Module 4 – Topic 4.2 – Lesson 4.2.4

White Box Testing

Lesson Outline

- Control flow graph
- Path testing
  - Test coverage – statement, branch, condition, loop, path
- Basis path testing
  - What is basis path
  - Cyclomatic complexity
  - McCabe's baseline method
- Guidelines and observations

White Box Testing

- Test **structure** of the software

- Explicit knowledge of the *internal structure* of the SUT
  - Based on the internal structure of the SUT
  - Generally require detailed programming skills

- Also know as path testing

Recall The Impossibility of Test Everything

```
func() {
    while (a) {
        if(b) {
            …
        } else {
            if(c) {
                …
            }
        }
        if(d) {
            …
        }
    }
    …
}
```

If the loop has 10 repetitions, how many distinct program execution paths are there?

- Can you execute all execution paths at lease once?

Executing all execution paths is in general infeasible!

- Every decision point doubles the number of paths (i.e., $2^{|decision|}$)
- Every loop powers the paths by the number of iterations

White box testing

- Control flow graph
- Control flow testing

White Box Testing – Key Idea

- Abstraction is the key
  - Programs $\rightarrow$ mathematical objects (e.g., program graphs)
- Analyse the abstract program representation
  - control flow, data flow, converge, etc.

- Amenable to rigorous definitions, mathematical analysis, and useful measurement
  - e.g., do we cover all statements, all branches, or all paths?

The General White Box Testing Process

1. The SUT's implementation is analysed
    - control flow graph
2. Paths through the SUT are identified
    - according to minimum test coverage criteria
3. Inputs are chosen to cause the SUT to execute the selected paths. This is called path sensitization.
4. Expected results for those inputs are determined.

5. Tests are run
6. Actual outputs are compared with the expected outputs
7. A determination is made as to the proper functioning of the SUT

Control Flow Graph

Control Flow Graph

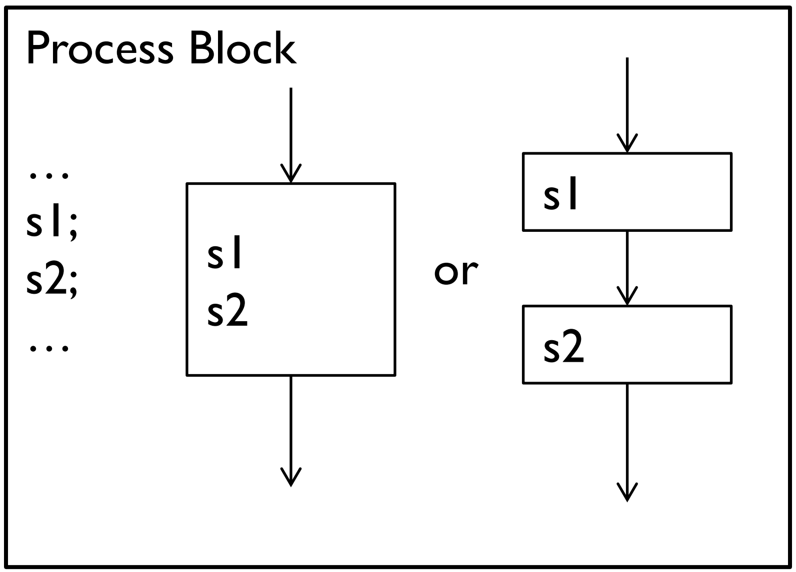
- A directed graph in which
  - nodes: statement fragments
  - edges: flow of control

The foundation of control flow testing

- Three types of nodes
  - process block, decision point, junction point

- Flow of control
  - An edge from node i to j: the statement fragment corresponding to node j can be executed immediately after the statement corresponding to node i

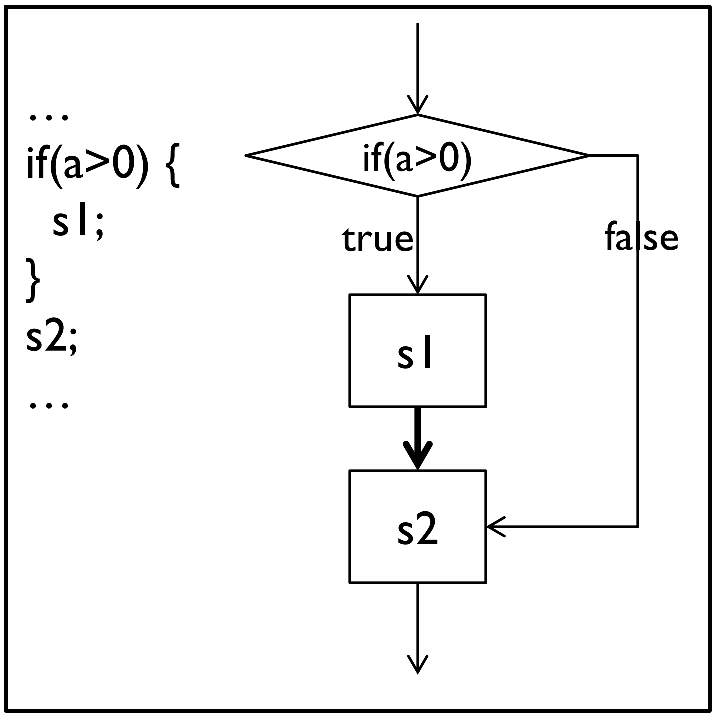
Process Block



NOTE: s1 and s2 are a simple non-conditional, non-loop statement. Same for the following slides.

