第二部分 软件测试基础知识

2.12 动态白盒测试（控制流测试）

本节教学目标

- 理解白盒测试与黑盒测试的区别
- 掌握动态白盒测试的方法
- 重难点：动态白盒测试的方法

内容回顾

- 什么是测试用例
  - 为实施测试而向被测试系统提供的输入数据、操作或各种环境设置以及期望结果等信息的一个特定集合
- 为什么要书写测试用例
  - 理思路，追踪过程，做之后过程的参考
- 缺陷定义
  - Bug的来历
  - IEEE的定义
  - 从产品内部看：
    - 软件缺陷是软件产品开发或维护过程中所存在的错误、毛病等各种问题；
  - 从产品外部看：
    - 软件缺陷是系统所需要实现的某种功能的失效或违背。

内容回顾

- 缺陷产生的原因及修复成本
- 缺陷报告的编写
- 缺陷严重性和优先级
- 缺陷状态及周期性
- 缺陷工具使用

内容回顾

- 测试过程管理
  - 创建产品
  - 创建项目（迭代）
  - 写测试计划
  - 测试计划评审并修改
  - 设计测试用例
  - 测试用例评审并修改
  - 创建测试版本
  - 提交测试

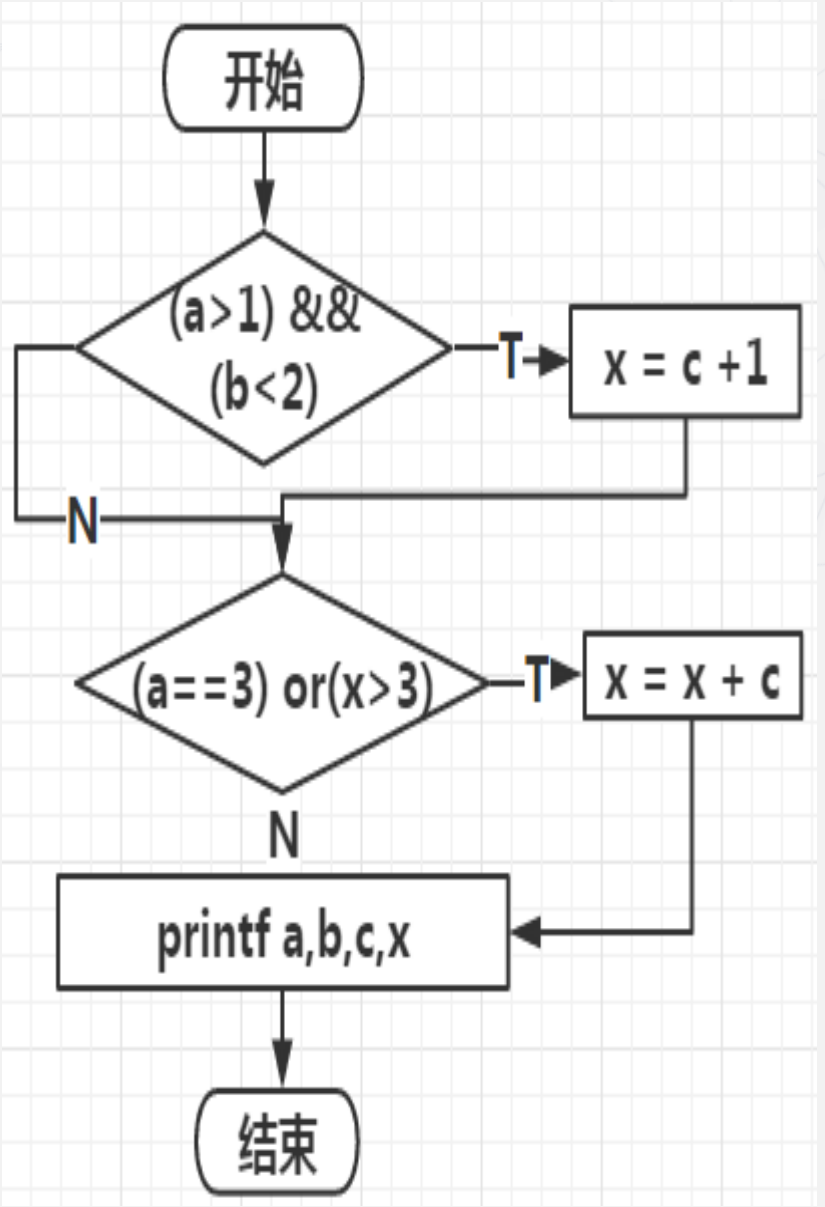
- 测试过程管理
  - 关联测试用例
  - 执行测试
  - 记录bug
  - 追踪bug
  - 分析.....

目录

白盒测试

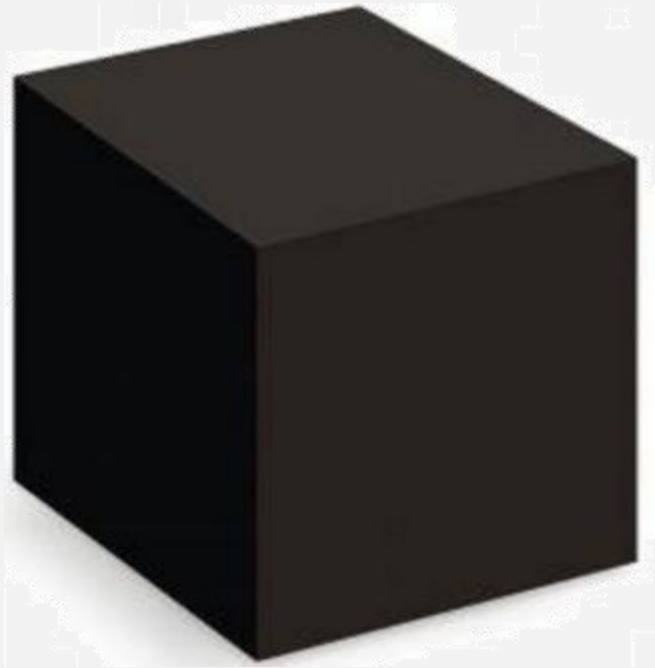
```
1 Int Func(int a,int b,int c,int x){
2   if(a>1)&&(b<2)
3       x = c + 1;
5    if(a==3)||(x>3)
5       x = x + c;
6    printf("a=%d,b=%d,c=%d,x=%d\n",a,b,c,x);
7    return x;
8 }
```

分别使用语句、判定、条件、判定条件、条件组合、路径覆盖设计测试用例



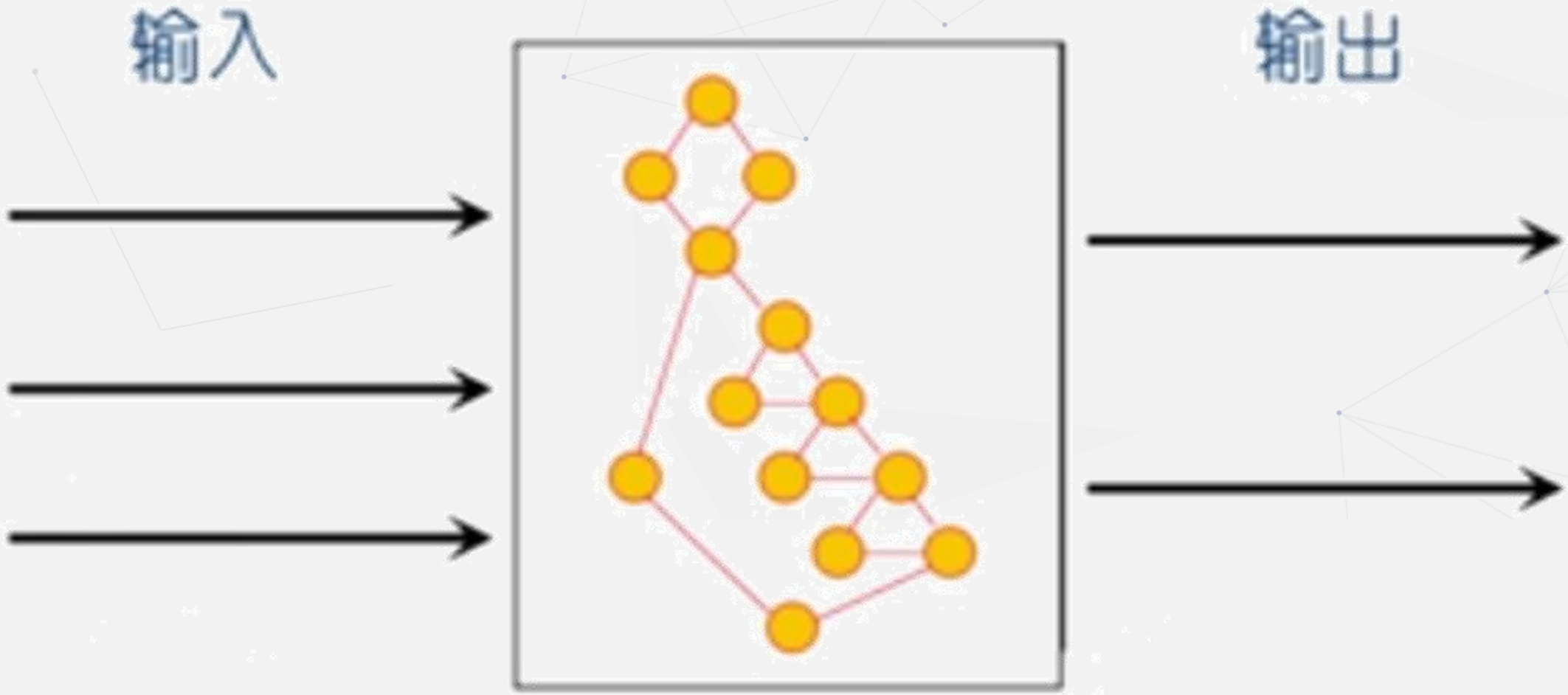
白盒测试概述

- 黑盒测试基本原理：



白盒测试概述

- 白盒测试基本原理：



白盒测试概述

- 白盒测试关注的对象：
  - 源代码：
    - 阅读源代码，检查代码规范性，并对照函数功能查找代码的逻辑缺陷、内存管理缺陷、数据定义和使用缺陷等
  - 程序结构：
    - 使用与程序设计相关的图表，找到程序设计的缺陷，或评价程序的执行效率

白盒测试概述

- 白盒测试与黑盒测试比较
  - 黑盒测试：功能级别
  - 白盒测试：函数级别

白盒测试概述

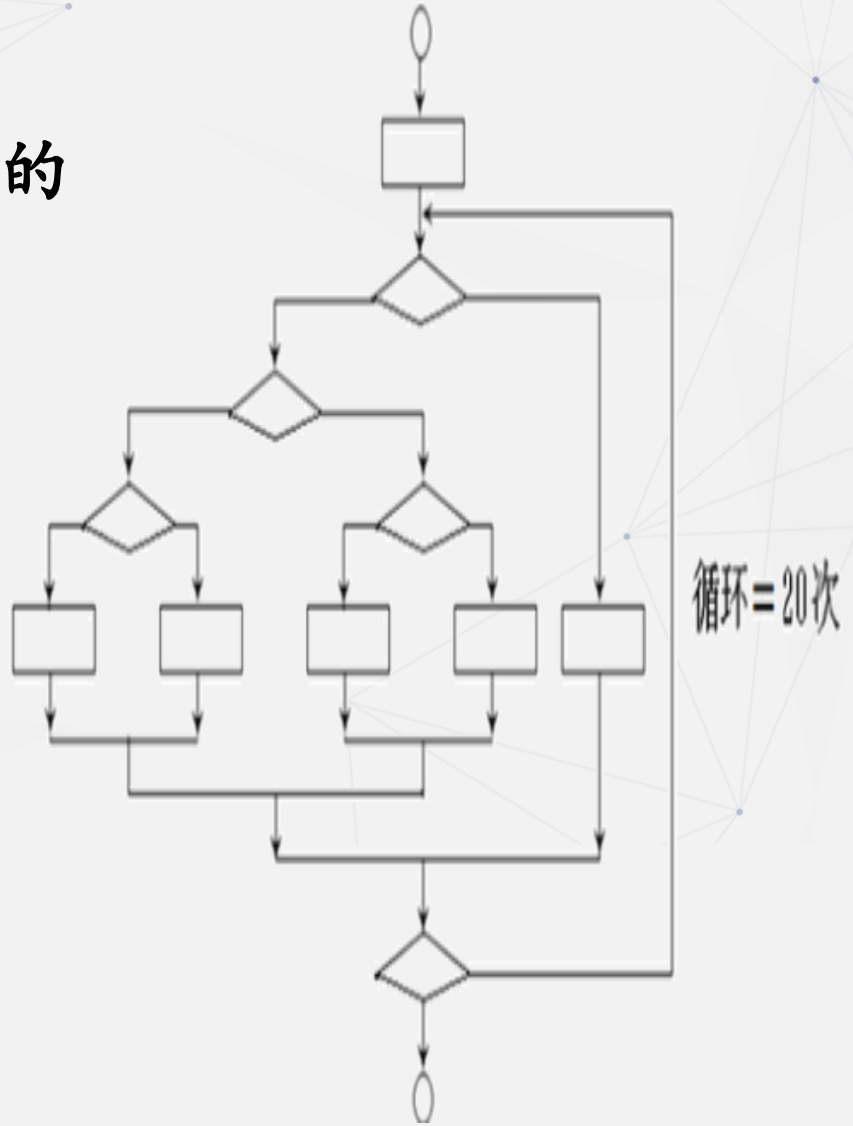
- 优势：
  - 针对性强，便于快速定位缺陷
  - 在函数级别开始测试工作，缺陷修复成本低
  - 有助于了解测试覆盖程度
  - 有助于代码优化和缺陷预防

白盒测试概述

- 不足和弊端：
  - 对测试人员要求高：
    - 测试人员需要具备一定的编程经验
    - 白盒测试工程师需要具备广博的知识
  - 成本高：
    - 白盒测试需要的时间长

白盒测试概述

- 白盒测试其他注意知识：
  - 通过测试无法证明，被测系统是没有缺陷的
  - 软件测试的经济学问题
  - 解决的办法——白盒测试
  - 对每一种可能的执行路径进行测试，是否可行？



程序缺陷的客观原因

- 用户需求越来越多
- 系统越来越庞大
- 程序的结构越来越复杂
- 程序正确实现的难度越来越高

白盒测试概述

- 控制流分析要解决的问题：
  - 什么因素导致程序结构变得复杂
  - 如何衡量程序结构的复杂程度
  - 控制程序执行流程发生变化的主要因素是什么
  - 如何测试这些因素并确保测试的效率

目录

程序结构分析

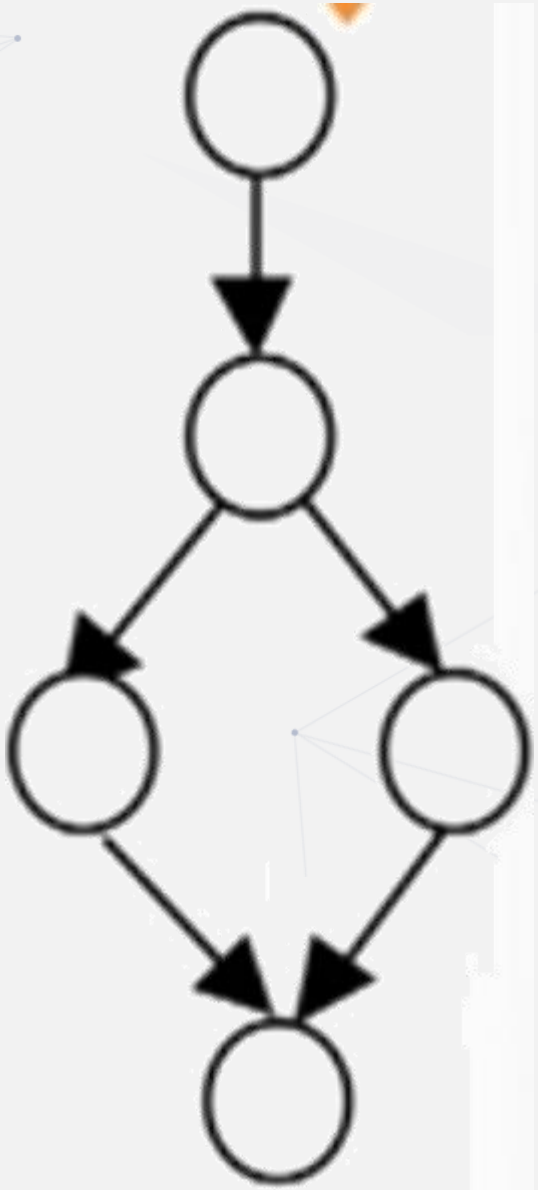
- 顺序结构：

```
void Func1( int a )
{
    int b;
    b = a + 1;
    printf( “a = %d, b = %d\n”, a, b );
}
```

