

第2章 算法--程序的灵魂

➤ 一个程序主要包括以下两方面的信息：

(1) **对数据的描述**。在程序中要指定用到哪些数据以及这些数据的类型和数据的组织形式

◆ 这就是数据结构(**data structure**)

(2) **对操作的描述**。即要求计算机进行操作的步骤

◆ 也就是算法(**algorithm**)

- 数据是操作的对象
- 操作的目的是对数据进行加工处理，以得到期望的结果
- 著名计算机科学家沃思(**Niklaus Wirth**)提出一个公式：

$$\text{算法} + \text{数据结构} = \text{程序}$$

- 一个程序除了算法和数据结构这主要要素外，还应当采用结构化程序设计方法进行程序设计，并且用某一种计算机语言表示
- 算法、数据结构、程序设计方法和语言工具是一个程序设计人员应具备的知识

- 算法是解决“做什么”和“怎么做”的问题
- 程序中的操作语句，是算法的体现
- 不了解算法就谈不上程序设计

2.1 什么是算法

2.2 简单的算法举例

2.3 算法的特性

2.4 怎样表示一个算法

2.5 结构化程序设计方法

2.1 什么是算法

- 广义地说，为解决一个问题而采取的方法和步骤，就称为“**算法**”
- 对同一个问题，可以有不同的解题方法和步骤（一题多解）
- 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（最优算法，如何评价？）

2.1 什么是算法

➤ 计算机算法可分为两大类别：

◆ 数值运算算法

◆ 非数值运算算法

➤ 数值运算的目的是求数值解

➤ 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

2.2简单的算法举例

例2.1 求 $1 \times 2 \times 3 \times 4 \times 5 \times \dots \times 1000$

➤ 可以用最原始的方法求。

◆ 步骤1：先算 1×2 。

◆ 步骤2：将步骤1得到的乘积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。


太繁琐

2.2简单的算法举例


➤ 改进的算法:

- ◆ 设变量 p 为被乘数
- ◆ 变量 i 为乘数
- ◆ 用循环算法求结果

2.2简单的算法举例

- **S1**: 使 $p=1$, 或写成 $1 \Rightarrow p$
- **S2**: 使 $i=2$, 或写成 $2 \Rightarrow i$
- **S3**: 使 p 与 i 相乘, 乘积仍放在变量 p 中, 可表示为: $p*i \Rightarrow p$

- **S4**: 使 i 的值加1, 即 $i+1 \Rightarrow i$
- **S5**: 如果 i 不大于5, 返回重新执行**S3**; 否则, 算法结束
- 最后得到 p 的值就是 $5!$ 的值

若求 $1 \times 3 \times 5 \times 7 \times 9 \times 11$

- **S1**: 使 $p=1$, 或写成 $1 \Rightarrow p$
- **S2**: 使 $i=3$, 或写成 $3 \Rightarrow i$
- **S3**: 使 p 与 i 相乘, 乘积仍放在变量 p 中, 可表示为: $p*i \Rightarrow p$
 相当于 $i \leq 11$
- **S4**: 使 i 的值加 2, 即 $i+2 \Rightarrow i$
- **S5**: 如果 i 不大于 11 返回重新执行 **S3**; 否则, 算法结束
- 最后得到 p 的值就是 $1 \times 3 \times 5 \times 7 \times 9 \times 11$ 的值

例2.2 有**50**个学生，要求将成绩在**80**分以上的学生的学号和成绩输出。

➤ 用 n_i 代表第 i 个学生学号， g_i 表示第 i 个学生成绩

S1: $1 \Rightarrow i$

S2: 如果 $g_i \geq 80$,

则输出 n_i 和 g_i ，否则不输出

S3: $i+1 \Rightarrow i$

S4: 如果 $i \leq 50$ ，返回到步骤**S2**，继续执行，
否则，算法结束

**例2.3 判定2000—2500年中的每一年是
否闰年，并将结果输出。**

➤ **闰年的条件：**

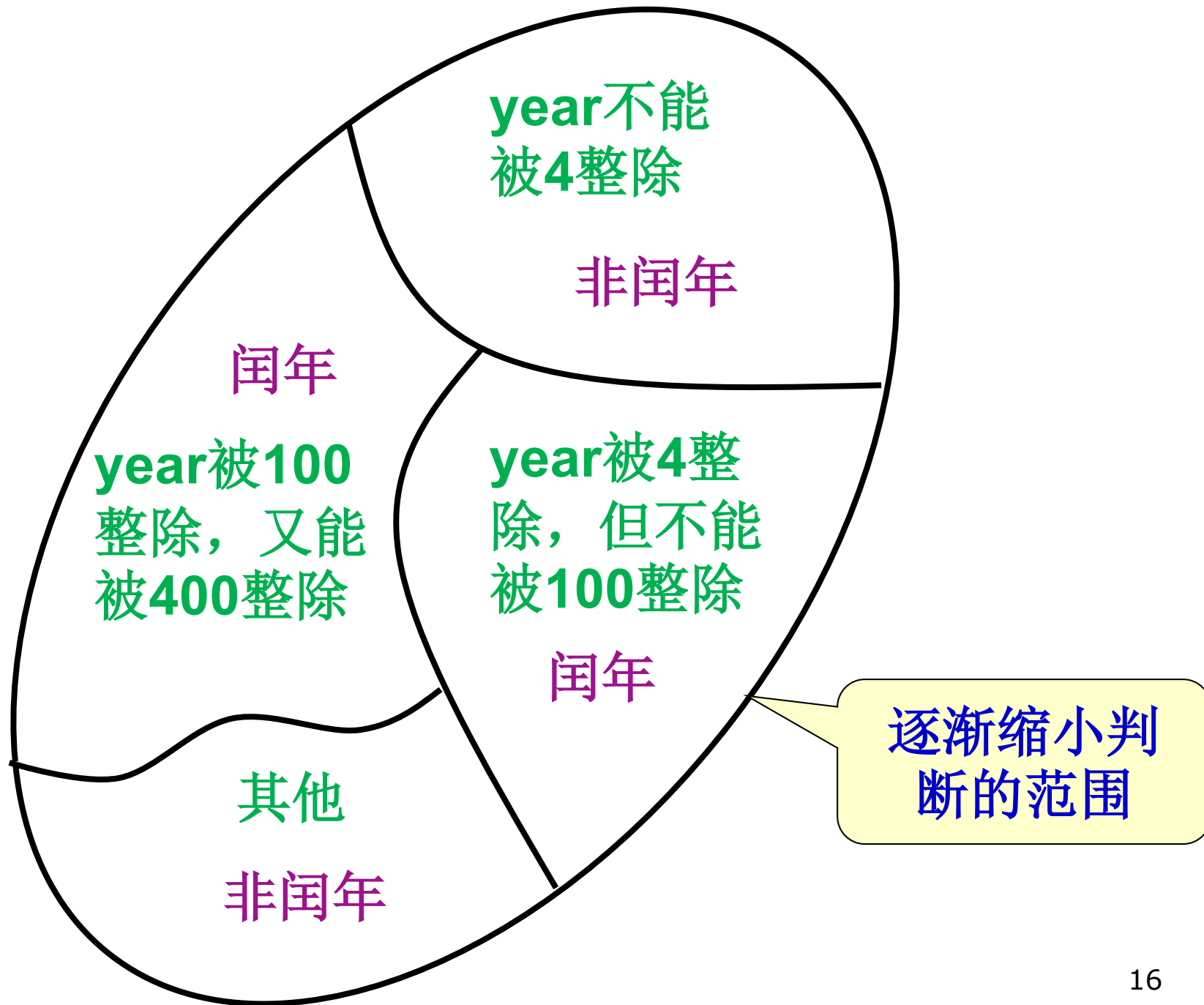
**(1)能被4整除，但不能被100整除的年份都是
闰年，如2008、2012、2048年**

(2)能被400整除的年份是闰年，如2000年

◆ **不符合这两个条件的年份不是闰年**

◆ **例如2009、2100年**

- 设 year 为被检测的年份。算法表示如下：
- ◆**S1:** $2000 \Rightarrow \text{year}$
 - ◆**S2:** 若 year 不能被4整除，则输出 year 的值和“不是闰年”。然后转到**S6**
 - ◆**S3:** 若 year 能被4整除，不能被100整除，则输出 year 的值和“是闰年”。然后转到**S6**
 - ◆**S4:** 若 year 能被400整除，则输出 year 的值和“是闰年”，然后转到**S6**
 - ◆**S5:** 其他情况输出 year 的值和“不是闰年”
 - ◆**S6:** $\text{year} + 1 \Rightarrow \text{year}$
 - ◆**S7:** 当 $\text{year} \leq 2500$ 时，转**S2**，否则停止



例**2.4** 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$

➤规律:

- ①第**1**项的分子分母都是**1**
- ② 第**2**项的分母是**2**，以后每一项的分母子都是前一项的分母加**1**
- ③ 第**2**项前的运算符为“-”，后一项前面的运算符都与前一项前的运算符相反

例**2.4** 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{99} - \frac{1}{100}$

sign—当前项符号
term—当前项的值
sum—当前各项的和
deno—当前项分母

➤ **S1: sign=1**

➤ **S2: sum=1**

➤ **S3: deno=2**

-1

➤ **S4: sign=(-1)*sign**

-1/2

➤ **S5: term=sign*(1/deno)**

1-1/2

➤ **S6: sum=sum+term**

3

➤ **S7: deno=deno+1**

满足，返回S4

➤ **S8: 若deno≤100返回S4；否则算法结束**

例**2.4** 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{99} - \frac{1}{100}$

sign—当前项符号
term—当前项的值
sum—当前各项的和
deno—当前项分母

➤ **S1: sign=1**

➤ **S2: sum=1**

➤ **S3: deno=2**

1

➤ **S4: sign=(-1)*sign**

1/3

➤ **S5: term=sign*(1/deno)**

1-1/2+1/3

➤ **S6: sum=sum+term**

4

➤ **S7: deno=deno+1**

满足，返回**S4**

➤ **S8: 若deno≤100返回S4; 否则算法结束**

例**2.4** 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$

➤ **S1: sign=1**

➤ **S2: sum=1**

➤ **S3: deno=2**

➤ **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**

99次循环后sum的值
就是所要求的结果

例2.5 给出一个大于或等于**2**的正整数，判断它是不是一个素数。

- 所谓素数(**prime**)，是指除了**1**和该数本身之外，不能被其他任何整数整除的数
- 例如，**13**是素数，因为它不能被**2**，**3**，**4**，...，**12**整除。

➤判断一个数 $n(n \geq 2)$ 是否素数：将 n 作为被除数，将2到 $(n-1)$ 各个整数先后作为除数，如果都不能被整除，则 n 为素数

S1: 输入 n 的值

S2: $i=2$ (i 作为除数)

S3: n 被 i 除，得余数 r

S4: 如果 $r=0$ ， 则输出 n “不是素数”， 否则执行**S5**

S5: $i+1 \Rightarrow i$

S6: 如果 $i \leq n-1$ ，返回**S3**；否则输出 n “是素数”，然后结束。

2.3算法的特性

➤一个有效算法应该具有以下**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。

(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。

2.3算法的特性

➤一个有效算法应该具有以下**特点**:

(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。

(4) 有一个或多个输出。算法的目的是为了求解, “解” 就是输出。

◆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。

2.3算法的特性

➤对于一般最终用户来说:

- ◆他们并不需要在处理每一个问题时都要自己设计算法和编写程序
- ◆可以使用别人已设计好的现成算法和程序
- ◆只需根据已知算法的要求给予必要的输入，就能得到输出的结果



2.4怎样表示一个算法

➤常用的方法有：

◆自然语言

◆传统流程图

◆结构化流程图

◆伪代码

◆.....

