Chapter 3

Fundamental Theory and Methods

3.1 Non-execution based Verification

3.2 Execution based Verification

3.3 Formal verification

3.4 Other methods

3.1 Non-execution based Verification

- Not execute the program, tester could use some tools to review and analyze the specification and program code.

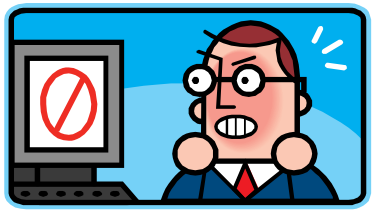
- walk through
- Inspection

3.1.1 Walk through

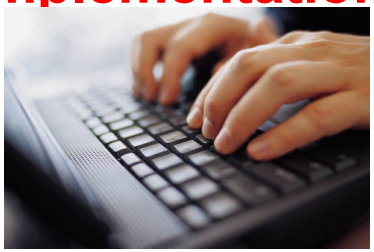
- Consist a walk through test group
- Check program code logic
- Generate test case, include input data and expect result.
- Put the data into program code, if the calculate result unequal the expect result, find an error.
- Check the code line by line.

a member of the SQA

a representative of next step



one or more representative of implementation



a client representative

The walkthrough team should consist of four to six individuals

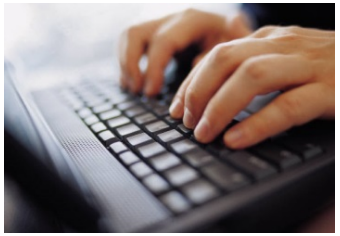
- The goal of the walkthrough team is to detect faults, not to correct them.
- The person leading the walkthrough guides the other members of the team through the code.
- The walkthrough can be driven by the lists of issues compiled by team members or by the code itself, with team members raising their concerns at the appropriate time.
- In both cases, each issue will be discussed as it comes up and resolved into either a fault that needs to be addressed or a point of confusion that will be cleared up in the discussion

3.1.2 Inspection

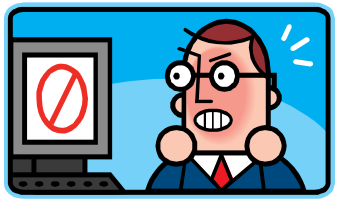
- An inspection is a far more formal activity than a code walkthrough.
- It is conducted by a team of three to six people

a moderator

one or more representative of implementation



a representative of next step



a recorder

The review process consists of five formal steps:

- In the overview step, the author of the module gives a presentation to the team.
- In the preparation step, the participants try to understand the code in detail and compile lists of issues, ranked in order of severity(严重性). They are aided in this process by a checklist of potential faults for which to be on the lookout.
- In the inspection step, a thorough walkthrough of the code is performed, aiming for fault detection through complete coverage of the code. Within a day of the inspection, the moderator produces a meticulous（小心翼翼） written report.
- In the rework step, the individual responsible for the code resolves all faults and issues noted in the written report. （**Individual Reviewer Issues Spreadsheet**）
- In the follow-up step, the moderator must make sure that each issue has been resolved by either fixing the code or clarifying confusing points. （**Review Report Spreadsheet**）

Individual Reviewer Issues Spreadsheet

<Work product information>					
<Reviewer Name>					
<Preparation Date>					
<Reviewer Preparation Time>					
	Location		*Checklist*		
Issue #	*Page*	*Line*	*Item*	*Severity*	*Description*
1	1	18	Typo	1	Change "accound" to "account"

Review Report Spreadsheet

<Work Product Information>						
Review Team Members		*Preparation Time*				
Leader						
Recorder						
Reviewer						
Reviewer						
Reviewer						
Reviewer						
	Total prep time					
Meeting date						
<Review Recommendation>						
		Location		Checklist		
Issue #	*Reviewer*	*Page*	*Line*	*Item*	*Severity*	Description
1		1	18	Typo	1	Change "accound" to "account"

- An important product of an inspection is the number and kinds of faults found rated by severity.（Here is a sample outline of a review report）
- If a module comes through an inspection exhibiting a significantly larger number of faults than other modules in the system, it is a good candidate for rewriting.
- If the inspection of two or three modules reveals a large number of errors of specific types, this may warrant (re)checking other modules for similar errors.
- If more than 5 percent of the material inspected must be reworked, the team must reconvene for a full re-inspection.

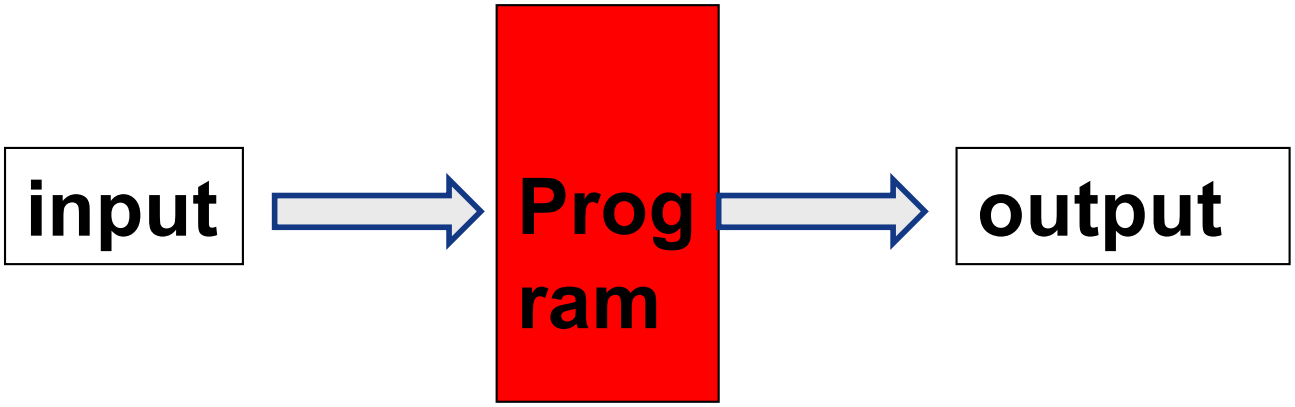
3.2 Execution-based Verification

- Use test cases to execute the program, get all results from the execution.

