投票 最多可选1项

设置

Are you ready?

A Yes

B No

提交

Review

- **Testing concept**
  - ✓ strategic
  - ✓ V & V
  - ✓ testing plan
  - ✓ test suite & test case
- **Unit testing and Integration testing**
- **System testing**
- **Validation testing**
- **Debugging**

Software Engineering

Part 3
Quality Management

Chapter 23
Testing Conventional Applications

Reproduced by Ning Li , 2022

Contents

Contents

- 23.6 Black-box Testing
  - 23.6.1 Graph-based Testing Methods
  - 23.6.2 Equivalence Partitioning
  - 23.6.3 Boundary Value Analysis
  - 23.6.4 Orthogonal Array Testing
- 23.7 Model-based Testing
- 23.8 Testing Documentation and Help Facilities
- 23.9 Testing for Real-Time Systems
- 23.10 Patterns for Software Testing

23.1 Testability (characteristics)

- Operability — it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

23.1 What is a “Good” Test?

- A good test has a **high** probability of finding an error
- A good test is not redundant.
- A good test should be neither too simple nor too complex (executed separately).

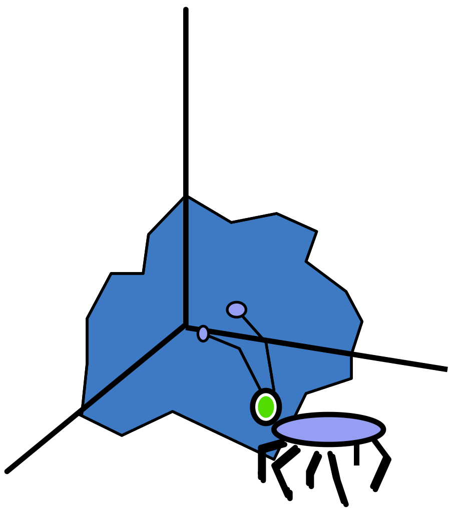
23.2 Internal and External Views

- External views: Knowing the specified function that a product **has been designed** to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
- Internal views: Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

23.2 Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



OBJECTIVE	to uncover errors
CRITERIA	in a complete manner
CONSTRAINT	with a minimum of effort and time

23.2 Testing Strategy

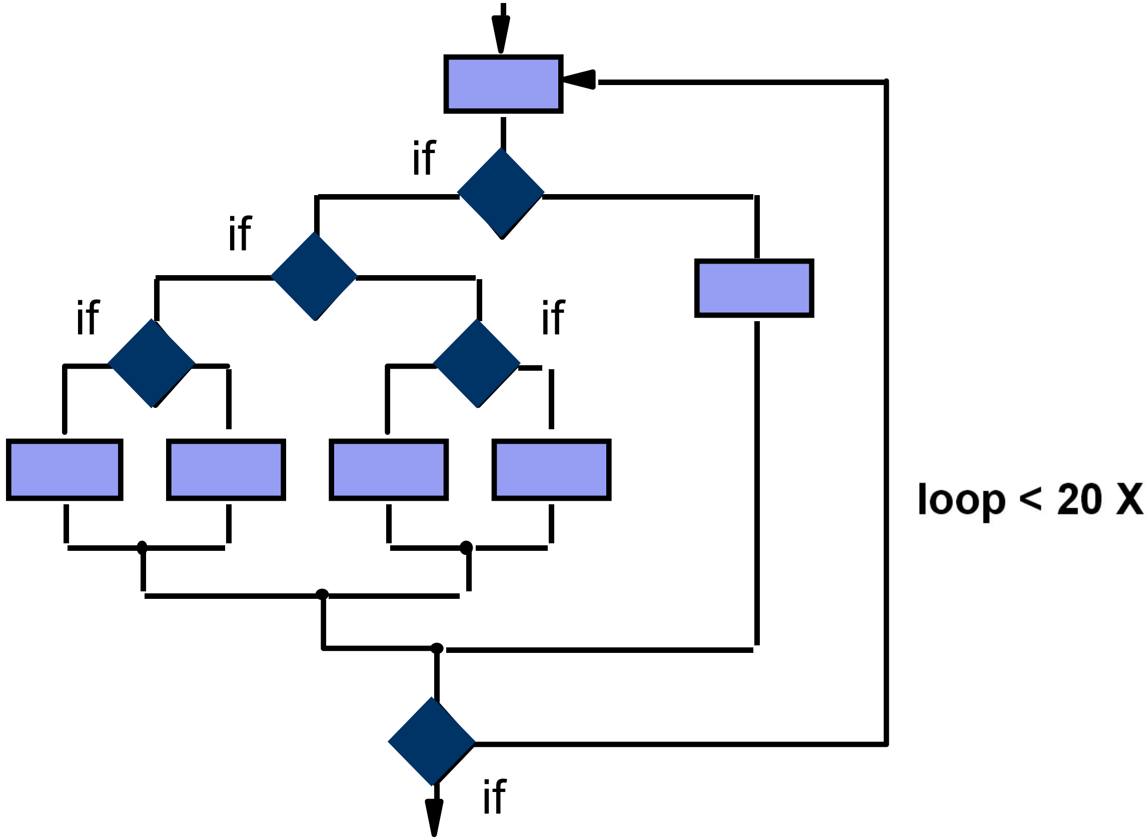
- Testcase :

  condition + input data + operation =>

  expected output: Yes (pass) or No (fail)

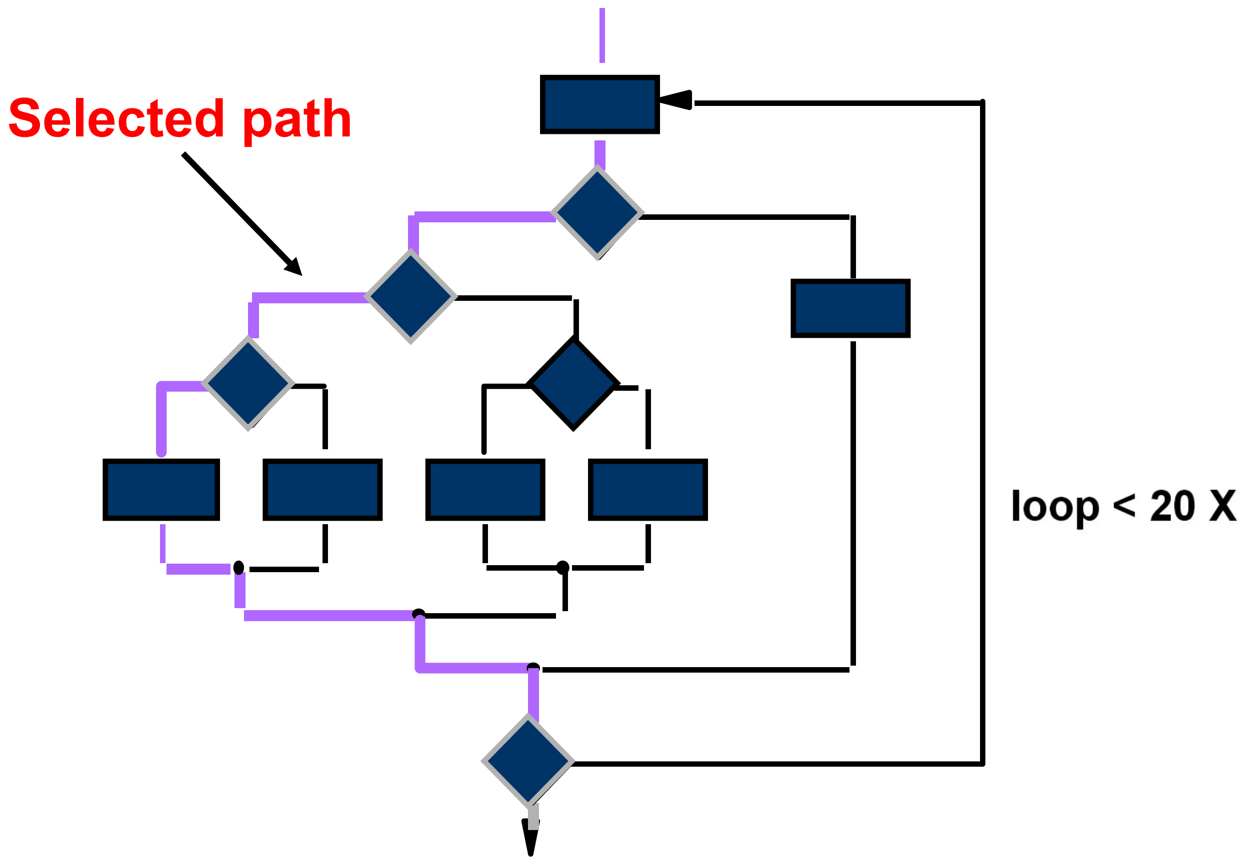
Test Case ID	Test Scenario	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TU01	Check Customer Login with valid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = pass99	User should Login into application	As Expected	Pass
TU02	Check Customer Login with invalid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = glass99	User should not Login into application	As Expected	Pass

23.2 Exhaustive Testing

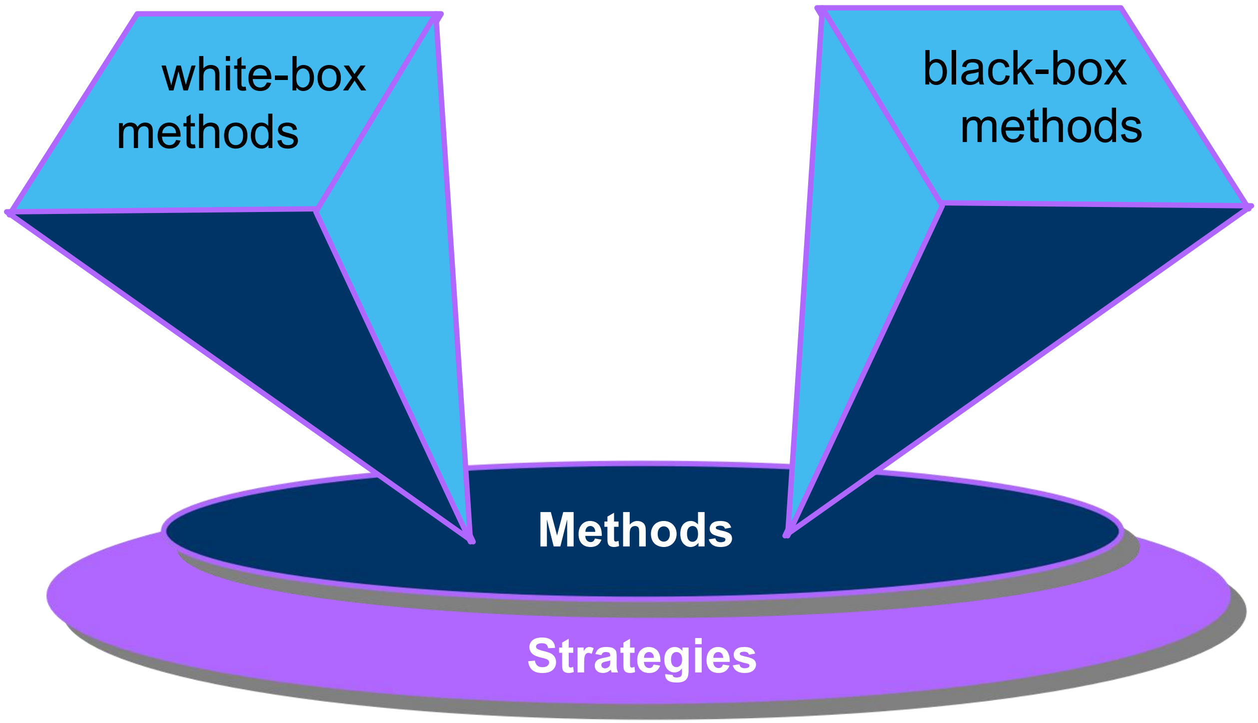


There are 10^{14} possible paths!
If 1 test per millisecond, it would take 3,170 years if we execute one test this program!!!