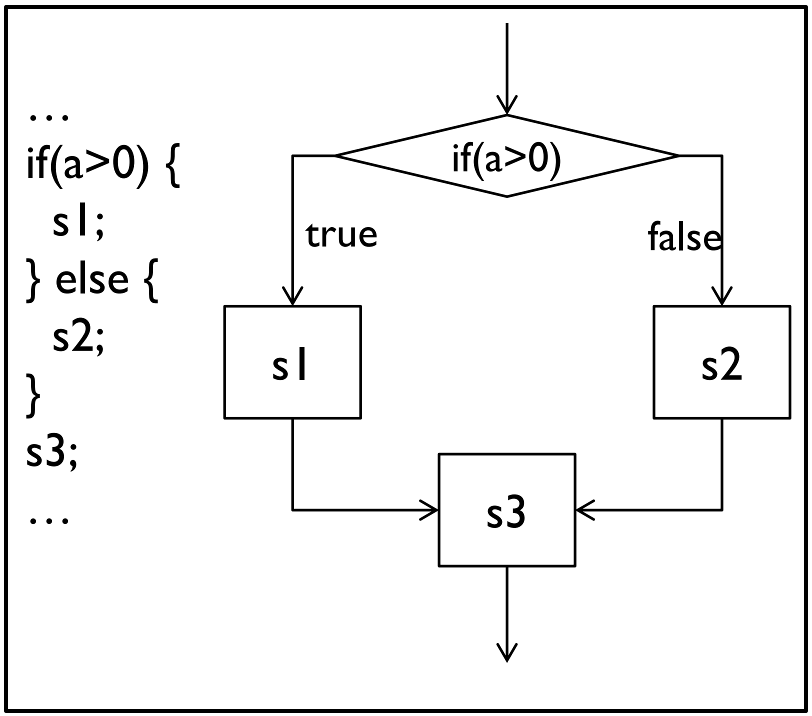
- A sequence of statements execute sequentially from beginning to end
- Do not contain decision points (if/while/for/switch statements)
- Any number of statements in the process block
- One control flow edge into the process block
- One control flow edge out of the process block

Binary Decision Point – if <condition>



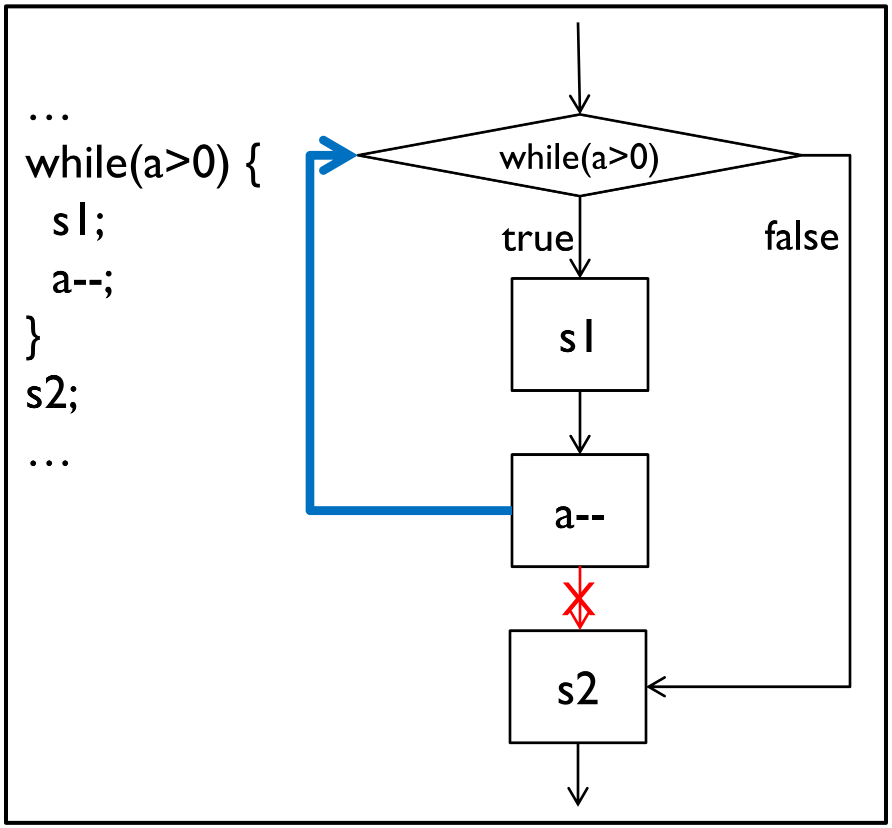
- Control flow can change at the decision point
- One control flow edge into the decision point
- Binary - two control flow edges out of the decision point
  - True branch
  - False branch
- Junction point at which control flows join together
  - e.g., s2 is a junction point