  There are two basic approaches to testing modules, each with its own weaknesses.

  - **Testing to Specifications, also known as black-box test.**
  - **Testing to Code, also called glass-box, white-box test.**

3.2.1 Black-box test



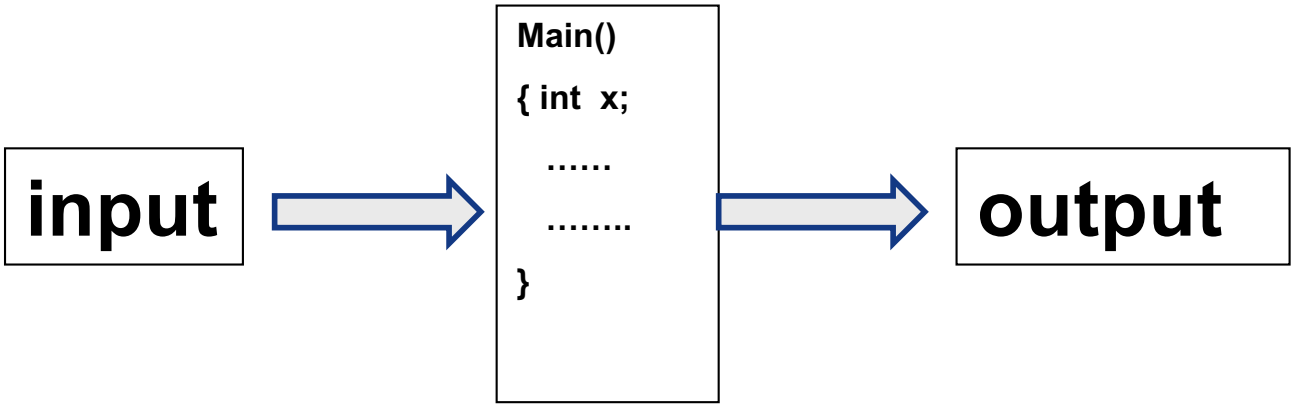
- Seeks to verify that the module conforms to the specified input and output while ignoring the actual code
- It is rarely possible to test all modules for all possible input cases, which may be a huge number of modules

- Equivalence testing **is a technique based on the idea that the input specifications give ranges of values for which the software product should work the same way.**
- Boundary value analysis **seeks to test the product with input values that lie on and just to the side of boundaries between equivalence classes.**

For example the input is (a,b), there are at least 5 equivalence classes:

functional testing, the tester identifies each item of functionality or each function implemented in the module and uses data to test each function.

3.2.2 White-box test

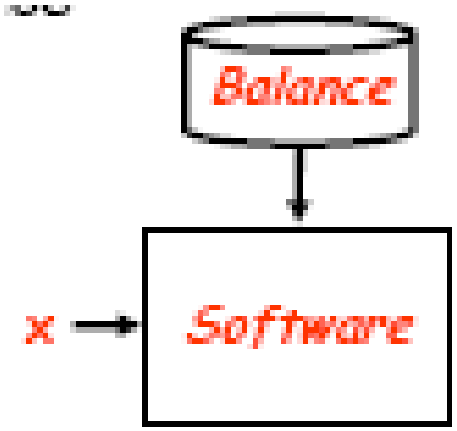


- Also called logic-driven, or path-oriented testing.
- testing each path through the code is generally not feasible, even for simple flowcharts.
- Moreover, it is possible to test every path without finding existing faults, because, for example, the fault lies in the decision criterion for selecting between paths.

- statement coverage **and amounts to running tests in which every statement in the code is executed at least once.**
- branch coverage**, makes sure that each branching point is tested at least once.**
- path coverage**, make sure that every different path is tested at least once.**

Example : black-box testing

- Discussion: MAC/ATM machine
  - Specs
- **Cannot withdraw more than $300**
- **Cannot withdraw more than your account balance**



Test case?

