



## 第五部分 软件编码、测试与质量保障

- 5.1 软件编程
- 5.2 软件测试
- 5.3 白盒测试
- 5.4 黑盒测试
- 5.5 变异测试
- 5.6 性能测试



# 上节回顾：软件测试的概念

- **IEEE**：测试是使用人工和自动手段来运行或检测某个系统的过程，其目的在于检验系统是否满足规定的需求或弄清预期结果与实际结果之间的差别。

该定义明确提出了软件测试以“检验是否满足需求”为目标。



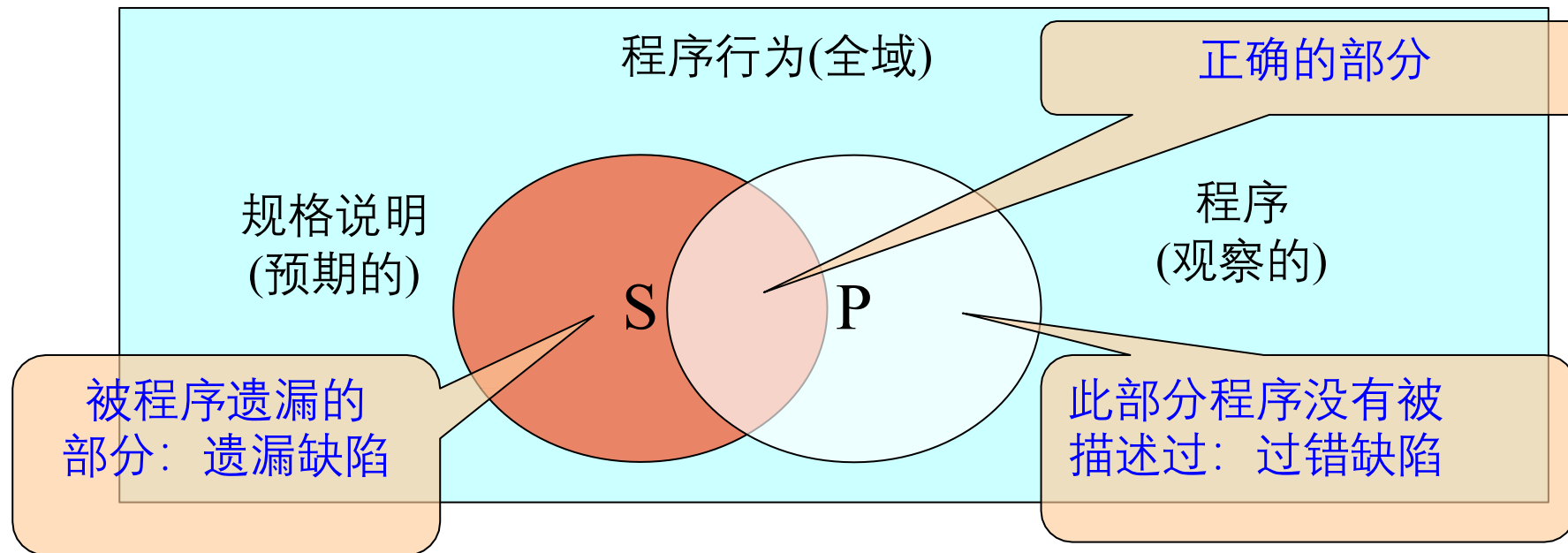
# 上节回顾：软件测试的目标

- 在程序交付测试之前，大多数程序员可以找到和纠正超过**99%**的错误；
- 在交付测试的程序中，每**100**条可执行语句的平均错误数量是**1-3**个；
- 软件测试的目的就是找出剩下的**1%**的错误。
- **Glen Myers**关于软件测试目的提出以下观点：
  - 测试是为了**发现错误**而执行程序的过程
  - 测试是为了证明“程序有错”，而**无法证明**“程序正确”
  - 一个好的测试用例在于**能够发现**至今未发现的错误
  - 一个成功的测试是**发现了**至今未发现的错误的测试



# 上节回顾：用Venn Diagram来理解测试

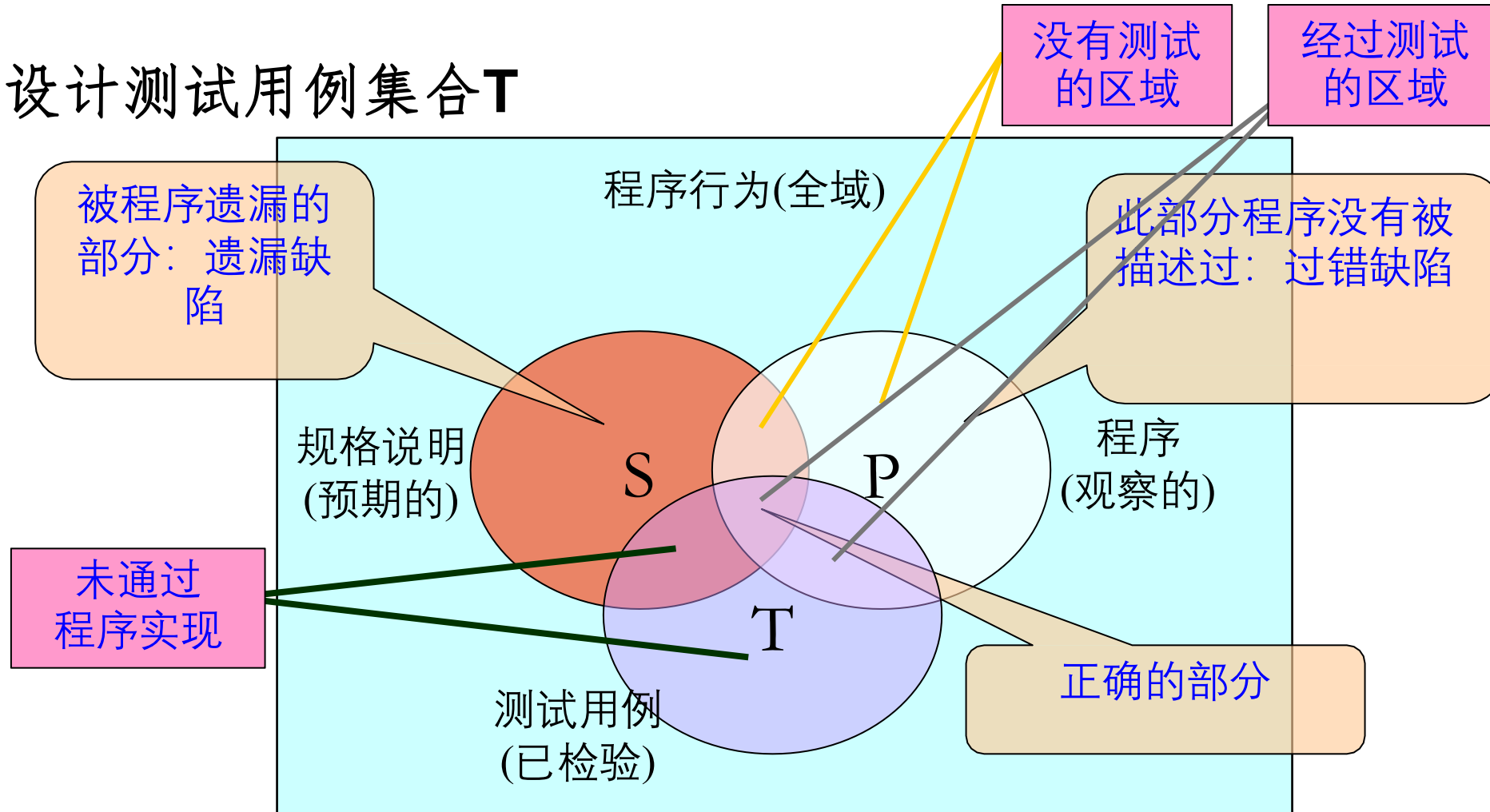
- 考虑一个程序行为全域，给定一段程序及其规格说明
  - 集合 $S$ 是所描述的行为；
  - 集合 $P$ 是用程序实现的行为；





# 上节回顾：用Venn Diagram来理解测试

## ■ 设计测试用例集合T





# 上节回顾：软件测试类型

测试对象角度

单元测试、集成测试、系统测试、验收测试

测试技术角度

黑盒测试（功能测试）、白盒测试（结构测试）

程序执行角度

静态测试、动态测试

人工干预角度

手工测试、自动化测试



# 典型的软件测试技术

- 测试的技术就是设计一组测试用例
  - 执行每个软件构件的内部逻辑和接口
  - 测试程序的输入和输出域以发现程序功能、行为和性能方面的错误
- 利用“白盒”测试用例设计技术执行程序内部逻辑
- 利用“黑盒”测试用例设计技术确认软件需求



## 5.3 白盒测试

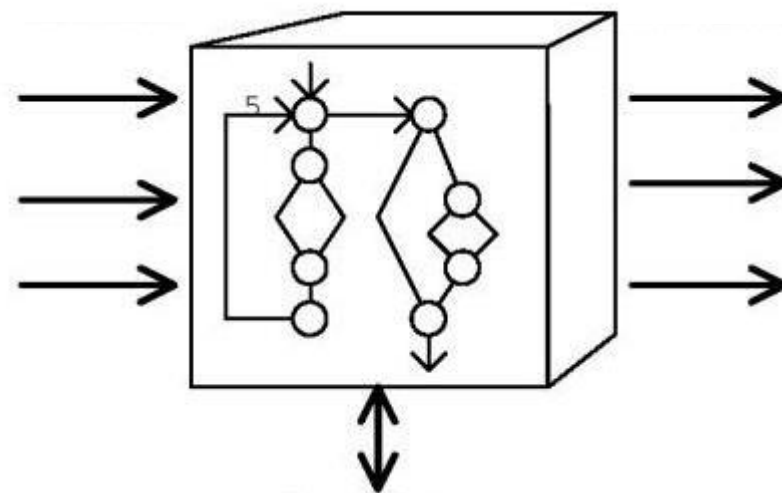
- 白盒测试概述
- 白盒测试方案



# 白盒测试的概念

## ■ 白盒测试(又称为“结构测试”或“逻辑驱动测试”)

— 把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。

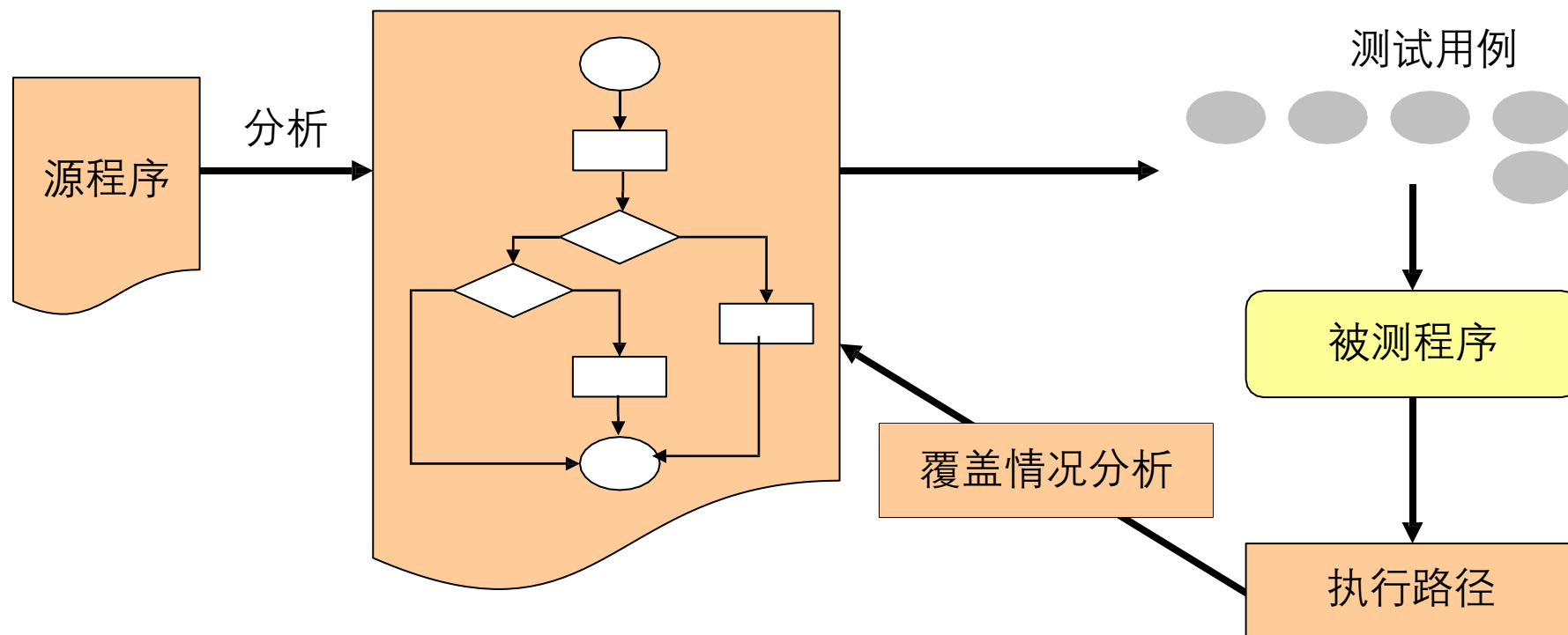




# 白盒测试的目的

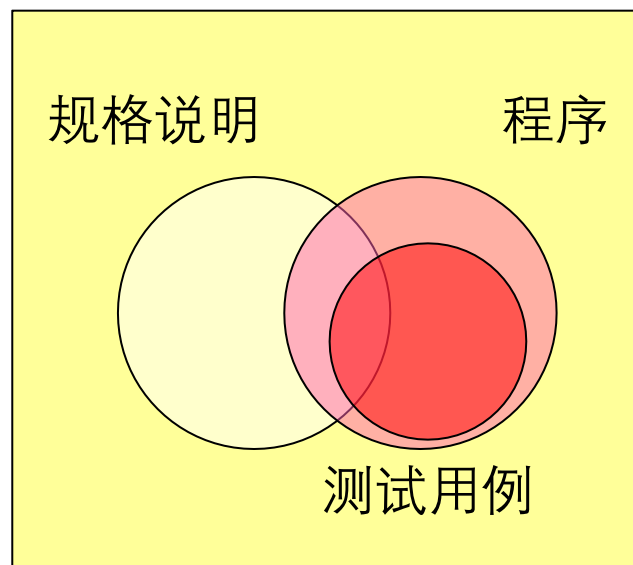
- 白盒测试主要对程序模块进行如下的检查：
  - 对模块的每一个独立的执行路径至少测试一次；
  - 对所有的逻辑判定的每一个分支(真与假)都至少测试一次；
  - 在循环的边界和运行界限内执行循环体；
  - 测试内部数据结构的有效性；
- 错误隐藏在角落里，聚集在边界处

# 白盒测试的过程



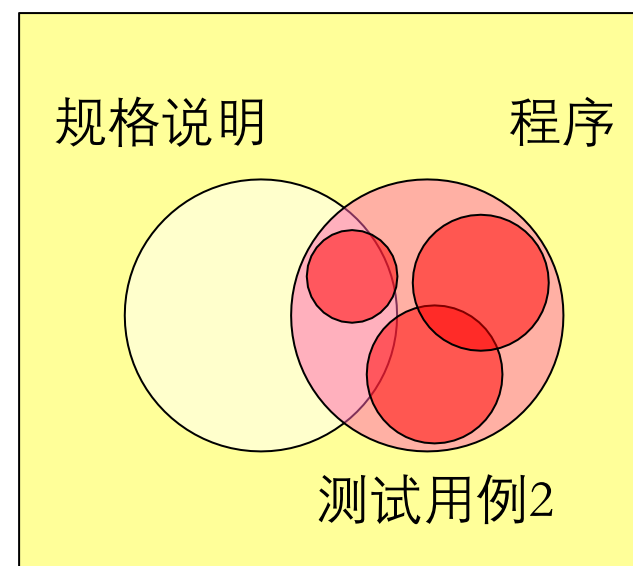
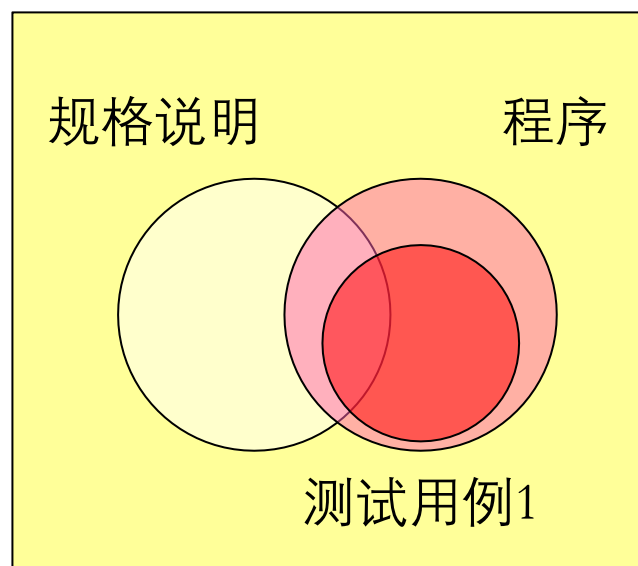


