



软件测试

第4章 白盒测试

白盒测试概念



- 白盒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。通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。

白盒测试技术



■ 本章主要内容

- ✧ 几种常见的白盒测试覆盖技术（语句，判定，条件，判定/条件，组合条件，路径覆盖）
- ✧ 控制流的测试覆盖技术（基本路径）
- ✧ 程序切片测试技术
- ✧ 变异测试

白盒测试



■ 程序结构表示方法

✧流程图

✧N-S图

✧控制流图

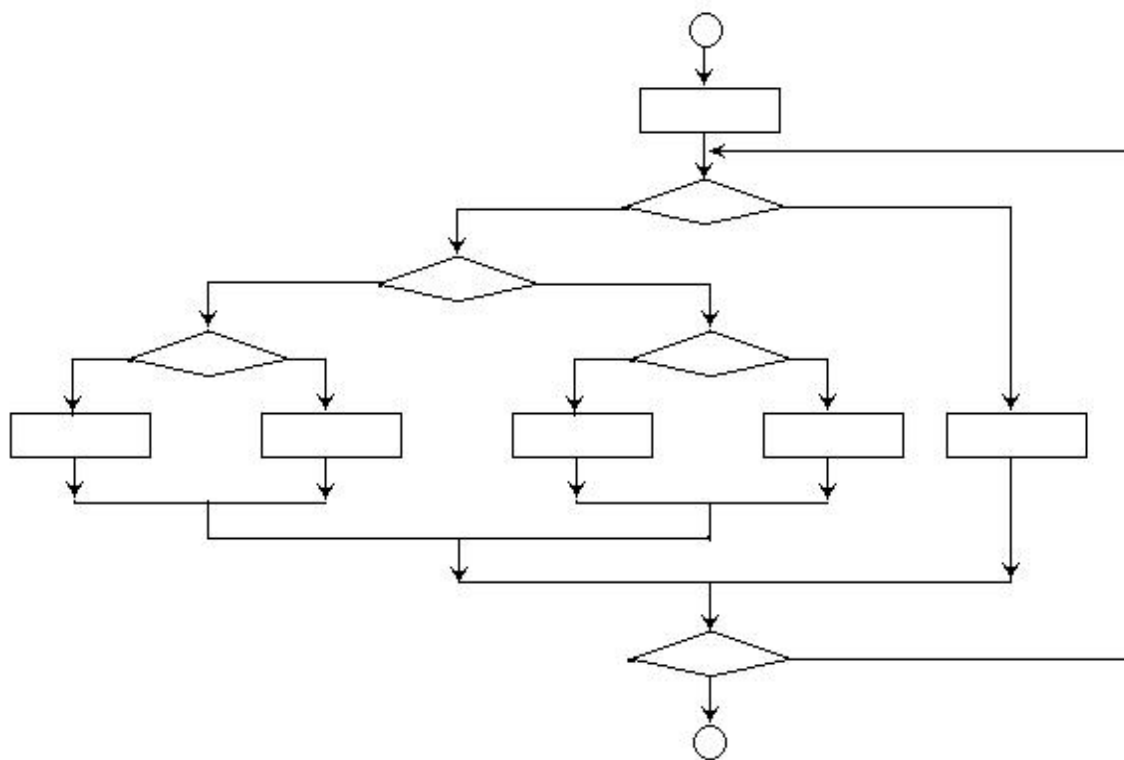
✧数据流图

本章主要内容



- 4.1 语句级覆盖
- 4.2 路径级覆盖
- 4.3 切片技术
- 4.4 变异技术

对所有程序逻辑路径测试？



循环 ≤ 20 次

测试覆盖标准



- ❖ 上页小程序的流程图，20次的循环。不同路径数高达 5^{20} ($=10^{13}$) 条。
 - 5分钟编写、执行和确认一测试用例需要1亿年。
 - 1秒编写、执行和确认一测试用例需要32万年。

4.1 语句级覆盖



■ 语句覆盖

■ 判定覆盖

■ 条件覆盖

■ 判定/条件覆盖

■ 条件组合覆盖

■ 路径覆盖

面向单语句的测试准则

面向语句间控制的测试准则

4.1.1 语句覆盖



```
Float fun(float A, float B, float X)
{
    if (A>1) && (B==0) X=X/A;
    if (A==2) || (X>1) X=X+1;
    return X;
}
```

4.1.1 语句覆盖



❖ 语句覆盖:

选取足够多的测试数据，使被测试程序中每个语句至少执行一次。

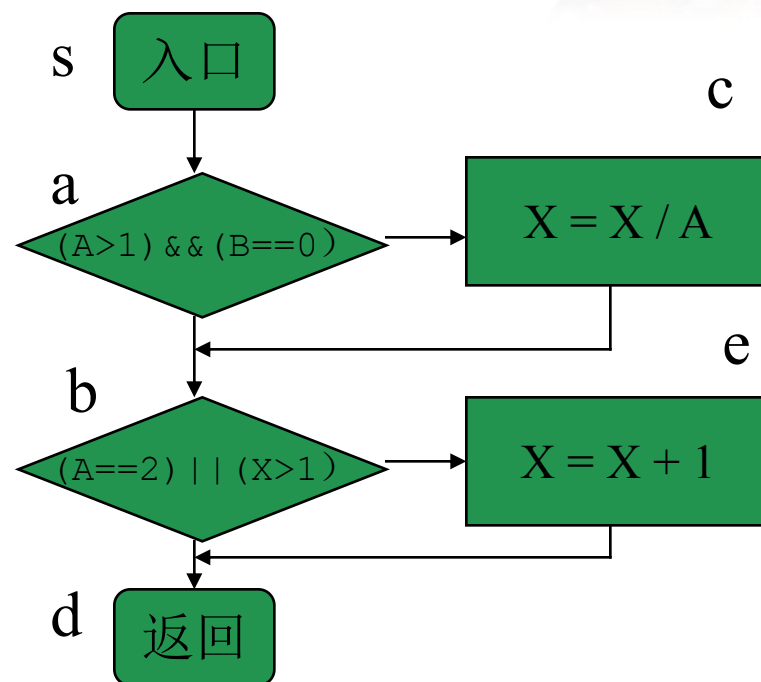
❖ 图中存在4条不同的路径:

- L1: sacbed
- L2: sabd
- L3: sabed
- L4: sacbd

❖ 为使每个语句都执行一次，程序的执行路径应是sacbed:

A=2, B=0, X=4

测试用例:



语句覆盖



- ❖ 第一个条件语句中的**AND**错误地编写成**OR**，上面的测试用例是不能发现这个错误的；
- ❖ 第二个条件语句中 $X > 1$ 误写成 $X > 0$ ，这个测试用例也不能暴露它；
- ❖ 沿着路径**abd**执行时，**X**的值应该保持不变，如果这一方面有错误，上述测试数据也不能发现。
- ❖ 一般认为“语句覆盖”是很不充分的一种标准，是最弱的逻辑覆盖准则。

4.1.2 判定覆盖



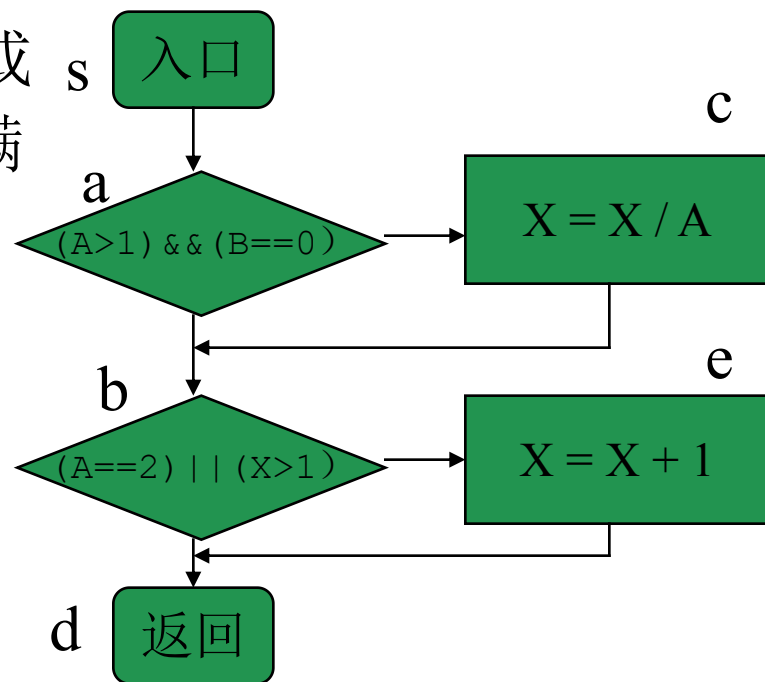
- 判定覆盖：选取足够多的测试数据，使被测试程序中不仅每个语句至少执行一次，而且每个判定的每种可能的结果都至少执行一次。

- 能够分别覆盖路径sacbed和sabd或sacbd和sabed的两组测试数据，都满足判定覆盖标准：

(1) $A=3, B=0, X=3$ (sacbd)

(2) $A=2, B=1, X=1$ (sabed)

测试用例



判定覆盖



- ❖ 程序中含有判定的语句包括**IF-THEN-ELSE**、**DO-WHILE**、**REPEAT-UNTIL**等，除了双值的判定语句外，还有多值的判定语句，如**CASE**语句。所以“判定覆盖”更一般的含义是：使得每一个判定获得每一种可能的结果。
- ❖ 判定覆盖并不能够覆盖每个条件语句正确。例如： **$A > 1$ AND $B > 0$, $A > 1$ OR $B > 0$**
- ❖ “判定覆盖”比“语句覆盖”严格，因为如果每个判定都执行过了，则每个语句也就执行过了。但是，“判定覆盖”还是很不够的，例如两个测试用例未能检查沿着路径**abd**执行时，**X**的值是否保持不变。

4.1.3 条件覆盖

- ◆ 条件覆盖：选取足够多的测试数据，使被测试程序中不仅每个语句至少执行一次，而且每个判定表达式中的每个条件都取到各种可能的结果。

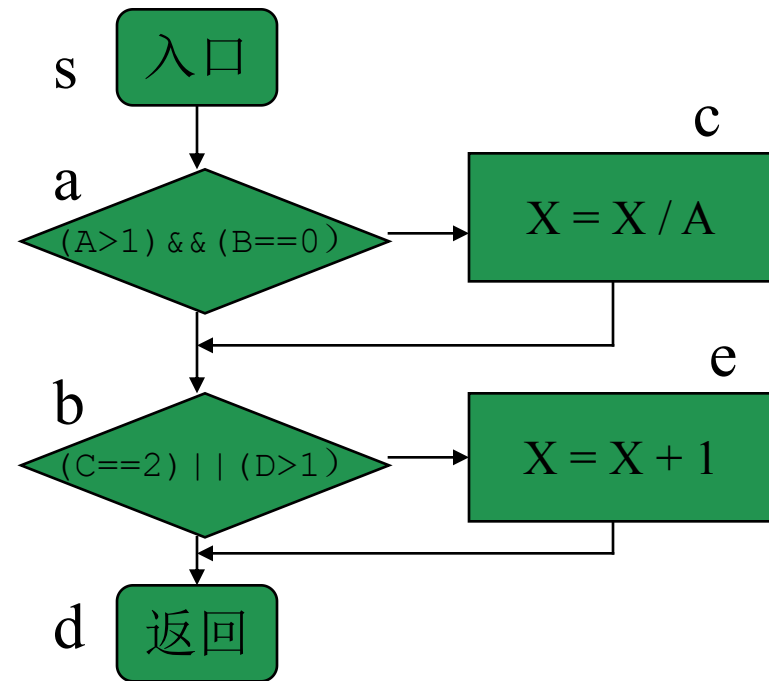
在a点： $A > 1$, $A \leq 1$, $B = 0$, $B \neq 0$;

在b点： $A = 2$, $A \neq 2$, $X > 1$, $X \leq 1$ 。

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

(2) $A = 1$, $B = 1$, $X = 1$ (sabd)

测试用例



条件覆盖



- ❖ “条件覆盖”通常比“判定覆盖”强，因为它使一个判定中的每一个条件都取到了两个不同的结果，而判定覆盖则不保证这一点。
- ❖ “条件覆盖”并不包含“判定覆盖”，如对语句 **IF(A AND B) THEN S** 设计测试用例使其满足“条件覆盖”，即使**A**为真并使**B**为假，以及使**A**为假而且**B**为真，但是它们都未能使语句**S**得以执行。

4.1.4 判定/条件覆盖

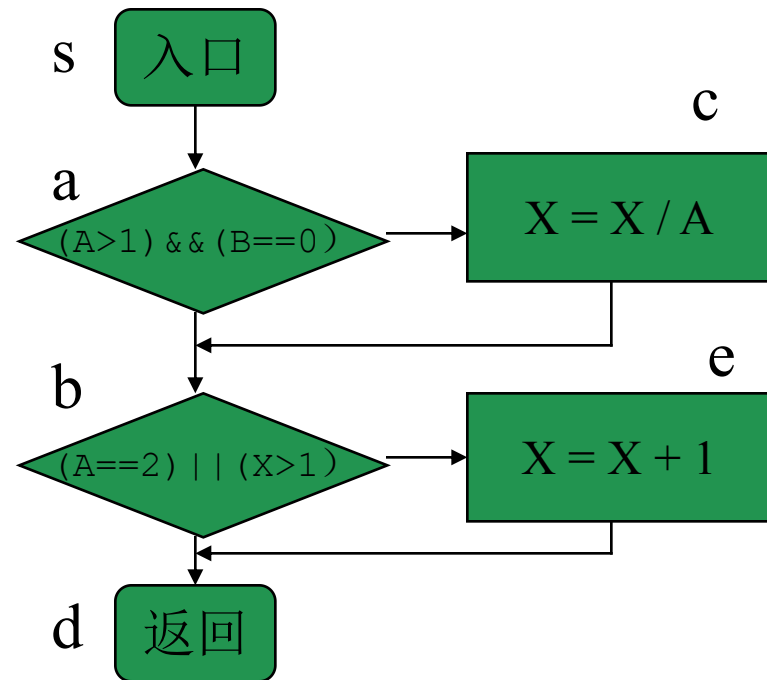


- ◆ 判定/条件覆盖：选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的结果，而且每个判定表达式也都取到各种可能的结果。

