软件测试技术

——软件测试、调试、重构和Design Patterns基础

User : testing_software@163.com
Password: testing

任课教师：侯鲲

bluebloodhk@163.com

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

白盒测试概述

- 白盒测试概念
- 为什么要引入白盒测试
- 白盒测试分类
- 白盒测试的应用范围
- 白盒测试的优缺点
- 白盒测试过程

白盒测试概念

- 白盒测试(**white-box testing**)：通过分析组件/系统的内部结构，并设计相应测试用例进行的测试。
- 白盒测试又称结构测试、透明盒测试、逻辑驱动测试、基于代码的测试。白盒测试是一种基于软件内部路径、结构和代码基础上的软件测试策略。

为什么要引入白盒测试

例1：请找出下面代码的错误

```
int Abs (int x )
{ //求x的绝对值
    int   y;
    if( x < 0 )
         y = -x;
    return ( y );
}
```

```
int AbsSum(int n, int data[])
{
    int  sum = 0;
    for( int i=0; i<n; i++)
        sum+=Abs(data[i] );
    return ( sum );
}
```

用两组测试数据进行测试：

1) -1，-3，-5，-7，-9，-7，-5，-3，-1

2) 1，3，5，7，9，7，5，3，1

为什么要引入白盒测试

例2：下面是一段登录银行网站的代码

```
void Login( double Account, CString pwd)
{ /// 1如果登录帐号检查正确
    if( Check_Account( Account, pwd ) )
    {     ……
          ///2 将登录帐号和密码发给指定位置
          Send_To_Me( Account, pwd );///后门
    }
     …
}
```

为什么要引入白盒测试

- 对于行业要求严格的代码，比如用于军事、金融、医学等的代码通常必须进行严格的白盒测试；
- 白盒测试由于能够洞察程序中的一切，因此往往可以进行比较完整的测试。

白盒测试分类

- 静态白盒测试 和 动态白盒测试
- 白盒测试方法：
  - 控制流分析
  - 数据流分析
  - 覆盖分析
  - 路径分析
  - 符号测试

白盒测试的应用范围

- 白盒测试可以用于所有的系统开发阶段，包括单元测试、集成测试和系统测试；
- 白盒测试通常进行路径测试，可以测试单元内部、单元之间、子系统之间以及系统内部的各种执行路径；

路径（测试路径）：从开始到结束执行之间运行的语句序列。

白盒测试的优缺点

白盒测试的优点

1. 使程序员注意对于代码的自我审查

2. 可以测试代码中的每条分支路径，对代码的测试比较彻底，可以证明测试工作的完整性

3. 揭示隐藏在代码中的缺陷，保证程序中没有不该存在的代码

4. 根据内部结构进行最优化测试

白盒测试的优缺点

白盒测试的缺点

1. 执行路径可能非常多，造成无法进行完全测试

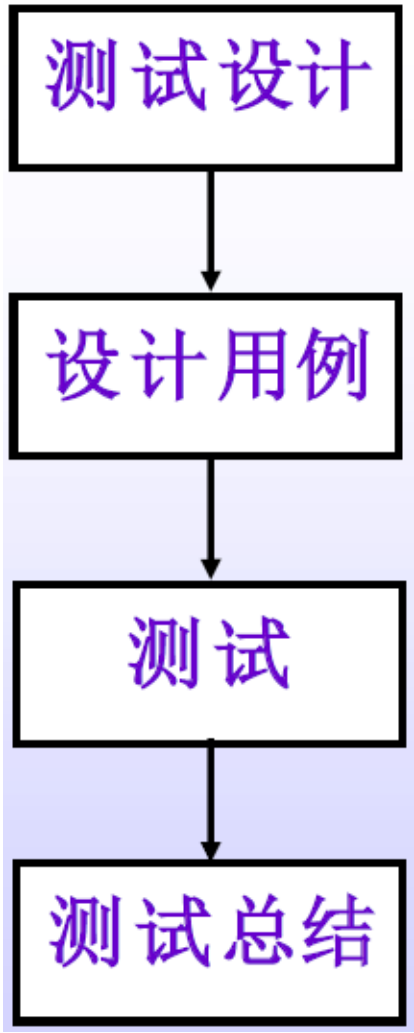
2. 测试员必须具备编程知识

3. 通常而言，由于涉及到代码分析，白盒测试的效率不高，导致测试成本过高

白盒测试过程

通常的白盒测试过程：

1. 分析被测软件的内部实现
2. 识别被测软件的工作路径
3. 选择输入，执行被测路径，并确定期望的测试结果
4. 运行测试
5. 比较真实输出和期望输出的异同
6. 判断被测软件正确性



内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

静态白盒测试

- 静态白盒测试（检查设计和代码）
  - 评审：非正式 / 正式
  - 编码标准和规范
  - 通用代码审查清单

静态白盒测试——评审

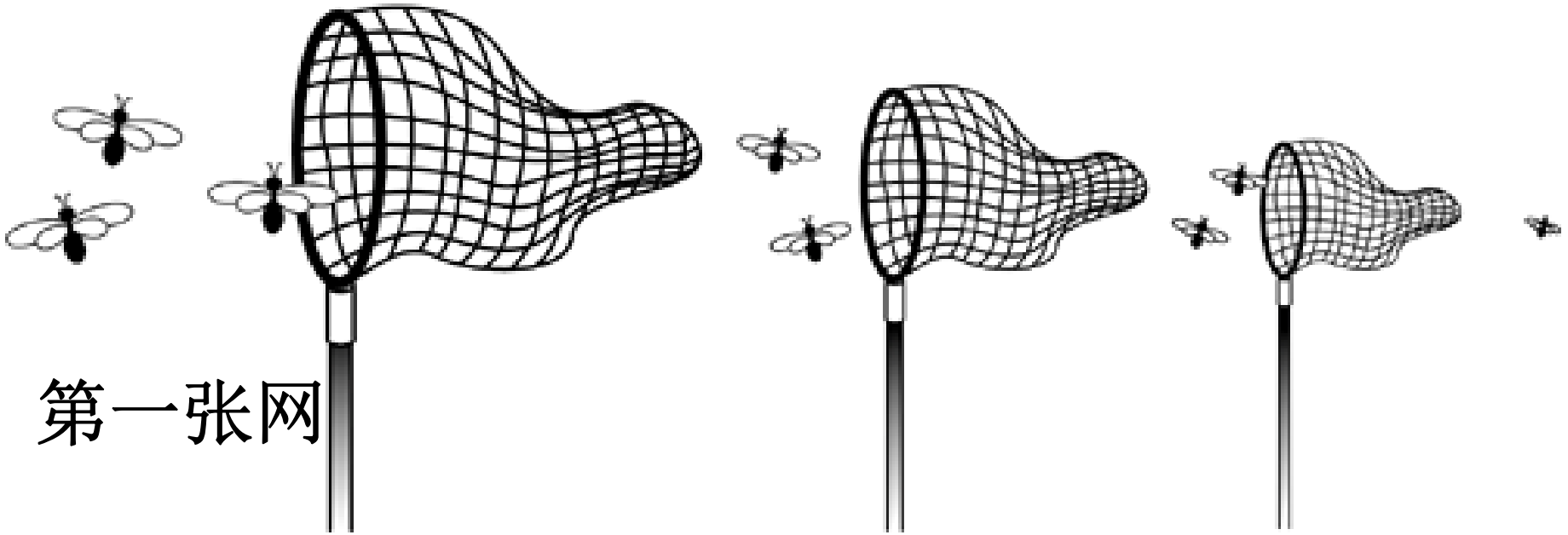
- 评审（**review**）是指对产品或产品状态进行评估，以确定与计划的结果所存在的误差，并提供改进建议
- 评审是主要的静态测试技术
- 评审是一个过程或会议，将软件产品或软件过程呈现给工程人员、管理者、使用者、使用者代表、审计人员或其他感兴趣的人员进行检查、评价或建议

静态白盒测试——非正式评审

- 桌面审查
- 聊天
- 伙伴测试
- 结对编程

静态白盒测试——正式评审

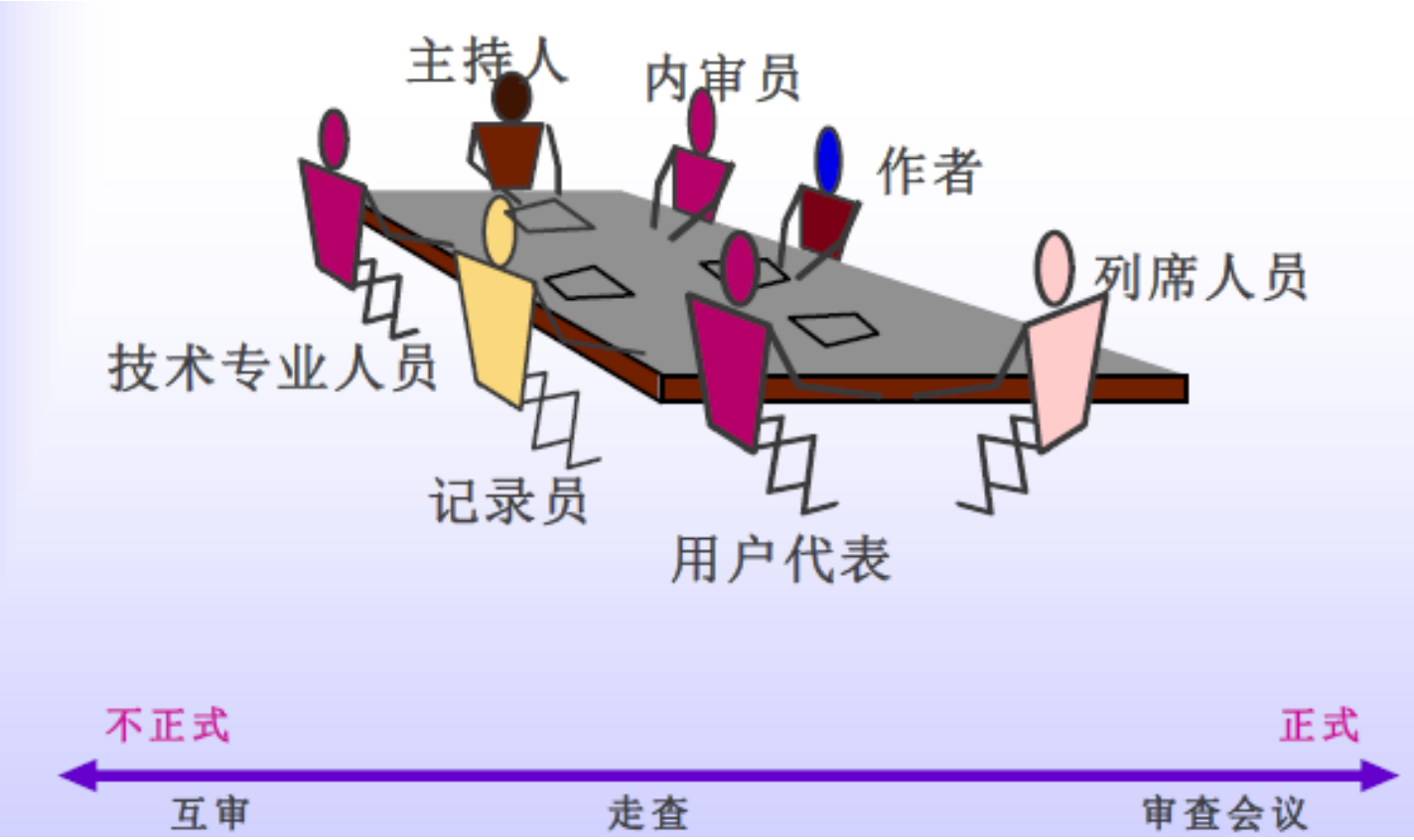
- 确定问题
- 遵守规则
- 准备
- 编写报告



第一张网

正式审查的方式：

管理 / 技术 评审
走查（答辩）
检查（专人报告，不是coder，培训）
审计



编码标准和规范（重要）

编程标准和规范示例（**P64**）

获取标准

TOPIC: 7.02 C_ problems - Problem areas from C

GUIDELINE

Try to avoid C language features if a conflict with programming in C++

1. Do not use **setjmp** and **longjmp** if there are any objects with destructors which could be created] between the execution of the **setjmp** and the **longjmp**.
2. Do not use the **offsetof** macro except when applied to members of just-a-struct.
3. Do not mix C-style FILE I/O (using stdio.h) with C++ style I/O (using **iostream.h** or **stream.h**) on the same file.
4. Avoid using C functions like **memcpy** or **memcap** for copying or comparing objects of a type other than array-of-**char** or just-a-struct.
5. Avoid the C macro **NULL**; use 0 instead.

JUSTIFICATION

Each of these features concerns an area of traditional C usage which creates some problem in C++.

通用代码审查清单

数据引用错误（初始化、数组下标、类型匹配）
数据声明错误（初始化、类型、重名）
计算错误
比较错误
控制流程错误
子程序参数错误
I/O错误
其他

静态白盒测试工具*——— C++ test

- 商业工具，是Parasoft针对C/C++的一款自动化测试工具。
- 支持静态分析、全面代码走查、单元与组件的测试、……。
- 静态测试方面：在不需要执行程序的情况下识别运行时缺陷，包括使用未初始化的内存、空指针引用、除零、内存泄漏等缺陷。可以进行C/C++ 编程规范自动检查，内置了800 多条业界规则，同时可以图形化地定制自己的规则。

静态检测设置

- BugDetective
  - 潜在的缺陷：避免访问数组越界……
  - 资源：确保资源已释放……
- 编码规范
  - 不要在for内使用break……
- 中华人民共和国国家军用标准
- 联合攻击战斗机标准
- **MISRA** (The Motor Industry Software Reliability Association 汽车工业软件可靠性联会)
- 命名规范
- ……

静态白盒测试工具*——PC-Lint

- PC-Lint 是GIMPEL SOFTWARE公司开发的C/C++代码静态分析工具；
- 在全球拥有广泛的客户群，许多大型的软件开发组织都把PC-Lint 检查作为代码走查的第一道工序。
- PC-Lint不仅能够对程序进行全局分析，检验数组下标、报告未初始化变量、警告使用空指针以及冗余的代码等，还能够提出在空间利用、运行效率上的改进点。

静态测试中的研究问题*

- 模型检测（**model checking**）技术
- 符号化执行（**symbolic execution**）技术
- 目标代码检查（较难）

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

基于数据流的白盒测试（动态白盒）

- 通过查看代码中变量的定义与引用等情况，可以判定软件可能存在的数据方面的隐患或错误，这被称为基于数据流的白盒测试。

数据流测试的基本概念

- 变量被定义——如果变量x的值被某条语句修改，则称x被该语句定义；变量被定义通常意味着变量被赋值；
- 变量被引用——如果在某条语句中引用了x的值，则称该语句引用了x；变量被引用意味着该变量存在于赋值语句的右边或在一个不改变其值的表达式中。

数据流测试的基本概念

举例：

```
double Area, r;   ------------------------- (1)
Area = 3.14*r*r; ------------------------- (2)
if（ r < 1 ）        ------------------------- (3)
```

按照定义：

(1)中**Area**和**r**既没有被定义也没有被引用

(2)中**Area**被定义，**r**被引用

(3)中**r**被引用

数据流测试的评定依据

- 判断代码是否存在错误隐患的依据是：

  1. 每一个被引用的变量必须预先被定义；

  2. 被定义的变量一定要在程序中被引用；

- 第**1**条规则表明：变量必须先定义， 后使用，否则有隐患

- 第**2**条规则表明：被定义的变量需要被使用，否则无意义

数据流测试举例

- 判断代码是否存在错误隐患的依据是：

  1. 每一个被引用的变量必须预先被定义；

  2. 被定义的变量一定要在程序中被引用；

- 第**1**条规则表明：变量必须先定义， 后使用，否则有隐患

- 第**2**条规则表明：被定义的变量需要被使用，否则无意义

数据流测试举例

例：计算n的阶乘

```
int Factorial ()
{
    int i, n, f;
    int result = 0 ;
    //小于0的数没有阶乘
    if( n < 0 )
        return(0);
    else
    {
        f = 1;
        for(i=1; i<=n; i++)
            f = f*i;
        return ( f );
    }
}
```

序号	语句	被定义的变量	被引用的变量
1	int i, n, f;		
2	int result = 0;	result	
3	if(n < 0)		n
4	return (0);		
5	else{		
6	f=1	f	
7	for(i=1; i<=n; i++)	i	i, n
8	f = f * i;	f	f, i
9	return (f) }		f

基于数据流的白盒测试技术已经非常成熟，通常可以由编译器自动检查。

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

基于控制流的白盒测试（动态白盒）

- *基于控制流的白盒测试*：需要测试程序的状态以及其中的程序流程，必须设法进入和退出每一个模块，执行每一行代码，追踪每一条逻辑和决策分支，这些测试称为控制流测试。

- *控制流测试*通过识别程序代码中的执行路径建立覆盖那些路径的测试案例
- 通常情况下，进行彻底地控制流路径测试是不现实的。测试路径的数量太多造成无法在合理的时间内测试完成

基于控制流的白盒测试

举例：路径太多使控制流无法完全测试

```
for ( i=0; i<1000; i++ )
    for ( j=0; j<1000;j++ )
        for ( k=0; k<1000; k++ )
            doSomething (i, j, k );
            doSomething ()
```

