

软件工程

3.3 白盒测试

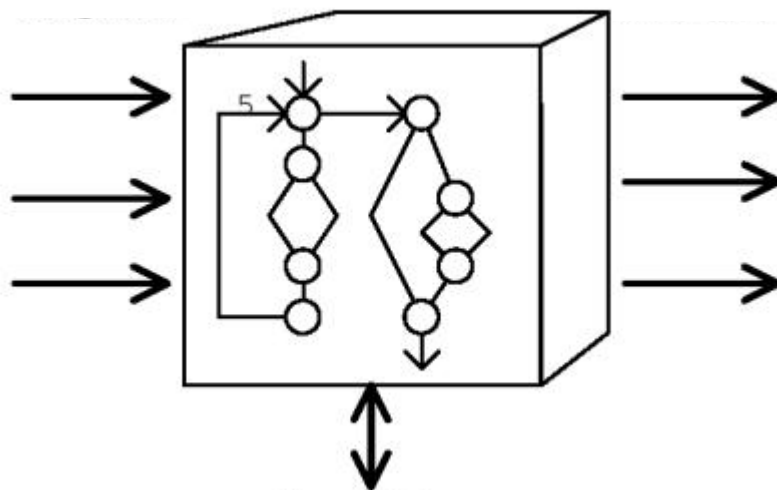
刘铭

2015年9月18日

1 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”)

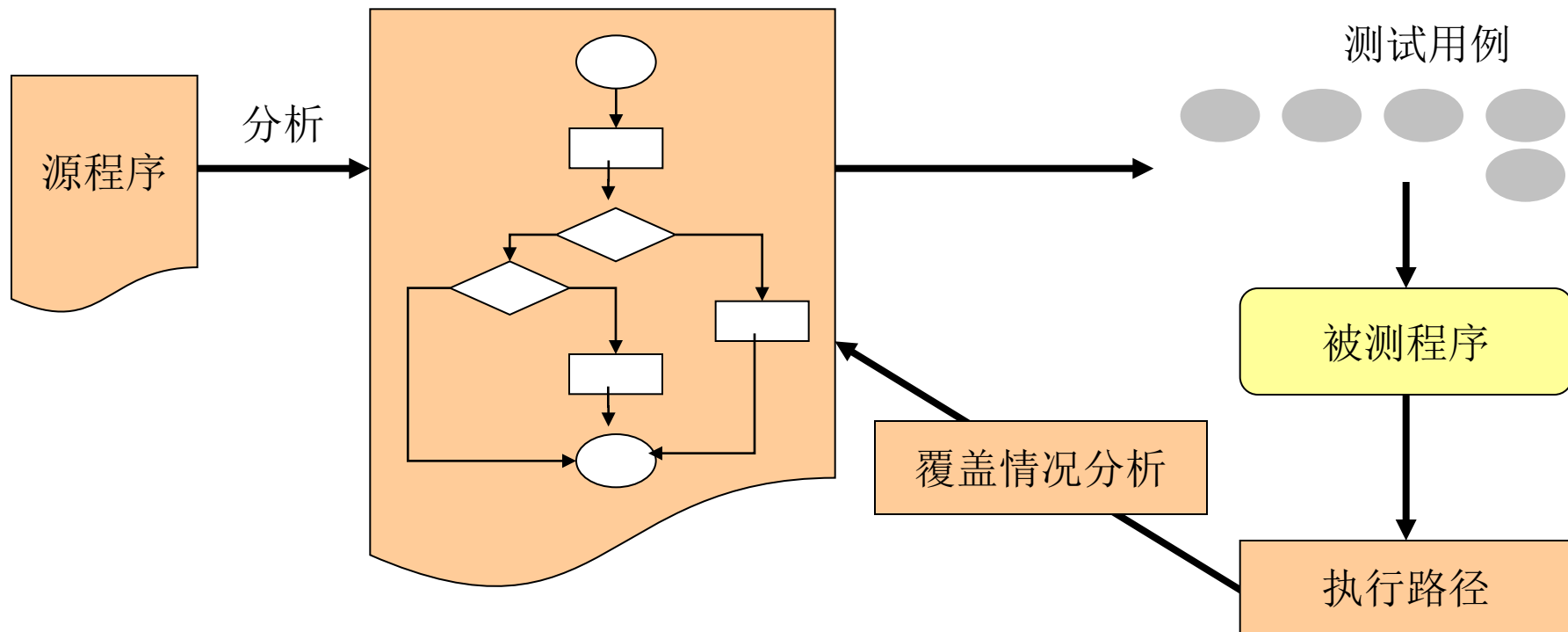
- 把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。



2 白盒测试的目的

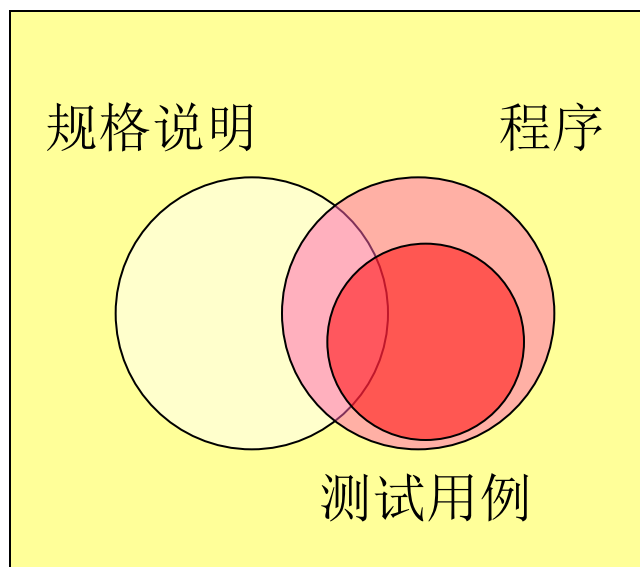
- 白盒测试主要对程序模块进行如下的检查：
 - 对模块的每一个独立的执行路径至少测试一次；
 - 对所有的逻辑判定的每一个分支(真与假)都至少测试一次；
 - 在循环的边界和运行界限内执行循环体；
 - 测试内部数据结构的有效性；
- 错误隐藏在角落里，聚集在边界处

白盒测试的过程



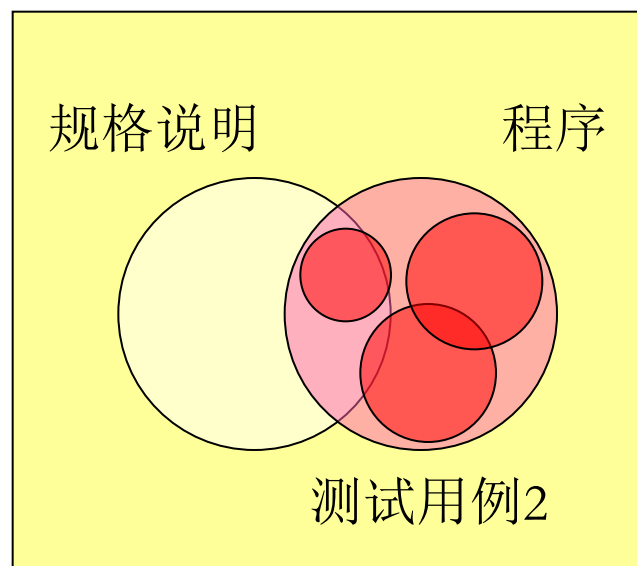
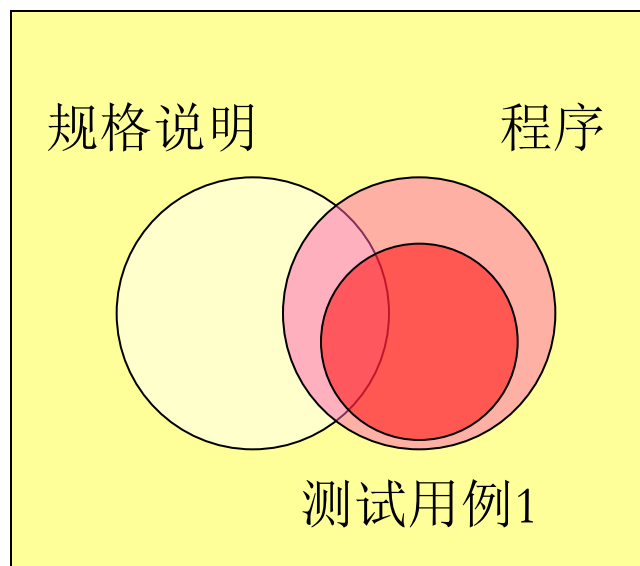
白盒测试用例中的输入数据从程序结构导出，
但期望输出务必从需求规格中导出。

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

3 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。

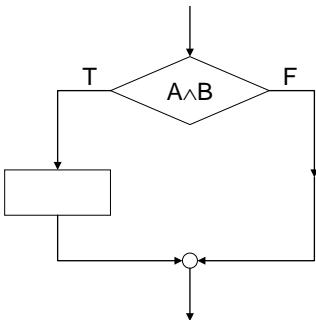
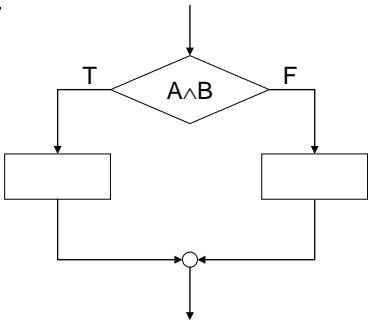
测试覆盖标准

- 为了衡量测试的覆盖程度，需要建立一些标准，目前常用的一些覆盖标准从低到高分别是：
 - **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
 - **结构覆盖：**
 - 基本路径测试
 - 控制结构测试
 - 条件测试
 - 数据流测试

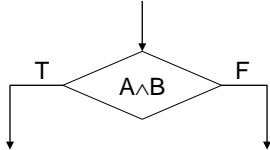
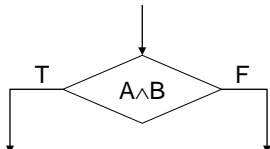
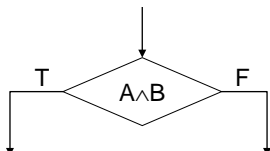
五种覆盖标准的对比

发现错误的能力	<div>弱</div> <div>↓</div> <div>强</div>	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

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		$A \wedge B = T$
判定覆盖		$A \wedge B = T$ $A \wedge B = F$

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		$A=T, A=F$ $B=T, B=F$
判定/条件覆盖		$A \wedge B=T, A \wedge B=F$ $A=T, A=F$ $B=T, B=F$
条件组合覆盖		$A=T \wedge B=T$ $A=T \wedge B=F$ $A=F \wedge B=T$ $A=F \wedge B=F$