# 白盒测试的Venn Diagram



注意：覆盖区域只能在程序所实现的部分

# 白盒测试的Venn Diagram



测试用例所覆盖的程序实现范围越大，就越优良  
设计良好的测试用例，使之尽可能完全覆盖软件的内部实现



# 测试覆盖标准

## ■ 白盒测试的特点：

- 以程序的内部逻辑为基础设计测试用例，又称逻辑覆盖法。
- 应用白盒法时，手头必须有程序的规格说明以及程序清单。

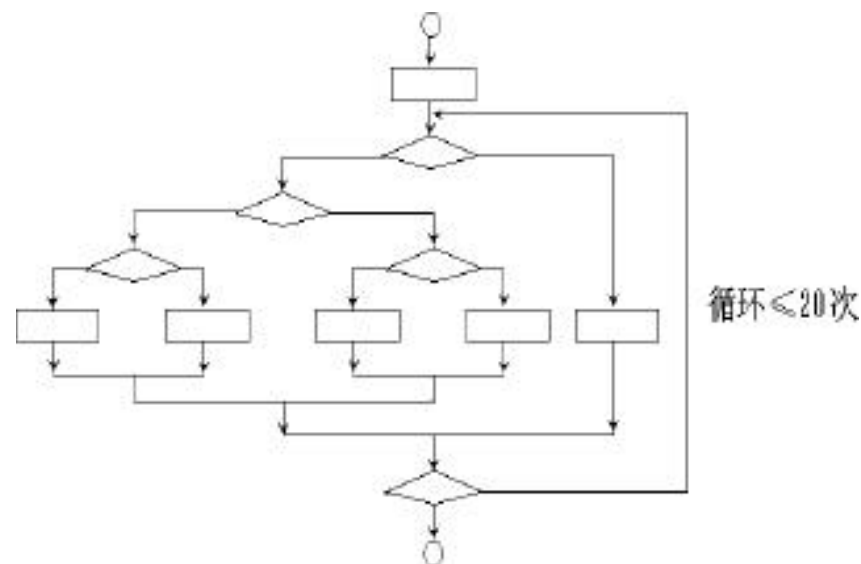
## ■ 白盒测试考虑测试用例对程序内部逻辑的覆盖程度：

- 最彻底的白盒法是覆盖程序中的每一条路径，但是由于程序中一般含有循环，所以路径的数目极大，要执行每一条路径是不可能的，只能希望覆盖的程度尽可能高些。



# 测试覆盖标准

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。
- 举例：某个小程序的流程图，包括了一个执行**20**次的循环。
  - 包含的不同执行路径数达 $5^{20}$ 条，对每一条路径进行测试需要1毫秒，假定一年工作 $365 \times 24$ 小时，要把所有路径测试完，需**3170**年。





## 5.3 白盒测试

- 白盒测试概述
- 白盒测试方案



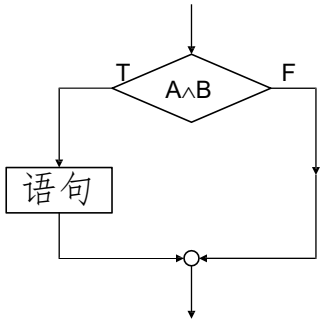
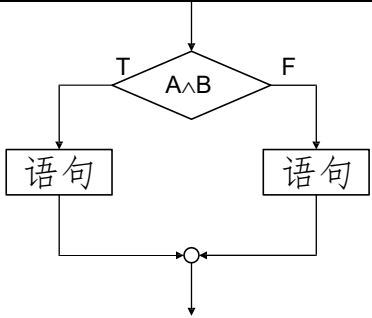


# 白盒测试方案

- 白盒测试方案技术之一：逻辑覆盖
  - 语句覆盖
  - 判定覆盖(分支覆盖)
  - 条件覆盖
  - 判定/条件覆盖
  - 条件组合覆盖
- 白盒测试方案技术之二：控制结构测试
  - 基本路径测试



# 逻辑覆盖：五种覆盖标准的对比

覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		<b><math>A \wedge B = T</math></b> 使得被测试程序中的每条可执行语句至少被执行一次。
判定覆盖		<b><math>A \wedge B = T</math></b> <b><math>A \wedge B = F</math></b> 每一判定的每个分支至少执行一次。

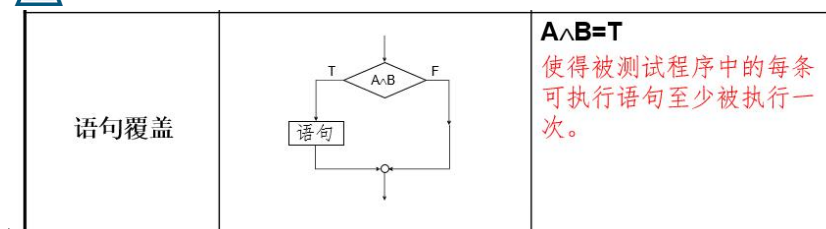


# 逻辑覆盖：五种覆盖标准的对比

覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		<b>A=T, A=F</b> <b>B=T, B=F</b> 每一判定中的每个条件，分别按“真”、“假”至少各执行一次。
判定/条件覆盖		<b>A∧B=T, A∧B=F</b> <b>A=T, A=F</b> <b>B=T, B=F</b> 同时满足判定覆盖和条件覆盖的要求。
条件组合覆盖		<b>A=T ∧ B=T</b> <b>A=T ∧ B=F</b> <b>A=F ∧ B=T</b> <b>A=F ∧ B=F</b> 求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次。



# 语句覆盖示例



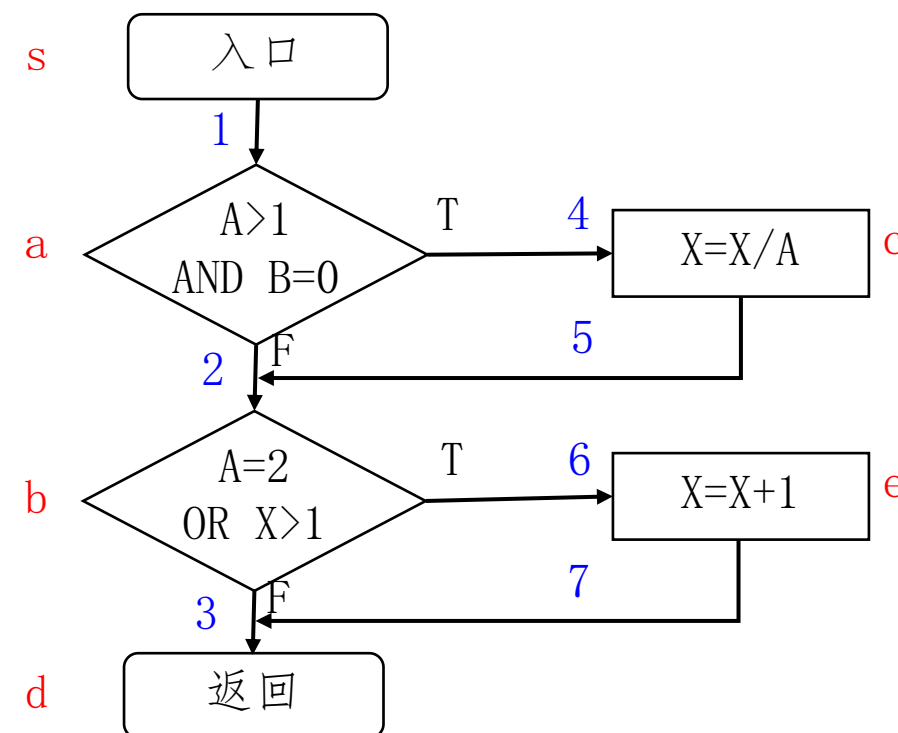
- 使得被测试程序中的每条可执行语句至少被执行一次。

Step1: 为了使每个语句都执行一次, 程序的执行路径应该是

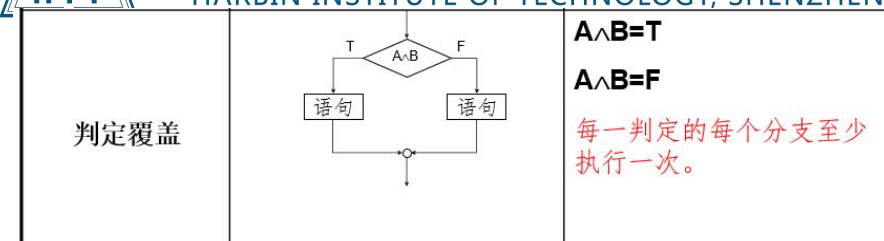
sacbed

Step2: 为此只需要输入下面的测试数据 (实际上X可以是任意实数)

$A=2, B=0, X=4$



## 判定覆盖示例



- 不仅每个语句必须至少执行一次，而且每个判定的每个分支都至少执行一次。

Step1: 程序的执行路径应该是

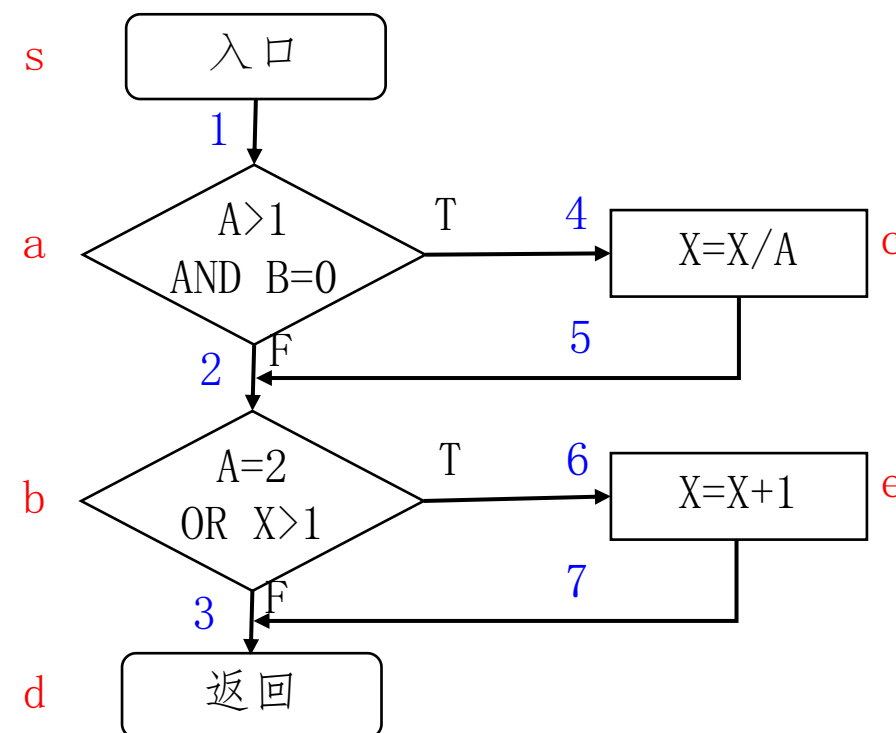
sacbed和sabd两组测试路径

Step2: 测试数据为

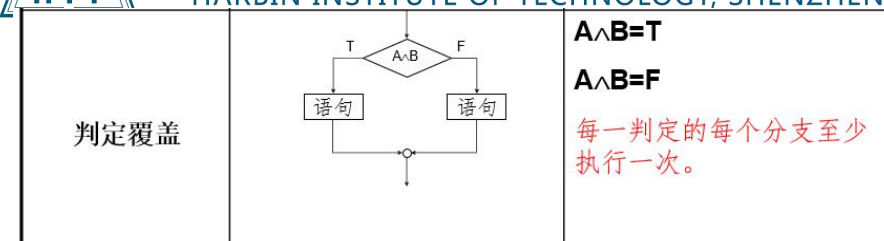
I.  $A=2, B=0, X=4$  (sacbed)

II.  $A=3, B=1, X=0$  (sabd)

还有没有其他测试路径可以满足判定覆盖？



# 判定覆盖示例



- 不仅每个语句必须至少执行一次，而且每个判定的每个分支都至少执行一次。

Step1: 程序的执行路径应该是

A) sa(T)cb(T)ed和sa(F)b(F)d两组测试路径

B) sa(F)b(T)ed和sa(T)cb(F)d两组测试路径

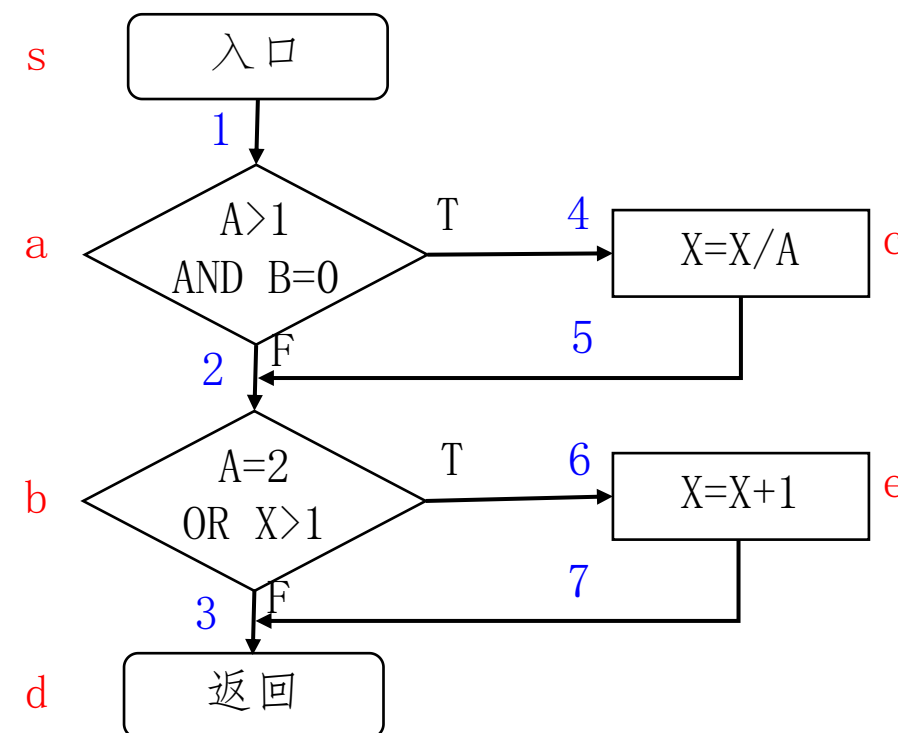
Step2: 测试数据为

I. A=2,B=0,X=4 (sacbed)

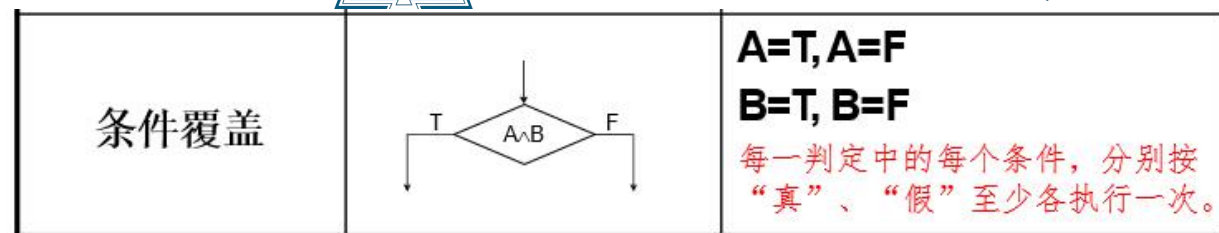
II. A=3,B=1,X=0 (sabd)

III. A=2,B=2,X=4 (sabed)

IV. A=3,B=0,X=0 (sacbd)



# 条件覆盖示例



- 不仅每个语句至少执行一次，而且使判定表达式中每个条件的可能取值至少各执行一次。

Step1: 在a点，A和B有下述结果出现：

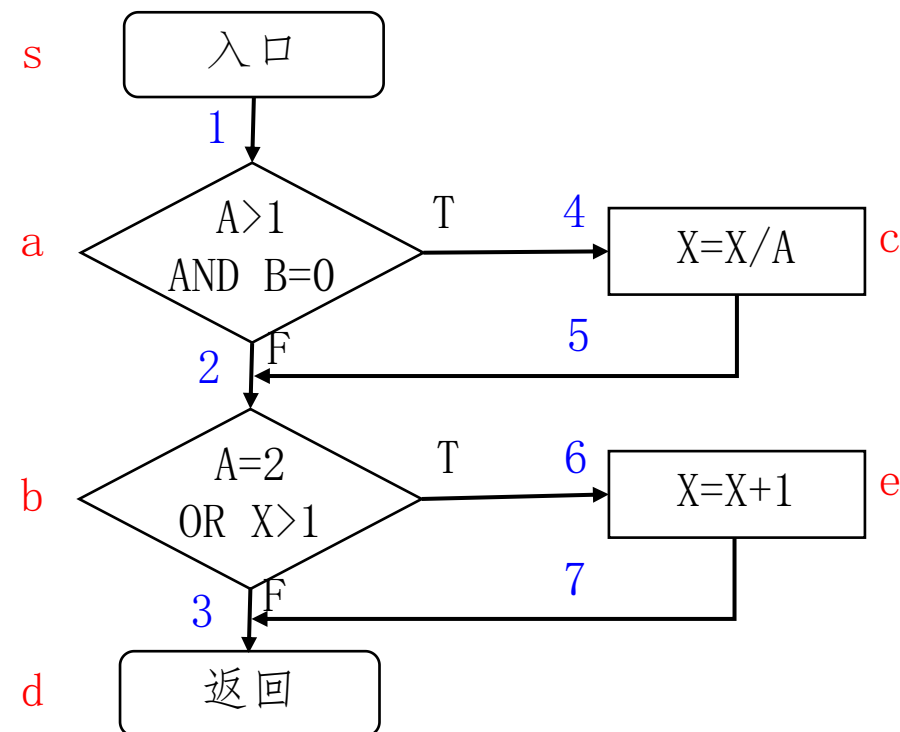
$A > 1, A \leq 1, B = 0, B \neq 0$

Step2: 在b点，A和B有下述结果出现：

$A = 2, A \neq 2, X > 1, X \leq 1$

Step3: 测试数据为

- I.  $A = 2, B = 0, X = 4$  (满足  $A > 1, B = 0, A = 2$ ) ( $X > 1$  未测试到)
- II.  $A = 1, B = 1, X = 1$  (满足  $A \leq 1, A \neq 2, X \leq 1$ ) ( $B \neq 0$  未测试到)
- III.  $A = 3, B = 1, X = 2$  (满足  $A > 1, B \neq 0, A \neq 2, X > 1$ )