2.4怎样表示一个算法

2.4.1 用自然语言表示算法

2.4.2 用流程图表示算法

2.4.3 三种基本结构和改进的流程图

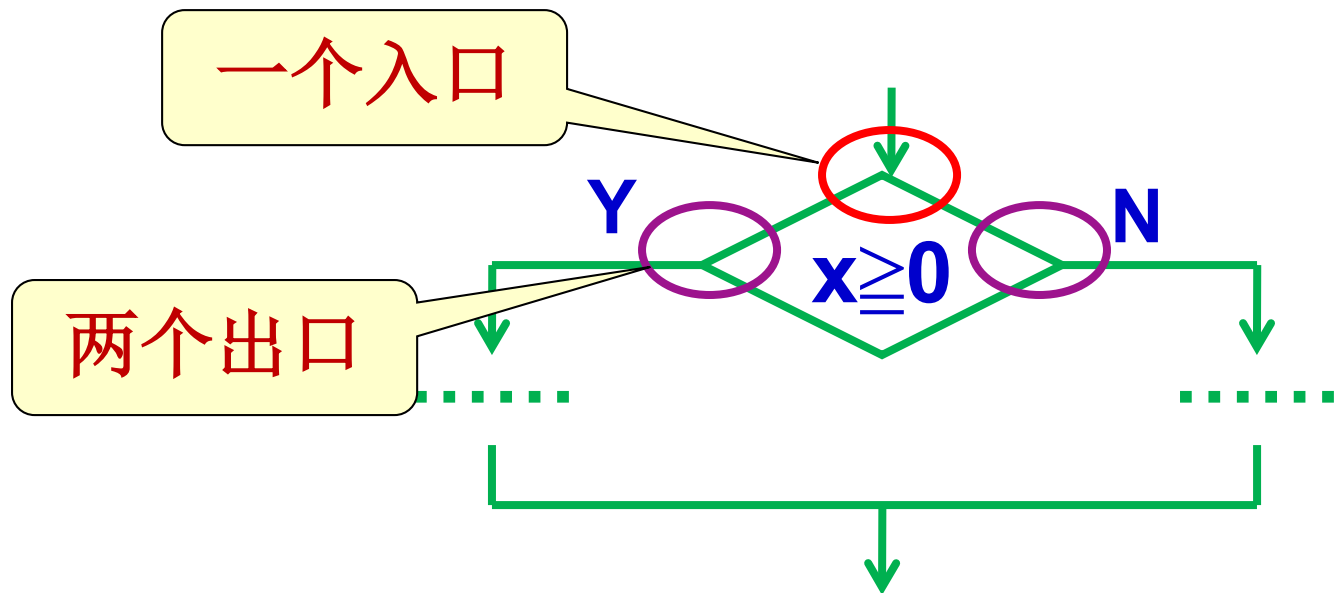
2.4.4 用N-S流程图表示算法

2.4.5 用伪代码表示算法

2.4.6 用计算机语言表示算法

2.4.1 用自然语言表示算法

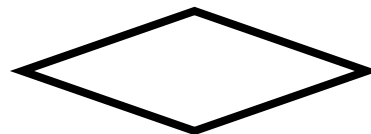
- 2.2节介绍的算法都是用自然语言表示的
- 用自然语言表示通俗易懂，但文字冗长，容易出现歧义性
- 用自然语言描述包含分支和循环的算法，很不方便
- 除了很简单的问题外，一般不用自然语言