X

1. >300
2. 0~300
3. <0

Balance

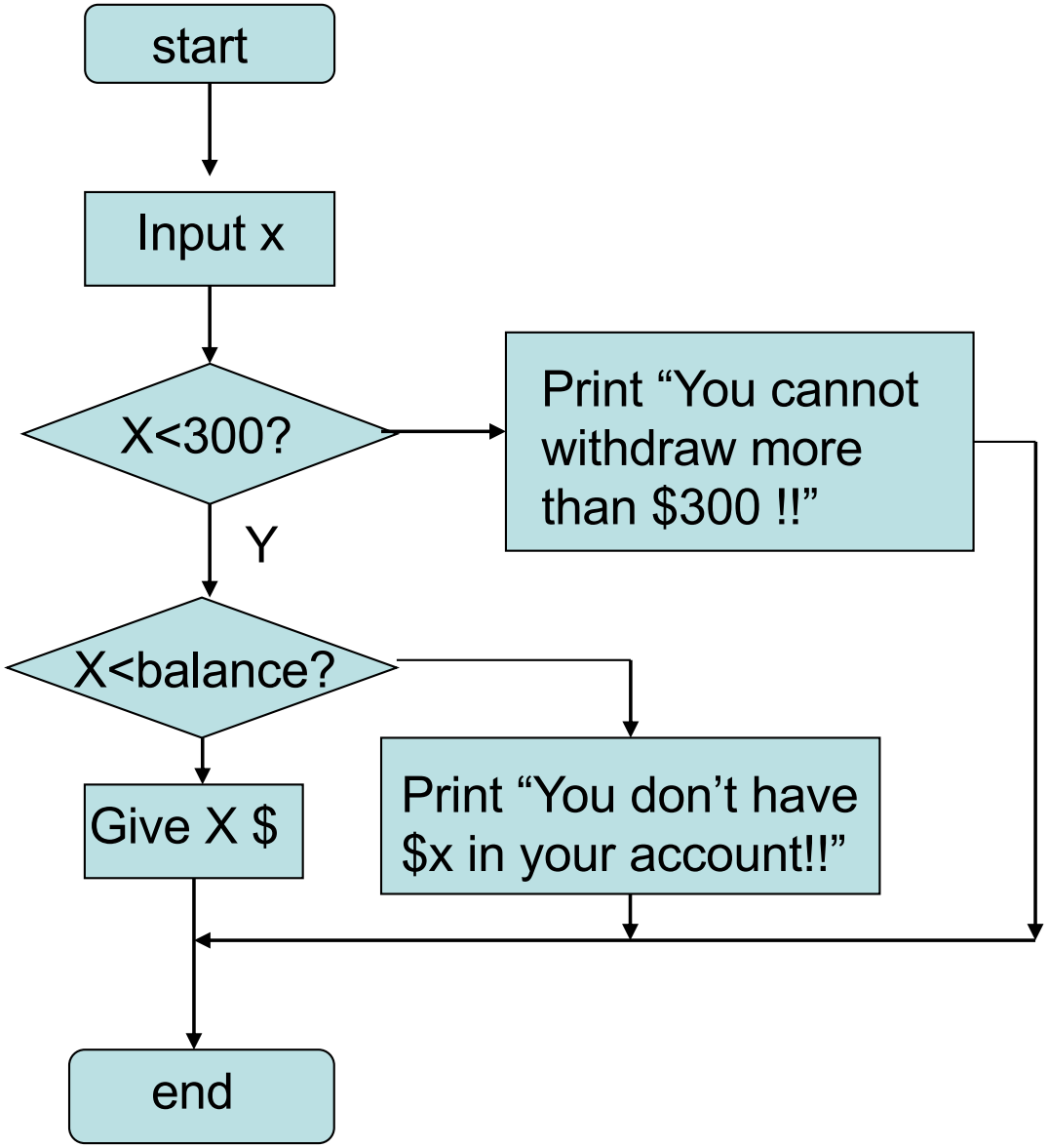
1. = >300,
2. 0~300

White-box Testing

- Example

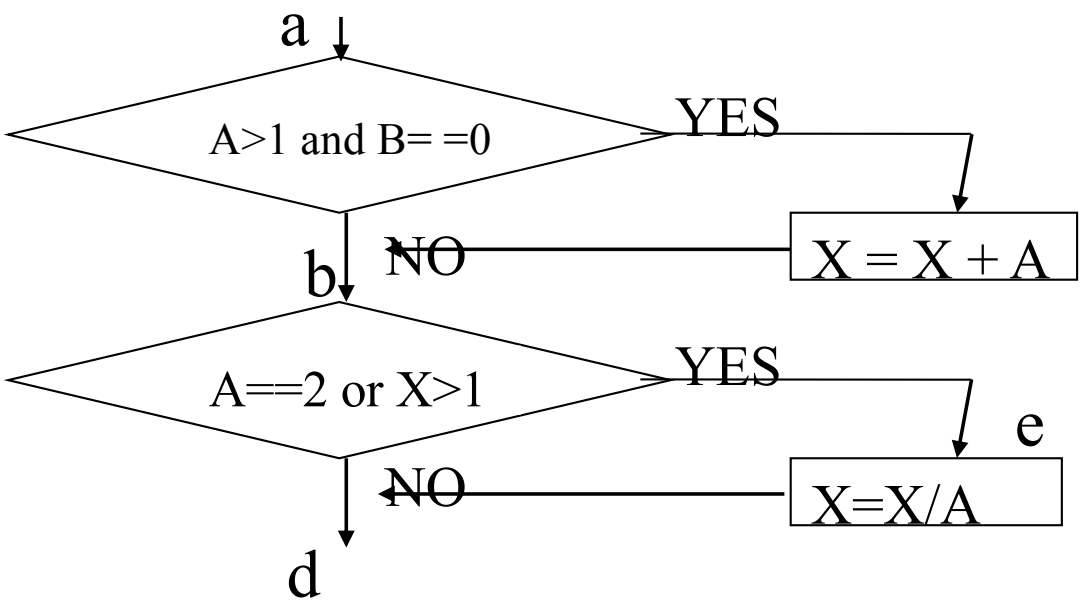
x: 1..1000;

```
1   INPUT-FROM-USER(x);
      If (x <= 300) {
2        INPUT-FROM-FILE(BALANCE);
           If (x <= BALANCE)
3               GiveMoney x;
4          else Print “You don’t have $x in your account!!”}
       else
5      Print “You cannot withdraw more than $300”;
6    Eject Card;
```


start
Input x
X<300?
Print “You cannot withdraw more than $300 !!”
Y
X<balance?
Print “You don’t have $x in your account!!”
Give X $
end

Test case?

How about example2 ?



For example2，branch coverage is: point A, should have A>1，A≤1，B=0，B≠0，point B, should have A=2，A≠2，X>1，X≤1。

so {2，0|2，5} and {0，1|0，0} is enough.

How about path coverage?