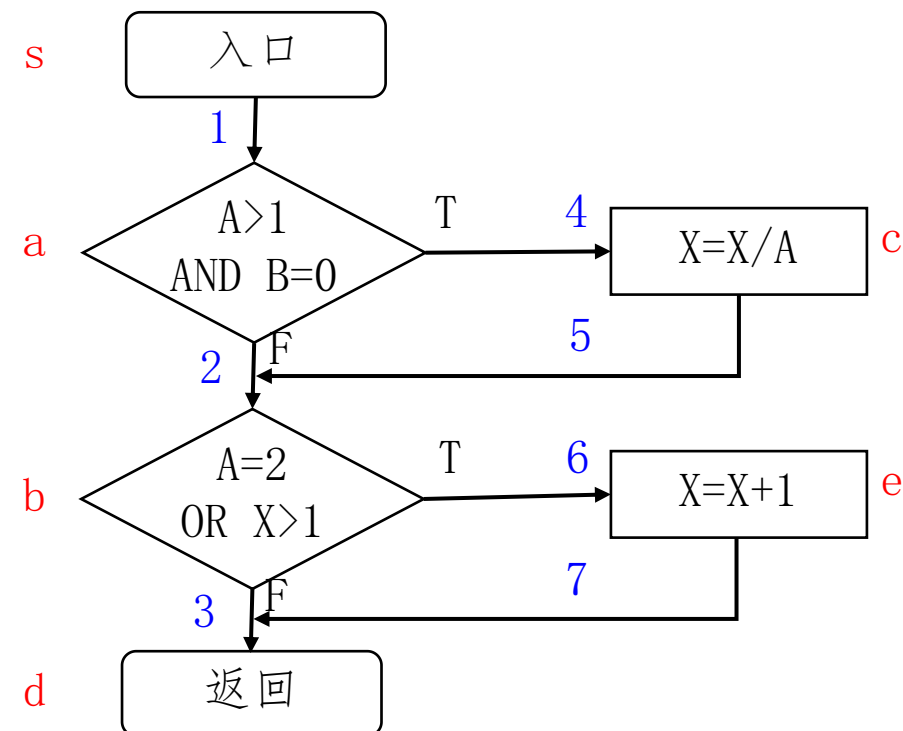
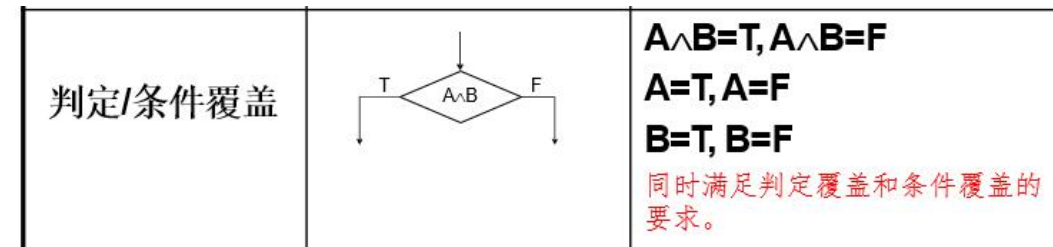
# 判定/条件覆盖示例

- 同时满足判定覆盖和条件覆盖的要求。

下面这个测试组合是不是判定/条件覆盖？

I.  $A=2, B=0, X=1$

II.  $A=1, B=1, X=2$







# 判定/条件覆盖示例

- 同时满足判定覆盖和条件覆盖的要求。

下面这个测试组合是不是判定/条件覆盖？

I.  $A=2, B=0, X=1$  (满足 $A>1, B=0, A=2$ 和 $X\leq 1$ 的条件)

II.  $A=1, B=1, X=2$  (满足 $A\leq 1, B\neq 0, A\neq 2$ 和 $X>1$ 的条件)

执行路径sacbed和sabed, 第二个判定表达式的值总为真。

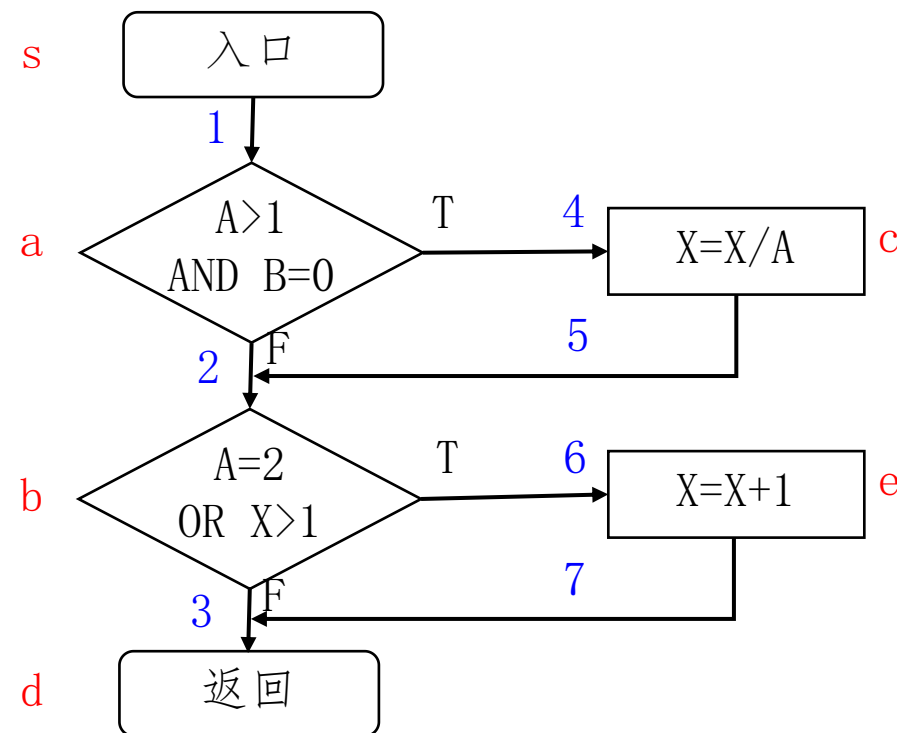
所以只满足条件覆盖, 不满足判定覆盖。

下面这个测试组合是不是判定/条件覆盖？

I.  $A=2, B=0, X=4$

II.  $A=1, B=1, X=1$

III.  $A=3, B=1, X=2$



# 判定/条件覆盖示例

- 同时满足判定覆盖和条件覆盖的要求。

下面这个测试组合是不是判定/条件覆盖？

I.  $A=2, B=0, X=1$  (满足  $A>1, B=0, A=2$  和  $X \leq 1$  的条件)

II.  $A=1, B=1, X=2$  (满足  $A \leq 1, B \neq 0, A \neq 2$  和  $X>1$  的条件)

执行路径sacbed和sabad, 第二个判定表达式的值总为真。

下面这个测试组合是不是判定/条件覆盖？

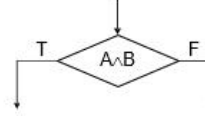
I.  $A=2, B=0, X=4$  (满足  $A>1, B=0, A=2$ ) ( $X>1$  未测试到)

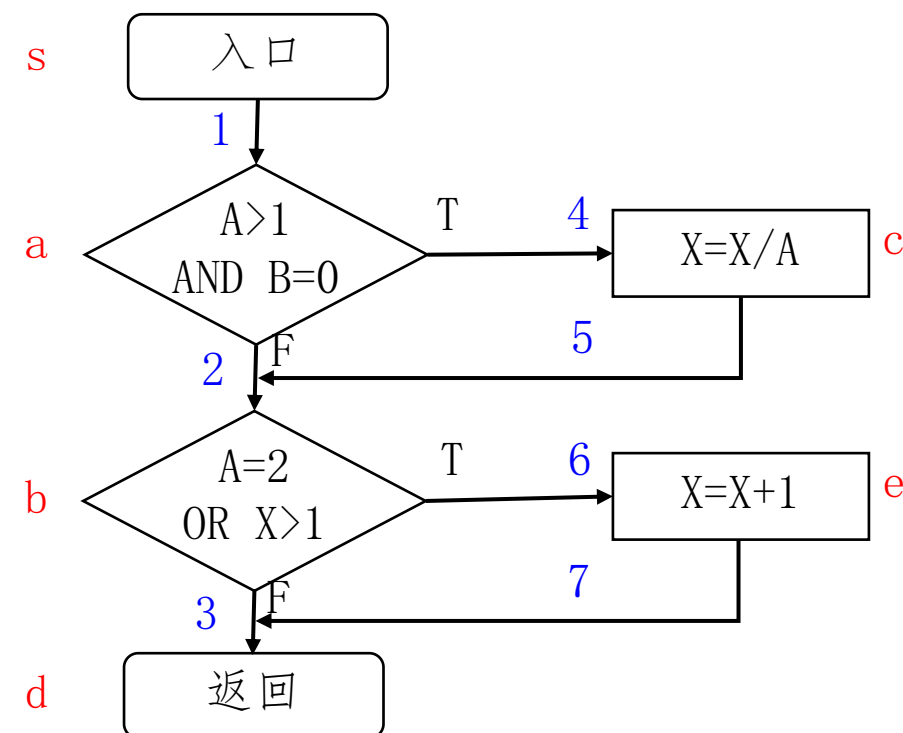
II.  $A=1, B=1, X=1$  (满足  $A \leq 1, A \neq 2, X \leq 1$ ) ( $B \neq 0$  未测试到)

III.  $A=3, B=1, X=2$  (满足  $A>1, B \neq 0, A \neq 2, X>1$ )

执行路径sacbed、sabd和sabad。

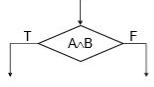
既满足条件覆盖，也满足判定覆盖。

判定/条件覆盖		$A \wedge B=T, A \wedge B=F$ $A=T, A=F$ $B=T, B=F$ 同时满足判定覆盖和条件覆盖的要求。
---------	---	---





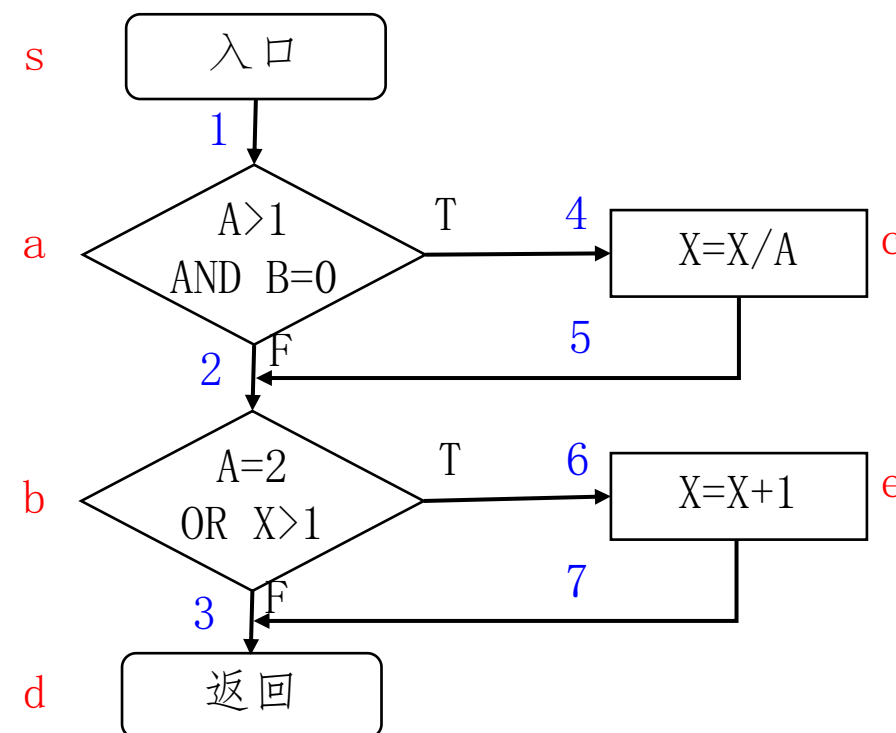
# 条件组合覆盖示例

条件组合覆盖		$A=T \wedge B=T$ $A=T \wedge B=F$ $A=F \wedge B=T$ $A=F \wedge B=F$ 求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次。
--------	---	--

- 每个判定表达式中条件的各种可能组合都至少出现一次。

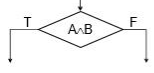
Step1: 共有8种可能的条件组合:

- 1)  $A>1, B=0$
- 2)  $A>1, B \neq 0$
- 3)  $A \leq 1, B=0$
- 4)  $A \leq 1, B \neq 0$
- 5)  $A=2, X>1$
- 6)  $A=2, X \leq 1$
- 7)  $A \neq 2, X>1$
- 8)  $A \neq 2, X \leq 1$





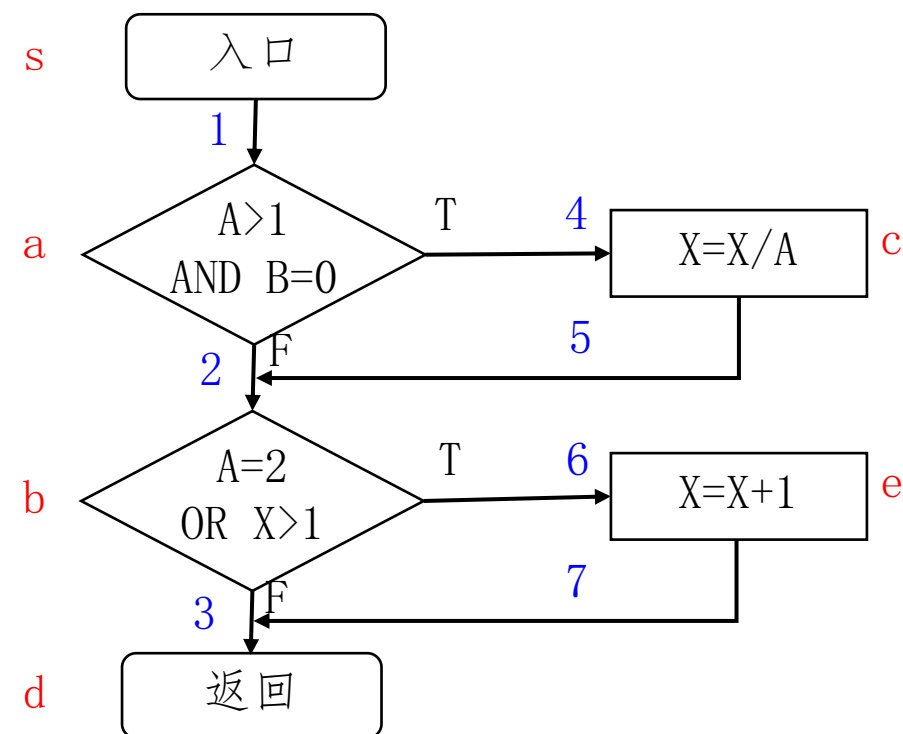
# 条件组合覆盖示例

条件组合覆盖		$A=T \wedge B=T$ $A=T \wedge B=F$ $A=F \wedge B=T$ $A=F \wedge B=F$ 求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次。
--------	---	--

- 每个判定表达式中条件的各种可能组合都至少出现一次。

Step2: 4组测试数据为:

- I.  $A=2, B=0, X=4$  (针对1,5组合, 执行路径sacbed)
- II.  $A=2, B=1, X=1$  (针对2,6组合, 执行路径sabed)
- III.  $A=1, B=0, X=2$  (针对3,7组合, 执行路径sacbed)
- IV.  $A=1, B=1, X=1$  (针对4,8两种组合, 执行sabd)





# 逻辑覆盖：五种覆盖标准的对比

发现 错 误 的 能 力	弱	语句覆盖	每条语句至少执行一次
	↓	判定覆盖	每一判定的每个分支至少执行一次
		条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
		判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
		条件组合覆盖	求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次
	强		



# 白盒测试方案

- 白盒测试方案技术之一：逻辑覆盖
  - 语句覆盖
  - 判定覆盖(分支覆盖)
  - 条件覆盖
  - 判定/条件覆盖
  - 条件组合覆盖
- 白盒测试方案技术之二：控制结构测试
  - 基本路径测试



# 白盒测试：基本路径测试

- 基本路径测试是一种白盒测试技术。使用这种技术设计测试用例时，首先计算过程设计结果的逻辑复杂度，并以该复杂度为指南定义执行路径的基本集合。
- 基本路径测试：
  - 在程序控制图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。
  - 设计出的测试用例要保证在测试中程序的每条可执行语句至少执行一次。
  - 程序中的每个条件至少被测试一次（True和False都要测试到）。



# 基本路径测试

## ■ 前提条件

- 测试进入的前提条件是在测试人员已经对被测试对象有了一定的了解，基本上明确了被测试软件的逻辑结构。

## ■ 测试过程

- 过程是通过针对程序逻辑结构设计和加载测试用例，驱动程序执行，以对程序路径进行测试。测试结果是分析实际的测试结果与预期的结果是否一致。





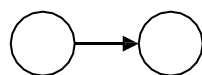
# 基本路径测试

- 在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。  
包括以下**4**个步骤：
  1. 以设计或源代码为基础，画出相应的流图
  2. 确定所得流图的环复杂度
  3. 确定独立路径的基本集合
  4. 准备测试用例，执行基本集合中每条路径

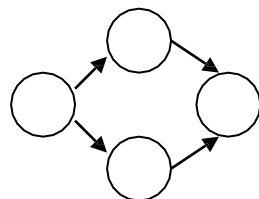
# (1) 画出相应的流图

## ■ 流图只有2种图形符号

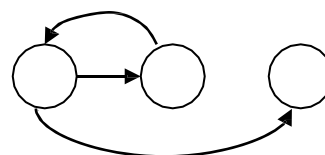
- 图中的每一个圆称为流图的**结点**，代表一条或多条语句。
- 流图中的箭头称为**边**或连接，代表控制流。