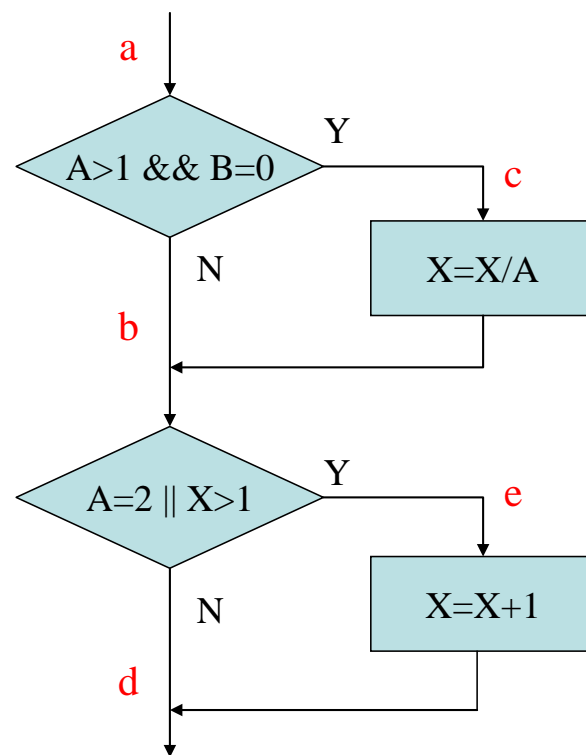
4 白盒测试技术

- 基本路径测试
- 控制结构测试

(1) 路径测试

- **路径测试：**设计足够多的测试用例，覆盖被测试对象中的所有可能路径。
- 对于左例，下面的测试用例则可对程序进行全部的路径覆盖。

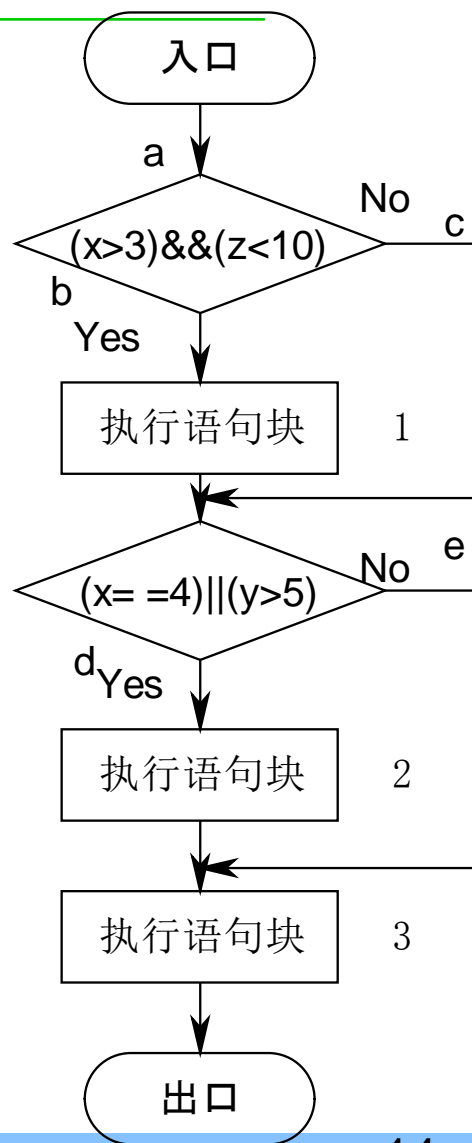
测试用例	通过路径
A=2、B=0、X=3	ace
A=1、B=0、X=1	abd
A=2、B=1、X=1	abe
A=3、B=0、X=1	acd



路径测试

- 对于左例，下面的测试用例则可对程序进行全部的路径覆盖。

测试用例	通过路径	覆盖条件
x=4、y=6、z=5	abd	T1、T2、T3、T4
x=4、y=5、z=15	acd	T1、-T2、T3、-T4
x=2、y=5、z=15	ace	-T1、-T2、-T3、T4
x=5、y=5、z=5	abe	T1、T2、-T3、-T4



基本路径测试

- 在实际中，即使一个不太复杂的程序，其路径都是一个庞大的数字，要在测试中覆盖所有的路径是不现实的。为了解决这一难题，只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行一次。
- 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行复杂性，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一个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。

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。

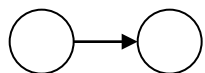
基本路径测试

- 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. 确定线性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

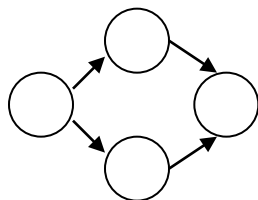
控制流图的符号

■ 流图只有2种图形符号

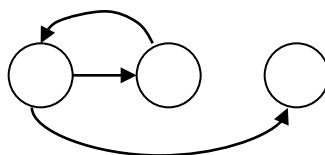
- 图中的每一个圆称为流图的结点，代表一条或多条语句。
- 流图中的箭头称为边或连接，代表控制流。



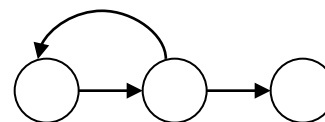
顺序结构



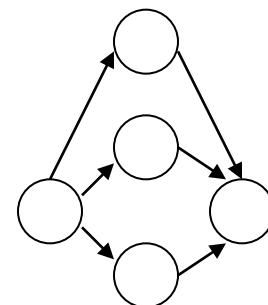
if 结构



while 结构

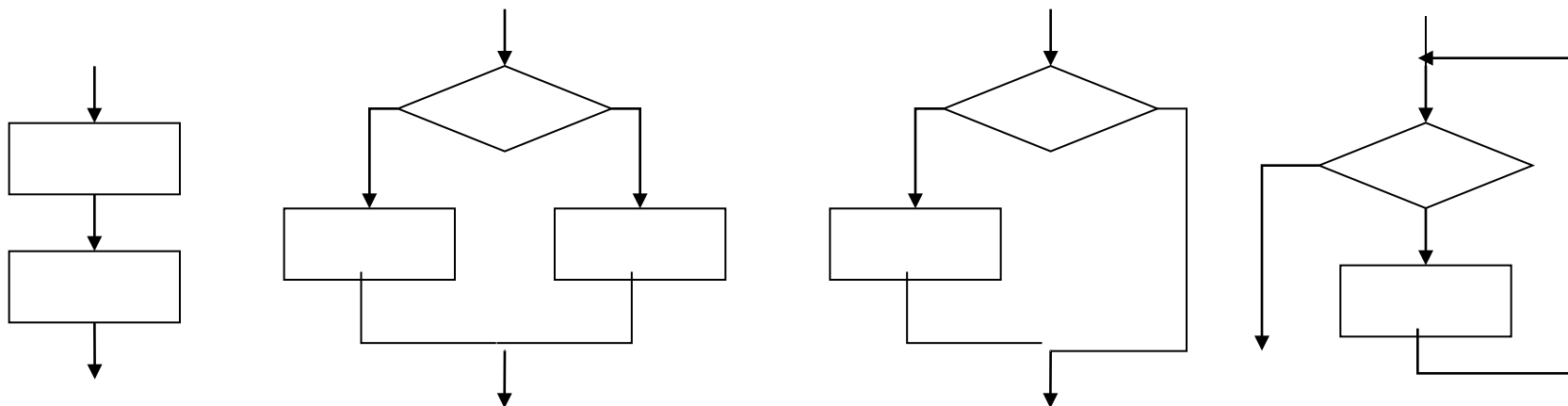


until 结构

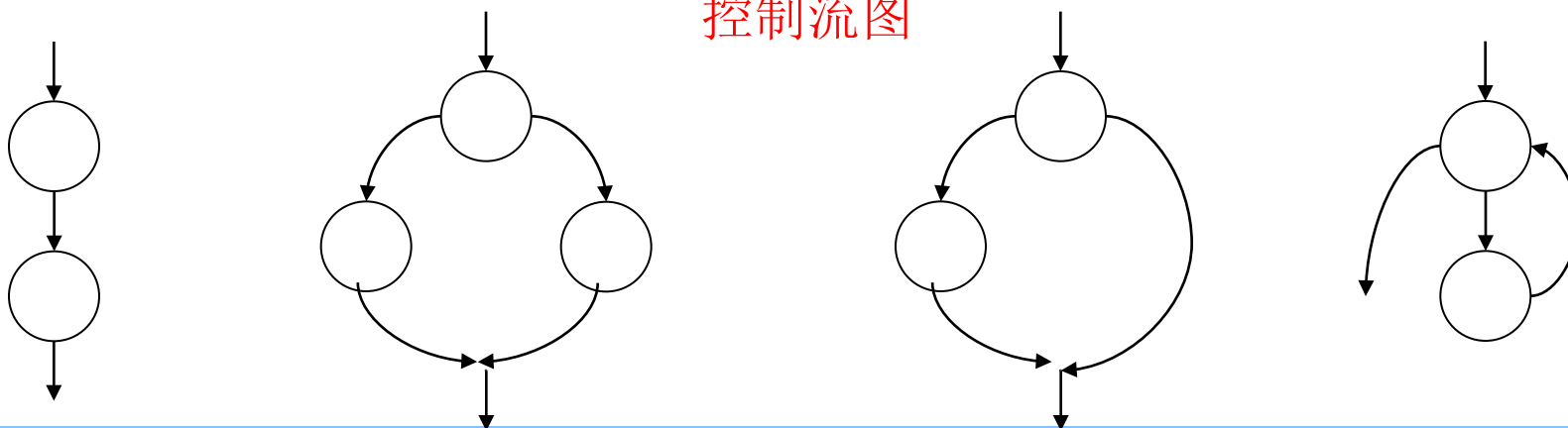


case 结构

程序流程图→控制流图



程序流程图



控制流图

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:

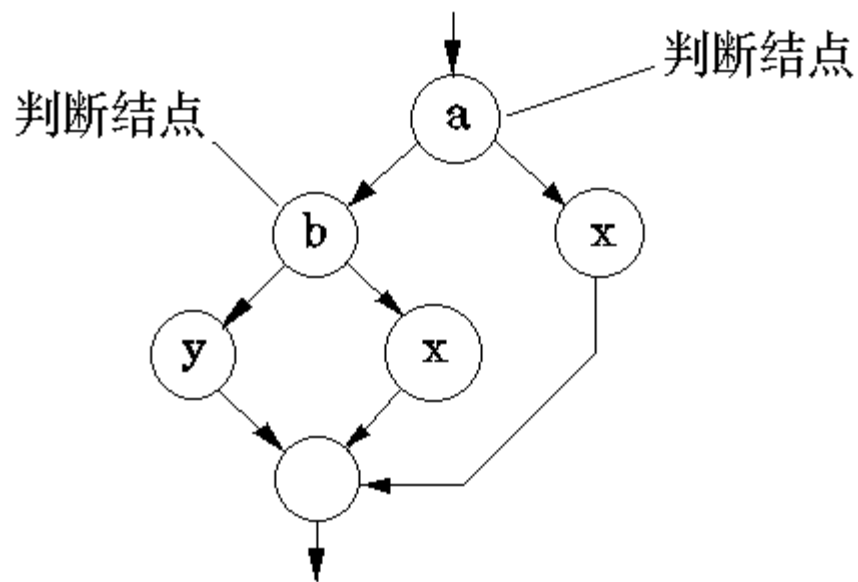
1 if a or b

2 x

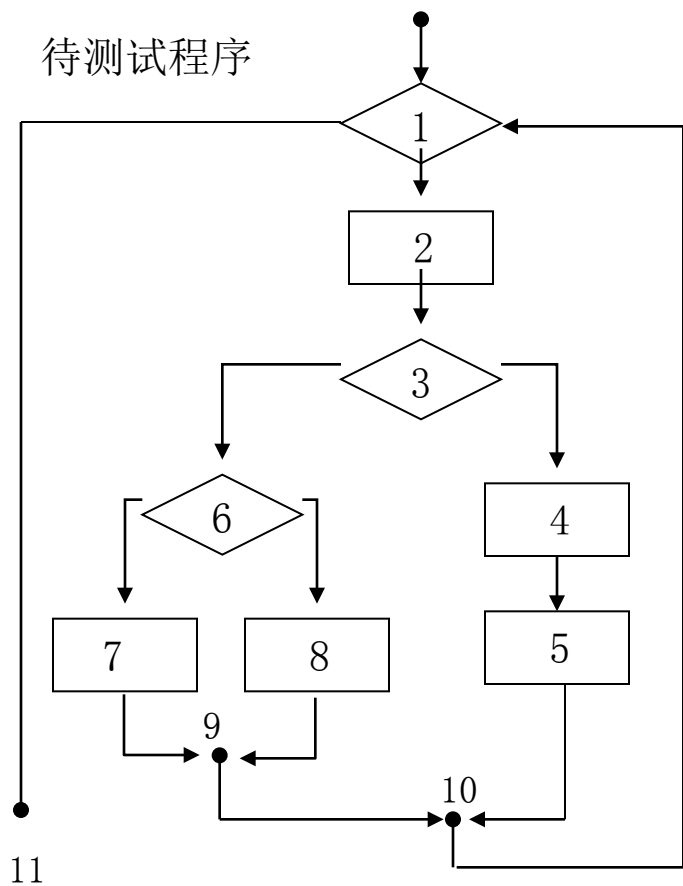
3 else

4 y

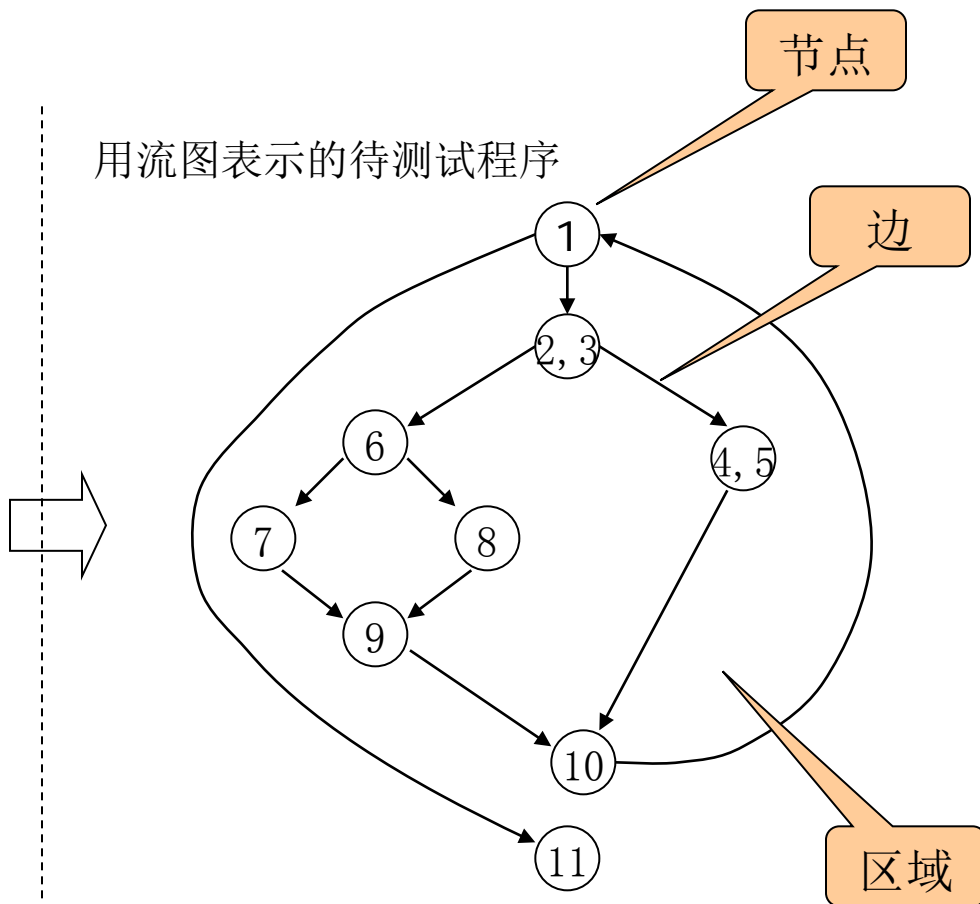
对应的逻辑为:



控制流图



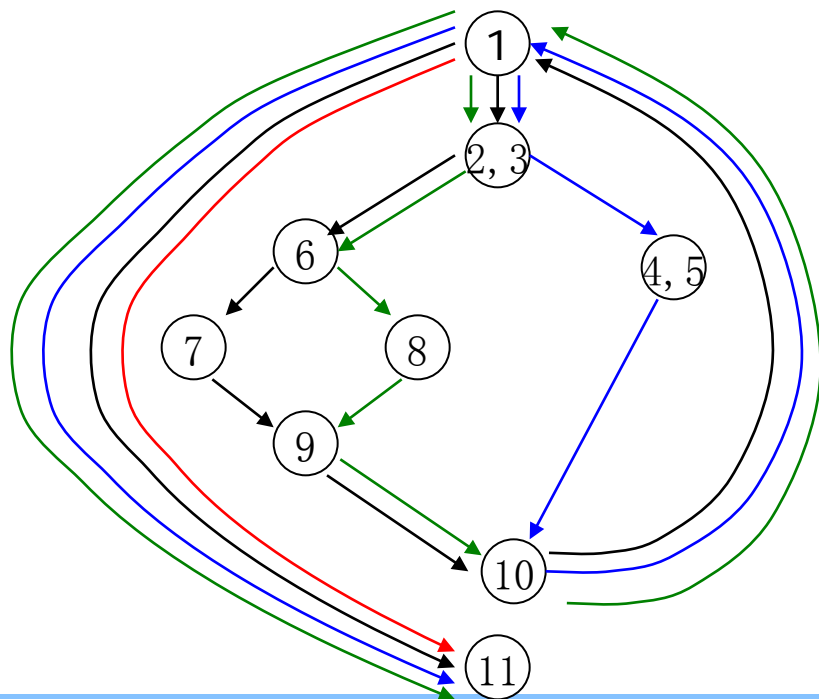
用流图表示的待测试程序



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

独立路径

- 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

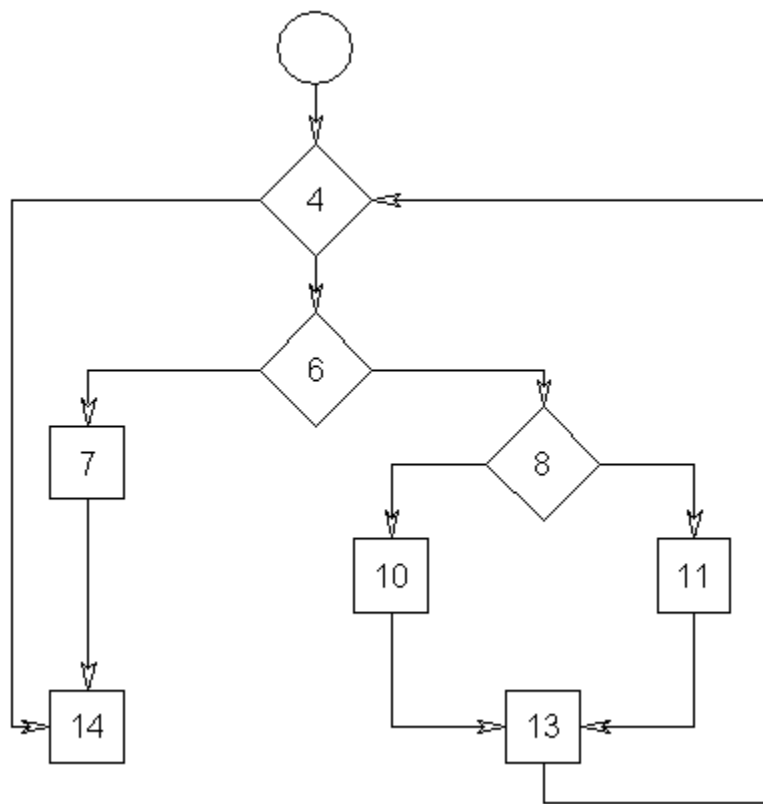
对以上路径的遍历，就是至少一次地执行了程序中的所有语句。

第一步：画出控制流图

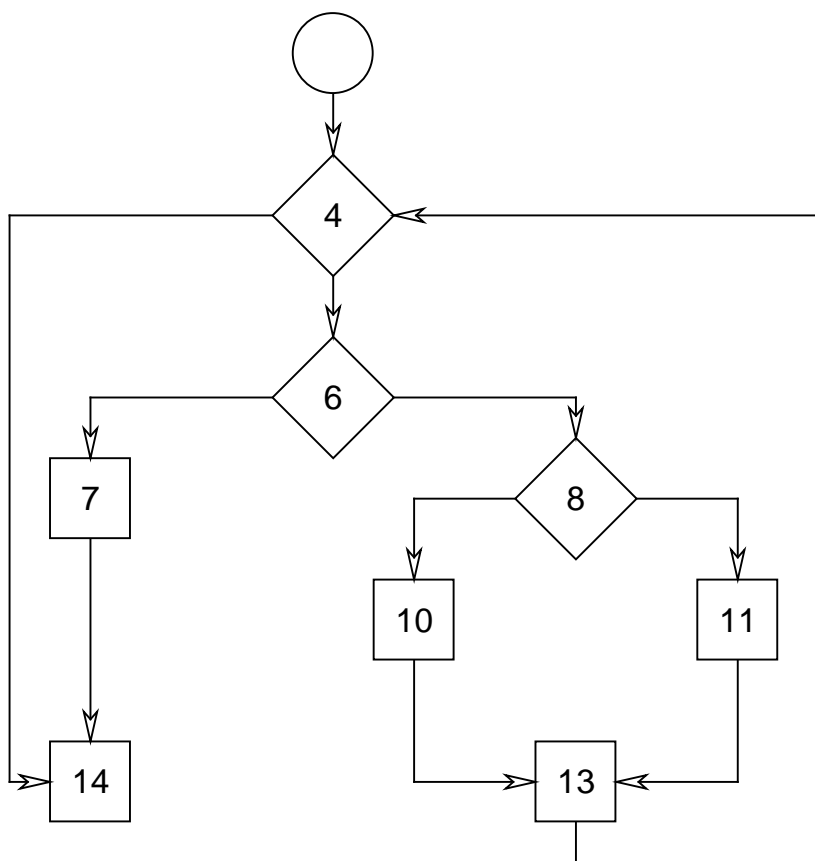
例4： 有下面的C函数，用基本路径测试法进行测试

```
void Sort(int iRecordNum,int iType)
```