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

(2) $A=1, B=1, X=1$ (sabd)

测试用例



判定/条件覆盖



- ❖ 判定/条件覆盖从表面来看，它测试了所有条件的取值，但是实际上某些条件掩盖了另一些条件
- ❖ 例如对于条件表达式 $(x > 3) \&\& (z < 10)$ 来说，必须两个条件都满足才能确定表达式为真。如果 $(x > 3)$ 为假则一般的编译器不在判断是否 $z < 10$ 了。对于第二个表达式 $(x == 4) \parallel (y > 5)$ 来说，若 $x == 4$ 测试结果为真，就认为表达式的结果为真，这时不再检查 $(y > 5)$ 条件了。因此，采用判定/条件覆盖，逻辑表达式中的错误不一定能够查出来了。
- ❖ $(A \&\& B)$ 运算，使用 $A=T$ 分别对应于 B 的 T 和 F ，使用 $A=F$ 对应于 B 的任意； $A \parallel B$ 运算，使用 $A=F$ 分别对应于 B 的 T 和 F ，使用 $A=T$ 对应于 B 的任意；)

4.1.5 MC/DC覆盖



❖ MC/DC (Modified Condition/Decision Coverage)

Executing the independent *true* and *false* outcomes of each condition.

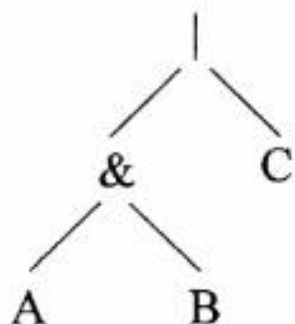
❖ A&&B , A||B 的MC/DC 设计

MC/DC覆盖

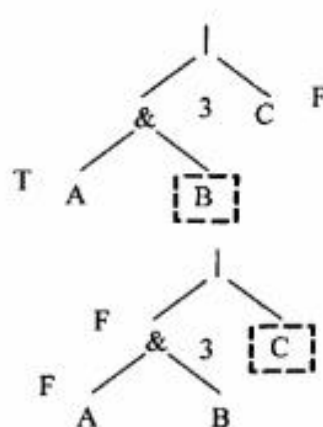
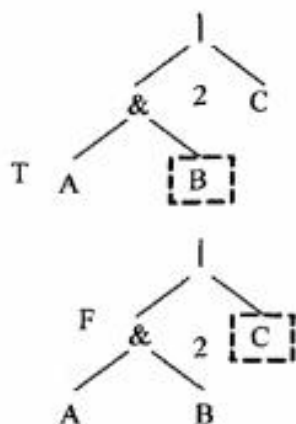
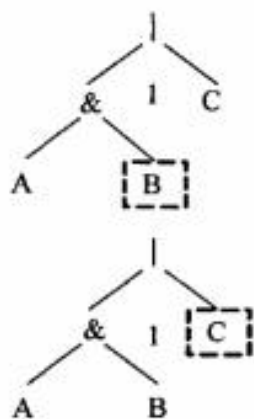
❖ 判别树分析法



$(A \& B) \mid C$



	(A	&	B)	C
1	T		T	F
2	F		T	F
3	T		F	F
4	T		F	F
5	F		F	T
6	F		F	F



4.1.6 条件组合覆盖



❖ 条件组合覆盖：选取足够多的测试数据，使得判定表达式中条件的各种可能组合都至少出现一次。

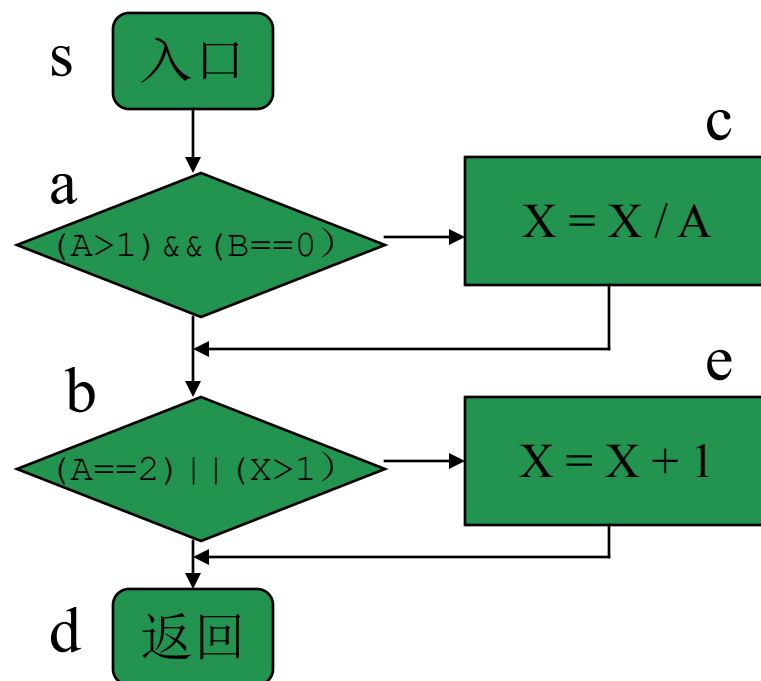
❖ 有八种可能的条件组合：

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

❖ 测试数据：

- (1) $A=2, B=0, X=4$ (sacbed, 1,5)
- (2) $A=2, B=1, X=1$ (sabed, 2,6)
- (3) $A=1, B=0, X=2$ (sabed, 3,7)
- (4) $A=1, B=1, X=1$ (sabed, 4,8)

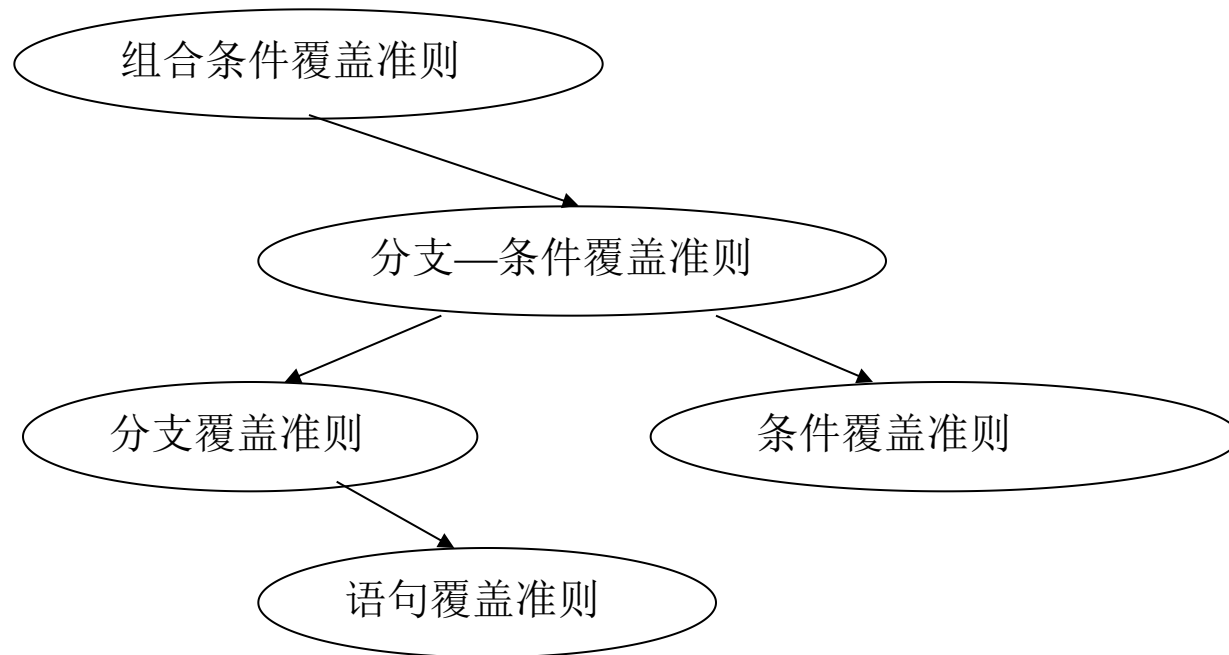
测试用例



测试覆盖关系



- ❖ 部分覆盖准则间的关系
- ❖ 路径测试与点的测试的覆盖关系？



4.2 路径覆盖

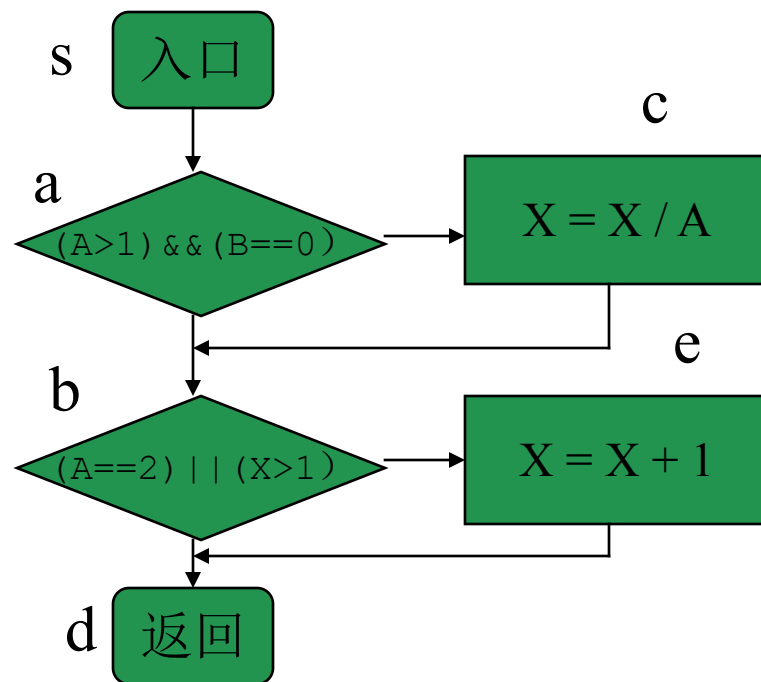


❖ 路径覆盖：选取足够多的测试数据，使得程序的每条可能路径都至少执行一次。

❖ 测试数据：

- (1) $A=1, B=1, X=1$ (sabd)
- (2) $A=1, B=1, X=2$ (sabed)
- (3) $A=3, B=0, X=1$ (sacbd)
- (4) $A=2, B=0, X=4$ (sacbed)

测试用例



4.2 路径覆盖



❖ 路径覆盖的难点

- ✓ 判定问题
- ✓ 循环问题

❖ 路径覆盖的一般处理方法

- ✓ 实际循环次数处理
- ✓ 选择循环次数处理
- ✓ 考虑图结构特征的处理

4.2.1 实际循环次数处理



❖ 结构分析（循环+分支+跳转）

❖ 实际循环次数

❖ 符号执行

4.2.2 选择循环次数测试



■ 简单循环测试方法(假设有 n 次循环):

- ✧ 整个跳过循环
- ✧ 只有一次通过循环
- ✧ 两次通过循环
- ✧ 某个 m 次通过循环, $m < n$
- ✧ $n-1, n, n+1$ 次通过循环

4.2.3 基于图结构的路径测试



■ 图结构

✓ 图 $G = (V, E)$ ，节点与边，可能存在环路

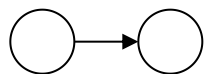
■ 图结构处理策略（Mccab圈复杂度）

✓ 一个图的基本路径集合数目是确定，图中的任意路径都可以表示为基本路径的线性组合。

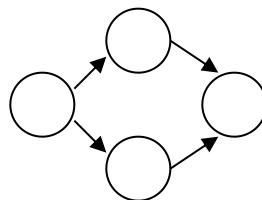
4.2.3 基本路径测试



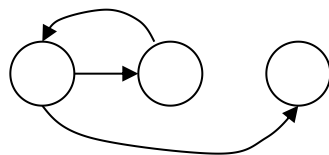
❖ 控制流图



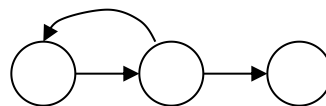
顺序结构



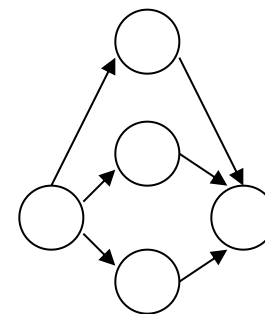
if 结构



while 结构



until 结构



Case 结构

4.2.3 基本路径测试--程序转控制流图



❖ 判定语句分解为条件

- 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:

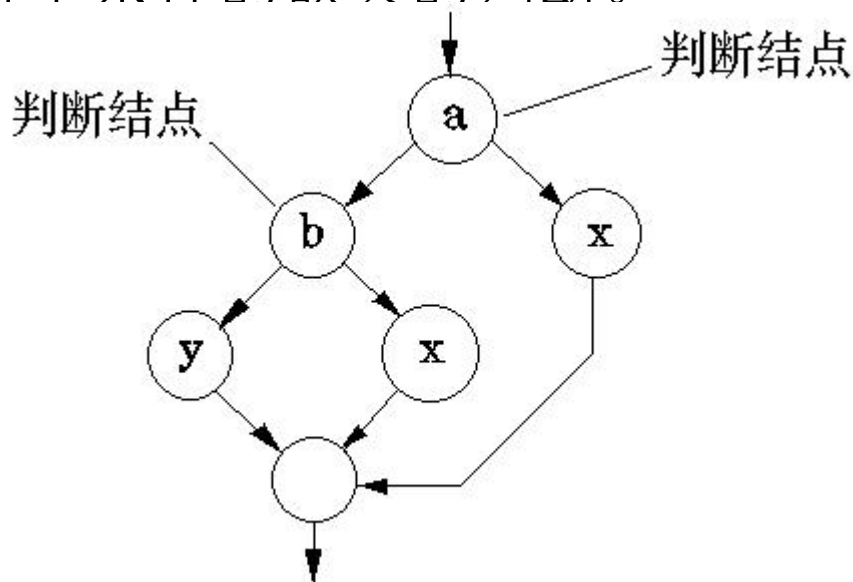
1 if a or b

2 x

3 else

4 y

对应的逻辑为:



❖ 顺序语句合并为单节

4.2.3 基本路径测试程序转控制流图



```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med  = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:           " + length);
    System.out.println ("mean:           " + mean);
    System.out.println ("median:         " + med);
    System.out.println ("variance:       " + var);
    System.out.println ("standard deviation: " + sd);
}
```


4.2.3 基本路径测试-程序转控制流图



```
public static void computeStats (int [ ] numbers)
```

```
{  
    int length = numbers.length;  
    double med, var, sd, mean, sum, varsum;
```

```
    sum = 0;
```

```
    for (int i = 0; i < length; i++)
```

```
    {  
        sum += numbers [ i ];
```

```
    }  
    med = numbers [ length / 2];  
    mean = sum / (double) length;
```

```
    varsum = 0;
```

```
    for (int i = 0; i < length; i++)
```

```
    {  
        varsum = varsum + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
```

```
    }  
    var = varsum / ( length - 1.0 );  
    sd = Math.sqrt ( var );
```

```
    System.out.println ("length: " + length);  
    System.out.println ("mean: " + mean);  
    System.out.println ("median: " + med);  
    System.out.println ("variance: " + var);  
    System.out.println ("standard deviation: " + sd);  
}
```



$i = 0$

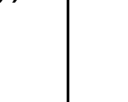


$i \geq \text{length}$



$i < \text{length}$

$i++$



$i = 0$



$i < \text{length}$

$i \geq \text{length}$



$i++$



4.2.3 基本路径测试--综合性实验作业



- 给定JAVA程序，自动绘制控制流图。
- 要求：
 - ✓ 能够实现三种基本结构的控制流图
 - ✓ 能够实现循环嵌套的控制流图
 - ✓ 能够实现循环与分支嵌套的控制流图
 - ✓ 能够以合适的数据结构存储图与路径信息

4.2.3 基本路径测试--基本路径计算

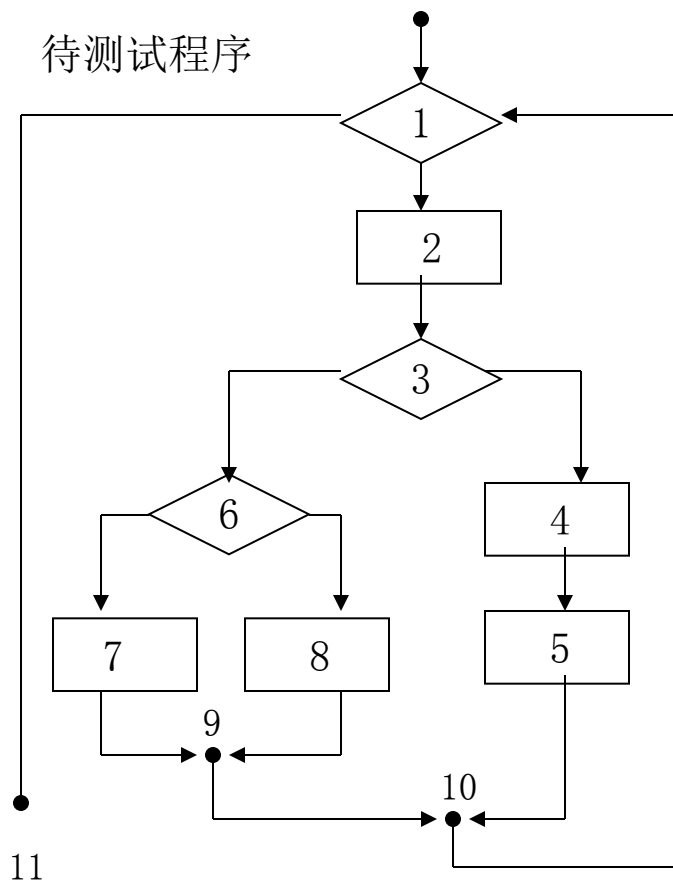


节点

边

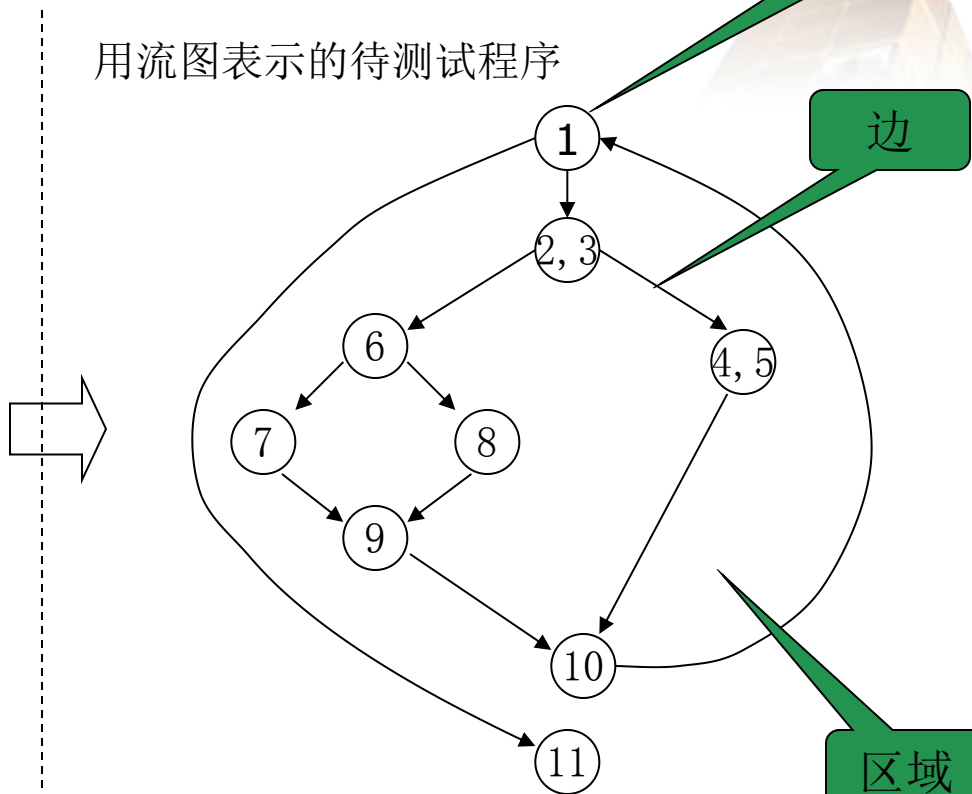
区域

待测试程序



11

用流图表示的待测试程序

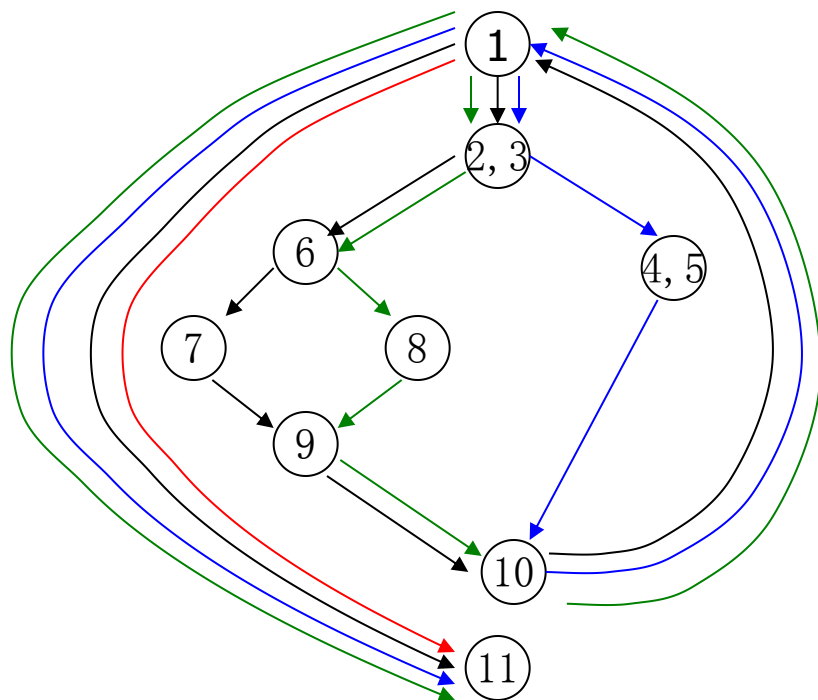


区域：由边和解点封闭起来的区域
计算区域：不要忘记区域外的部分

4.2.3 基本路径测试--基本路径计算



基本路径：至少沿一条新的边移动的路径



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。

4.2.3基本路径测试 - 计算圈复杂度



❖ 圈复杂度

有以下三种方法计算圈复杂度：

1. 流图中区域的数量对应于环型的复杂性；
2. 给定流图 G 的圈复杂度 $V(G)$ ，定义为 $V(G)=E-N+2$ ， E 是流图中边的数量， 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 是流图 G 中判定结点的数量。

4.2.3 基本路径测试—计算原则



❖ 三个原则

- 每条基本路径需要包含一条**新**边
- 每条基本路径不能**新**出现同一个判定节点的**TRUE**和**False**
- 每条基本路径的尽可能**少**的改变原基本路径的判定方向。

4.2.3 基本路径测试—举例



❖ 例1：如复杂条件多判定分支

```
Float fun(float A,float  
B,float X)  
{  
    if (A>1) && (B==0) X=X/A;  
    if (A==2) || (X>1) X=X+1;  
    return X;  
}
```