Binary Decision Point – if <condition> else



- Control flow can change at the decision point
- One control flow edge into the decision point
- Binary - two control flow edges out of the decision point
  - True branch
  - False branch
- Junction point at which control flows join together
  - e.g., s3 is a junction point

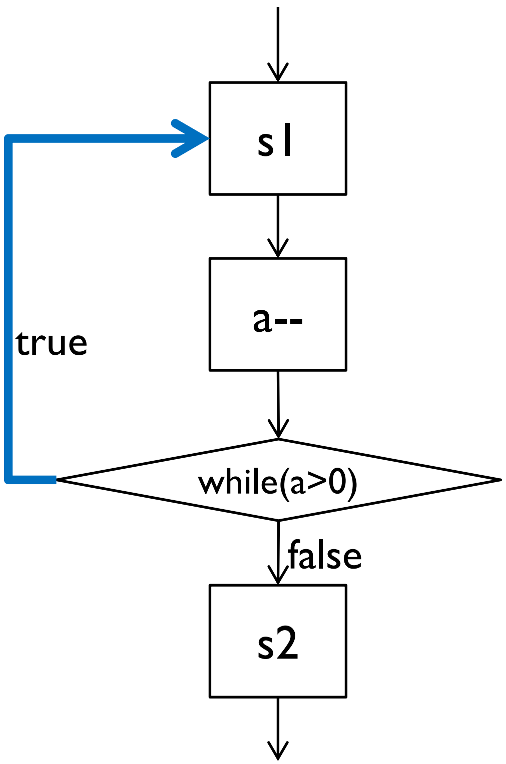
Binary Decision Point – Pretest while <condition>



- Control flow can change at the decision point
- One control flow edge into the decision point
- **Binary** - two control flow edges out of the decision point
  - True branch
  - False branch
- **Loop back** to the decision point from the last statement of repeated body

Binary Decision Point – Posttest while <condition>

```
...
do {
  s1;
  a--;
} while(a>0)
s2;
...
```



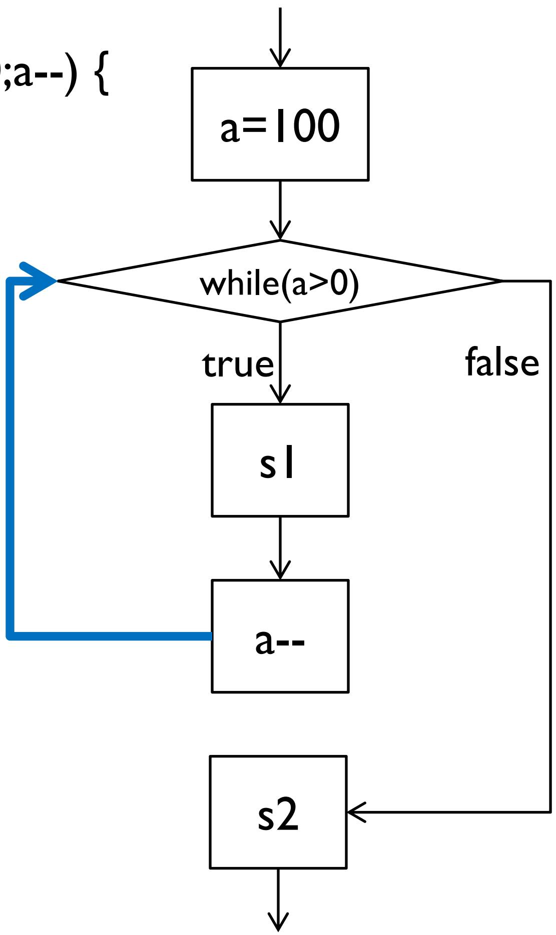
- Control flow can change at the decision point
- One control flow edge into the decision point
- Binary - two control flow edges out of the decision point
  - True branch
  - False branch
- **Loop back** from the decision point to the first statement of repeated body

Binary Decision Point – for <condition>

```
…
for(a=100;a>0;a--) {
  s1;
}
s2;
…
```

transform
Into

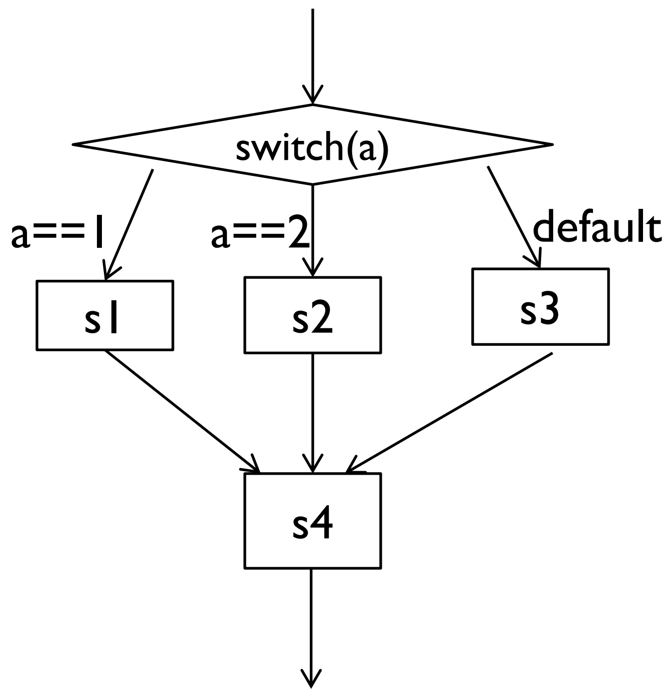
```
…
a=100;
while(a>0) {
  s1;
  a--;
}
s2;
…
```