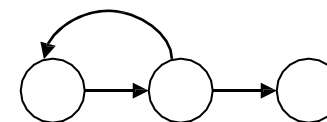
顺序结构



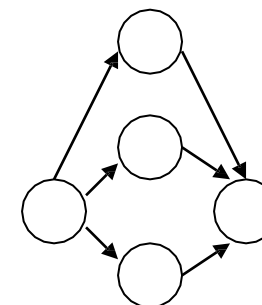
if 结构



while 结构



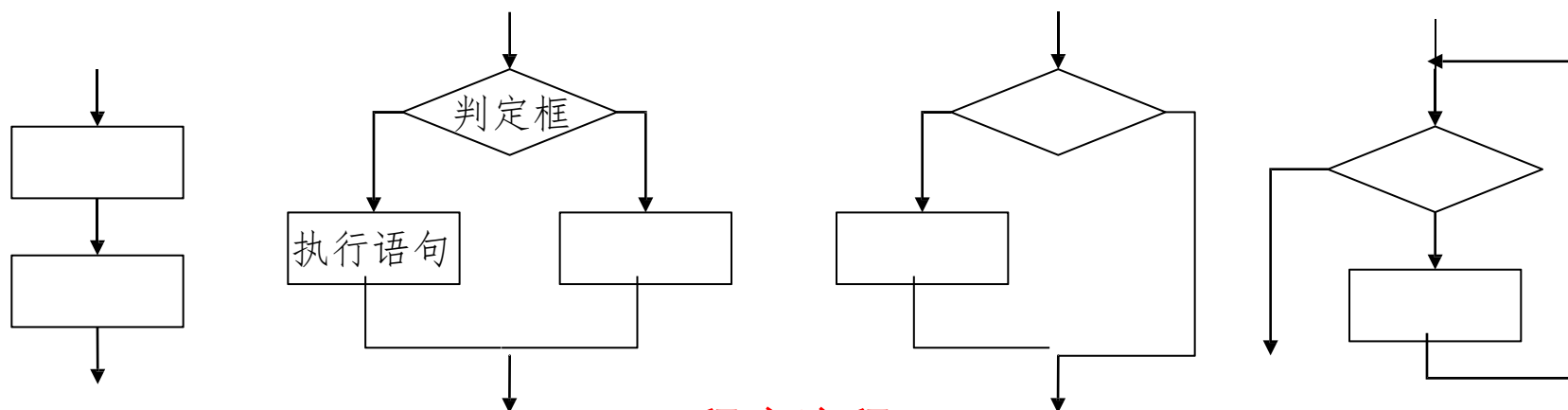
until 结构



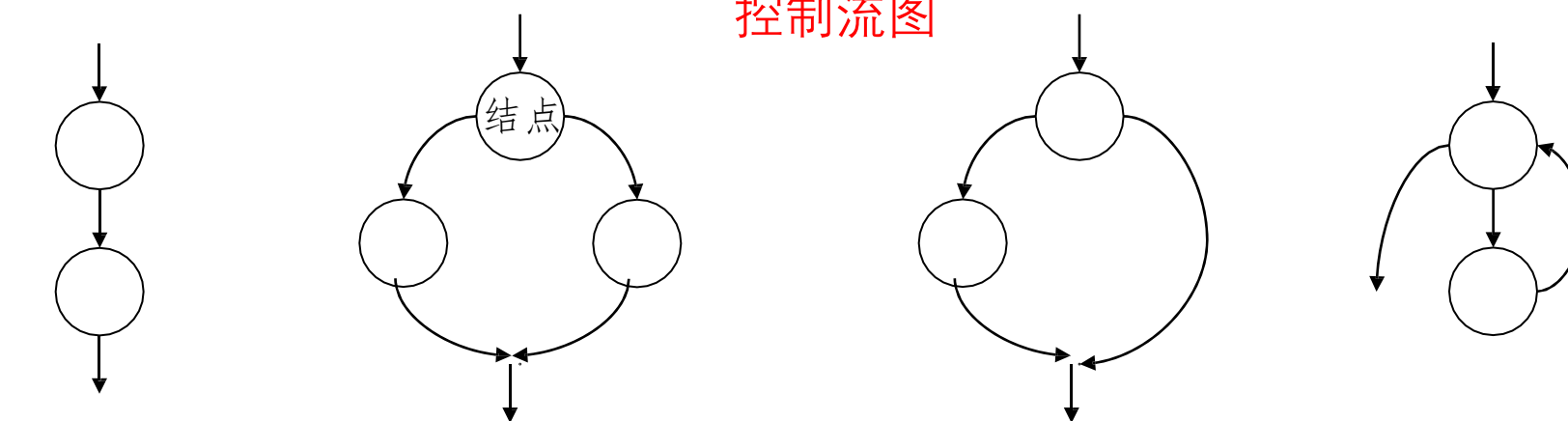
case 结构



# 程序流程图 → 控制流图



程序流程图



控制流图

## 代码 → 控制流图

- 如果判断中的条件表达式是由一个或多个逻辑运算符(OR, AND, NAND, NOR)连接的复合条件表达式,则需要改为一系列只有**单条件**的嵌套的判断。

- 例如:

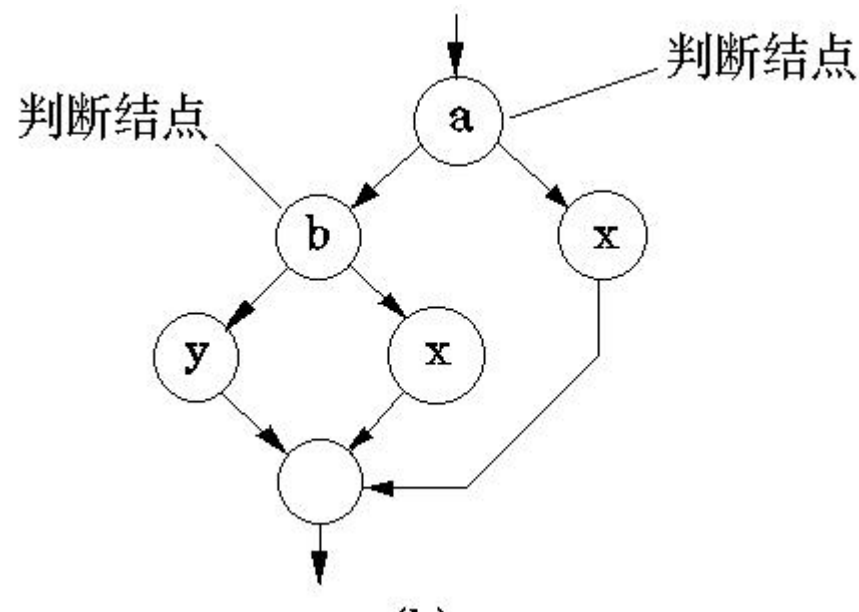
1 if a or b

2   x

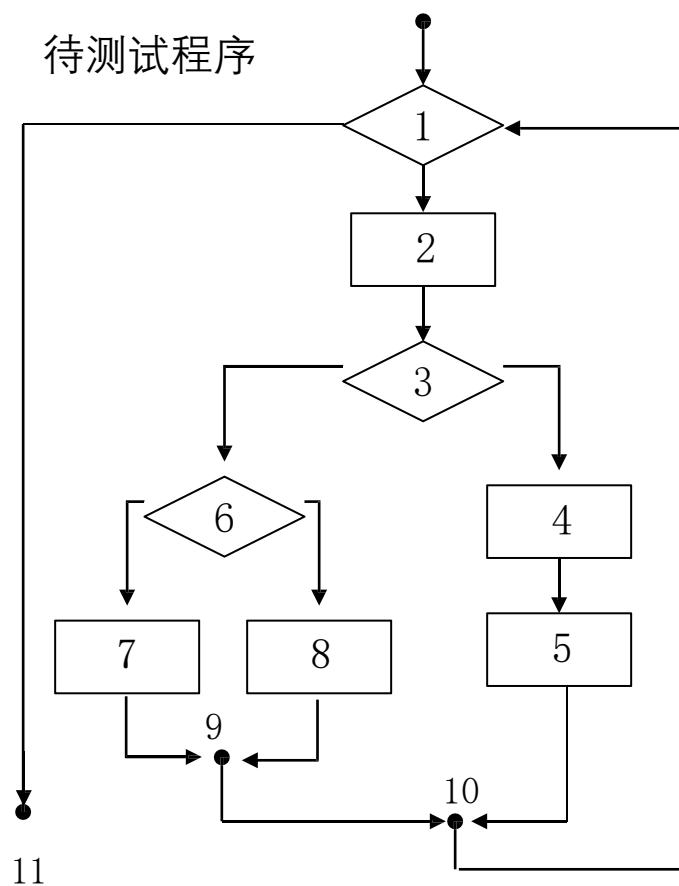
3 else

4   y

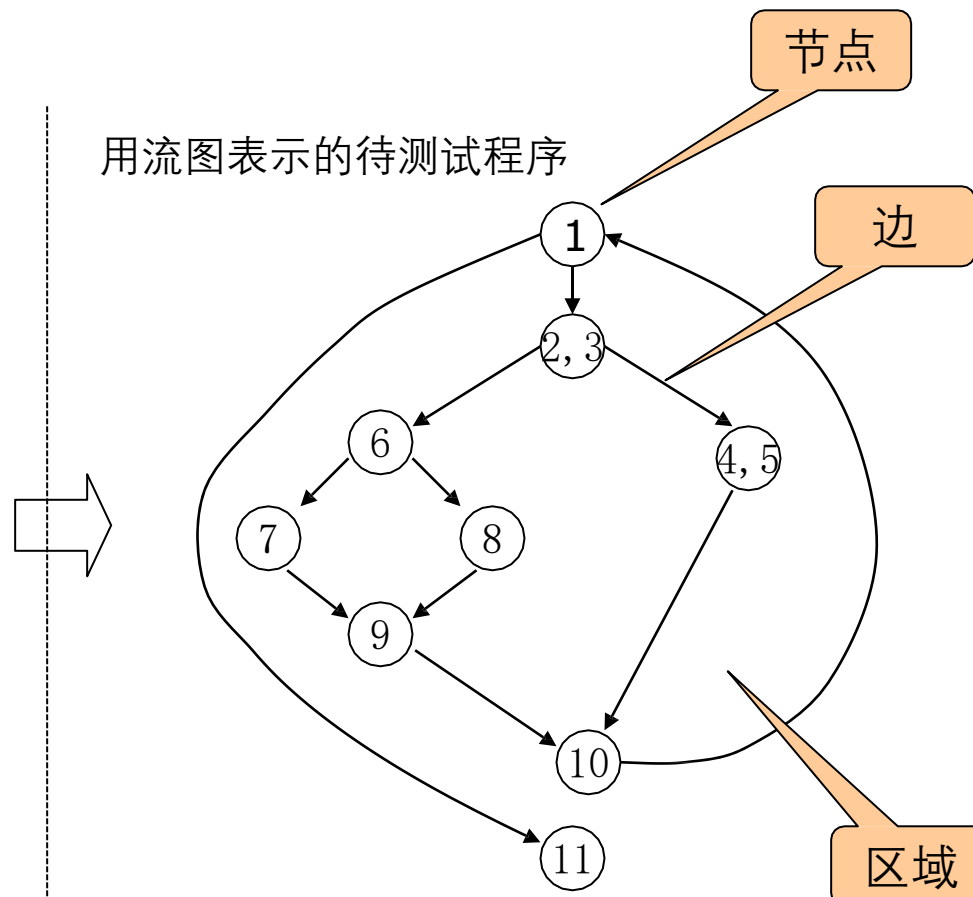
对应的逻辑为:



# [例] 画出流图



用流图表示的待测试程序



区域：由边和节点封闭起来的区域  
计算区域：不要忘记区域外的部分



## (2) 确定所得流图的环复杂度

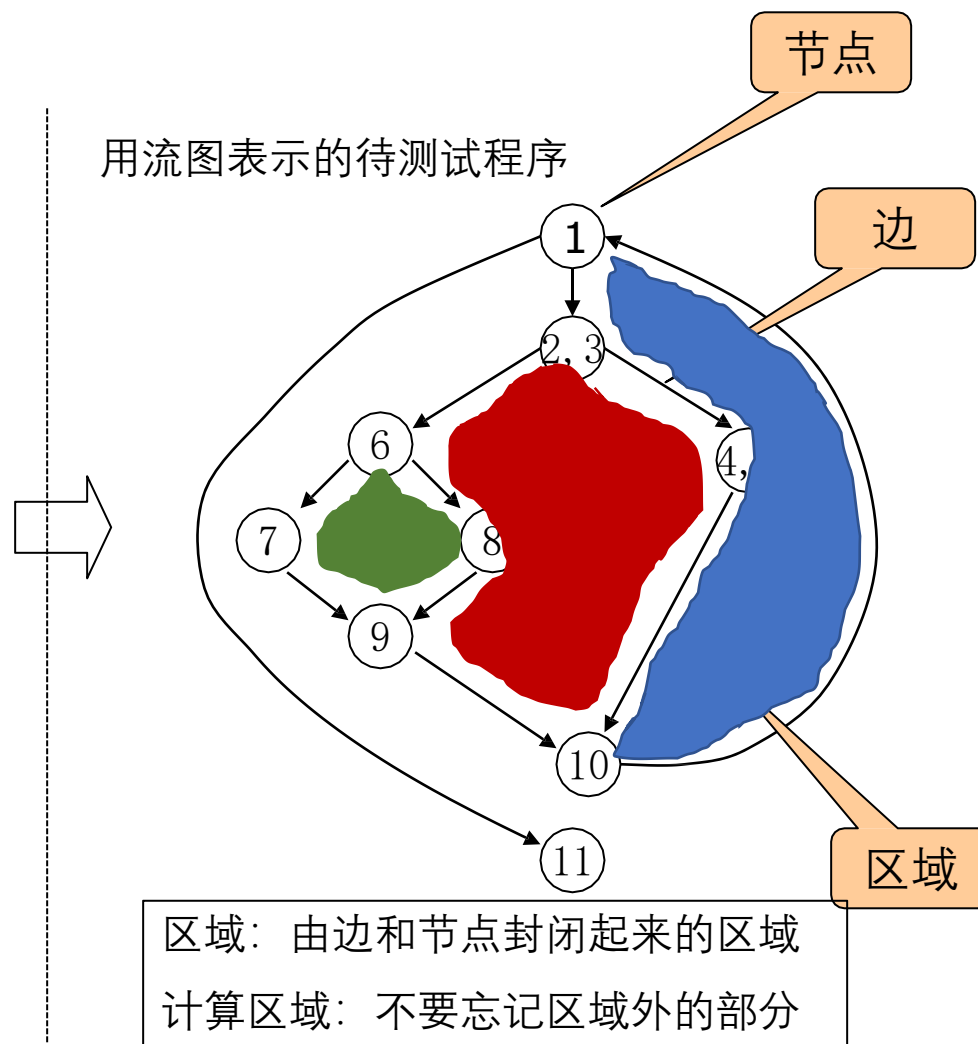
- 环复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。
- 有以下三种方法计算环复杂度：
  1. 流图中区域的数量；
  2. 给定流图G的圈复杂度 $V(G)$ ，定义为 $V(G)=E-N+2$ ，E是控制流图中边的数量，N是流图中结点的数量；
  3. 给定流图G的圈复杂度 $V(G)$ ，定义为 $V(G)=P+1$ ，P是控制流图G中判定结点的数量。

# [例] 画出流图

## ■有以下三种方法计算环复杂度:

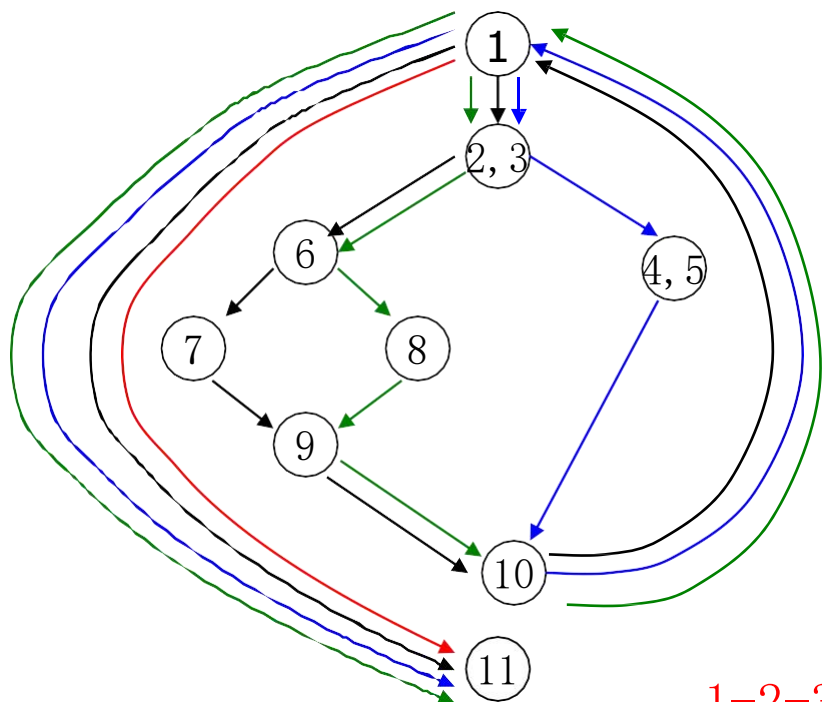
1. 流图中区域的数量:  $3+1=4$
2. 给定流图G的圈复杂度 $V(G)$ , 定义为 $V(G)=E-N+2$ , E是控制流图中边的数量, N是流图中结点的数量;  $11-9+2=4$
3. 给定流图G的圈复杂度 $V(G)$ , 定义为 $V(G)=P+1$ , P是控制流图G中判定结点的数量:  $3+1=4$