起止框



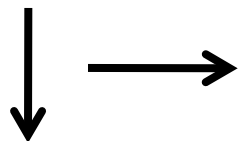
输入输出框



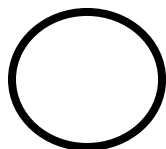
判断框



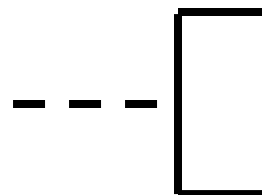
处理框



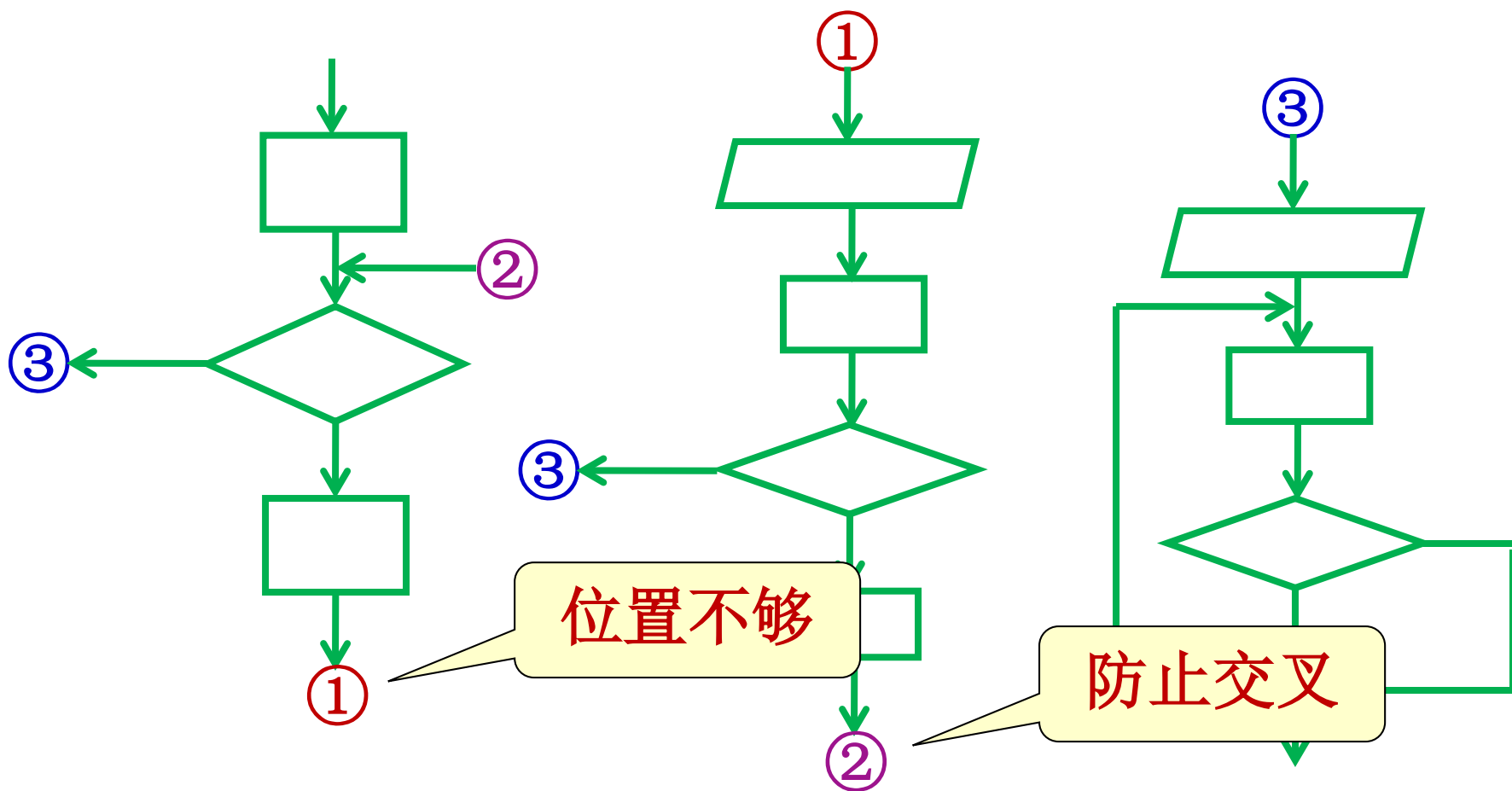
流程线



连接点



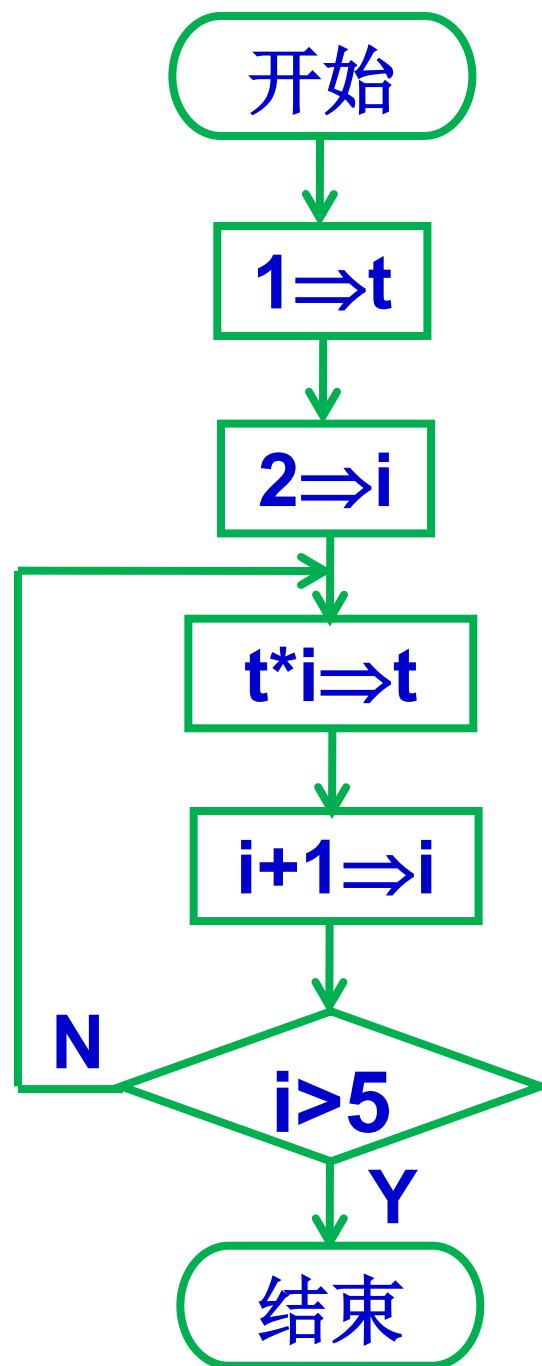
注释框



例2.6 将例2.1的算法用流程图表示。

求 $1 \times 2 \times 3 \times 4 \times 5$

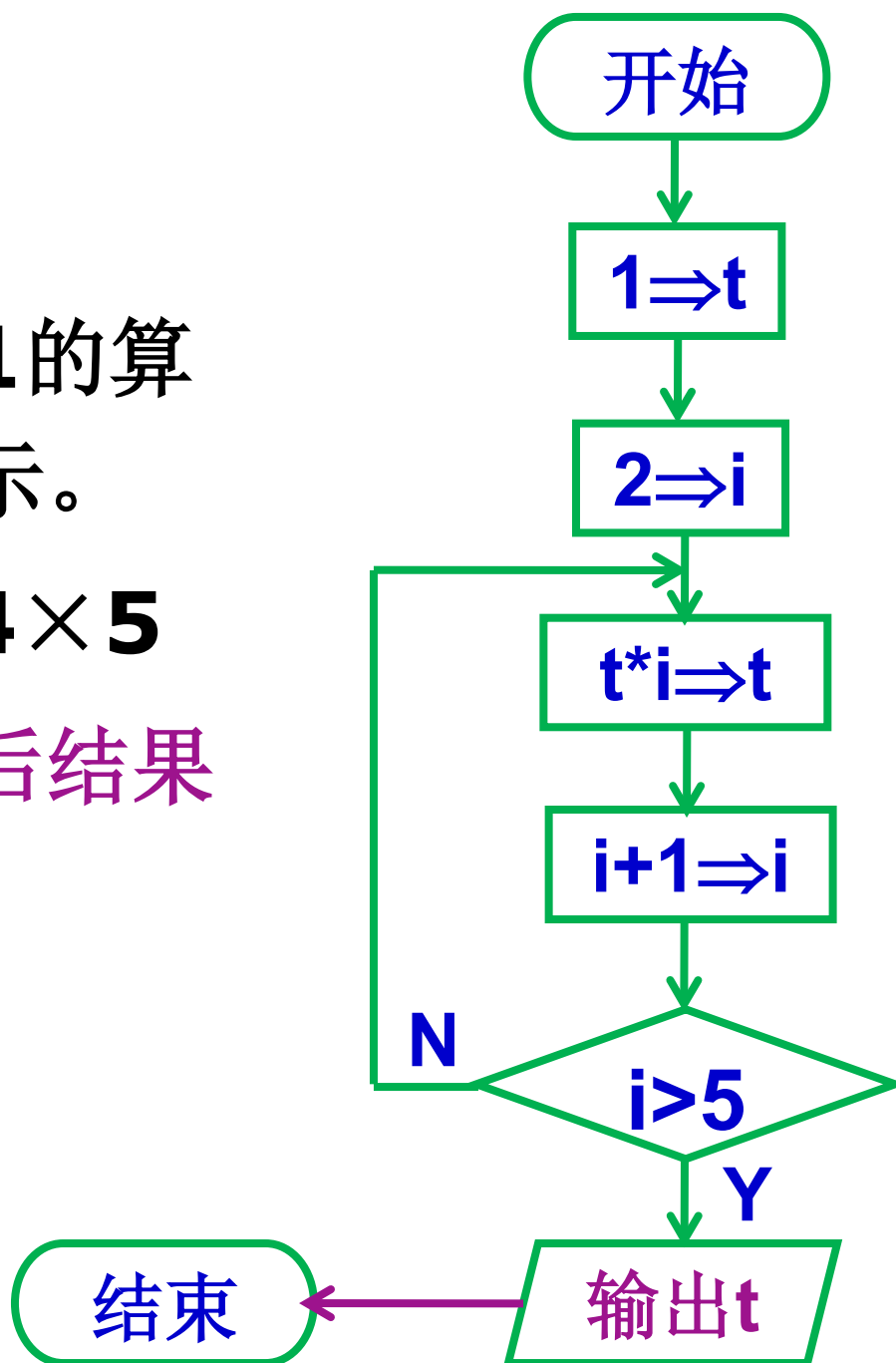
➤ 如果需要将最后结果输出：



例2.6 将例2.1的算法用流程图表示。

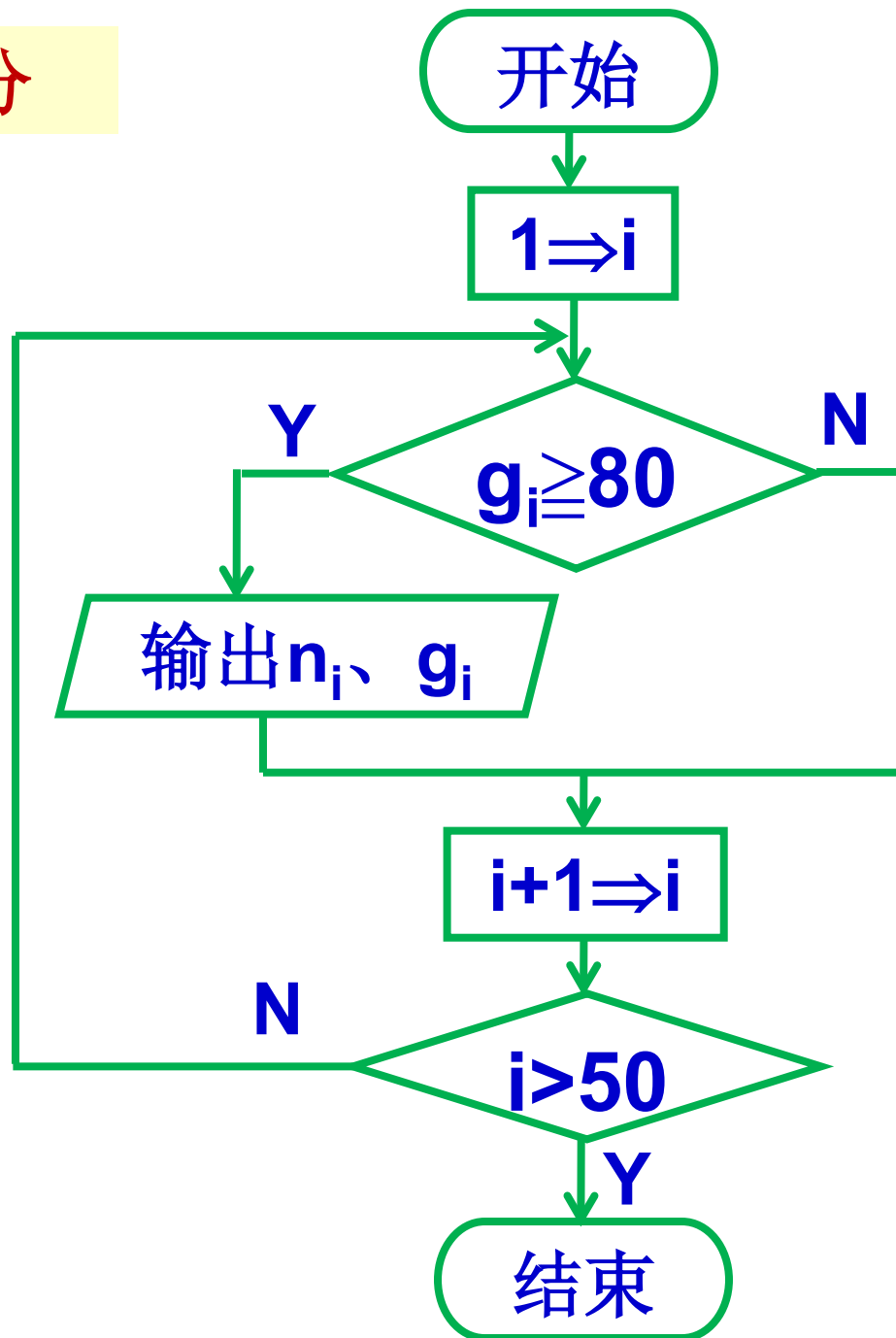
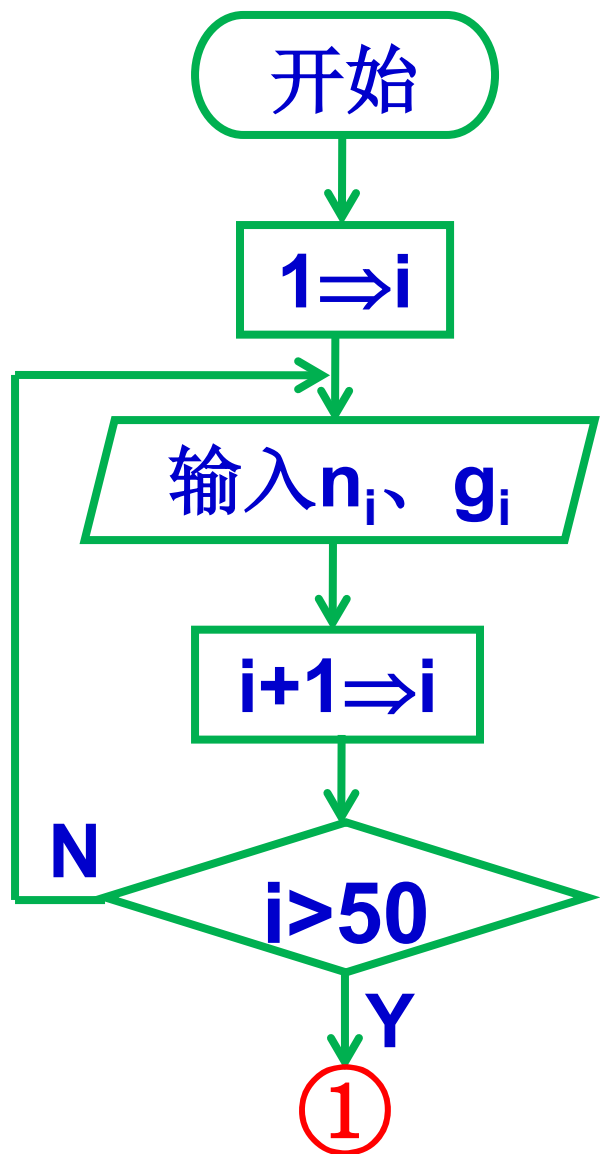
求 $1 \times 2 \times 3 \times 4 \times 5$