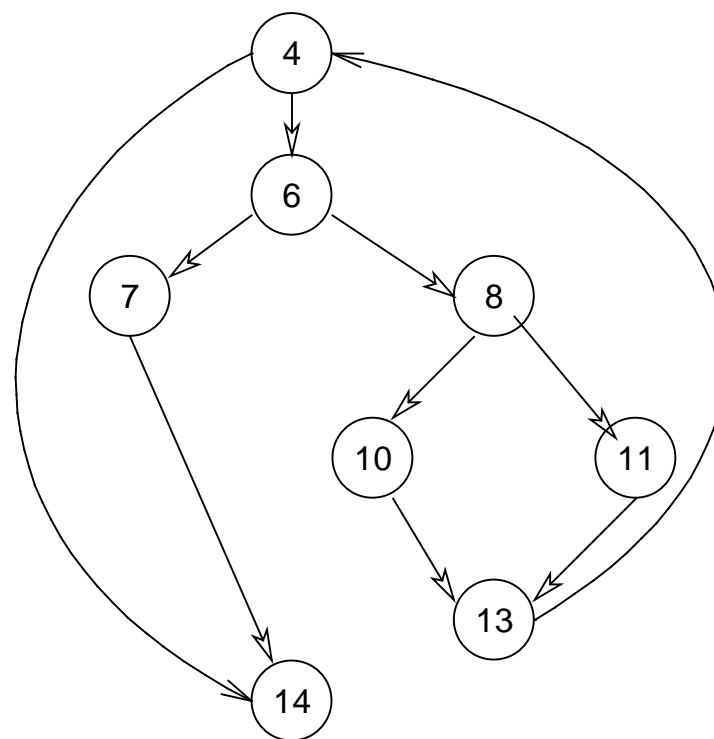
```
1. {  
2.   int x=0;  
3.   int y=0;  
4.   while (iRecordNum-- > 0)  
5.   {  
6.       if(0==iType)  
7.           { x=y+2; break;}  
8.       else  
9.           if (1==iType)  
10.               x=y+10;  
11.       else  
12.           x=y+20;  
13.   }  
14. }
```



第一步：画出控制流图



程序流程图



控制流图

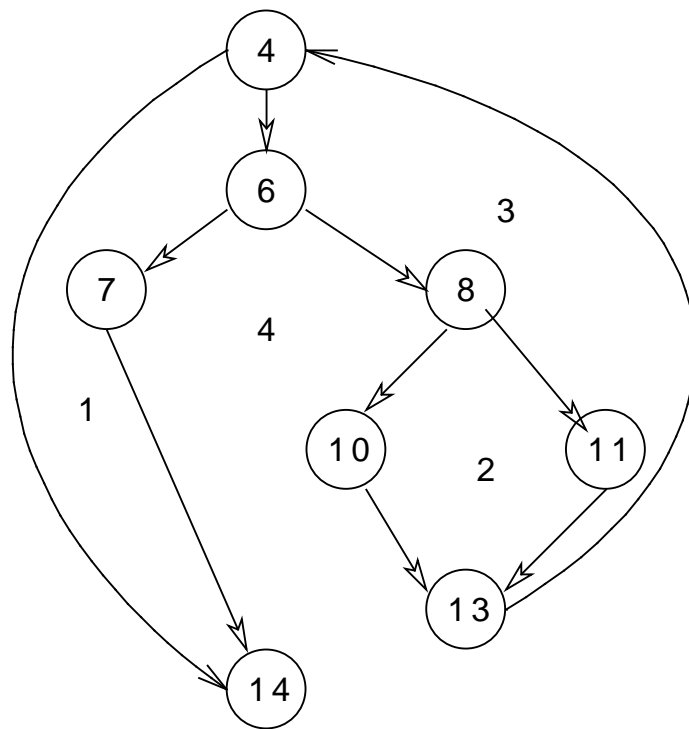
第二步：确定所得流图的环境复杂度

- 环境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境复杂度：
 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。

第二步：计算环复杂度

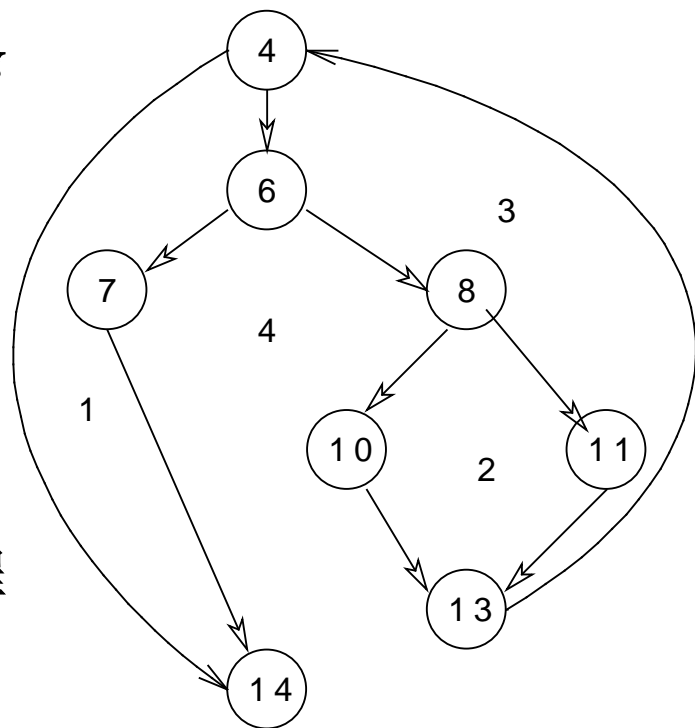
■ 对应上图中的环复杂度，计算如下：

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



第三步：确定线性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。



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。

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

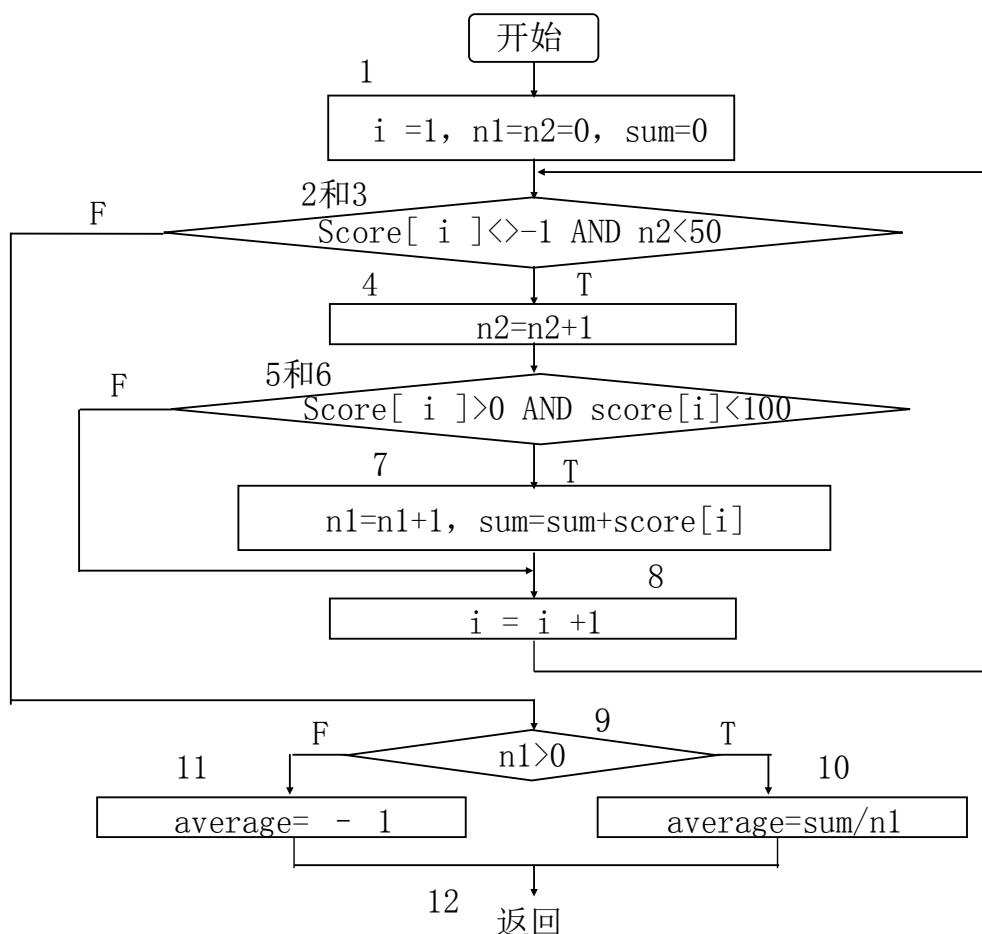
预期结果: x=20

```
void Sort(int iRecordNum,int iType)
```