流图的环复杂度为4



### (3) 确定独立路径的基本集合

- 独立路径：一条路径，至少包含一条在定义该路径之前不曾用过的边(至少引入程序的一个新处理语句集合或一个新条件)。



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

对以上路径的遍历，就是至少一次地执行了程序中的所有语句。

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11是一条独立路径吗？

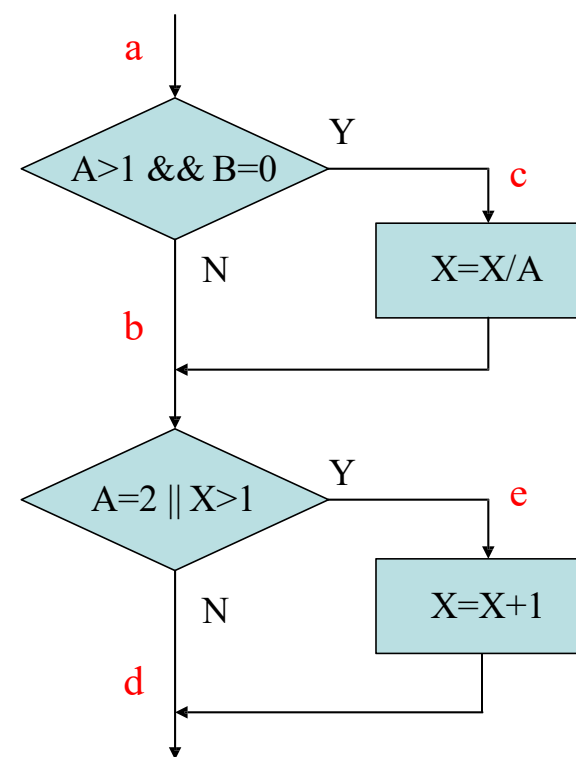




## (4) 设计测试用例

### ■ 根据路径设计合适的测试用例

通过路径	测试用例
ace	A=2、B=0、X=3
abd	A=1、B=0、X=1
abe	A=2、B=1、X=1
acd	A=3、B=0、X=1





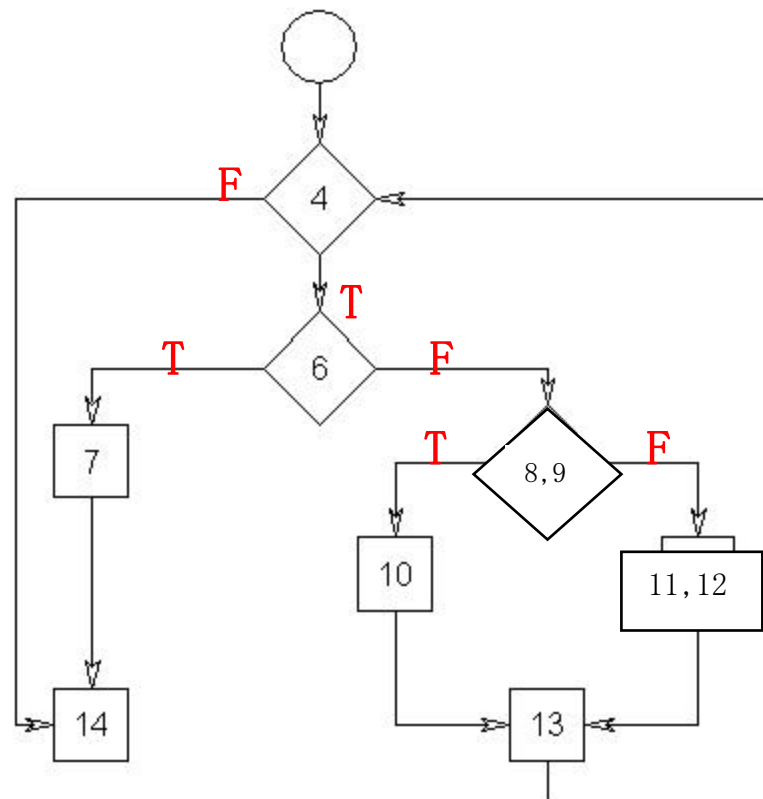
# 基本路径测试例子

- 在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。  
包括以下**4**个步骤：
  1. 以设计或源代码为基础，画出相应的流图
  2. 确定所得流图的环复杂度
  3. 确定独立路径的基本集合
  4. 准备测试用例，执行基本集合中每条路径

# 例1： 第一步画出控制流图

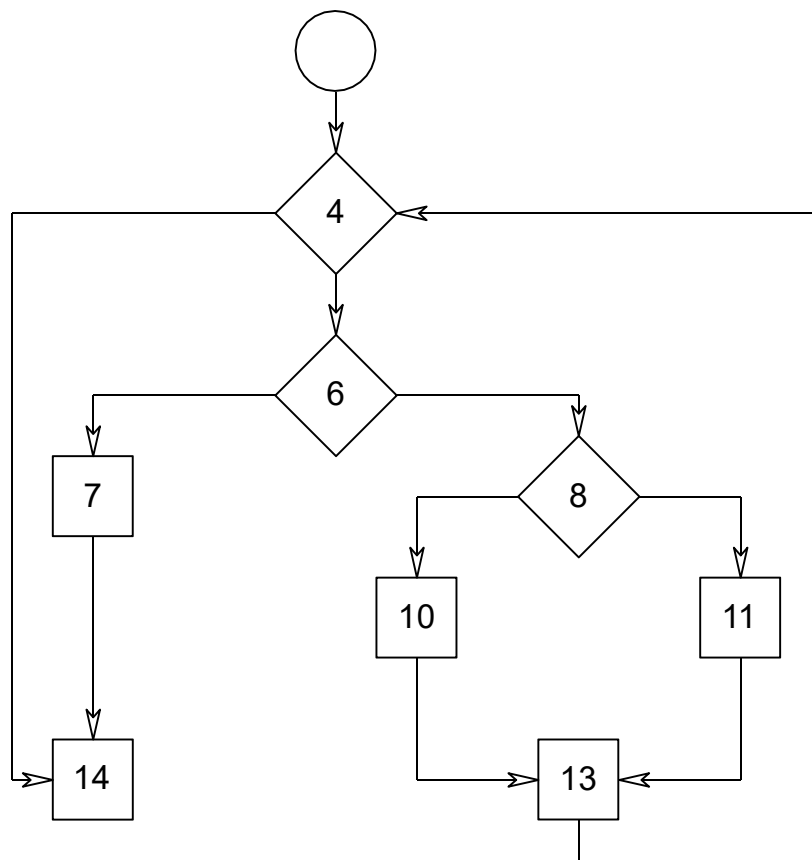
例： 有下面的C函数， 用基本路径测试法进行测试

```
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. }
```

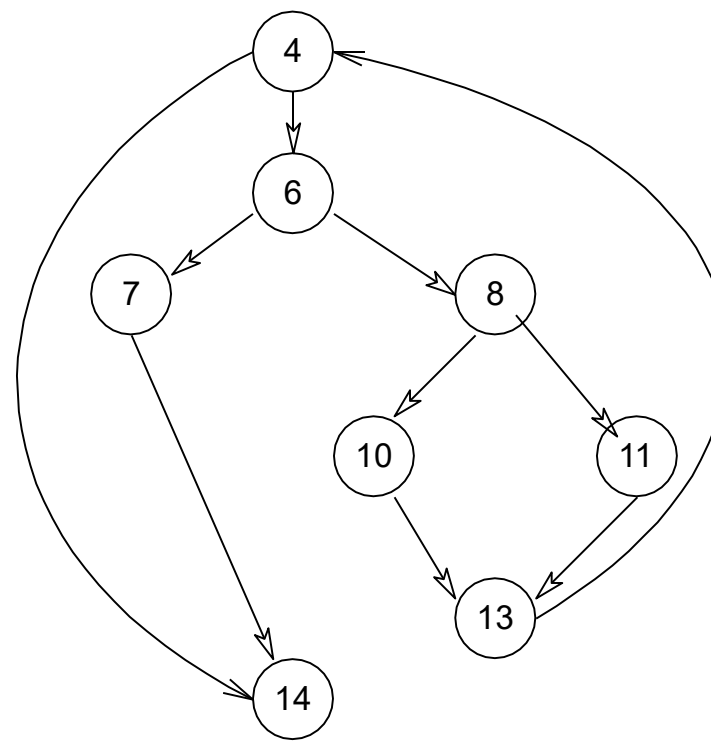




# 例1： 第一步画出控制流图



程序流程图

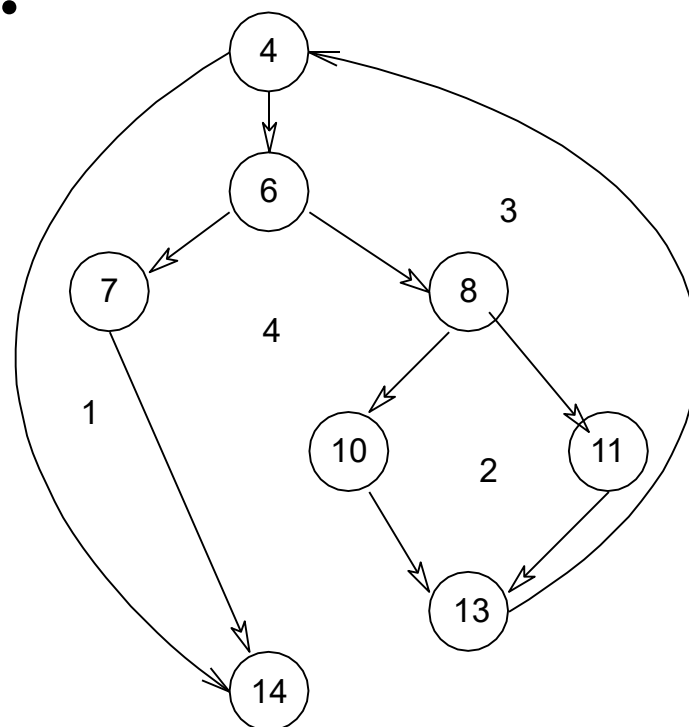


控制流图

## 例1： 第二步计算环复杂度

■ 对应上图中的环复杂度， 计算如下：

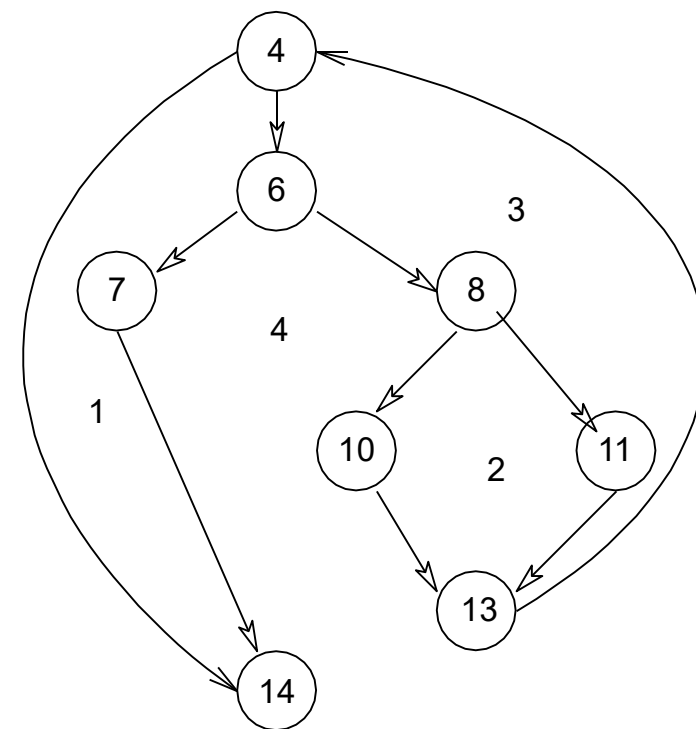
- 流图中有4个区域；
- $V(G) = 10 \text{ 条边} - 8 \text{ 结点} + 2 = 4$ ；
- $V(G) = 3 \text{ 个判定结点} + 1 = 4$ 。



区域： 由边和节点封闭起来的区域  
计算区域： 不要忘记区域外的部分

# 例1： 第三步确定独立路径的基本集合

- 根据上面的计算方法，可得出四个独立的路径。  
(一条独立路径是指，和其他的独立路径相比，至少引入一个新处理语句或一个新判断的程序通路。  
 $V(G)$  值正好等于该程序的独立路径的条数。)
  - 路径1：4-14
  - 路径2：4-6-7-14
  - 路径3：4-6-8-10-13-4-14
  - 路径4：4-6-8-11-13-4-14
- 根据上面的独立路径，去设计输入数据，使程序分别执行到上面四条路径。





# 例1： 第四步准备测试用例

路径1: 4-14

输入数据: iRecordNum=0, 或者取  
iRecordNum<0的某一个值

预期结果: x=0

路径2: 4-6-7-14

输入数据: iRecordNum=1,iType=0

预期结果: x=2

路径3: 4-6-8-10-13-4-14

输入数据: iRecordNum=1,iType=1

预期结果: x=10

路径4: 4-6-8-11-13-4-14

输入数据: iRecordNum=1,iType=2

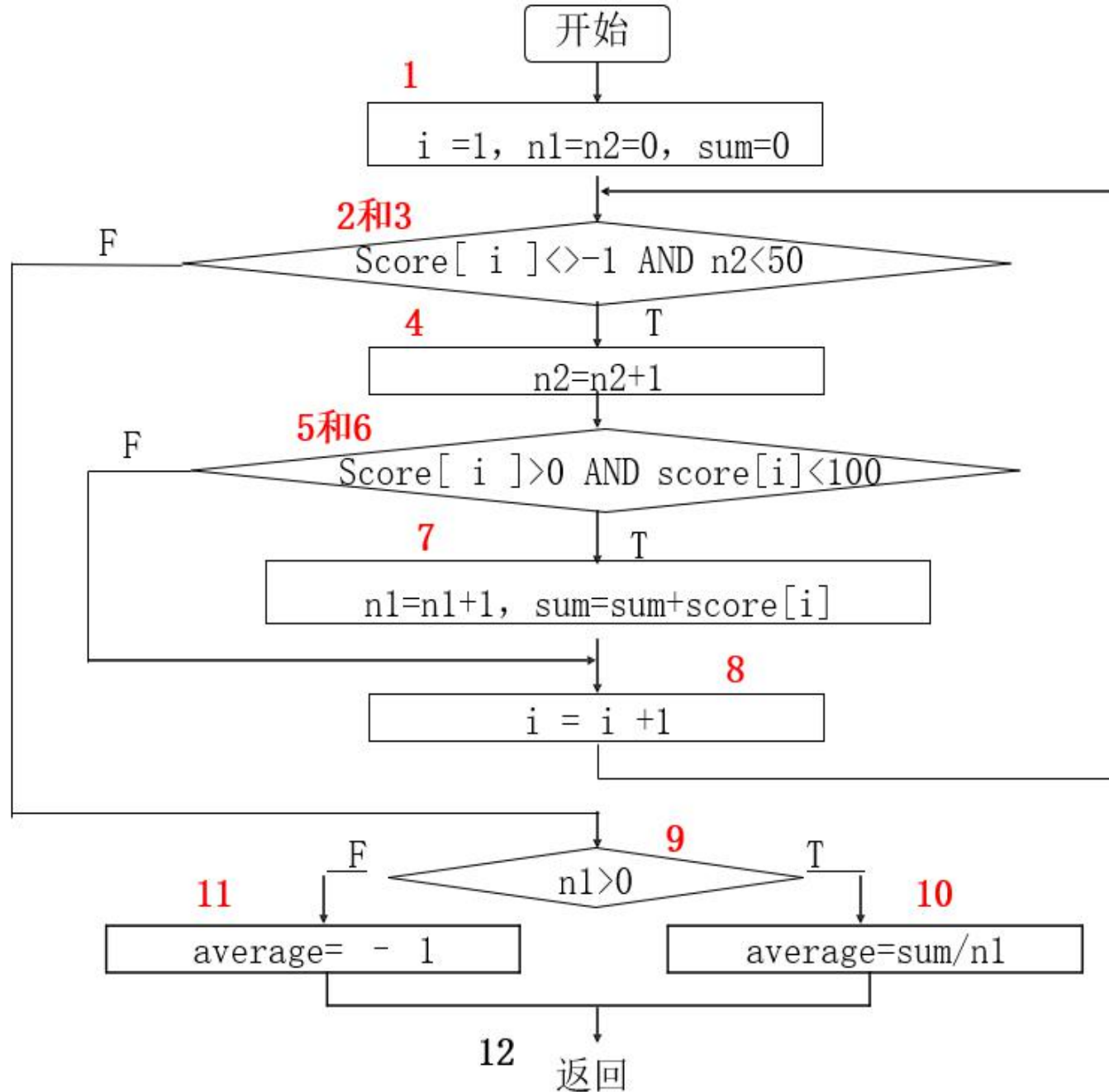
预期结果: x=20

```
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. }
```