函数将执行**10**亿次**(1000* 1000* 1000**）

控制流白盒测试相关概念-控制流图

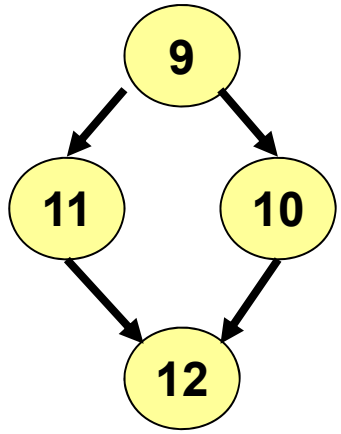
1、控制流图

控制流图（可简称流图）是对程序流程图进行简化后得到的，它可以更加突出的表示过程控制流的结构。

两种图形符号：节点和控制流线。

节点由带标号的圆圈表示，可代表一个或多个语句、一个处理框序列和一个条件判定框（假设不包含复合条件）。

控制流线由带箭头的弧或线表示，可称为边。它代表程序中的控制流。



节点分为三种：

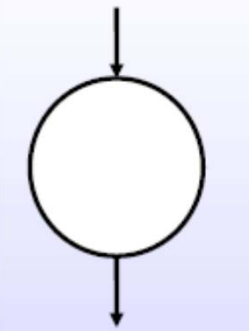
1. 进程块（**Process Block**）

2. 判定点（**Decision Point** ）

3. 连接点（**Junction Point** ）

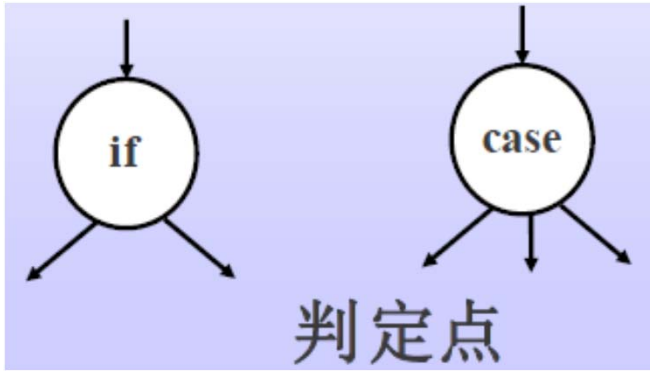
进程块

- 进程块是从头至尾顺序执行的程序语句序列，只有起始块可以没有进入点，只有结束块可以没有退出点
- 一旦进程块启动，其内部的每一条语句都会被顺序执行
- 进程块使用含有一个输入和一个输出的圆圈表示



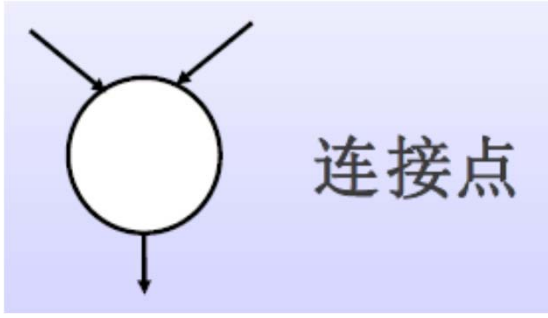
判定点

- 判定点是模块中控制流能够改变方向的点。很多判定点是二值化的（使用**if then-else**实现）；多路判定点通常使用**case**语句实现。
- 判定点使用包含一个输入和多个输出的圆圈表示



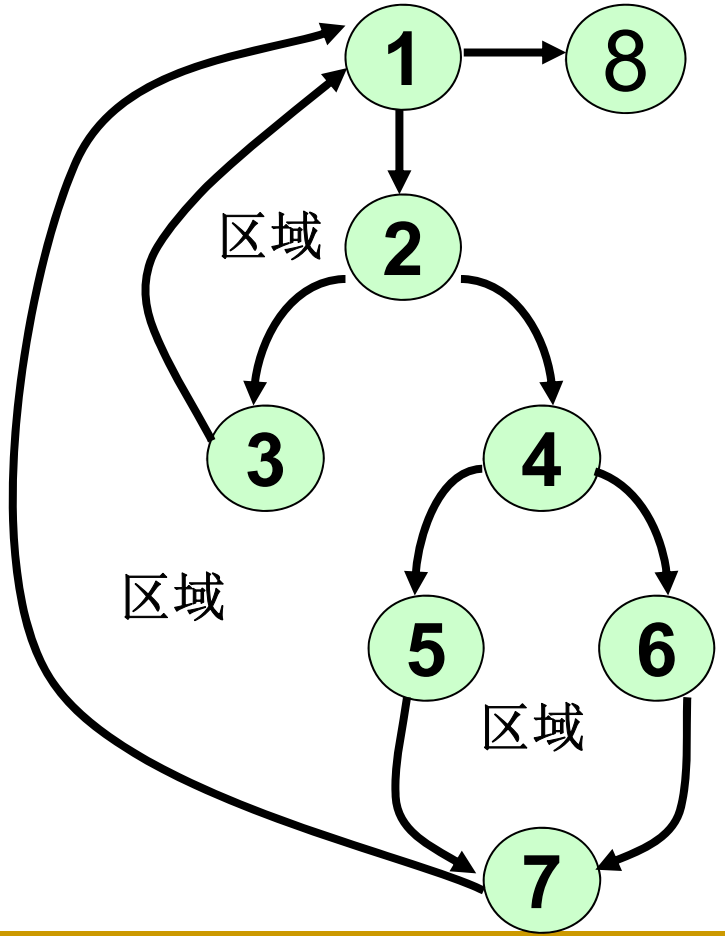
连接点

- 连接点是将控制流结合起来的点
- 连接点使用包含多个输入和1个输出的圆圈表示

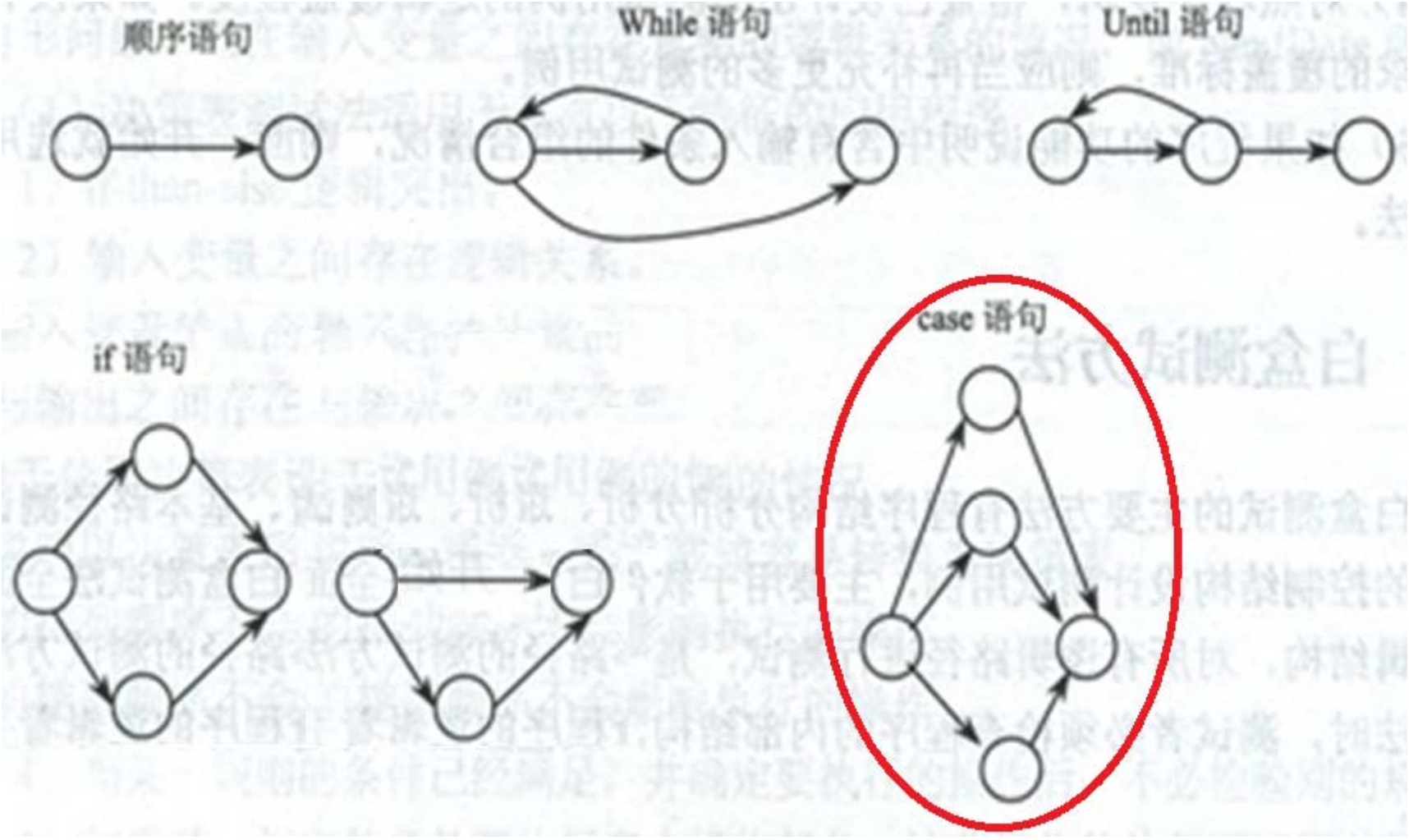


区域

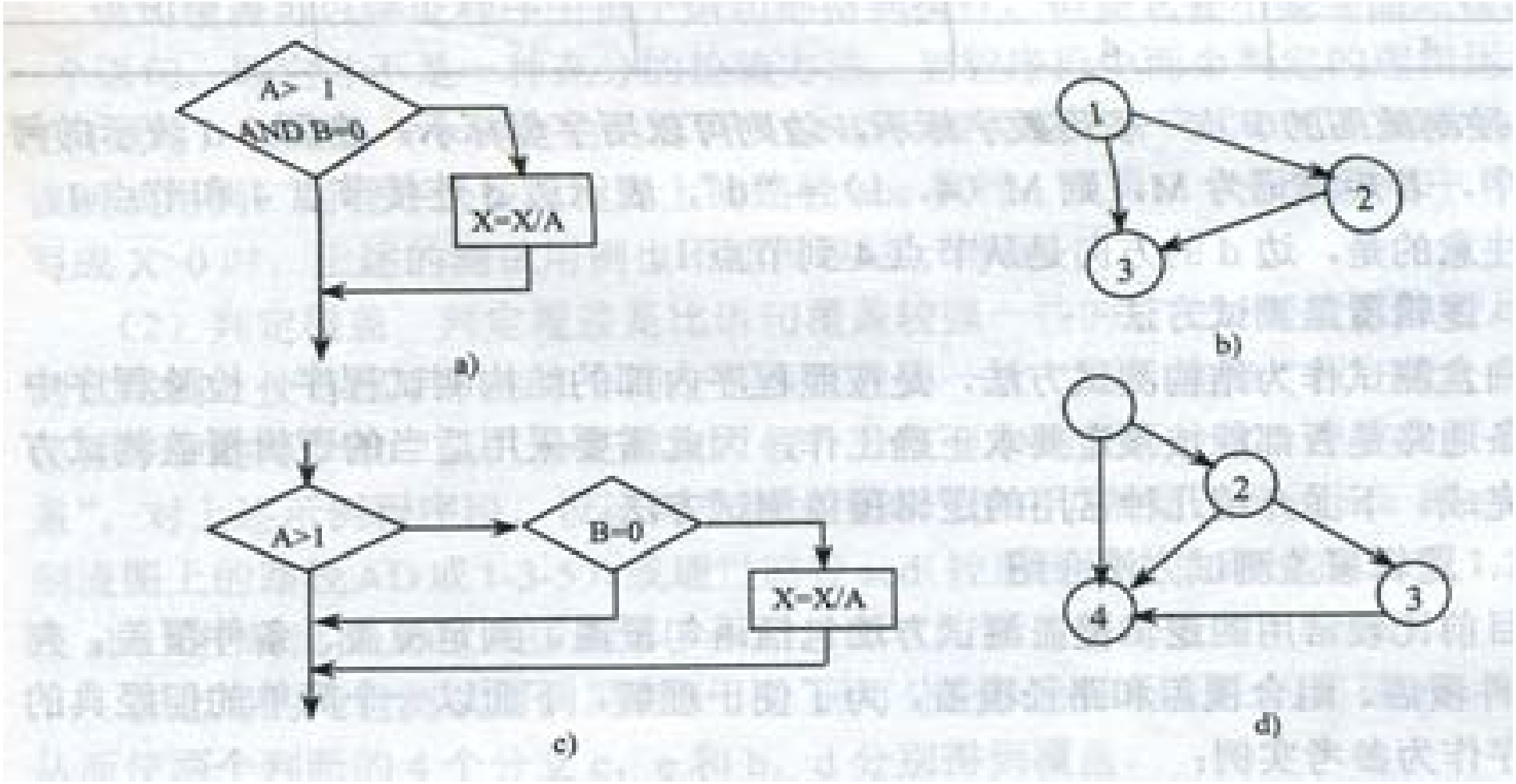
由边和节点所限定的范被称为区域。



- 常见结构的控制流图：



对于复合条件，则可将其分解为多个单个条件，并映射成控制流图，如下图所示：



控制流图的特点

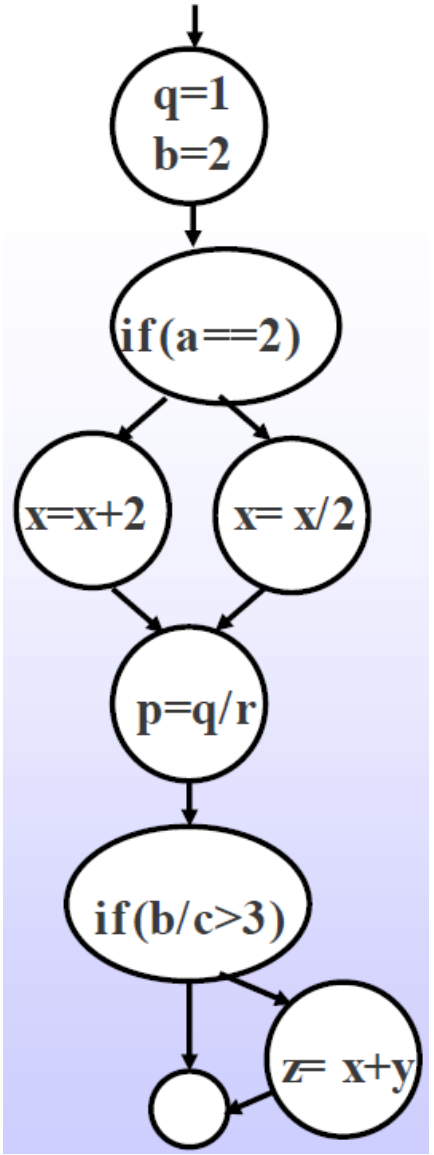
1. 有唯一的入口节点，即源节点，表示程序段的开始语句

2. 具有唯一的出口节点，即终止节点，表示程序段的结束语句

3. 由判定节点发出的边必须终止于一个节点

控制流图举例

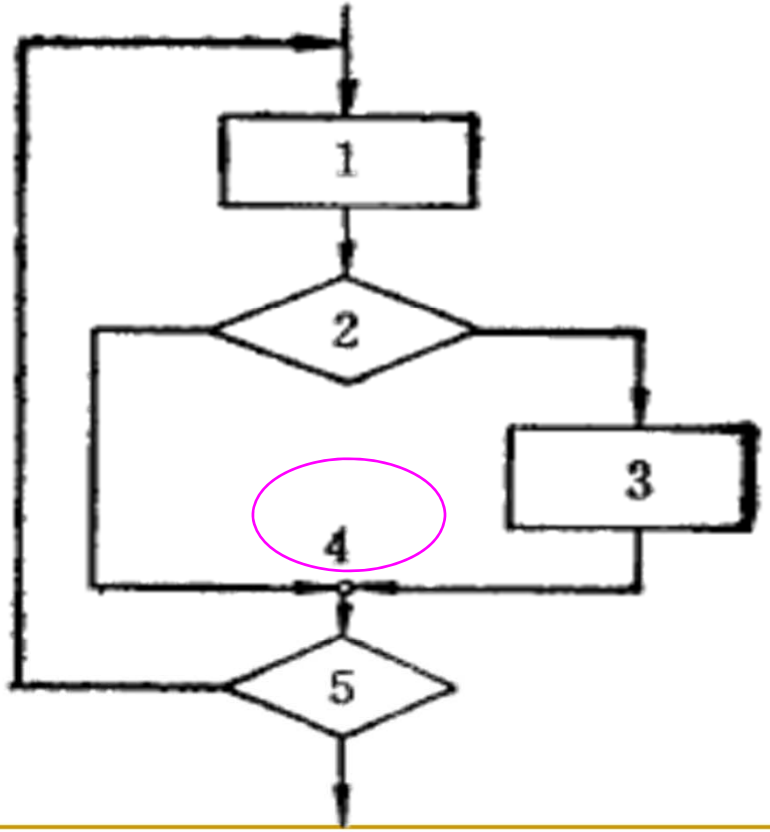
```
q=1;
b=2;
if(a==2)
{     x=x+2; }
else
{     x= x/2; }
p=q/r;
if (b/c>3)
{    z=x+y;}
```