- Control flow can change at the decision point
- One control flow edge into the decision point
- Binary - two control flow edges out of the decision point
  - True branch
  - False branch
- **Loop back** to the decision point from the last statement of repeated body

N-ary Decision Point – switch <condition>

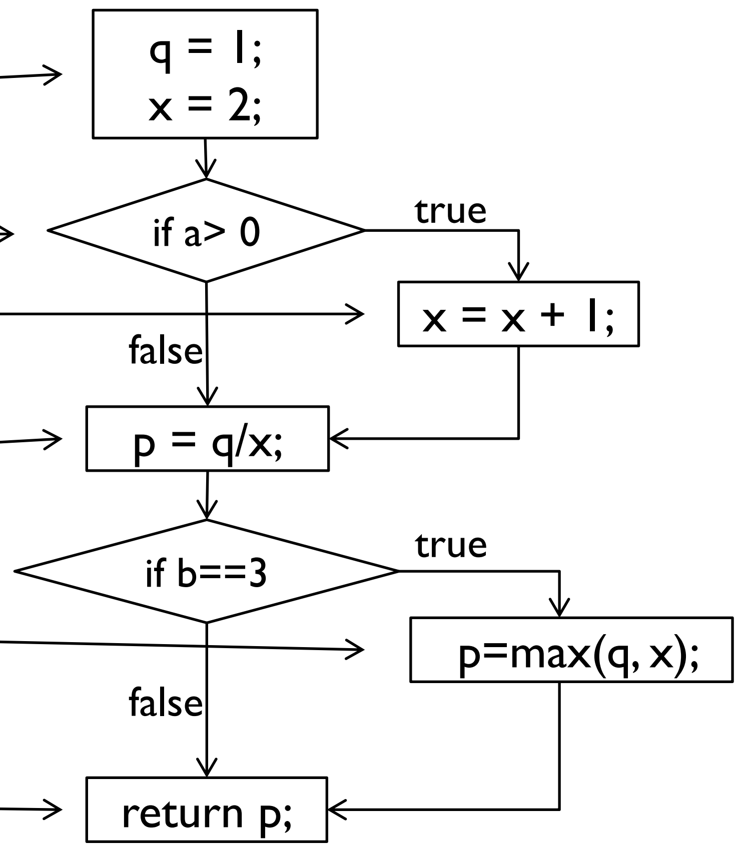
```
…
switch(a) {
  case 1:
    s1;
    break;
  case 2:
    s2;
    break;
  default:
    s3;
}
s4;
…
```



- Control flow can change at the decision point
- One control flow edge into the decision point
- N control flow edges out of the decision point
- Junction point at which control flows join together
  - e.g., s4 is a junction point

Control Flow Graph – Example

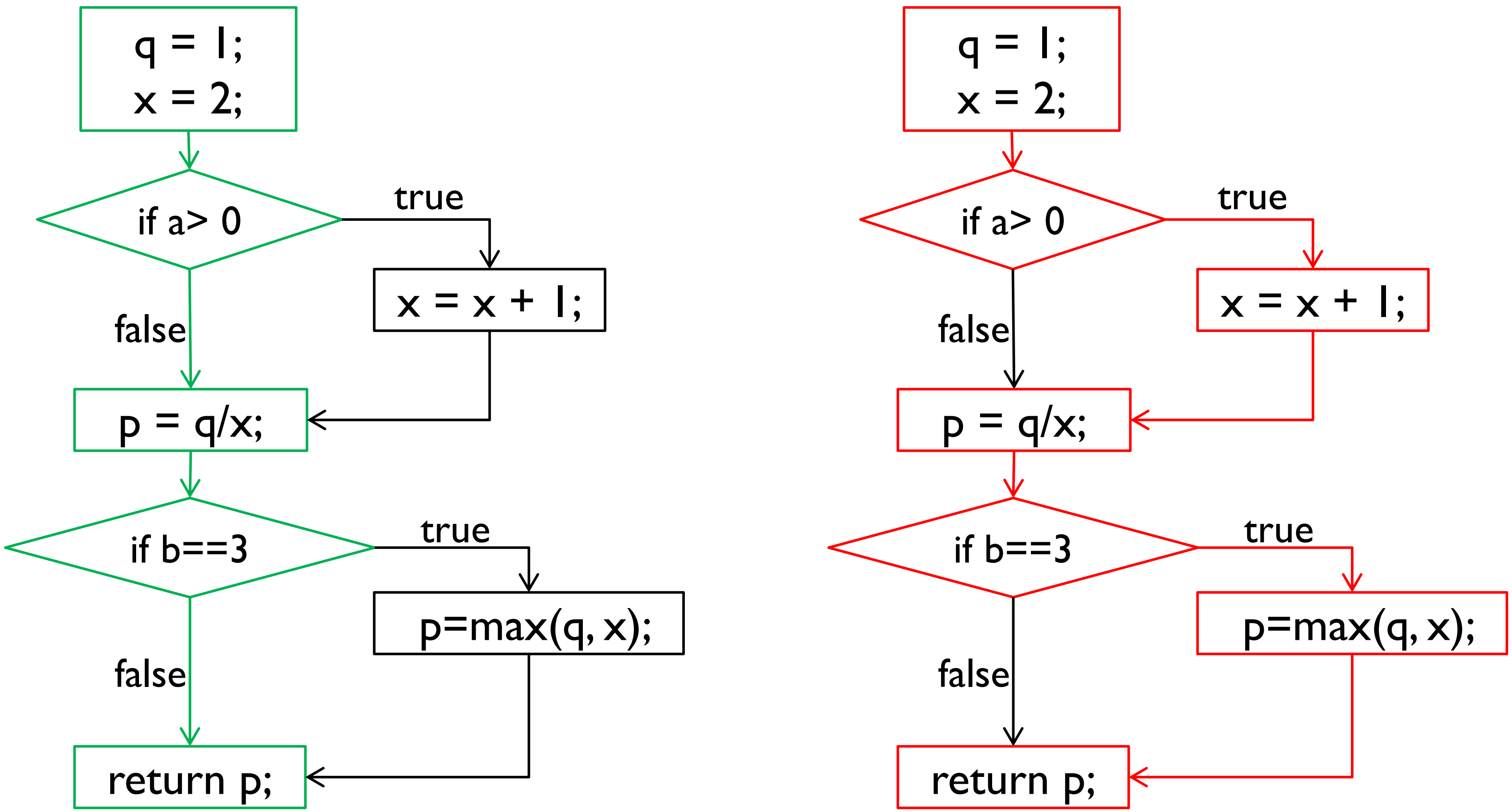
```
int computeP(int a, int b) {
    int q, x, p;
    q = 1;
    x = 2;
    if(a > 0) {
        x = x + 1;
    }
    p = q/x;
    if(b == 3) {
        p = max(q, x);
    }
    return p;
}
```