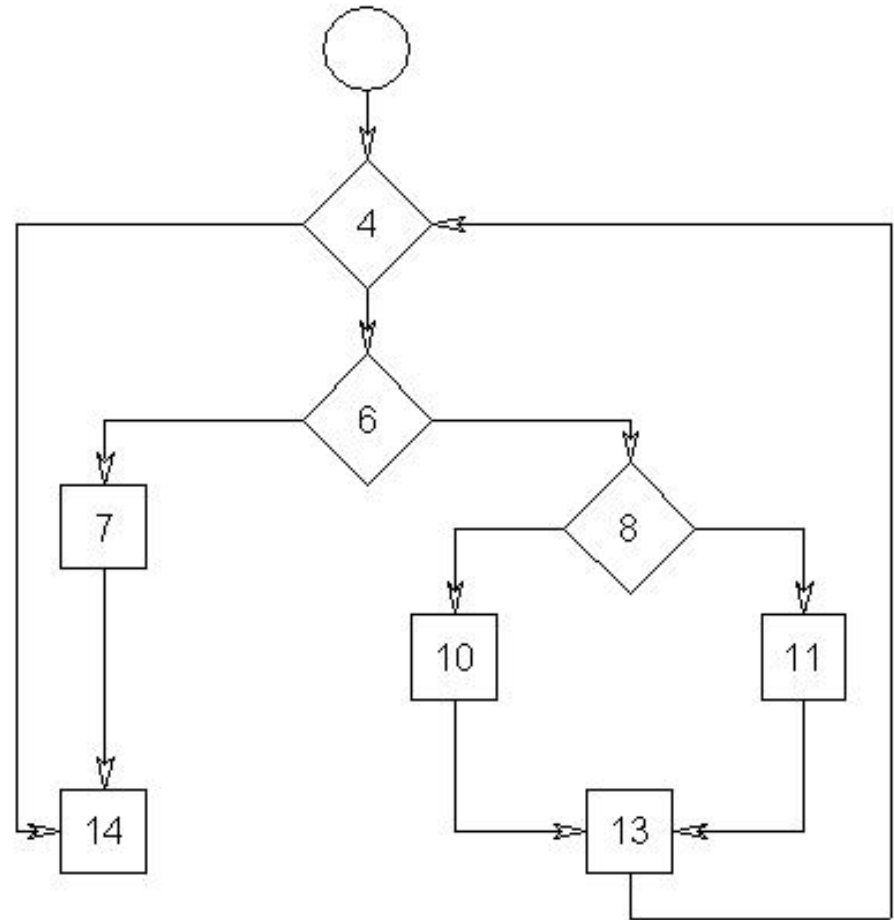
❖ 例2：循环

4.2.3 基本路径测试—举例



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. }
```

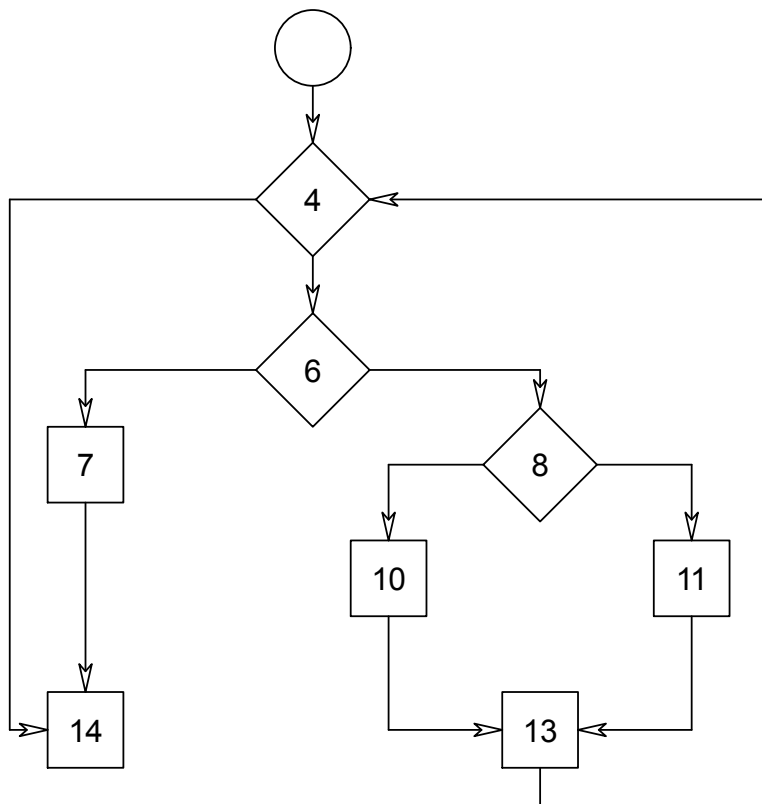


程序流程图

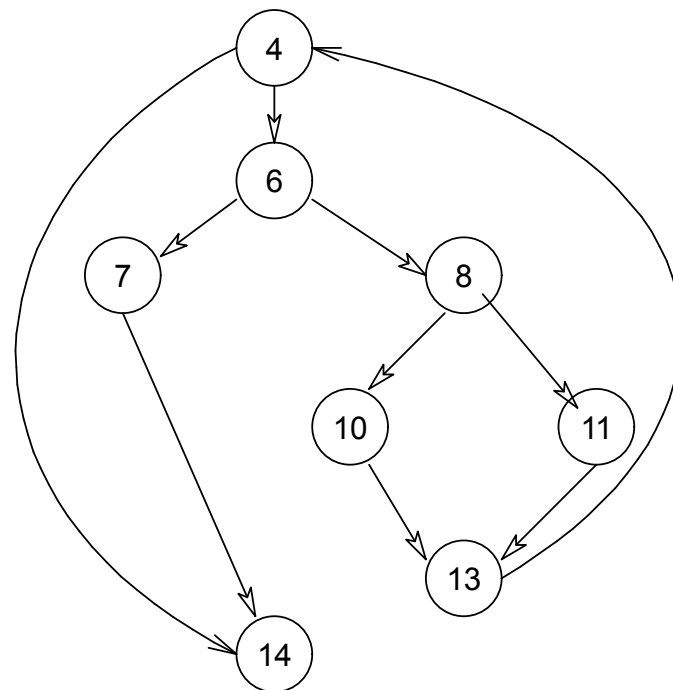
4.2.3 基本路径测试—举例



❖ 画出其程序流程图和对应的控制流图如下



程序流程图



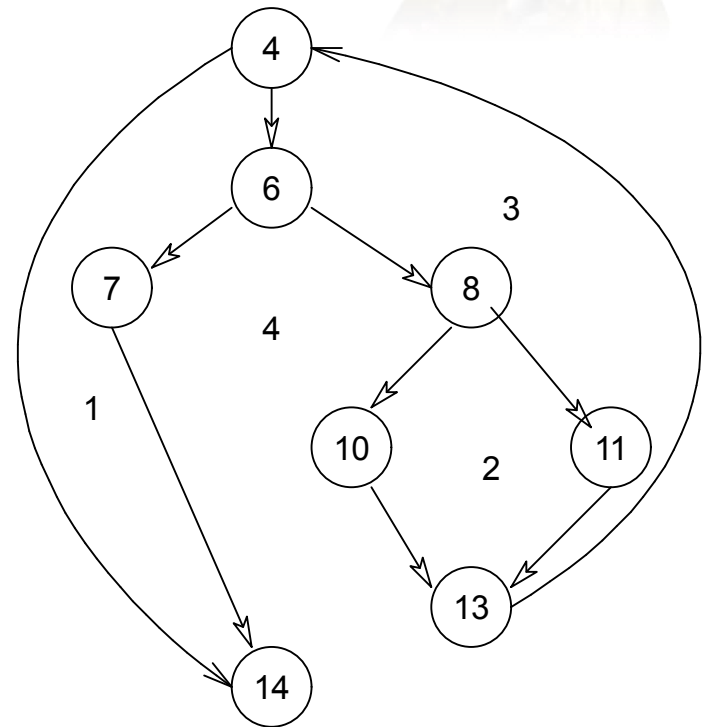
控制流图

4.2.3基本路径测试 – 举例



对应上面图中的圈复杂度，计算如下：

- ✓ 流图中有四个区域；
- ✓ $V(G)=10\text{条边}-8\text{结点}+2=4$;
- ✓ $V(G)=3\text{个判定结点}+1=4$ 。



4.2.3基本路径测试 – 举例



❖ 第三步：导出测试用例

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。)

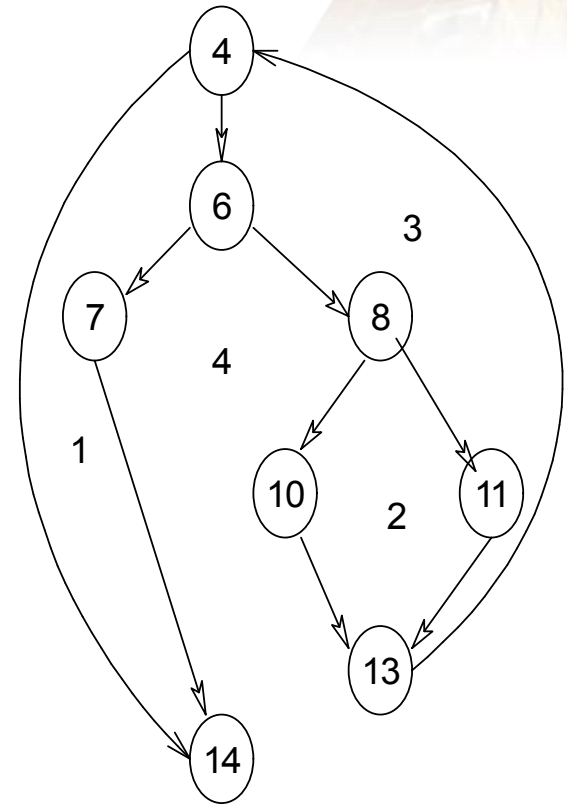
✓ 路径1：4-14

✓ 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。



4.2.3基本路径测试 – 举例



❖ 第四步：准备测试用例

为了确保基本路径集中的每一条路径的执行，根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到，满足上面例子基本路径集的测试用例是：

4.2.3基本路径测试 – 举例



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

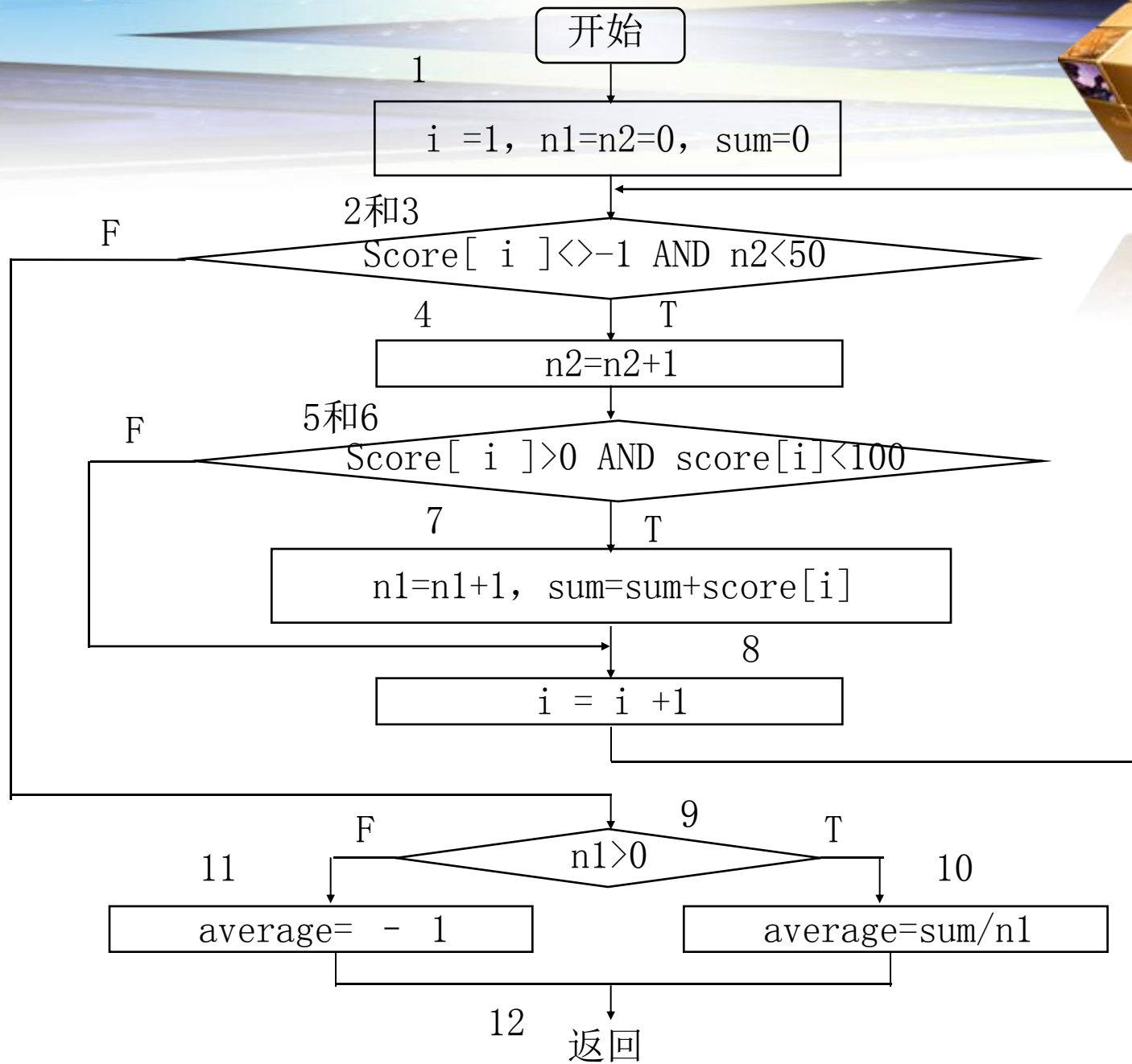
预期结果: x=20