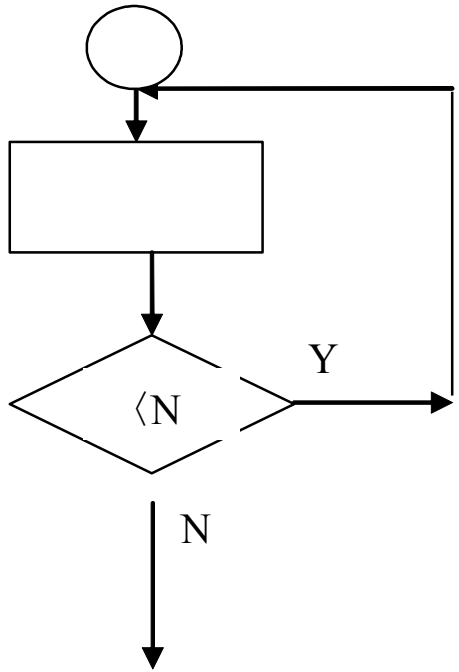
4

Predicates Test case are: 2x2x2x2=16

- How about loop ?
- Branch coverage is two,
- but path coverage is N+1!

Does coverage guarantee absence of faults?

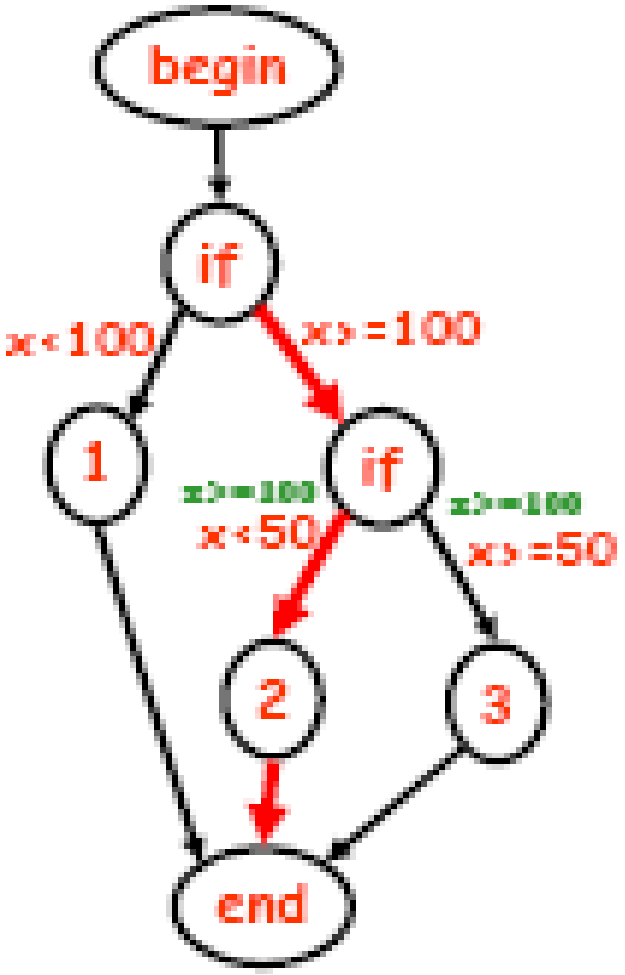
- Can we always get 100% coverage?



Surprise Quiz

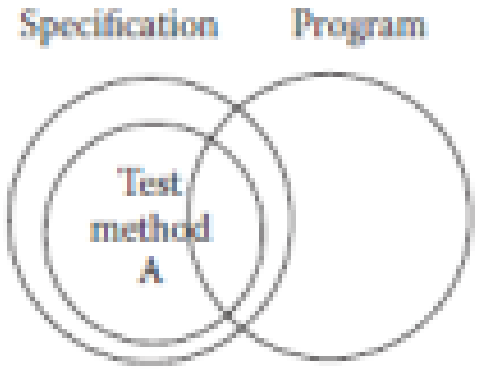
- Determine test cases so that each print statement is executed at least once

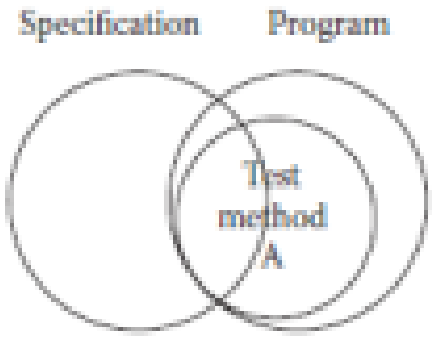
```
input(x);
if (x < 100)
print “Line 1”;
else {
if (x < 50) print “Line 2”
else print “Line 3”;
}
```



Question

- What is the weaknesses of black-box test?
- What is the weakness of white-box test?





3.3 Formal verification

- Correctness proofs

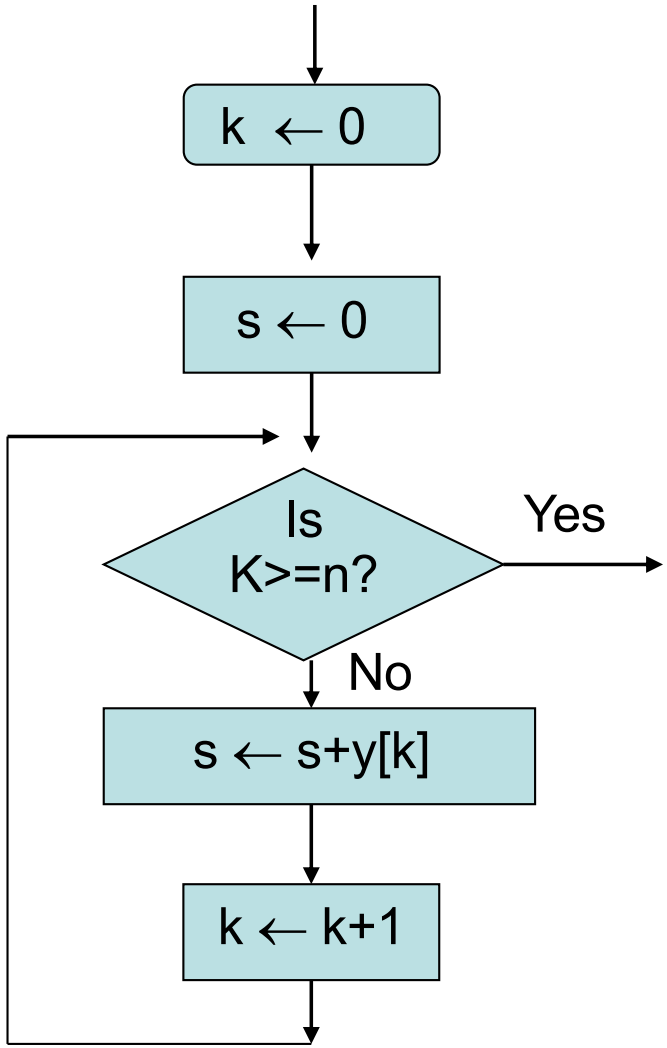
It is a mathematical technique for show that a product is correct, in other word, that it satisfies its specifications.