程序结构分析

- 条件判定结构：

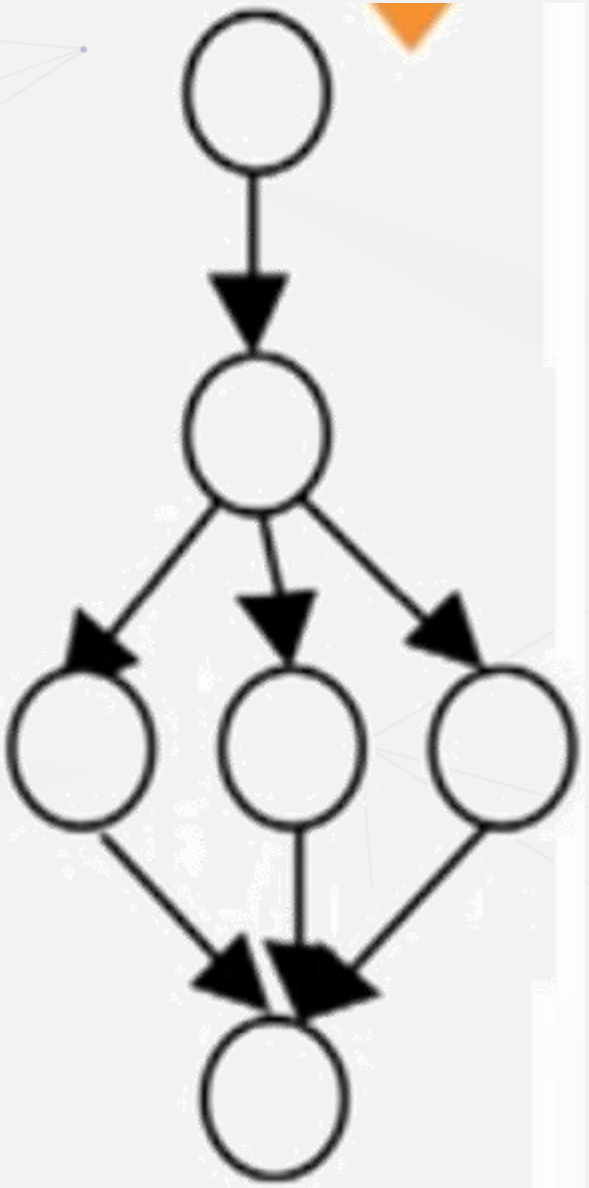
```
void Func2( int a )
{
    int b = 0;
    if( a > 1 )
        b = a + 1;
    else
        b = a – 1;
    printf( “a = %d, b = %d\n”, a, b );
}
```



程序结构分析

- 条件判定结构：

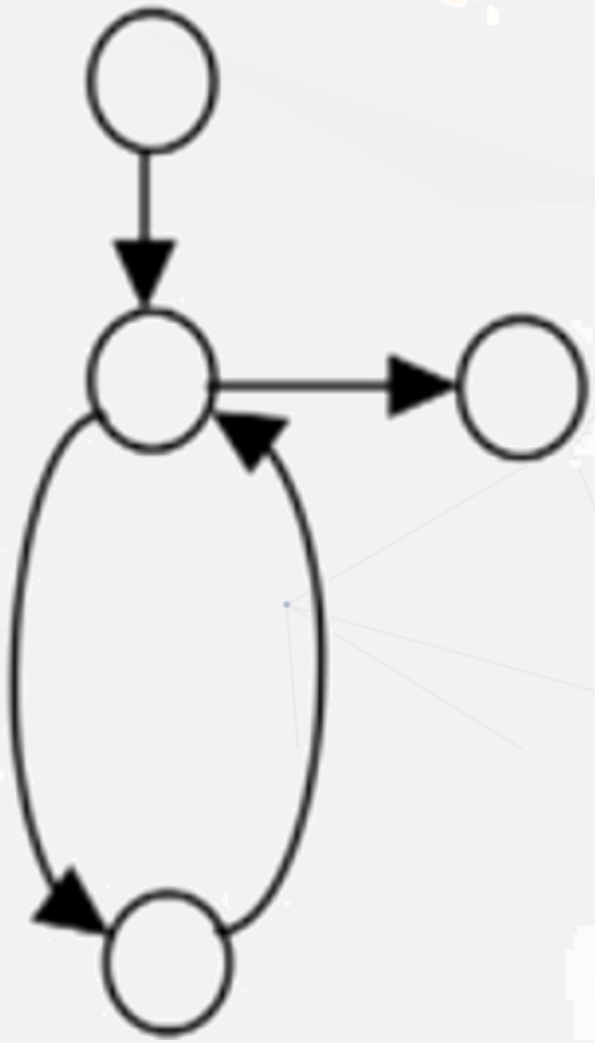
```
void Func3( int a )
{
    int b = 0;
    switch( a ){
        case 0: b = a; break;
        case 1: b = a * 2; break;
        case 2: b = a * 3; break;
        default: break;
    }
    printf( “a = %d, b = %d\n”, a, b );
}
```



程序结构分析

- while-do循环结构

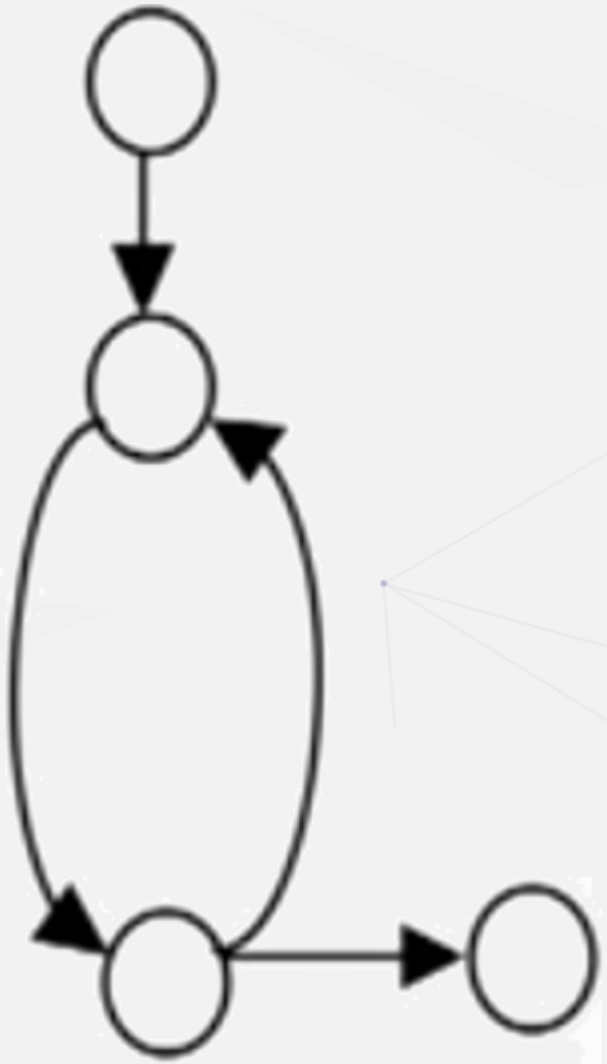
```
void Func4( int a )
{
    int b = 0;
    int i = 1;
    while( i < 10 ){
        b = b + a*i;
        i ++;
    }
    printf( "a = %d, b = %d\n", a, b );
}
```



程序结构分析

- do-while 循环

```
void Func5( int a )
{
    int b = 0;
    int i = 1;
    do{
        b = b + b*i;
        i ++;
    }while( i < 10 );
    printf( "a = %d, b = %d\n", a, b );
}
```