```
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. }
```

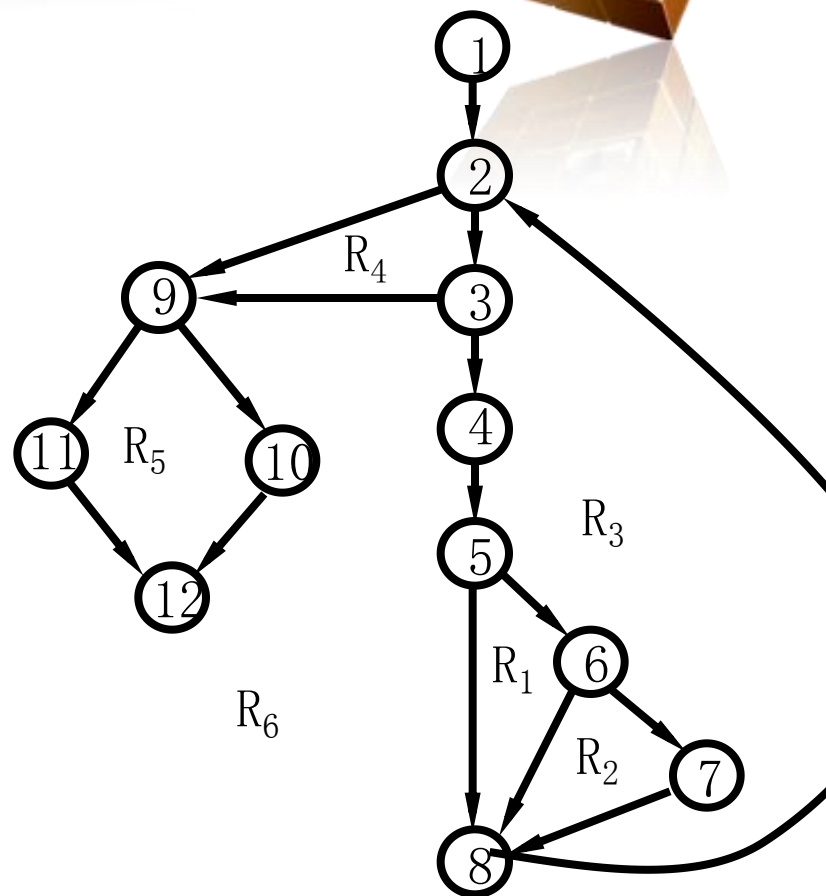
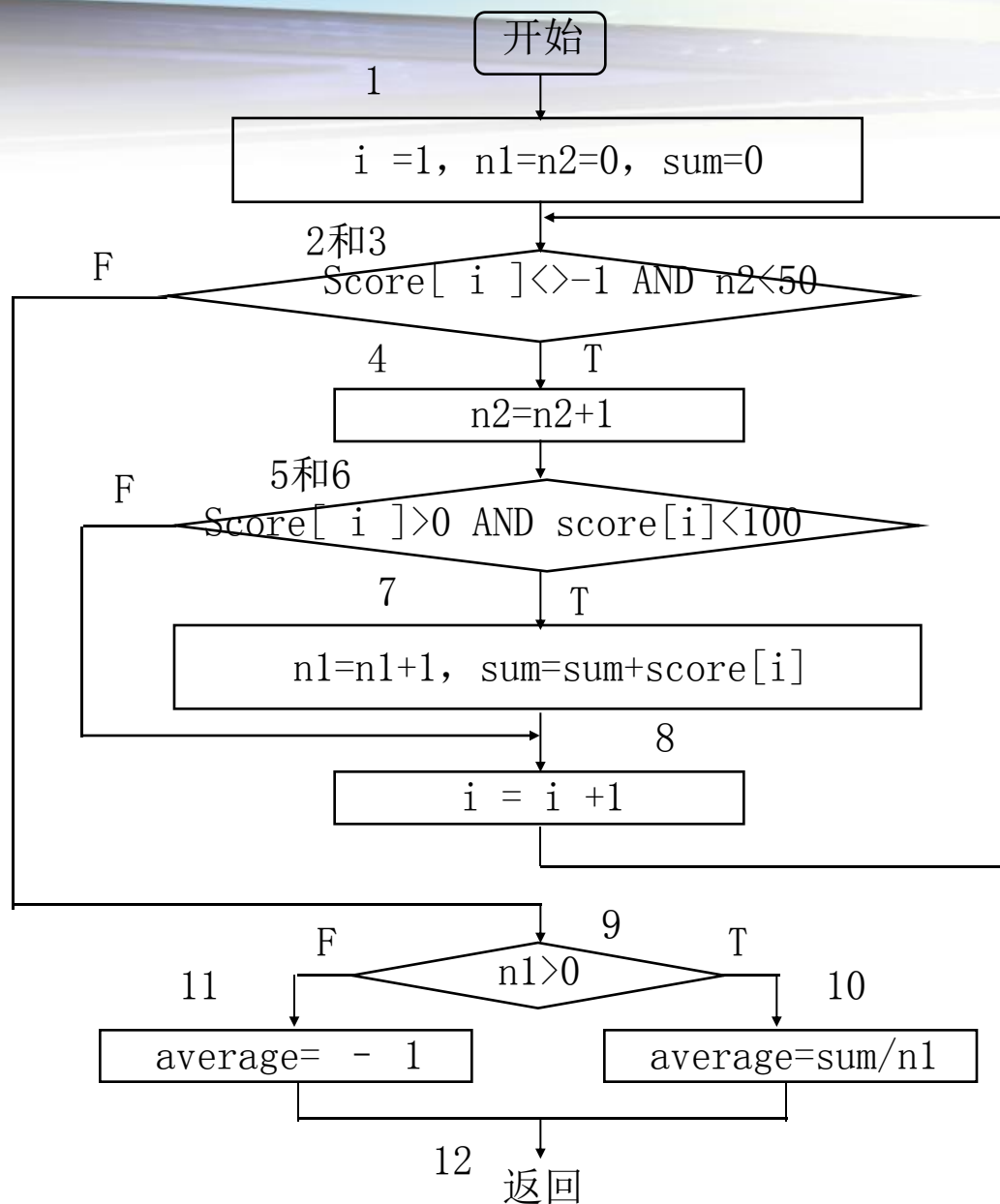
4.2.3基本路径测试 – 举例



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。



步骤1：导出过程的流程图。



4.2.3基本路径测试 – 举例

步骤2: 确定环形复杂性度量 $V(G)$:

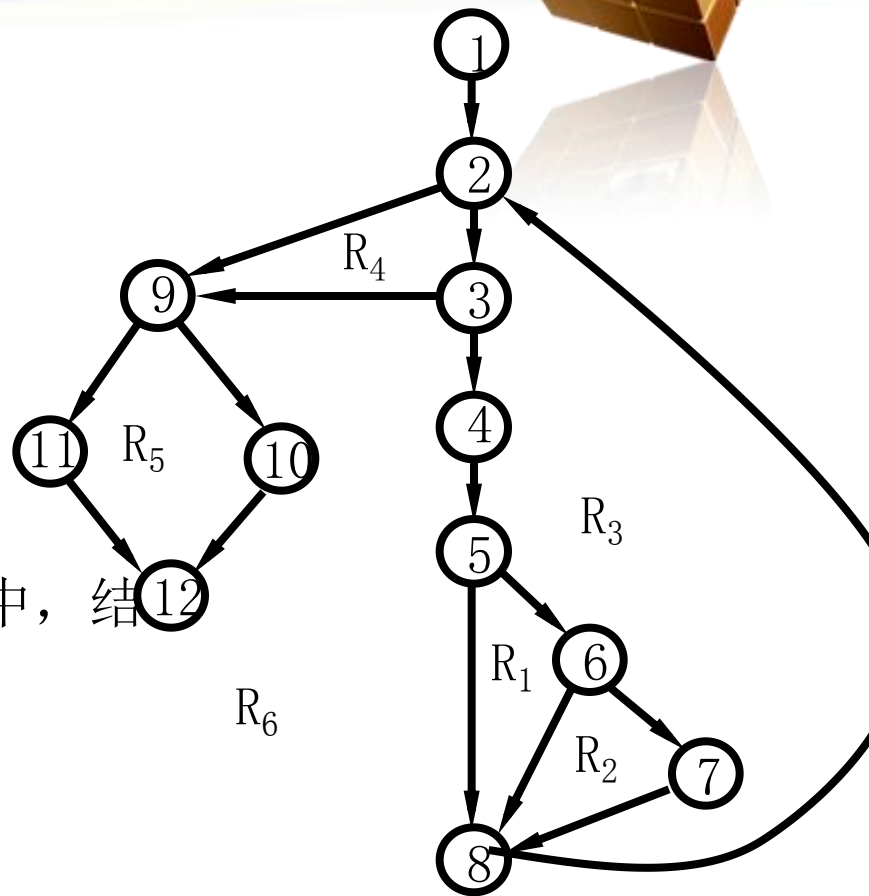
1) $V(G) = 6$ (个区域)

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

其中 E 为流图中的边数, N 为结点数;

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

其中 P 为谓词结点的个数。在流图中, 结点2、3、5、6、9是谓词结点。



4.2.3基本路径测试 – 举例



步骤3：确定基本路径集合（即独立路径集合）。于是可确定6条独立的路径：

路径1：1-2-9-10-12

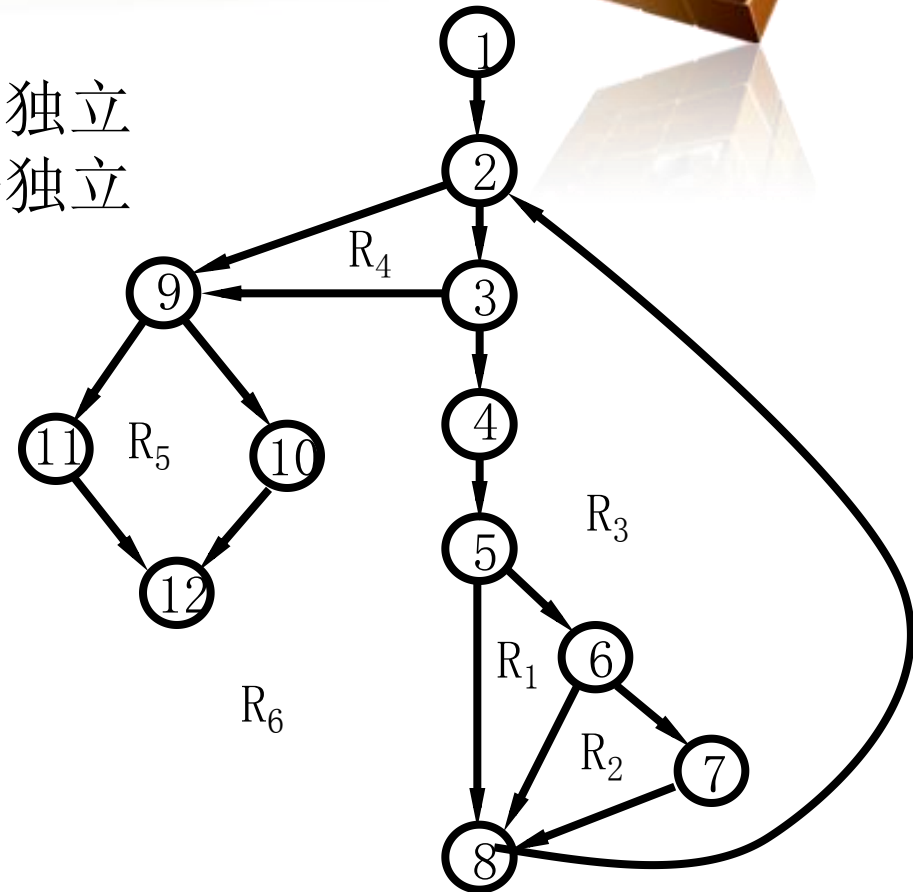
路径2：1-2-9-11-12

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...



4.2.3基本路径测试 – 举例

步骤4：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。

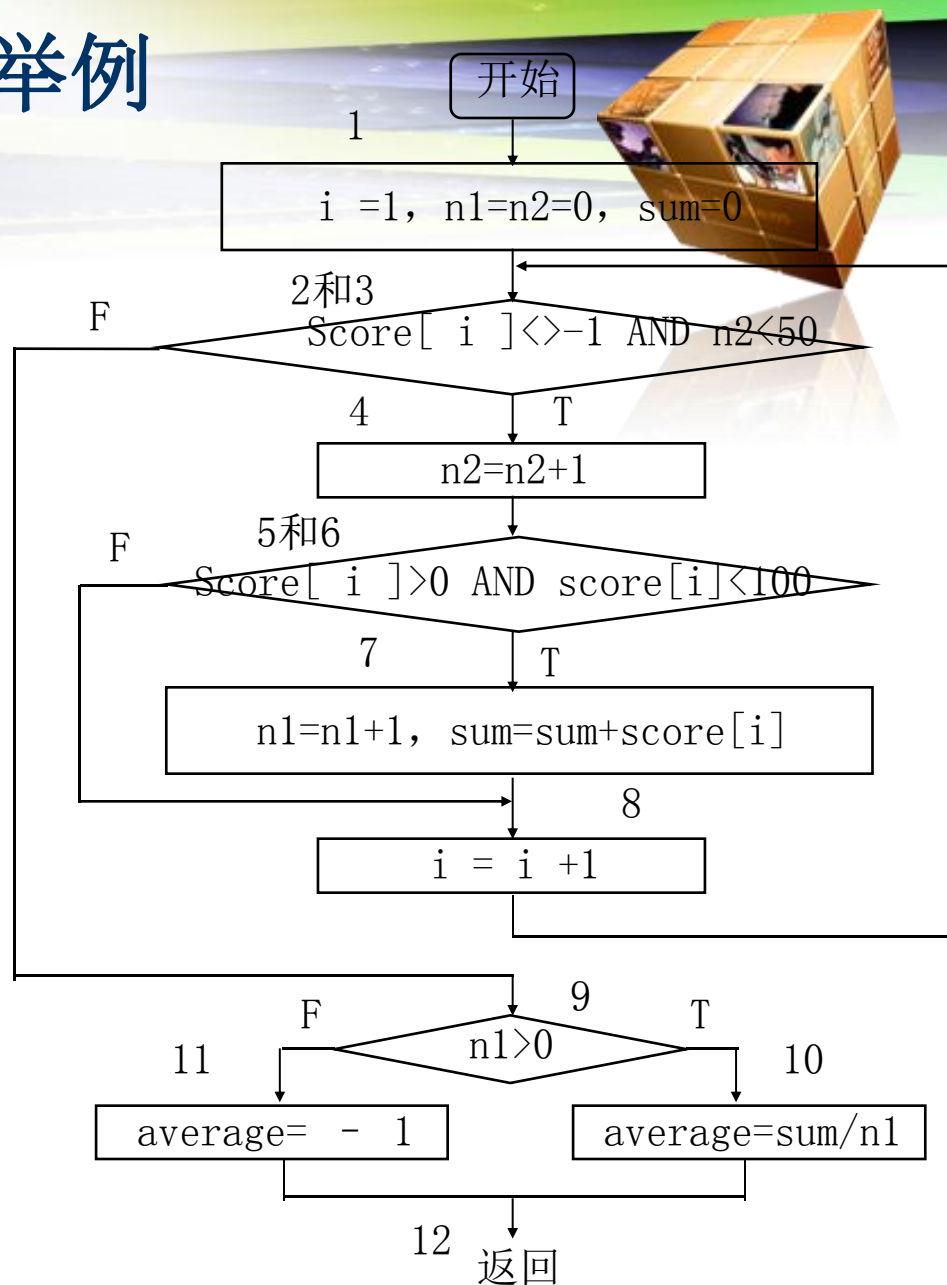
1) 路径1(1-2-9-10-12)的测试用例：

score[k]=有效分数值，当 $k < i$ ；

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

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。