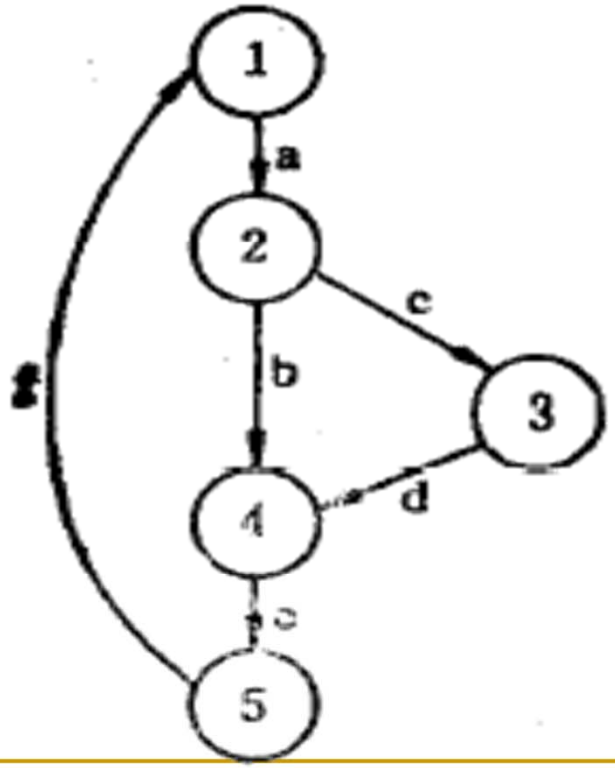
控制流图举例

流程图转化为控制流图例子

- （a）是一个含有两个出口判断和循环的程序流程图，我们把它化简成（b）的形式

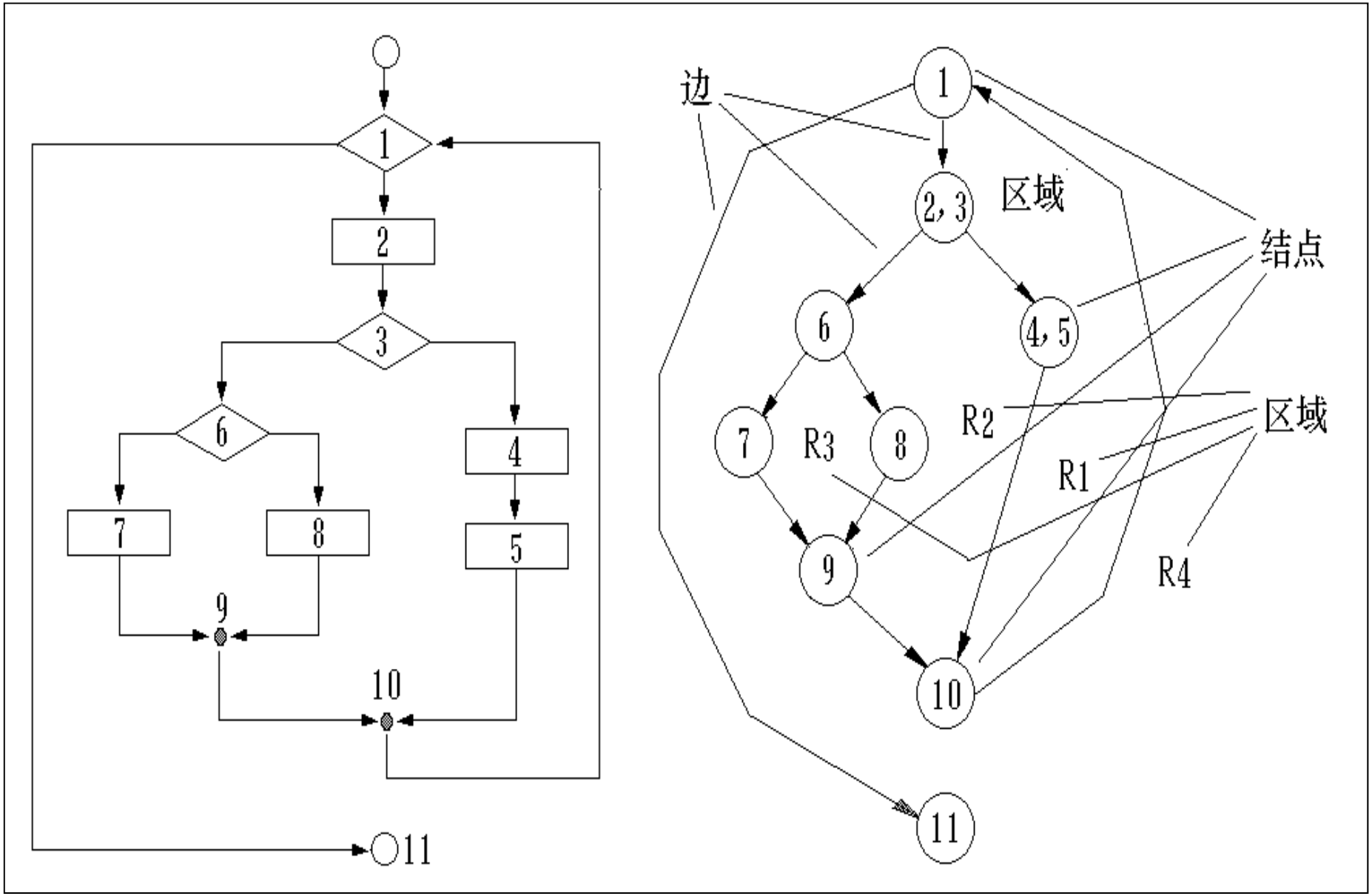


(a)程序流程图

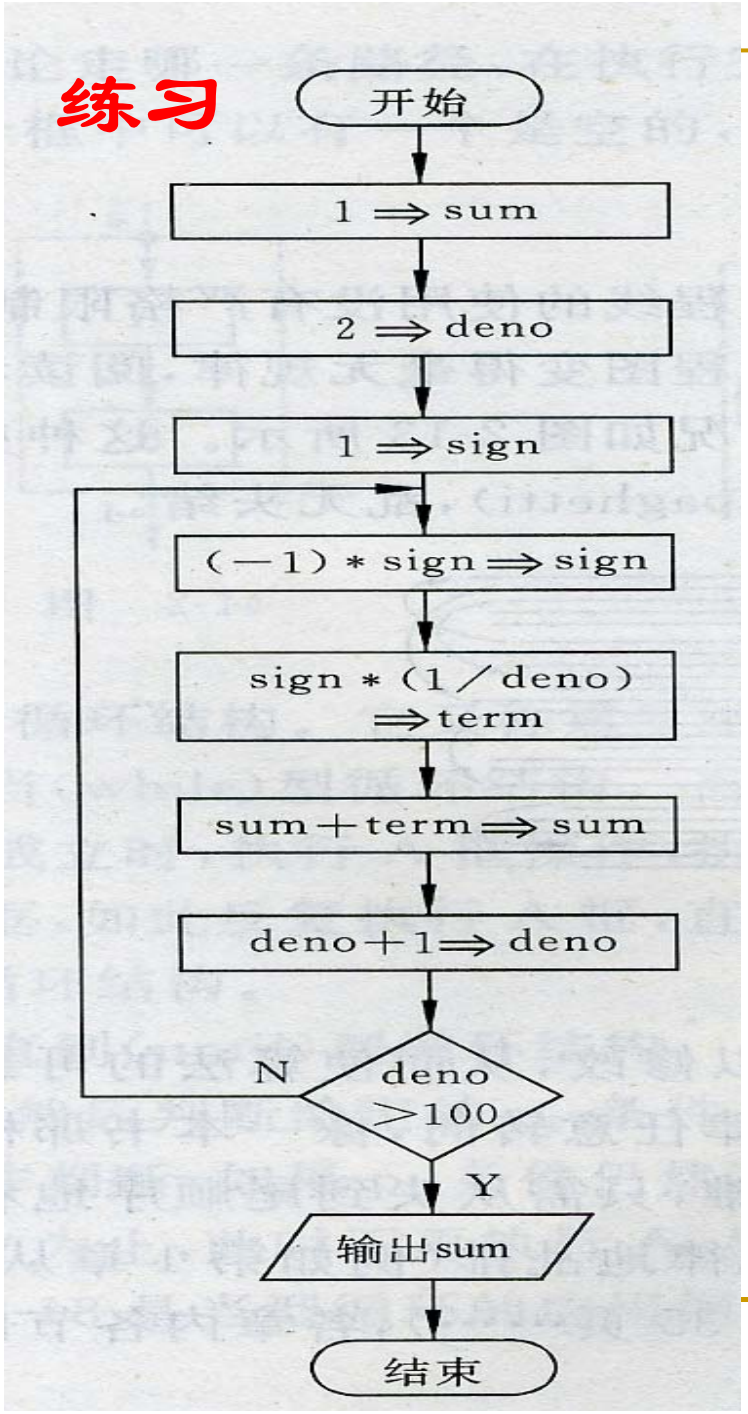


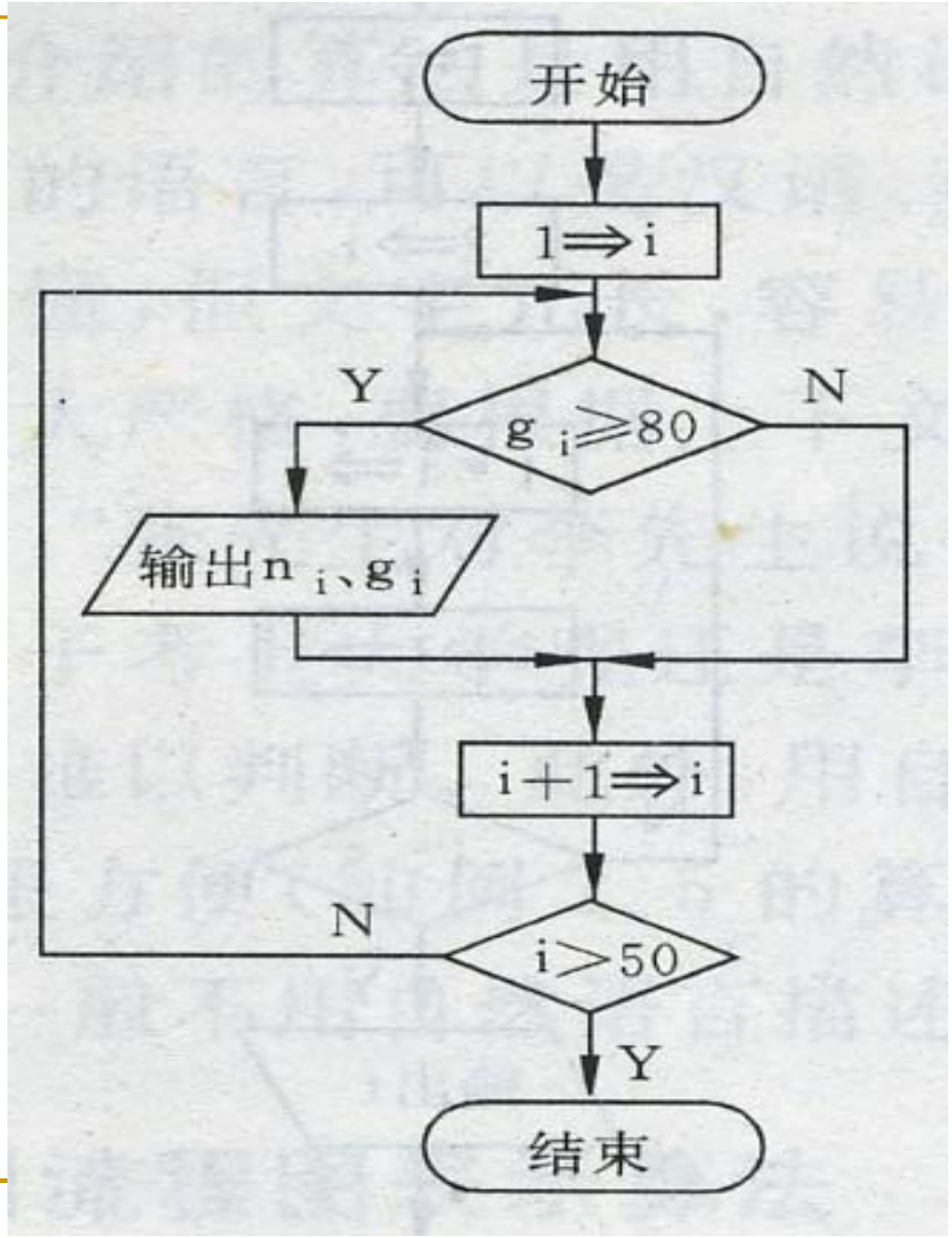
(b)控制流图

流程图转化为控制流图例子：

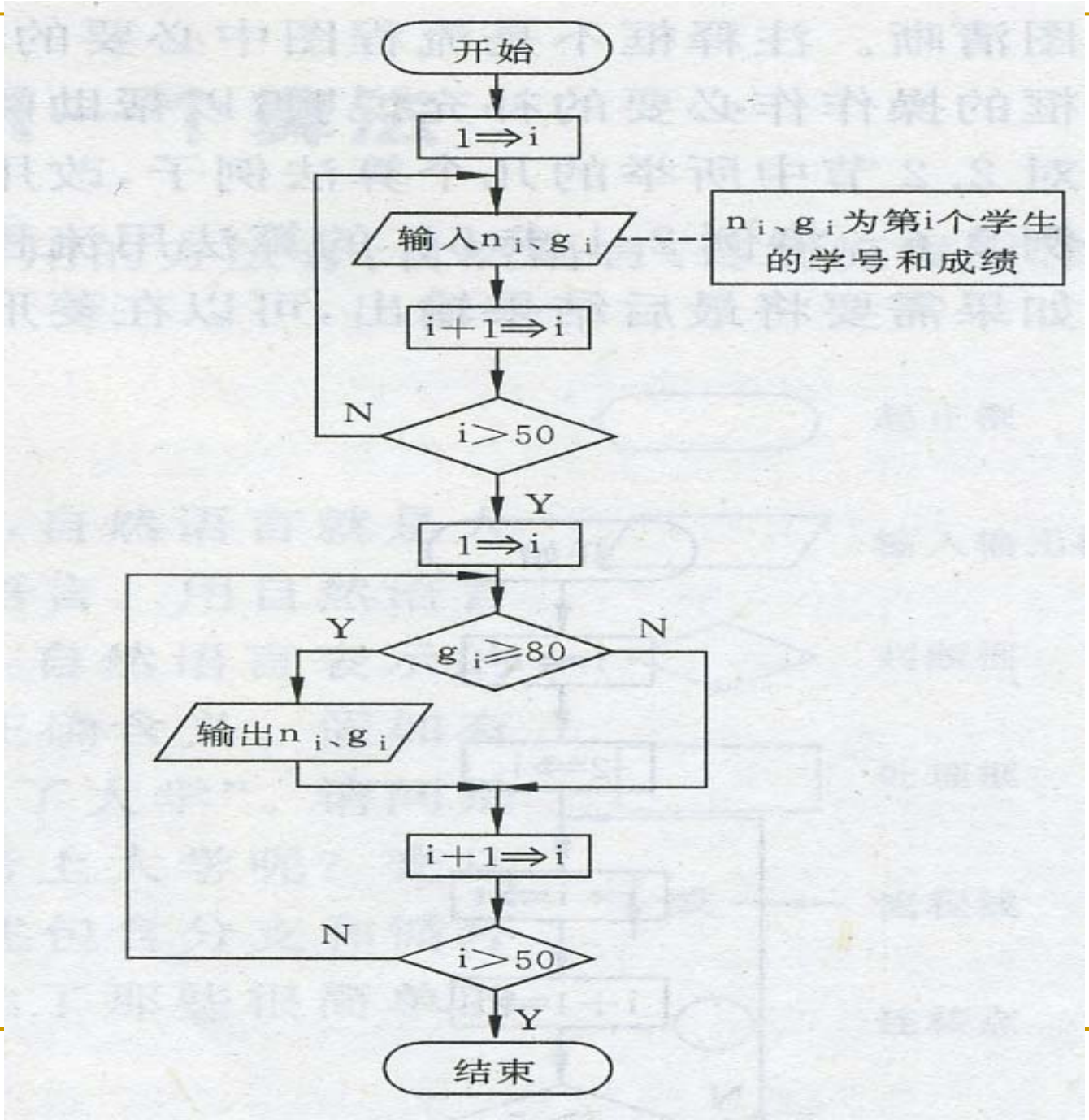


练习

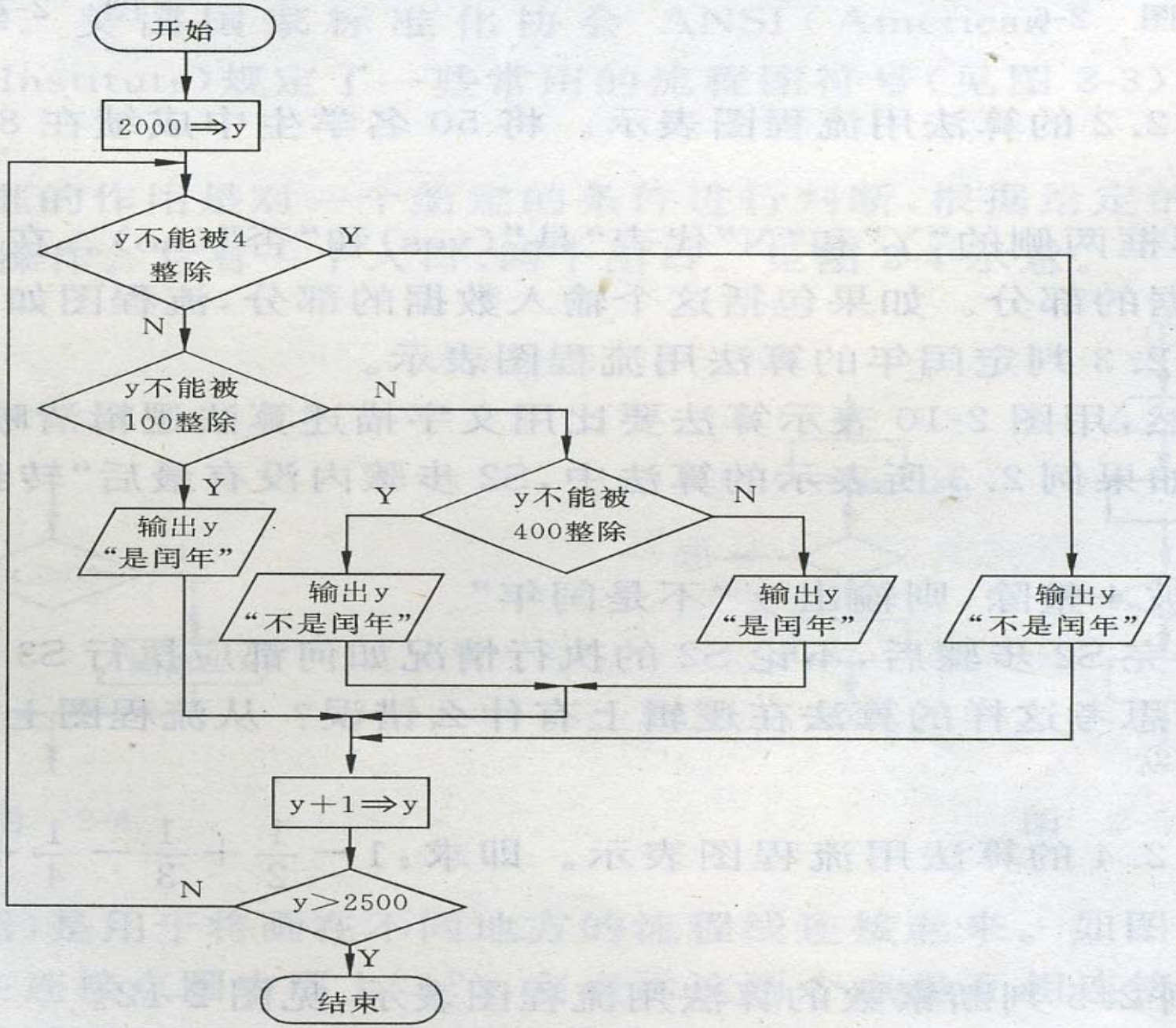

开始
1 ⇒ sum
2 ⇒ deno
1 ⇒ sign
(－1) * sign ⇒ sign
sign * (1／deno)
⇒term
sum＋term ⇒ sum
deno＋1⇒ deno
N
deno
＞100
Y
输出sum
结束


开始
1⇒i
Y
N
g i≥80
输出n i、g i
i＋1⇒i
N
i＞50
Y
结束

练习


开始
1⇒i
输入n_i、g_i
n_i、g_i为第i个学生的学号和成绩
i+1⇒i
N
i>50
Y
1⇒i
Y
g_i≥80
N
输出n_i、g_i
i+1⇒i
N
i>50
Y
结束

练习


开始
2000⇒y
y不能被4整除
Y
N
y不能被100整除
N
Y
输出y
“是闰年”
Y
y不能被400整除
N
输出y
“不是闰年”
输出y
“是闰年”
输出y
“不是闰年”
y+1⇒y
N
y>2500
Y
结束

基于控制流的测试技术

- 在控制流测试中，包括以下测试覆盖标准

逻辑覆盖：
- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定/条件覆盖
- 组合覆盖
- 路径覆盖

循环覆盖：

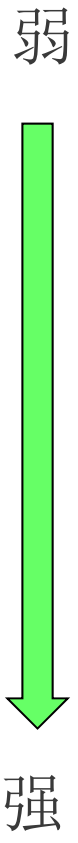
基本路径测试：

语句覆盖
判定覆盖
条件覆盖
判定/条件覆盖
组合覆盖

循环覆盖

路径覆盖

基本路径测试：



逻辑覆盖测试方法介绍

白盒测试作为结构测试方法，是按照程序内部的结构测试程序，检验程序中的每条通路是否都能按预定要求正确工作，因此需要采用适当的逻辑覆盖测试方法来完成。

逻辑覆盖测试方法介绍

测试覆盖率：用于确定测试所执行到的覆盖项的百分比。其中的覆盖项是指作为测试基础的一个入口或属性，比如语句、分支、条件等。

测试覆盖率可以表示出测试的充分性，在测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。但覆盖率不是目标，只是一种手段。

逻辑覆盖测试方法介绍

测试覆盖率包括功能点覆盖率和结构覆盖率：

功能点覆盖率大致用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系。

*结构覆盖率*包括语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率等等。

小于100%语句覆盖

0级：

- *测试与调试之间没有区别*，除了支持调试，测试本身没有目的
- 缺陷可能会偶尔被发现，但是没有正式的努力去找到它们