23.2 Selective Testing



23.3 Software Testing



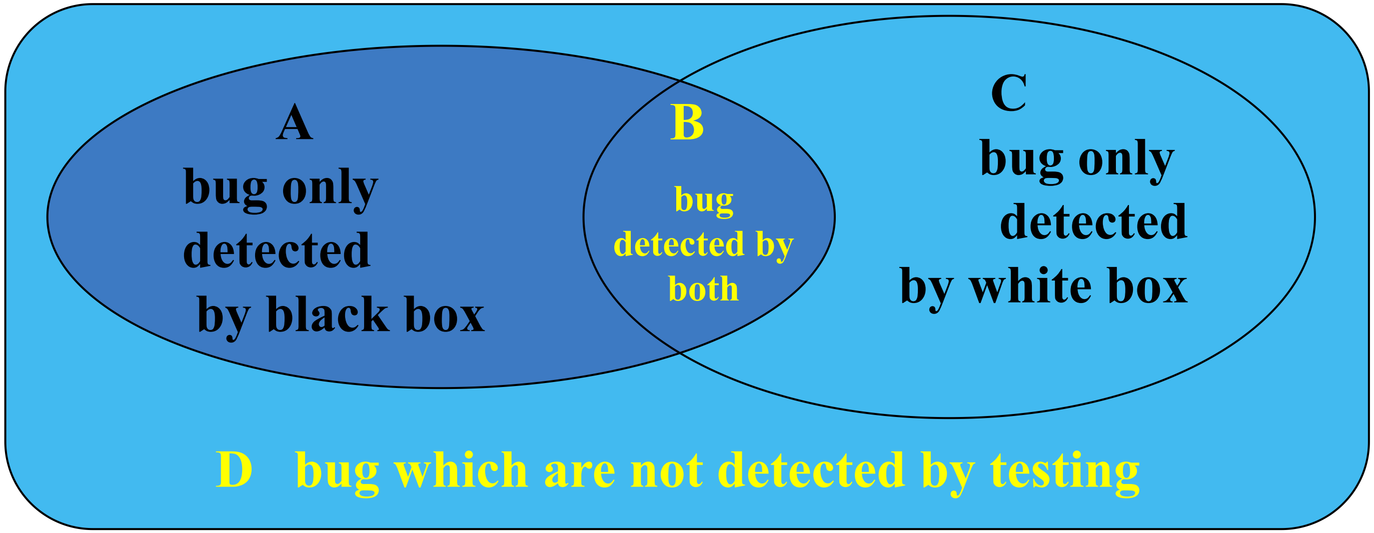
23.3 Software Testing

- White box:

  structure of the test objects

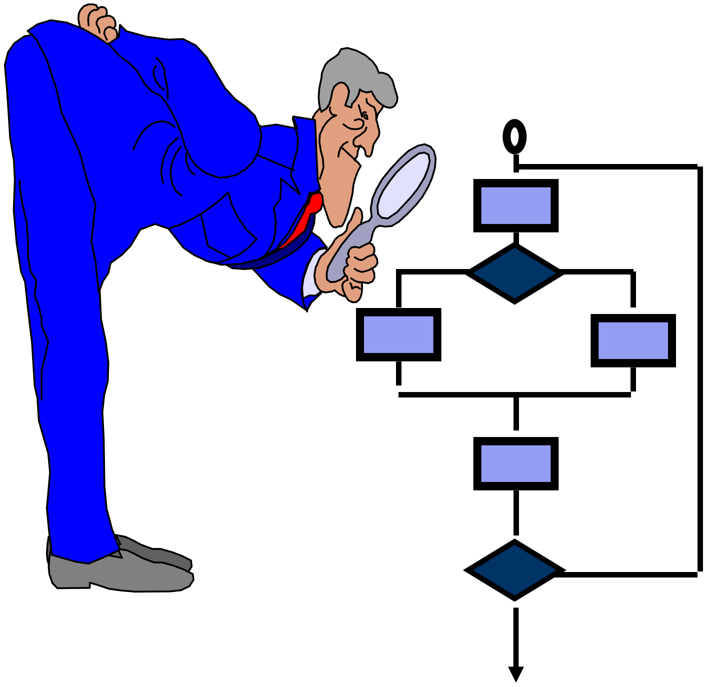
- Black box:

  functionality of the test objects



A＋B＋C＋D ＝ all bugs

23.3 White-Box Testing



Our goal is to ensure that all statements and conditions have been executed at least once ...

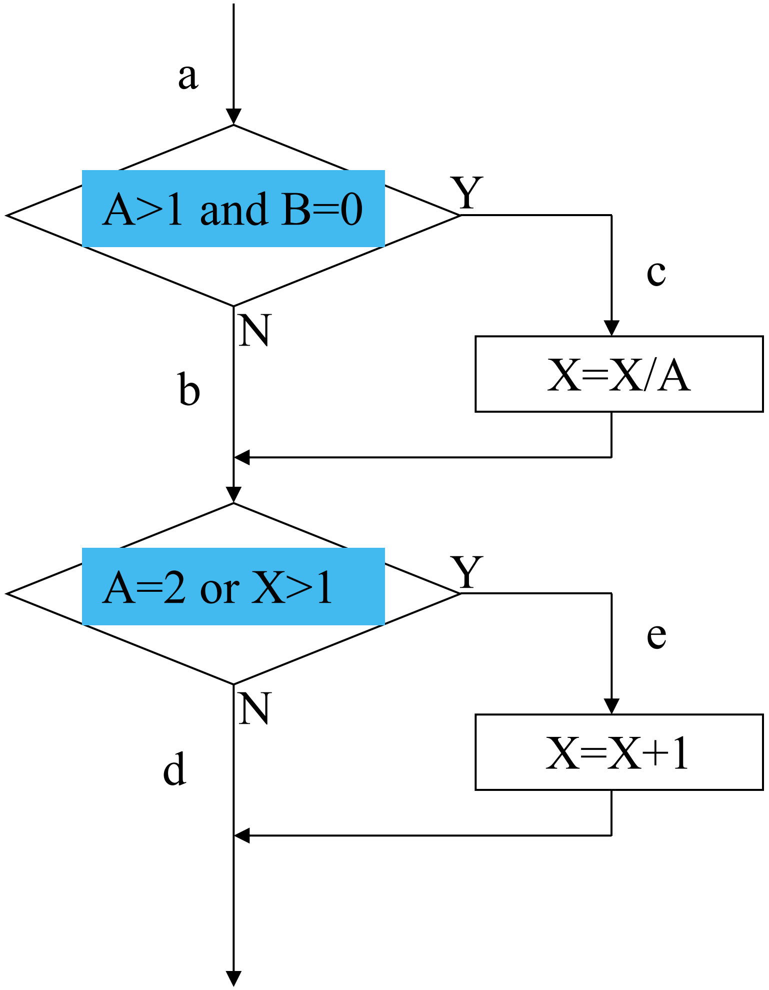
23.3 White-Box Testing

Choose limited test cases which can uncover the bug effectively amonge the countless items .

White-box testing methods:

1. statement coverage
2. brach coverage
3. decision converage
4. branch/decision coverage
5. combination coverage
6. basic path coverage (include sec 23.4)

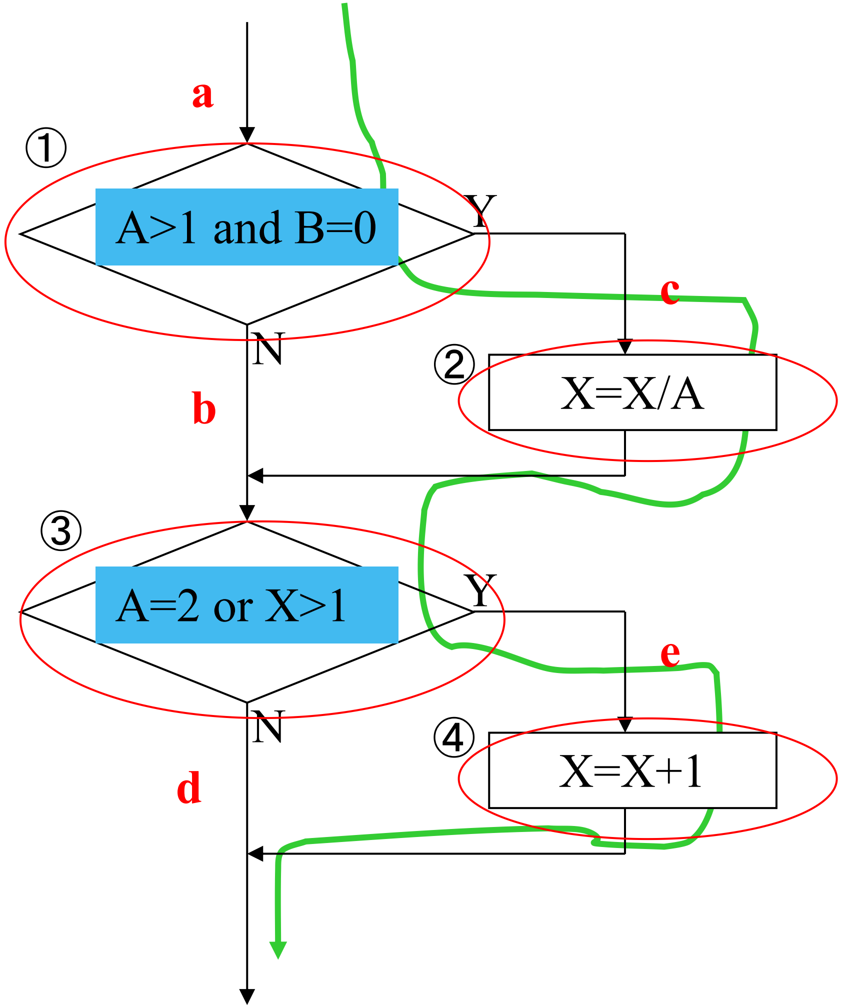
23.3 White-Box Testing Design




```
int MySample(int a, b, x)
{

    if ( (a>1) && (b==0) )
    {
        x = x/a;
    }

    if ( (a==2) || (x>1) )
    {
        x = x+1;
    }

}
```

23.3 White-Box Testing Design

1. statement coverage: every statement is executed



4 statements:

①③conditional statements

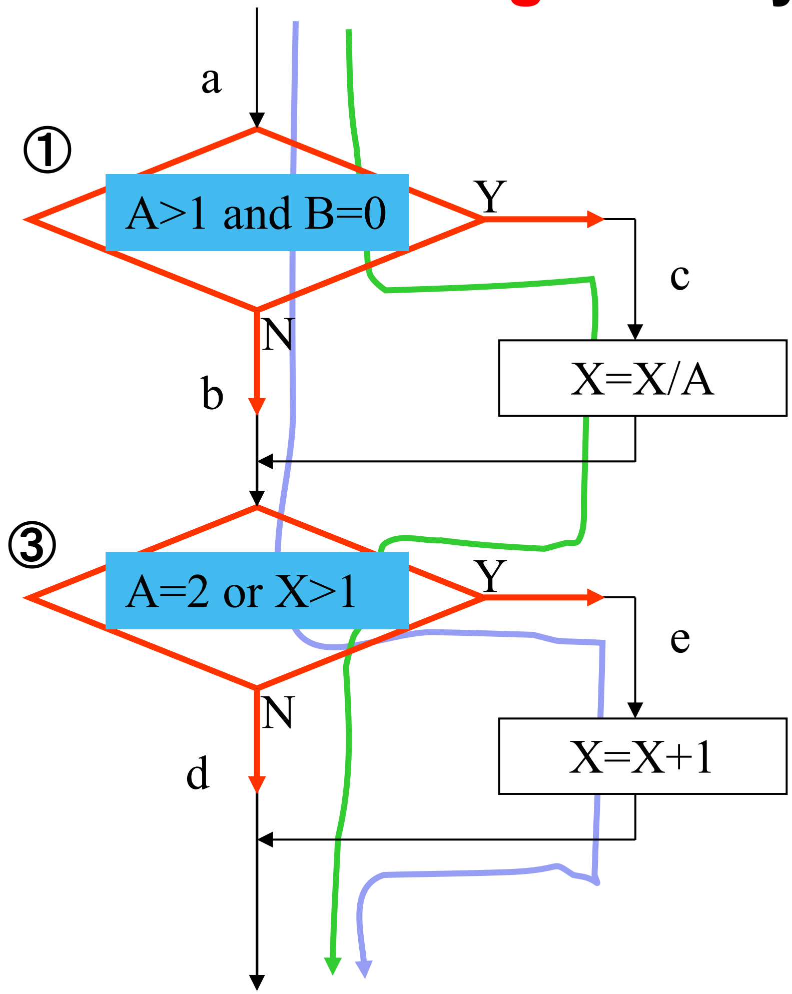
②④assignment statements

Cover all statements: **ace**

Input: A=2，B=0, X=3

23.3 White-Box Testing Design

2. branch coverage: every branch is executed



①③ : conditional statements

Cover all branch(Y/N):