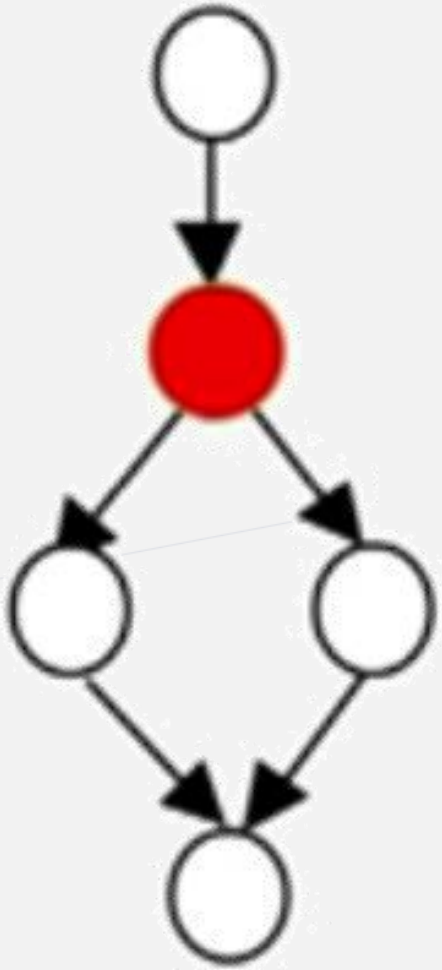


程序结构分析

- 常见程序结构：

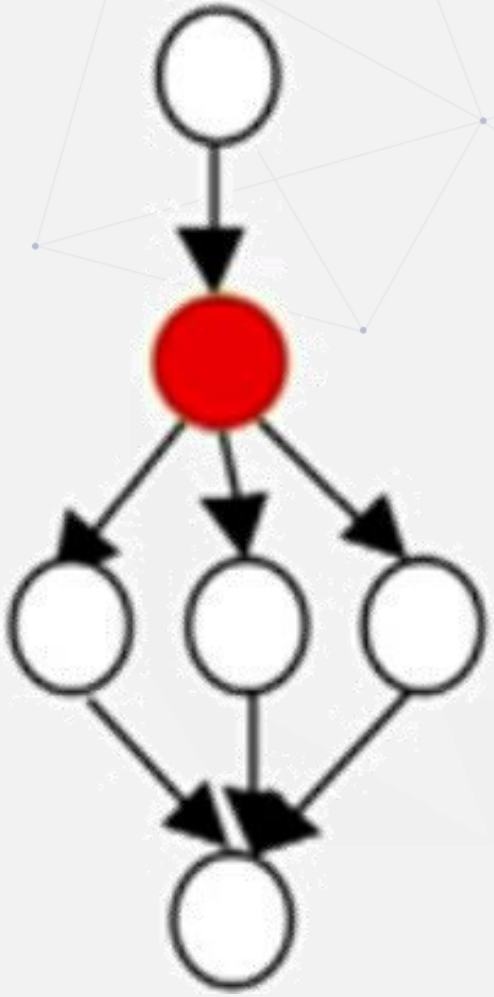
串行结构

<



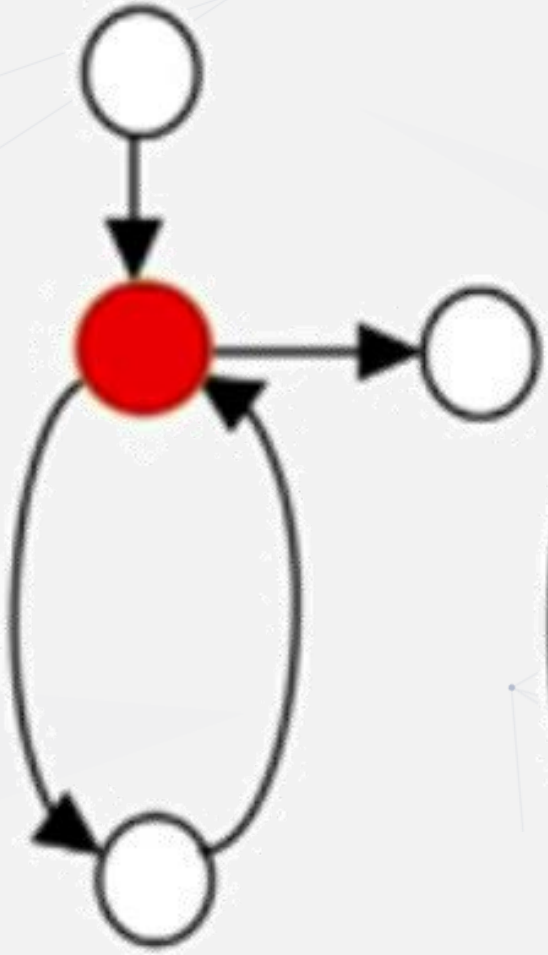
两分支的条件判断

<



多分支的条件判定

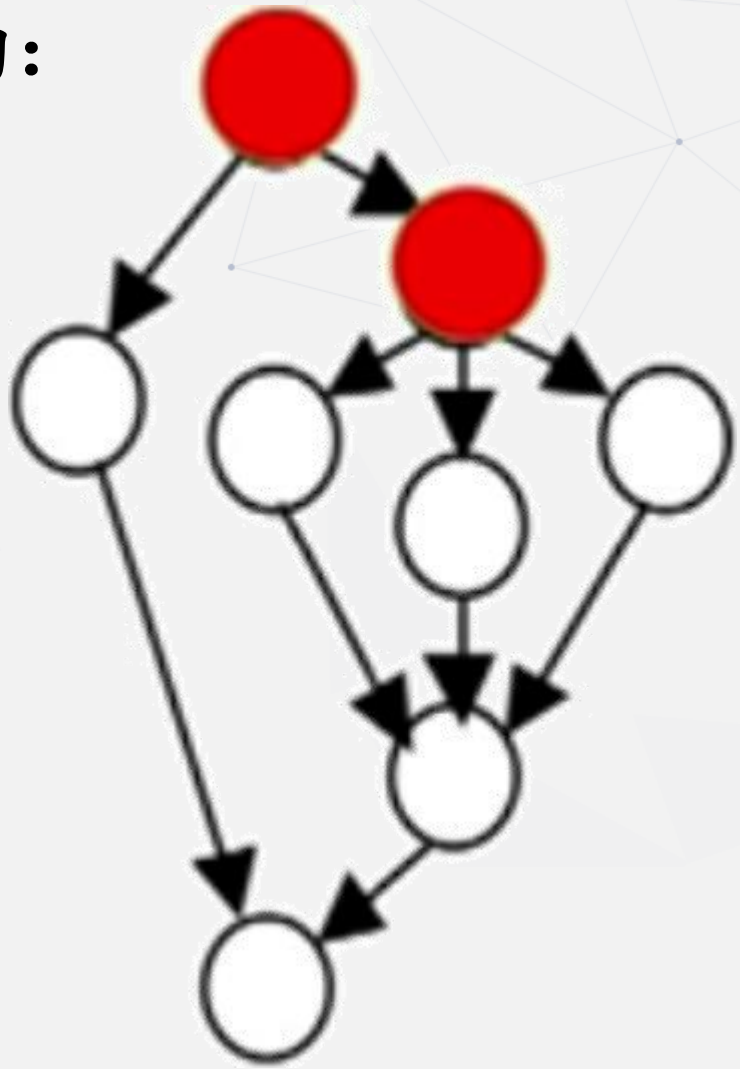
<



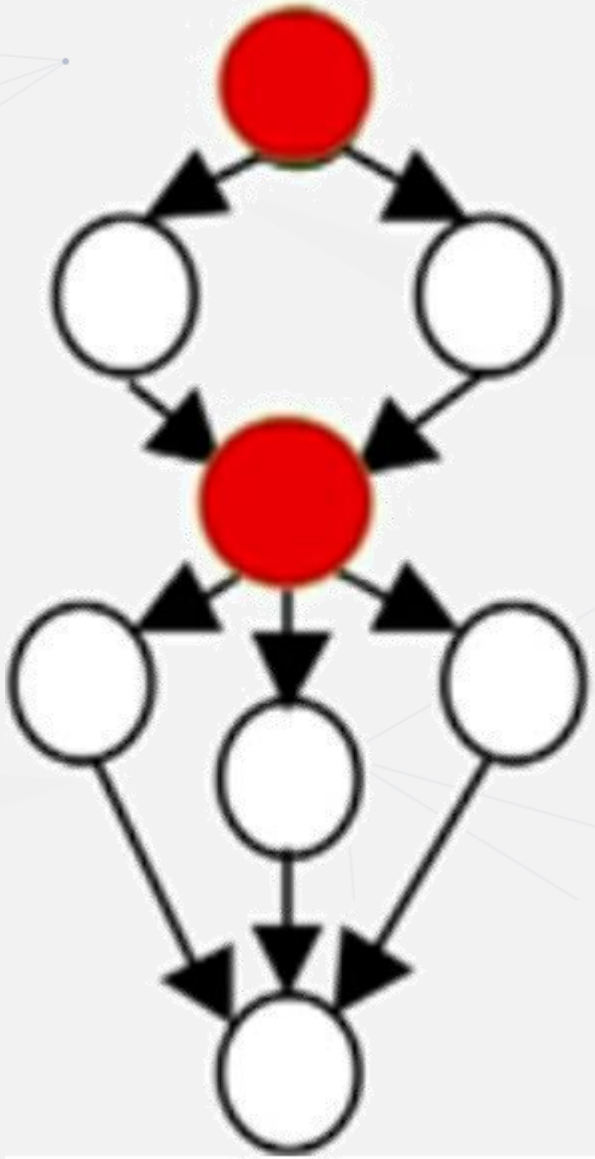
循环结构

程序结构分析

常见的程序结构：



嵌套



串联

程序结构分析

- 控制流主要解决的问题：
  - 导致程序结构变得复杂的主要原因是什么？
  - 控制程序执行流程发生变化的主要因素是什么？

    ——判定节点
  - 使用什么方式进行测试？

    ——逻辑覆盖

目录

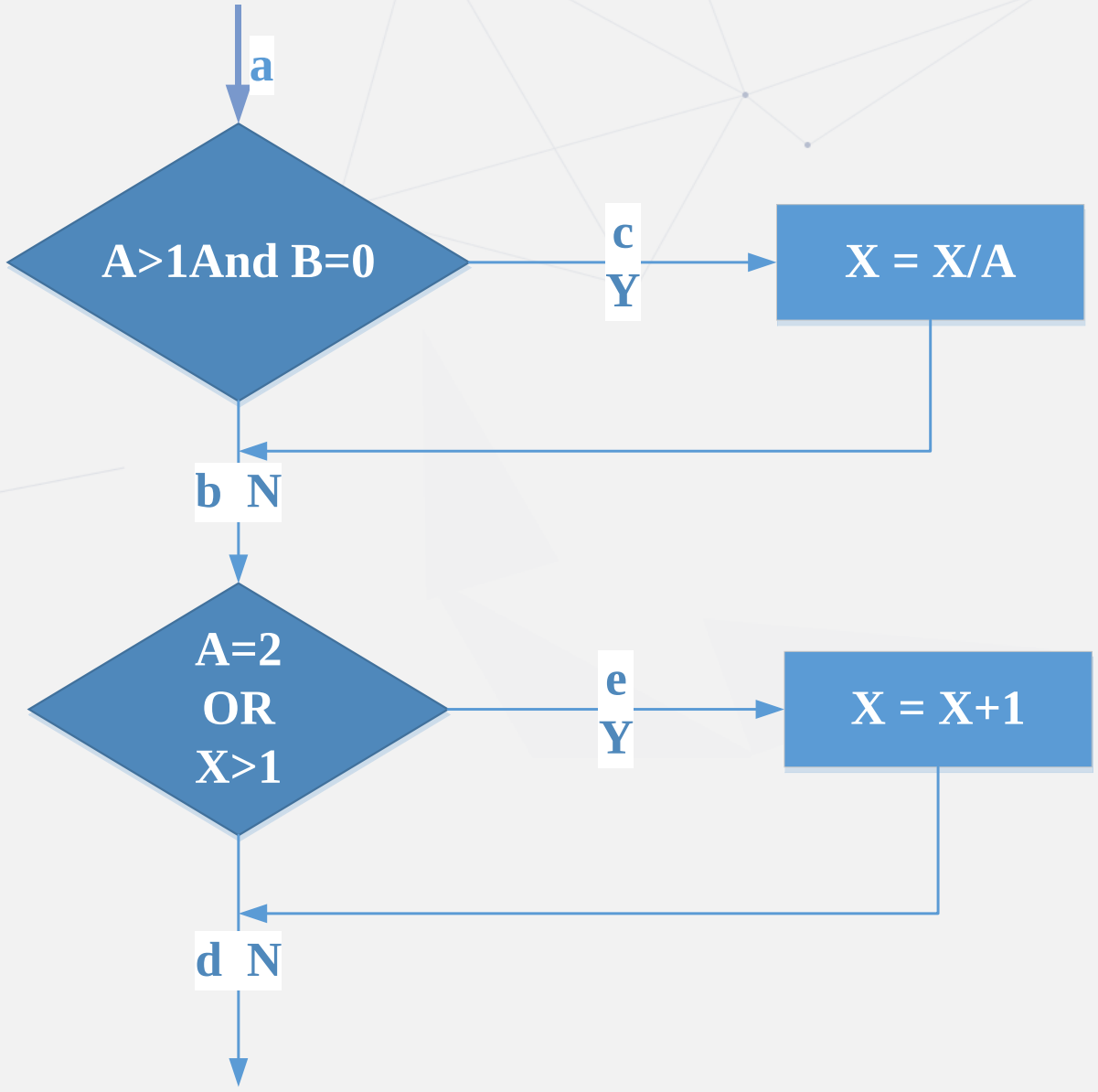
语句覆盖(sentence cover）

```
void main()
{
 float A,B,X;
scanf(“%f%f%f”,&A,&B,&X);
if((A>1)&&(B==0))
    X=X/A;
 if((A==2)||(X>1))
     X=X+1;
  printf(“%f”,X);
}
```

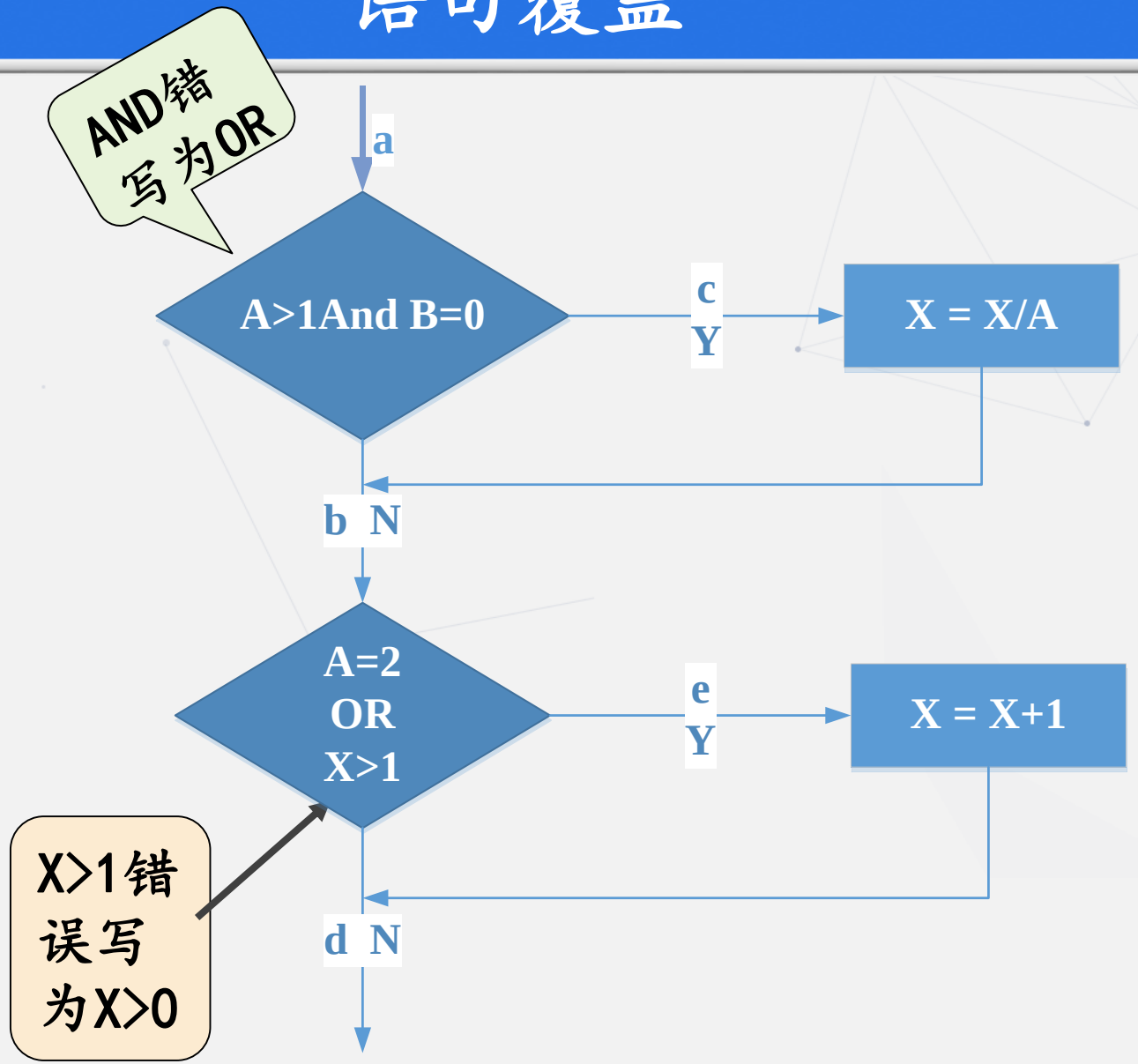
语句覆盖定义

- 语句覆盖：是一个比较弱的测试标准，设计若干测试用例，使得程序中每个可执行语句至少都能被执行一次。

语句覆盖



语句覆盖



- 使程序中每个语句至少执行一次
- 设计一个能通过路径ace的例子就可以了
- 测试用例输入数据：
- A=2，B=0，X=3

判定覆盖—概念

- 判定覆盖：使得程序中每个分支至少都获得一次“真值”或“假值”，又称分支覆盖。是一个比“语句覆盖”稍强的测试标准。

判定覆盖

```
void main()
{
 float A,B,X;
scanf(“%f%f%f”,&A,&B
 ,&X);
if((A>1)&&(B==0))
    X=X/A;
 if((A==2)||(X>1))
     X=X+1;
  printf(“%f”,X);
}
```