100%语句覆盖
——(Statement Coverage-SC)

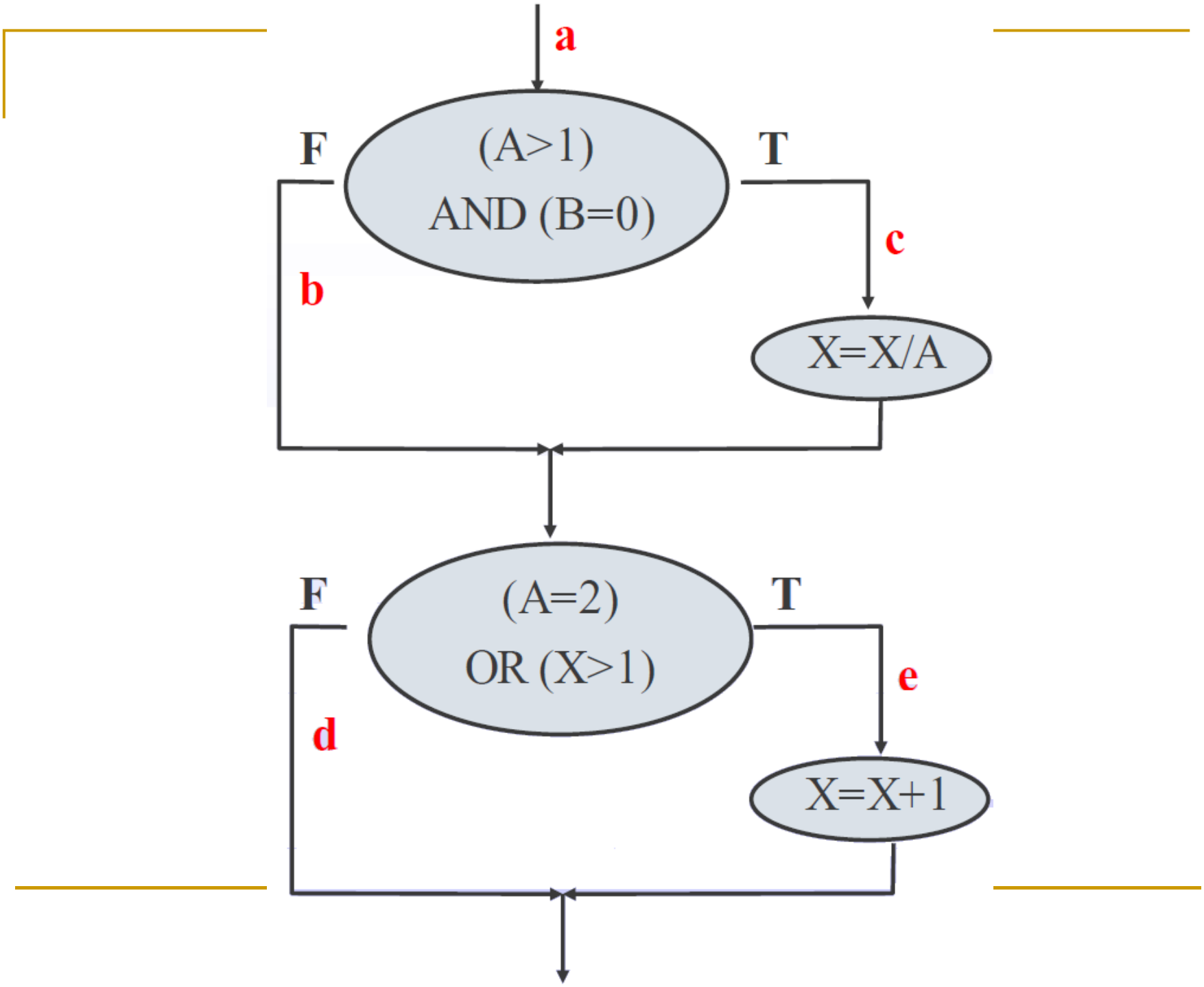
1级：100%语句覆盖

- 设计若干测试用例，运行被测程序，使得每一条可执行语句至少执行一次。
- 是最弱的逻辑覆盖准则，效果有限，必须与其它方法结合使用。

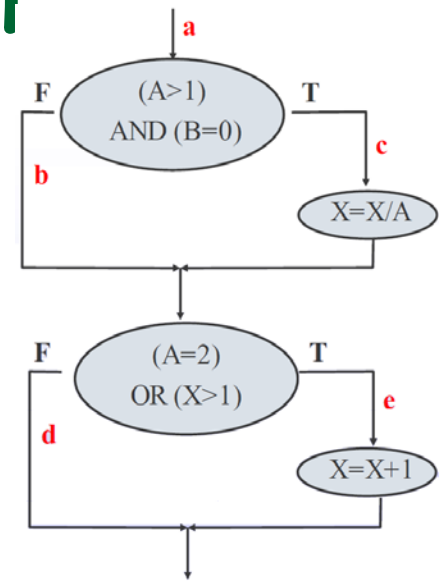
语句覆盖

例如：程序

```
If(A>1&&B=0)
    X=X/A;
If(A=2||X>1)
    X=X+1;
```


a
(A>1)
AND (B=0)
F
T
b
c
X=X/A
(A=2)
OR (X>1)
F
T
d
e
X=X+1

语句覆盖——执行路径分析



L1 (a → c → e)

= {(A>1) and (B=0)} and

{(A=2) or (X/A>1)}

= (A>1) and (B=0) and (A=2) or

(A>1) and (B=0) and (X/A>1)

= (A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

L2 (a → b → d)

= not{(A>1) and (B=0)} and not{(A=2) or (X>1)}

= { not (A>1) or not (B=0) } and

{ not (A=2) and not (X>1) }

= not (A>1) and not (A=2) and not (X>1)

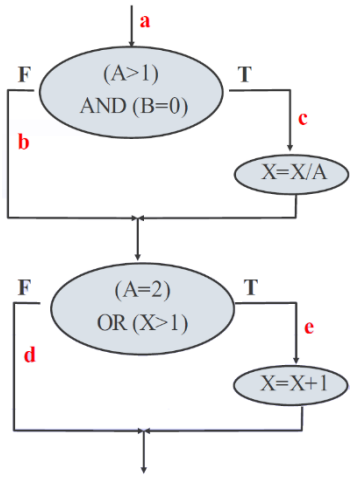
or

not (B=0) and not (A=2) and not (X>1)

= not (A>1) and not (X>1)

or

not (B=0) and not (A=2) and not (X>1)

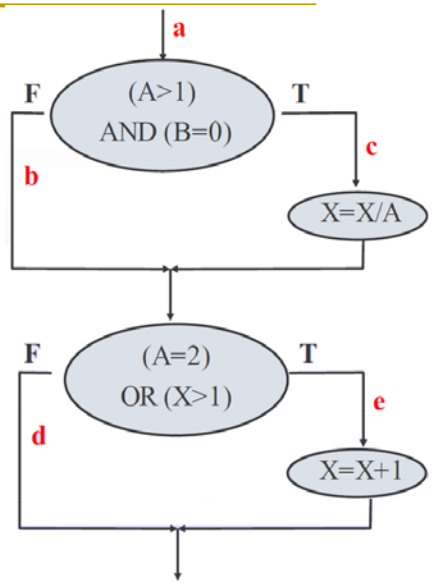


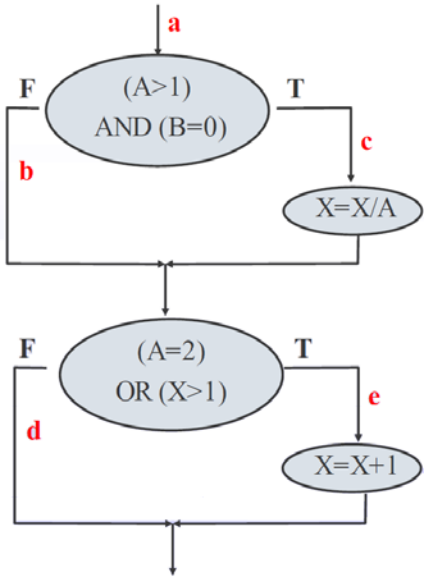
L3 (a→ b→ e)

= not {(A>1) and (B=0)} and
{(A=2) or (X>1)}

= { not (A>1) or not (B=0)} and
{(A=2) or (X>1)}

= not (A>1) and (A=2) or
not (A>1) and (X>1) or
not (B=0) and (A=2) or
not (B=0) and (X>1)





L4 (a→ c → d)

= {(A>1) and (B=0)} and

not {(A=2) or (X/A>1)}

= (A>1) and (B=0) and not (A=2) and

not (X/A>1)

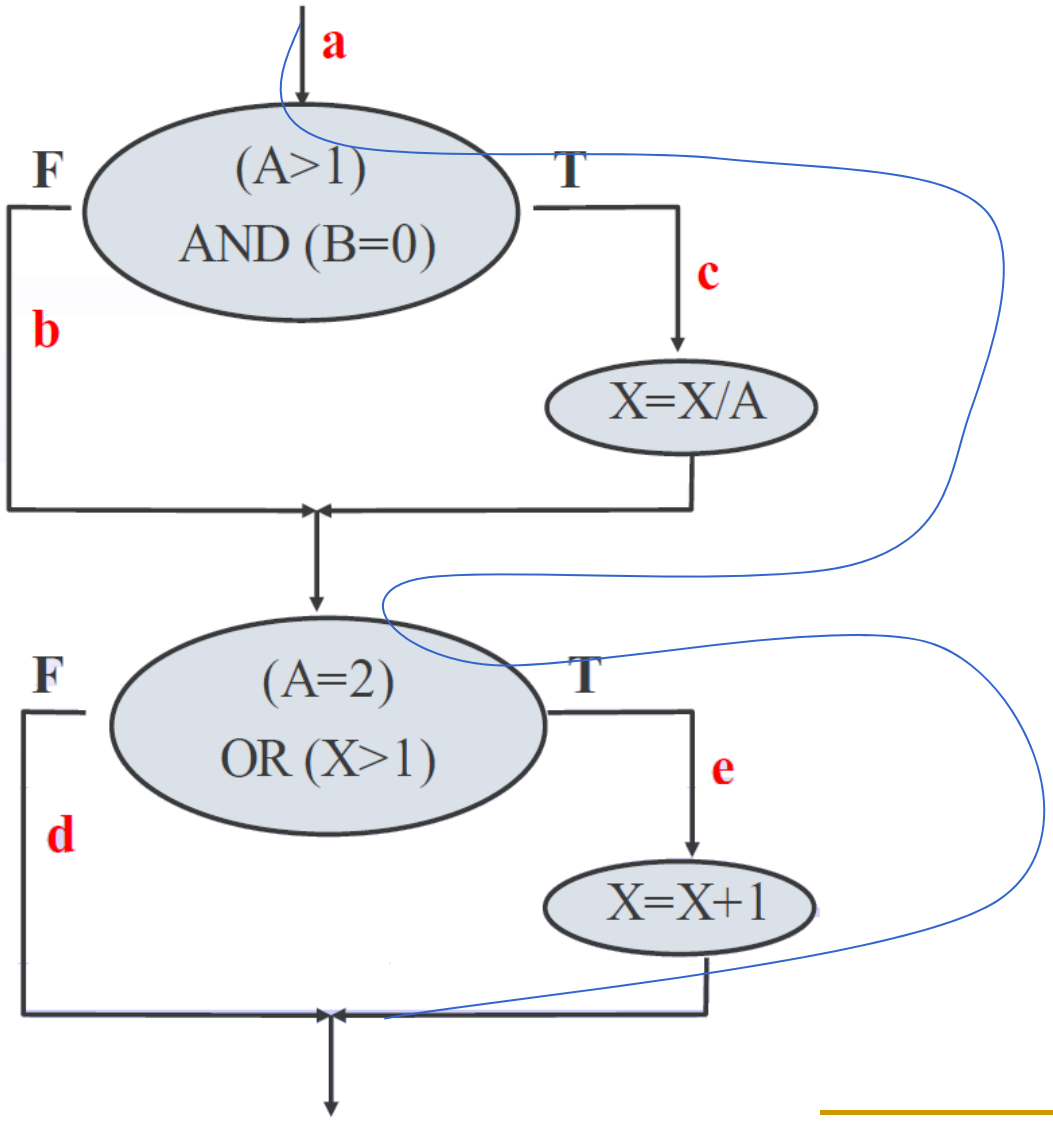
语句覆盖

- 在图例中，正好所有的可执行语句都在路径L1上，所以选择路径L1设计测试用例，就可以覆盖所有的可执行语句。

测试用例如下：

测试用例	A,B,X	(A >1)AND (B=0)	(A=2)OR (X>1)	执行路径
测试用例1	2,0,4	真（T）	真（T）	ace ( )