acd, abe

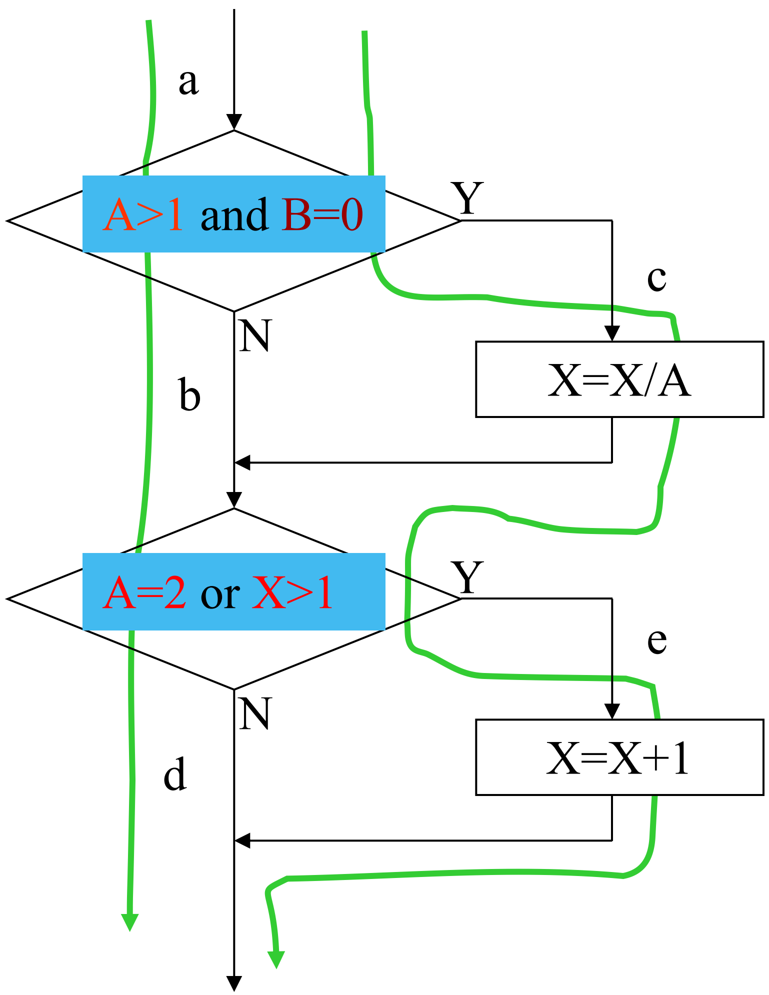
Input data:

A=3，B=0，X=1 (acd)

A=2，B=1，X=3 (abe)

23.3 White-Box Testing Design

3. decision coverage: every decision is executed



Four decisions:

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

Cover true and false of four decisions:

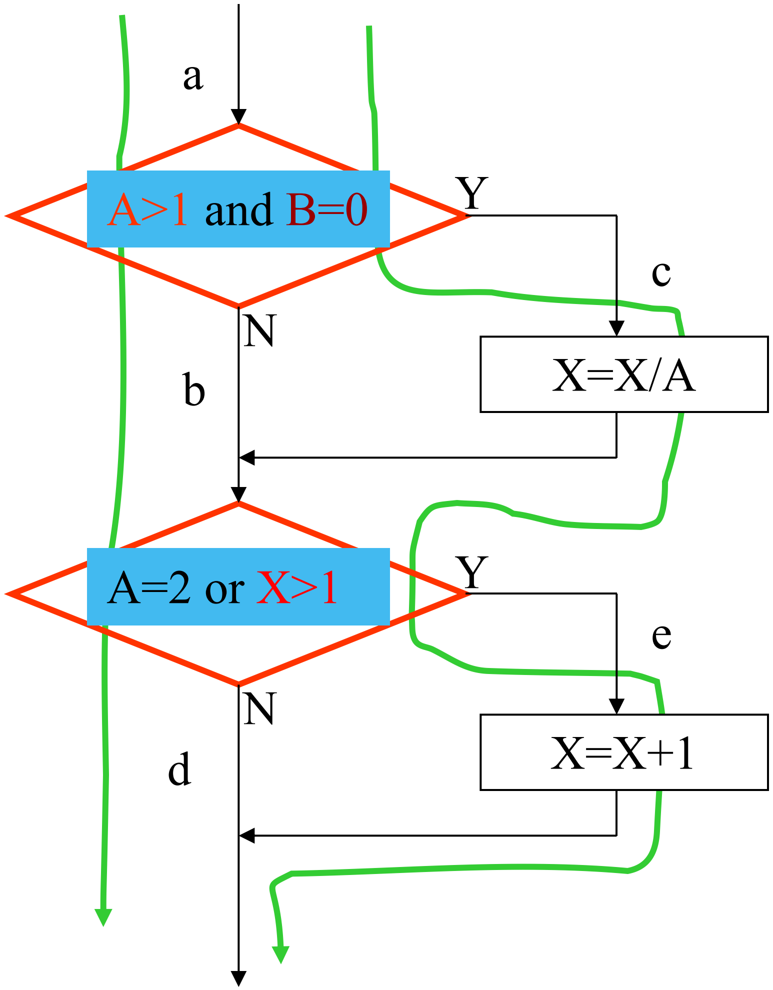
Input data:

A=2，B=0, X=4 (ace)

A=1, B=1, X=1 (abd)

23.3 White-Box Testing Design

4. branch/decision coverage: every branch and every decision is executed at least one time



Four decisions:

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

Cover two conditions,
true and false of four decisions:

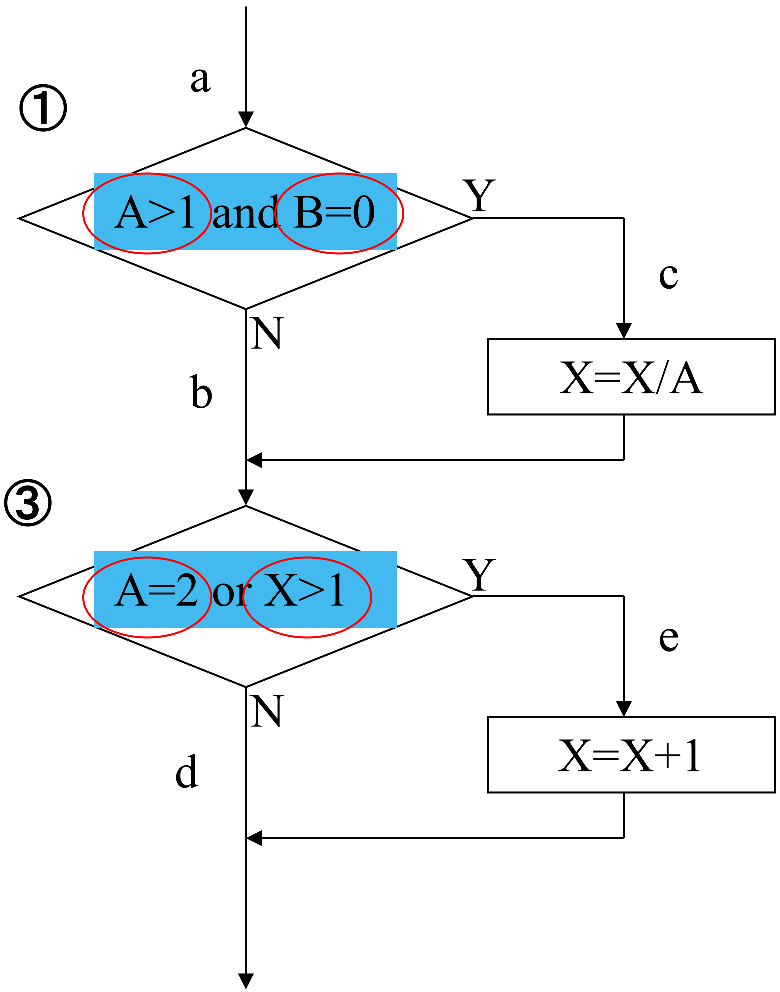
Input data:

A=2, B=0, X=4 (ace)

A=1, B=1, X=1 (abd)

23.3 White-Box Testing Design

5. Decision combination: each decision combination is executed.



Top：①

1. True and True → A>1, B=0

2. True and False → A>1, B<>0

3. False and True → A<=1, B=0

4. False and False → A<=1, B <> 0

Bottom: ③

5. True and True → A=2, X>1

6. True and False → A=2, X<=1

7. False and True → A<>2, X>1

8. False and False → A<>2, X <= 1

23.3 White-Box Testing Design

5. Decision combination: each decision combination is executed.

Decision combination analysis.

Top \ Bottom	⑤A = 2 、 X > 1	⑥A = 2 、 X ≦ 1	⑦A ≠ 2 、 X > 1	⑧A ≠ 2 、 X ≦ 1
①A > 1 、 B = 0	○	○	○	○
②A > 1 、 B ≠ 0	○	○	○	○
③A ≦ 1 、 B = 0	×	×	○	○
④A ≦ 1 、 B ≠ 0	×	×	○	○

○: conditions can hold at the same time

×: conditions cannot hold at the same time

23.3 White-Box Testing Design

5. Decision combination: each decision combination is executed.

Decision combination analysis.

Valid: 12 combination

Bottom / Top	⑤A =2 、X >1	⑥A =2 、X ≦1	⑦A ≠2 、X >1	⑧A ≠2 、X ≦1
①A >1 、B=0	○	○	○	○
②A >1 、B ≠0	○	○	○	○
③A ≦1 、B=0	×	×	○	○
④A ≦1 、B ≠0	×	×	○	○

（①A＞1 、 B＝0 ）&& （⑤A＝2 、X＞1 ）：A＝2、B＝0 、X＝4

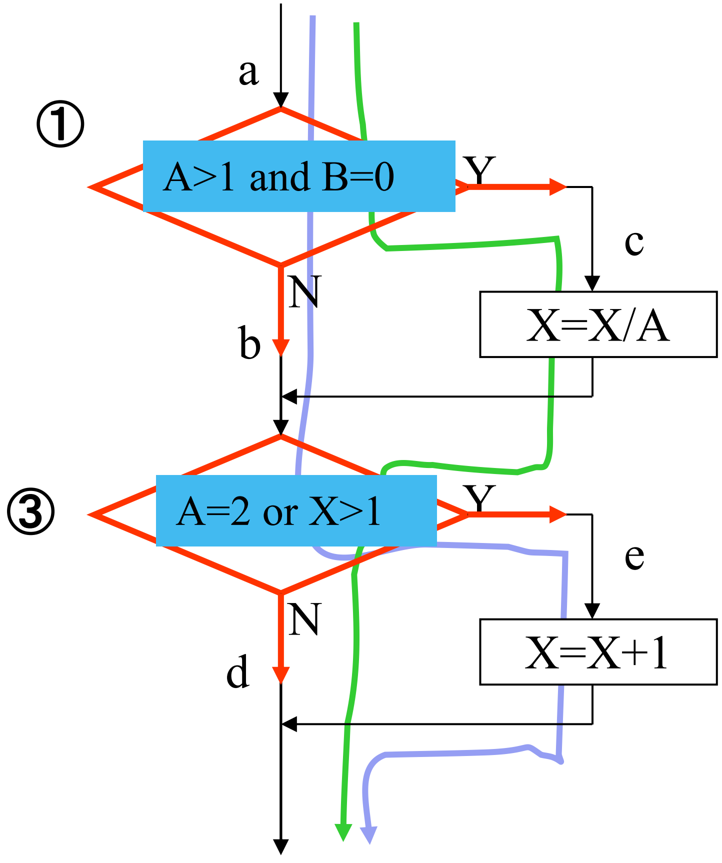
（②A＞1 、 B≠0 ）&& （⑥A＝2 、X≦1 ）：A＝2、B＝1 、X＝1

....