To see how correctness is proven, consider the fragment (next slide)

We now show that the code fragment is correct- after execute, the variable s will contain the sum of the n elements of array y.

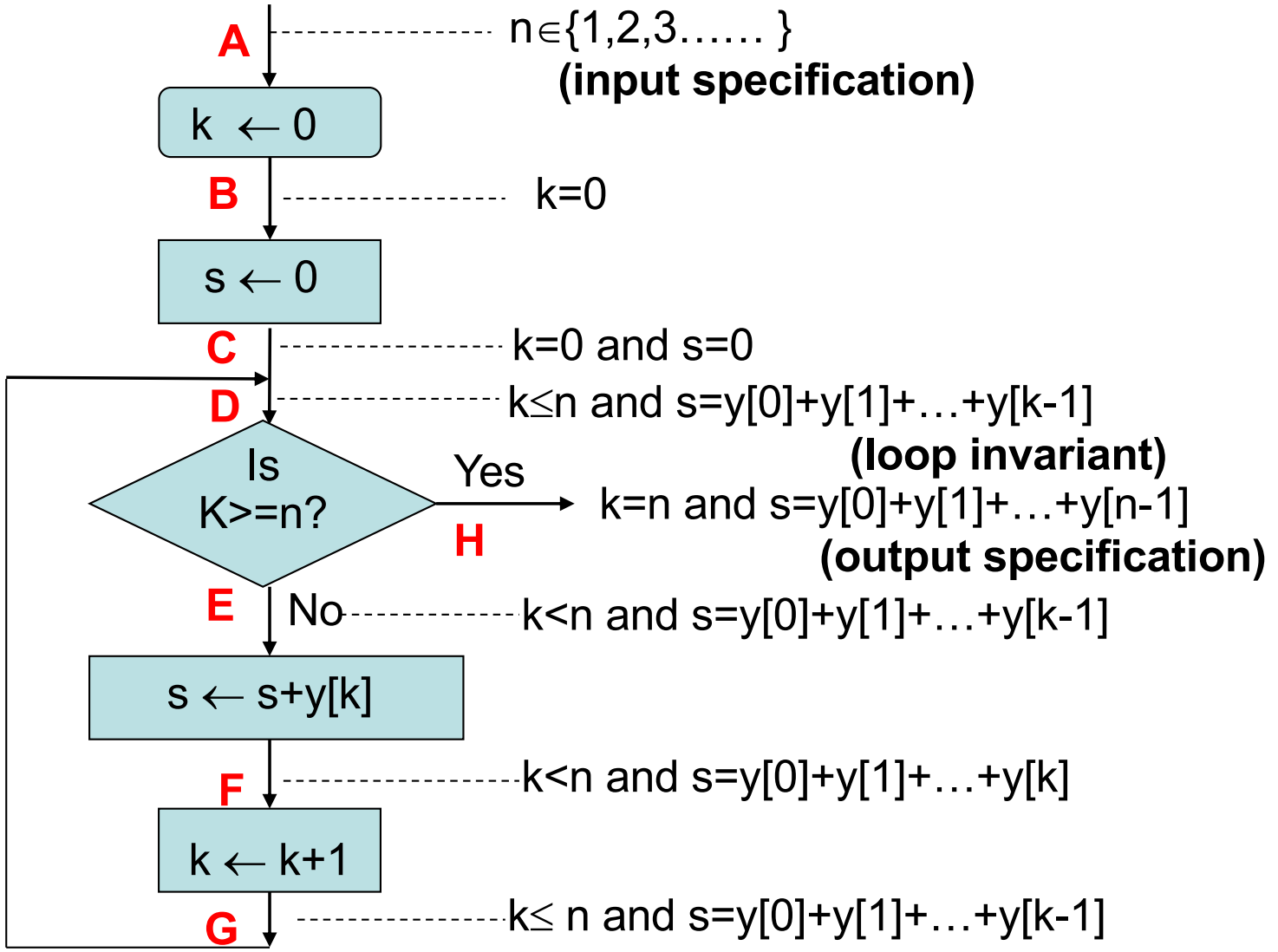
```
Int k,s;
Int y[n];
。 。 。 。 。 。
k=0;
s=0;
While (k<n)
{
   s=s+y[k];
   k=k+1;
 }
```

Code fragment



Flowchart

An assertion is placed before and after each statement, at the place labeled with the letters A through H;



Input specification, n is a positive integer,

A: $n \in \{1,2,3\ldots\ldots\}$ (1)

Output specification is that if control reaches point H, the value of ***s*** contains the sum of ***n*** values stored in array ***y***,

H: $s = y[0]+y[1]+\ldots+y[n-1]$ (2)

A stronger output specification is,

H: $k=n$ and $s = y[0]+y[1]+\ldots+y[n-1]$ (3)

How proven from A to H? key is proven the invariant of the loop, that is,

D: $k \leq n$ and $s=y[0]+y[1]+\ldots+y[k-1]$ (4)

First B: $k=0$ (5)

This is easy to prove, because of assignment statement

$k \leftarrow 0$ is executed.