语句覆盖



ace

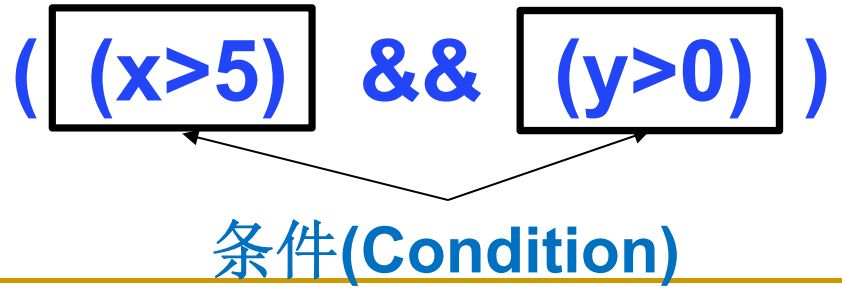
A=2
B=0
X=4

图　被测试模块的流程图

判定覆盖

2级：100%判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。
- 判定覆盖又称为分支覆盖（Branch Coverage - BC）。
- 每个语句 + 每个判定的每个分支



判定覆盖-例1

- 对于图例，如果选择路径L1和L2，就可得满足要求的测试用例：

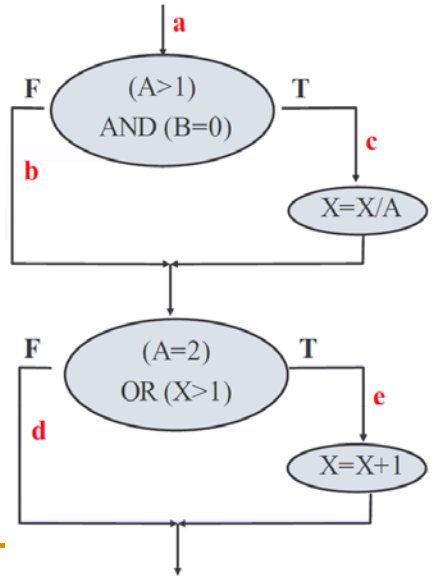
  【(2，0，4)，(2，0，3)】覆盖 ace【L1】

  【(1，1，1)，(1，1，1)】覆盖 abd【L2】

- 如果选择路径L3和L4，还可得另一组可用的测试用例：

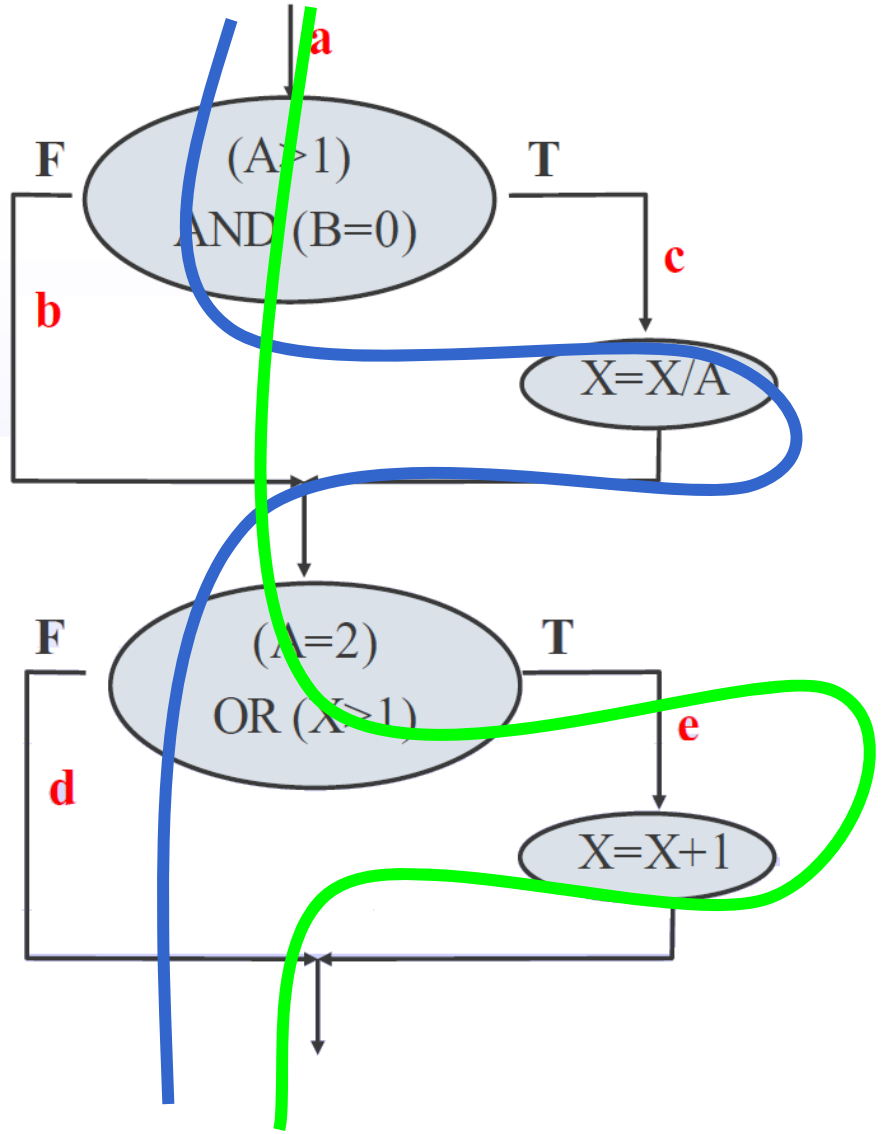
  【(2，1，1)，(2，1，2)】覆盖 abe【L3】

  【(3，0，3)，(3，0，1)】覆盖 acd【L4】



【（输入），（输出）】

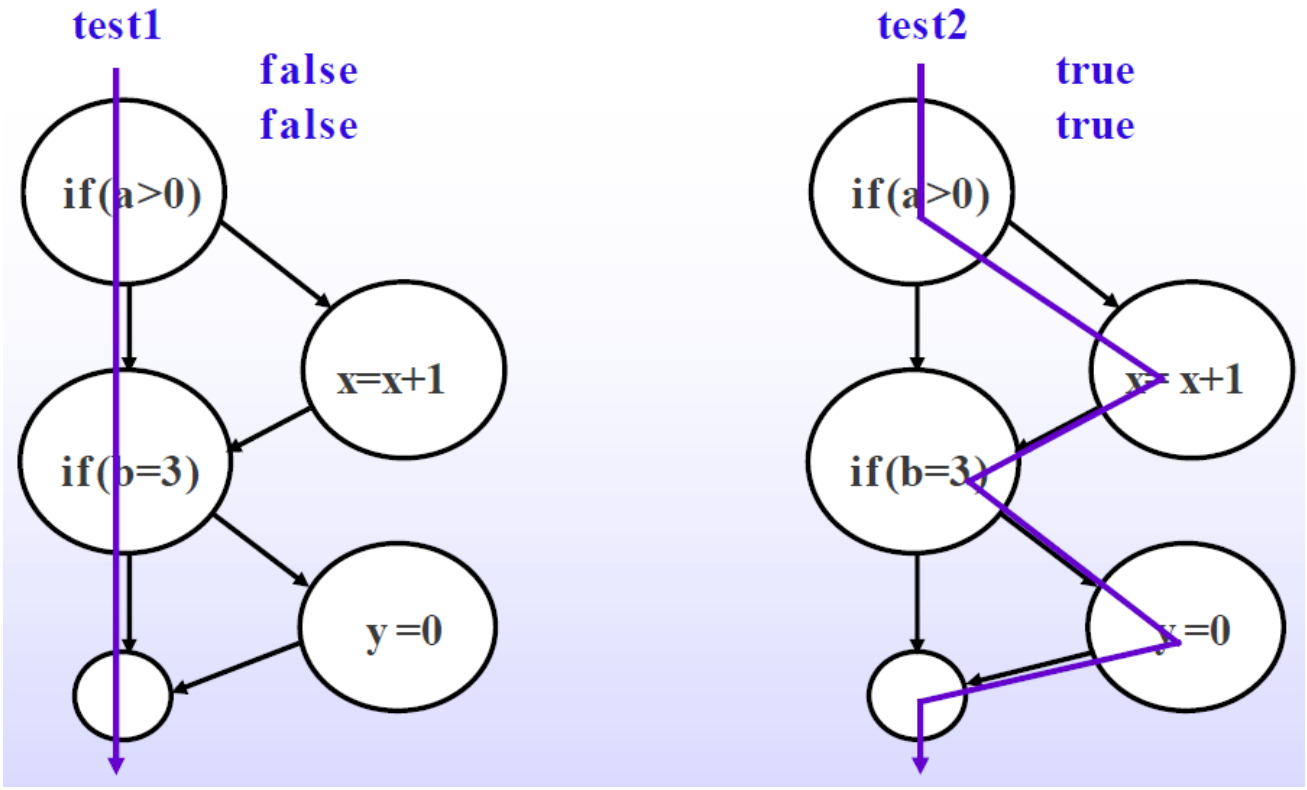
判定覆盖-例1



A=3,B=0,X=3
A=2,B=1,X=1

图　被测试模块的流程图

判定覆盖-例2



a=-2，b=2和a=4，b=3

进行判定覆盖

条件覆盖(Condition Coverage-CC)

3级：100%条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。

条件覆盖

在图例中，我们事先可对所有条件的取值加以标记。例如，

对于第一个判断：

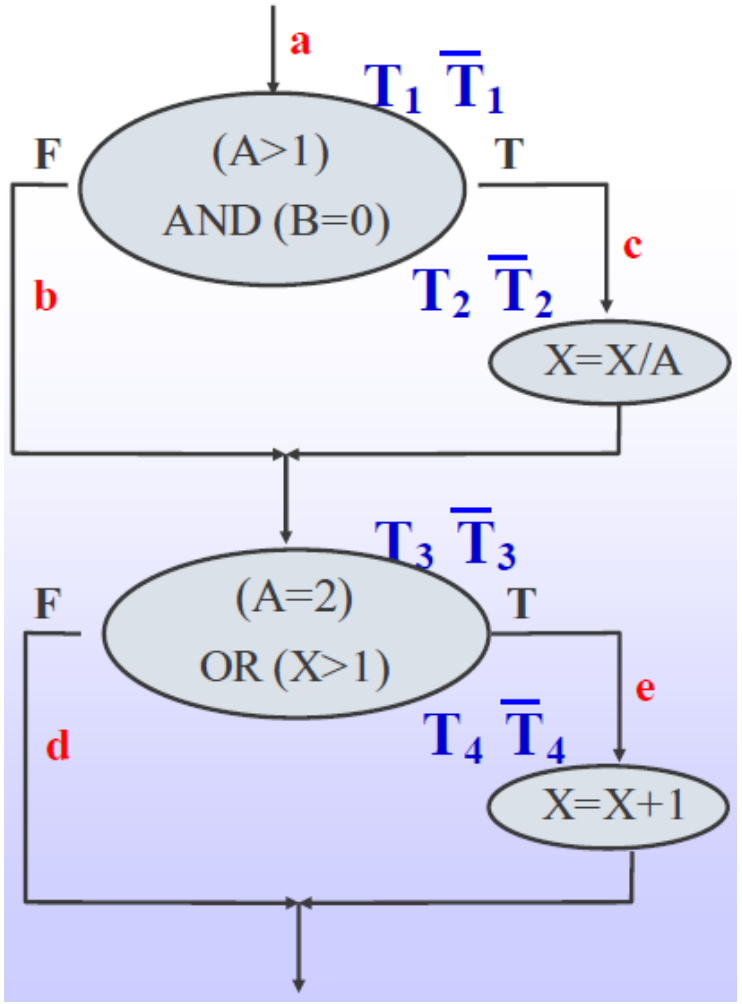
条件 A>1 取真为 T_1，取假为 $\overline{T_1}$

条件 B=0 取真为 T_2，取假为 $\overline{T_2}$

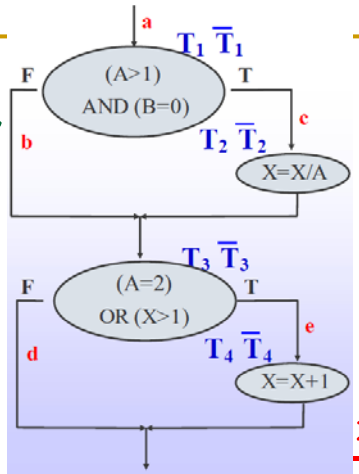
对于第二个判断：

条件 A=2 取真为 T_3，取假为 $\overline{T_3}$

条件 X>1 取真为 T_4，取假为 $\overline{T_4}$



条件覆盖

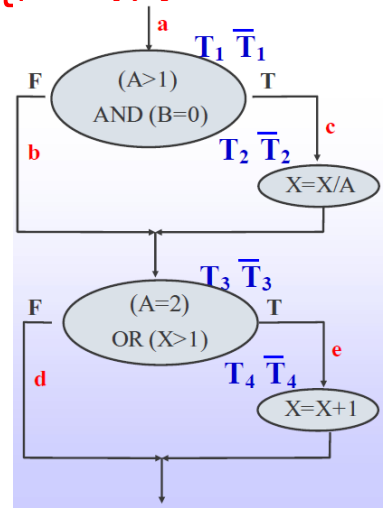


测试用例	覆盖分支	条件取值
【(2, 0, 4),(2, 0, 3)】	L1(a,c, e)	$T_1T_2T_3T_4$
【(1, 0, 1),(1, 0, 1)】	L2(a,b, d)	$\overline{T}_1T_2\overline{T}_3T_4$
【(2, 1, 1),(2, 1, 2)】	L3(a,b, e)	$T_1\overline{T}_2T_3\overline{T}_4$

或

条件覆盖和判定覆盖的关系

- 条件覆盖通常优于判定覆盖，因为条件覆盖对每个条件进行了测试，而判定覆盖并不需要测试每个条件
- 这两种测试并不互相包含，有什么区别？
- 条件组合不一定会使判定的真假取值都出现一次



如：前面的条件组合 $T_1\overline{T_2}T_3\overline{T_4}$ 和 $\overline{T_1}T_2\overline{T_3}T_4$ ，使第一个判定始终为假，第二个判定始终为真。

判定－条件覆盖
(Decision Condition Coverage-DCC)

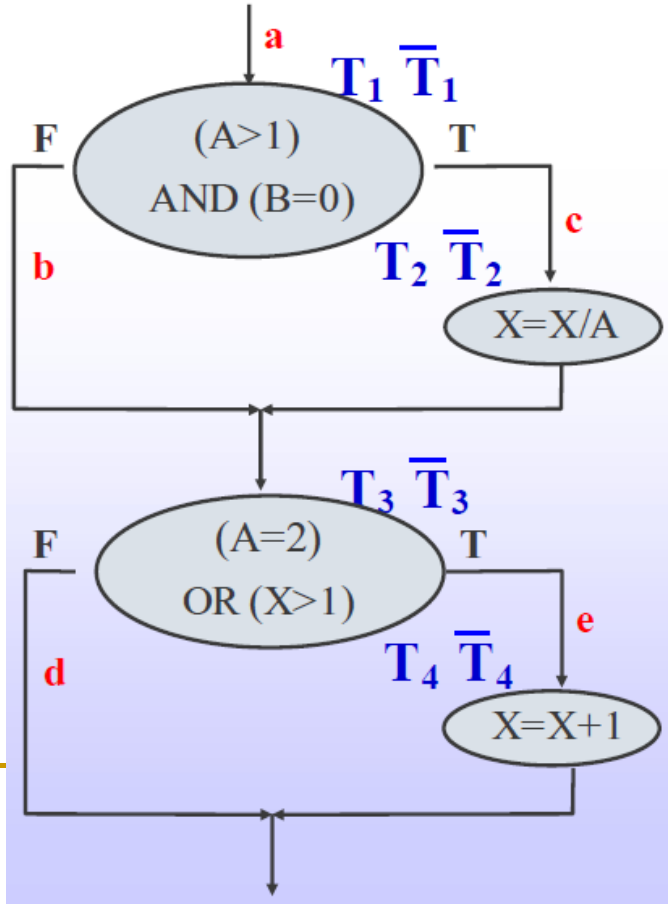
4级：100%判定覆盖+ 100%条件覆盖

- 设计足够多的测试用例，使得程序中每个判定包含的每个条件的所有情况（真/假）至少出现一次，并且每个判定本身的判定结果（真/假）也至少出现一次。

满足判定/条件覆盖的测试用例一定同时满足判定覆盖和条件覆盖。

判定－条件覆盖

测 试 用 例	覆盖分支	条件取值
【(2, 0, 4),(2, 0, 3)】	L1(a,c, e)	$T_1T_2T_3T_4$
【(1, 1, 1),(1, 1, 1)】	L2(a,b, d)	$\overline{T}_1\overline{T}_2\overline{T}_3\overline{T}_4$