23.3 White-Box Testing Design

6. **path coverage:** each path is executed.

(branch combination)



①③ combination

1. **True and True → ace**
2. **True and False → acd**
3. **False and True → abe**
4. **False and False → abd**

Test **ALL** possible paths.

a,c,e

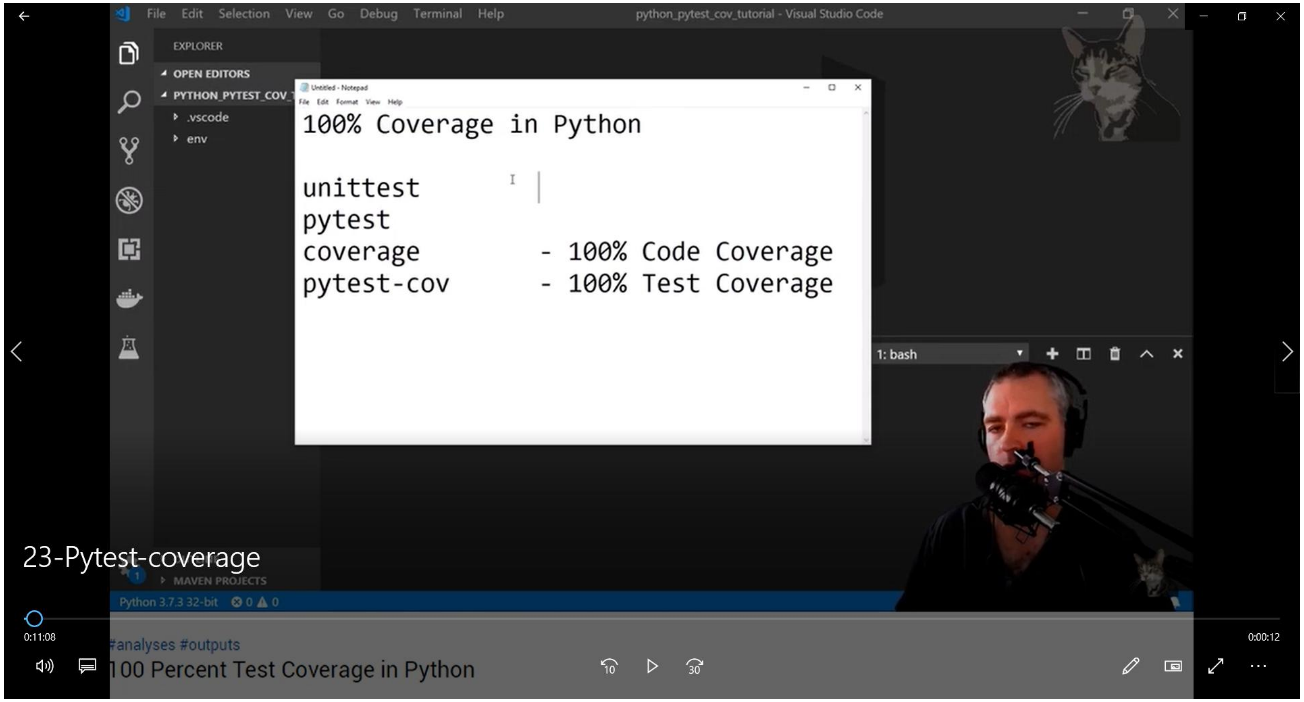
a,c,d

a,b,e

a,b,d

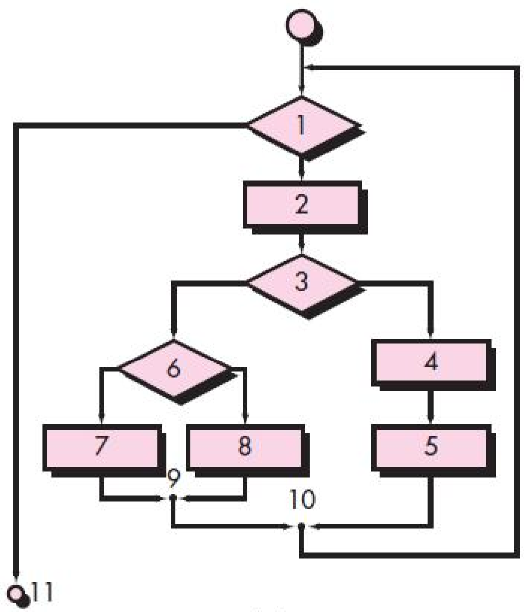
23.3 White-Box Testing Design

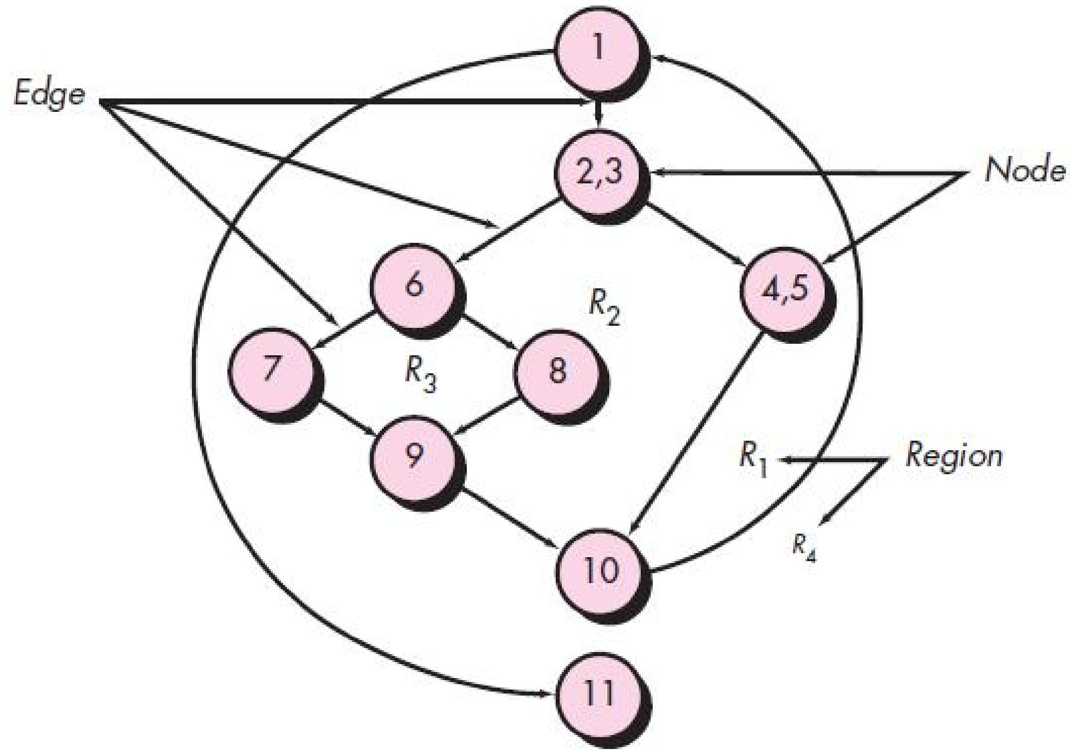
Let's watch! (tool: pytest, coverage)



https://www.youtube.com/watch?v=7BJ_BKeeJyM

23.4 Basis Path Testing - Flow chart





region: R1,R2,R3,R4
enclosed areas: R1,R2,R3

23.4 Independent Program Path

- An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.
- When stated in terms of a flow graph, an independent path must **move along** at least one **edge** that has not been traversed before the path is defined

单选题 1分 此题未设置答案，请点击右侧设置按钮 设置

Suppose you have these paths:

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

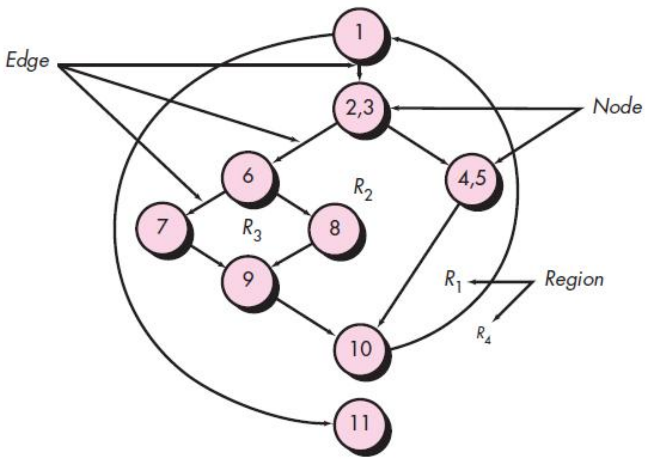
path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Consider this path:

path 5: 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Q: Is path5 is an independent path?

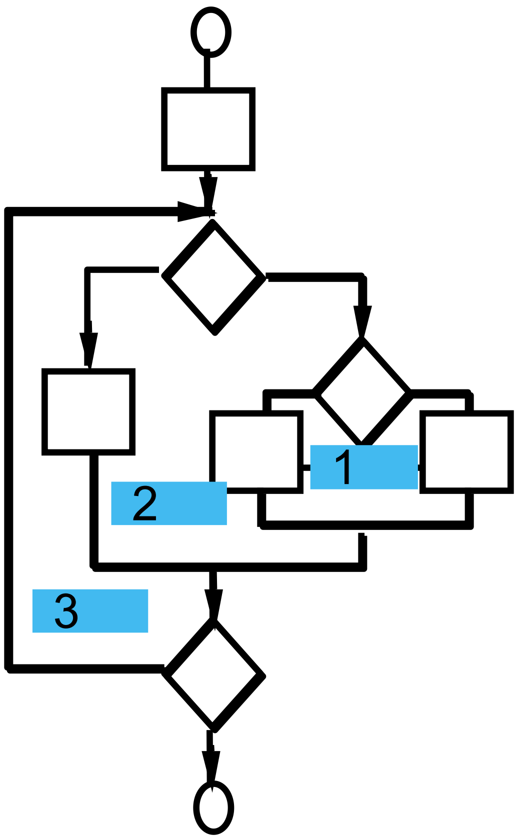


A Yes

B No

提交

23.4 Basis Path Testing



First, we compute the cyclomatic complexity:

$$V(G) = E - N + 2 = 9 - 7 + 2 = 4$$

Other simple computation:

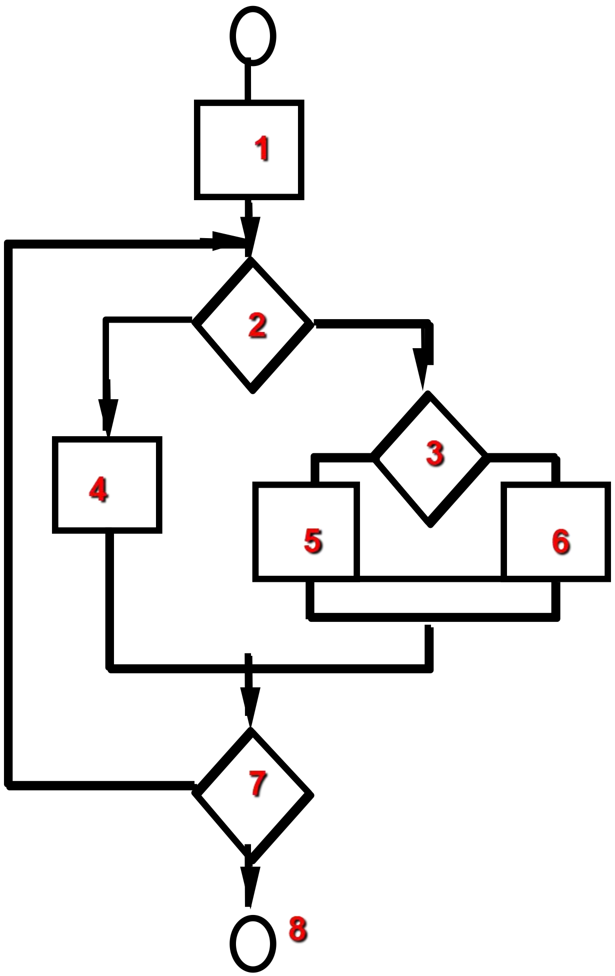
number of simple condition + 1 (3+1)

or

number of enclosed areas + 1 (3+1)