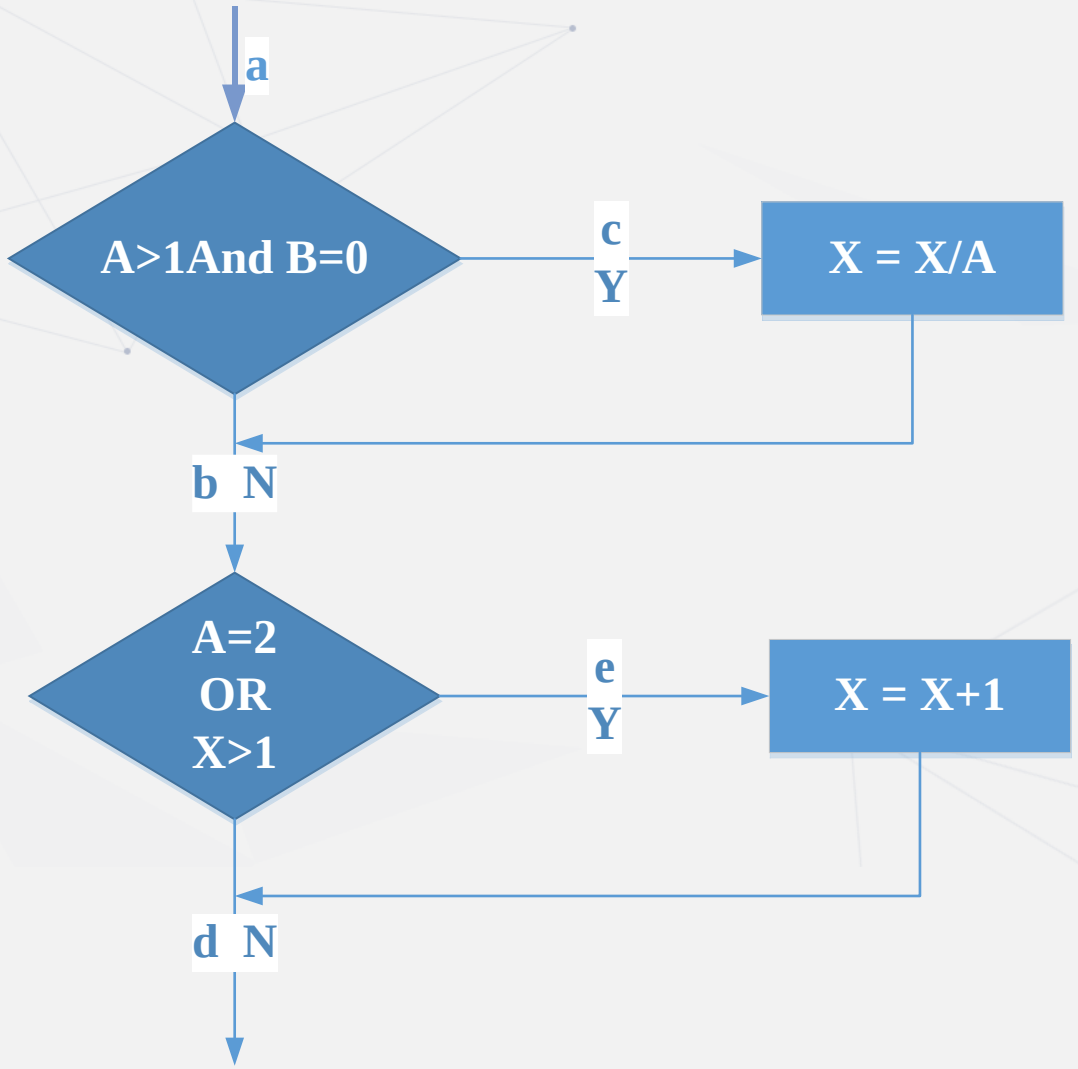
判定覆盖使用

- 用例设计：
- 路径acd
- 输入用例数据：

A= 3 B=0 X=1

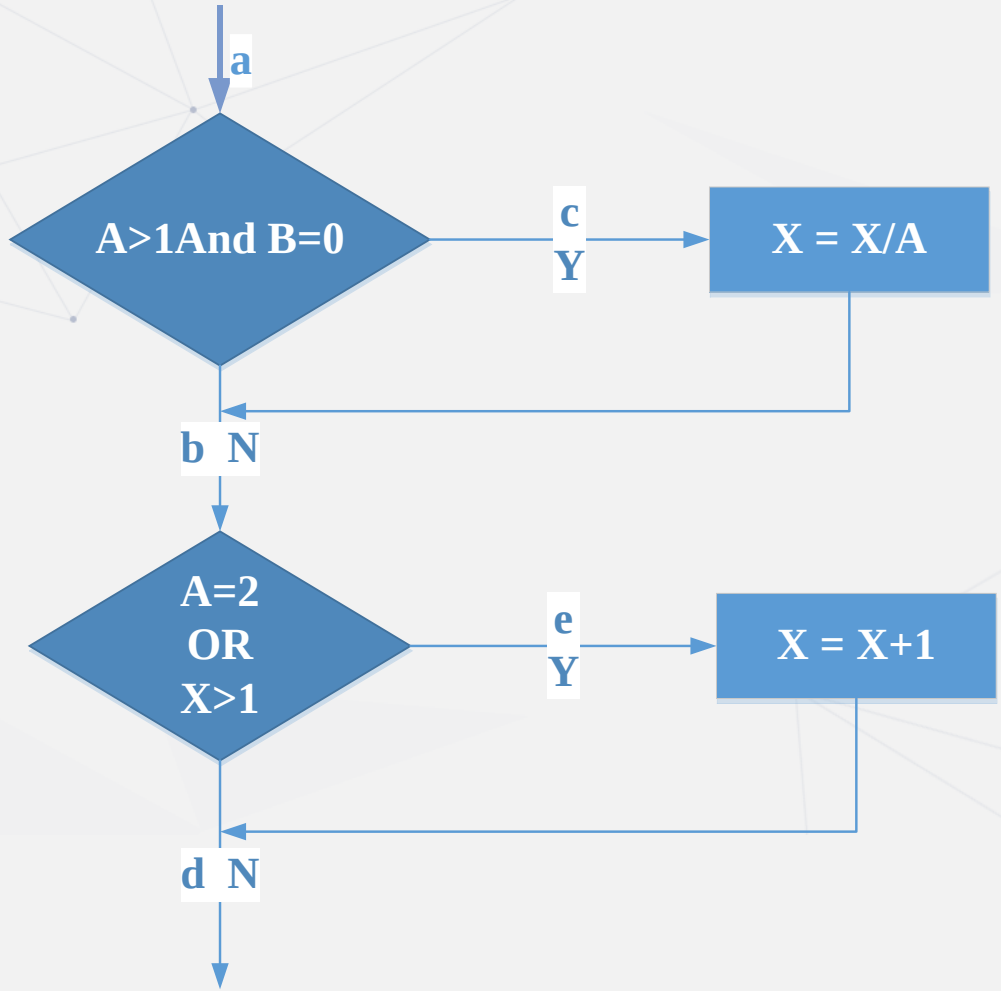
- 路径abe
- 输入用例数据：

A= 2 B=1 X=3



判定覆盖使用

- 用例设计（还可以是）
- 路径abd
- 路径ace

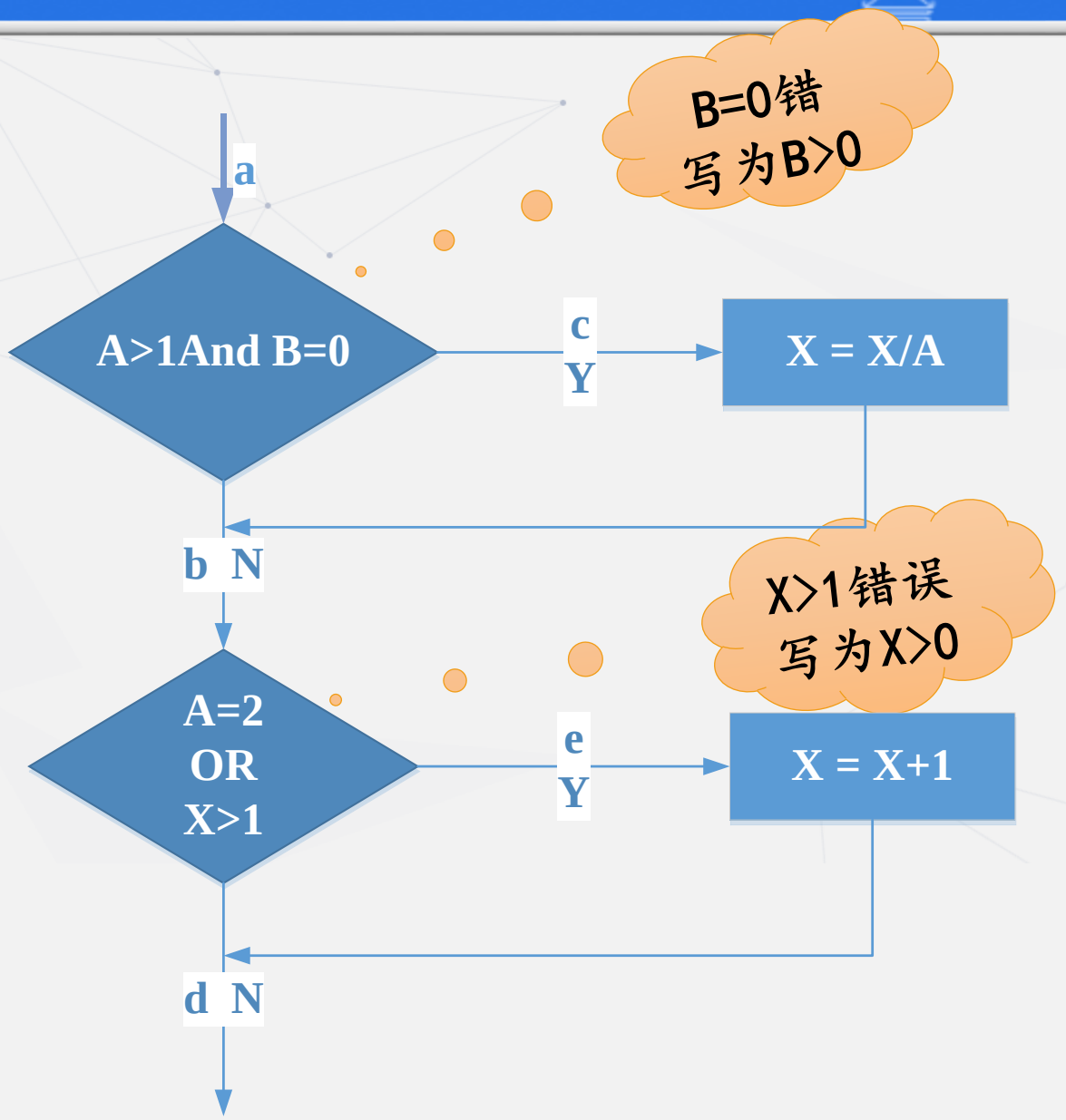


判定覆盖使用

- 覆盖路径：acd和abe

或：abd和ace

- 判定覆盖也不充分



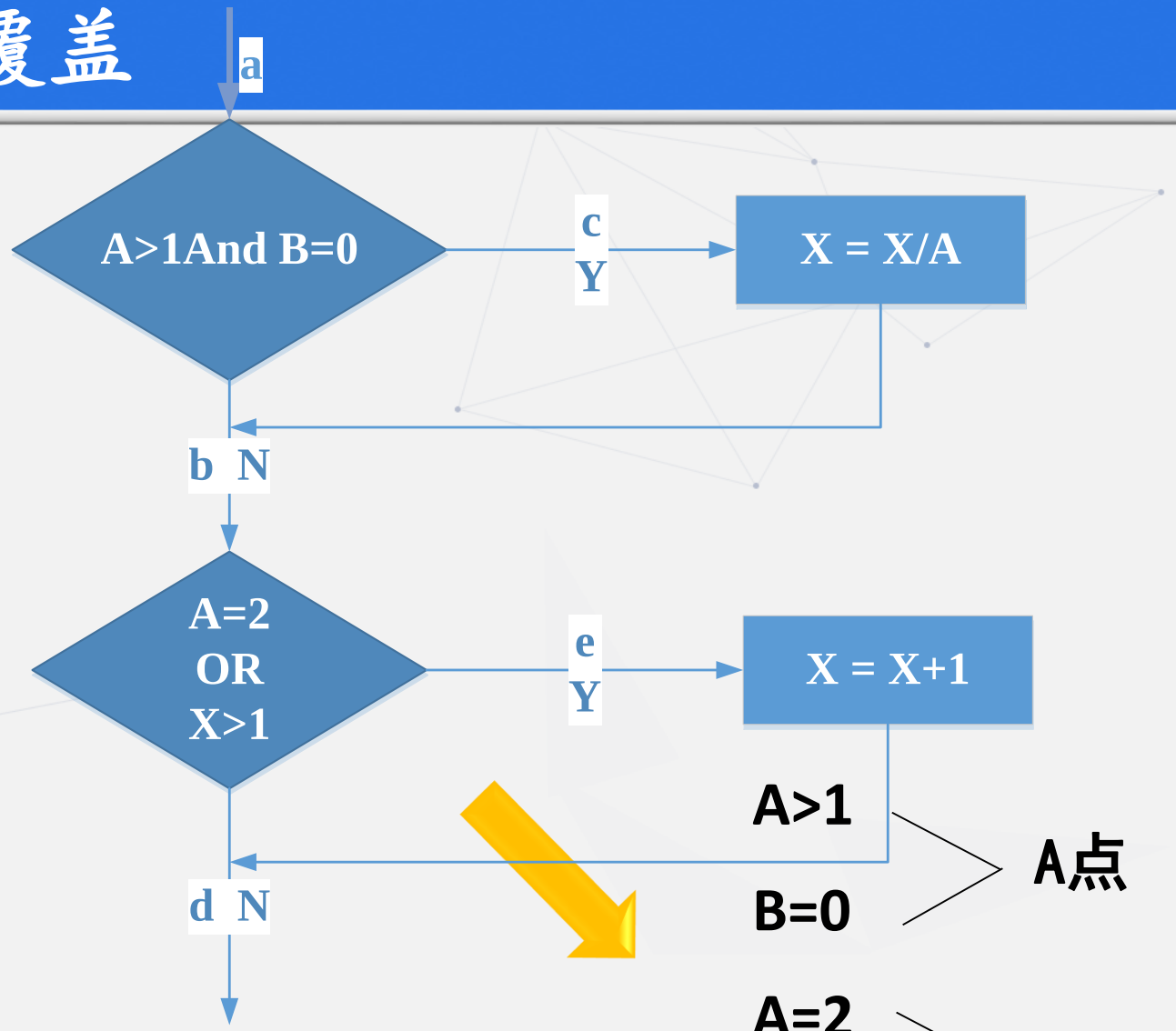
条件覆盖

- 条件覆盖：设计若干测试用例，使得每个判定中每个条件的可能取值至少满足一次。

条件覆盖

```
void main()
{
 float A,B,X;
scanf(“%f%f%f”,&A,&B
 ,&X);
if((A>1)&&(B==0))
    X=X/A;
 if((A==2)||(X>1))
     X=X+1;
  printf(“%f”,X);
}
```