At point C, the second assignment statement $s \leftarrow 0$ is executed, the follow assertion is true:

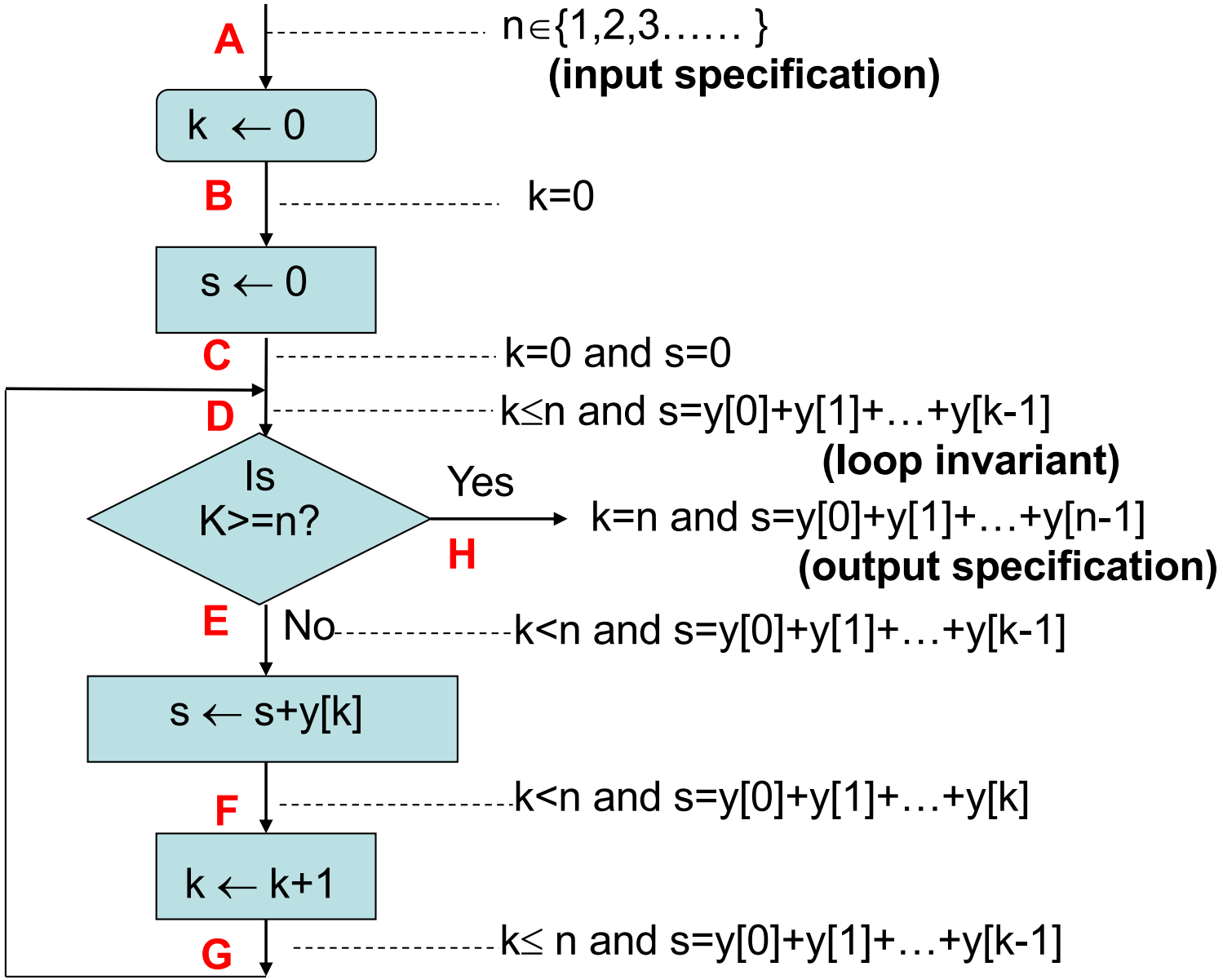
$$k=0 \text{ and } s=0 \qquad (6)$$

Now the loop is entered. Before the loop execute, assertion(6) holds k=0 and s=0. By assertion(1), $n \geq 1$. so it follows that $k \leq n$ is required. Because k=0, it follows k-1=-1, so the sum in (4) is empty s=0. there for loop invariant (4) is true just before the first loop is entered.

Now, assume that, at some stage during the execution of the code fragment, the loop invariant holds. That is k to some value k_0, $0 \leq k_0 \leq n$,

$$D: \quad k_0 \leq n \text{ and } s=y[0]+y[1]+\ldots+y[k_0-1] \qquad (7)$$

We should to prove execute the code fragment, the loop invariant is still true or get the output specification.


A
n∈{1,2,3…… }
(input specification)
k ← 0
B
k=0
s ← 0
C
k=0 and s=0
D
k≤n and s=y[0]+y[1]+…+y[k-1]
(loop invariant)
Is
K>=n?
Yes
k=n and s=y[0]+y[1]+…+y[n-1]
(output specification)
H
E
No
k<n and s=y[0]+y[1]+…+y[k-1]
s ← s+y[k]
F
k<n and s=y[0]+y[1]+…+y[k]
k ← k+1
G
k≤ n and s=y[0]+y[1]+…+y[k-1]

Pass to the test box. If $k_0 \geq n$, then because $k_0 \leq n$ by hypothesis, it follows that $k_0 = n$, this implies that

$$H: K_0 = n \text{ and } s = y[0] + y[1] + \ldots + y[n-1] \quad (8)$$

Just the output specification(3).