Path Testing

Execution Path Through the CFG

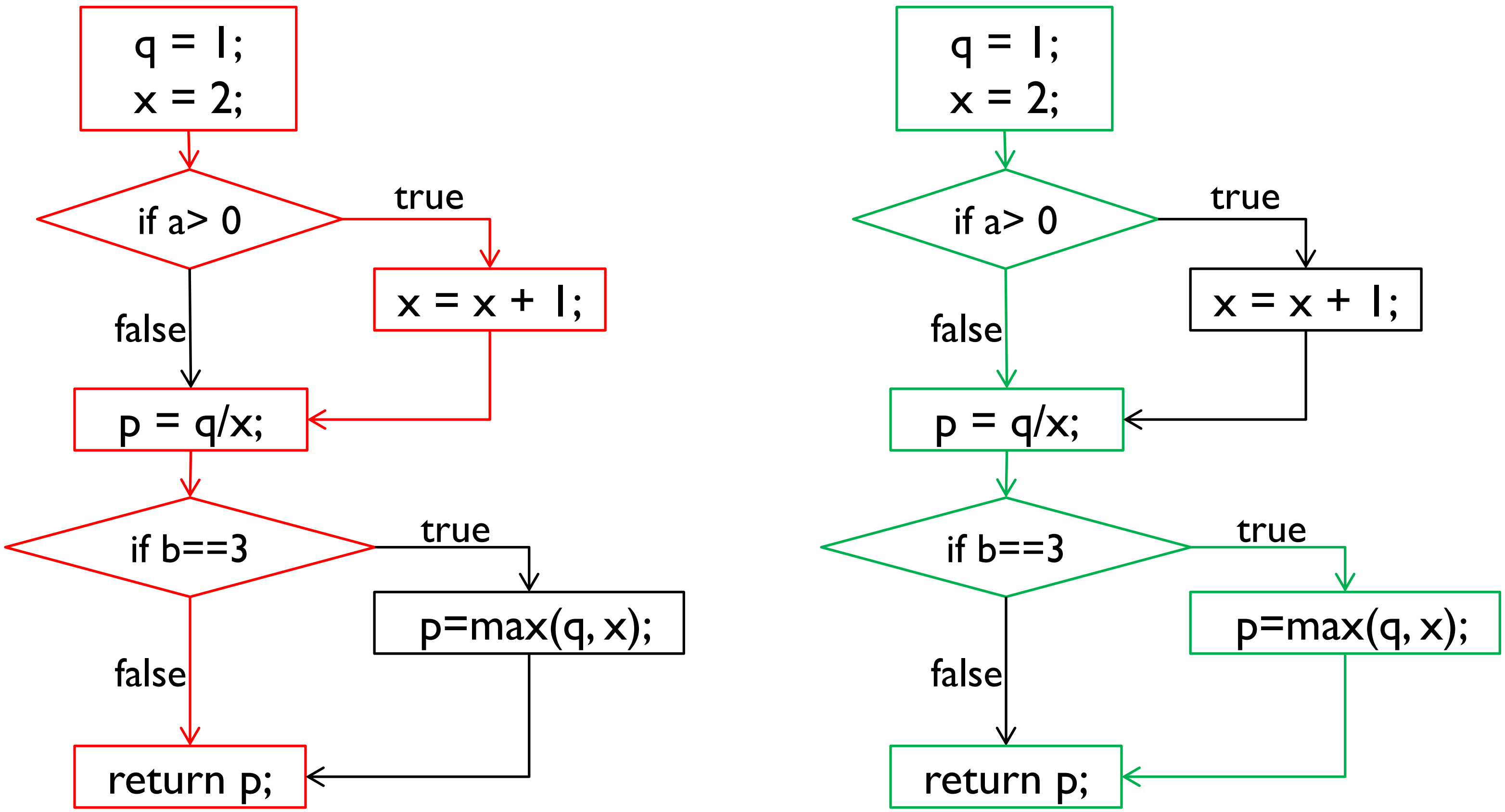
- A sequence of adjacent edges through the CFG