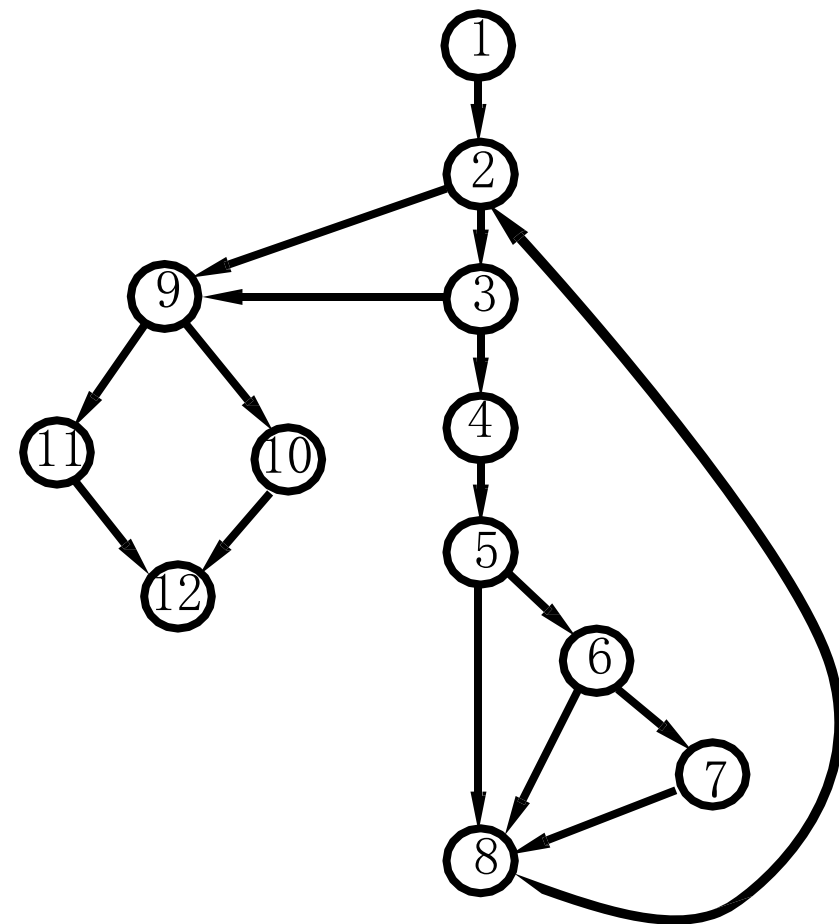
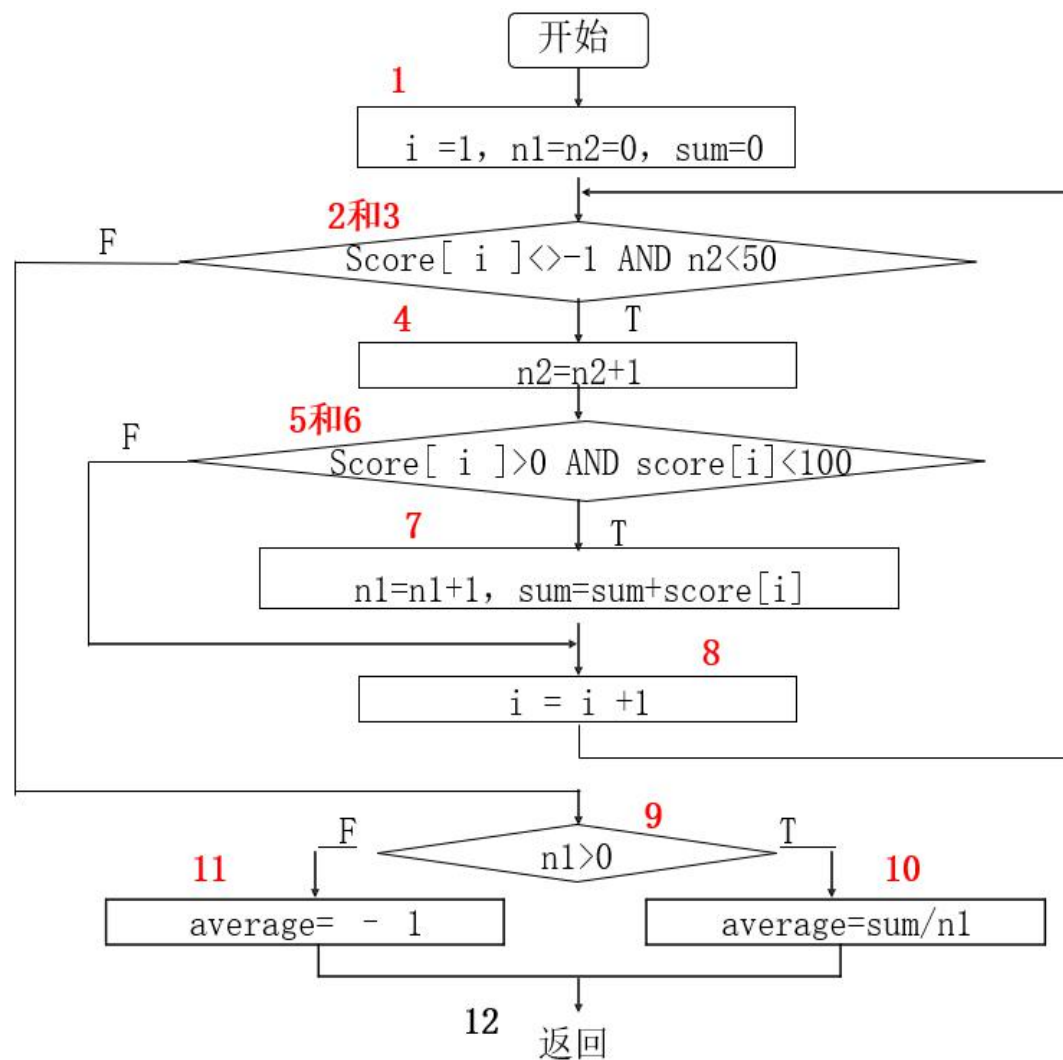
## 例2：学生成绩统计

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。





## 例2： 第一步画出控制流图





## 例2： 第二步计算圈复杂度

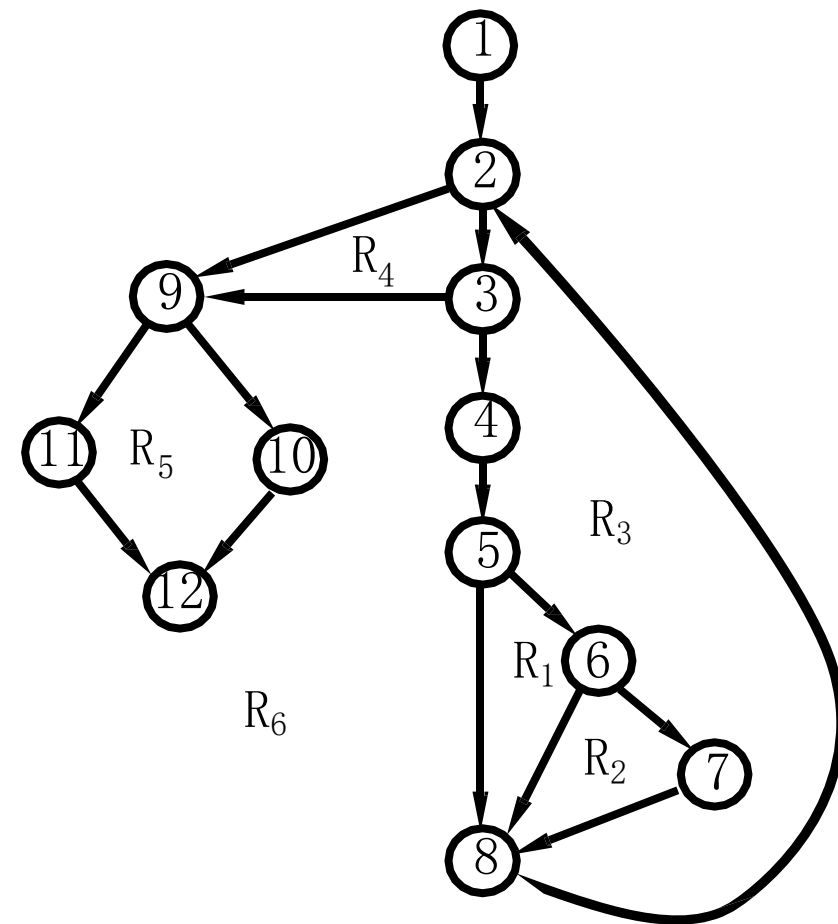
$V(G) = 6$  个区域

$$V(G) = E - N + 2 = 16 - 12 + 2 = 6$$

其中E为流图中的边数，N为结点数；

$$V(G) = P + 1 = 5 + 1 = 6$$

其中P为判定结点的个数。在流图中，结点2、3、5、6、9是判定结点。



## 例2： 第三步确定基本路径集合

路径1： 1-2-9-10-12

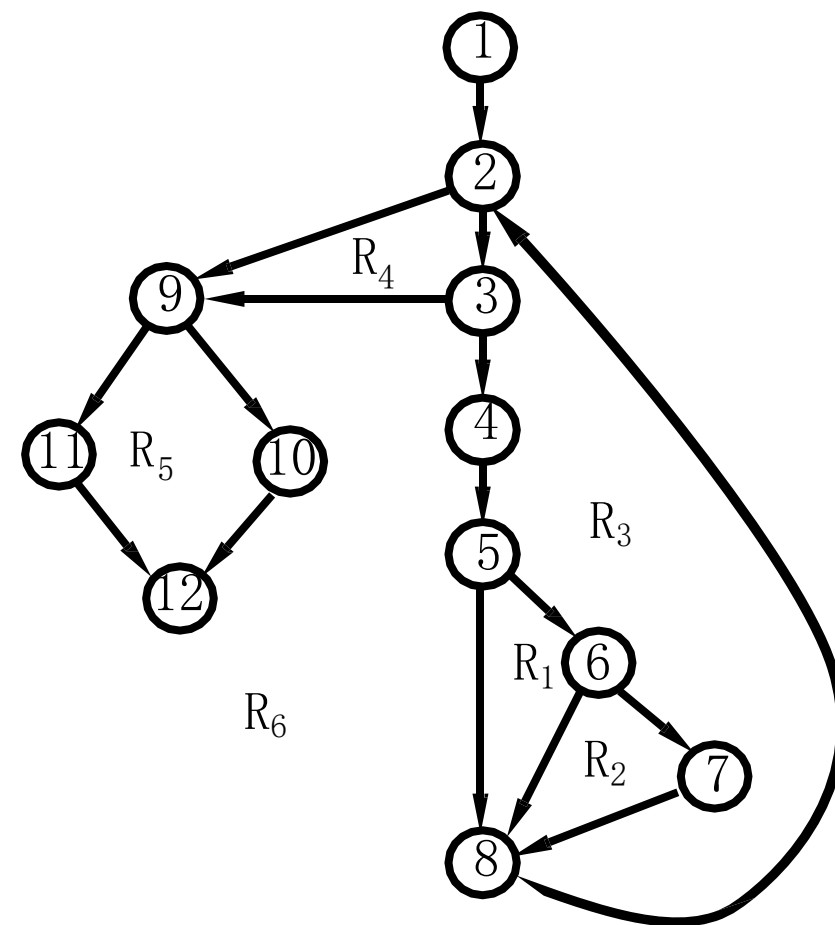
路径2： 1-2-9-11-12

路径3： 1-2-3-9-10-12

路径4： 1-2-3-4-5-8-2...

路径5： 1-2-3-4-5-6-8-2...

路径6： 1-2-3-4-5-6-7-8-2...

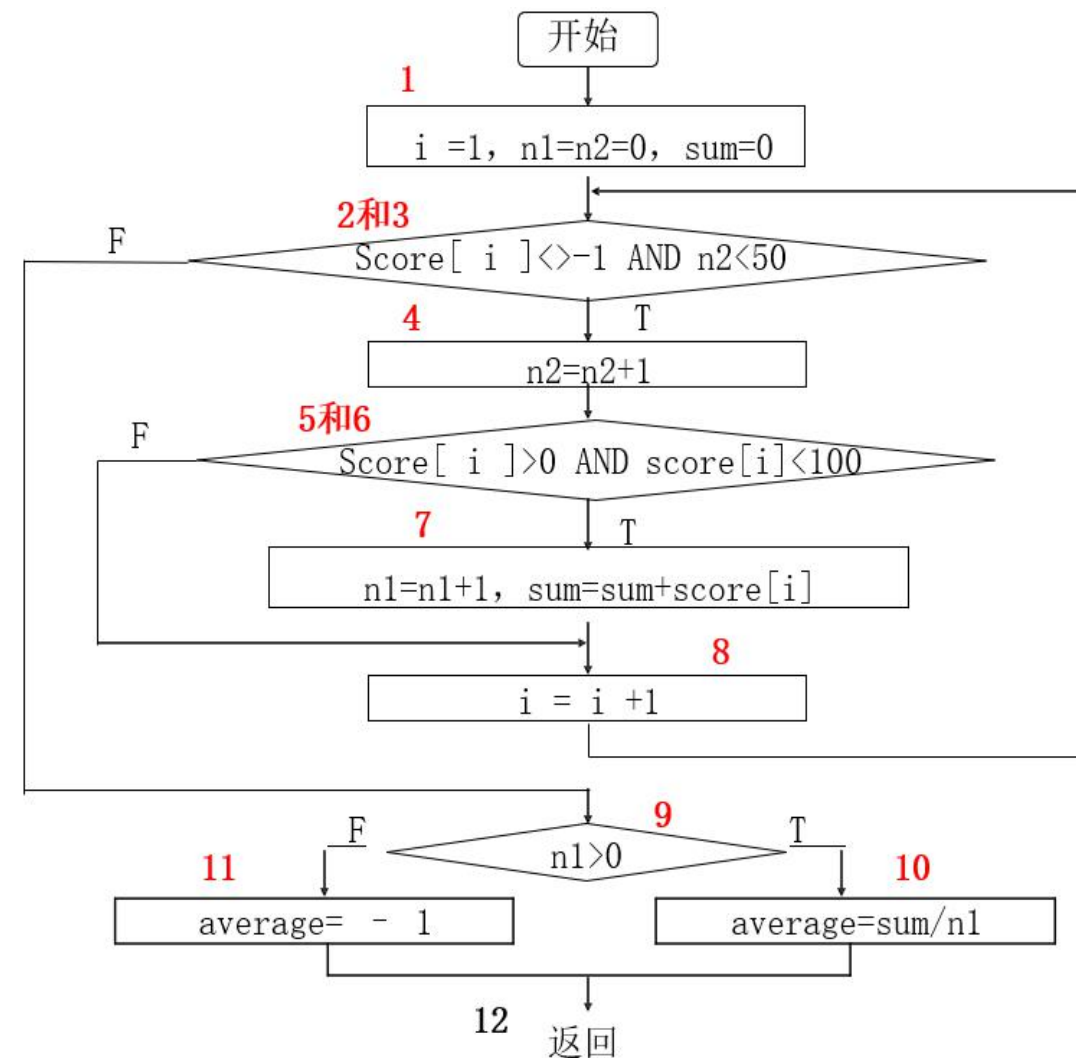


## 例2： 第四步导出测试用例

- 路径1(1-2-9-10-12)的测试用例：

达不到这个路径！

某些独立路径不能以独立的方式测试，也就是说，程序正常流程不能形成独立执行该路径所需要的数据组合。在这种情况下，**这个路径必须作为另一个路径的一部分来测试。**



## 例2：第四步导出测试用例

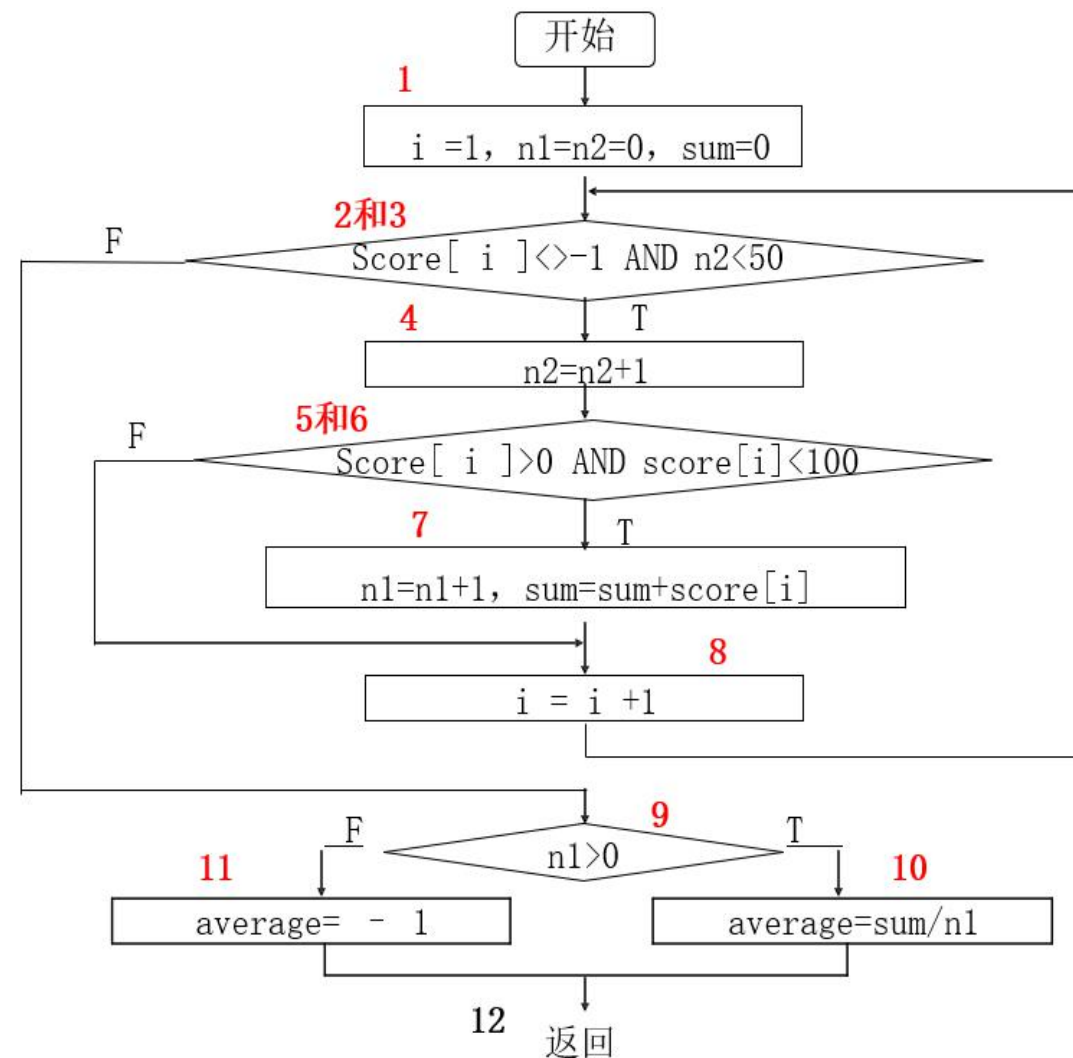
- 路径1(1-2-...-2-9-10-12)的测试用例：

$\text{score}[k]$ =有效分数值, 当  $k < i$ ;

$\text{score}[i] = -1$ ,  $2 \leq i \leq 50$ ;

Example: [1,3,5,6,-1]

- 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 $\text{sum}$ 和平均分 $\text{average}$ 。