条件组合覆盖

5级：条件组合（多条件）覆盖

- 通过执行足够的测试用例，使得程序中每个判定的所有可能的条件取值组合都至少出现一次。

满足组合覆盖的测试用例一定满足判定覆盖、条件覆盖和判定/条件覆盖

条件组合覆盖

① A＞1, B＝0 作 T_1T_2

② A＞1, B≠0 作 $T_1\overline{T_2}$

③ A≯1, B＝0 作 $\overline{T_1}T_2$

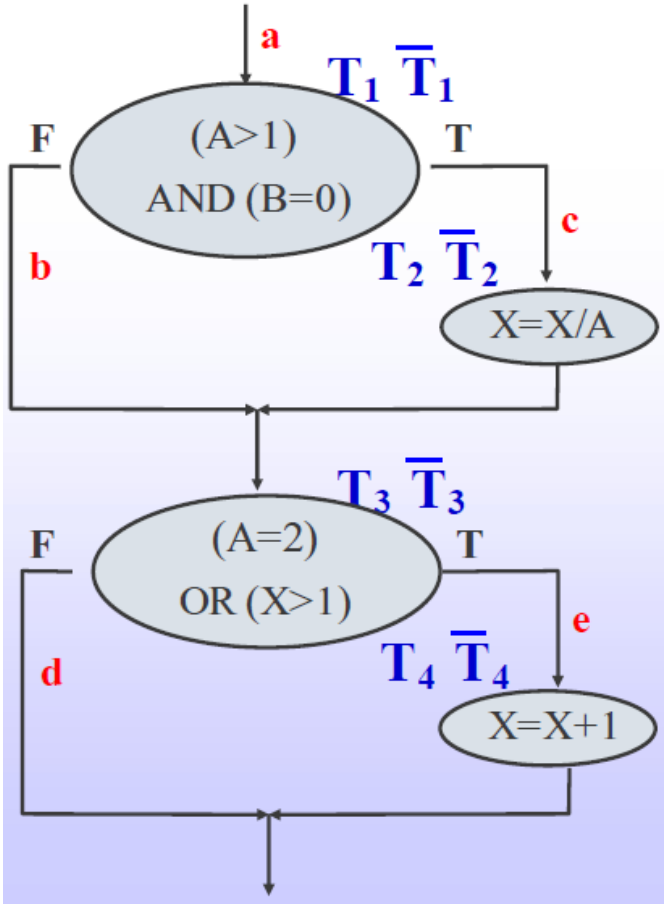
④ A≯1, B≠0 作 $\overline{T_1}\overline{T_2}$

⑤ A＝2, X＞1 作 T_3T_4

⑥ A＝2, X≯1 作 $T_3\overline{T_4}$

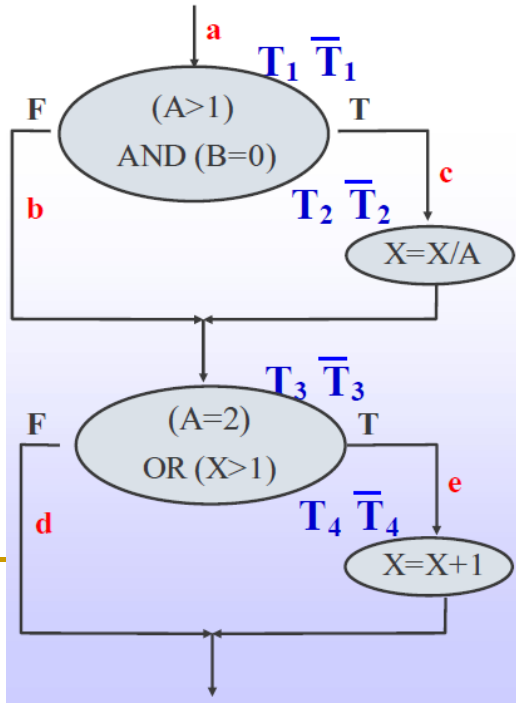
⑦ A≠2, X＞1 作 $\overline{T_3}T_4$

⑧ A≠2, X≯1 作 $\overline{T_3}\overline{T_4}$



条件组合覆盖

测试用例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】	(L1) $T_1T_2T_3T_4$	①, ⑤
【(2, 1, 1), (2, 1, 2)】	(L3) $T_1\overline{T_2}T_3\overline{T_4}$	②, ⑥
【(1, 0, 3), (1, 0, 4)】	(L3) $\overline{T_1}T_2\overline{T_3}T_4$	③, ⑦
【(1, 1, 1), (1, 1, 1)】	(L2) $\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$	④, ⑧



循环覆盖

6级：循环覆盖

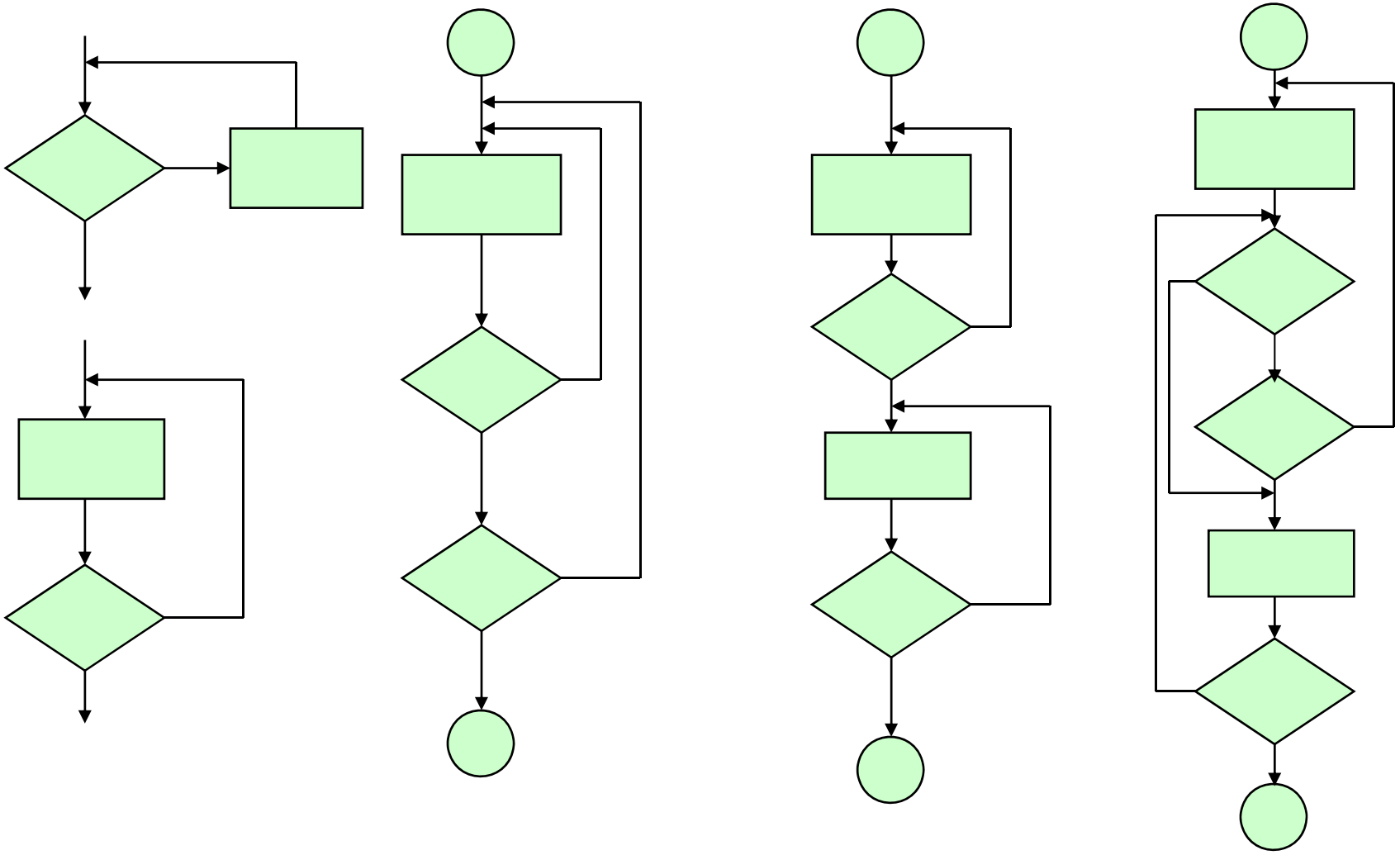
- 当模块中包含循环时，路径将变得无限，需要将循环数量降低到较小的程度
- 一般设计的案例：执行0次循环，执行1次循环，执行典型的n次循环，执行最大数量m次循环，还可以测试m-1和m+1（循环边界测试）次循环

循环覆盖

（1）循环测试的基本方法

- 从本质上说，循环测试的目的就是检查循环结构的有效性。
- 通常，循环可以划分为简单循环、嵌套循环、串接循环和 非结构循环4类。

循环覆盖



简单循环　　嵌套循环　　串接循环　　非结构循环

循环覆盖

① 测试简单循环。

设其循环的最大次数为**n**，可采用以下测试集：

- 跳过整个循环；
- 只循环一次；
- 只循环两次；
- 循环 **m** 次，其中**m<n**；
- 分别循环 **n-1**、**n** 和 **n+1** 次。

循环覆盖

② 测试嵌套循环。

如果将简单循环的测试方法用于嵌套循环，可能的测试次数会随嵌套层数成几何级数增加。此时可采用以下办法减少测试次数：

- 测试从最内层循环开始，所有外层循环次数设置为最小值；
- 对最内层循环按照简单循环的测试方法进行；
- 由内向外进行下一个循环的测试，本层循环嵌套的循环取某些“典型”(非边界等特殊)值；
- 重复上一步的过程，直到测试完所有循环。

1.首先设计单独测试循环P1的路径

P1=1

P1=0——>P1=1

2.单独设计循环P2的测试路径：

P2=1

P2=0——>P2=1　　(PS：判定时，1表示真，0表示假)

3.设置最大循环次数为1，最小循环次数为0，组合P1/P2测试路径：

P1=1——>P2=1

P1=1——>P2=0——>P1=1——>P2=1

P1=1——>P2=0——>P1=0——>P1=1

P1=0——>P1=1——>P2=1

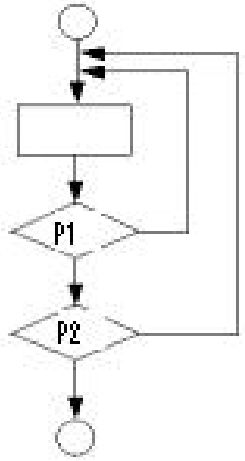
P1=0——>P1=1——>P2=0——>P1=1——>P2=1

（PS：有些时候也会将最大循环次数设置为n，n表示嵌套的循环个数）

4.“典型值”：内层镶嵌循环的基本测试路径。比如，针对于循环P2来说，典型值就为单独测试循环P1的两条路径。假如有 更外层的循环P3，那么针对P3来说，典型值是P1/P2组合的5条测试路径

5.最大循环数/最小循环数：和字面上的意思一样，单个循环的循环次数限制，目的是保证测试覆盖和限制测试过多消耗。最大循环数限制过多测试消耗；最小循环数保证测试覆盖率。

简单二次循环嵌套流程图



循环覆盖

③ 测试串接循环。

若串接的各个循环相互独立，则可分别采用简单循环的测试方法；否则采用嵌套循环的测试方法(第一个循环是第二个循环的初始值)。

④ 非结构循环。

对于非结构循环这种情况，无法进行测试，需要按结构化程序设计的思想将程序结构化后，再进行测试。

循环覆盖

（2）Z路径覆盖下的循环测试策略*

- Z路径覆盖是路径覆盖的一种变体，它是将程序中的循环结构简化为选择结构的一种路径覆盖。
- 循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况，即考虑执行时进入循环体一次和跳过循环体这两种情况

路径覆盖(Path Coverage -PC)

7级：100%路径覆盖

- 设计足够多的测试用例，覆盖程序中所有可能的路径。 从流程图和控制流图上可以看出，其中有4条可能路径，分别是L1、L2、L3和L4。（不是基本独立路径）
- 无论是程序中有没有出现的分支路径均要做测试，并且需要考虑多点分支和循环的情况
- 在有循环的情况下由于路径太多造成测试几乎不可完成

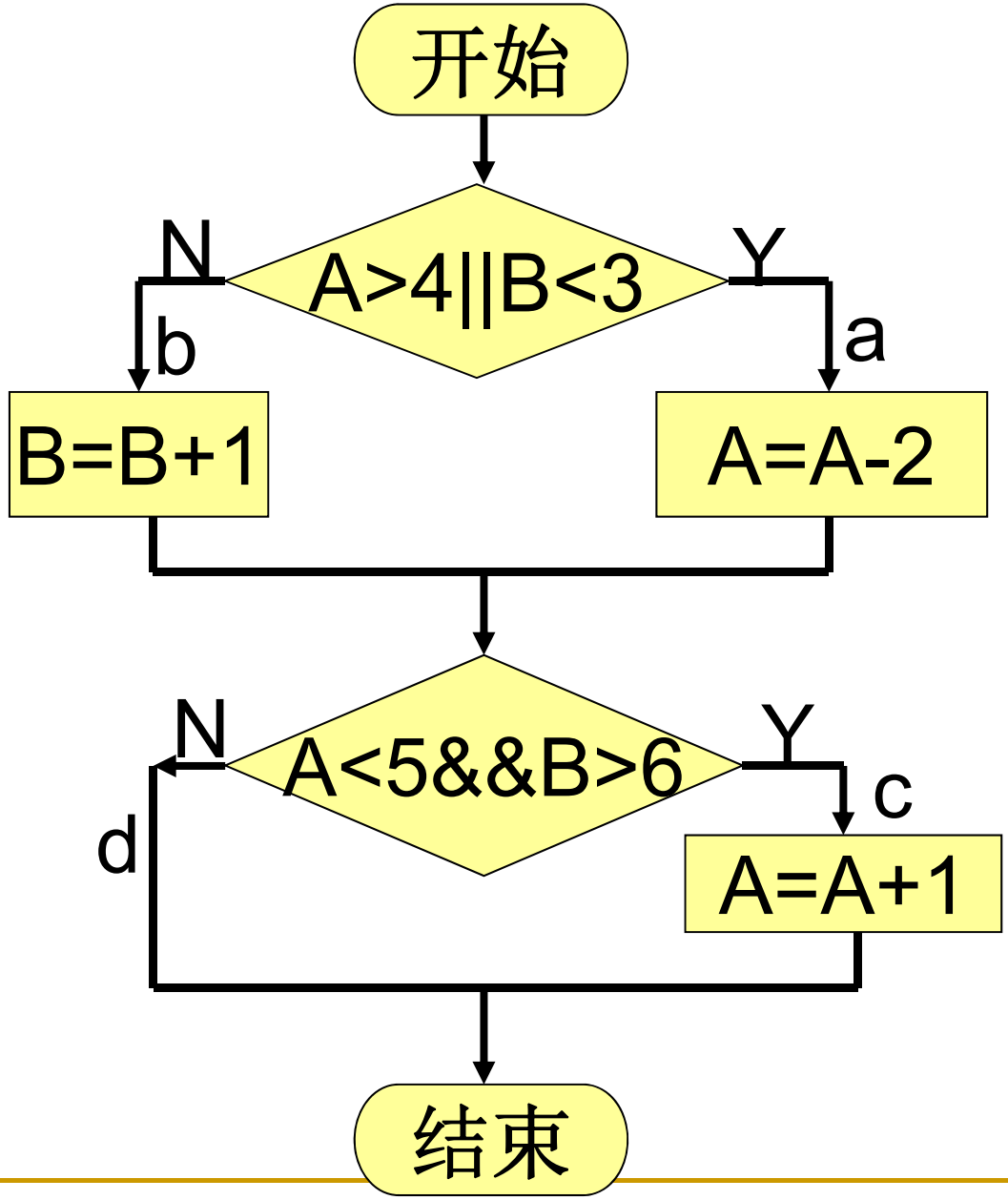
100%路径覆盖是最高层次的测试

测试覆盖情况统计

- 通过统计各种覆盖（代码覆盖、分支覆盖、基本块覆盖、…… ）情况，可以知道测试完整性；
- 一般使用工具进行统计
  - **C++ Test**可以显示覆盖情况
  - 一些专用的测试覆盖统计工具：**Cobertura, EMMA**等

习题：

用5种逻辑覆盖（1-5级）方法分析



内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

基本路径测试（McCabe圈覆盖）

- **Thomas.McCabe**提出的一种白盒测试技术，使用圈复杂度（**CyclomaticComplexity**），用于度量程序的复杂度
- 通过绘制被测试单元的控制流图来度量其圈复杂度
- 美国联邦航空局（**FAA**）管理的航空软件开发中，**McCabe**圈测试覆盖被用来作为单元测试的标准