Two execution paths of compute(int, int)

Execution Path Through the CFG

- A sequence of adjacent edges through the CFG



Another two execution paths of compute(int, int)

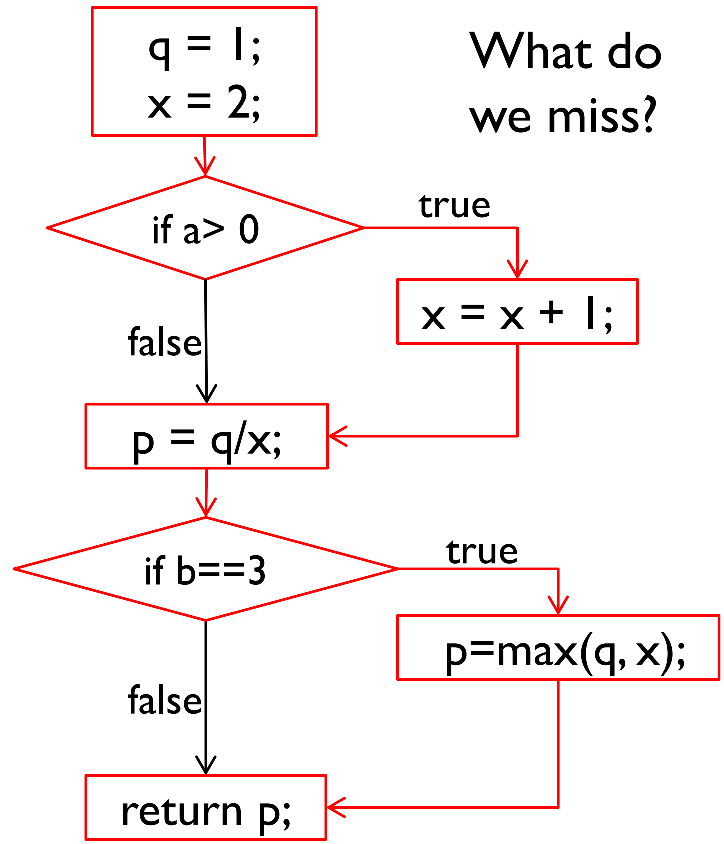
Test Coverage Criteria

- Test efficiency: redundancy & gap – can be measured by test coverage
- Test coverage can be defined on control flow graph
  - The percentage of the code that has been tested vs. that which is there to test

Level 1 100% statement coverage	Every statement is executed at least once
Level 2 100% branch coverage	Every branch is executed at least once
Level 3 100% condition coverage	Every condition has a TRUE and FALSE at least once
Level 4 100% multiple condition coverage	Use the knowledge of how the compiler actually evaluates the multiple conditions
Level 5 Loop coverage	At least execute loop zero times and one time
Level 6 100% path coverage	Feasible only for code without loops

Level 1 – 100% Statement Coverage

- Select execution path(s) to cover **all the CFG nodes** at least once



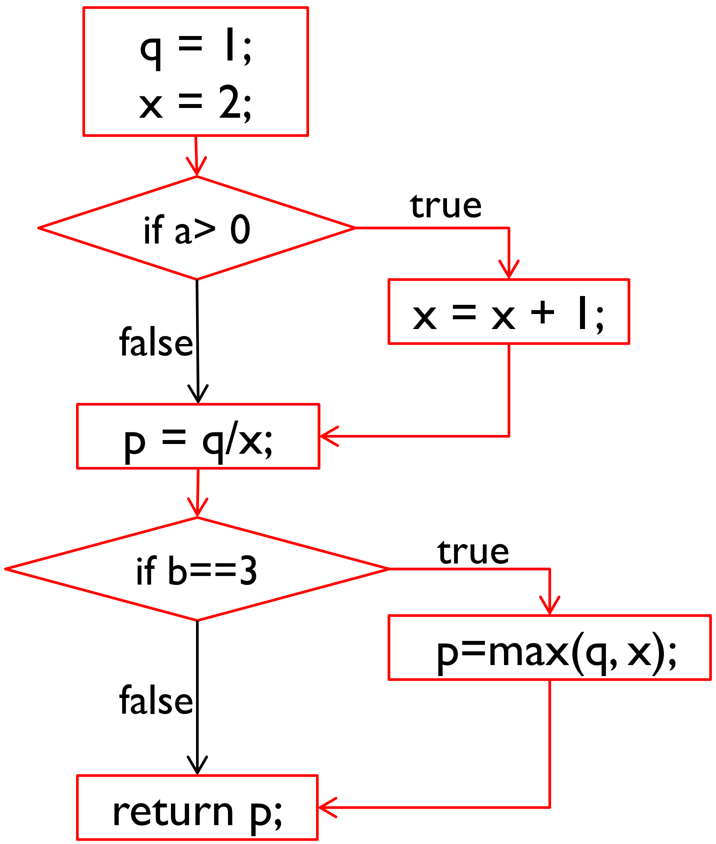
What do we miss?