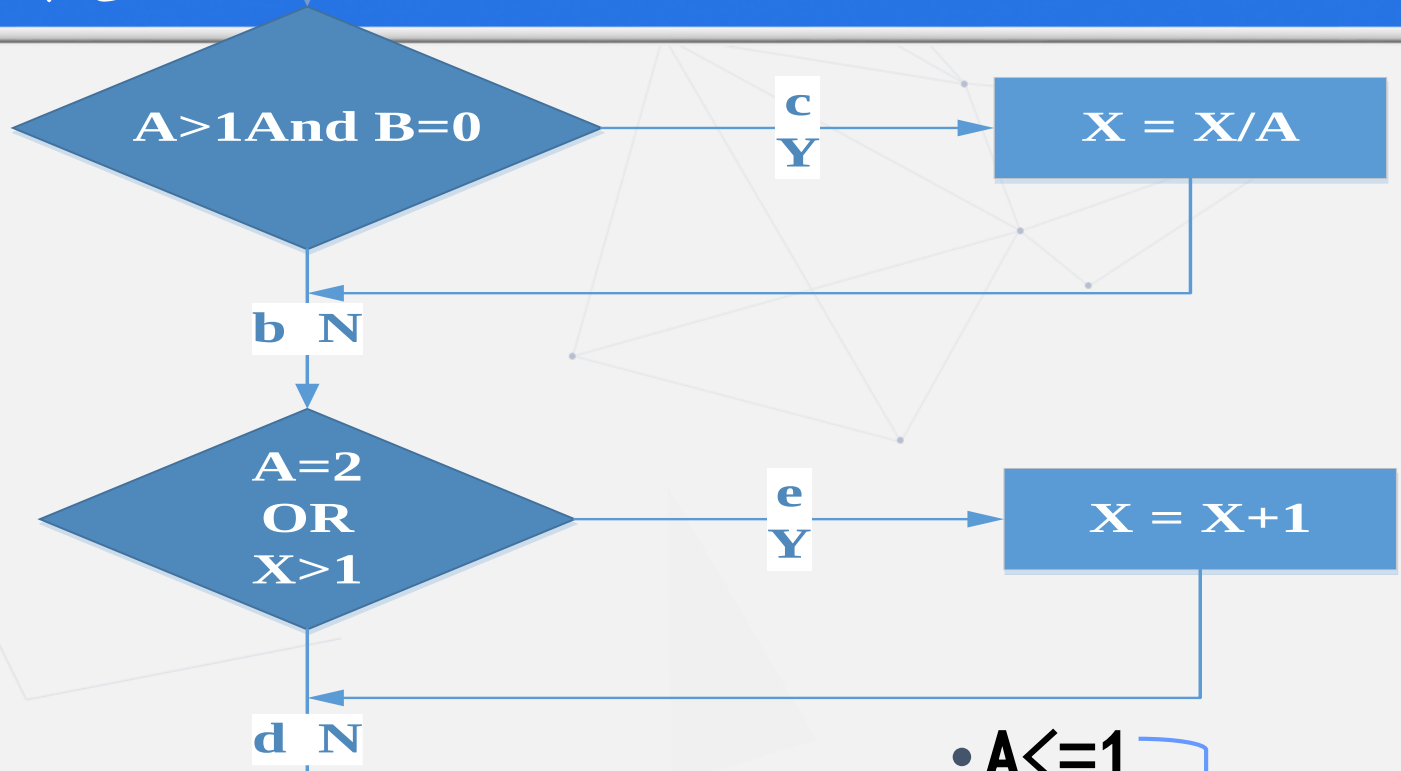
条件覆盖



A>1 T1
A<=1 F1
B=0 T2
B!=0 F2

A=2 T3
A!=2 F3
X>1 T4
X<=1 F4

条件覆盖



- A>1
- B=0
- A=2
- X>1

T

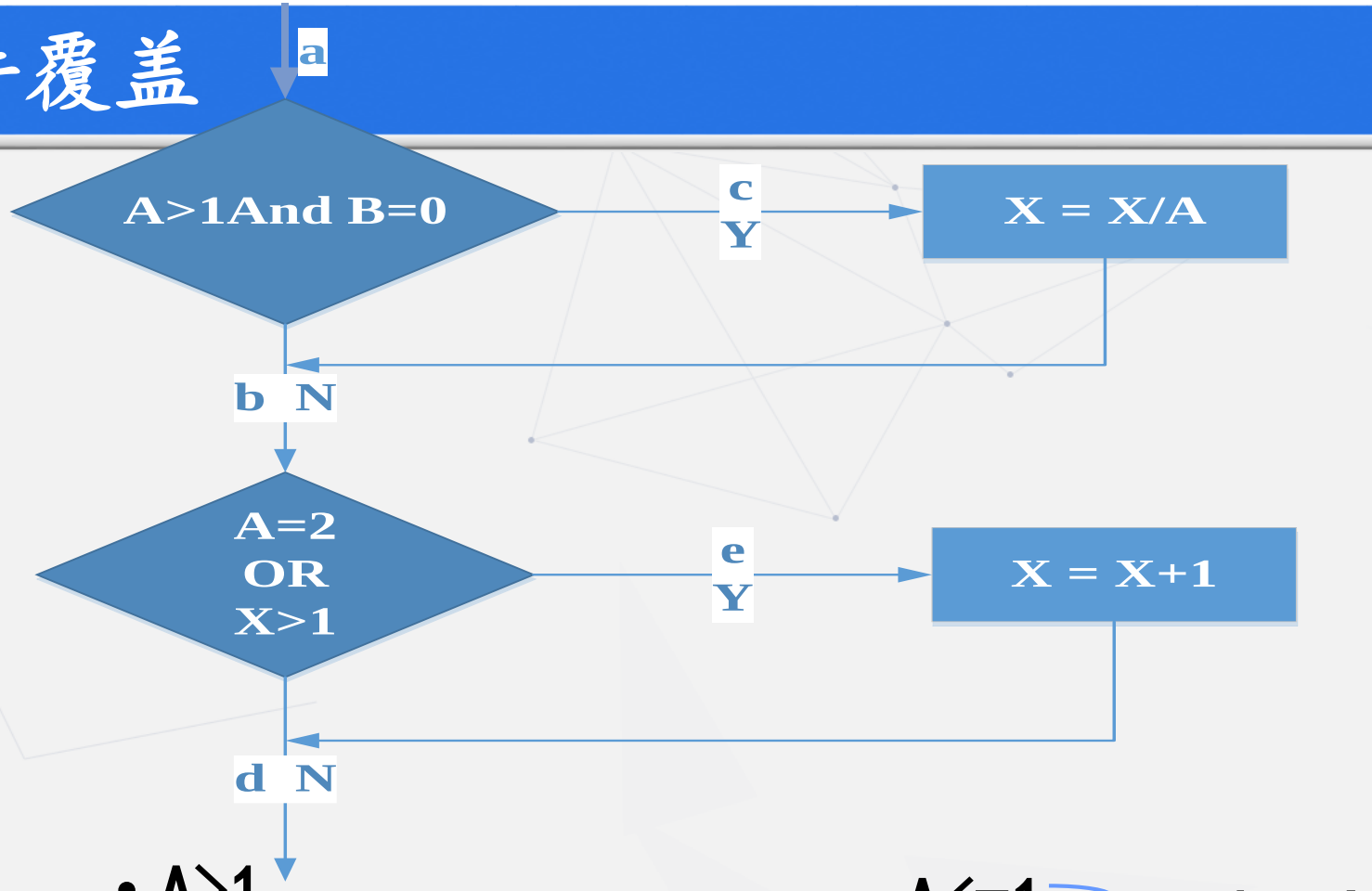
- A=2
- B=0
- X=4
- ace

- A<=1
- B!=0
- A!=2
- X<=1

F

- A=0
- B=2
- X=-1
- abd

条件覆盖



- A>1
- B!=0
- A=2
- X<=1

- A=2
- B=6
- X=0
- ace

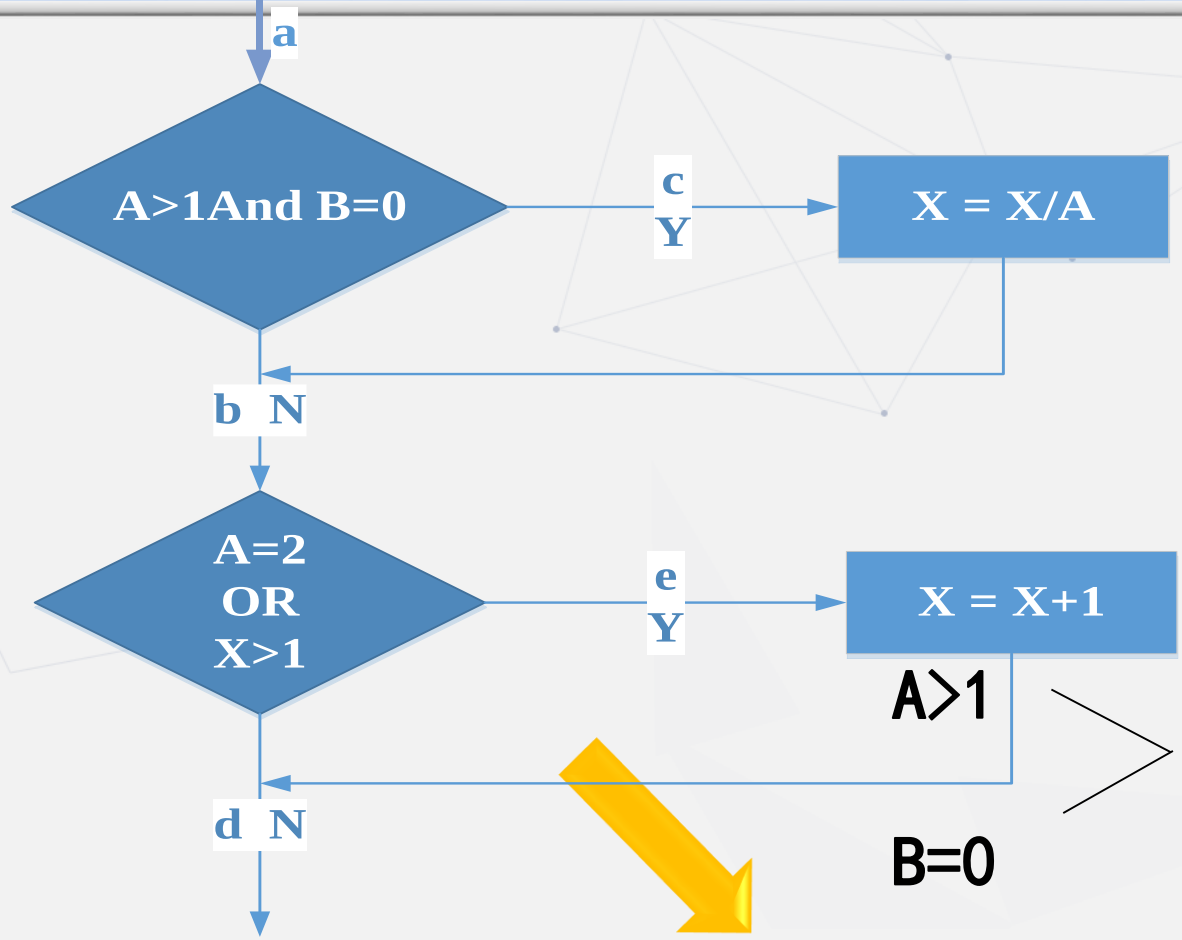
- A<=1
- B=0
- A!=2
- X>1

- A=-6
- B=0
- X=2
- abd

条件判定覆盖定义

- 条件判定覆盖：设计若干测试用例，使得判定中所有条件可能至少执行一次取值，同时，使得所有判定的可能至少执行一次。

条件判定覆盖分析



A>1

B=0

P1

A=2

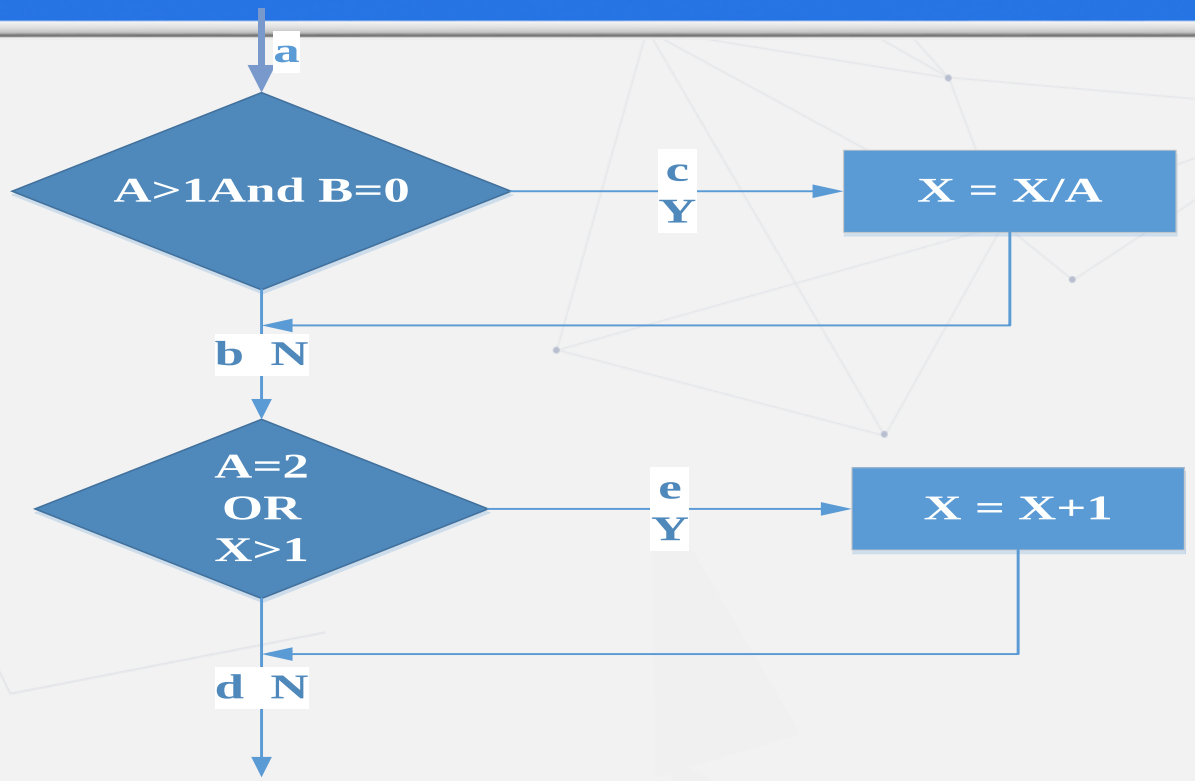
X>1

P2

A>1 T1
A<=1 F1
B=0 T2
B=!0 F2

A=2 T3
A!=2 F3
X>1 T4
X<=1 F4

条件判定覆盖分析使用



- A>1
- B=0
- A=2
- X>1

T

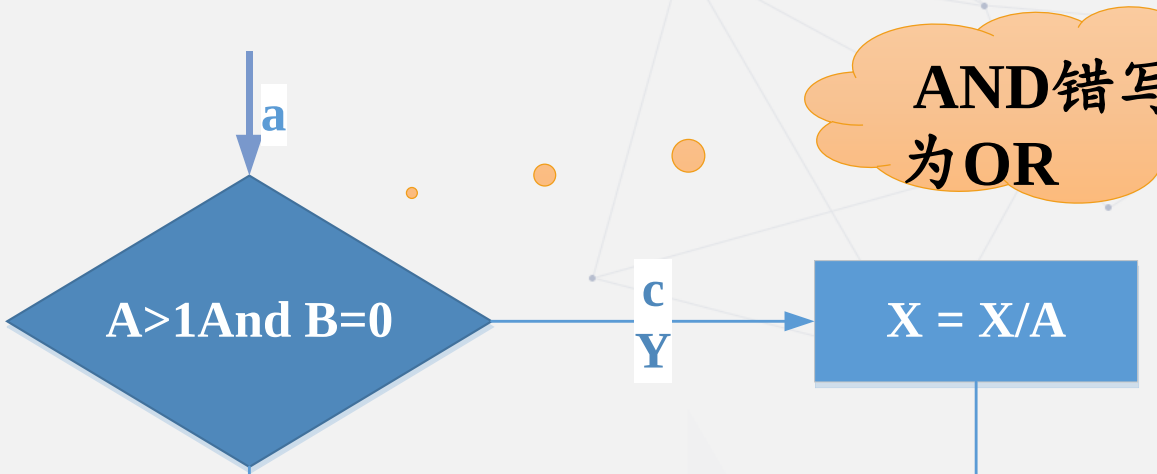
- A=2
- B=0
- X=4

- A<=1
- B!=0
- A!=2
- X<=1

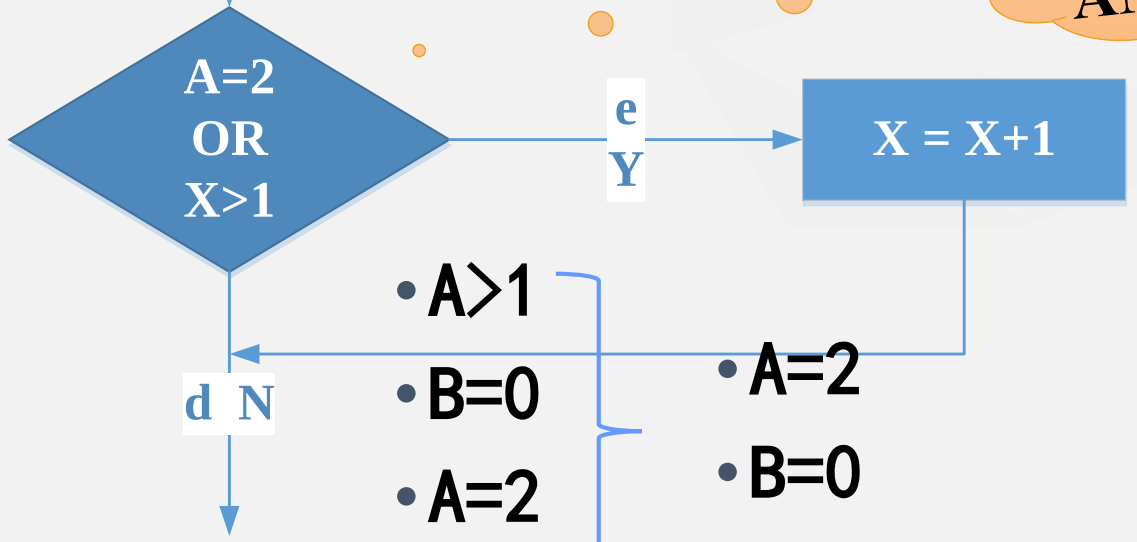
F

- A=0
- B=2
- X=-1

条件判定覆盖使用分析



AND错写为OR



OR错写为AND

- A>1
- B=0
- A=2
- X>1

- A=2
- B=0
- X=4

- A<=1
- B!=0
- A!=2
- X<=1