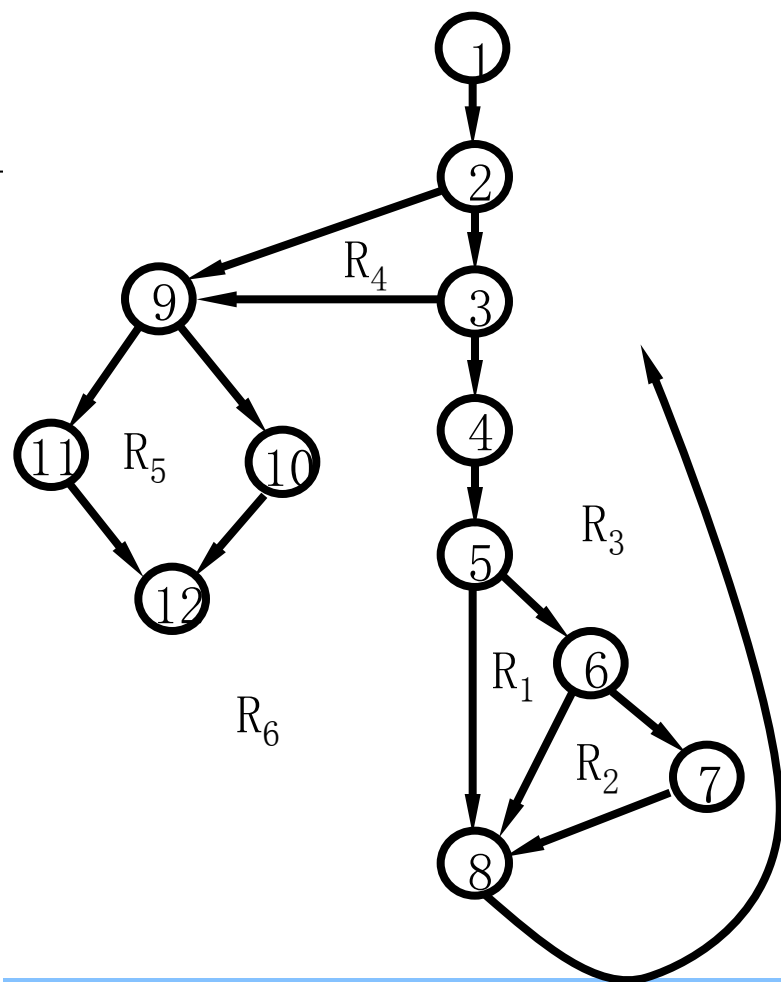
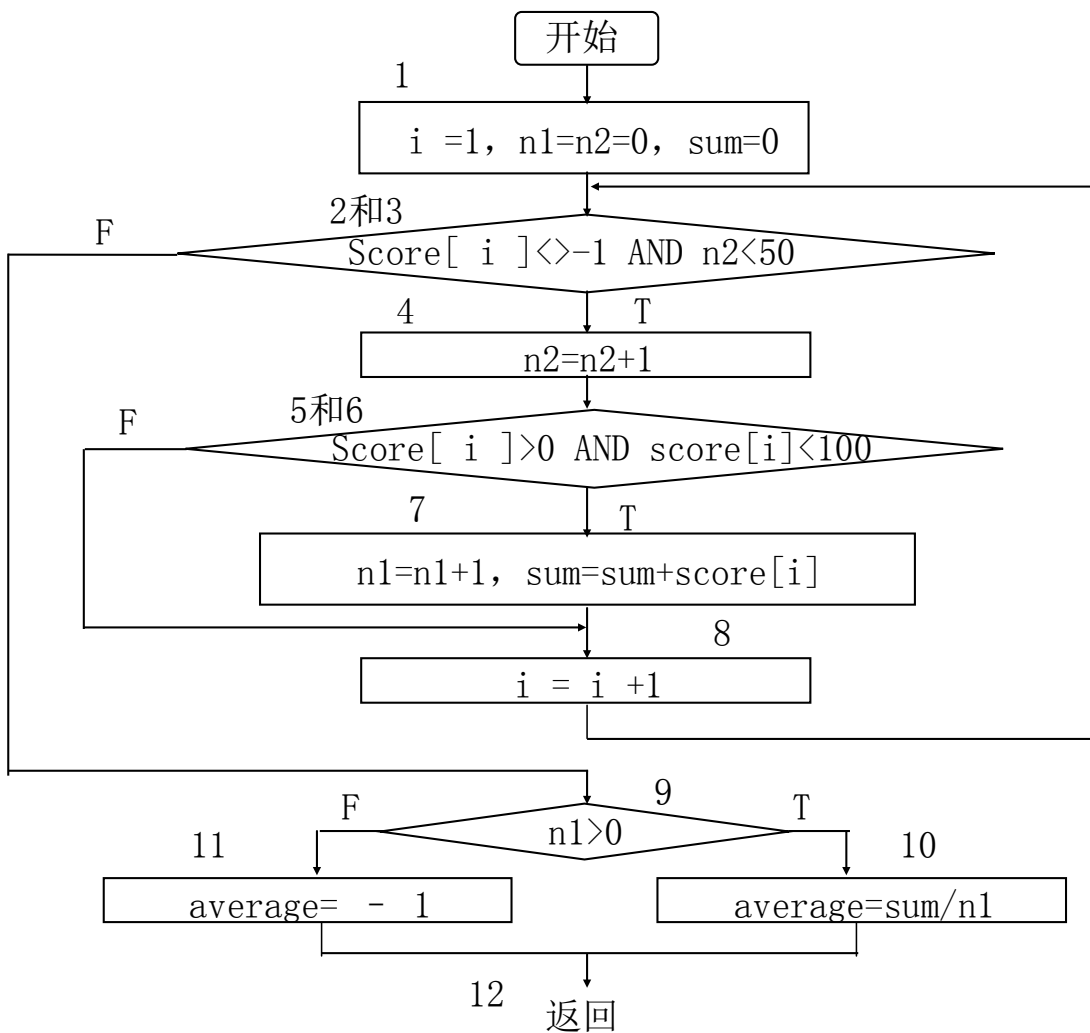
```
1. {  
2.   int x=0;  
3.   int y=0;  
4.   while (iRecordNum-- > 0)  
5.   {  
6.       if(0==iType)  
7.           {x=y+2; break;}  
8.       else  
9.           if(1==iType)  
10.              x=y+10;  
11.          else  
12.              x=y+20;  
13.      }  
14. }
```

示例1：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。



第一步：画出控制流图



第二步：计算圈复杂度

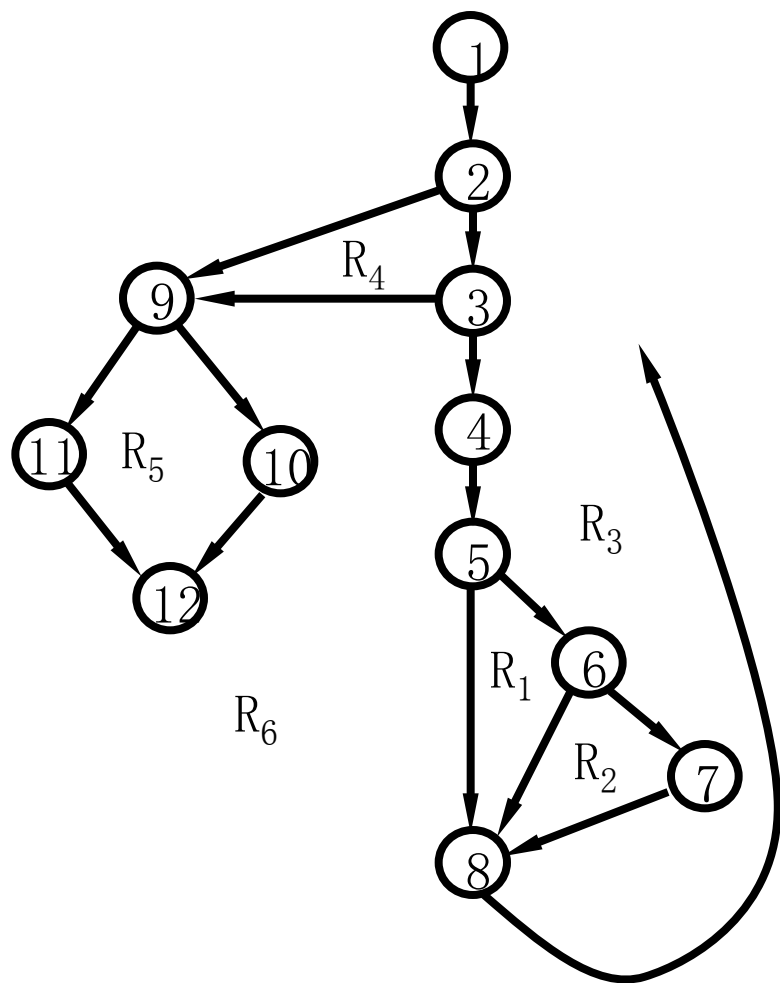
$$V(G) = 6 \text{ (个区域)}$$

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

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** 是谓词结点。



第三步：确定基本路径集合

路径1: 1-2-9-10-12

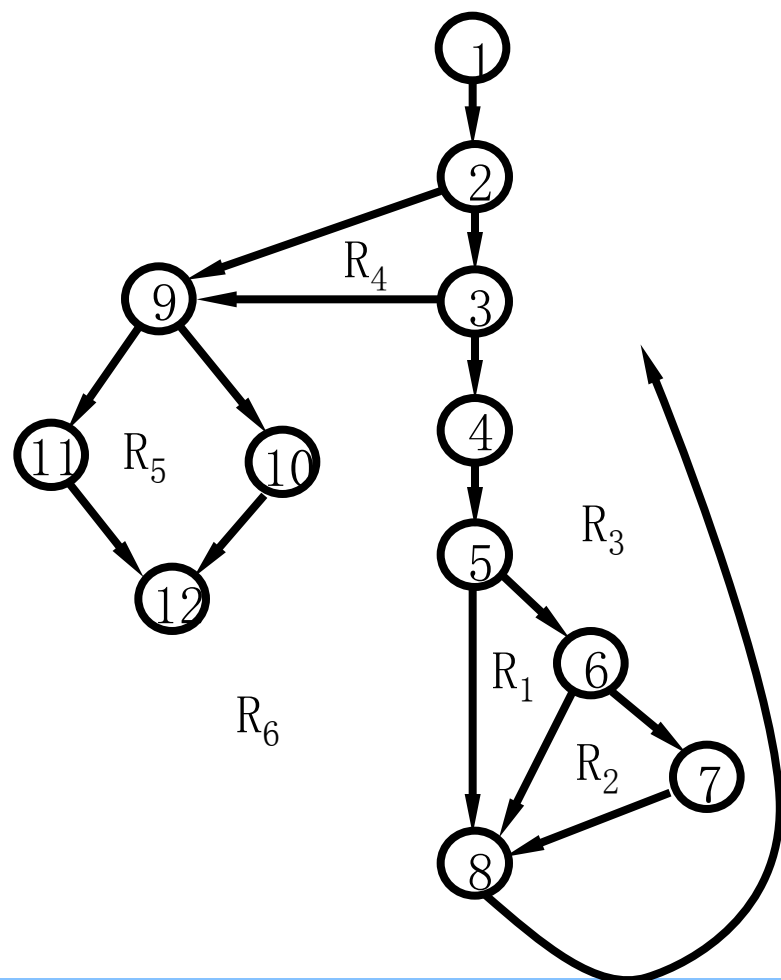
路径2: 1-2-9-11-12

路径3: 1-2-3-9-10-12

路径4: 1-2-3-4-5-8-2...