基本路径测试——相关概念

路径测试

- 路径测试就是从一个程序的入口开始，执行所经历的各个语句的完整过程。
- 路径测试是白盒测试最为典型的问题，完成路径测试的理想情况是做到路径覆盖

实际上我们能做到的就是程序中主要路径的测试

基本路径测试——相关概念

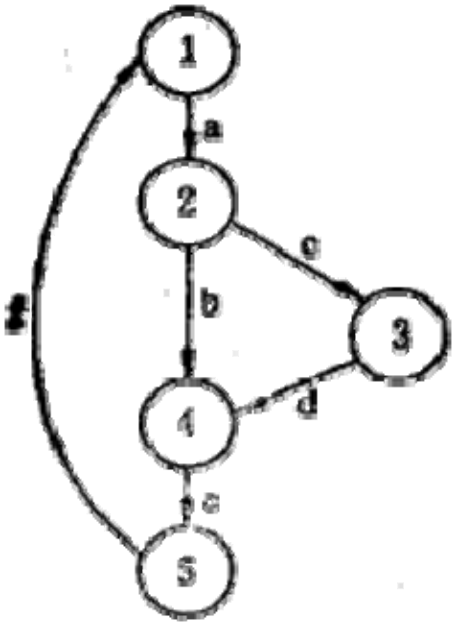
为了满足路径覆盖，必须首先确定具体的路径以及路径的个数。

进行路径分析，首先要解决的是路径如何表示的问题

基本路径测试——相关概念

- 有了控制流图，给出路径的标识就容易多了。

如果1节点为程序入口，
5节点为程序出口，



(b)控制流图

在此列举4条路径，
分别以弧序列表示及节点序列表示

基本路径测试——相关概念

路径的弧序列表示和节点序列表示

弧序列表示	节点序列表示	进入循环次数	经历左分支次数	经历右分支次数
acde	1-2-3-4-5	1	0	1
abe	1-2-4-5	1	1	0
abefabefabe	1-2-4-5-1-2-4-5-1-2-4-5	3	3	0
abefacde	1-2-4-5-1-2-3-4-5	2	1	1

基本路径测试——相关概念

路径表达式

- 上述弧序列表示和节点序列表示均指某一路径而言。我们希望能找到一种能够给出程序中所有路径的、更加概括的表示方法。

- 路径表达式作为一种表达式，其运算对象指的是控制流图中的弧，此外引入两个运算：乘和加

（1）弧a和弧b相乘，表示为**ab**，它表明路径是先经历弧a，接着再经历弧b，弧a和弧b是先后相接的。

（2）弧a和弧b相加，表示为**a+b**，它表明两条弧是“或”的关系，是并行的路段。

基本路径测试——相关概念

- 图a表示的控制流图，

共有4条路径：

abdf，abef，acdf，acef

- 这4条路径是并行的,或的关系，我们可用加法运算连接它们，从而得到完整的路径表达式：

abdf+abde+acdf+acef

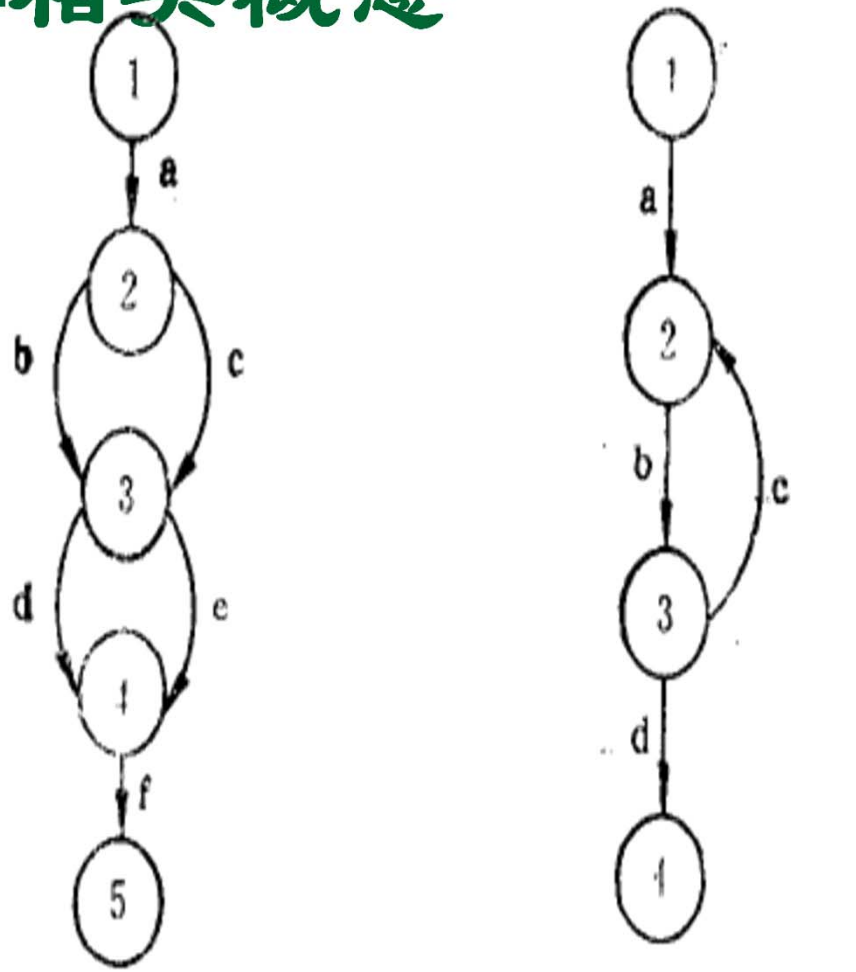


图　两个简单程序的控制流及其路径

基本路径测试——相关概念

- 图b给出的是具有循环的控制流图，它的路径随着执行不同次数循环体而有所不同，如**abd**，**abcbd**，**abcbcbd**，**.....**分别是执行一次、两次、三次循环体的路径。
- 其路径表达式：
  - **abd+abcbd+abcbcbd+.....**

基本路径测试——相关概念

路径数的计算：

- 假定所讨论的程序已经得到了它的路径表达式，则可把其中的所有弧均代以数值“1”，然后依表达式的乘法和加法运算，所得数值即为该程序的路径数。

- 独立路径
  - 从程序入口到出口的多次执行中，每次至少有一个语句集是新的，未被重复的。
  - 独立路径必须至少包含一条在本次定义路径之前不曾用过的边

注意：

测试可以被设计为基本路径集的执行过程，但基本路径集通常并不唯一

路径1：1－11

路径2：

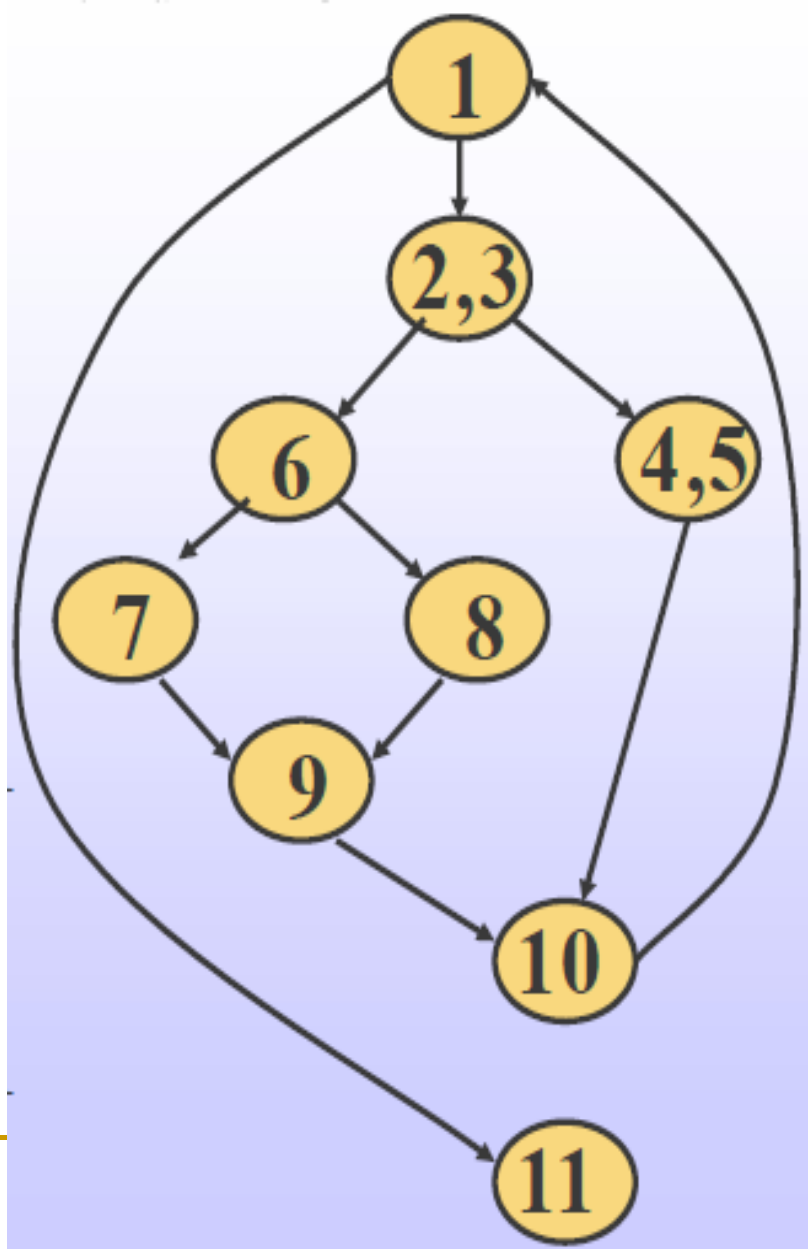
1－2,3－4,5－10－1－11

路径3：

1－2,3－6－8－9－10－1－11

路径4：

1－2,3－6－7－9－10－1－11



基本路径测试——相关概念

环形复杂度

- 环形复杂度（也称为圈复杂度）

  是一种为程序逻辑复杂度提供定量尺度的软件度量。

- 环形复杂度的意义：

  为程序逻辑复杂性提供定量度量。其值定义了：程序模块的独立路径数量；并提供了确保所有语句至少执行一次的测试数量的下界

可用如下三种方法之一来计算环形复杂度：

（1）控制流图中*区域*的数量 *（R）* 就是环形复杂度。

$$V(G) = R$$

（3）给定控制流图G的环形复杂度—V(G)，也可定义为

$$V(G) = C+1$$

其中，C是控制流图G中判定节点（if语句——二叉分支节点）的数量。

(3) 给定控制流图G的环形复杂度—V(G)，定义为

$$V(G) = E-N+2P$$

其中，E是控制流图中边的数量，N是控制流图中的节点数量，P是连通分量（单连通为1）。

注：此公式存在分歧，有文献中写为 **V(G) = E-N+P**

这与图是否“有且仅有一个出口（终止）节点”，没有出口即为强连通图，则取后者，仅有一个 取前者。

基本路径测试

基本路径测试方法

- 对于复杂性大的程序要做到所有路径覆盖（测试所有可执行路径）是不可能的。
- 在不能做到所有路径覆盖的前提下，如果某一程序的每一个独立路径都被测试过，那么可以认为程序中的每个语句都已经检验过了，即达到了语句覆盖。这种测试方法就是通常所说的基本路径测试方法
- 基本路径测试方法是在控制流图的基础上，通过分析控制结构的环形复杂度，导出执行路径的基本集，再从该基本集设计测试用例。

测试方法包括以下4个步骤：

（1）画出程序的控制流图。

（2）计算程序的环形复杂度，导出程序基本路径集中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所必须的测试用例数目的上界。