In this case, V(G) = 4

23.4 Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4,

there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

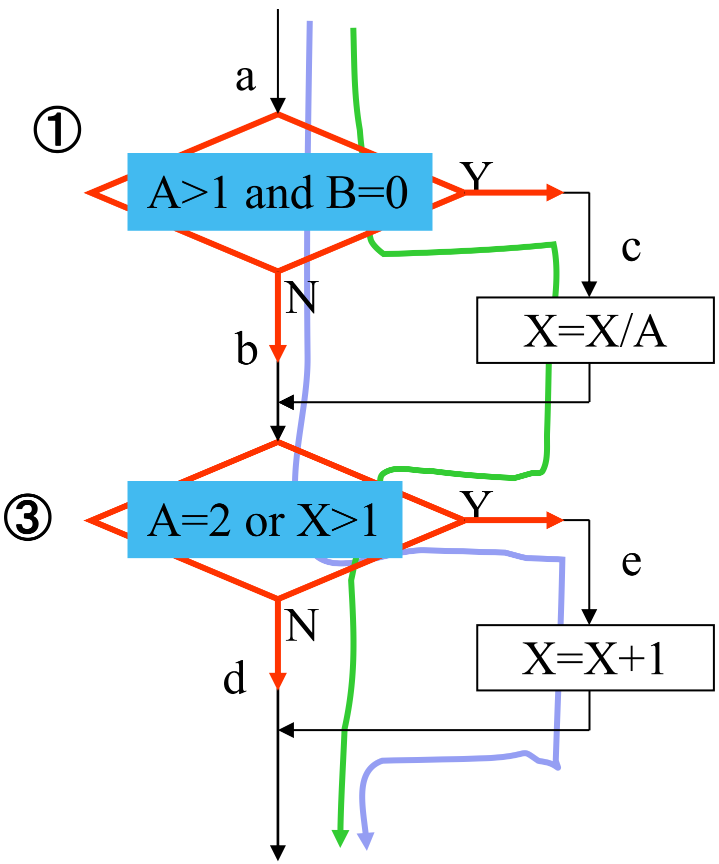
Finally, we derive test cases to exercise these paths.

23.4 Deriving Test Cases

- *Steps for* path design:
  - Using the design or code as a foundation, draw a corresponding **flow graph**.
  - Determine the cyclomatic complexity of the resultant **flow graph**.
  - Determine a **basis** set of linearly independent **paths**.
  - Prepare **test cases** that will force execution of each path in the basis set.

23.4 Basis Path Testing

Question: Please find basic paths.



Test **ALL** possible paths.

a,c,e

a,c,d

a,b,e

a,b,d

Please find basic paths.

a,c,e

a,c,d

a,b,e

Take a break

Five minutes

23.5 Control Structure Testing

- Condition testing — a test case design method that exercises the **logical conditions** contained in a program module

- Data flow testing — selects **test paths** of a program according to the locations of definitions and uses of variables in the program (du pair)

23.5 Data Flow Testing

- The data flow testing method selects test paths of a program according to *the **locations** of definitions and uses of **variables*** in the program.
  - Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number
    - DEF(S) = {X | statement S contains a definition of X}
    - USE(S) = {X | statement S contains a use of X}
  - A ***definition-use (DU) chain*** of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'

23.5 Data Flow Testing

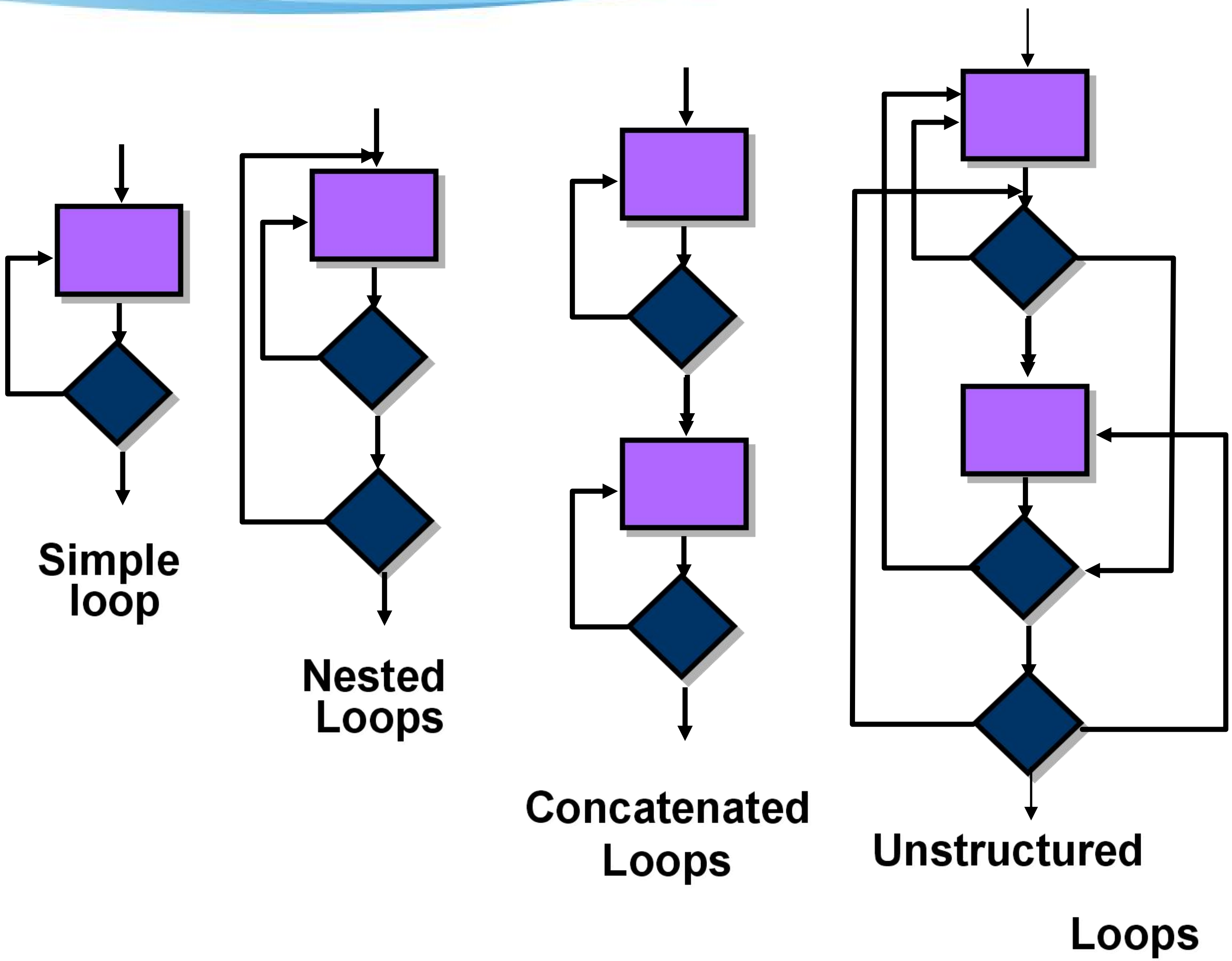
```
If ( Condition 1 ) {
    x = a;
}
Else {
    x = b;
}

If ( Condition 2 ) {
    y = x + 1;
}
Else {
    y = x - 1;
}
```

What test cases do we need?

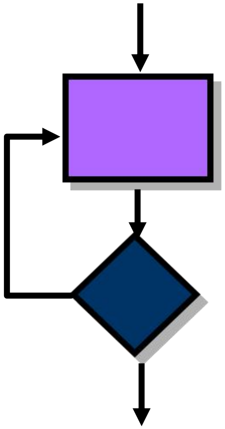
Definitions: 1) x = a; 2) x = b;
Uses: 1) y = x + 1; 2) y = x – 1;

test case:
1. (x = a, y = x + 1)
2. (x = a, y = x – 1)
3. (x = b, y = x + 1)
4. (x = b, y = x – 1)

23.5 Loop Testing



23.5 Loop Testing: Simple Loops

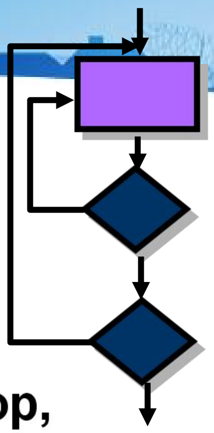


Minimum conditions—Simple Loops

1. Skip loop entirely
2. One pass
3. Two passes
4. N-1, N, and N+1 passes [N is the maximum number of passes]
5. M passes, where 2 < M < N-1

where n is the maximum number of allowable passes

23.5 Loop Testing: Nested Loops



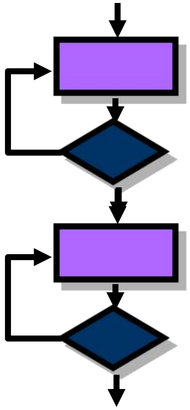
Nested Loops

- Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.
- Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.
- Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

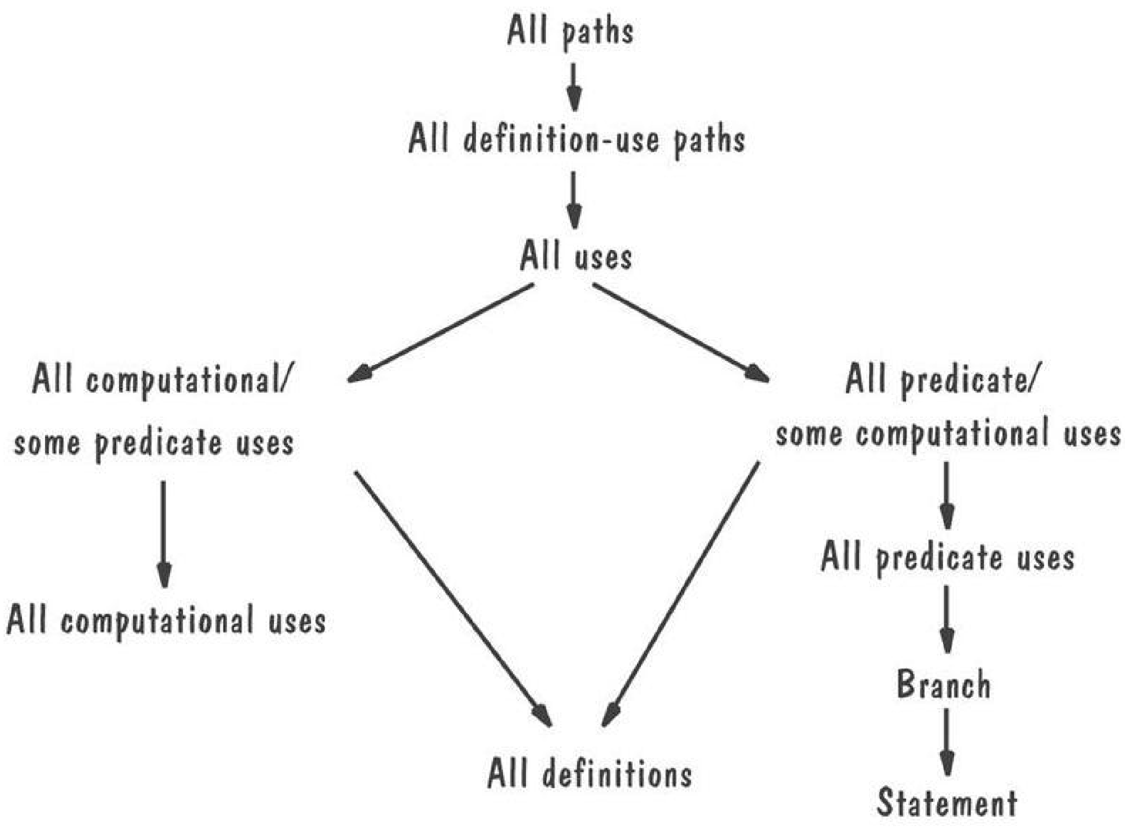
for example, the final loop counter value of loop 1 is used to initialize loop 2.