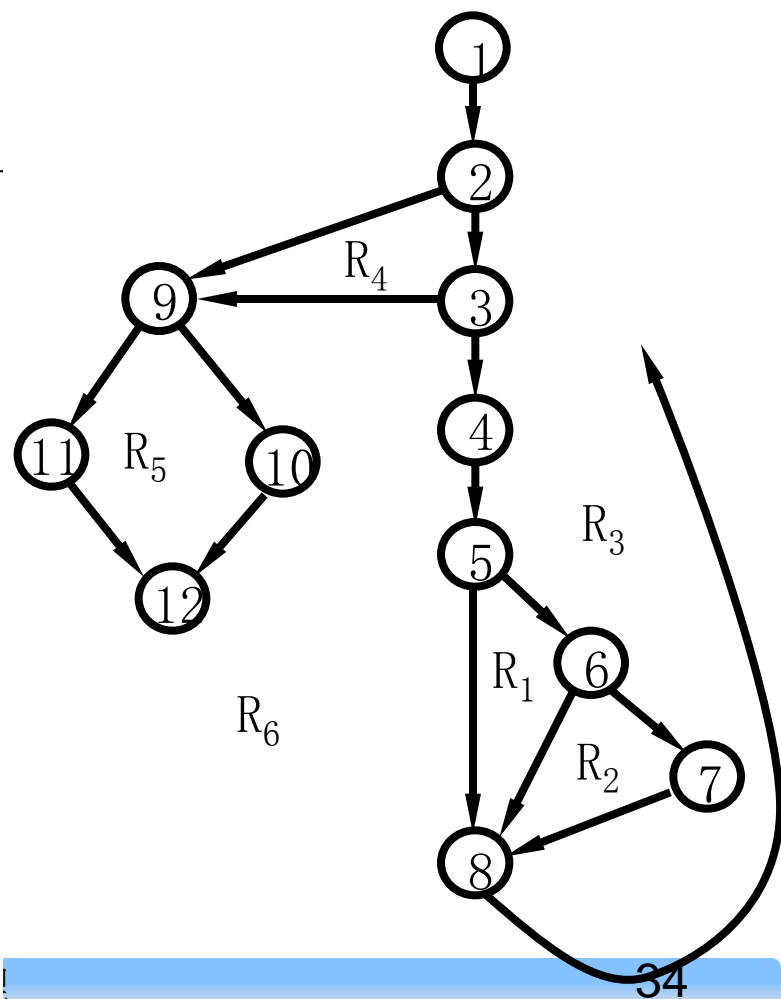
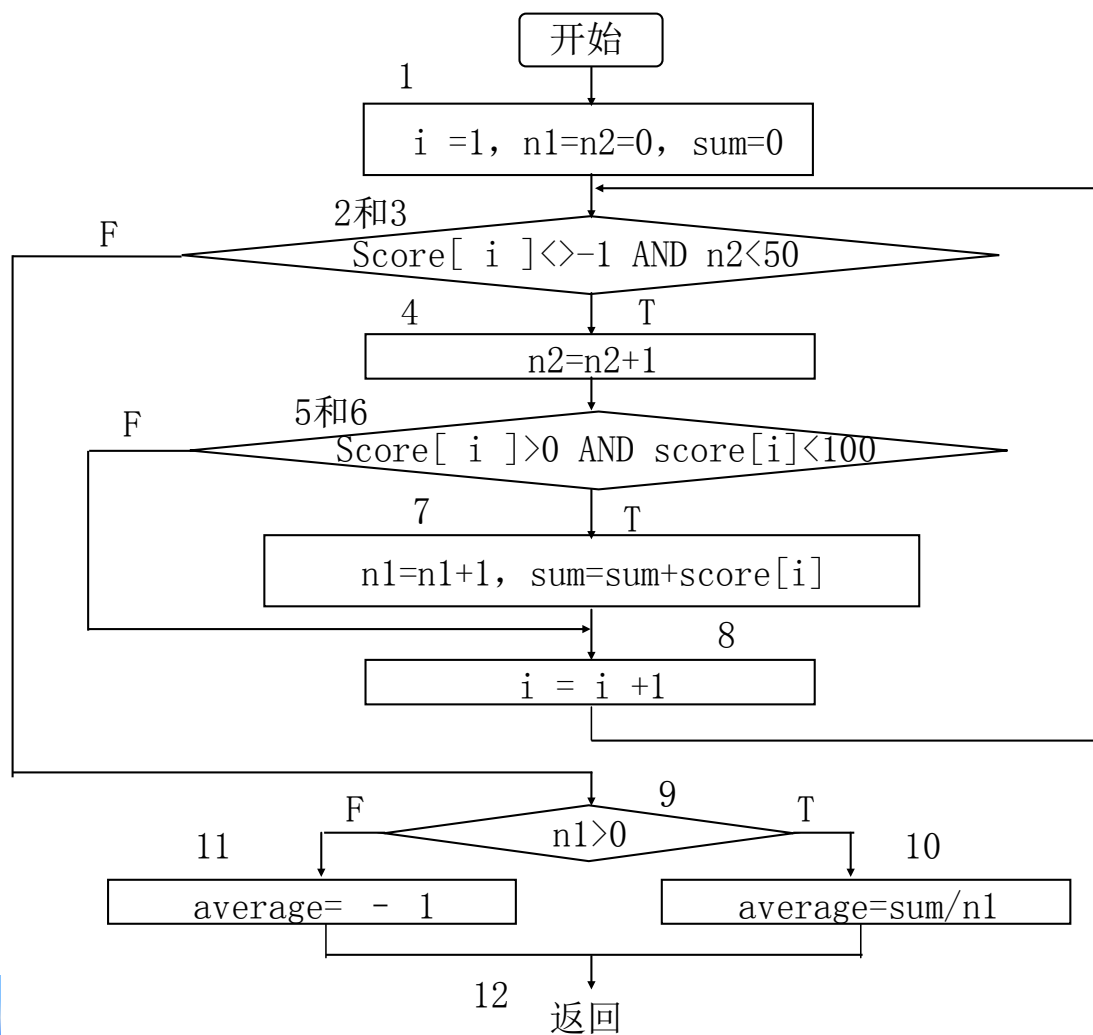
路径5: 1-2-3-4-5-6-8-2...

路径6: 1-2-3-4-5-6-7-8-2...



判定路径达不到

- 路径1-2-9-10-12达不到:



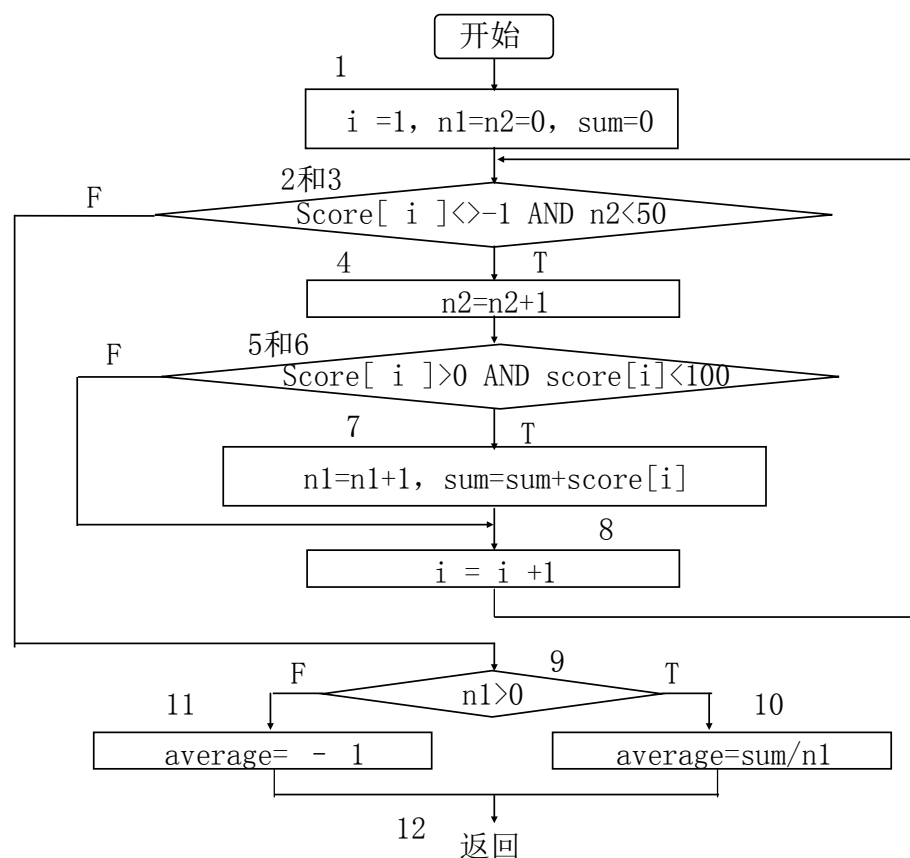
第四步：导出测试用例

- 路径1(1-2-...-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。



第四步：导出测试用例

路径2(1-2-9-11-12)的测试用例：

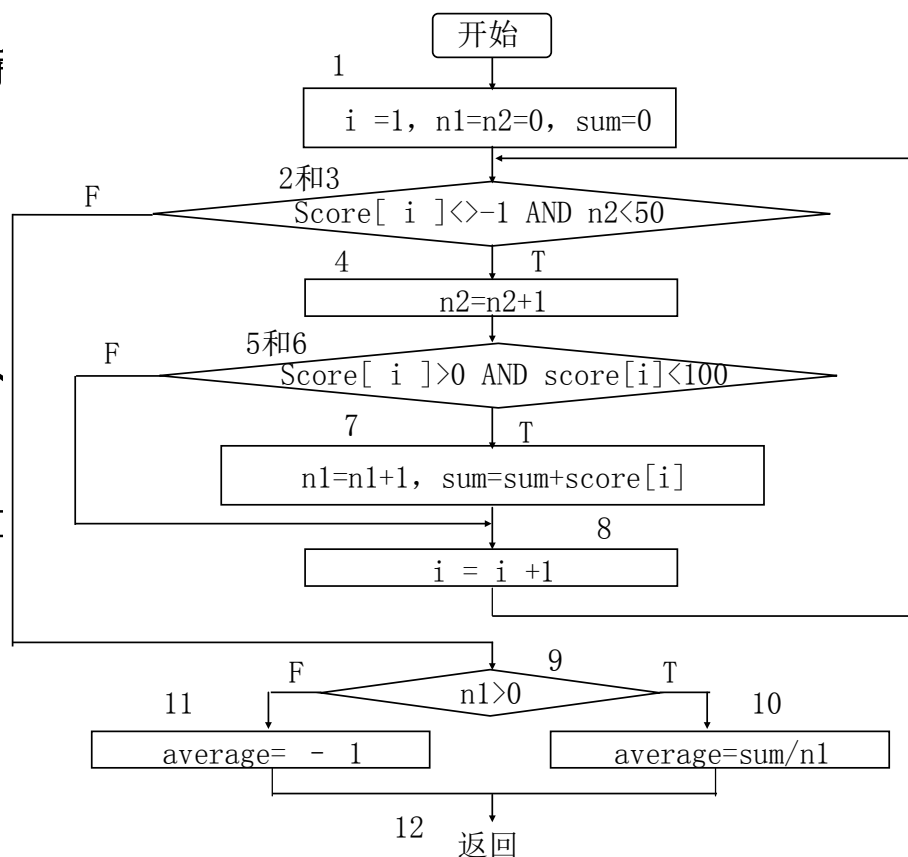
score[1]= -1 ;

期望的结果：**average = -1**，其他量保持初值。

路径3(1-2-...-2-3-9-10-12)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。



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**；

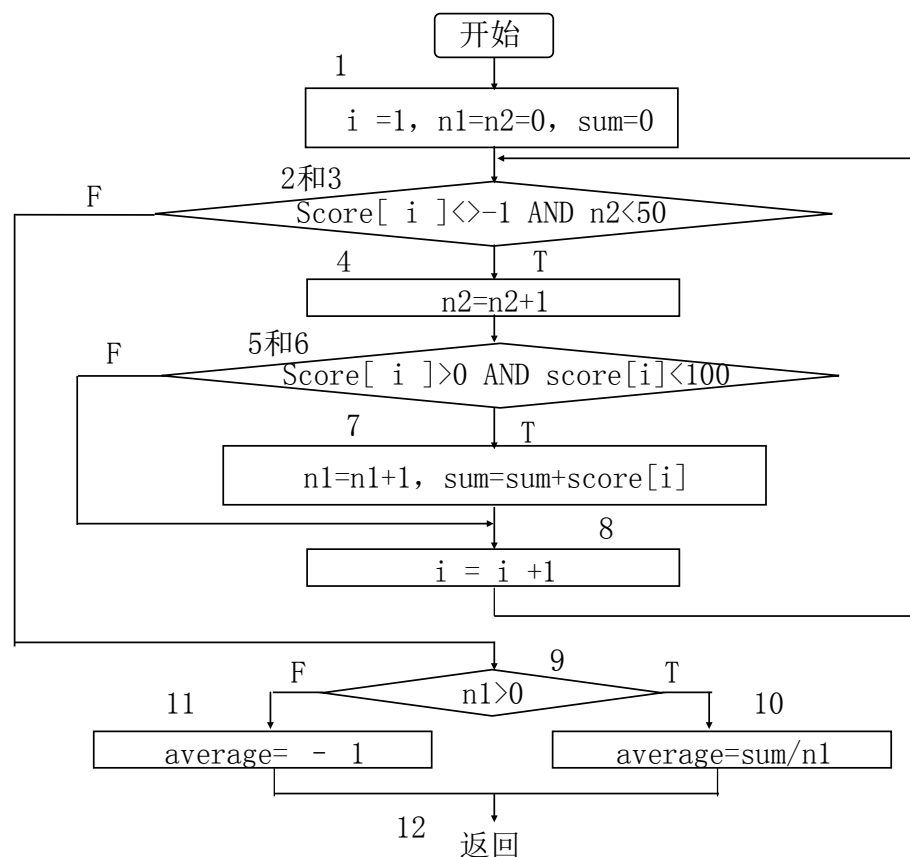
期望结果：根据输入的有效分数算出正确的分数个数**n1**、总分**sum**和平均分**average**。