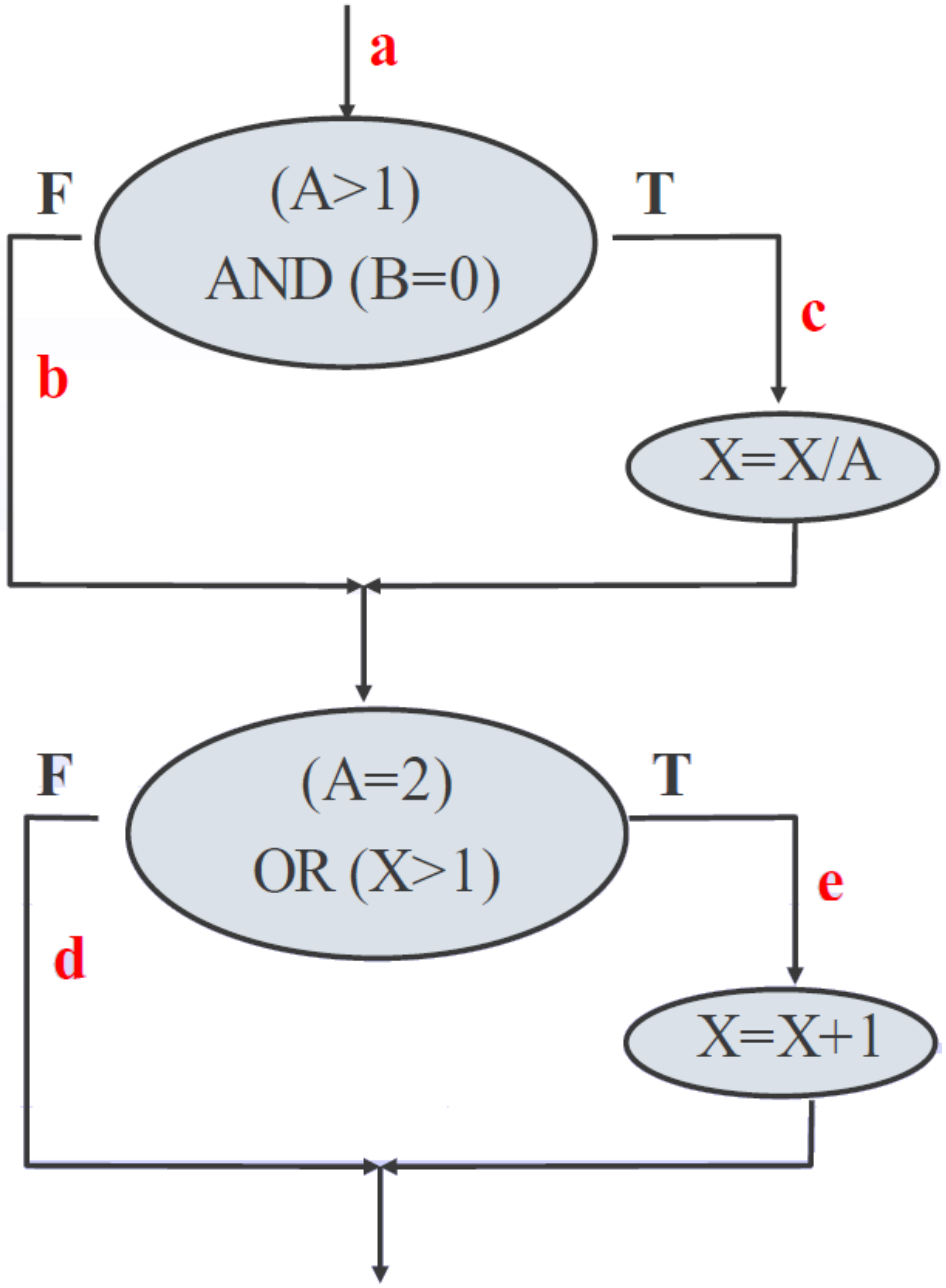
（3）导出基本路径集，确定程序的独立路径。

（4）根据（3）中的独立路径，设计测试用例的输入数据和预期输出。

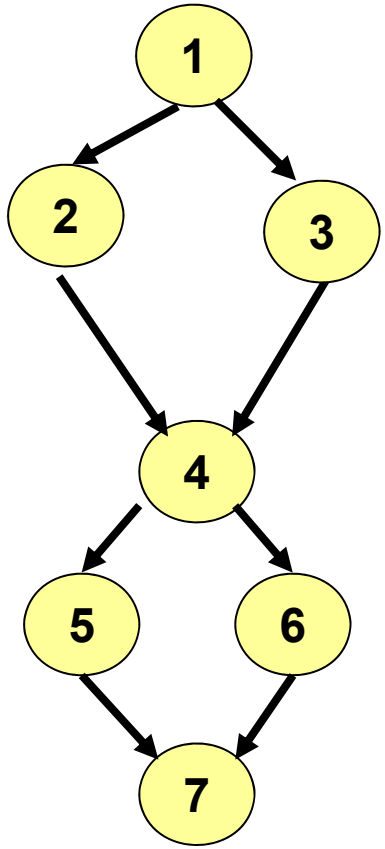
例1：

程序：

```
If(A>1 && B=0)
    X=X/A;
If(A=2 || X>1)
    X=X+1;
```



（1）画出控制流图：



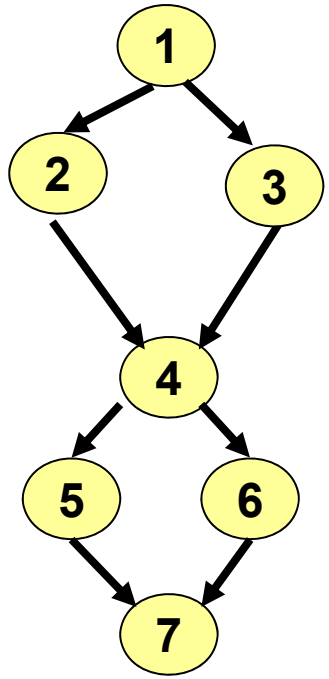
边没有编号，路径以点的序列标记，主要是考虑到考试时统一结果

(2)计算环形复杂度（基本路径数量）：

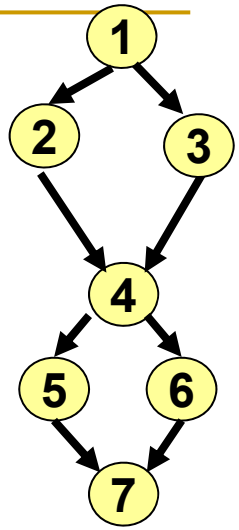
V(G)= E–N+2 = 8–7+2 = 3

V(G)= R = 3

V(G)= C + 1 = 2 + 1 = 3



（3）导出独立路径及测试用例：



P1：1-2-4-5-7

P2：1-3-4-6-7

P3：1-2-4-6-7

Px：1-3-4-5-7 是基本（独立）路径否？

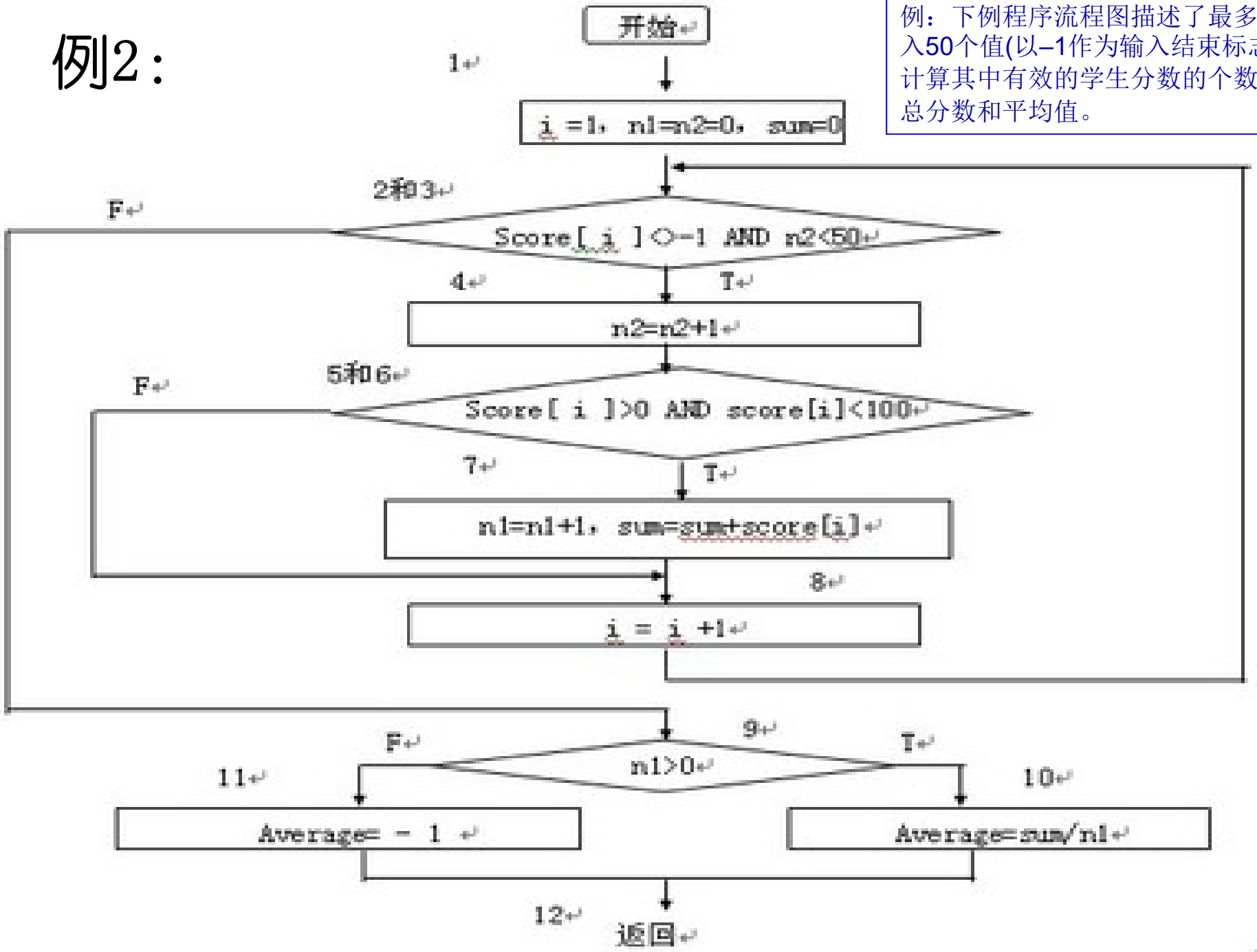
Px = P1 + P3 - P2 ， 即 **Px是P1~P3的线性组合；**

个人理解：基本路径方法是要保证每个语句都能覆盖到，又要求用例最少。 基本路径选取不是唯一的。

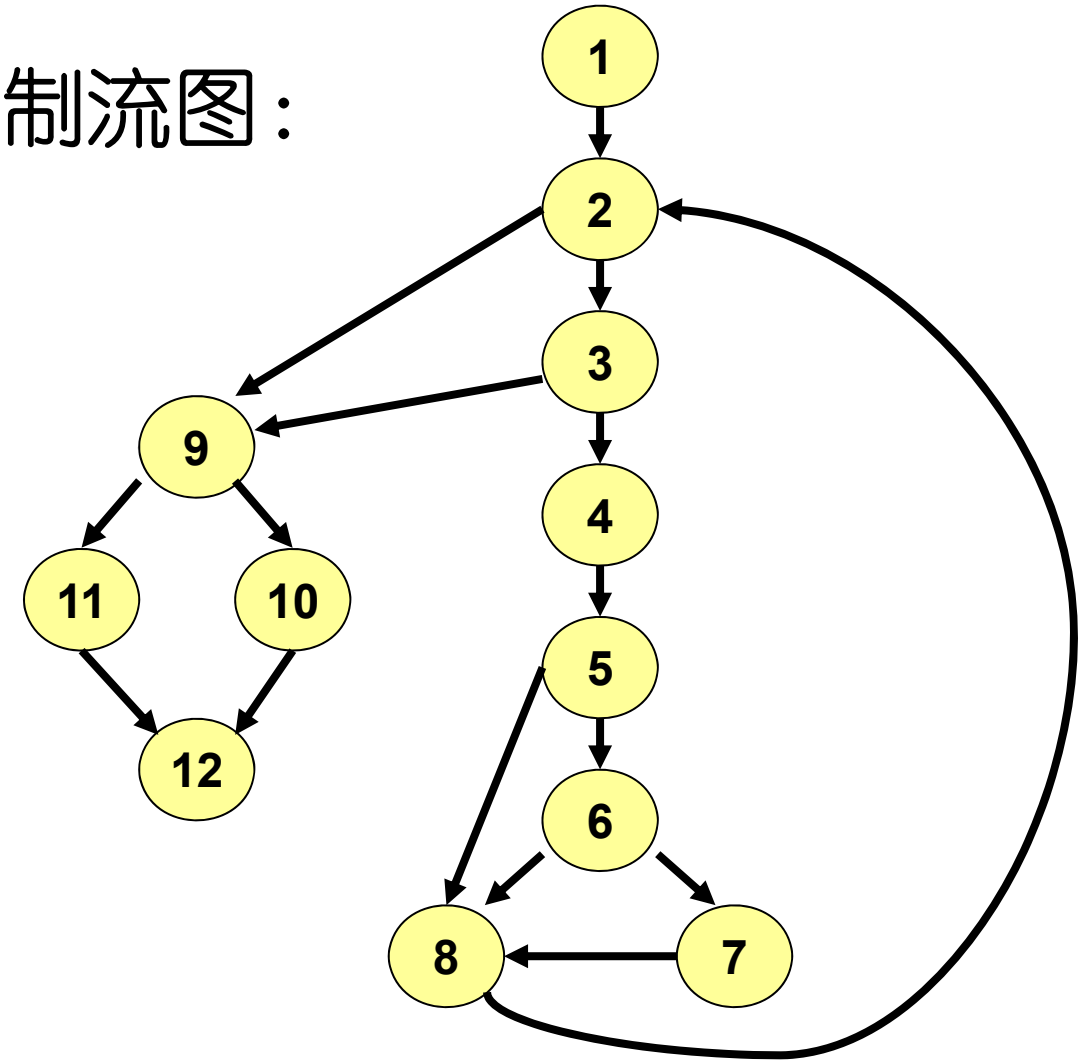
用例编号	基本路径	输入值	期望结果
1	1-2-4-5-7	A=1， B=1，X=1	X=1
2	1-3-4-6-7	A=2， B=0，X=4	X=3
3	1-2-4-6-7	A=1， B=1，X=2	X=3

例2：

例：下例程序流程图描述了最多输入50个值(以–1作为输入结束标志)，计算其中有效的学生分数的个数、总分数和平均值。



(1) 画出控制流图：



（2）计算环形复杂度：

$$V(G)= E–N+2 = 16–12+2 = 6$$

（3）导出独立路径和测试用例：

P1：1-2-9-10-12
P2：1-2-9-11-12
P3：1-2-3-9-10-12
P4：1-2-3-4-5-8-2
P5：1-2-3-4-5-6-8-2
P6：1-2-3-4-5-6-7-8-2

Px：1-2-3-9-11-12　是独立路径否？

为每一条独立路径各设计一组测试用例，以便强迫程序沿着该路径至少执行一次。

1)路径1(1-2-9-10-12)的测试用例：

score[k]=有效分数值，当k < i ;

score[i]=–1, 2≤i≤50;

期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

2)路径2(1-2-9-11-12)的测试用例：

score[1]= – 1 ;

期望的结果：average = – 1 ，其他量保持初值。

3)路径3(1-2-3-9-10-12)的测试用例：

输入多于50个有效分数，即试图处理51个分数，要求前51个为有效分数;

期望结果：n1=50、且算出正确的总分和平均分。

4)路径4(1-2-3-4-5-8-2…)的测试用例：

score[i]=有效分数，当i<50;

score[k]<0， k< i ;

期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average

习题

1、使用基本路径测试方法，为以下程序段设计测试用例。

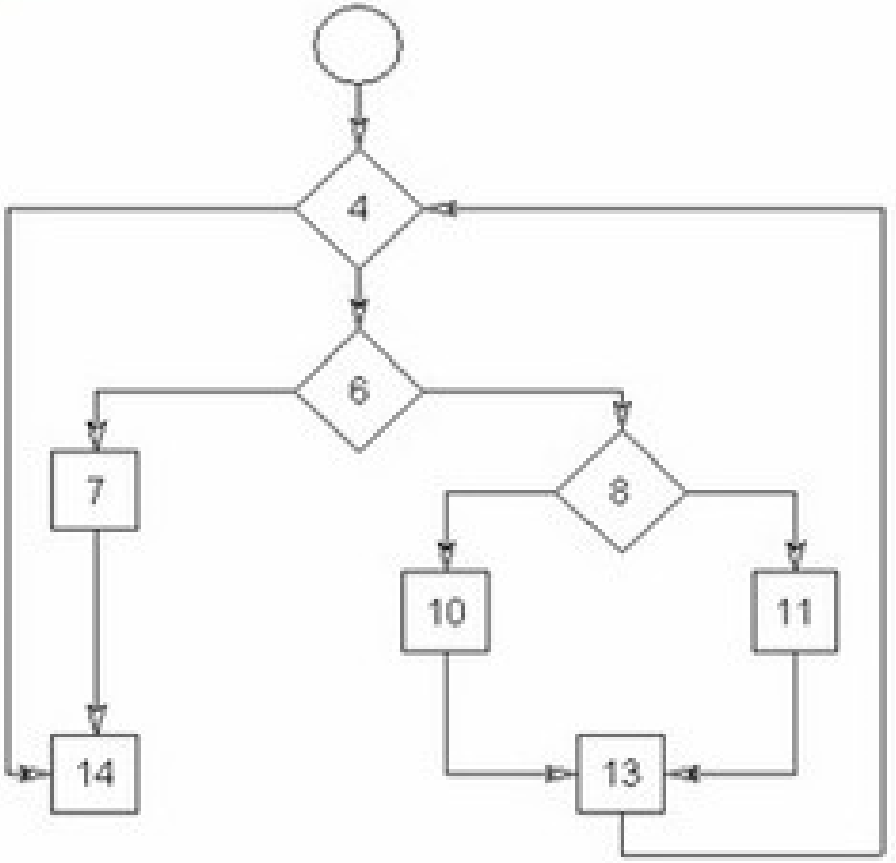
```
void Do (int X,int A,int B)
{
  if ( (A>1)&&(B=0) )
       X = X/A;
  if ( (A=2)||(X>1) )
       X = X+1;
}
```

习题

2、使用基本路径测试方法，为以下程序段设计测试用例。

例：有下面的C函数，用基本路径测试法进行测试

```
    void  Sort(int iRecordNum,int iType)
1.  {
2.    int x=0;
3.    int y=0;
4.    while (iRecordNum-- > 0)
5.    {
6.        if(0= =iType)
7.         { x=y+2; break;}
8.        else
9.            if (1= =iType)
10.          x=y+10;
11.           else
12.             x=y+20;
13.   }
14. }
```



程序流程图

习题

3、下面的程序段用于求给定数组中的最小值

```
int Min( int iNum, int* buf)
{     int min = buf[0];
      for( int i=1; i<iNum; i++ )
      {
          if( min> * ( buf + i ) )   min = *( buf + i );
       }
          return( min );
}
```

(1)绘制其控制流图

(2)计算其基本路径数并列举出基本路径

(3)写出测试用例

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

Foster的ESTCA覆盖准则

- **K.A.Foster**发现，对于大小判定型的分支，不仅要判定大小，还需要判定相等，因为一般容易在判定大小的边界出错，即相等处
- 在常规的白盒覆盖测试中，往往将相等条件归于某一分支而忽略它本身，造成程序缺陷

```
/// rel可以是
<, = 和>
if( A rel B )
 {  …}
else
 {  …}
```

ESTCA准则的内容

- **K.A.Foster**提出了一种经验型的测试覆盖准则，称为（**Error Sensitive TestCases Analysis** – 缺陷敏感测试用例分析）规则。
- ESTCA规则包括2条：

[规则1] 对于**A rel B**型的分支，应适当地选择**A**与**B**的值，使得**A < B, A = B**和**A> B**的情况分别出现一次

这是为了检测逻辑符号写错的情况，如将“**A<=B**”错写为“**A<B**”。

ESTCA准则的内容

[规则2] 对于**A rel C**（**rel**是**>** 或**<**，**A**是变量，**C**是常量）型的分支，
当**rel**为**<** 时，应适当地选择**A**的值，使：
A = C - M（**M**是距**C**最小的机器允许的正数，若**A**和**C**均为整型时，**M = 1**）
同样，当**rel**为**>**时，应适当地选择**A**，使：**A = C + M**

这是为了检测“差1”之类的错误
（如本应是“if（A > 1)”而错成“**if (A > 0）”)**

内容提要

- 白盒测试概述
- 静态白盒测试简介
- 基于数据流的白盒测试-数据覆盖
- 基于控制流的白盒测试-代码覆盖
- 基本路径测试（**McCabe**圈覆盖）
- **Foster**的**ESTCA**覆盖准则
- **xUnit**简介

xUnit简介

- 白盒测试是主要的单元测试方法；
- 一般由开发人员实施白盒单元测试；
- 目前广泛使用的动态白盒测试工具是**xUnit**测试框架家族。

xUnit简介

单元测试工具分为2类：

- 自动化单元测试工具：可以自动产生白盒测试用例，自动执行测试并生成测试报告等。如C++Test、Visual Unit等；
- 单元测试框架：提供了一系列的类库和接口，需要用户自己编写测试代码来完成单元测试工作。如xUnit家族等；

xUnit简介

- ✓ xUnit家族成员
- ✓ JUnit：测试Java语言代码
- ✓ XUnit, NUnit, 和 Visual Studio(MSTest)
- ✓ CPPUnit：测试C++语言代码
- ✓ PyUnit：测试Python语言代码
- ✓ SUnit：测试SmallTalk语言代码
- ✓ vbUnit：测试VB语言代码
- ✓ utPLSQL：测试Oracle's PL/SQL语言代码
- ✓ CUnit：测试C语言代码

…

单元测试框架的作用

- 单元测试框架可以支持动态白盒单元测试过程中需要的功能；
- 动态白盒单元测试的一般过程：
  - 编写测试代码
  - 执行单元测试
  - 评估测试结果

单元测试框架的作用

- 对于“编写测试代码”的支持：提供测试代码基本类库
- 对于“执行单元测试”的支持：提供一个测试运行器(**test runner**、命令行或者**GUI**工具)，可以执行一个或所有的单元测试
- 对于“评估测试结果”的支持：执行测试后可以给出测试结果信息：执行了多少测试；多少/哪些测试失败；失败的代码位置；等等

白盒测试中的研究问题*

- 如何自动生成具有较高覆盖率的白盒测试用例？

  Concolic(concrete & symbolic) testing

- 测试的完整性问题——错误注入
- 回归测试时的白盒测试用例选择问题