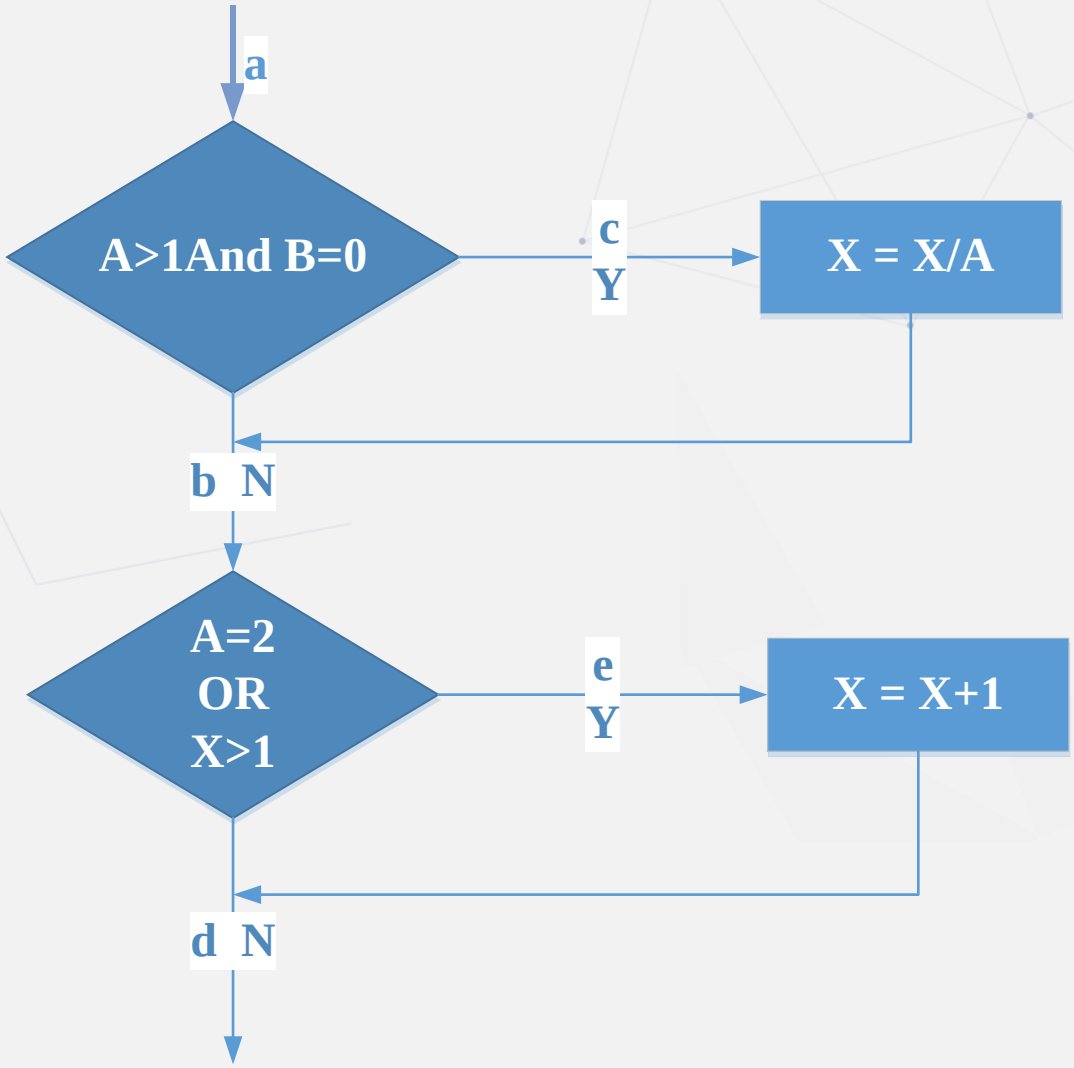
- A=0
- B=2
- X=-1

条件判定覆盖并不完美

条件组合覆盖定义

- 条件组合覆盖：设计若干测试用例，使得判定中条件的各种组合都至少执行一次。

条件组合覆盖使用分析



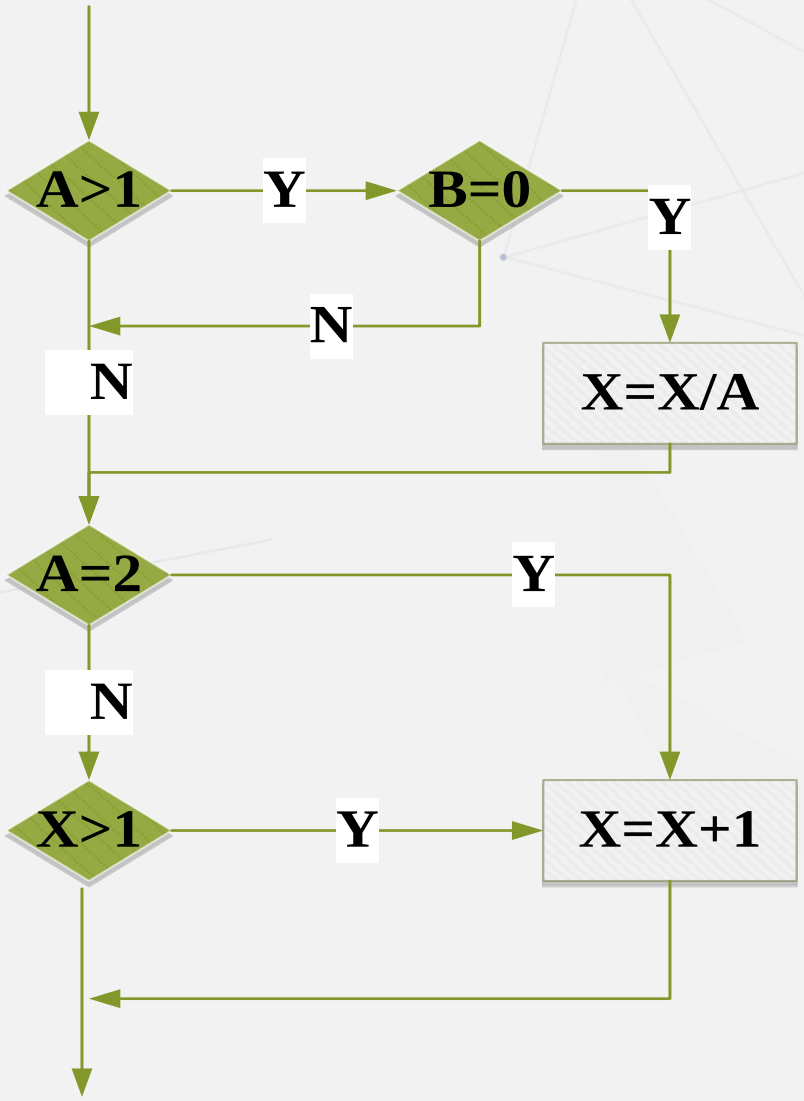
a点

A>1	B=0	(1)
A>1	B!=0	(2)
A<=1	B=0	(3)
A<=1	B!=0	(4)

b点

A=2	X>1	(5)
A=2	X<=1	(6)
A!=2	X>1	(7)
A!=2	X<=1	(8)

条件组合覆盖使用分析



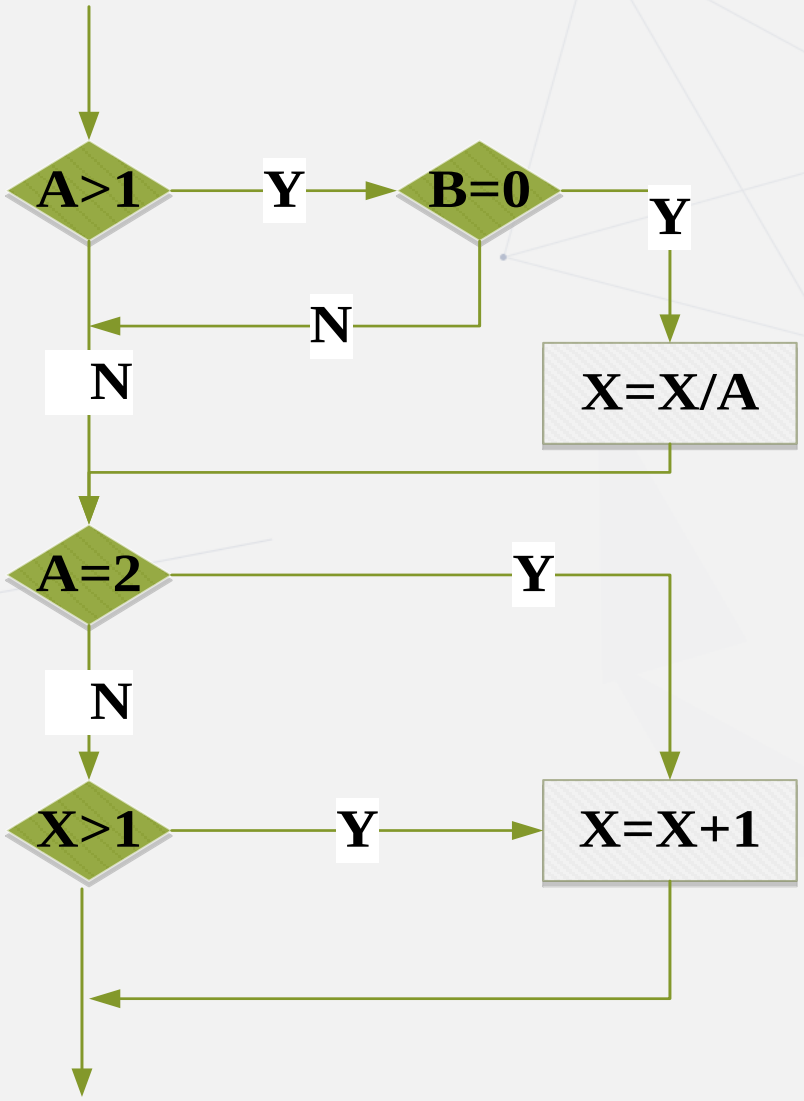
A>1 B=0 (1)
A>1 B=0 (2)
A<=1 B=0 (3)
A<=1 B=0 (4)

A=2 X>1 (5)
A=2 X<=1 (6)
A=2 X>1 (7)
A=2 X<=1 (8)

(1，5) (a，c，e)

输入数据：A=2 B=0 X=4

条件组合覆盖使用分析



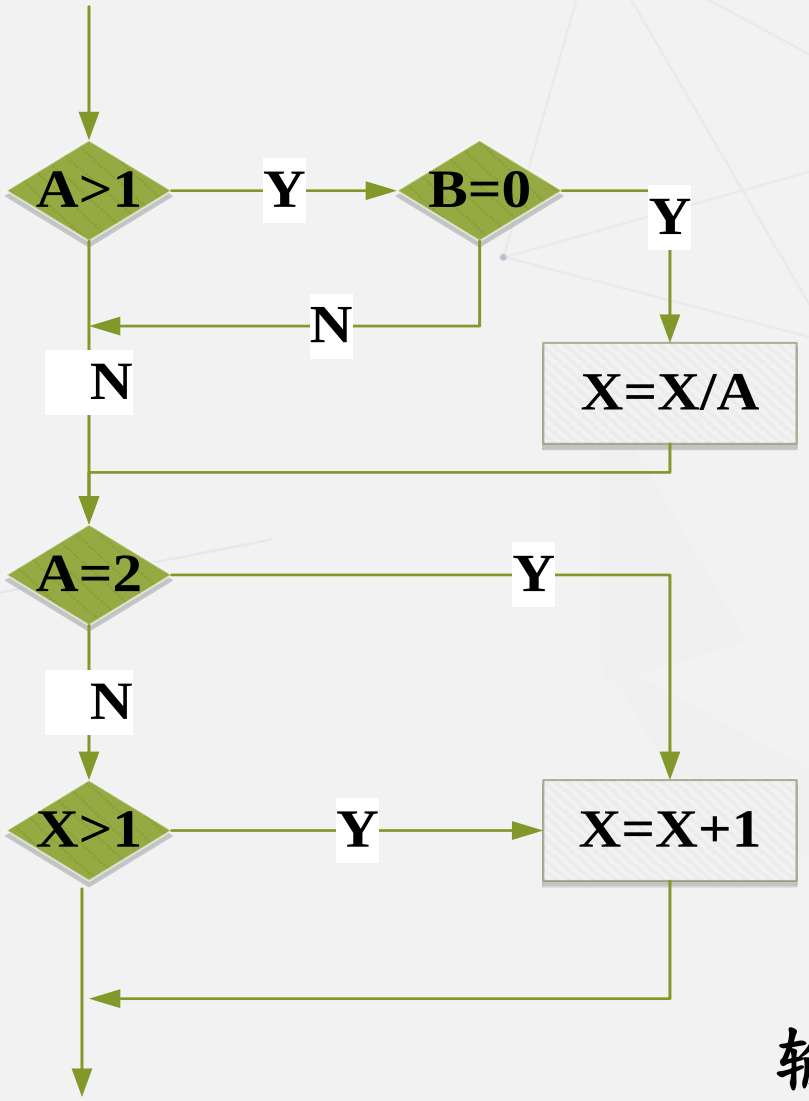
A>1 B=0 (1)
A>1 B=0 (2)
A<=1 B=0 (3)
A<=1 B=0 (4)

A=2 X>1 (5)
A=2 X<=1 (6)
A=2 X>1 (7)
A=2 X<=1 (8)

(2，6) (a，b，e)

输入数据：A=2 B=1 X=1

条件组合覆盖使用分析



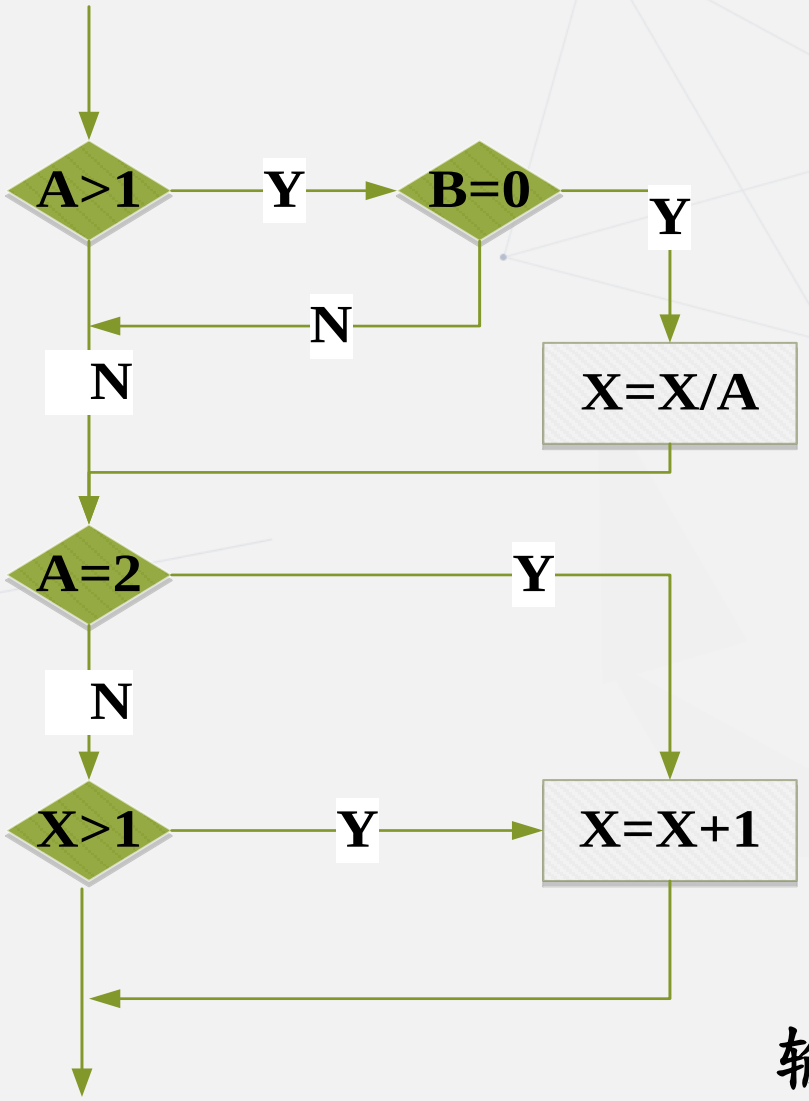
A>1 B=0 (1)
A>1 B=0 (2)
A<=1 B=0 (3)
A<=1 B=0 (4)

A=2 X>1 (5)
A=2 X<=1 (6)
A=2 X>1 (7)
A=2 X<=1 (8)

(3，7) (a，b，e)

输入数据：A=1 B=0 X=2

条件组合覆盖使用分析



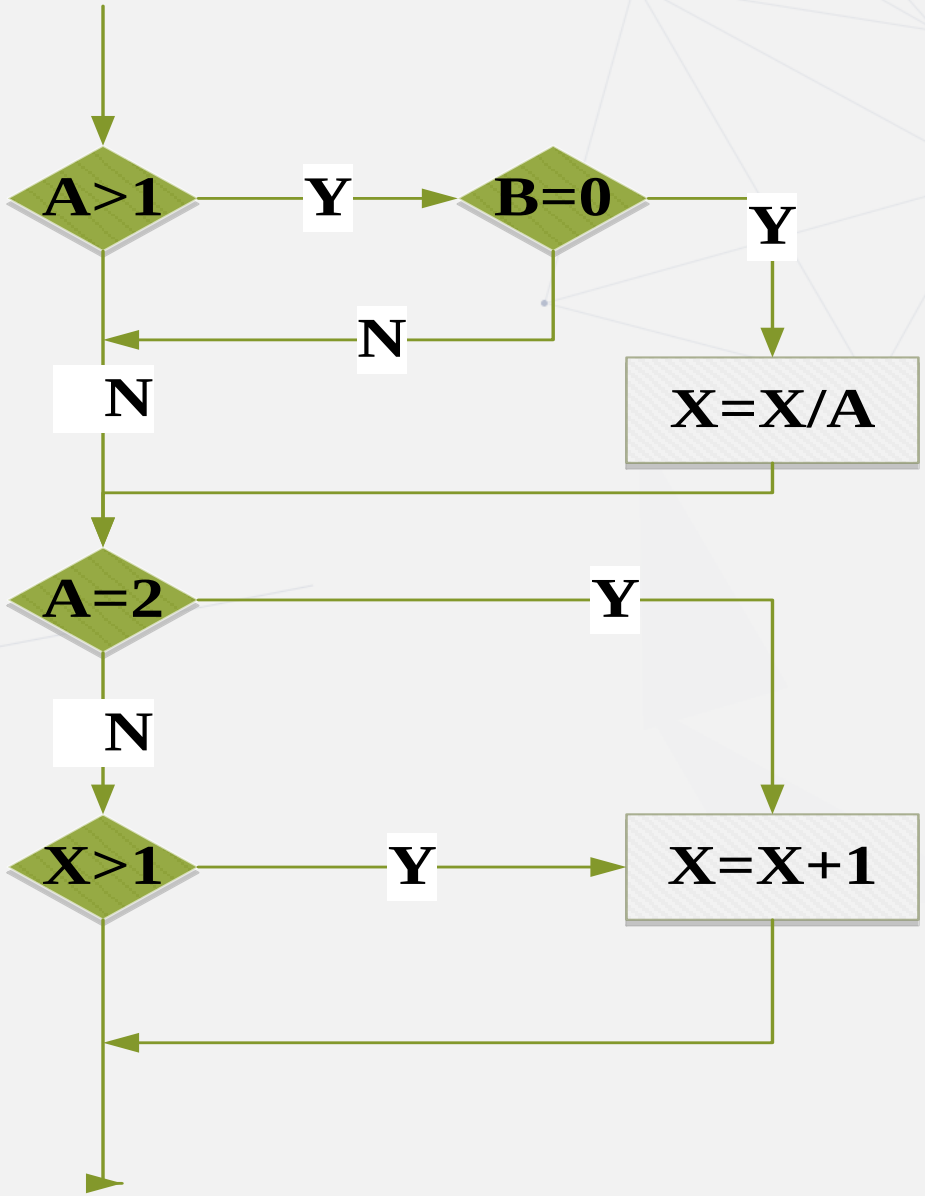
A>1 B=0 (1)
A>1 B=0 (2)
A<=1 B=0 (3)
A<=1 B=0 (4)