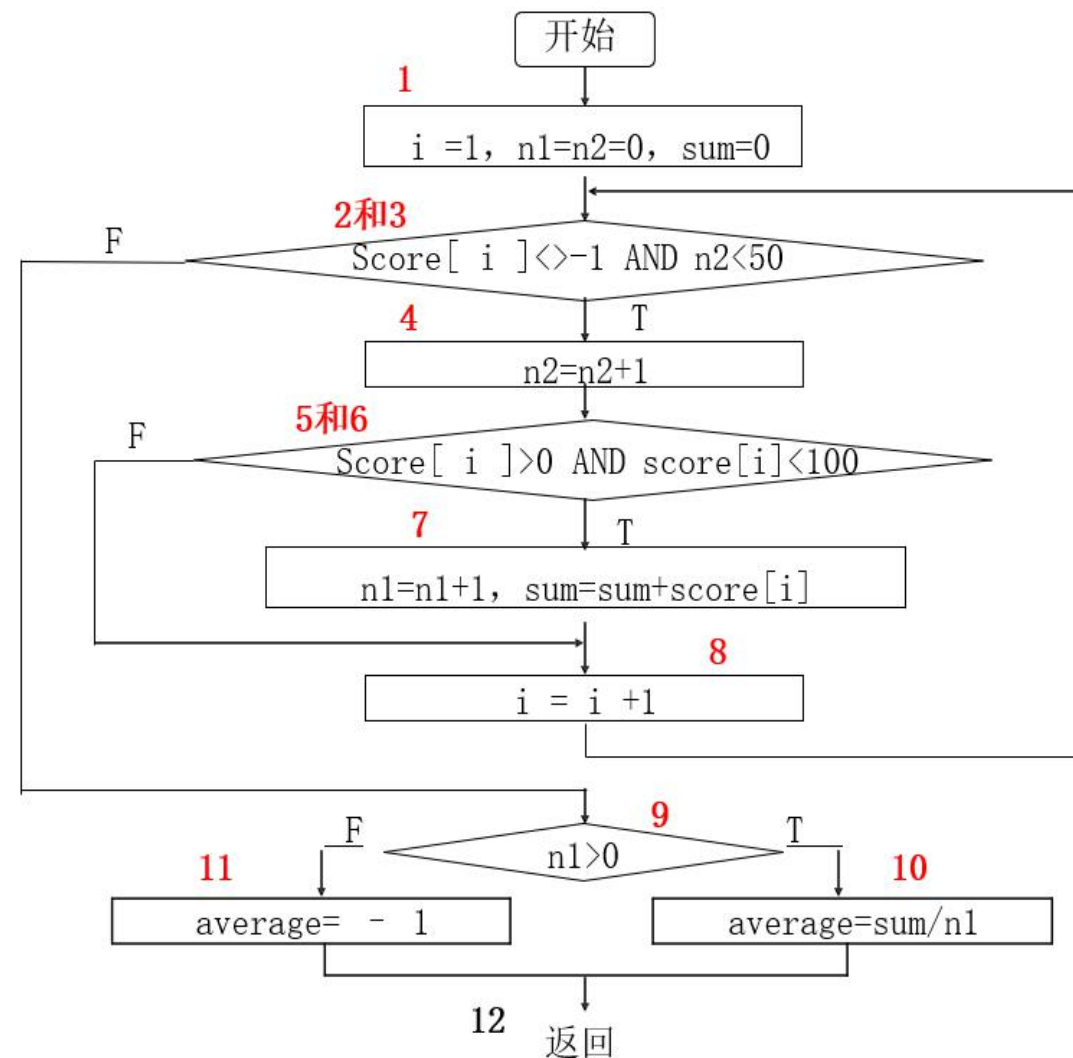


## 例2：第四步导出测试用例

- 路径2(1-2-9-11-12)的测试用例：

$\text{score}[1] = -1;$

- 期望结果：average = -1，其他量保持初始值。



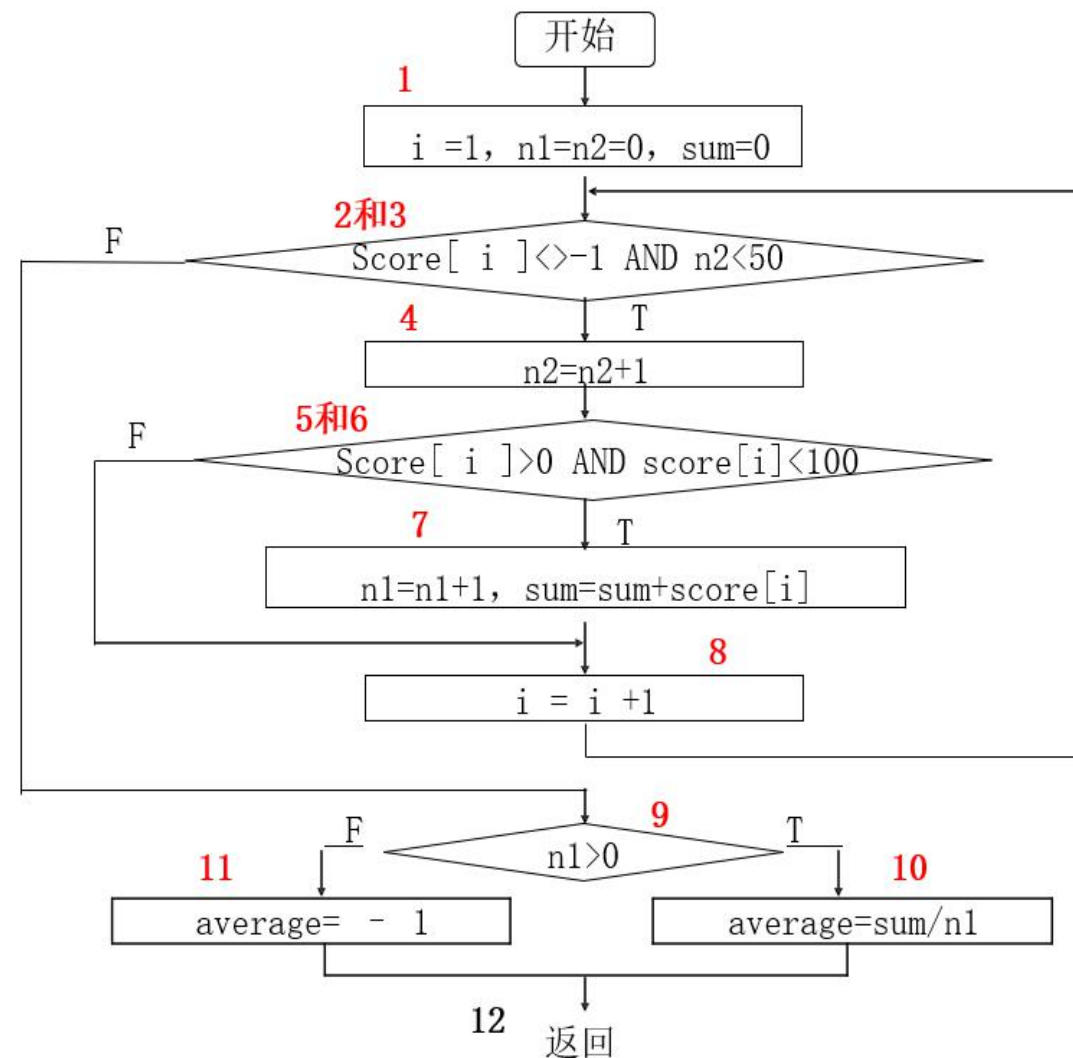
## 例2： 第四步导出测试用例

- 路径3 (1-2-...-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，；

Example=[1,4,6...,80,100,-1]

- 期望结果：n1=50且算出正确的总分和平均分。





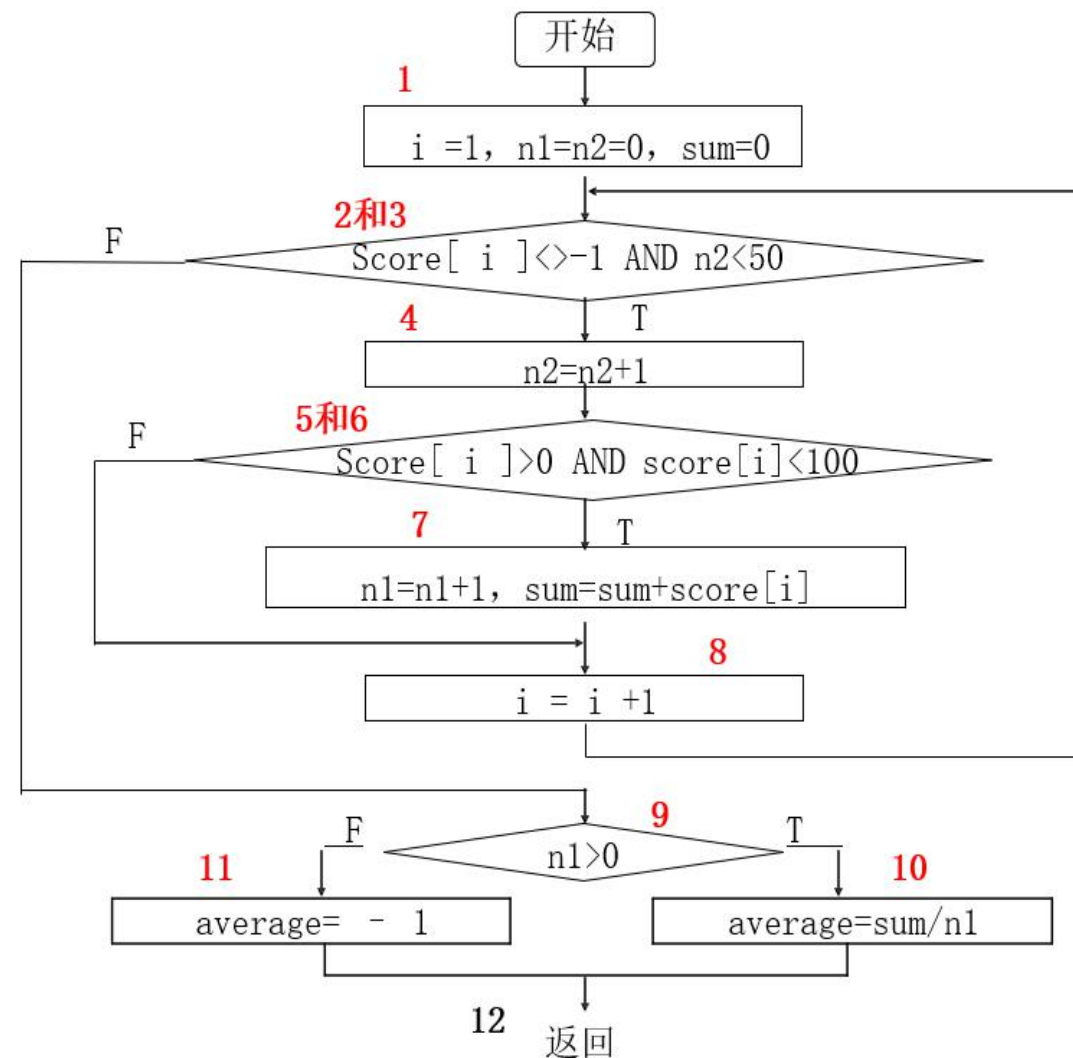
## 例2： 第四步导出测试用例

- 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,  $k < i$  ;

Example: [1,3,5,-9,-1]

- 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。





## 例2： 第四步导出测试用例

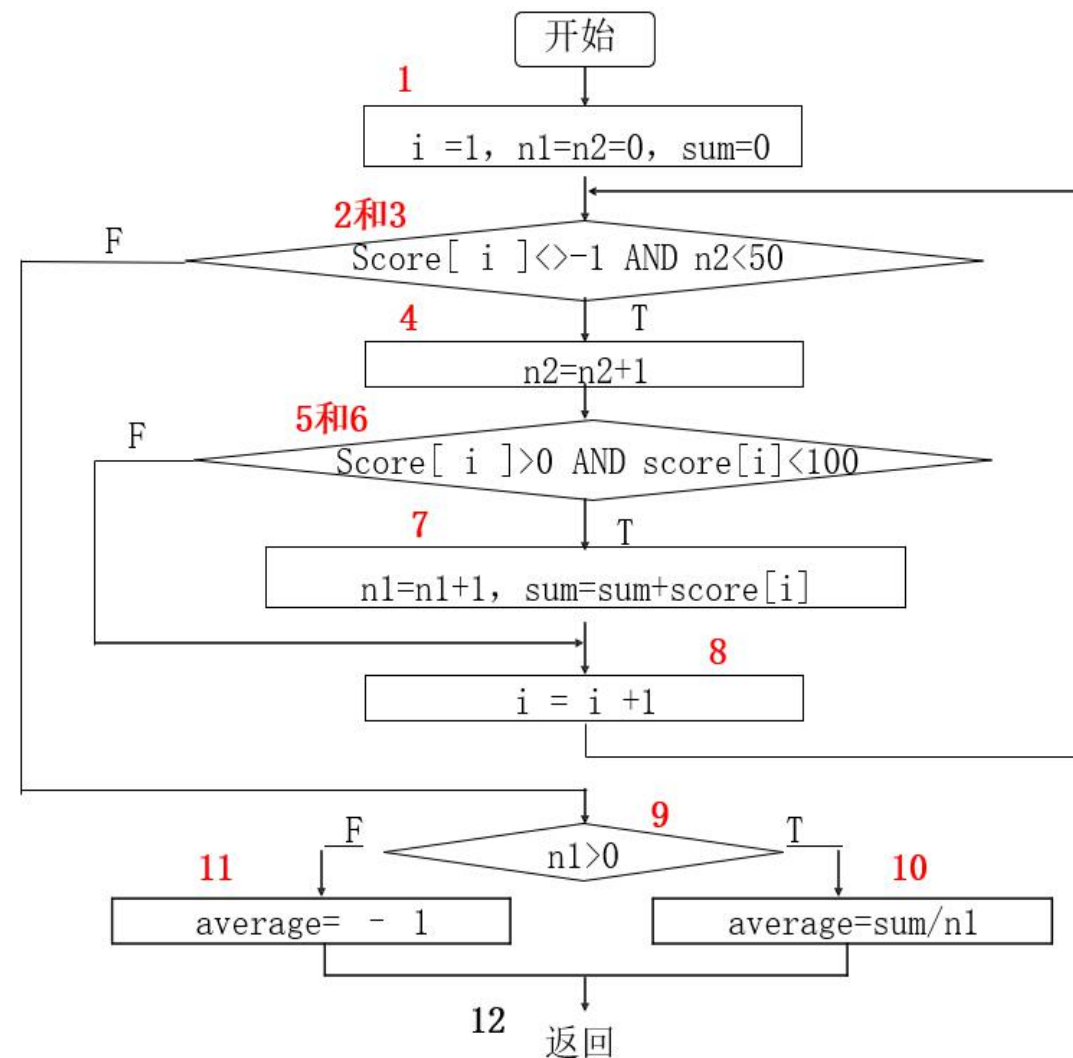
- 路径5 (1-2-3-4-5-6-8-2...)的测试用例:

score[i]=有效分数, 当 $i < 50$ ;

score[k]>100,  $k < i$  ;

Example: [1,3,5,101,-1]

- 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。





## 例2： 第四步导出测试用例

- 路径6(1-2-3-4-5-6-7-8-2...)的测试用例：

score[i]=有效分数, 当 $i < 50$ ;

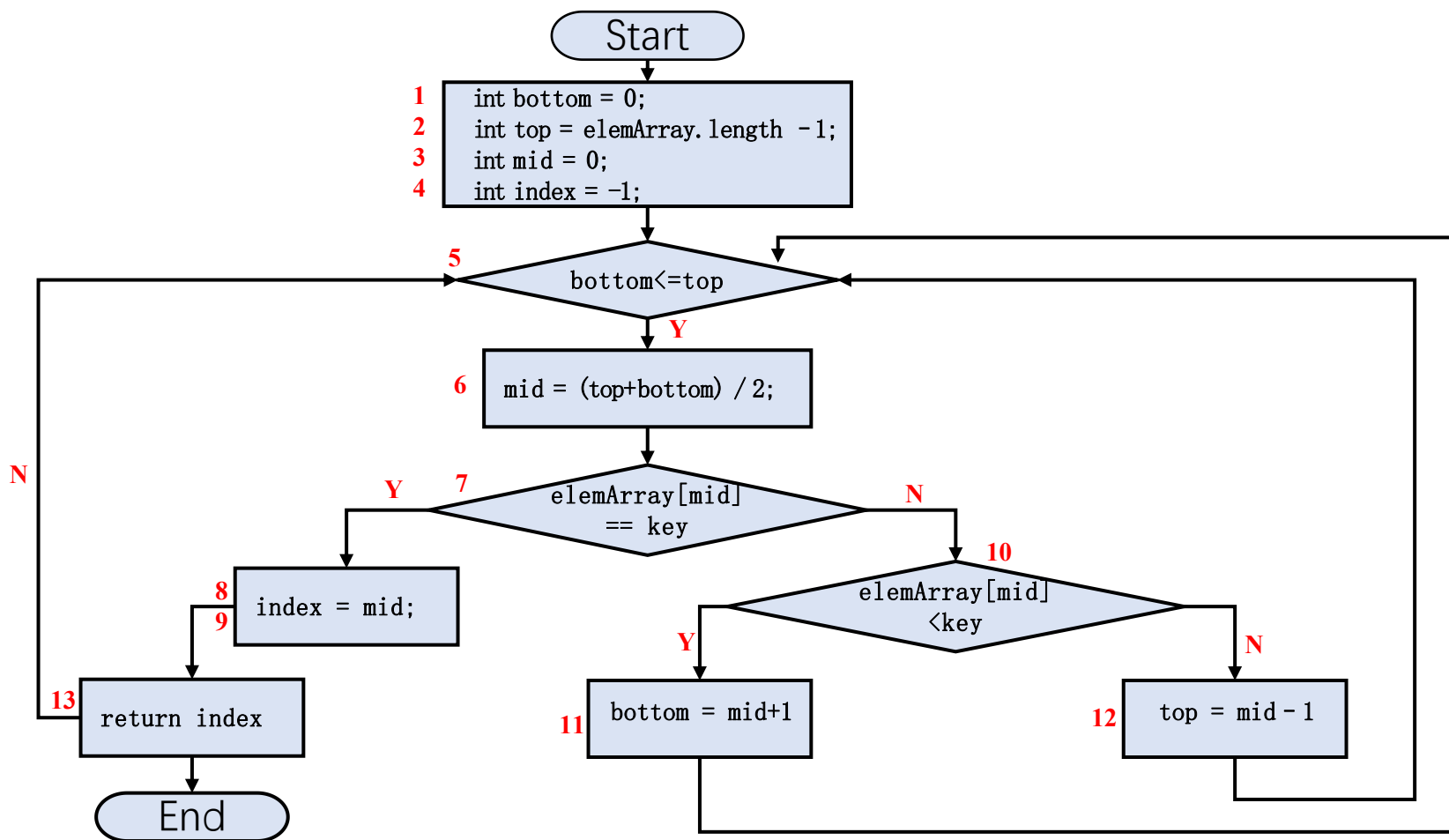
- 期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