➤ 如果需要将最后结果输出：

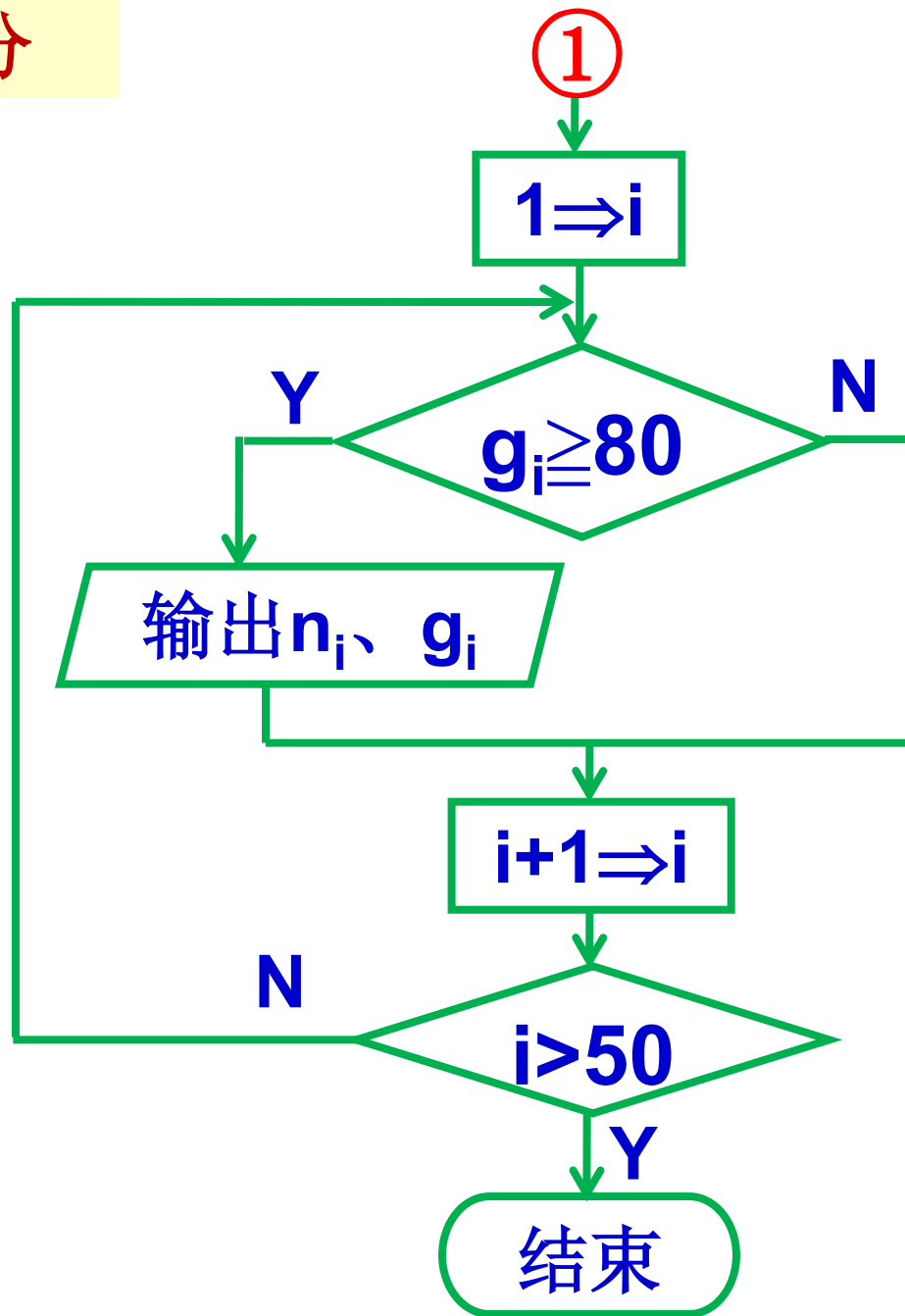
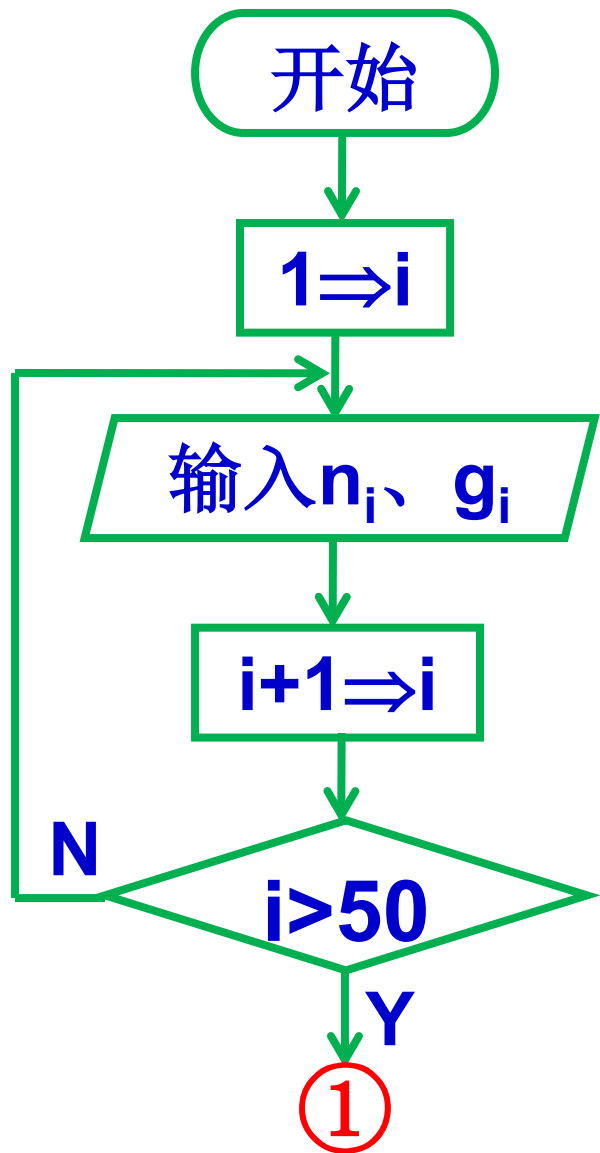