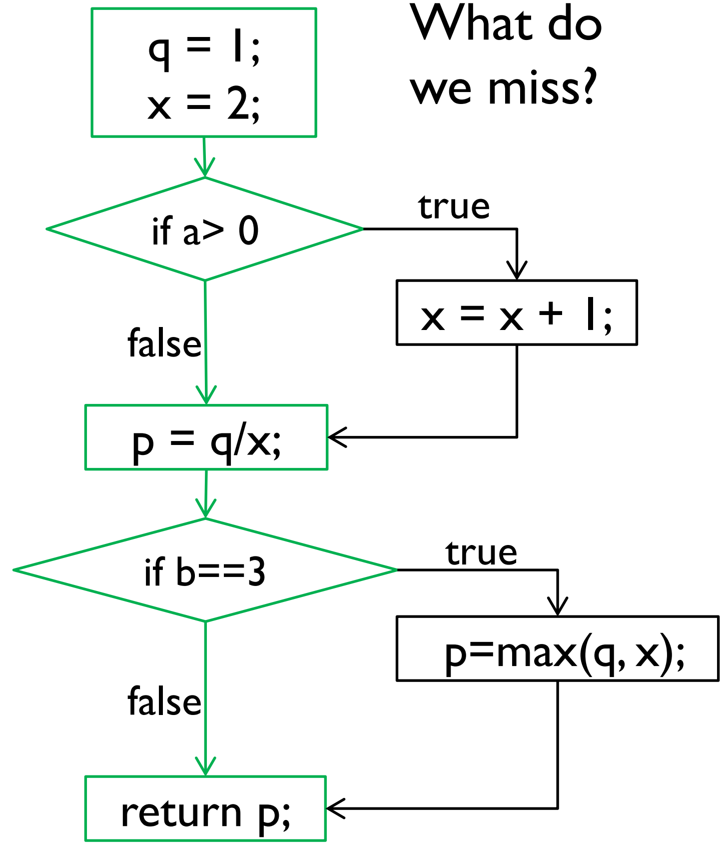
Statement coverage is generally not an acceptable level of testing.

Testing less than 100% statement coverage for new software is unconscionable and should be criminalized.

Level 2 – 100% Decision (Branch) Coverage

- Every **decision** that has a **TRUE** and **FALSE** outcome is evaluated at lease once