(i之前的数据为前置数据)



4.2.3基本路径测试 – 举例

2) 路径2(1-2-9-11-12)的测试用例:

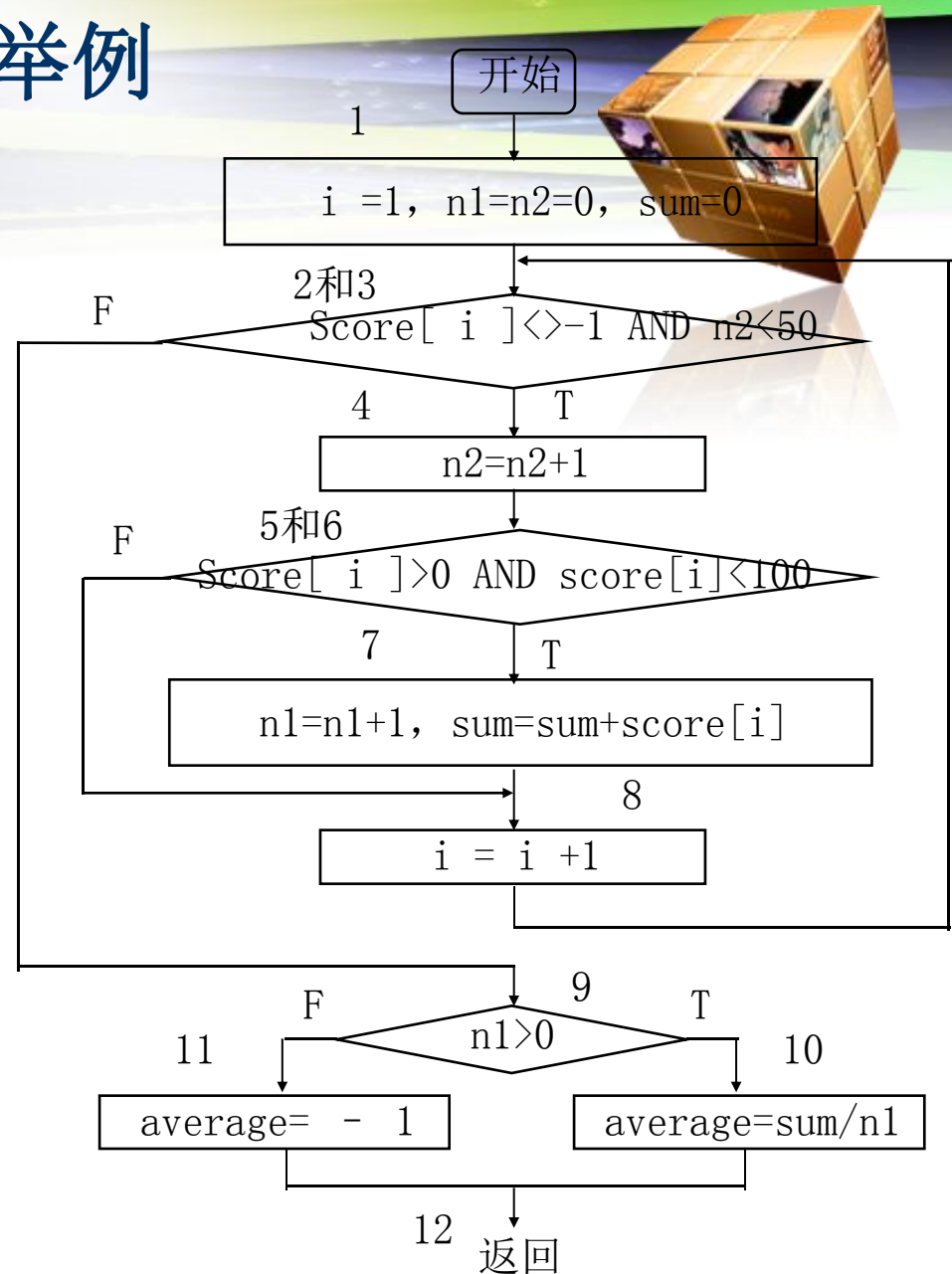
$\text{score}[1] = -1$;

期望的结果: $\text{average} = -1$, 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。



4.2.3基本路径测试 – 举例



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, $k < i$;

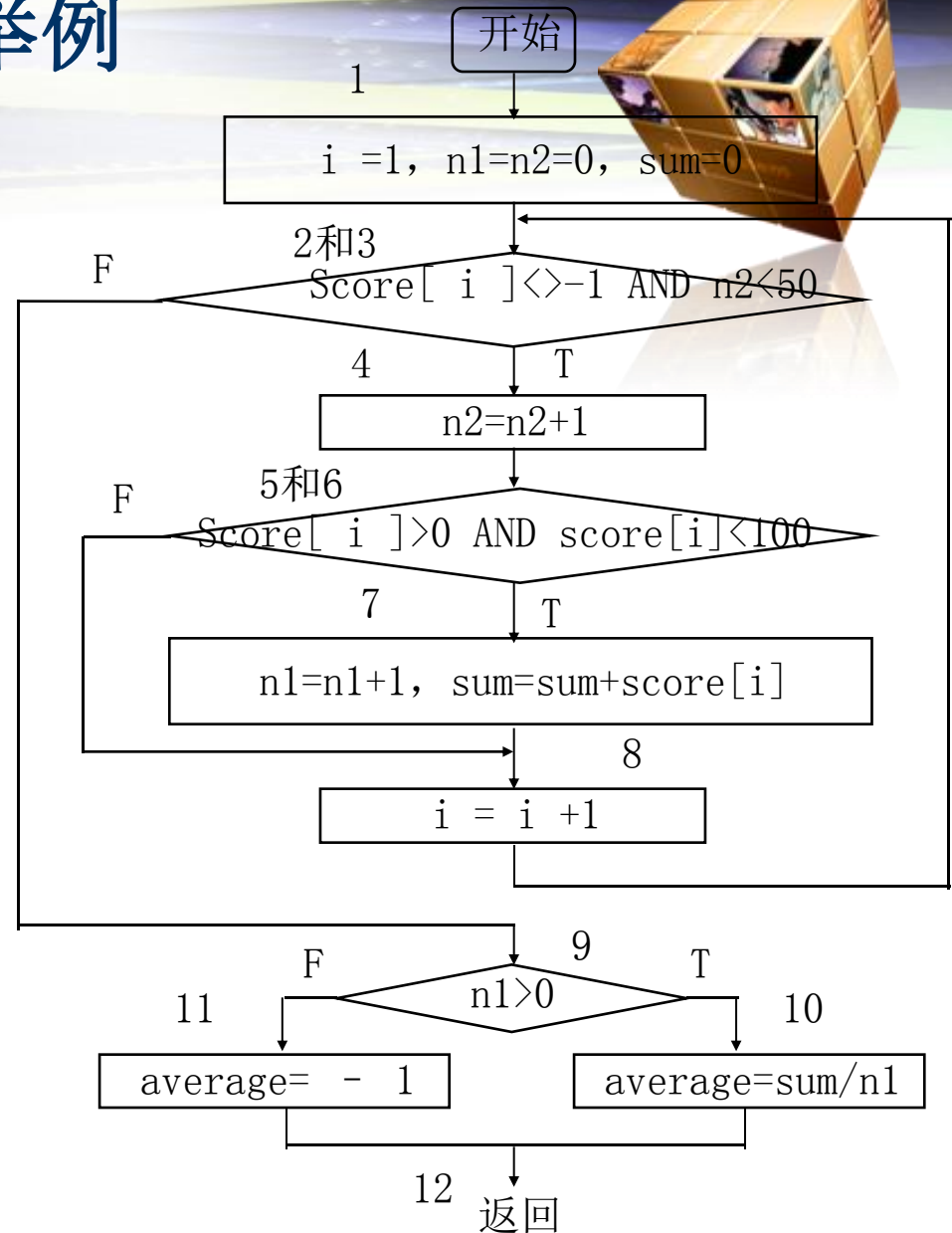
期望结果: 根据输入的有效分数
算出正确的分数个数n1、总分sum
和平均分average。

5) 路径5的测试用例:

score[i]=有效分数, 当 $i < 50$;

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

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。

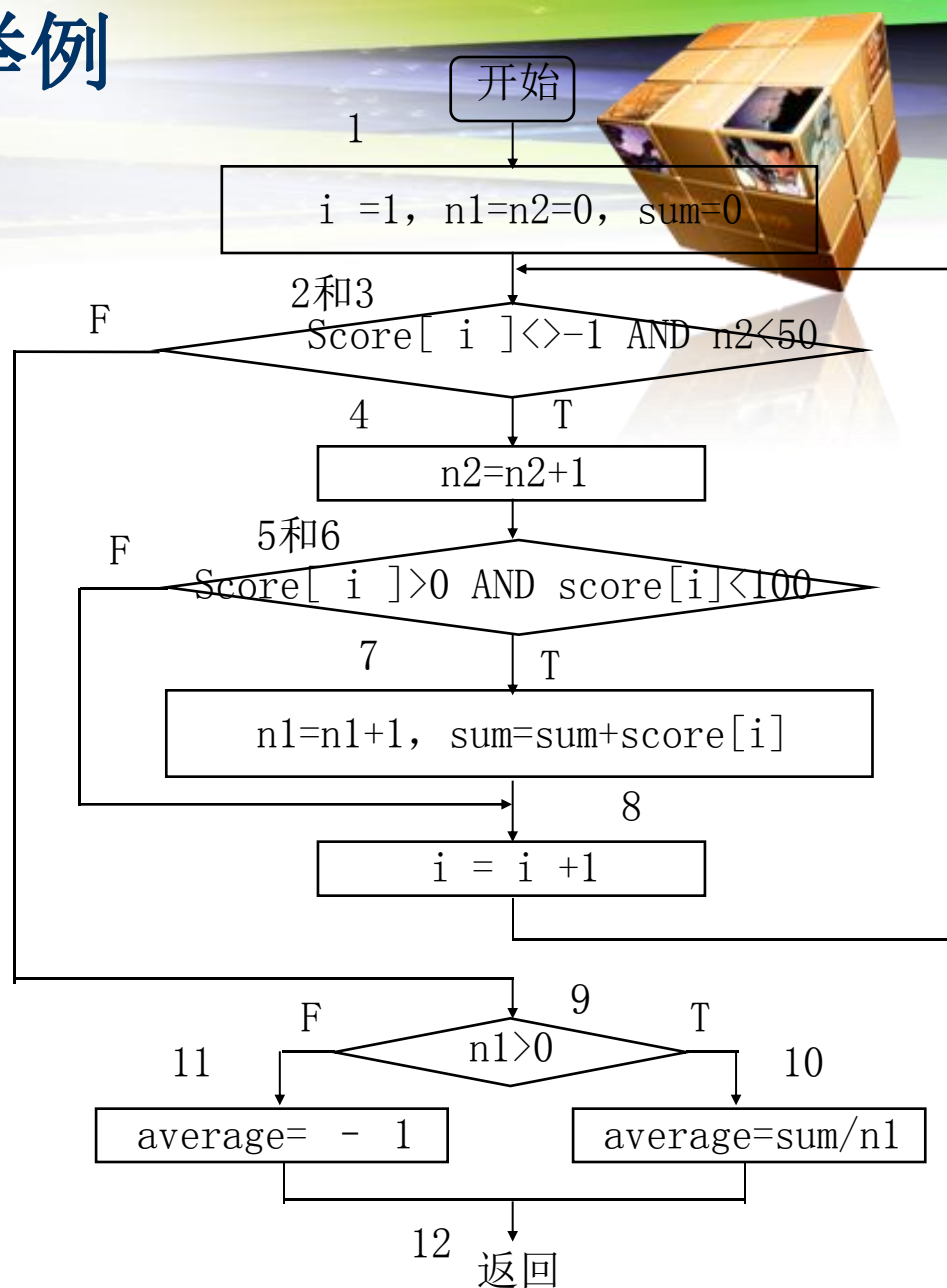


4.2.3基本路径测试 – 举例

6) 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。



基本路径测试作业



- ❖ 选择排序的基本路径测试用例生成
- ❖ 三角形判别的基本路径测试用例生成

4.3 程序切片测试



■ 思想

- 对程序进行裁剪，使得所得到的程序代码仍能反映原程序的部分特征
- 程序切片专门针对这类问题，它按切片准则来裁剪程序，使人们能把注意力集中在相关的程序代码上

4. 3程序切片测试



■ 定义

给定一个程序 P 和 P 中的一个变量集合 V ，变量集合 V 在语句 n 上的一个片，记作 $S(V,n)$ ，是 P 中对 V 中的变量值作出贡献的所有语句集合

■ 贡献：控制依赖、数据依赖

■ 如果不管是否包含语句片段， V 中各变量的值都保持不变，那么排除该语句片段