例2.7 例**2.2**的算法用流程图表示。有**50**个学生，要求将成绩在**80**分以上的学生的学号和成绩输出。

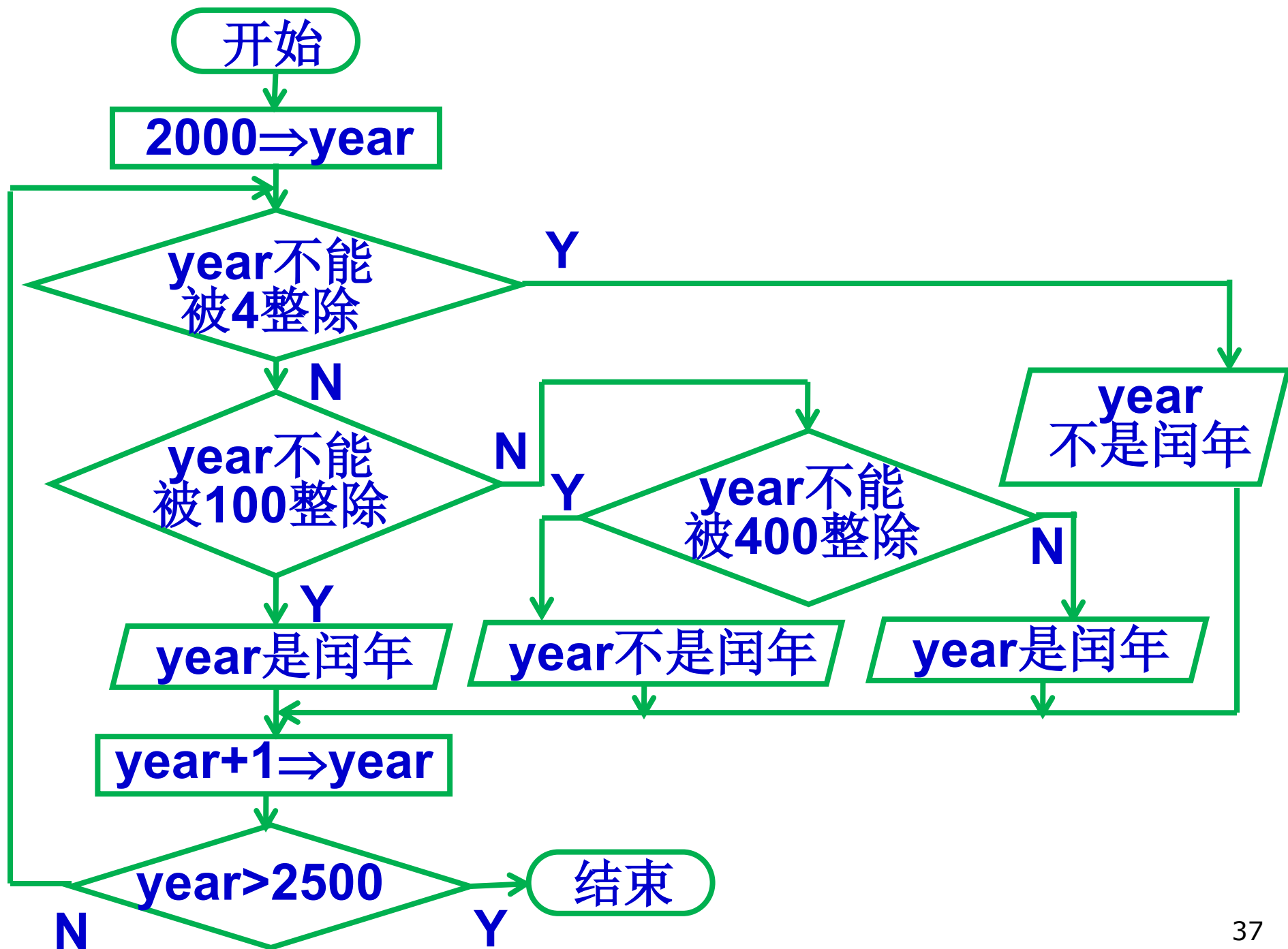
如果包括输入数据部分



如果包括输入数据部分

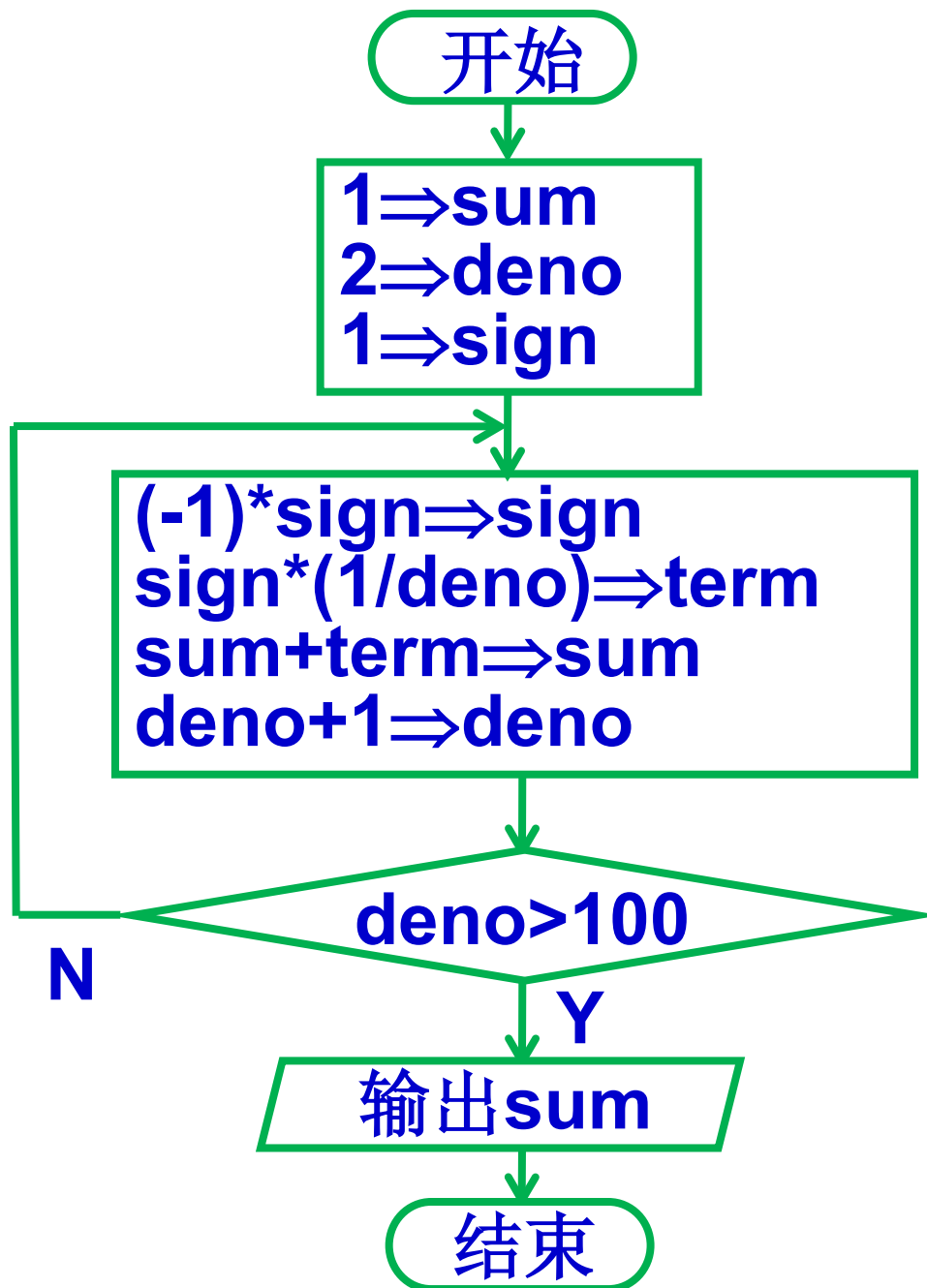


例2.8 例2.3判定闰年的算法用流程图表示。判定**2000—2500**年中的每一年是否闰年，将结果输出。

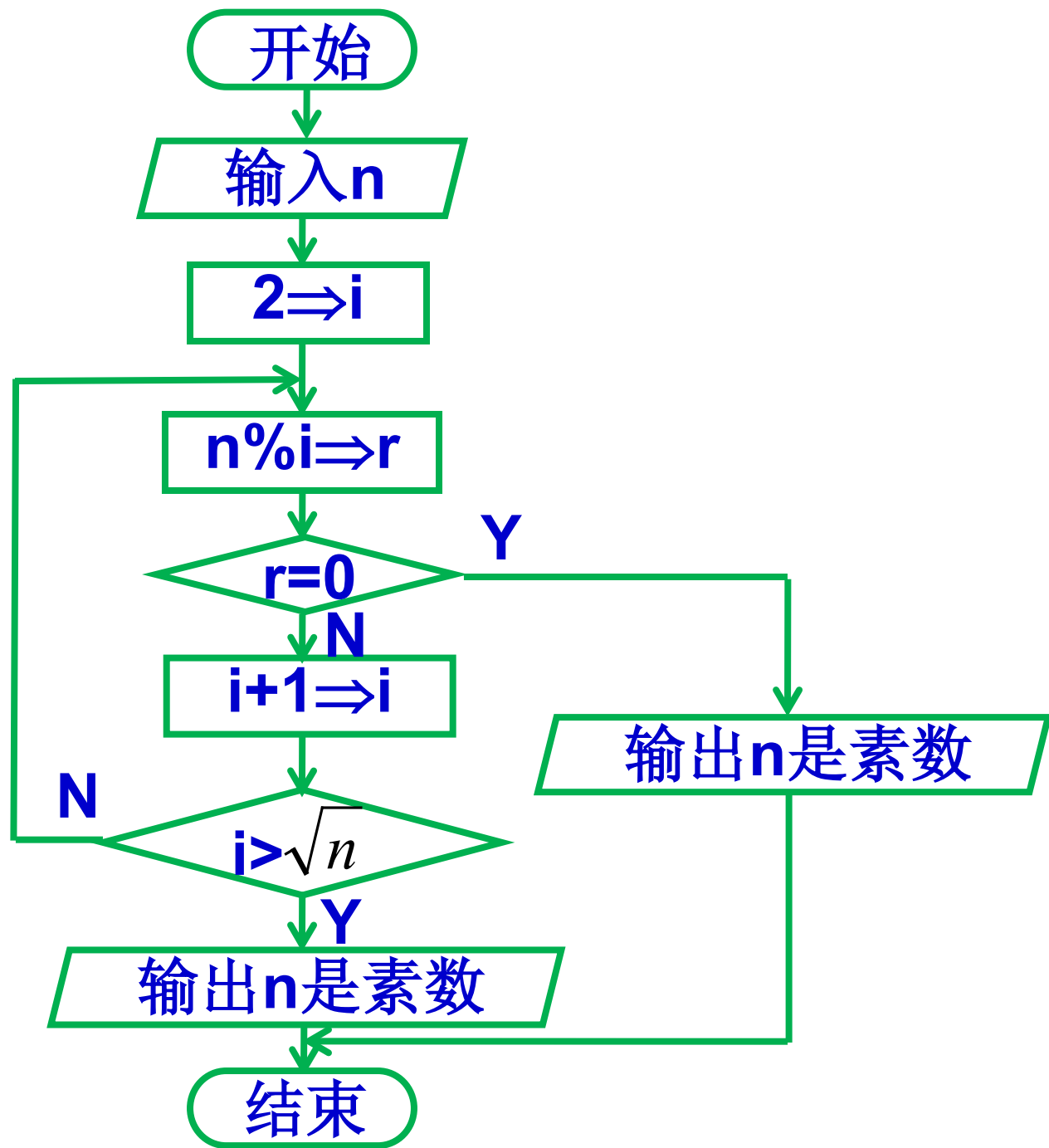


例2.9 将例**2.4**的算法用流程图表示。求

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$$



例2.10 例2.5判断素数的算法用流程图表示。对一个大于或等于**3**的正整数，判断它是不是一个素数。



- 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
- 流程线不要忘记画箭头，否则难以判定各框的执行次序

2.4.3 三种基本结构和改进的流程图

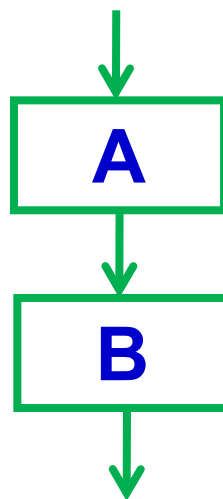
1. 传统流程图的弊端

- 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
- 使用者可以毫不受限制地使流程随意地转来转去，使人难以理解算法的逻辑

2.4.3 三种基本结构和改进的流程图

2. 三种基本结构

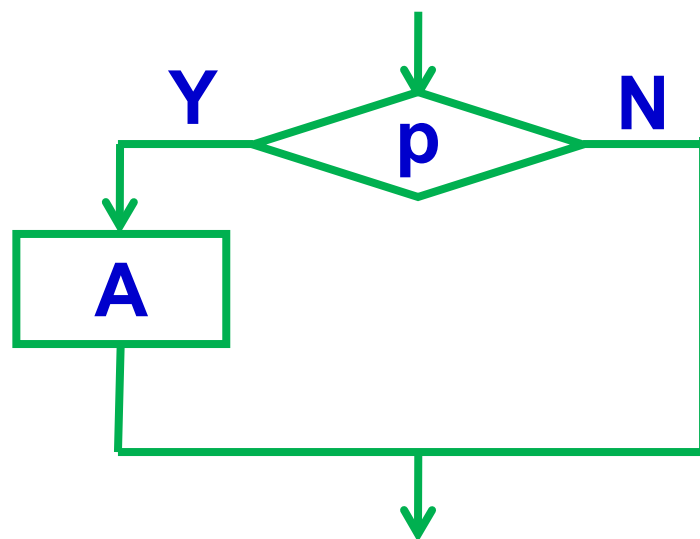
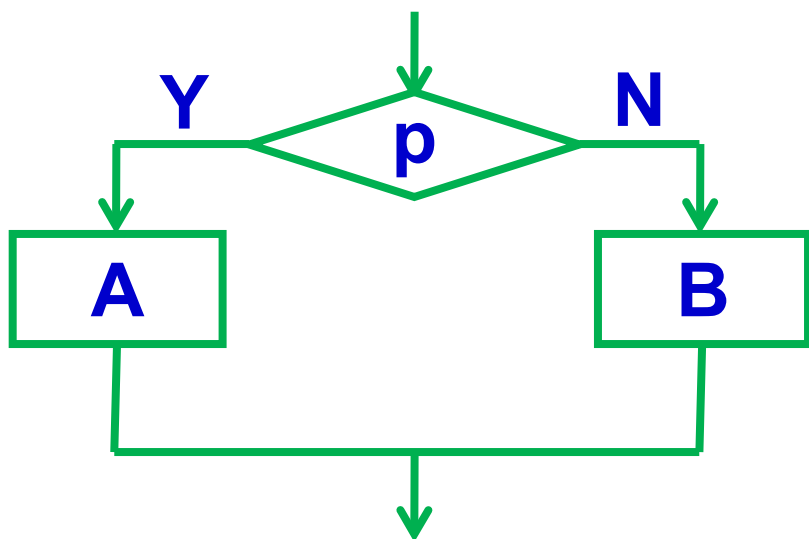
(1) 顺序结构



2.4.3 三种基本结构和改进的流程图

2. 三种基本结构

(2) 选择结构



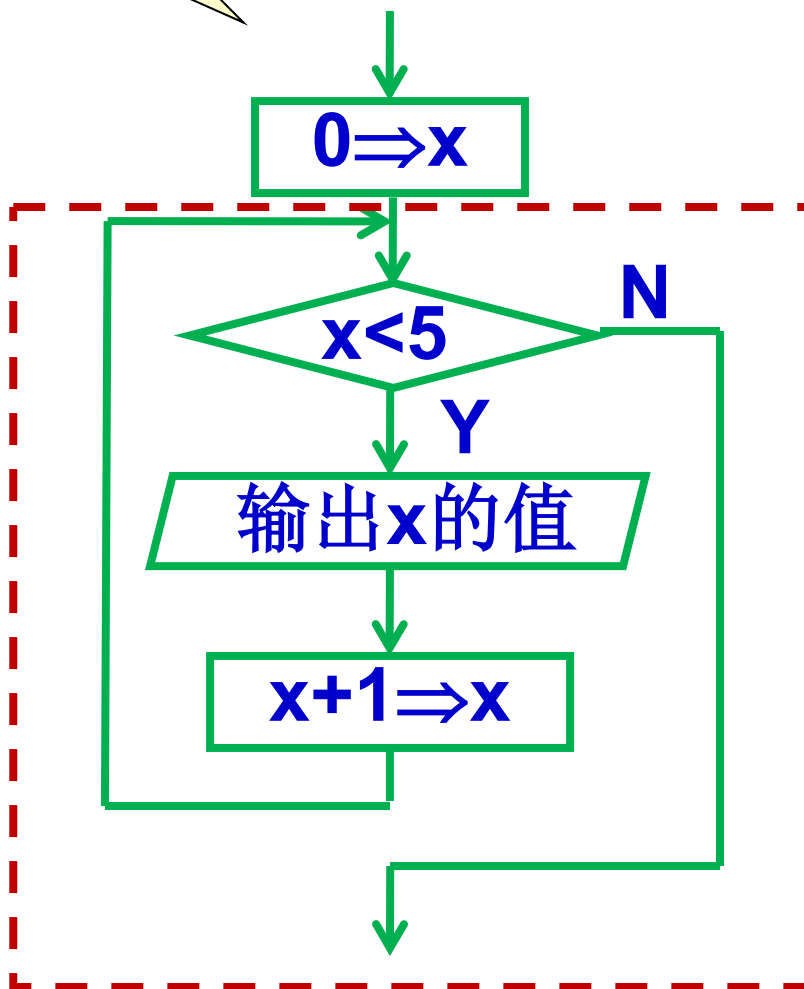
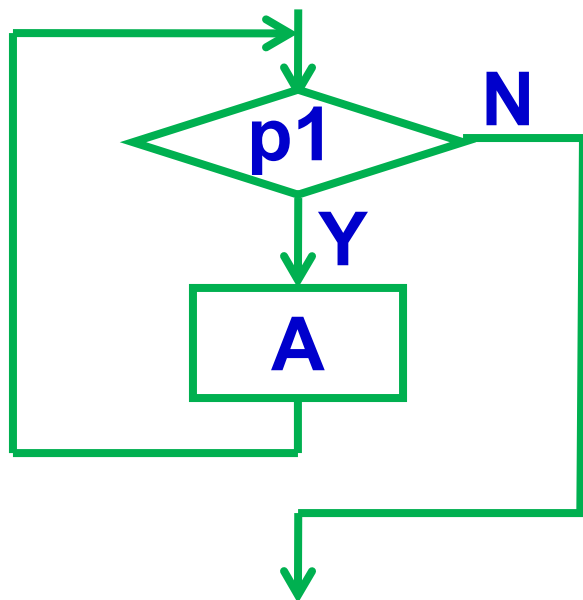
2.4.3 三种基本结构的改进的流程图