A=2 X>1 (5)
A=2 X<=1 (6)
A=2 X>1 (7)
A=2 X<=1 (8)

(4, 8) (a, b, d)

输入数据：A=1 B=1 X=1

条件组合覆盖使用分析



A>1 B=0 (1)
A>1 B=0 (2)
A<=1 B=0 (3)
A<=1 B=0 (4)

A=2 X>1 (5)
A=2 X<=1 (6)
A=2 X>1 (7)
A=2 X<=1 (8)

其他：……

测试用例优化

- 尽量选取边界测试数据
  - 如a > 1,可以选择a = 1 ,这样测试用例覆盖到边界
- 尽量避免“与”“或”关系的屏蔽现象
  - 如与关系表达式，若要满足判定结果为假，只要任一条件为假即可
  - 如（a>1）AND （b<2）
  - 选择a = 1,b =2 优于选择a = 1,b = 1

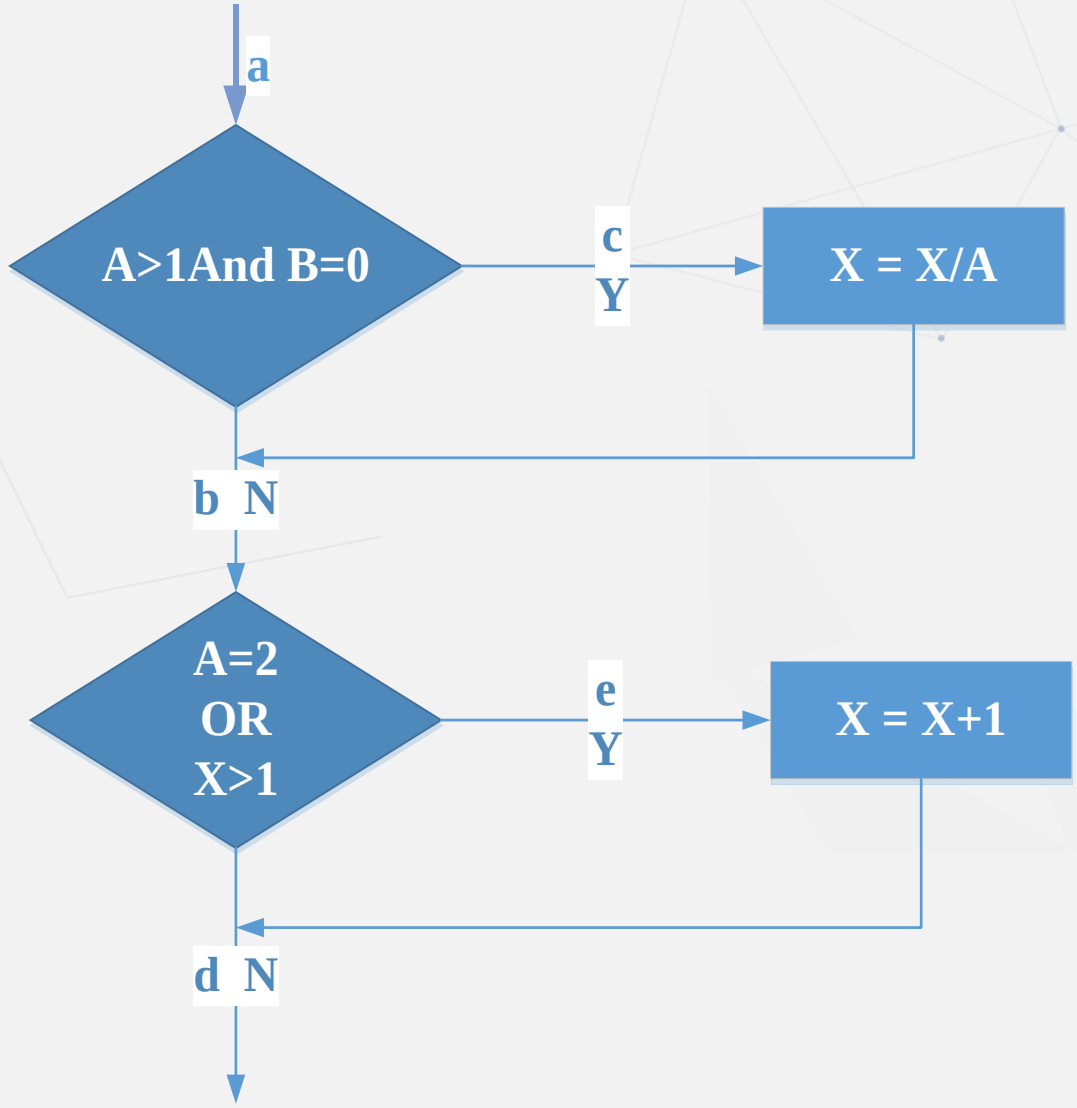
实践练习

- 计算nextdate的函数，进行判定、条件覆盖测试，并分析其中的利弊

路径覆盖

- 路径覆盖：相当强的覆盖标准，设计足够多的测试用例，覆盖程序中所有可能的路径

路径覆盖使用



- 路径分析

—abd,ace,acd,abe

输入数据：

abd:A=1,B=1,X=1

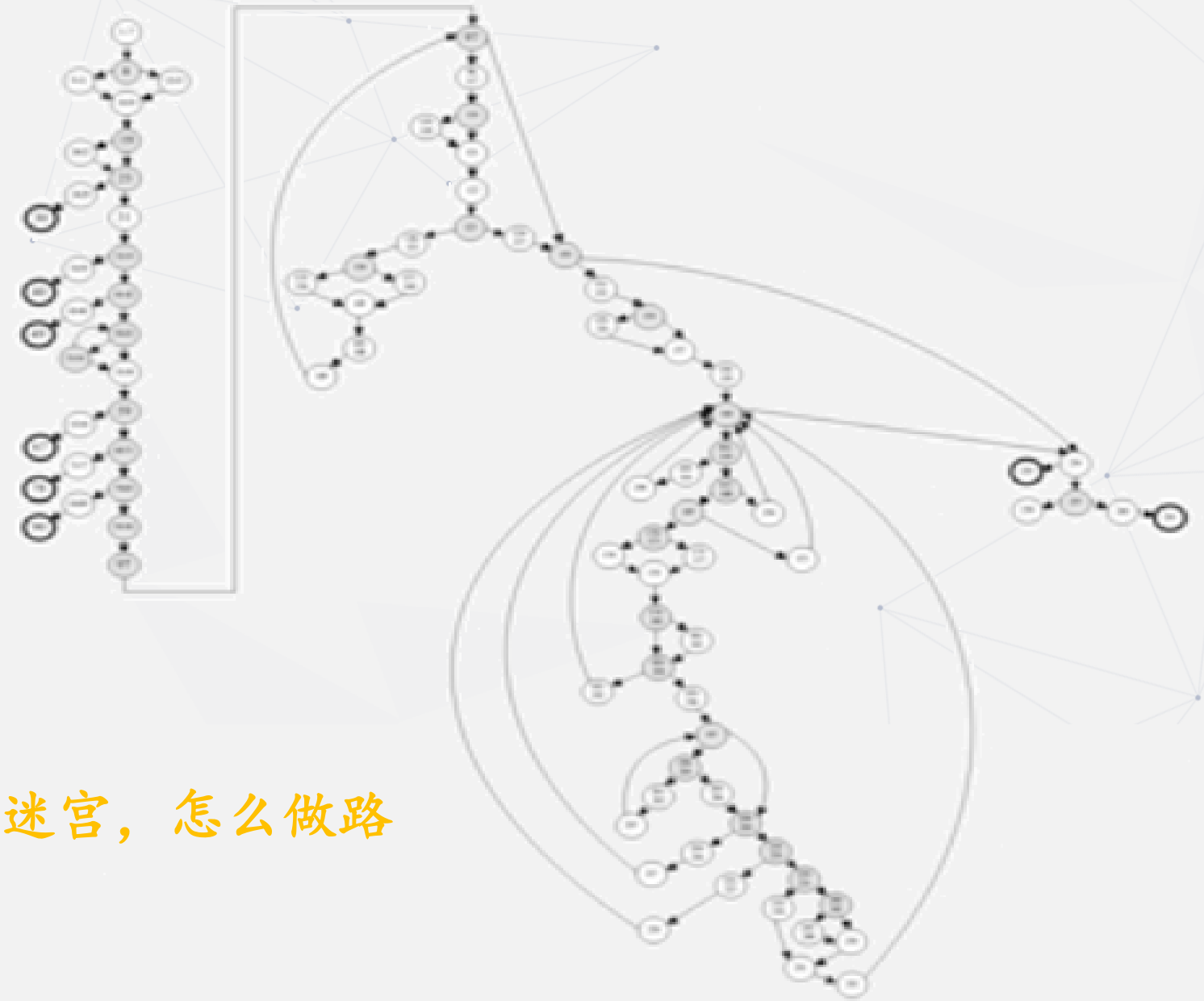
ace:A=2,B=0,X=4

abe:A=1,B=1,X=2

acd:A=3,B=0,X=1

路径测试

- 当程序中存在多个判定和循环时，它们之间形成串联、嵌套等多种形式，使路径基本不可穷尽



- 有些路径像迷宫，怎么做路径测试呢？

路径测试

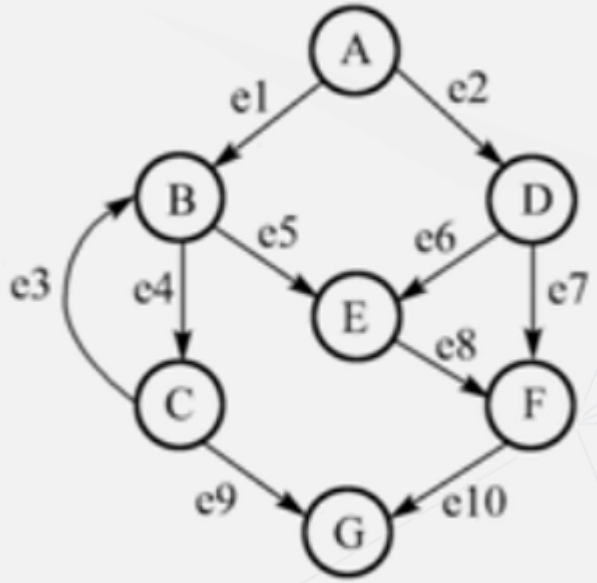
1 画出程序图

2 计算环复杂度

3 设计路径测试用例

路径测试

- 程序图：压缩后的程序流程图，也是一种特殊形式的有向图
- 画程序图的压缩原则：
  - 剔除注释语句
  - 剔除所有数据变量声明语句
  - 所有连续的串行语句压缩为一个节点
  - 所有循环次数压缩为一次循环：无论某个循环结构将循环多少次，仅考虑执行循环体和不执行循环体这两种情况



路径测试

- 计算环复杂度
  - 直观观察法
  - 公式计算法
  - 判定节点法

路径测试

- 计算环复杂度（1）直观观察法：
- 如右图，观察程序图中将二维平面分割为封闭区域

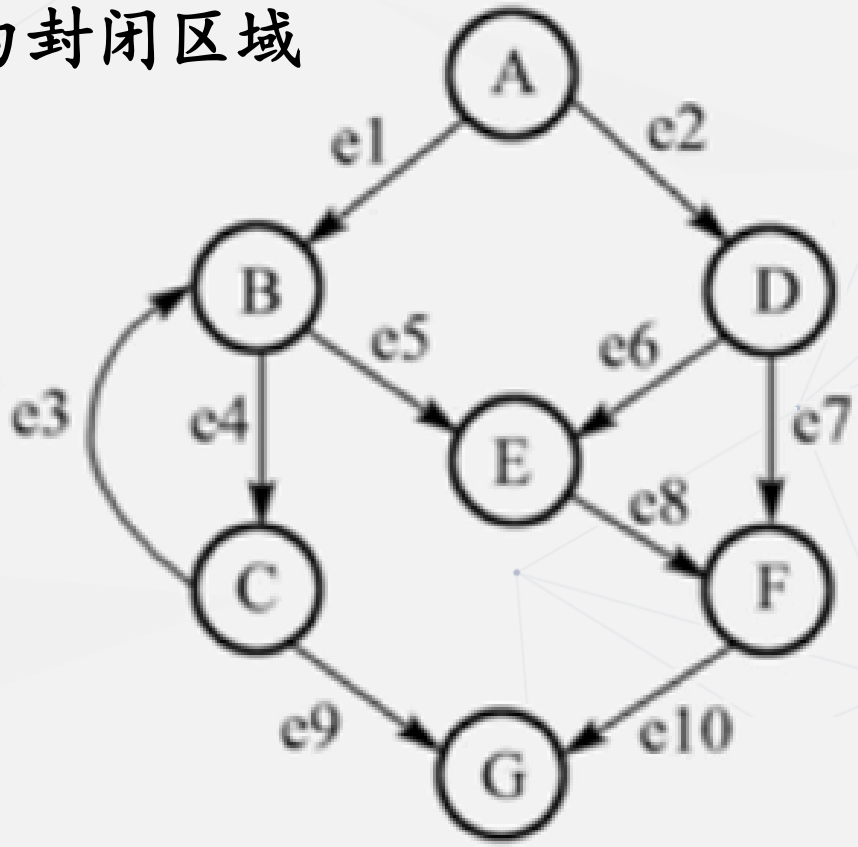
和开放区域的个数

区域1：节点A、B、E、D所围成

区域2：节点B、C所围成

区域3：节点B、C、 G 、F 、E所围成

区域4：节点B、E、F所围成



另有一个外部的开放区域，得到程序图的环复杂度为5

路径测试

- 计算环复杂度（2）公式法：
- V(G) = e–n+1 (e表示边的数目，n表示节点的数目)
- 其他书上也有V(G) = e–n+2
- 使用此公式应满足的两个前提条件：

1 程序图中不包含孤立节点

2 程序图必须是一个强连通图，即对于程序图的任意两个节点I、J，在该节点对之间至少能找到一条路径能从I执行到J，且同时至少找到一条路径从J到I

路径测试

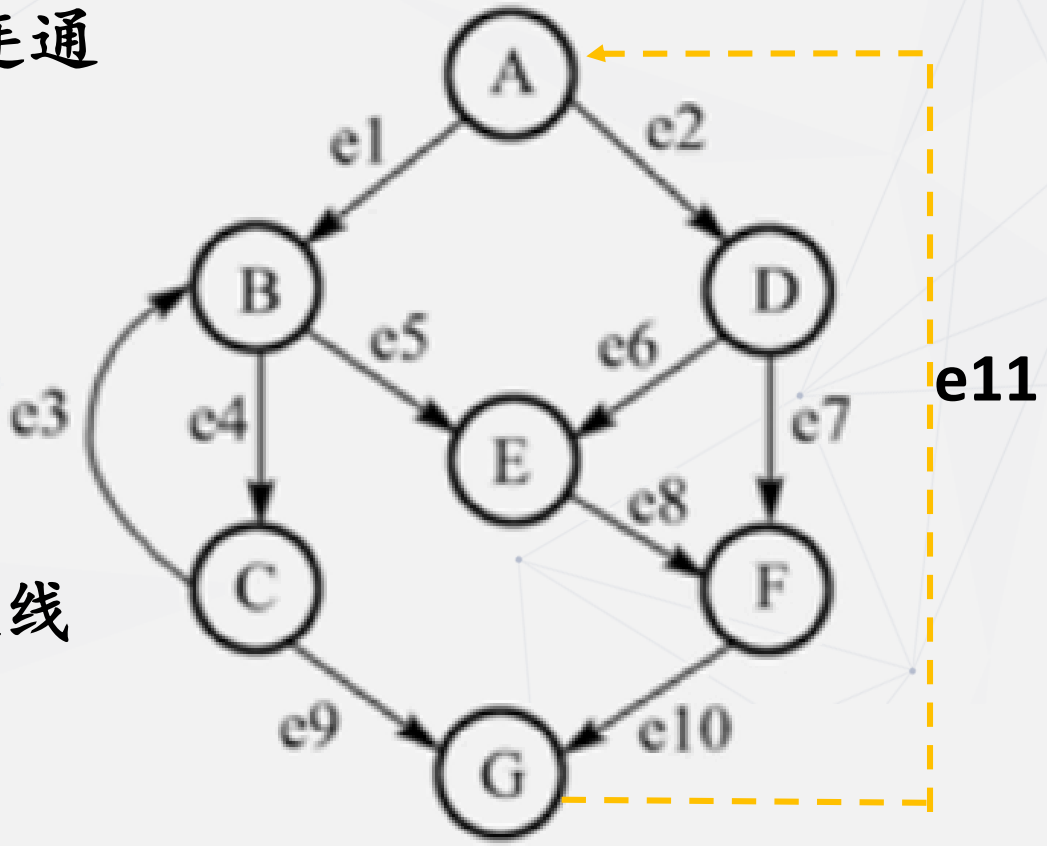
- 计算环复杂度（2）公式法：
  - 需要了解单向连通、双向联通、无连通
  - 非强连通图怎样使用公式计算呢？