Practice:

Please write three pieces of code (simple loop, nested loops, concatenated loops), then design test case for them.

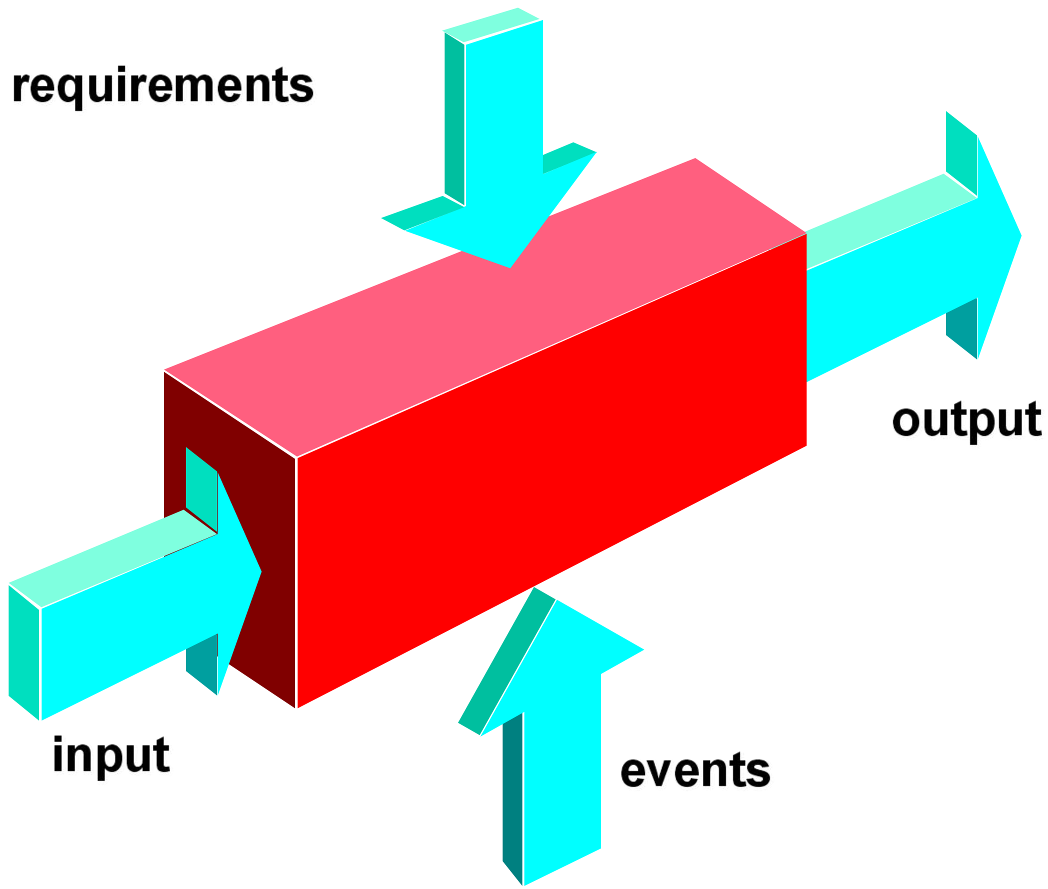
23.5 Relative Strengths of Test Strategies



Take a break

Five minutes

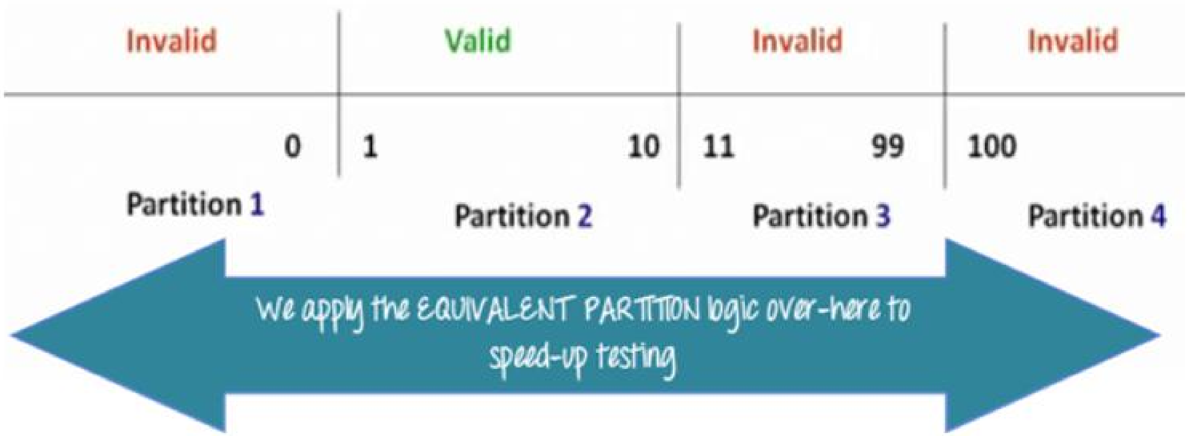
23.6 Black-Box Testing



23.6 Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

23.6 Equivalence Partitioning

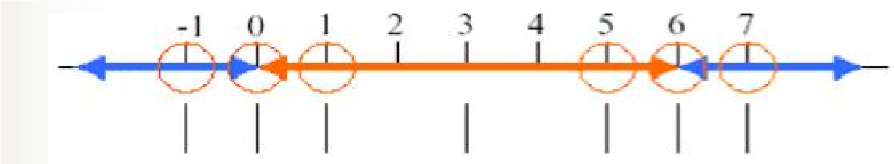


Ticket values : 1 to 10 are considered valid.

1. Any number greater than 10 (let say 11) is considered invalid.
2. Any number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid.
4. Any 3 digit number (say 100) is invalid.

23.6 Boundary Value Analysis

Boundary value is likely to be faulty.

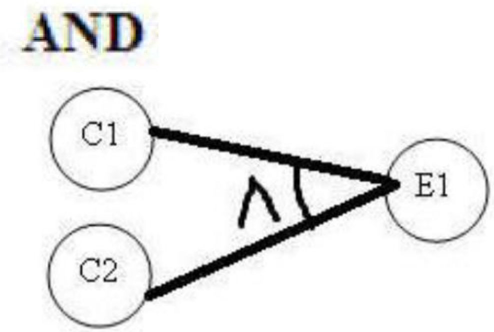


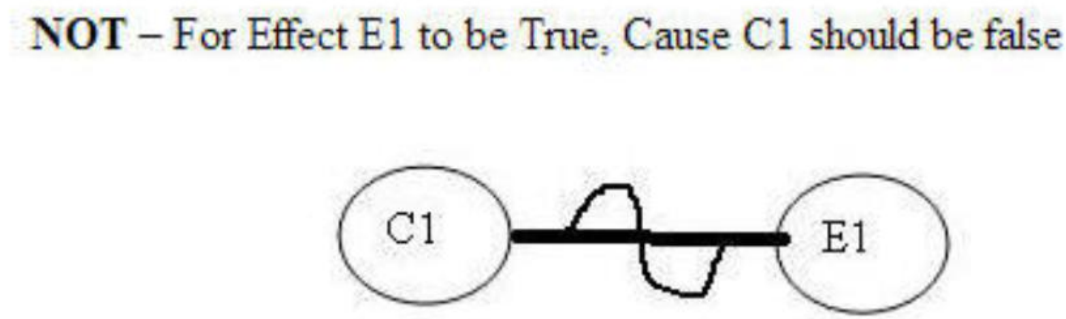
Equivalent Class Partitioning -> Boundary value : [0,6]

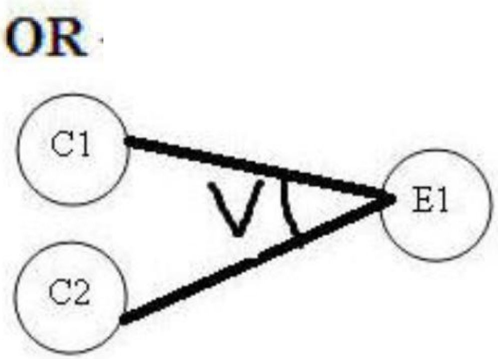
1. **Minimum: 0 (-1)**
2. **Just above the minimum: 1**
3. **A nominal value: 4**
4. **Just below the maximum: 5**
5. **Maximum: 6 (7)**

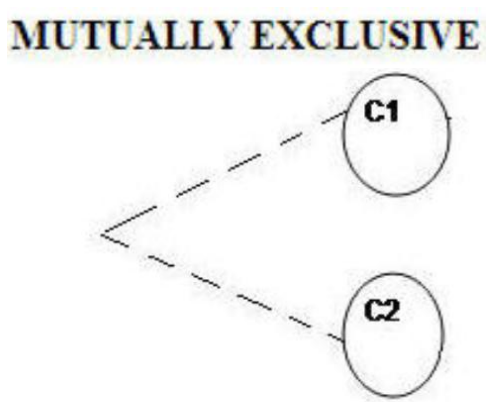
23.6 Cause-effect graph & Decision table

Causes are the input conditions ;
Effects are the results of those input conditions









C1 or C2 is true
(but just one hold)

23.6 Cause-effect graph & Decision table

The "Print message" is software that read two characters and, depending on their values, messages must be printed.

- The first character must be an "A" or a "B".
- The second character must be a digit.
- If the first character is an "A" or "B" and the second character is a digit, the file must be updated.
- If the first character is incorrect (not an "A" or "B"), the message X must be printed.
- If the second character is incorrect (not a digit), the message Y must be printed.

23.6 Cause-effect graph & Decision table

Solution:

The causes for this situation are:

C1 – First character is A

C2 – First character is B

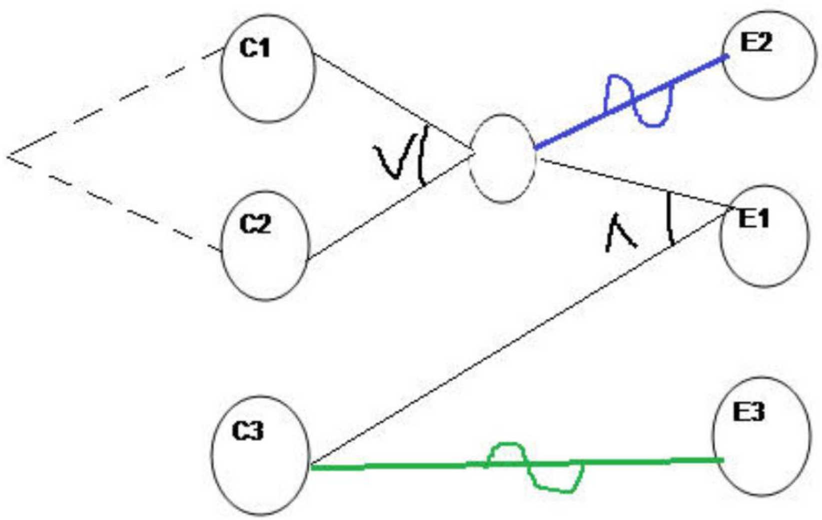
C3 – the Second character is a digit

The effects (results) for this situation are

E1 – Update the file

E2 – Print message "X"

E3 – Print message "Y"



23.6 Cause-effect graph & Decision table

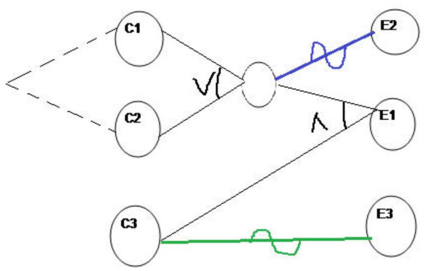
Here we are representing True as 1 and False as 0

Actions
C1
C2
C3
E1
E2
E3

Actions	TC1	TC2
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

Actions			TC3	TC4
C1	1		0	0
C2		1	0	0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1



23.6 Cause-effect graph & Decision table

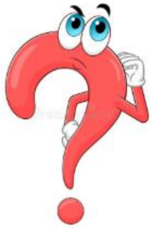
Test case： TC1 & TC2

TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

23.7 Combination Testing

Input A	*Input B*	*Input C*	*Input D*
A1 A2	B1 B2	C1 C2	D1 D2 D3

Test all possible combinations of parametric values



23.7 Combination Testing

Input A	Input B	Input C	Input D
A1 A2	B1 B2	C1 C2	D1 D2 D3

Test cases

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A1	B1	C1	D2
A1	B1	C1	D3
A1	B1	C2	D1
A1	B1	C2	D2
A1	B1	C2	D3
A1	B2	C1	D1
A1	B2	C1	D2
A1	B2	C1	D3
A1	B2	C2	D1
A1	B2	C2	D2
A1	B2	C2	D3

Input A	Input B	Input C	Input D
A2	B1	C1	D1
A2	B1	C1	D2
A2	B1	C1	D3
A2	B1	C2	D1
A2	B1	C2	D2
A2	B1	C2	D3
A2	B2	C1	D1
A2	B2	C1	D2
A2	B2	C1	D3
A2	B2	C2	D1
A2	B2	C2	D2
A2	B2	C2	D3

Number of test cases?