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**；

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**。

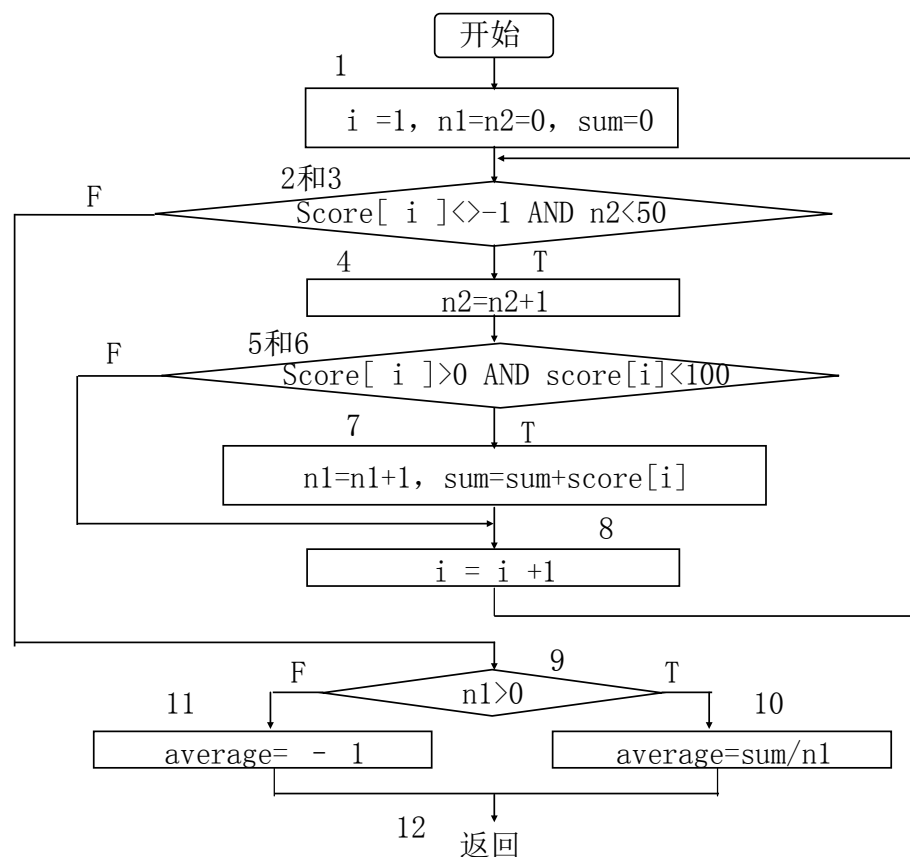


第四步：导出测试用例

路径6(1-2-3-4-5-6-7-8-2...)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。



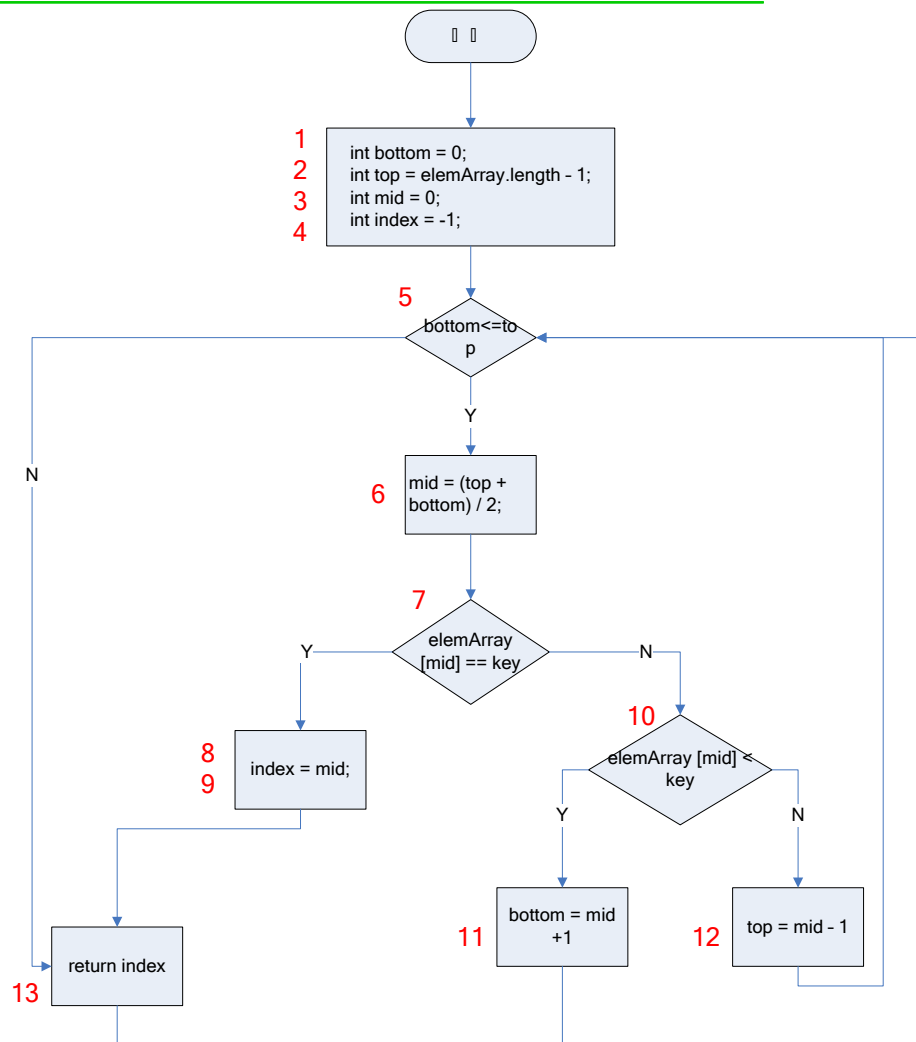
示例2：二分搜索法

- 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。

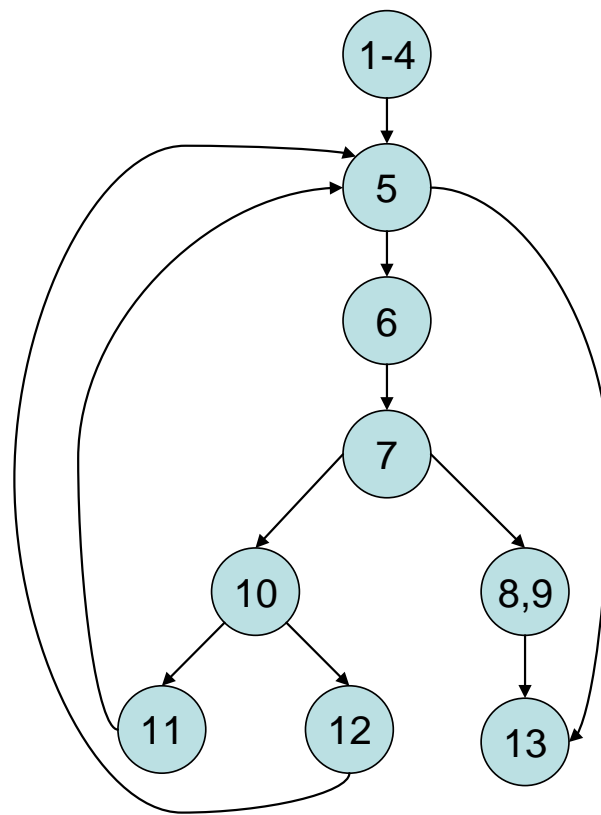
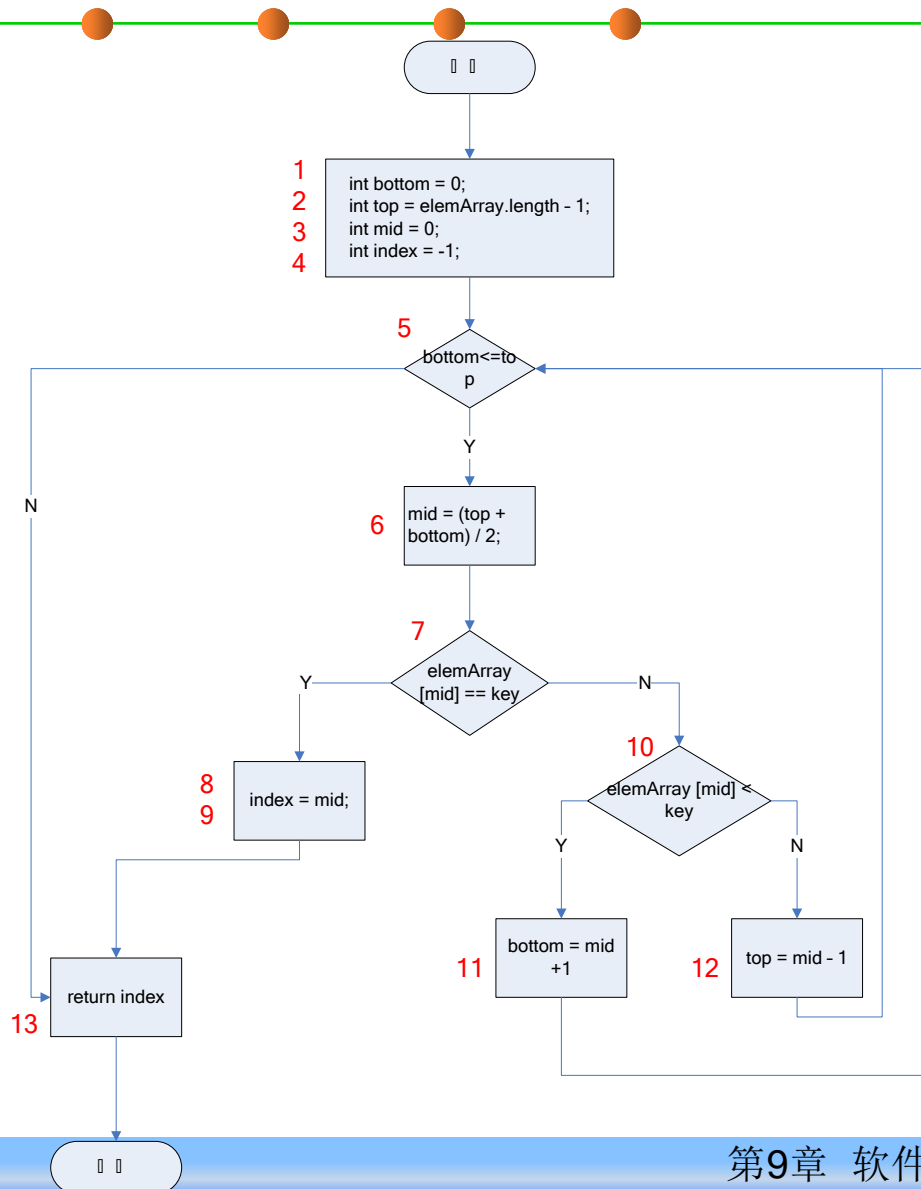
```
int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bommom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }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

模块程序流程图

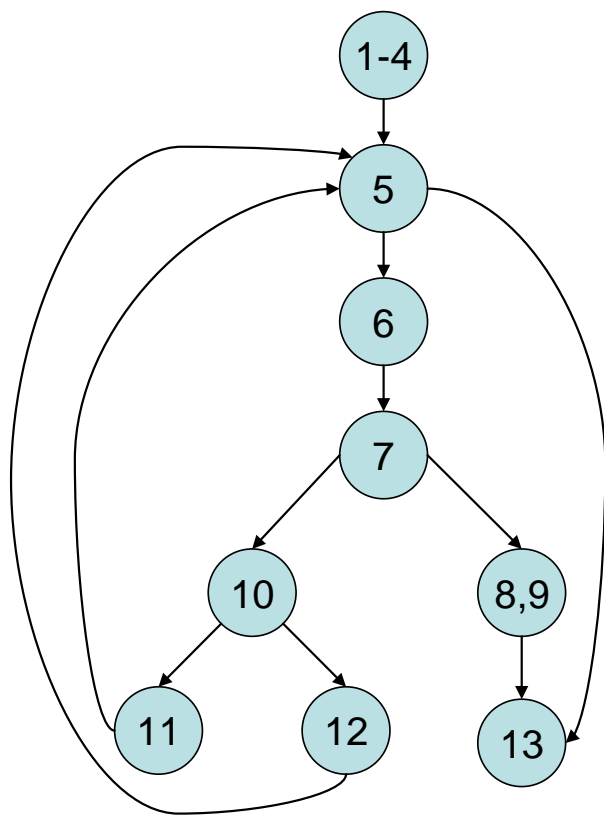
```
int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bommom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }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



