1 条件成立时反复执行 A 操作，直到 p_1 条件不成立为止。图 1.38 是执行过程示意，即：(1)先判断 p_1 条件；(2)若 p_1 成立，执行 A；(3)再判断 p_1 条件；(4)如 p_1 成立，再执行 A，如此反复；(5)判断 p_1 条件，若 p_1 条件不成立，不再执行 A 而脱离本结构。

直到型循环结构用图 1.39 形式表示，图 1.40 表示执行此结构的过程：（1）先执行 A 操作；（2）判断 p_2 条件；（3）若 p_2 不成立，再执行 A；（4）再判断 p_2 条件；（5）如 p_2 不成立，再执行 A；（6）再判断 p_2，如 p_2 条件成立，不再执行 A，脱离本循环。

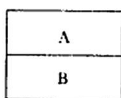


图 1.35

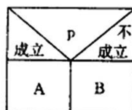


图 1.36

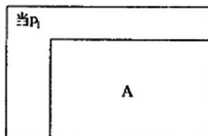


图 1.37

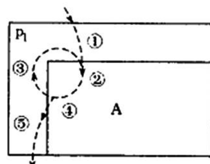


图 1.38

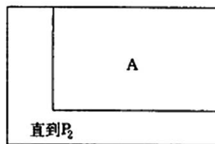


图 1.39

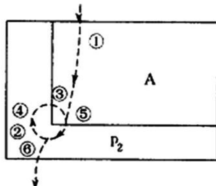


图 1.40

从图中看出：执行的路线从本结构的上方进入。对于当型循环（图 1.38）来说，先遇到

p_1 条件，然后进入 A 框。因此很容易知道图 1.38 形式是当型循环结构（先判断后执行）。对于直到型循环（图 1.40）来说，执行路线从上而下进入本结构，首先遇到的是 A 框，然后再遇 p_2 条件。因此很容易看出这种形式是直到型循环（先执行，后判断）。而不必死记。

在初学时，为清楚起见，可如图 1.37 和图 1.39 那样，写明"当 p_1"或"直到 p_2"，待熟悉之后，可以不写"当"和"直到"字样，只写"p_1"和"p_2"，如图 1.38 和图 1.40 那样。从图的形状即可知道是当型或直到型。

用以上三种 N-S 流程图中的基本框，可以组成复杂的 N-S 流程图，以表示算法。

应当说明，在图 1.35、图 1.36、图 1.37、图 1.39 中的"A"框或"B"框，可以是一个简单的操作（如读入数据或打印输出等），也可以是三个基本结构之一。例如，图 1.35 所示的顺序结构，其中的 A 可以又是一个选择结构，B 可以又是一个循环结构。如图 1.41 所示那样。

由这两个基本结构组成一个顺序结构。通过下面的几个例子，读者可以了解如何用 N-S 流程图表示算法。

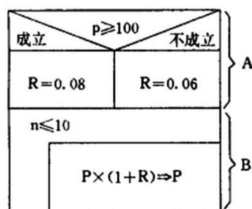


图 1.41

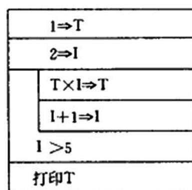


图 1.42

四、用 N-S 结构化流程图表示算法

【例 1.11】 将例 1.1 的求 5！算法用 N-S 图表示。

见图 1.42，它和图 1.7 对应。

【例 1.12】 将例 1.2 的算法用 N-S 图表示。将 50 名学生中成绩高于 80 分的学号和成绩打印出来。

见图 1.43 和图 1.44。它们和图 1.8 与图 1.9 对应，请读者注意图 1.43 和图 1.44 中直到型循环的终止循环的条件的表示方法（是"i>50"而不是"i≤50"）。为什么？

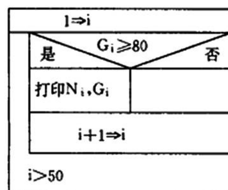


图 1.43

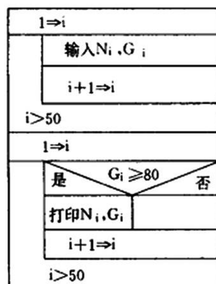


图 1.44

试读结束：需要全本请在线购买： www.ertongbook.com