On the other hand, $k_0 < n$, and (7), becomes

$$E: \quad k_0 < n \text{ and } s = y[0] + y[1] + \ldots + y[k_0 - 1] \quad (9)$$

Just the assertion of E, then the statement $s \leftarrow s + y[k_0]$ is executed, the point F is

$$F: \quad k_0 < n \text{ and } s = y[0] + y[1] + \ldots + y[k_0 - 1] + y[k_0]$$
$$= y[0] + y[1] + \ldots + y[k_0] \quad (10)$$

The next statement to be executed is $k_0 \leftarrow k_0 + 1$. then because before increasing $k_0 < n$, now $k_0 \leq n$. and (10) becomes

$$G: \quad k_0 \leq n \text{ and } s = y[0] + y[1] + \ldots + y[k_0 - 1] \quad (11)$$

Assertion (11) is equal to assertion (7). That by assumption, holds at point D.

Now we have proved that $0 \leq k \leq n$, the invariant (4) is true .

Last prove is the terminates of the loop. Since the initial k is 0, and each iteration the loop increases the k by 1, surely k will reaches n, k=n. Then by assertion (8), got the output specification (3).

From the example, we could know, that only a 5 lines simple codes, the correctness proofs need 5 pages!

So for a more large program code, it is impossible to use correctness proofs for all codes. Even it is the most reliable method.

3.4 Other methods

- Def-use test
- Mutation test
- Regression Test
- Fault Statistics and Reliability Analysis
- Clean room

3.4.1 Def-use test

– Data-flow based adequacy criteria

- **All definitions criterion**
  - ***Each definition to some reachable use***
- **All uses criterion**
  - ***Definition to each reachable use***
- **All def-use criterion**
  - ***Each definition to each reachable use***

3.4.2 Mutation test

- Error seeding
  - *Introducing artificial faults to estimate the actual number of faults*
- Program mutation testing
  - *Distinguishing between original and mutants*
    - **Competent programmer assumption**
      - **Mutants are close to the program**
    - **Coupling effect assumption**
      - **Simple and complex errors are coupled**

3.4.3 Regression Testing

- Developed first version of software
- Adequately tested the first version
- Modified the software; Version 2 now needs to be tested
- How to test version 2?

- Approaches
  - *Retest entire software from scratch*
  - *Only test the changed parts, ignoring unchanged parts since they have already been tested*
  - *Could modifications have adversely affected unchanged parts of the software?*

Regression Testing

- “Software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program.”

3.4.4 Fault Statistics

- Fault statistics **provide a useful metric for determining whether to continue testing a module of product or whether to recode it.**
- **The number of faults detected via execution- and non-execution-based techniques must be recorded.**
- **Data on different types of faults found via code inspections (faults such as misunderstanding the design, initializing improperly, and using variables inconsistently) can be incorporated into checklists for use during later reviews of the same product and future products.**

Fishing Creel Counts and Fault Insertion

For example, that the Rogue River is stocked with 1000 marked trout(鳟鱼)；and that during the fishing season, the creel report totals are 500 fish caught, of which 300 are marked trout.

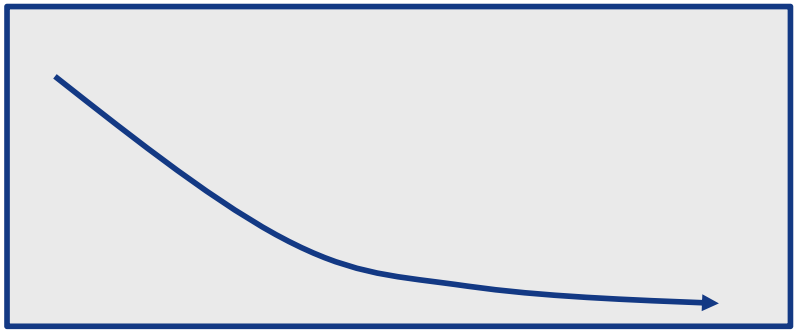
The stream management team would conclude that 60% of the trout in the Rogue River are hatchery (planted) fish. The total populations could also be estimated;

in this case, the 500 fish caught represent 30% of the total population. Total fish population is 1667.

The same idea can be applied to estimate the success of test cases and the number of remaining faults not caught by an existing set of test cases.

3.4.5 Reliability Analysis

- Reliability analysis **uses statistical-based techniques to provide estimates of how many faults are remaining and how much longer it is desirable to keep testing.**
- **It can also be applied in both implementation and integration phases.**



3.4.6 Clean-room

The *Clean-room technique* is a combination of several different software development techniques.

Under this technique, a module isn't compiled until it has passed an inspection, or another kind of non-execution-based review, such as a code walkthrough or inspection.

The relevant metric is testing fault rate**, which is the total number of faults detected per KLOC (thousand lines of code).**

The Cleanroom technique has had considerable success in finding and weeding out faults before execution-based testing.

In one case, all faults were found via correctness-proving techniques, using largely informal proofs, but with a few formal proofs as well. The resulting code was found to be free of errors both at compilation and at execution time.

In a study of 17 other Cleanroom software products, the products did not perform quite as faultlessly, but they still achieved remarkably low fault rates, an average of 2.3 faults per KLOC.

But average there are 25 faults per KLOC!