2*2*2*3=24

23.7 Combination Testing

Consider the parameters shown in the table below.

Parameter name	Value 1	Value 2	Value 3	Value 4
Enabled	True	False	*	*
Choice type	1	2	3	*
Category	a	b	c	d

Enabled: 2
Choice Type: 3
Category: 4

- all pair tests(an exhaustive test): 24 tests (2 x 3 x 4).
- pair-wise tests: multiplying the two largest values (3 and 4) indicates that a pair-wise tests would involve 12 tests.

Enabled	Choice	Category
true	1	a
false	1	b
true	1	c
true	1	d
false	2	a
false	2	b
true	2	c
false	2	d
true	3	a
false	3	b
false	3	c
false	3	d

23.7 Combination Testing

Input A	Input B	Input C	Input D
A1 A2	B1 B2	C1 C2	D1 D2 D3

Input A	*Input B*	*Input C*	*Input D*
A1	B1	C1	D1
A1	B1	C2	D2
A1	B2	C1	D3
A2	B1	C2	D1
A2	B2	C1	D2
A2	B2	C2	D3

23.7 Combination Testing

T-wise/T-way combinatorial testing

Input A	Input B	Input C	Input D
A1 A2	B1 B2	C1 C2	D1 D2 D3

Test cases (T=3)

Input A	Input B	Input C	Input D
A1	B1	C1	D1
A1	B2	C2	D1
A2	B1	C2	D1
A2	B2	C1	D1
A1	B1	C2	D2
A1	B2	C1	D2
A2	B1	C1	D2
A2	B2	C2	D2
A1	B1	C1	D3
A1	B2	C2	D3
A2	B1	C2	D3
A2	B2	C1	D3

23.7 Combination Testing

```python
from allpairspy import AllPairs


parameters = [
    ["A1", "A2"],
    ["B1", "B2", "B3"],
    [5, 10, 15, 20]
]

print("PAIRWISE:")
for i, pairs in enumerate(AllPairs(parameters)):
    print("{:2d}: {}".format(i+1, pairs))
```

for i, pairs in enumerate(AllPa...

Run: PairwiseTest

```
PAIRWISE:
 1: ['A1', 'B1', 5]
 2: ['A2', 'B2', 5]
 3: ['A2', 'B3', 10]
 4: ['A1', 'B3', 15]
 5: ['A1', 'B2', 10]
 6: ['A2', 'B1', 15]
 7: ['A2', 'B1', 20]
 8: ['A1', 'B2', 20]
 9: ['A1', 'B3', 20]
10: ['A1', 'B3', 5]
11: ['A1', 'B2', 15]
12: ['A1', 'B1', 10]
```

23.7 Combination Testing

Available Tools

1. CATS (Constrained Array Test System) *)	[Sherwood] Bell Labs.	
2. OATS (Orthogonal Array Test System) *)	[Phadke] ATT	
3. AETG	Telecordia	Web-based, commercial
4. IPO (PairTest) *)	[Tai/Lei]	
5. TConfig	[Williams]	Java-applet
6. TCG (Test Case Generator) *)	NASA	
7. AllPairs	Satisfice	Perl script, free, GPL
8. Pro-Test	SigmaZone	GUI, commercial
9. CTS (Combinatorial Test Services)	IBM	Free for non-commercial use
10. Jenny	[Jenkins]	Command-line, free, public-domain
11. ReduceArray2	STSC, U.S. Air Force	Spreadsheet-based, free
12. TestCover	Testcover.com	Web-based, commercial
13. DDA *)	[Colburn/Cohen/Turban]	
14. Test Vector Generator		GUI, free
15. OA1	k sharp technology	
16. TESTONA	Assystem Germany	GUI, free for non-comercial use
17. AllPairs	[McDowell]	Command-line, free
18. Intelligent Test Case Handler (replaces CTS)	IBM	Free for non-commercial use
19. CaseMaker	Díaz & Hilterscheid	GUI, commercial
20. PICT	Microsoft Corp.	Command-line, open source at http://github.com/microsoft/pict
21. rdExpert	Phadke Associates, Inc.	
22. OATSGen *)	Motorola	
23. SmartTest	Smartware Technologies Inc.	GUI, commercial
24. EXACT *)	[Yan/Zhang]	
25. AllPairs	MetaCommunications	Free
26. ATD	AtYourSide Consulting	GUI, commercial
27. ACTS [formerly: FireEye]	NIST	GUI
28. Bender RBT Inc.	BenderRBT	GUI, commercial
29. Pairwise Test Case Generator	TestersDesk	Web-based
30. Combo-Test	The Australian eHealth Research Centre	Command-line, free
31. IPO-s *)	[Calvagna/Gargantini]	

https://pairwise.yuuniworks.com/

http://www.pairwise.org/tools.asp

https://github.com/thombashi/allpairspy

Take a break

Five minutes

投票 最多可选1项

设置

Are you ready?

- A Yes
- B No

提交

Review

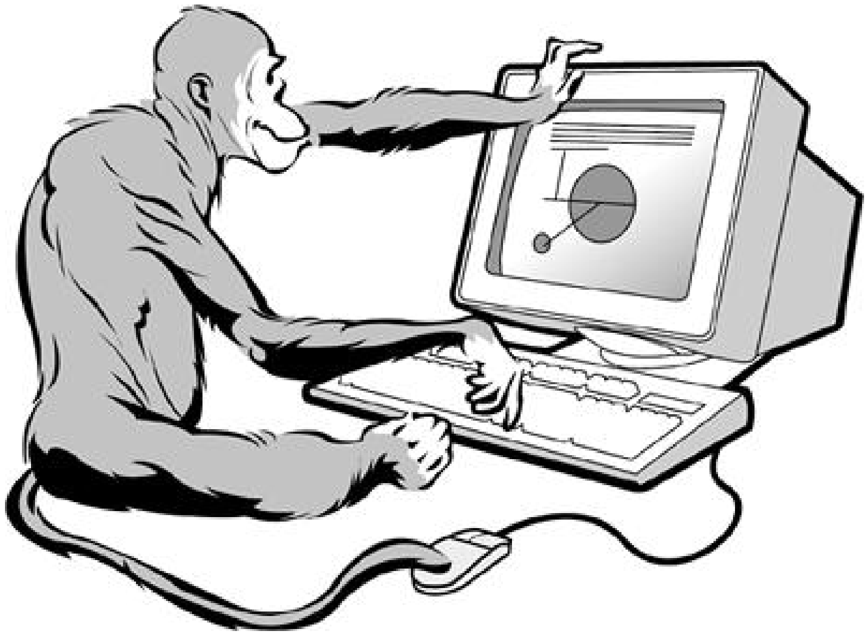
- ✓ Concept: "Good" testing, test oracle(expected result)
- ✓ **White-box testing**
  - statement coverage, branch coverage, decision coverage
  - branch/decision coverage, combination coverage, basic path coverage
  - Data flow testing, loop testing
- ✓ **Black-box testing**
  - Equivalence Partitioning, boundary value analysis
  - Cause-effect graph & Decision table
- ✓ Combination testing
- ✓ Usage-based testing (Operational Profile)

23.8 Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

23.8 Random Testing: Monkeys and Gorillas

- Test Monkey
  - The use of a test monkey to simulate how your customers will use your software in no way insinuates that computer users are related to apes.



Test monkeys will test forever as long as they have electricity and the

23.8 Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

23.8 Usage-Based Testing

Usage-Based Statistical Testing(UBST)

- *Reliability:* Probability of failure-free operation for a specific time period or a given set of input under a specific environment
  - ▷ Reliability: customer view of quality
  - ▷ Probability: statistical modeling
  - ▷ Time/input/environment: OP

- OP: Operational Profile
  - ▷ Quantitative characterization of the way a (software) system will be used.
  - ▷ Generate/execute test cases for UBST
  - ▷ Realistic reliability assessment
  - ▷ Development decisions/priorities

23.8 Usage-Based Testing

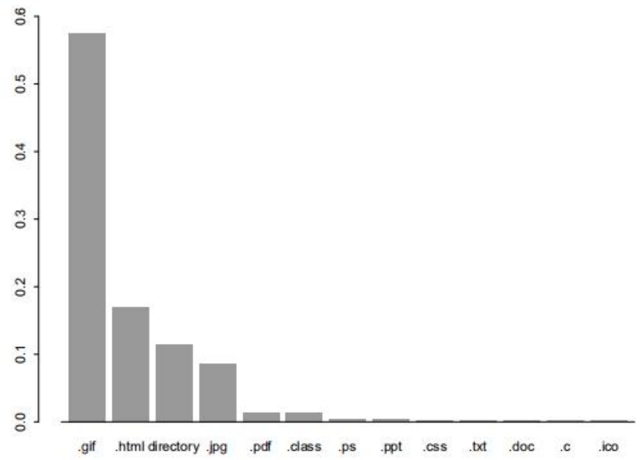
UBST General steps:

1. Information collection.
2. OP construction
3. UBST under OP.
4. Analysis (reliability!) and follow up.

23.8 Usage-Based Testing

- Example: Table 8.4, p.112
  - file type usage OP for SMU/SEAS

File type	Hits	% of total
.gif	438536	57.47%
.html	128869	16.89%
directory	87067	11.41%
.jpg	65876	8.63%
.pdf	10784	1.41%
.class	10055	1.32%
.ps	2737	0.36%
.ppt	2510	0.33%
.css	2008	0.26%
.txt	1597	0.21%
.doc	1567	0.21%
.c	1254	0.16%
.ico	849	0.11%
Cumulative	753709	98.78%
Total	763021	100%

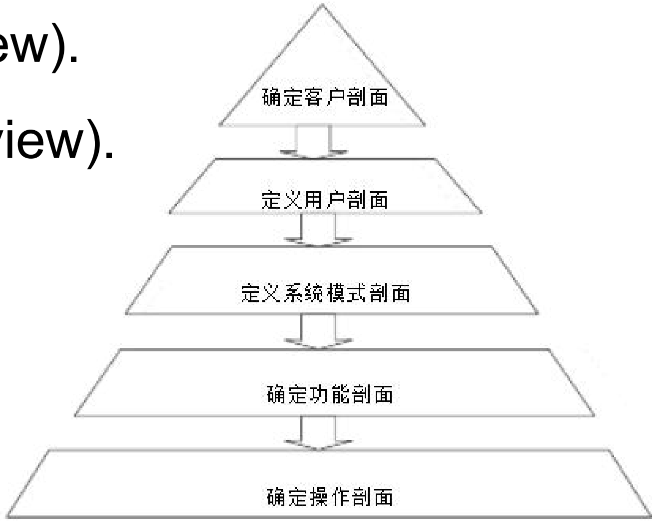


23.8 Usage-Based Testing

Generic steps for OP Development (Musa1)

(One OP for each homogeneous group of users or operations)

1. Find the customer profile (external view).
2. Establish the user profile (external view).
3. Define the system modes (internal view).
4. Determine the functional profile (internal view).
5. Determine the operational profile (internal view).



23.8 Usage-Based Testing

Generic steps for OP Development (Musa1)

1. Find the customer profile (external view).

Weight assignment:

. By #customers

. By importance/marketing concerns, etc.