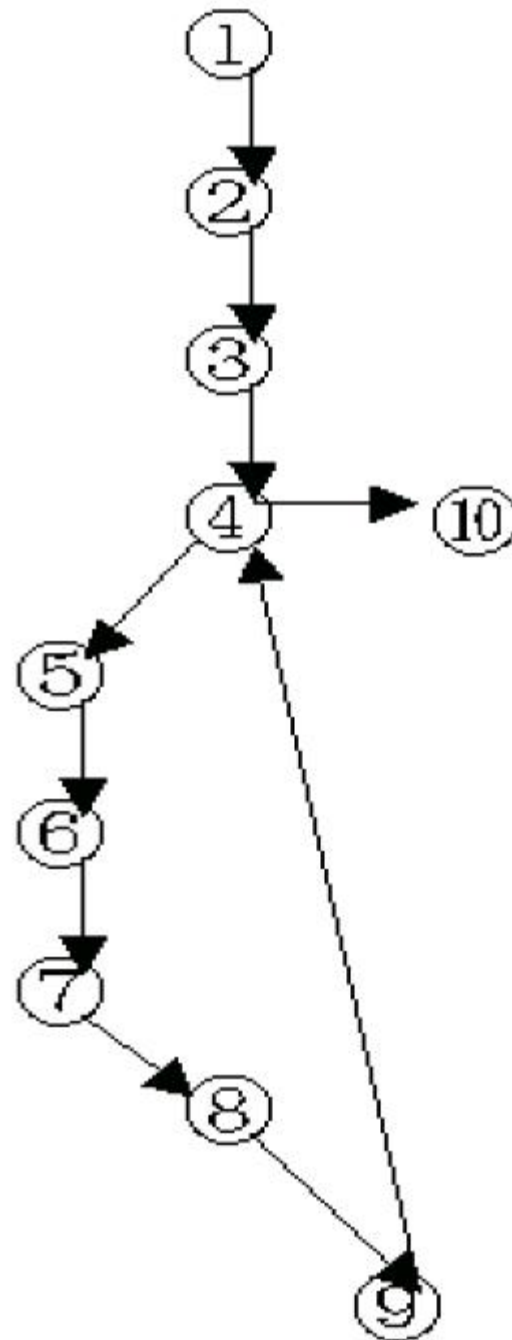
4.3 程序切片测试—程序依赖



- 控制依赖：假设C是包含结点n1和n2的控制流图，n1是一个判断结点，如果下面两个条件同时成立，则称结点n2控制依赖于n1：
 - 至少存在一条从n1到程序出口的路径，该路径包含n2；
 - 至少存在一条从n1到程序出口的路径，该路径不包含n2。
- 数据依赖：假设D是包含结点n1和n2的数据依赖图，如果下面两个条件同时成立，则称结点n2数据依赖于n1：
 - 变量v在n1处定义、在n2处引用；
 - 存在一条从n1到n2的非空路径，不包含任何重定义v的结点。

4. 3程序切片测试

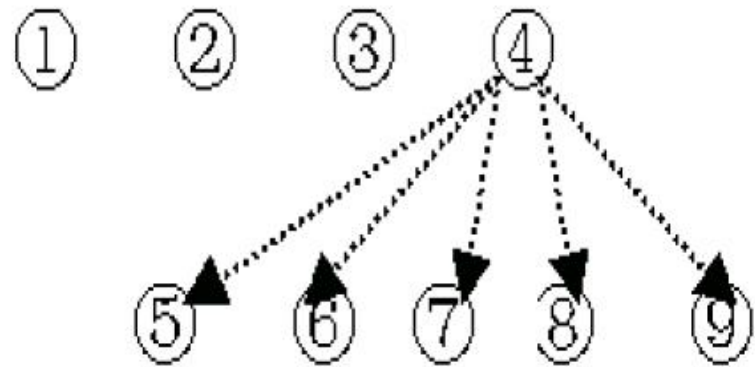
```
1  input(k);  
2  t:=0;  
3  f:=1;  
4  while (k > 0) do  
5  {  t:=t + k;  
6    f:=f * k;  
7    k:=k - 1;  
8    output(t);  
9    output(f); }
```



4. 3程序切片测试—控制依赖关系



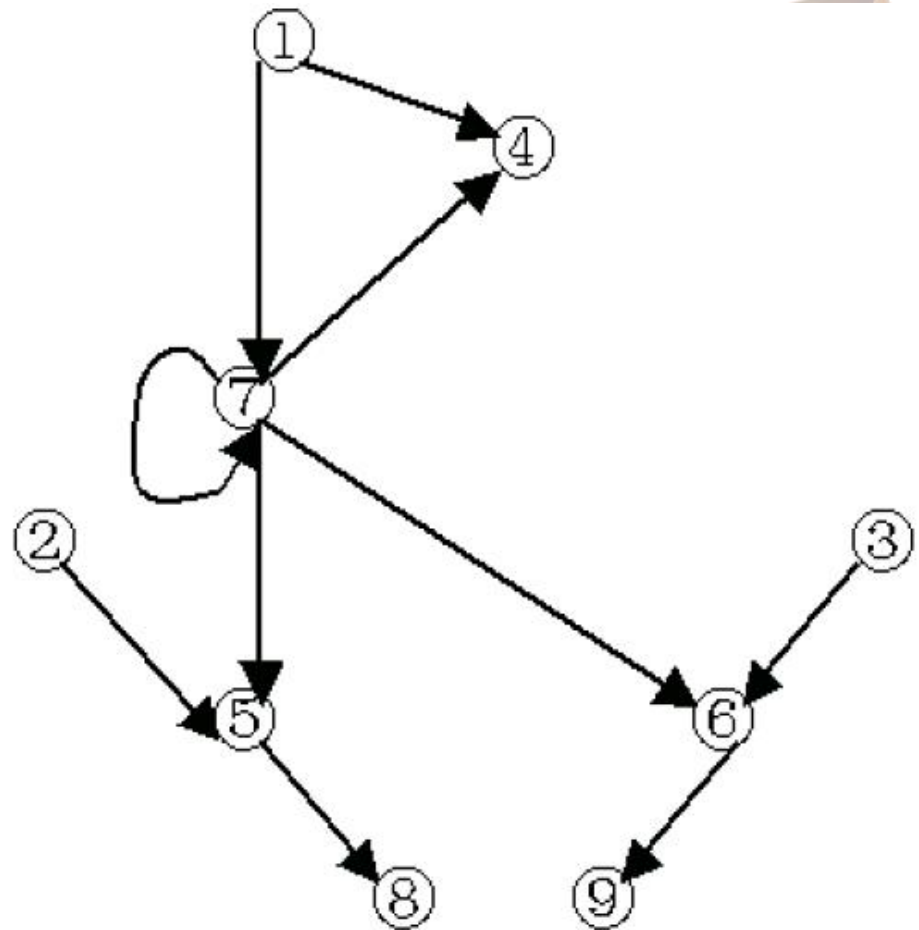
```
1  input(k);  
2  t:=0;  
3  f:=1;  
4  while (k > 0) do  
5  {  t:=t + k;  
6    f:=f * k;  
7    k:=k - 1;  
8    output(t);  
9    output(f); }
```



4.3 程序切片测试—数据依赖关系



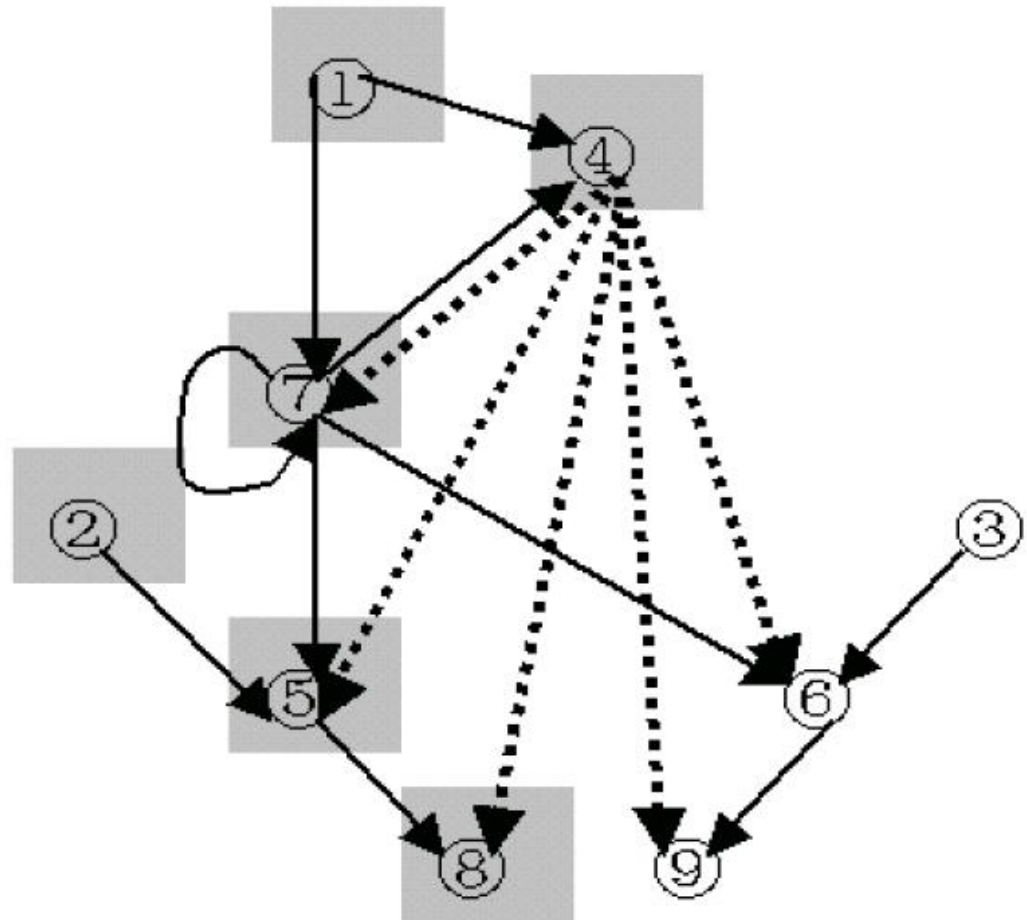
```
1  input(k);  
2  t:=0;  
3  f:=1;  
4  while (k > 0) do  
5  {  t:=t + k;  
6    f:=f * k;  
7    k:=k - 1;  
8    output(t);  
9    output(f); }
```



4.3 程序切片测试—程序依赖关系