输出1,2,3,4,5

2.三种基本结构

(3) 循环结构

① 当型循环结构



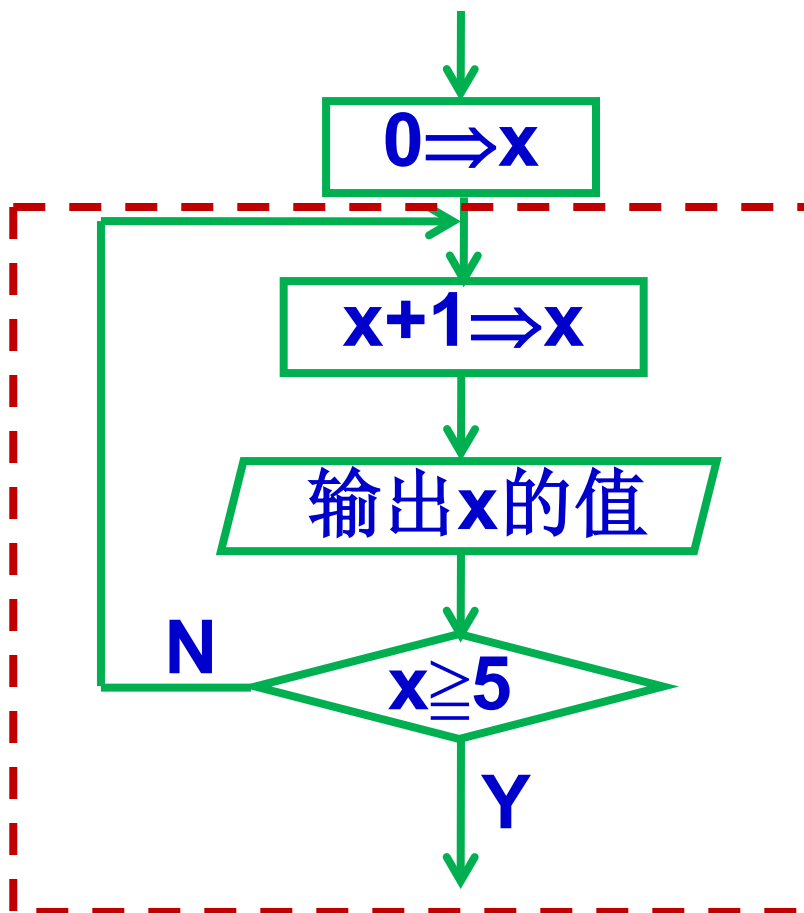
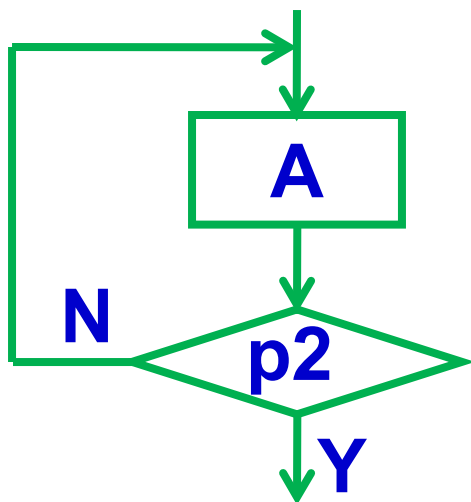
2.4.3 三种基本结构的改进的流程图

输出1,2,3,4,5

2.三种基本结构

(3) 循环结构

② 直到型循环结构



➤ 以上三种基本结构，有以下共同特点：

(1) 只有一个入口

(2) 只有一个出口

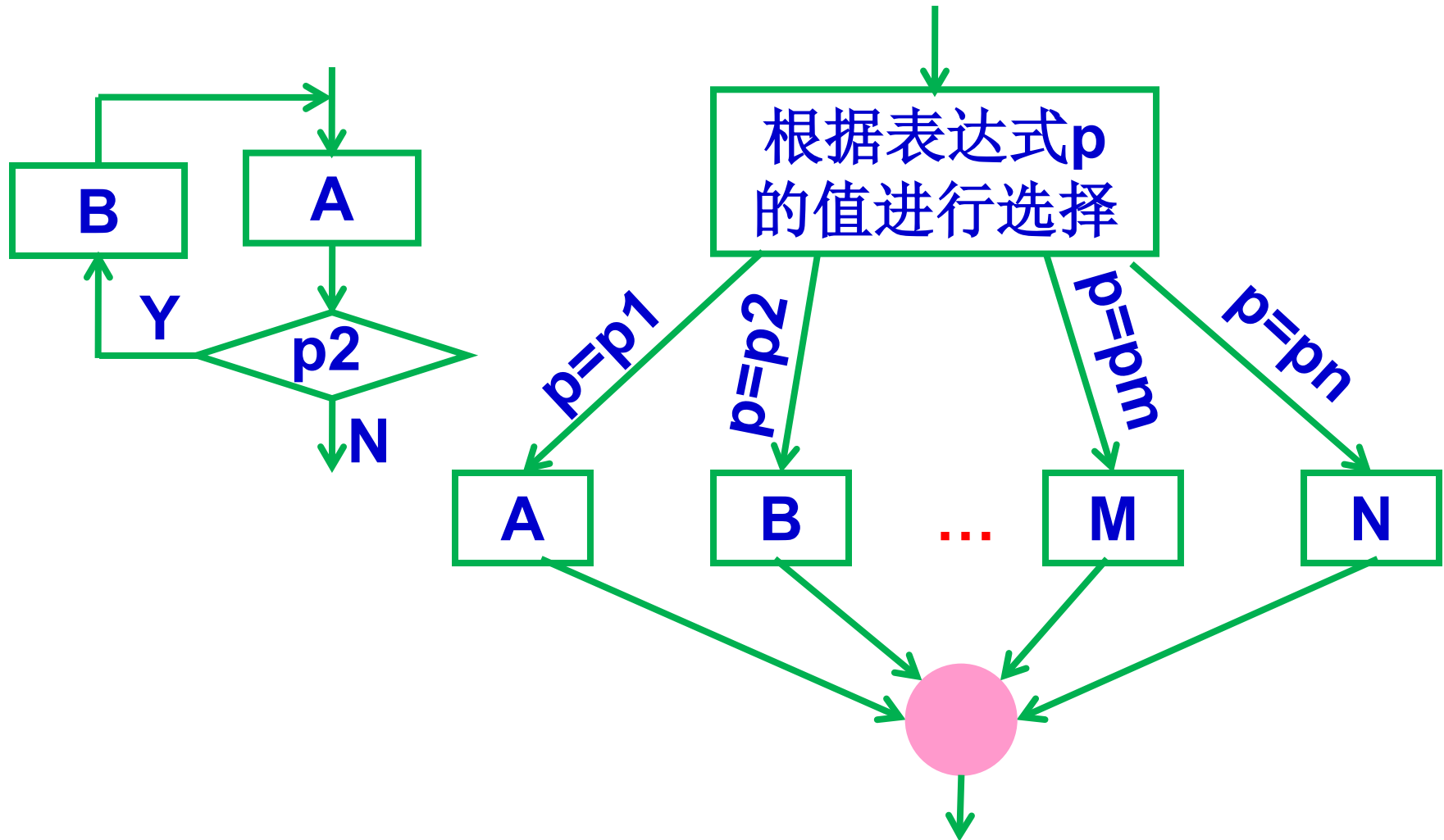
- 一个判断框有两个出口

- 一个选择结构只有一个出口

(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

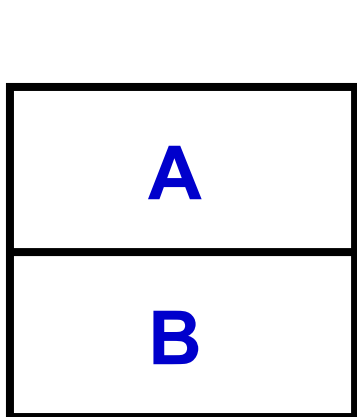
(4) 结构内不存在“死循环”

➤ 由三种基本结构派生出来的结构：

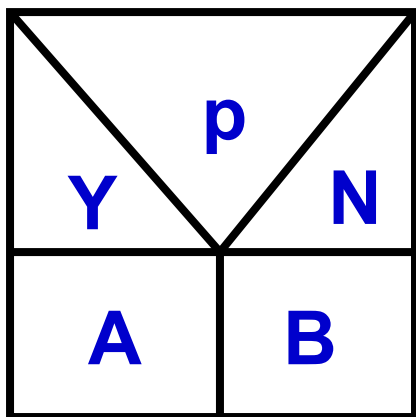


2.4.4 用N-S流程图表示算法

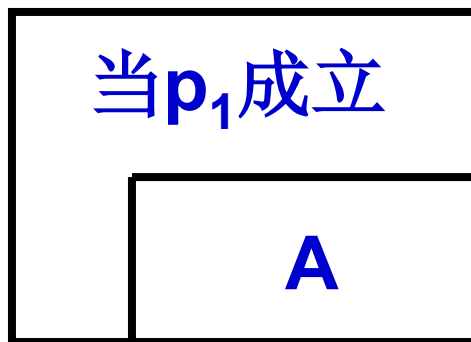
➤ **N-S**流程图用以下的流程图符号：



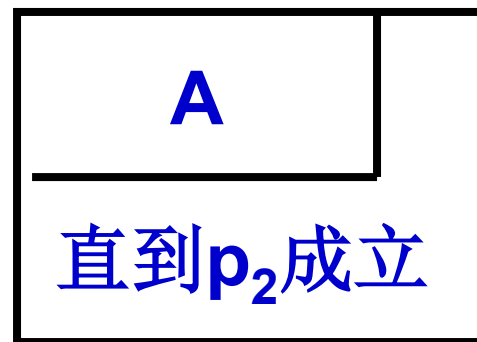
顺序结构



选择结构



循环结构
(当型)