Customer Type	Weight
corporation	0.5
government	0.4
education	0.05
other	0.05

23.8 Usage-Based Testing

Generic steps for OP Development (Musa1)

2. Establish the user profile (external view).

Weighting factor assignment for user weights within customer types:
. by users (equal usage intensity)
. by usage frequency
. other factors also possible

▷ customer profile used to calculate comprehensive user profile:
0.8×0.5 (com) $+ 0.9 \times 0.4$ (gov) $+$
0.9×0.05 (edu) $+ 0.7 \times 0.05$ (etc)
$= 0.84$

User Type	User Profile by Customer Type					Overall User Profile
	ctype	com	gov	edu	etc	
	weight	0.5	0.4	0.05	0.05	
end user		0.8	0.9	0.9	0.7	0.84
dba		0.02	0.02	0.02	0.02	0.02
programmer		0.18	–	–	0.28	0.104
third party		–	0.08	0.08	–	0.036

23.8 Usage-Based Testing

Generic steps for OP Development (Musa1)

3. Define the system modes (internal view).

- System mode
  - A set of functions/operations
  - For operational behavior analysis
  - Practicality: expert for system mode
- Example modes
  - Business use mode
  - Personal use mode
  - Attendant mode
  - System administration mode
  - Maintenance mode
  - Probabilities (weighting factors)

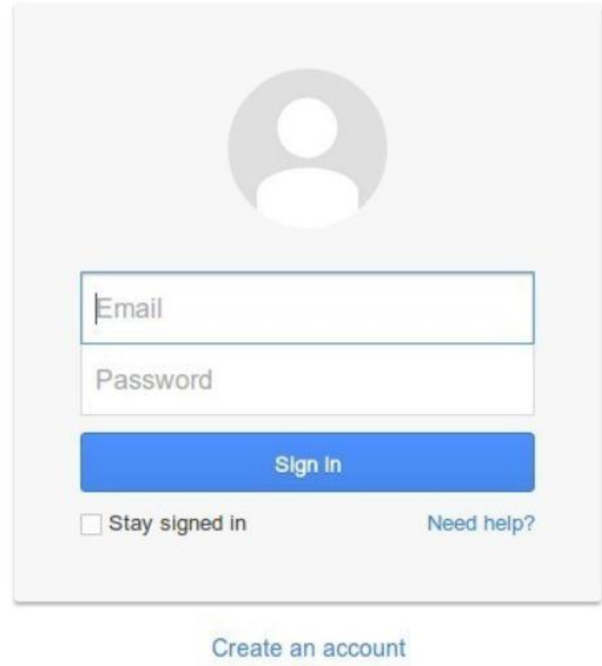
23.8 Usage-Based Testing

Generic steps for OP Development (Musa1)

4. Determine the functional profile (internal view).
   - ❑ Identifying functions
   - ❑ Creating Function list
   - ❑ Determining occurrence probabilities

5. Determine the operational profile (internal view).
   - ❑ Refining functional profile into OP
   - ❑ Defining operations
     - Partitioning input space into operations
   - ❑ Obtaining occurrence probabilities

Exercise



Gmail Login Screen

Test Requirement:

1. Username should contain letter and number.
2. Username should be at least 6 characters.
3. Username should not be more than 40 characters.
4. Username should not contain spaces.
5. Password should contain combination of letter and numbers.
6. Password should not contain spaces.
7. Password should be at least 6 characters.
8. Password should not be more than 40 characters.

Exercise - black-box testing

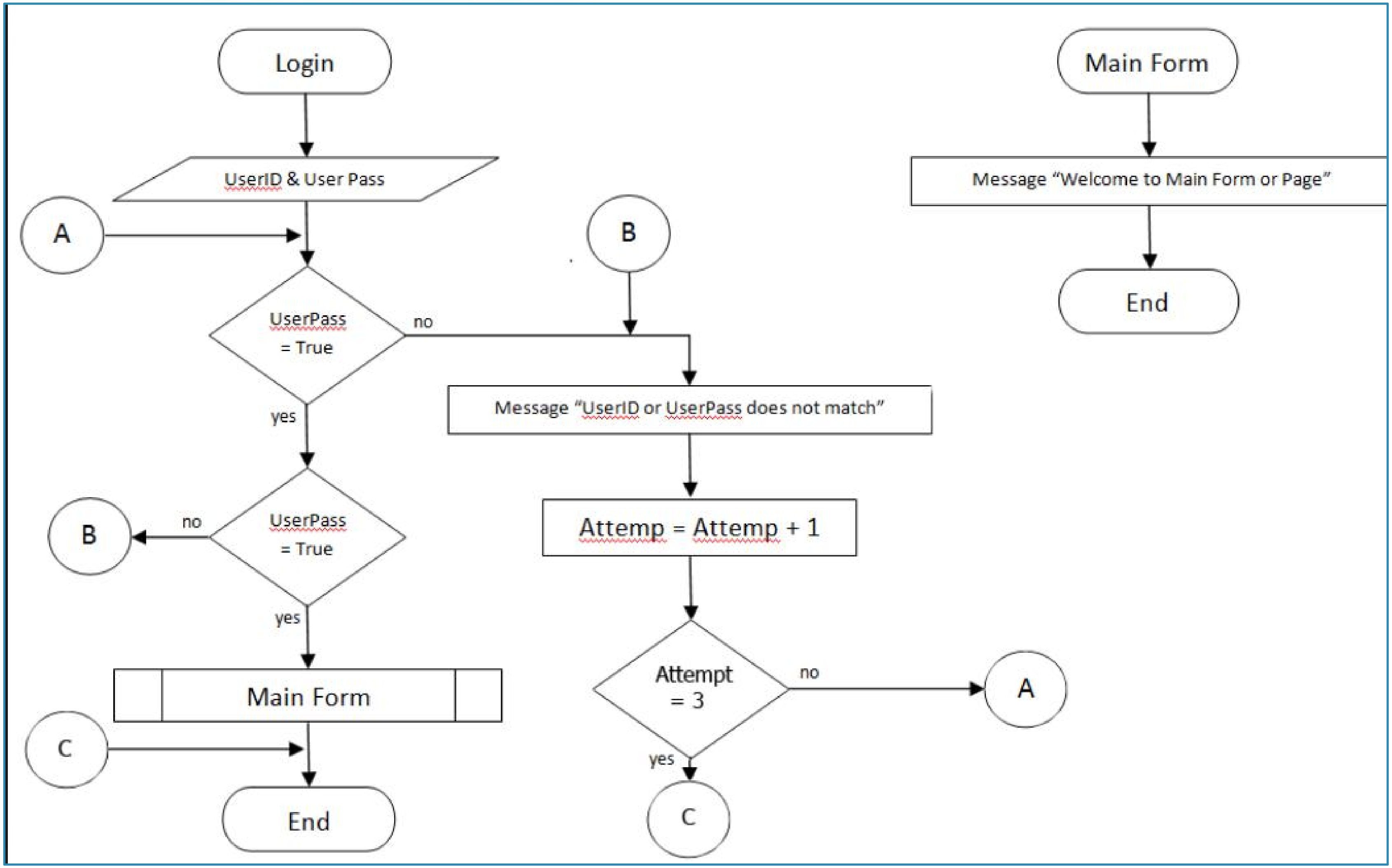
- **Positive test cases.**

1. Enter valid username and password.
2. Click on forgot password link and retrieve the password for the username.
3. Click on register link and fill out the form and register username and password.
4. Use enter button after typing correct username and password.
5. Use tab to navigate from username textbox to password textbox and then to login button.

- **Negative test cases**

1. Enter valid username and invalid password.
2. Enter valid password but invalid username.
3. Keep both field blank and hit enter or click login button.
4. Keep username blank and enter password.
5. Keep password blank and enter username.
6. Enter username and password wrong.

Exercise - white-box testing



Summary

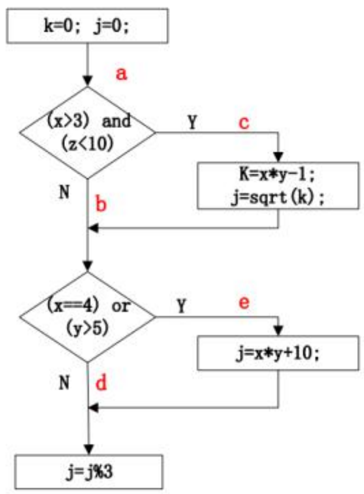
- ✓ Concept: “Good” testing, test oracle(expected result)
- ✓ **White-box testing**
  - statement coverage, branch coverage, decision coverage
  - branch/decision coverage, combination coverage, basic path coverage
  - loop testing, Data flow testing
- ✓ **Black-box testing**
  - Equivalence Partitioning, boundary value analysis
  - Cause-effect graph & Decision table
- ✓ Combination testing
- ✓ Usage-based testing (Operational Profile)

Assignment 4

Deadline: April 17

Q1: White-box testing

```
void DoWork (int x, int y, int z)
{
    int k=0,j=0;

    if((x>3) && (z<10))
    {
        k=x*y-1;
        j=sqrt(k);      //block 1
    }

    if((x==4) || (y>5))
    {
        j=x*y+10;      //block 2
    }

    j=j%3;            //block 3
}
```



Design testcases with the following techniques:

1) statement coverage
2) branch coverage
3) decision coverage
4) decision/branch coverage
5) decision combination coverage
6) basic path coverage

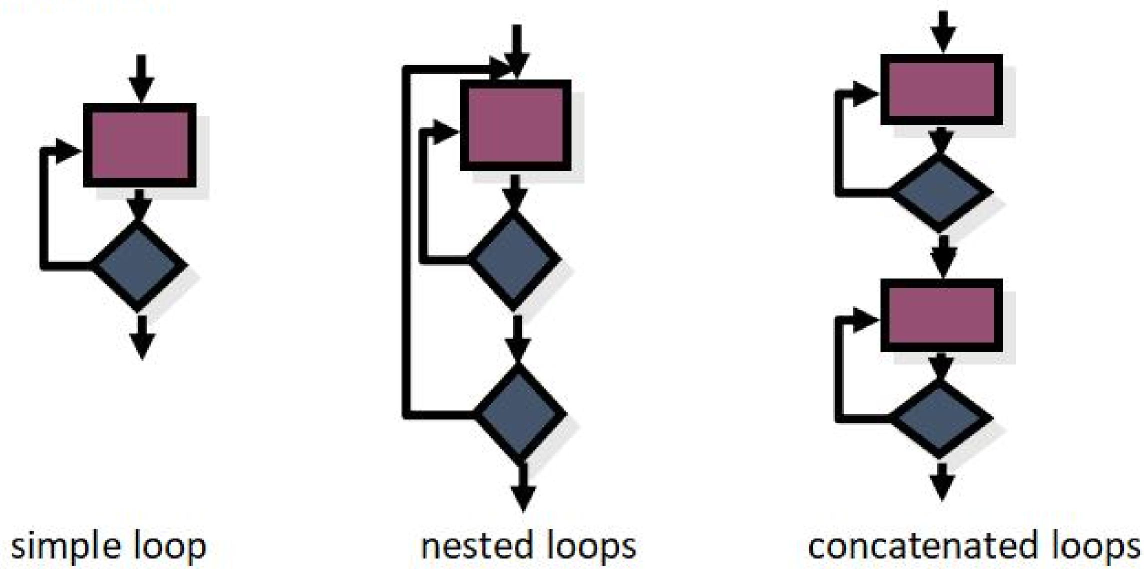
program: input: x, y, z ; output: k,j

Design six test cases tables for each coverage technique.

Assignment 4

Q2: White-box testing

Please write three pieces of code (simple loop, nested loops, concatenated loops), then design test case for them.



Assignment 4

Q3: Black-box testing

A program accepts as input three integers which it interprets as the lengths of sides of a triangle. It reports whether the triangle is equilateral, isosceles, or scalene (neither equilateral nor isosceles).

Design test cases with the following techniques:

1) Equivalence Class Partitioning :

    List valid and invalid equivalence classes you designed

2) Boundary Value Analysis

    List all boundary conditions what you can consider

THE END