```
1  input(k);  
2  t:=0;  
3  f:=1;  
4  while (k > 0) do  
5  {  t:=t + k;  
6    f:=f * k;  
7    k:=k - 1;  
8    output(t);  
9    output(f); }
```



4.3 程序切片测试



■ 根据程序依赖图计算静态切片

- 在程序依赖图中求出所关心结点的逆向传递闭包就可以得出它所数据依赖、控制依赖的结点
- 例如，上页中带阴影的结点即是程序中结点8的逆向传递闭包
- 由以上计算所得结点对应的程序语句所组成的程序段即为所求静态切片)

4.3 程序切片测试—程序切片



```
1  input(k);
2  t:=0;
3  f:=1;
4  while (k > 0) do
5  {  t:=t + k;
6     f:=f * k;
7     k:=k - 1;
8     output(t);
9     output(f); }
```

(a) 程序例子

```
1  input(k);
2  t:=0;
4  while (k > 0) do
5  {  t:=t + k;
7     k:=k - 1;
8     output(t);
                                }
```

(b) 程序例子关于切片准则
 $C=\langle 8, t \rangle$ 的静态切片

4.3 程序切片测试



■ 程序切片作用

- 程序调试
- 软件测试
- 软件维护
- 软件复用

4.4 变异测试—基本概念



❖ 变异算子：在符合语法规则前提下，变异算子定义了从原有程序生成差别极小程序（即变异体）的转换规则。例如：将“+”操作符变异为“-”操作符。

选择被测程序 p 中的条件表达式 $a + b > c$ 执行该变异算子，将得到条件表达式 $a - b > c$ ，并生成变异体 p' 。

4.4 变异测试--变异算子种类



序号	变异算子	描述
1	AAR	用一数组引用替代另一数组引用
2	ABS	插入绝对值符号
3	ACR	用数组引用替代常量
4	AOR	算术运算符替代
5	ASR	用数组引用替代变量
6	CAR	用常量替代数组引用
7	CNR	数组名替代
8	CRP	常量替代
9	CSR	用常量替代变量
10	DER	DO 语句修改
11	DSA	DATA 语句修改
12	GLR	GOTO 标签替代
13	LCR	逻辑运算符替代
14	ROR	关系运算符替代
15	RSR	RETURN 语句替代
16	SAN	语句分析
17	SAR	用变量替代数组引用
18	SCR	用变量替代常量
19	SDL	语句删除
20	SRC	源常量替代
21	SVR	变量替代
22	UOI	插入一元操作符

ABS	$\{(e, \text{abs}(e)), (e, -\text{abs}(e))\}$
AOR	$\{(x, y) x, y \in \{+, -, *, /, \%\}\}$
LCR	$\{(x, y) x, y \in \{\&\&, \}\}$
ROR	$\{(x, y) x, y \in \{>, >=, <=, ==, !=\}\}$
.....	

软件测试大赛--变异算子



表 2 面向过程程序的变异算子

变异算子	描述
运算符变异	对关系运算符 “<”、“<=”、“>”、“>=” 进行替换，如将 “<” 替换为 “<=”
	对自增运算符 “++” 或自减运算符 “--” 进行替换，如将 “++” 替换为 “--”
	对与数值运算的二元算术运算符进行替换，如将 “+” 替换为 “-”
	将程序中的条件运算符替换为相反运算符，如将 “==” 替换为 “!=”
数值变异	对程序中整数类型、浮点数类型的变量取相反数，如将 “i” 替换为 “-i”
方法返回值变异	删除程序中返回值类型为void的方法
	对程序中方法的返回值进行修改，如将 “true” 修改为 “false”

软件测试大赛--变异算子



表 3 面向对象程序的变异算子

变异算子	描述
继承变异	增加或删除子类中的重写变量
	增加、修改或重命名子类中的重写方法
	删除子类中的关键字super，如将“return a*super.b”修改为“return a*b”
多态变异	将变量实例化为子类型
	将变量声明、形参类型改为父类型，如将“Integer i”修改为“Object i”
	赋值时将使用变量替换为其它可用类型
重载变异	修改重载方法的内容，或删除重载方法
	修改方法参数的顺序或数量

4.4 变异测试--变异基本概念



- ❖ 一阶变异体：在原有程序 p 上执行单一变异算子并形成变异体 p' ，则称 p' 为 p 的一阶变异体。
- ❖ 高阶变异体：在原有程序 p 上依次执行多次变异算子并形成变异体 p' ，则称 p' 为 p 的高阶变异体。若在 p 上依次执行 k 次变异算子并形成变异体 p' ，则称 p' 为 p 的 k 阶变异体。
- ❖ 可杀除变异体：若存在测试用例 t ，在变异体 p' 和原有程序 p 上的执行结果不一致，则称该变异体 p' 相对于测试用例集 T 是可杀除变异体。

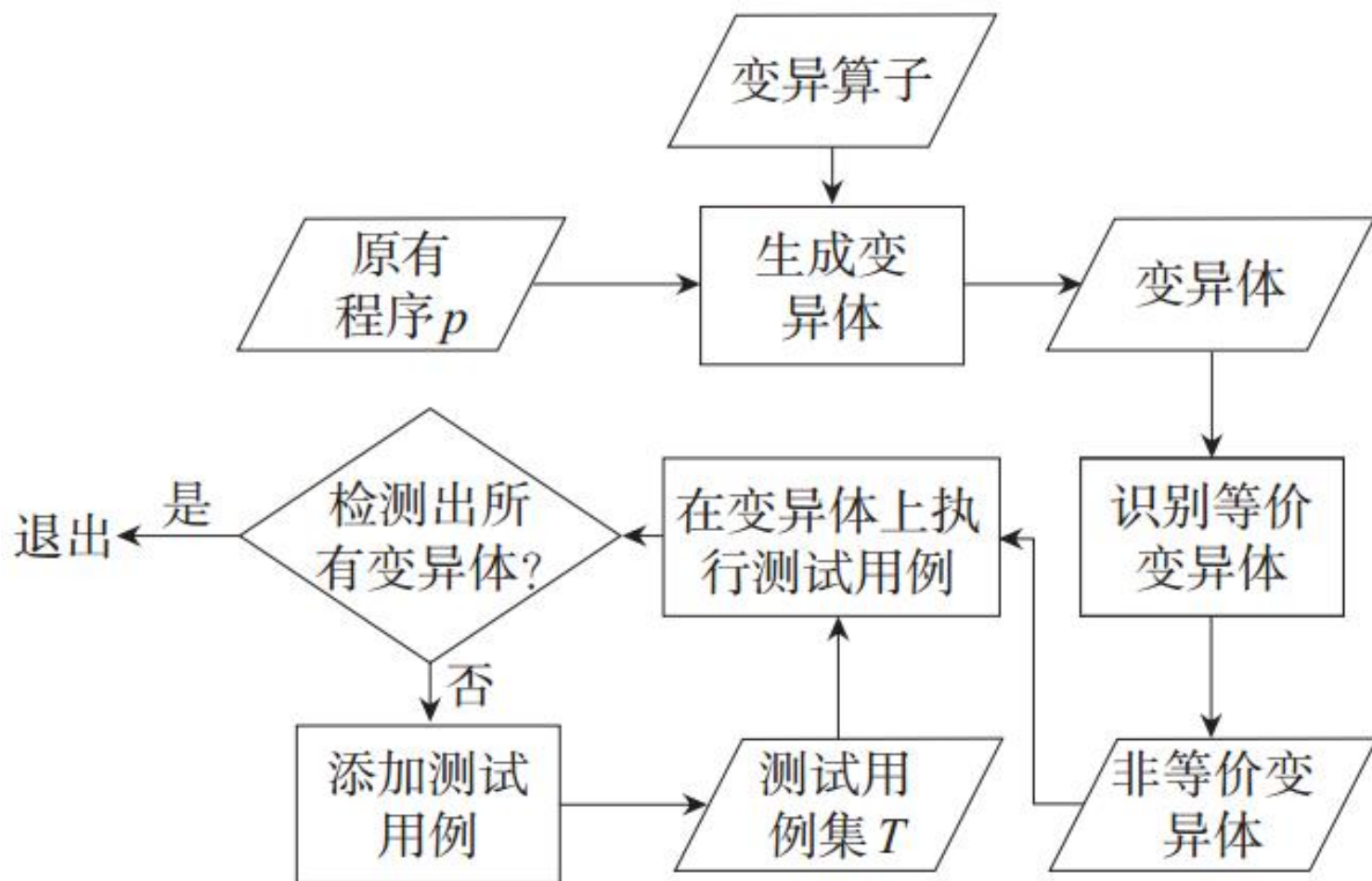
4.4 变异测试--变异基本概念



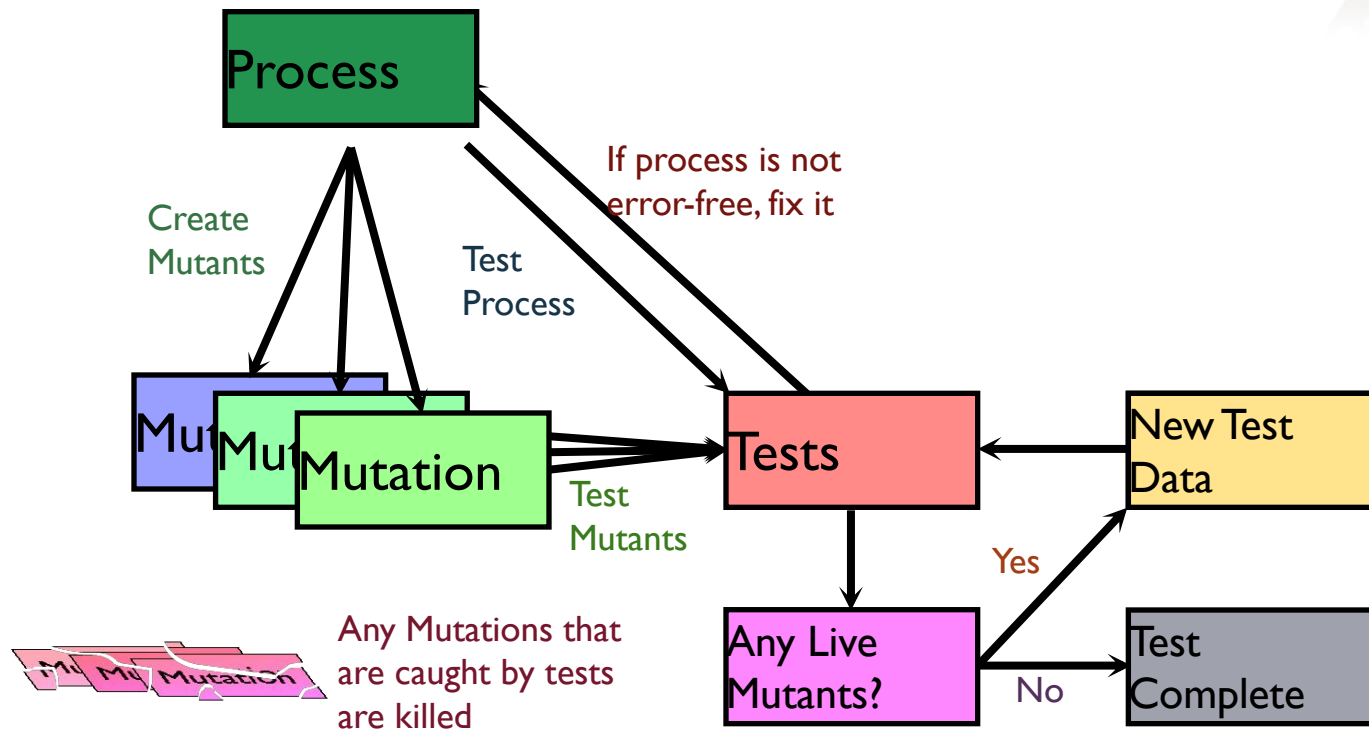
- ❖ 可存活变异体)：若不存在任何测试用例 t ，在变异体 p' 和原有程序 p 上的执行结果不一致，则称该变异体 p' 相对于测试用例集 T 是可存活变异体。
- ❖ 等价变异体：若变异体 p' 与原有程序 p 在语法上存在差异，但在语义上与 p 保持一致，则称 p' 是 p 的等价变异体。

程序 p	变异体 p'
<pre>for (int i=0; i<10; i++){ sum += a[i]; }</pre>	<pre>for (int i=0; i!=10; i++){ sum += a[i]; }</pre>

4.4 变异测试--变异测试流程



4.4 变异测试--变异流程



4.4 变异测试--程序变异举例



原始程序

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

6 mutants

**Each represents a
separate program**

With Embedded Mutants

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    Δ 1 minVal = B;
    if (B < A)
    Δ 2 if (B > A)
    Δ 3 if (B < minVal)
    {
        minVal = B;
    Δ 4 Bomb ();
    Δ 5 minVal = A;
    Δ 6 minVal = failOnZero (B);
    }
    return (minVal);
} // end Min
```

*Replace one
variable with
another*

Changes operator

*Immediate runtime
failure ... if reached*

*Immediate runtime
failure if B==0 else
does nothing*

总结：



“白盒”法全面了解程序内部逻辑结构、对所有逻辑路径进行测试。“白盒”法是穷举路径测试。在使用这一方案时，测试者必须检查程序的内部结构，从检查程序的逻辑着手，得出测试数据。贯穿程序的独立路径数是天文数字。但即使每条路径都测试了仍然可能有错误。

- 第一、穷举路径测试决不能查出程序违反了设计规范，即程序本身是个错误的程序；
- 第二、穷举路径测试不可能查出程序中因遗漏路径而出错。
- 第三、穷举路径测试可能发现不了一些与数据相关的错误。

编程建议




❖ **必须使用断言** 确认最后提交的测试用例全部正确，Webide或者客户端能运行得分 使用 `setup()`和`teardown()`方法进行环境初始化 使用 `timeout`限制测试用例运行时间，避免测试用例出现无响应情况 **Evosuite**生成的捕获异常的自动测试用例，可能导致变异无法跑通， 学生自己注意

编程建议

对double类型进行判断



```
@Test
public void test() {
    double a = 1.0;
    assertEquals(1.0,a);
}
```


 The method assertEquals(double, double) from the type Assert is deprecated

2 quick fixes available:

[@ Add @SuppressWarnings 'deprecation' to 'test\(\)'](#)

[Configure problem severity](#)

```
@Test
public void test() {
    double a = 1.0;
    assertEquals(1.0,a,0.00001);
}
```

 **void org.junit.Assert.assertEquals(double expected, double actual, double delta)**

Asserts that two doubles are equal to within a positive delta. If they are not, an [AssertionError](#) is thrown. If the expected value is infinity then the delta value is ignored. NaNs are considered equal: assertEquals(Double.NaN, Double.NaN, *) passes

Parameters:

expected expected value

actual the value to check against expected

delta the maximum delta between expected and actual for which both numbers are still considered equal.

Press 'F2' for focus

编程建议

对控制台输出进行测试



```
import static org.junit.Assert.*;

public class BinPackageTest {

    String sep;
    BinPackage bp;
    PrintStream console = null;
    ByteArrayInputStream in = null;
    ByteArrayOutputStream out = null;
    InputStream input = null;

    @Before
    public void setUp() throws Exception {
        bp = new BinPackage();
        out = new ByteArrayOutputStream();
        input = System.in;
        console = System.out;
        System.setOut(new PrintStream(out));
        sep = System.getProperty("line.separator");
    }

    @After
    public void tearDown() throws Exception {
        System.setIn(input);
        out.close();
        System.setOut(console);
    }

    @Test(timeout=4000)
    public void test1() {
        in = new ByteArrayInputStream(("2" + sep + "1 2" + sep + "3" + sep + "2 1 1" + sep + "2 2 2" + sep + "2 3 2").getBytes());
        System.setIn(in);
        bp.entrance();
        String ans = out.toString();
        assertEquals("false" + sep + "true" + sep + "2 2" + sep + "false" + sep, ans);
    }
}
```



编程建议

对异常报错进行测试

```
import static org.junit.Assert.*;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class AccountTest {
    private static final double DOUBLE_DELTA = 1e-15;

    @Rule
    public ExpectedException thrown= ExpectedException.none();

    @Test(timeout=4000)
    public void test1() {
        Bank bank = new Bank();
        Account checkingAccount = new Account(Account.CHECKING);
        Customer bill = new Customer("Bill").openAccount(checkingAccount);
        bank.addCustomer(bill);
        thrown.expect(IllegalArgumentException.class);
        thrown.expectMessage("amount must be greater than zero");
        checkingAccount.withdraw(-100.0);
    }
}
```


实验设想



- ❖ 以白盒测试实验中的三角形例子，设想如下实验方案
 - 需要完成语句、判定、条件、判定/条件，组合条件，路径，基本路径覆盖分析
 - 要求将各测试用例（按一定结构）存储于MySQL数据库
 - 每种类型一个测试脚本，根据测试用例存储结构不同实现自动测试。
 - 最后对每种类型进行测试覆盖分析



Q & A