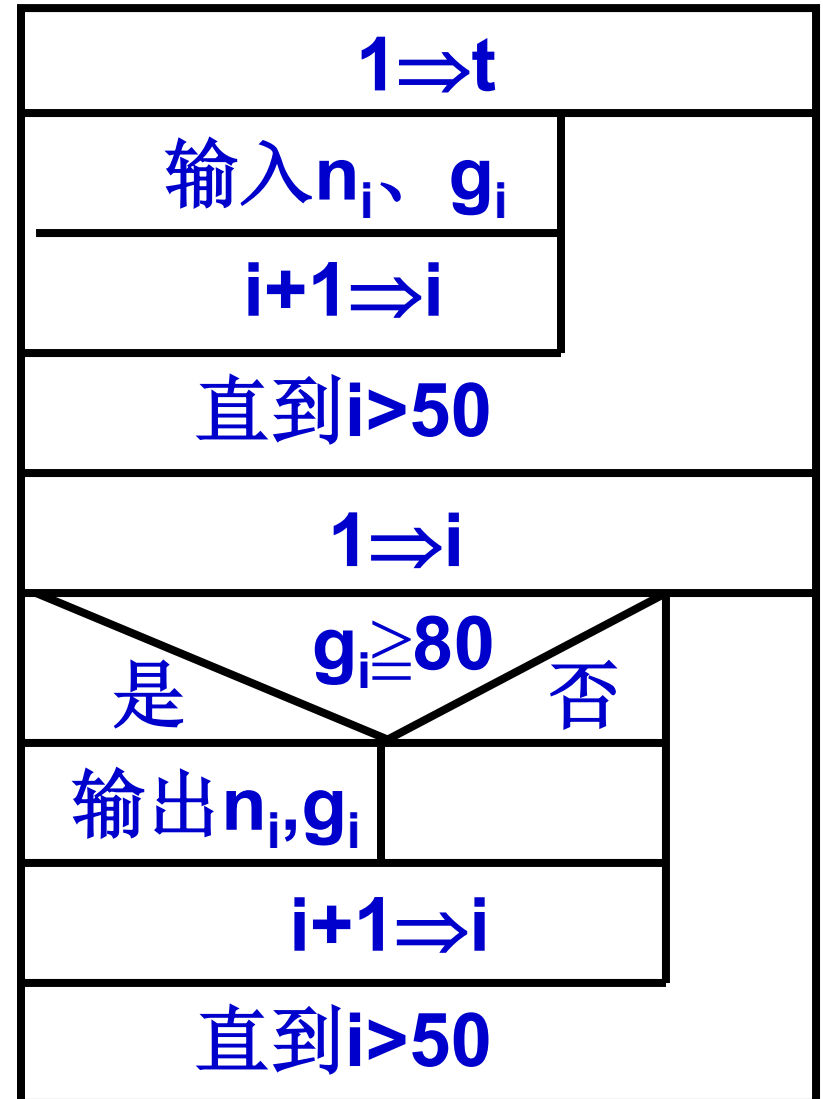


循环结构
(直到型)

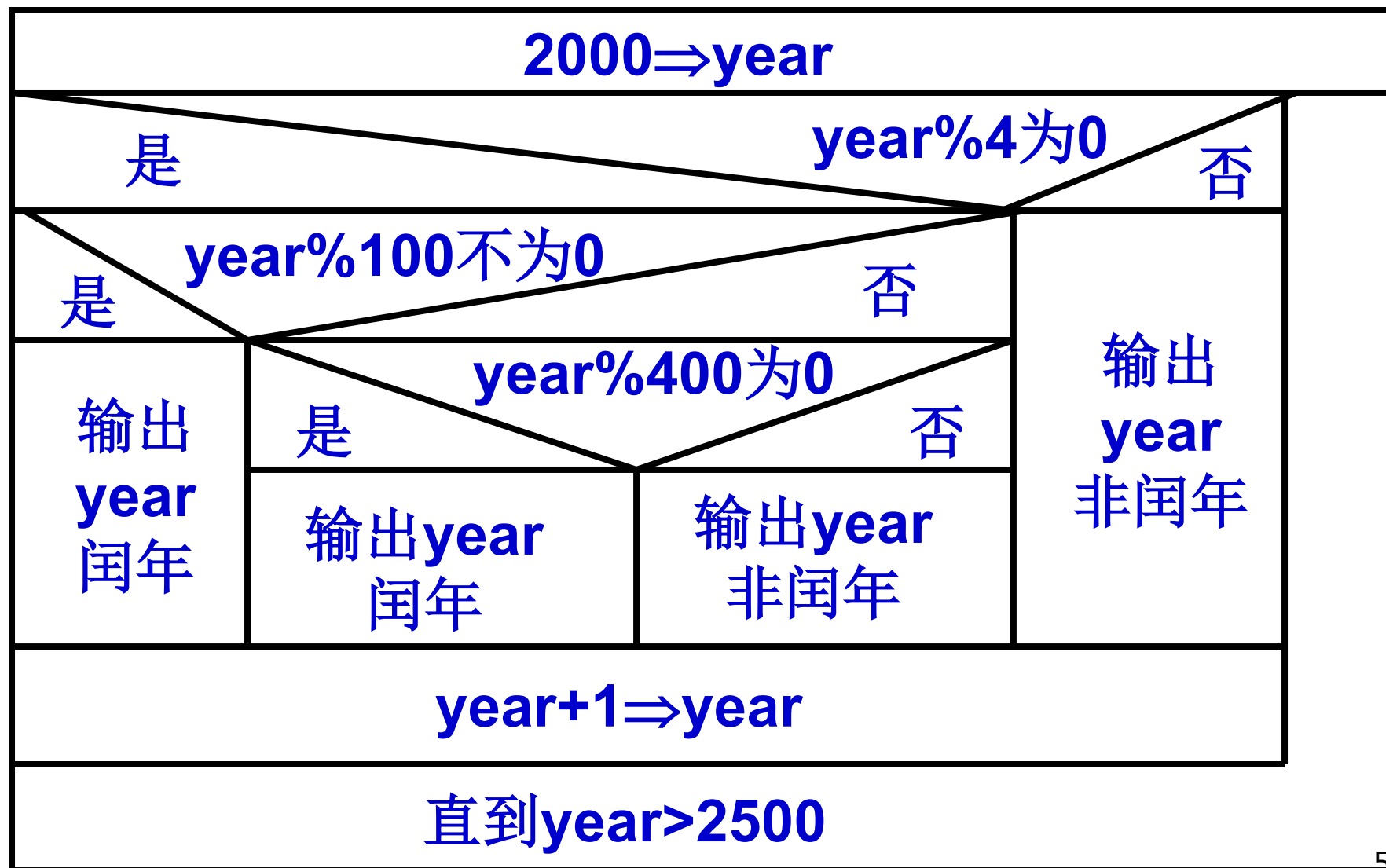
例**2.11**将例**2.1**的求**5!**算法用**N-S**图表示。