控制流图

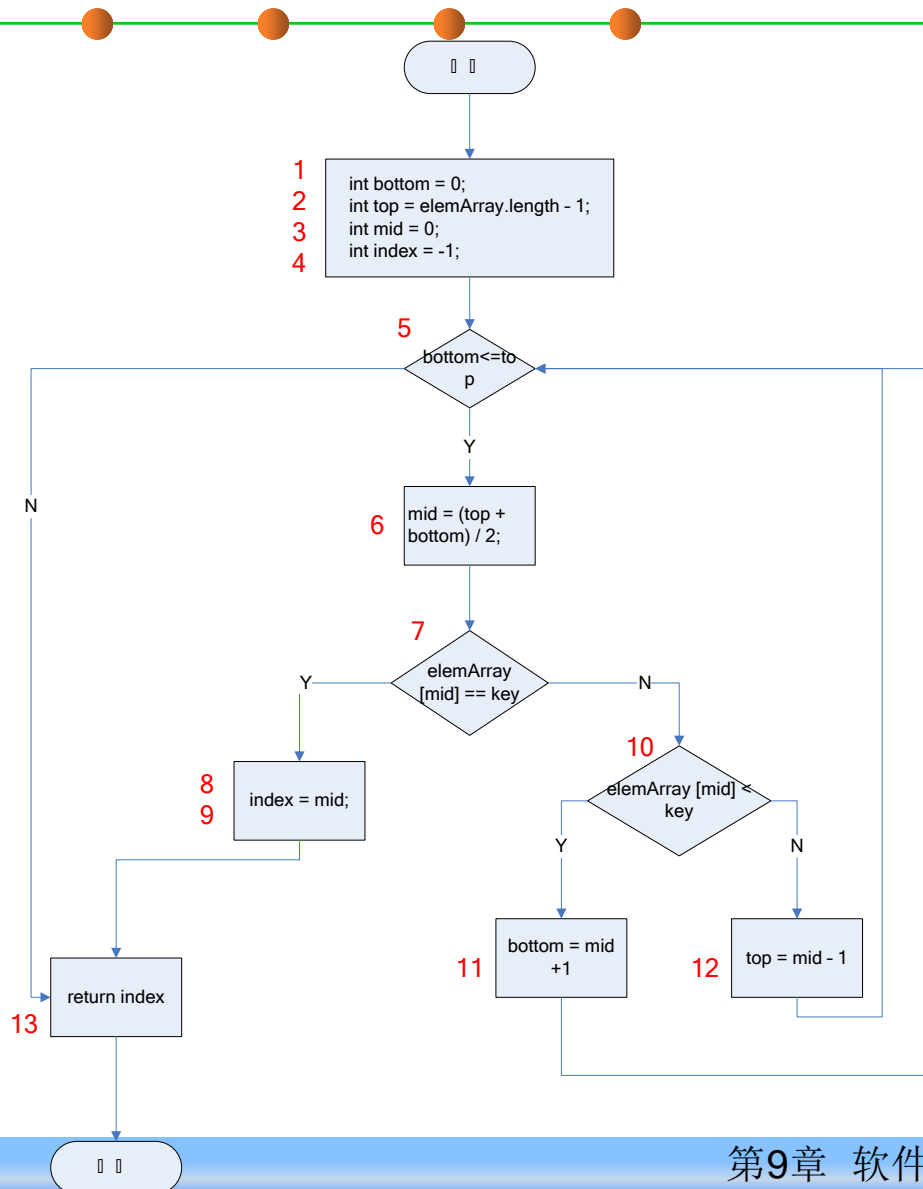


V(G)及独立路径



- $V(G)=4$ 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

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
 - 确定现行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。**

(2) 控制结构测试

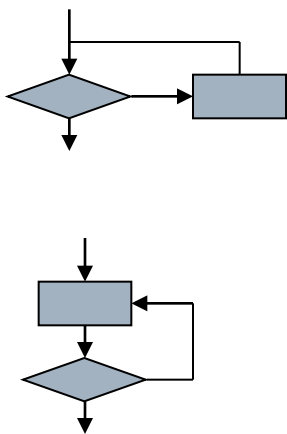
- 基本路径测试是控制测试技术之一，尽管基本路径测试是简单且有效的，但其本身并不充分。
- 控制结构测试提高了白盒测试的质量
 - 逻辑测试
 - 循环测试

逻辑测试

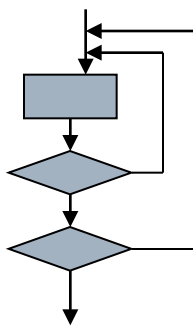
- 组成条件的元素：布尔算符、布尔变量、括号、关系算符、算术表达式等
- 逻辑测试通过检查程序模块中包含的逻辑条件进行测试用例设计
- 逻辑测试方法侧重于测试程序中的每个条件以确保其不包含错误
- 例：if **a>b or a+b>100** then

循环测试

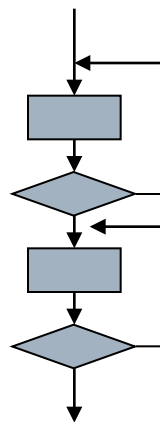
- 循环测试是一种白盒测试技术，注重于循环构造的有效性。
- 四种循环：简单循环，串接(连锁)循环，嵌套循环和不规则循环



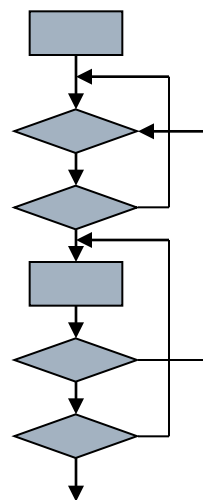
简单循环



嵌套循环



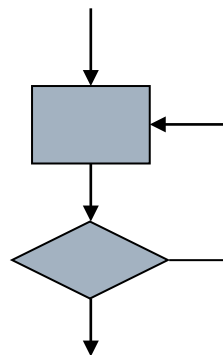
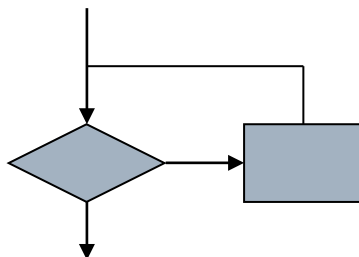
串接循环



无结构循环

简单循环

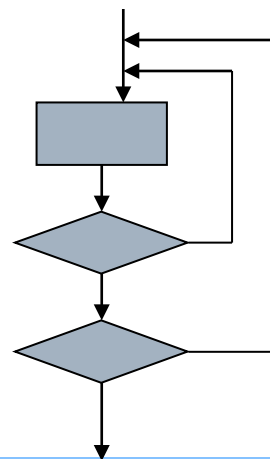
- 对于简单循环，测试应包括以下几种，其中的 n 表示循环允许的最大次数。
 - 零次循环：从循环入口直接跳到循环出口。
 - 一次循环：查找循环初始值方面的错误。
 - 二次循环：检查在多次循环时才能暴露的错误。
 - m 次循环：此时的 $m < n$ ，也是检查在多次循环时才能暴露的错误。
 - n (最大)次数循环、 $n+1$ (比最大次数多一)次的循环、 $n-1$ (比最大次数少一)次的循环。



嵌套循环

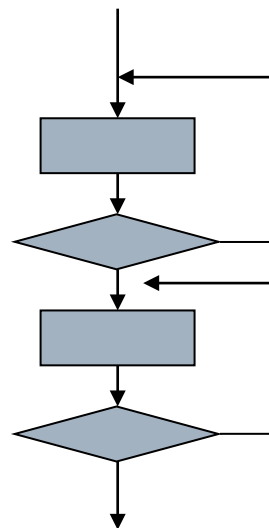
■ 对于嵌套循环：

- 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。



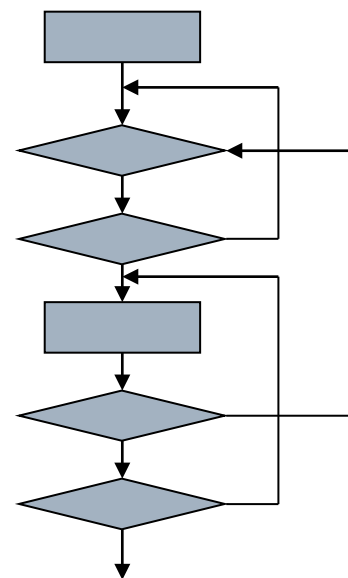
串接循环

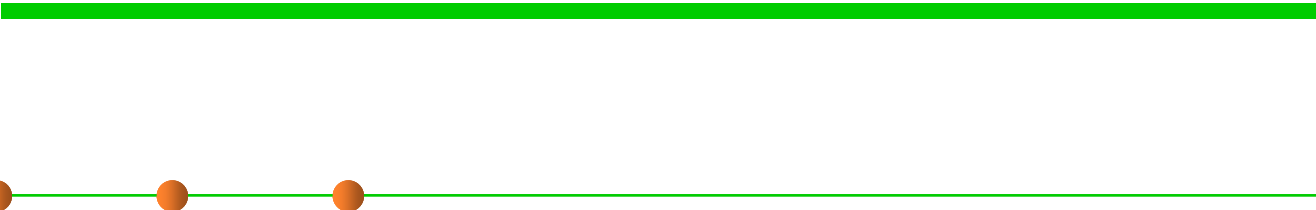
- 对于串接循环，要区别两种情况。
 - 如果各个循环互相独立，则串接循环可以用与简单循环相同的方法进行测试。
 - 如果有两个循环处于串接状态，而前一个循环的循环变量的值是后一个循环的初值。则这几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。



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。





结束

2015年9月18日