——程序图改造

怎样改造？

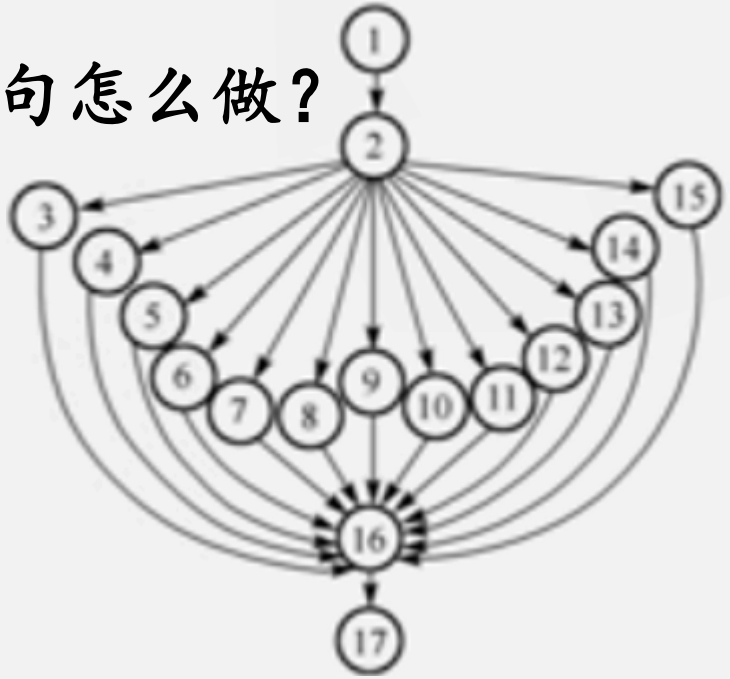
增加虚拟未节点到起始节点的连通线

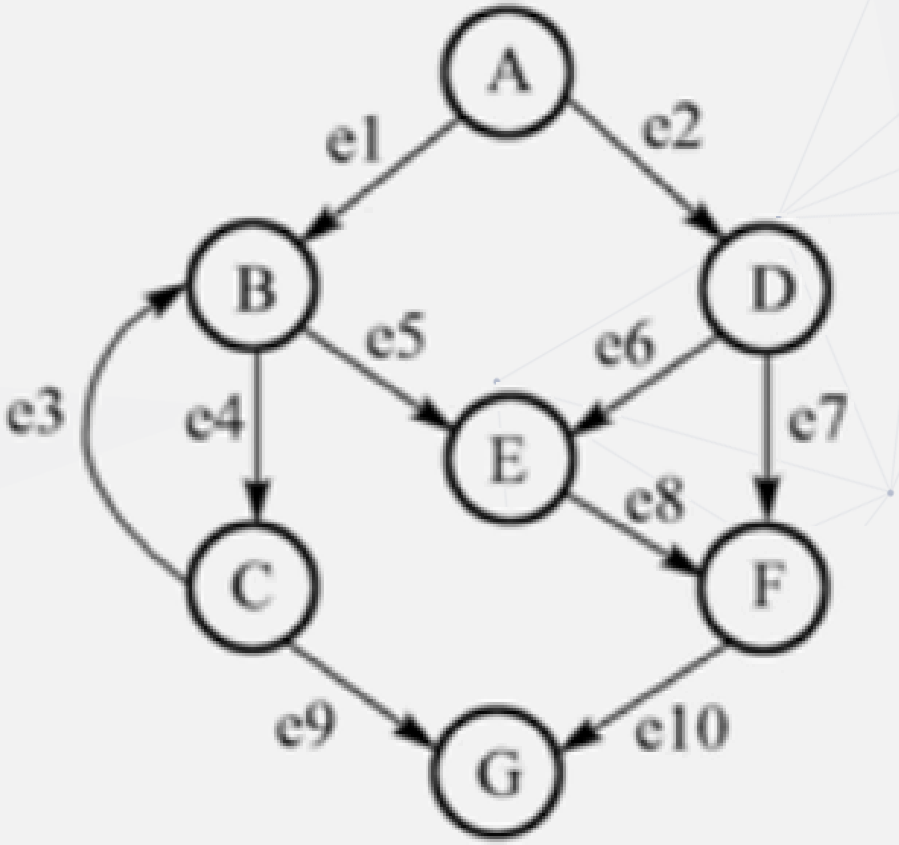
- V(G)=11-7+1 =5



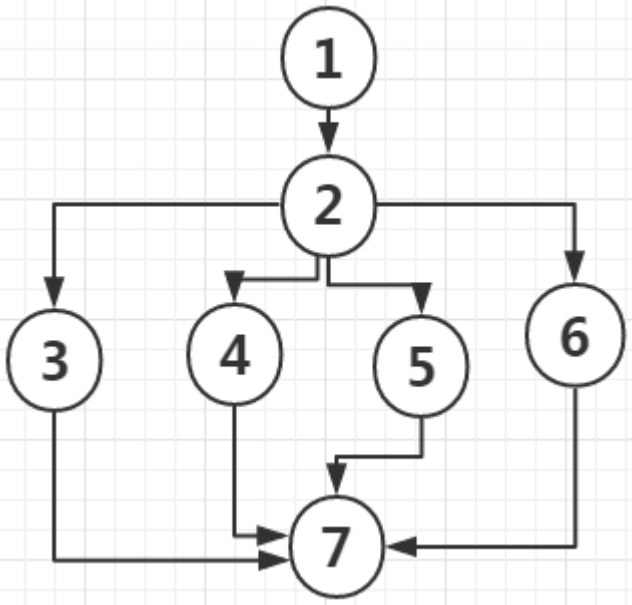
路径测试

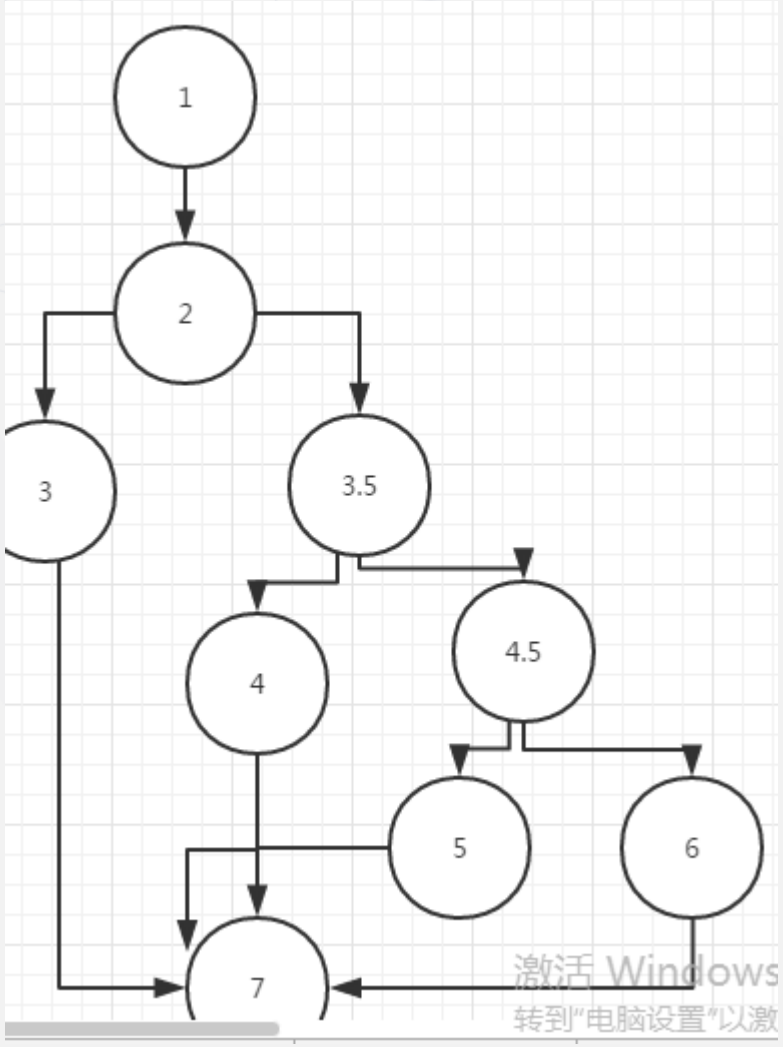
- 计算环复杂度（3）判定节点法：
  - 利用代码中度量判定节点的数目来计算环复杂度
  - V(G)=P+ 1（p代表判定节点的数目）
  - 通常情况判定节点非常容易识别
  - 遇到switch语句怎么做？
  -





路径测试





路径测试

- 设计路径测试用例要求：
  - 测试的完备性，通过对独立路径的测试达到对所有路径的测试覆盖
  - 测试的无冗余性，每条路径都是独立的
  - 每条路径代表的是一种对判定决策的新的访问方式

路径测试

- 对路径的测试，核心和难点：

  1 如何确定独立路径集合的规模

  2 如何从整个路径集合中抽取独立路径的集合，以确保路径的独立性和独立路径集合的完备性

  3 如何保证每条独立路径的可行性

  4 如何从独立路径设计测试用例

路径测试

- 抽取独立路径：

1 确定主路径：在所有路径中找到一条最复杂的路径作为主路径，复杂体现在：

1）包含尽可能多的判定节点（包含条件判定和循环判定）

2）包含尽可能复杂的判定表达式

3）对应尽可能高的执行概率

4）包含尽可能多的执行语句

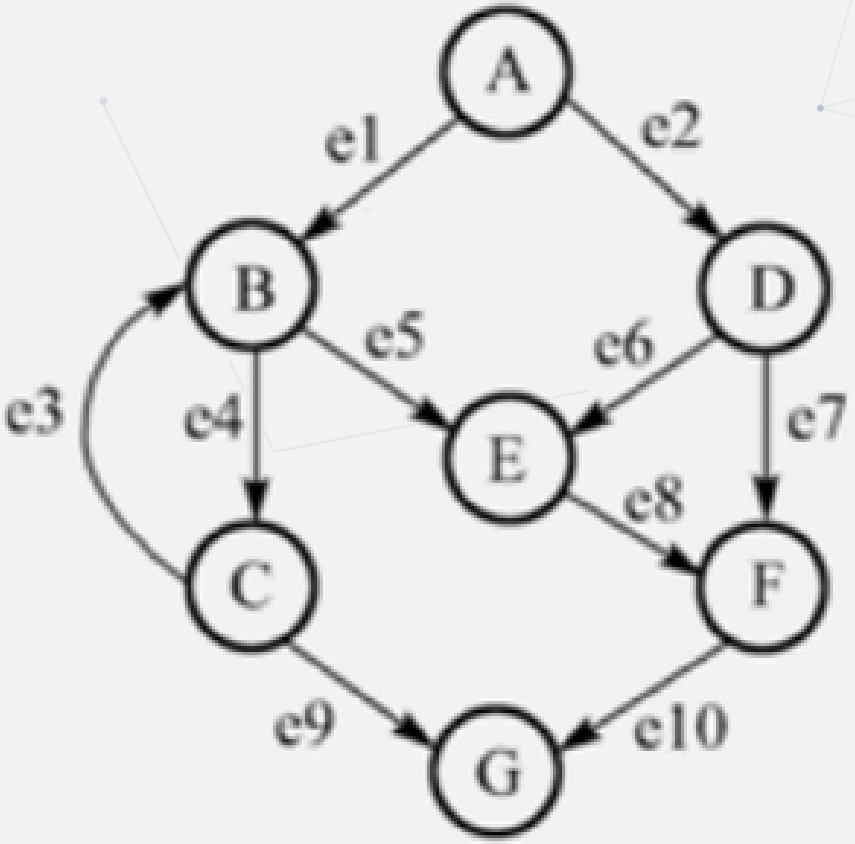
2 基于主路径抽取其他独立路径

1）基于主路径依次在该路径的判定节点处执行一个新的分支

3 独立路径规模：等于程序图的环复杂度

路径测试

- 举例：



- Path1:A,B,C,G（经过判定节点A,B,C）
- Path2: A,D,E,F,G
- Path3:A,B,E,F,G
- Path4:A,B,C,B,C,G
- Path5:A,D,F,G

路径测试

- 设计测试用例步骤：

1 根据程序源代码生成程序图

2 计算程序图的环复杂度，确定独立路径集合的大小

3 以最复杂的路径为主路径（基础路径），并在此基础上通过覆盖所有判定分支确定其他路径

4 剔除不可行路径，必要时补充其他重要路径

5 根据得到的独立路径集合设计对应的测试用例

路径测试举例

- 针对nextDate函数计算环复杂度，并抽取独立路径，最终转化成测试用例
- 代码见文本文件

路径测试举例

- 独立路径提取：
  - Path1:A,6,8,12,13,16,18,20,21,B,34,35
  - Path2:A,6,7,16,18,20,21,B,34,35
  - Path3:A,6,8,9,16,18,20,21,B,34,35
  - Path4:A,6,8,12,15,16,18,20,21,B,34,35
  - Path5:A,6,8,12,13,16,18,33,34,35
  - Path6:A,6,8,12,13,16,18,20,21,28,34,35

路径测试举例

- 不可行路径分析：
  - 对照源代码，比较发现当程序执行判定节点6的e3分支时，意味着函数输入2、4、6、9或11月，此时程序不可能执行到e16分支上，因此path1,path3,path4是不可行路径
  - 导致不可行路径的主要原因：多个判定表达式中涉及的简单判定条件存在一定约束关系，如存在e16分支时，必然存在e2分支
  - 不可行路径可以作为修改代码的建议，这也就是进行白盒测试的意义，在抽取独立路径时，格外注意，确保都是可行路径

路径测试

- 路径测试的修改：
- 独立路径提取：
  - Path1:A,6,8,12,13,16,18,20,21,28,34,35
  - Path2:A,6,7,16,18,20,21,B,34,35
  - Path3:A,6,8,9,16,18,20,21,28,34,35
  - Path4:A,6,8,12,13,16,18,20,21,28,34,35
  - Path5:A,6,8,12,13,16,18,33,34,35

路径测试

- 路径的补充：
  - 补充执行概率较高的路径，确保用户最常执行的路径无缺陷
  - 补充涉及复杂算法的路径，确保对复杂算法的处理正确无误
  - 可以根据路径的执行概率来确定补充路径
    - 例如：满足e2执行概率 ： 7/12(全年有7个月含31天)
    - e3执行概率:5/12
    - e13的执行概率 12/365
    - 路径的执行概率=所有边的概率乘积
  - 如果函数的输入包含无效数据，则执行概率需将这些无效情况考虑进来

路径测试

- nextDate 函数补充路径的执行概率

序号	独立路径	访问的判定分支	执行概率	备注
1	Path6	e2, e14	56. 38%	测试31天月份的普通日期
2	Path7	e3, e5, e14	32. 26%	测试30天月份的普通日期
3	Path8	e3, e6, e9, e14	6. 05%	测试非闰年的2月份的普通日期

路径测试

- 根据设计路径转成测试用例

序号	输入	预期输出	备注
001	2012-12-31	2013-1-1	Path1
002	2012-7-31	2012-8-1	Path2
003	2012-6-30	2012-7-1	Path3
004	2011-2-28	2011-3-1	Path4
005	2012-2-15	2012-2-16	Path5
006	2012-7-15	2012-7-16	Path6
007	2012-6-15	2012-6-16	Path7
008	2011-2-15	2011-2-16	Path8

内容总结

- 逻辑覆盖
  - 语句，判定，条件，判定条件，条件组合
  - 路径覆盖
    - 程序图
    - 计算圈复杂度，确定路径数量
    - 根据路径测试规则，设计测试路径（含分析不可行和补充必要路径）
    - 最终生成测试用例

Question