Summary

1. *Non-execution based Verification*
   - *walkthrough*
   - *inspection*
2. *Execution based Verification*
   - *black-box*
   - *white-box*
2. *Formal verification*
3. *Other methods*
   - **Def-use test**
   - **Mutation test**
   - **Regression Test**
   - **Fault Statistics and Reliability Analysis**
   - **Clean room**

测试计划实例与作业一

XXXX支撑软件与可视化软件

测试计划

目录

1、简介

- 1.1编写目的
- 为了全面、系统地对“XXX支撑软件与可视化软件”进行评估与测试，从而保证系统长期稳定的运行，组织对该软件进行系统的总体综合测试。
- 1.2项目背景
- “XXX验证平台”是中国兵器工业XXX研究所承担的国防基础科研项目。XXX所与XXX大学合作，对该项目的关键技术——XXX支撑技术和可视化技术展开研究。本项目就是在此背景下开展的，通过应用虚拟试验技术、计算机网络技术、可视化技术、数据库技术等，研制“XXX的支撑软件和可视化软件”，并最终集成在“XXX中。
- 本软件系统由以下7个二级软件组成：

  系统管理、实时通讯、三维显示、。。。。。

1.3测试范围

本次系统测试将采用开发相关测试软件和利用成熟的测试工具相结合的方式，现场对系统进行测试。按照系统需求任务书的要求，分别进行系统的功能测试、性能测试、故障测试以及安全性测试等相关内容。

本次测试的主要目标包括：

系统软件单元功能测试——根据软件详细设计文档的要求，完成各个软件模块的单元功能测试工作；

系统软件集成功能测试——根据软件需求规格说明的要求，完成软件功能测试工作；

系统软件性能测试——根据技术协议书和软件需求规格说明的要求，完成软件性能测试工作；

系统故障测试——对软件系统在网络故障、数据库故障等异常情况下进行测试工作；

安全性测试——对系统用户登陆、密码保护等安全管理进行测试工作。

- 1.4参考文档
- 《GB 8566-88 计算机软件开发规范》
- 《GJB 438A-97 武器系统软件开发文档》
- Xxxxxxxx

2、测试参考文档和测试提交文档

- 2.1测试参考文档
- 本测试计划参照GB8567——88标准编制，下表列出了制定测试计划时所使用的文档，并标明了各文档的可用性：

2.2测试提交文档

在测试完成后，需要提交给用户相关的测试评估报告与改进报告，并提交详细的测试报告。

3、测试进度

测试活动	计划开始日期	实际开始日期	结束日期
单元功能测试	20XX年 3月16日	20XX年3月16日	20XX年3月26日
集成功能测试	20XX年3月27日	20XX年3月27日	20XX年3月31日
性能测试	20XX年4月2日	20XX年4月3日	20XX年4月5日
故障测试	20XX年4月8日	20XX年4月8日	20XX年4月9日
安全测试	20XX年4月10日	20XX年4月10日	20XX年4月11日
对测试进行评估	20XX年4月15日	20XX年4月16日	20XX年4月18日
用户验收测试	20XX年5月11日		

4.1 测试资源

4.1 人力资源

姓名	角色	具体职责或注释
胡　飞	总体策划人、方案设计人	策划总体规模，测试内容，规划测试方案
A	方案设计人、测试技术设计人	制定测试方案，确定测试深度，测试技术设计
B	计划人、记录人	计划测试进程，记录测试情况
C	计划人、测试人、记录人	计划整个进程以及各个阶段的进度安排、重点任务；测试并记录测试情况
D	计划人、测试人、记录人	测试并记录测试情况
E	测试人、记录人	测试并记录测试情况
F	测试人、记录人	测试并记录测试情况
G	测试人、记录人	测试并记录测试情况

4.2 测试环境

下表列出了测试的系统环境：

机器名 环境	数据库服务器	图形工作站	管理计算机
软件环境（相关软件、操作系统等）	Windows XP Oracle 9i		Windows XP
硬件环境（设备、网络等）	Xeon Mp 2.7×2 CPU/4smp/2GB/DVD/1000M×2	HP Workstation xw6200	HP Workstation xw6200

4.3 测试工具

下表列出此项目测试中使用的工具：

用途	工具名称	生产厂商/自产	版本
压力与性能测试	Jmeter	Jakarta	1.9.1
压力测试网络检测	Webstress	国外软件	6.18
申请一定量的内存	Men	自产	1.0

五、测试策略

5.1 功能测试

由于对测试对象的功能测试应侧重于所有可直接追踪到用例或业务功能和业务规则的测试需求。这种测试的目标是核实数据的接受、计算、处理和输出是否正确，以及业务规则的实施是否恰当，所以此次的功能测试全部为黑盒测试，测试的主要目的是检测以下错误：

1）是否有不正确或遗漏的功能？

2）在接口上，输入是否能正确的接受？能否输出正确的结果？

3）是否有数据错误或外部信息（例如数据文件）访问错误？

4）是否有初始化或终止性错误？

测试方法：

1）等价类划分

XXXXXXXXXXXXX

XXXXXXXXXXXXX

2） 边界值分析

Xxxxxxxxxxxxxx

Xxxxxxxxxxxxxx

3) 确立测试用例

5.1.1 具体功能模块1

5.1.2 具体功能模块2

Xxxxxxxxxxxxxxxxx

5.1.n 具体功能模块n

学生作业样本(e-learning)

作业一

软件功能性测试

寻找具有需求说明的（或熟悉软件功能需求）软件，采用面向功能的测试方法，生成黑盒测试用例，使用测试记录工具，报告所有发现的错误。

提交的报告：

- 测试计划
- 测试报告
- 测试结果分析报告

作业 提交时间：第6周周六以前（10月25日前）

Outline of a review report

1. Introduction
    a. Work product identification
    b. Review team members and roles
2. Preliminary issue list
    a. Potential fault
    b. Severity
3. Prioritized action item list
    a. Identified fault
    b. Severity
4. Summary of individual reports
5. Review statistics
    a. Total hours spent
    b. Faults sorted by severity
    c. Faults sorted by location
6. Review recommendation
7. Appendix with the full review packet

return