## 例3：二分搜索法

- 某算法的程序伪代码如下所示，它完成的基本功能是：
  - 输入一个自小到大顺序排列的整型数组 **elemArray** 和一个整数 **key**，算法通过二分搜索法查询 **key** 是否在 **elemArray** 中出现；
  - 若找到，则在 **index** 中记录 **key** 在 **elemArray** 中出现的位置；
  - 若找不到，则为 **index** 赋值 -1。算法将 **index** 作为返回值。

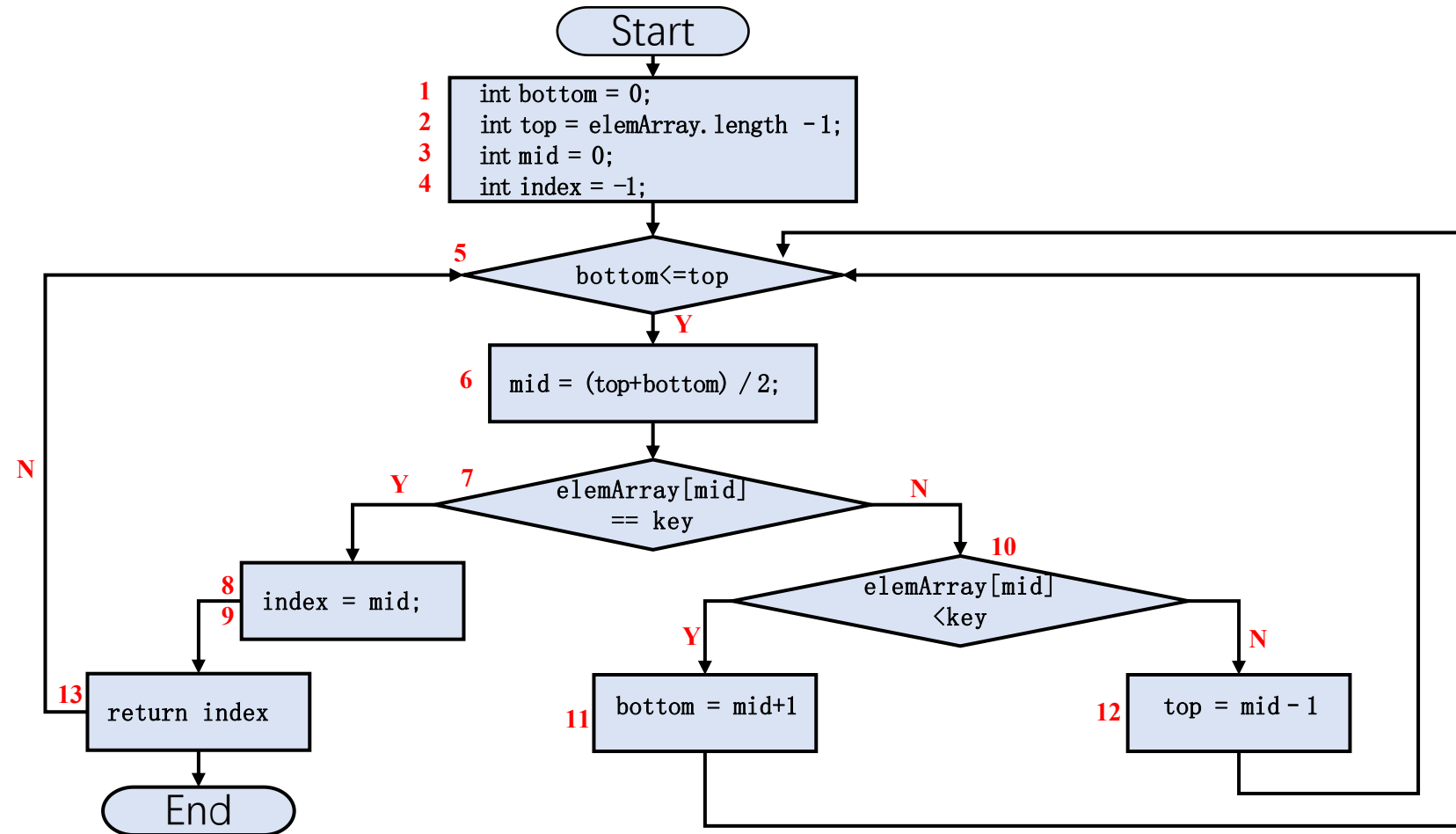
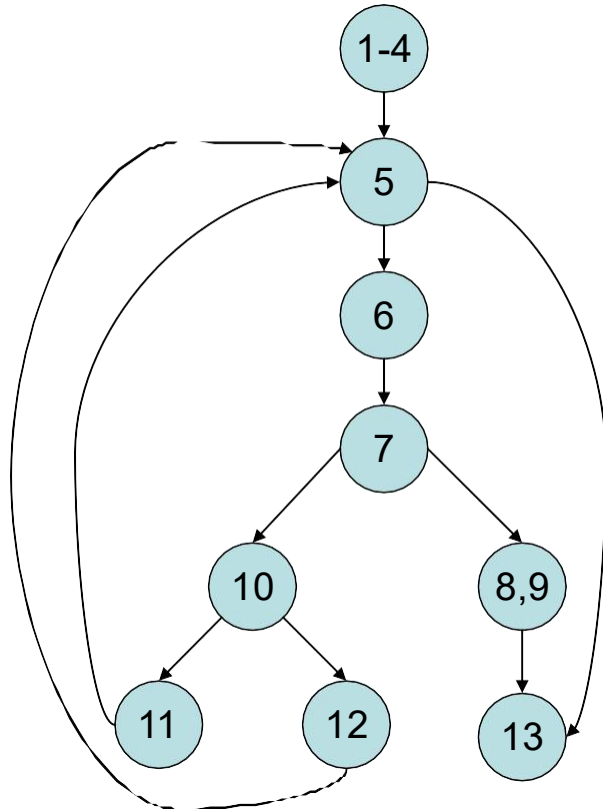
```
int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bommom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }
       else
       {
10          if (elemArray [mid] < key)
11              bottom = mid + 1;
12          else
13              top = mid - 1;
       }
   }
   return index;
}
```

## 例3：模块程序流程图



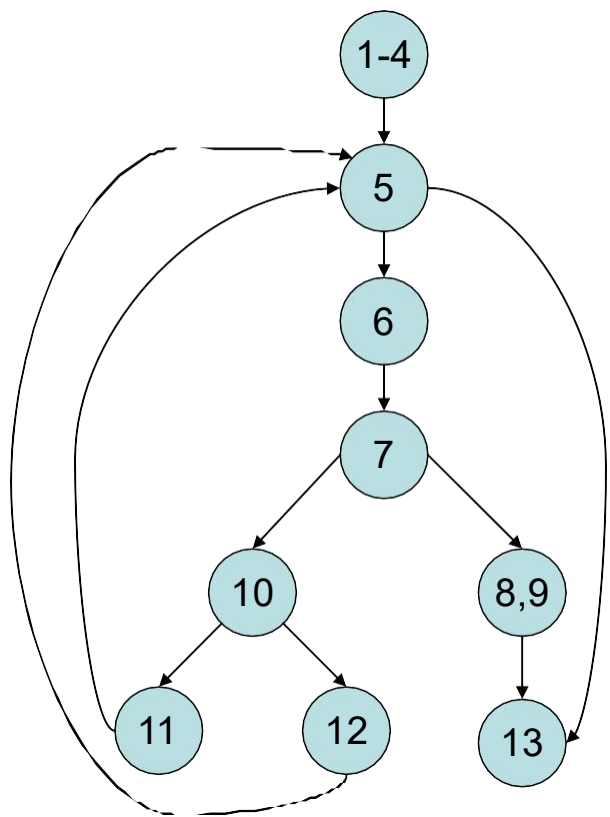
```
int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bommom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }
      else
      {
10          if (elemArray [mid] < key)
11              bottom = mid + 1;
          else
12              top = mid - 1;
      }
13  }
  return index;
}
```

## 例3：控制流图





## 例3: $V(G)$ 及独立路径



- $V(G)=4$ 个区域
- $V(G)=3$ 个判断节点+1
- $V(G)=11$ 条边-9个节点+2
- 独立路径:
  - 路径1: 1-4, 5, 13
  - 路径2: 1-4, 5, 6, 7, 8, 9, 13
  - 路径3: 1-4, 5, 6, 7, 10, 11, 5, 13
  - 路径4: 1-4, 5, 6, 7, 10, 12, 5, 13

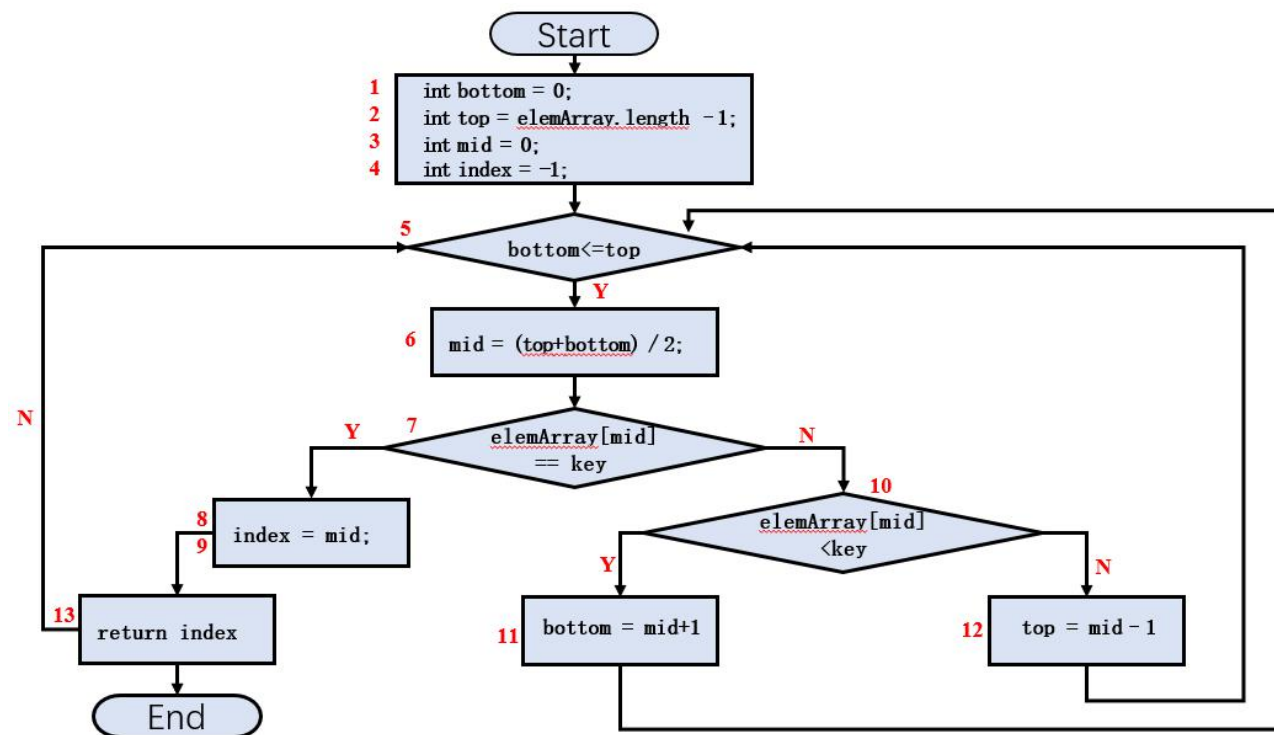
## 例3：测试用例

### ■ 独立路径：

- 路径1: 1-4, 5, 13
- 路径2: 1-4, 5, 6, 7, 8, 9, 13
- 路径3: 1-4, 5, 6, 7, 10, 11, 5, 13
- 路径4: 1-4, 5, 6, 7, 10, 12, 5, 13

### ■ 测试用例：

序号	输入: elemArray	输入: Key	期望输出: index
1		20	-1
2	1, 2, 3, 4, 5	3	2
3	1, 2, 3, 4, 5	4	3
4	1, 2, 3, 4, 5	2	1





# 小结

## ■ 使用路径测试技术设计测试用例的步骤如下：

- 根据过程设计结果画出相应的流图
- 计算流图的环形复杂度
- 确定独立路径的基本集合
- 设计可强制执行基本集合中每条路径的测试用例

## ■ 注意：

- 某些独立路径不能以独立的方式被测试(即穿越路径所需的数据组合不能形成程序的正常流)。在这种情况下，**这些路径必须作为另一个路径测试的一部分来进行测试。**
- “圈复杂度”表示：**只要最多 $V(G)$ 个测试用例就可以达到基本路径覆盖，但并非一定要设计 $V(G)$ 个用例。**
- 但是：**测试用例越简化，代表测试越少、可能发现的错误就越少。**



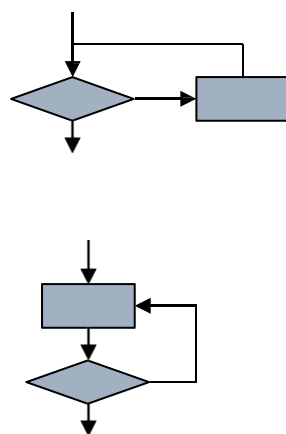


# 控制结构测试

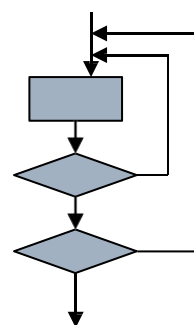
- 基本路径测试是控制测试技术之一，尽管基本路径测试是简单且有效的，但其本身并不充分。
- 控制结构测试提高了白盒测试的质量
  - 逻辑测试
  - 循环测试

# 循环测试

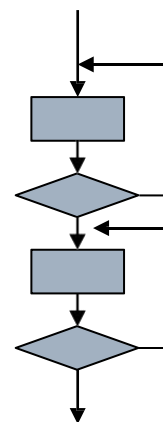
- 循环测试是一种白盒测试技术，注重于循环构造的有效性。
- 四种循环：简单循环，串接(连锁)循环，嵌套循环和不规则循环



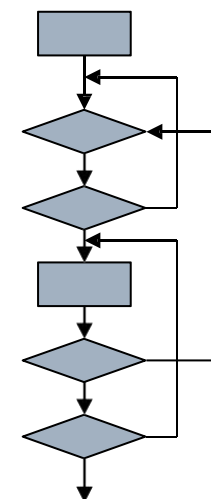
简单循环



嵌套循环



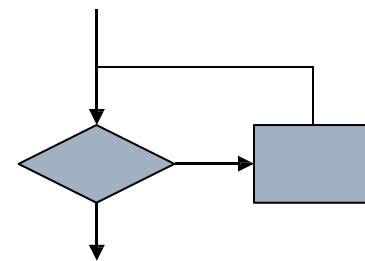
串接循环



无结构循环

# 简单循环

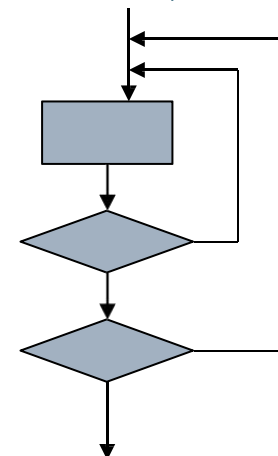
- 对于简单循环，测试应包括以下几种，其中的 $n$ 表示循环允许的最大次数。
  - 零次循环：从循环入口直接跳到循环出口。
  - 一次循环：查找循环初始值方面的错误。
  - 二次循环：检查在多次循环时才能暴露的错误。
  - $m$ 次循环：此时的 $m < n$ ，也是检查在多次循环时才能暴露的错误。
  - $n$ (最大)次数循环、 $n+1$ (比最大次数多一)次的循环、 $n-1$ (比最大次数少一)次的循环。



# 嵌套循环

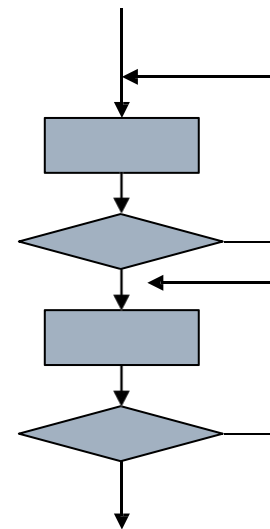
## ■ 对于嵌套循环：

- 从最内层循环开始，设置所有其他层的循环为最小值；
- 对最内层循环做简单循环的全部测试。测试时保持所有外层循环的循环变量为最小值。另外，对越界值和非法值做类似的测试。
- 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。
- 反复进行，直到所有各层循环测试完毕。
- 对全部各层循环同时取最小循环次数，或者同时取最大循环次数。对于后一种测试，由于测试量太大，需人为指定最大循环次数。



# 串接循环

- 对于串接循环，要区别两种情况。
  - 如果各个循环互相独立，则串接循环可以用与简单循环相同的方法进行测试。
  - 如果有两个循环处于串接状态，而前一个循环的循环变量的值是后一个循环的初值。则这几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。





# 非结构循环

- 对于非结构循环，不能测试，应重新设计循环结构，使之成为其它循环方式，然后再进行测试。