例**2.12** 将例**2.2**的
算法用**N-S**图表示
。将**50**名学生中成
绩高于**80**分者的学
号和成绩输出。

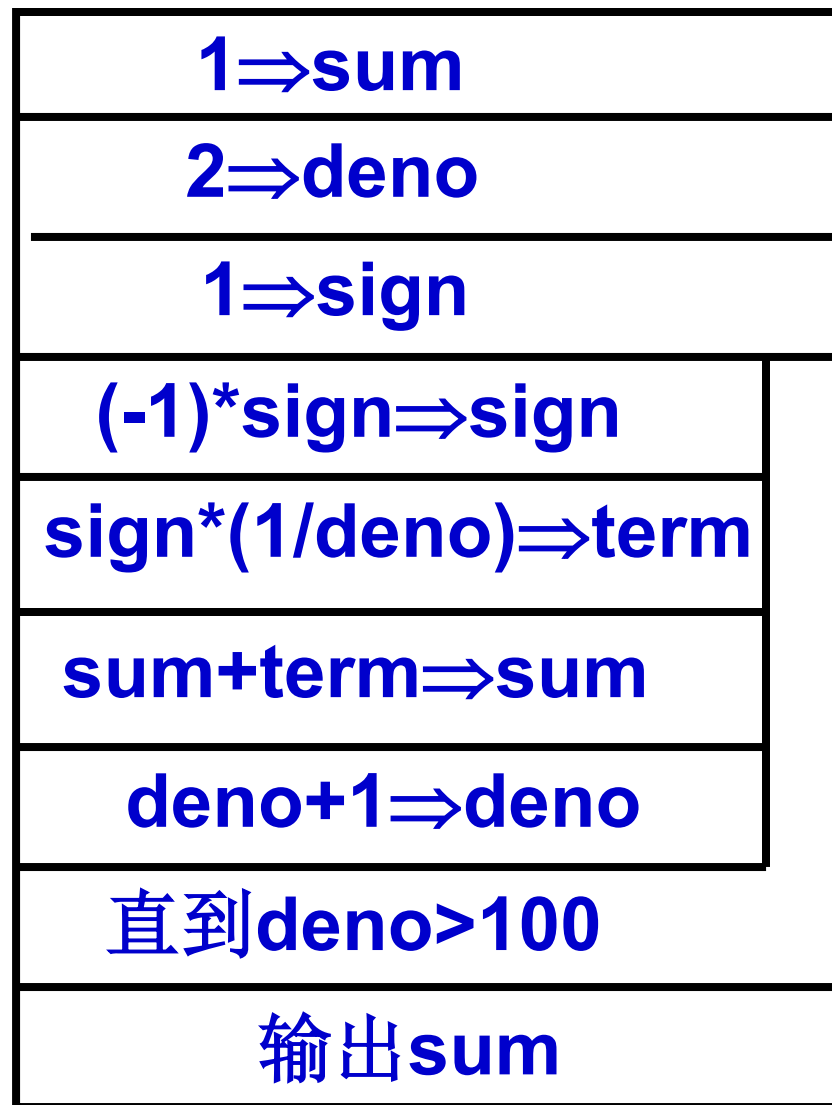


例2.13 将例2.3判定闰年的算法用N-S图表示



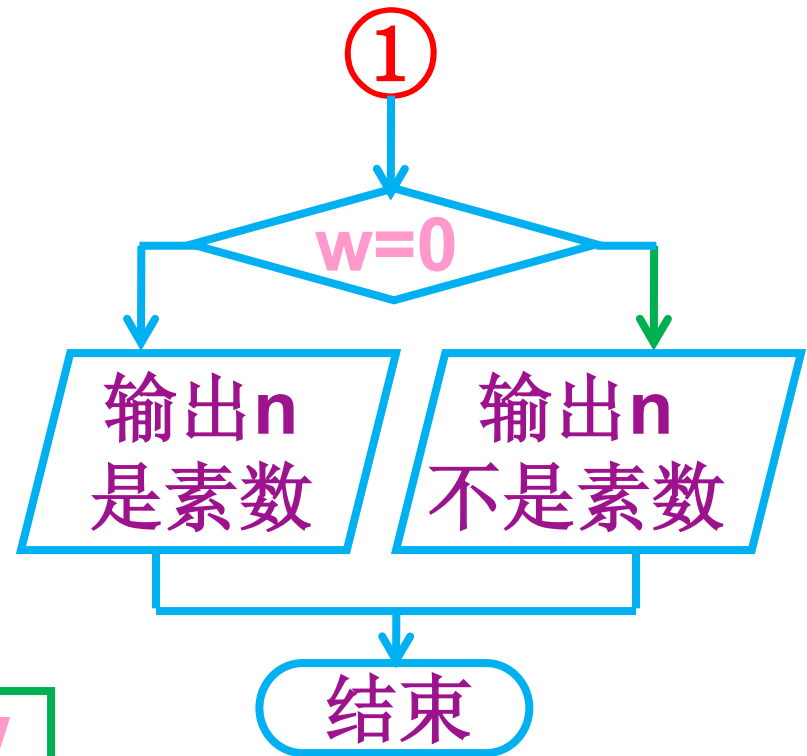
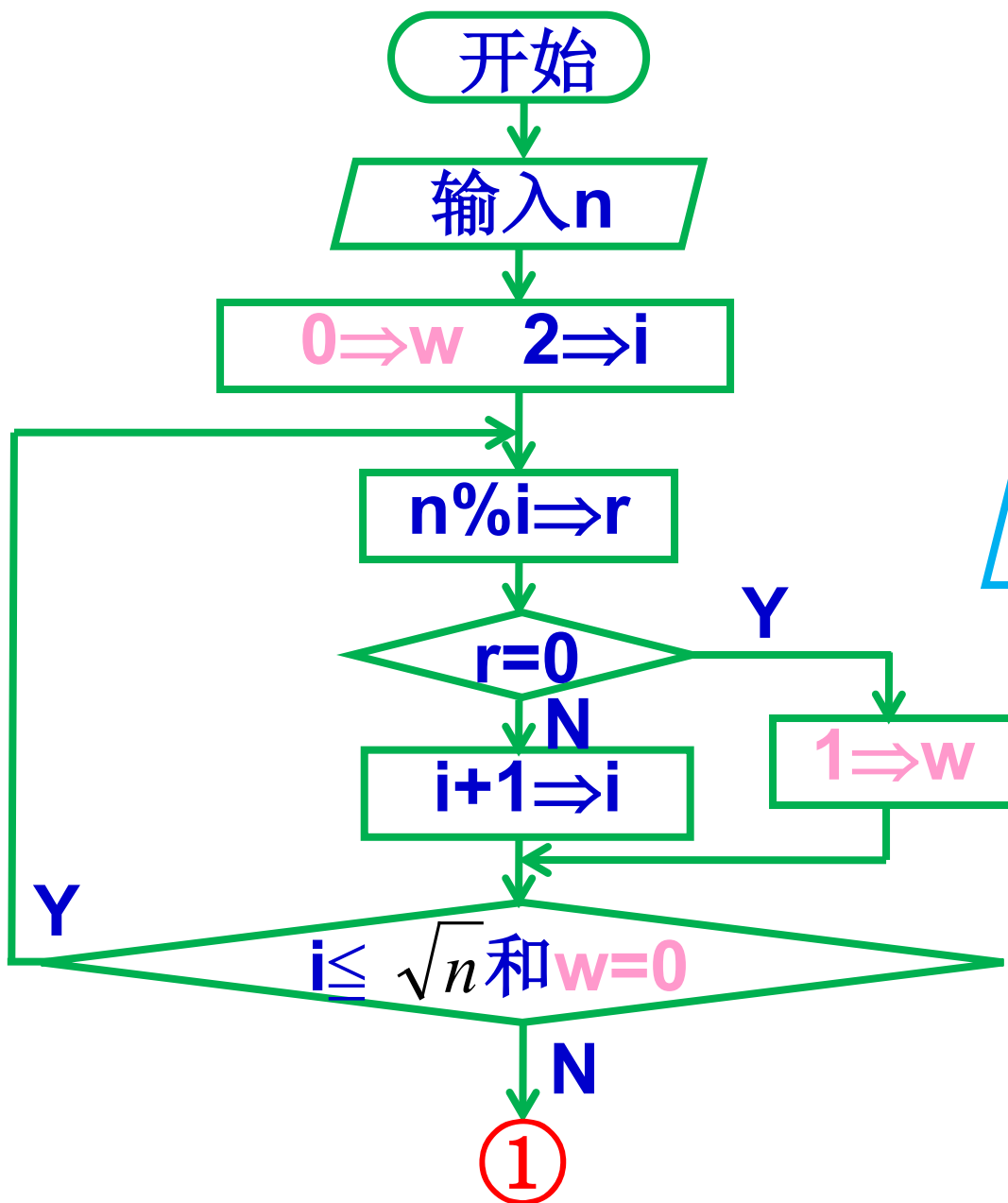
例**2.14** 将例**2.4**
的算法用**N-S**图
表示。求

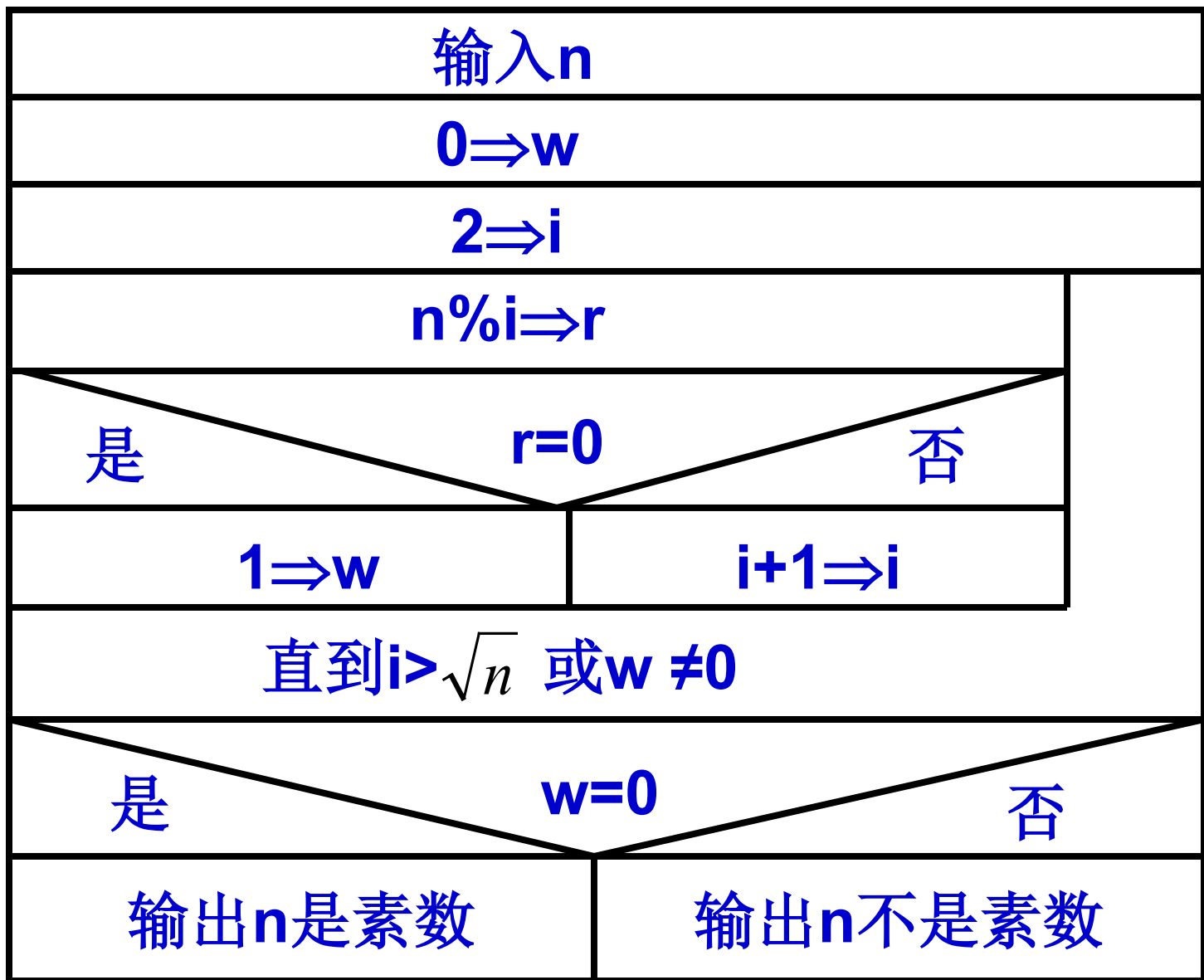
$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$$



例2.15 将例2.5判别素数的算法用**N-S**流程图表示。

- ◆例**2.10**的流程图不是由三种基本结构组成的
- ◆循环有两个出口，不符合基本结构的特点
- ◆无法直接用**N-S**流程图的三种基本结构的符号来表示
- ◆先作必要的**变换**





- 一个结构化的算法是由一些基本结构顺序组成的
- 在基本结构之间不存在向前或向后的跳转，流程的转移只存在于一个基本结构范围之内
- 一个非结构化的算法可以用一个等价的结构化算法代替，其功能不变
- 如果一个算法不能分解为若干个基本结构，则它必然不是一个结构化的算法

2.4.5用伪代码表示算法

- 伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法
- 用伪代码写算法并无固定的、严格的语法规则，可以用英文，也可以中英文混用

例2.16 求5!。

begin

(算法开始)

1 \Rightarrow t

2 \Rightarrow i

while $i \leq 5$

{ $t * i \Rightarrow t$

$i + 1 \Rightarrow i$

}

print t

end

(算法结束)

例**2.17** 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$

begin

1 \Rightarrow sum

2 \Rightarrow deno

1 \Rightarrow sign

while deno \leq 100

{ (-1)*sign \Rightarrow sign

sign*1/deno \Rightarrow term

sum+term \Rightarrow sum

deno+1 \Rightarrow deno

}

print sum

end

2.4.6用计算机语言表示算法

- 要完成一项工作，包括**设计算法**和**实现算法**两个部分。
- 设计算法的目的是为了实现算法。
- 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。

例2.18 将例**2.16**表示的算法（求**5!**）用**C**语言表示。

```
#include <stdio.h>  
int main( )  
{ int i,t;  
    t=1;  
    i=2;  
    while(i<=5)  
    { t=t*i;  
        i=i+1;  
    }  
    printf("%d\n",t);  
    return 0;  
}
```


例2.19 将例**2.17**表示的算法（求多项式

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$$

的值）用**C**语言表示。

```
#include <stdio.h>  
int main( )  
  { int sign=1;  
    double deno = 2.0,sum = 1.0, term;  
    while (deno <= 100)  
    { sign = -sign;  
      term = sign/deno;  
      sum = sum+term;  
      deno = deno+1;  
    }  
    printf ("%f\n",sum);  
    return 0;  
  }
```

2.5结构化程序设计方法

- 结构化程序设计强调程序设计风格和程序结构的规范化，提倡清晰的结构。
- 结构化程序设计方法的**基本思路**是：把一个复杂问题的求解过程分阶段进行，每个阶段处理的问题都控制在人们容易理解和处理的范围内。

2.5结构化程序设计方法

➤采取以下方法保证得到结构化的程序：

- (1) 自顶向下；
- (2) 逐步细化；
- (3) 模块化设计；
- (4) 结构化编码。