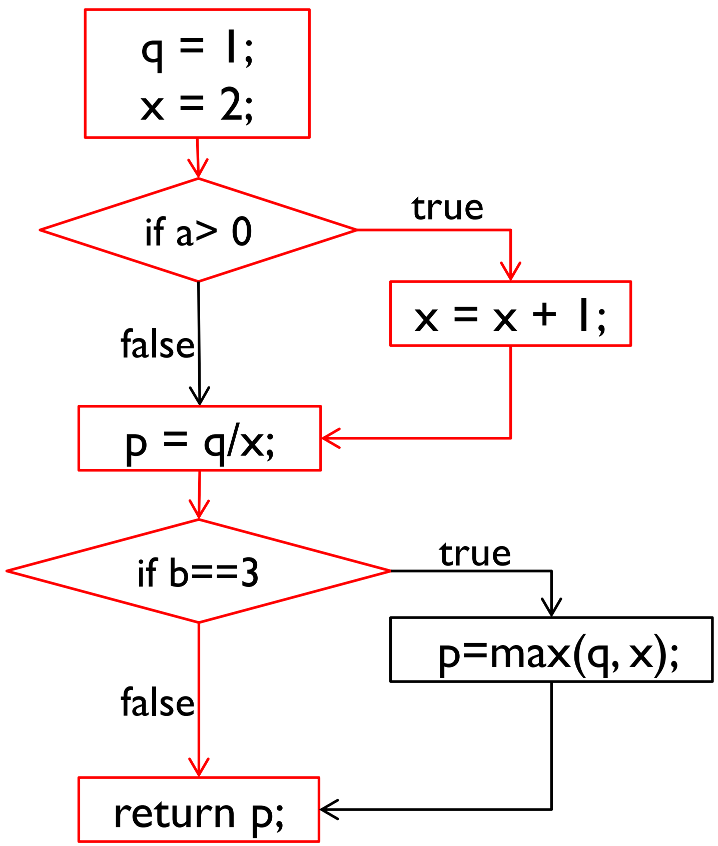


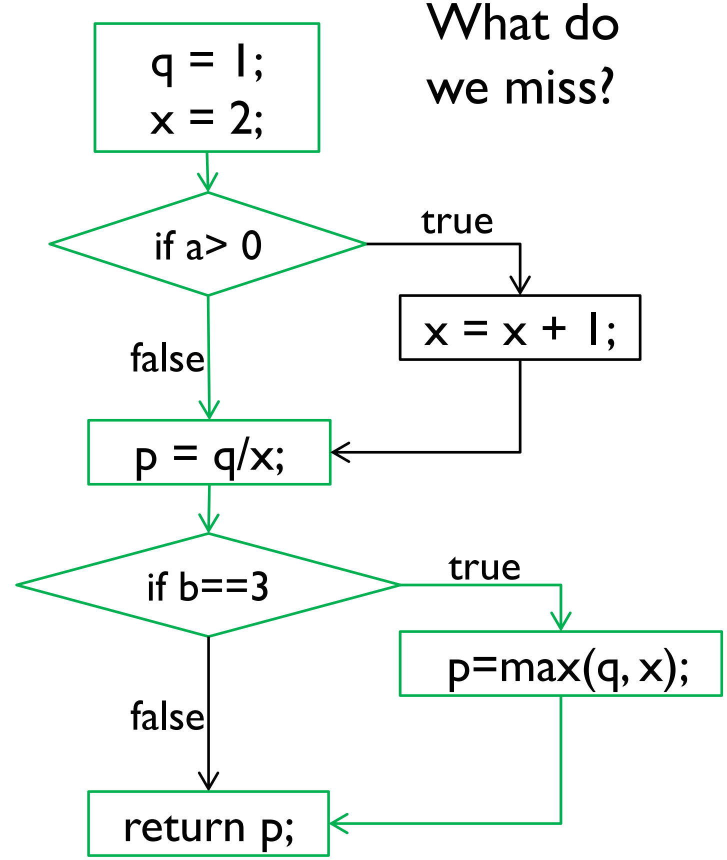


What do we miss?

Level 2 – 100% Decision (Branch) Coverage

- Every **decision** that has a **TRUE** and **FALSE** outcome is evaluated at lease once

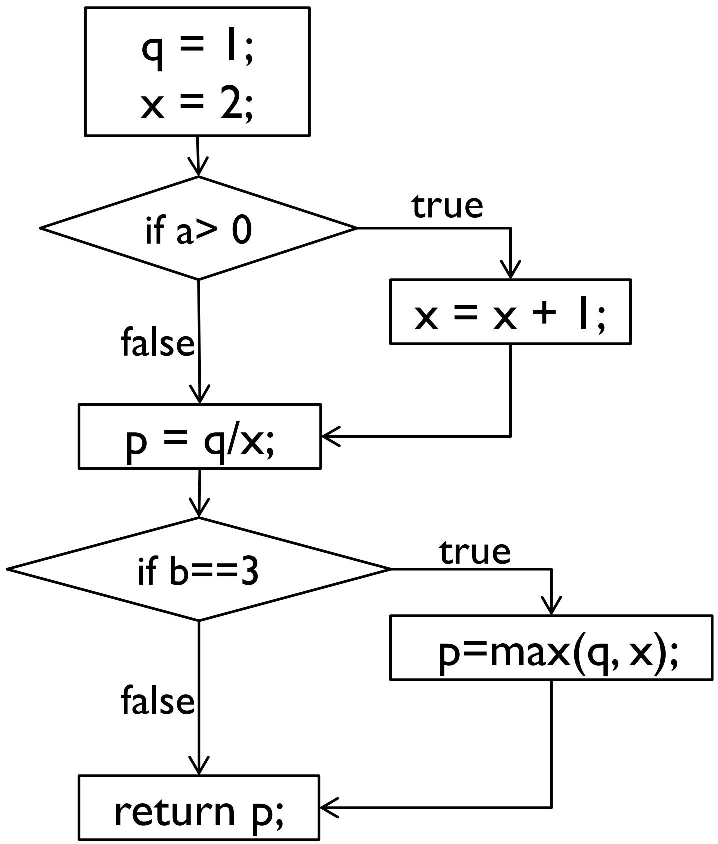




What do we miss?

Level 6 – 100% Path Coverage

- Create test cases such that **all execution paths** are executed at least once



Feasible only for code without loops. But if the code has too many decision points, there can still be a large number ($2^{|decision|}$) of execution paths.

For code with loops, the number of paths can be enormous and thus pose an intractable testing problem.

If the left program loops 10 times, we will have 2^{2*10}= 1,048,675 paths.

Level 5 – Loop Coverage

- Execute the loop zero times (min)
- Execute the loop one time (mim+)
- Execute the loop n times where n is a small number representing a typical loop value (norminal)
- Execute the loop its maximum number of times m (if known) (max).
- In addition you might try m-1 (max-) and m+1 (max+).

Consider the times of loop execution as a bounded quantity. Apply BVT principle to test loop.

Focus on only executing the loop how many times, but not the coverage of execution paths in the loop.

Level 3 – 100% Condition Coverage

- Each condition that has a TRUE and FALSE outcome that makes up a decision is evaluated at least once

```
if(x&&y) {
  conditionedStatement;
}
```

Achieve condition coverage with two test cases:

- x=TRUE, y=FALSE
- x=FALSE, y=TRUE

Do you see any problems?

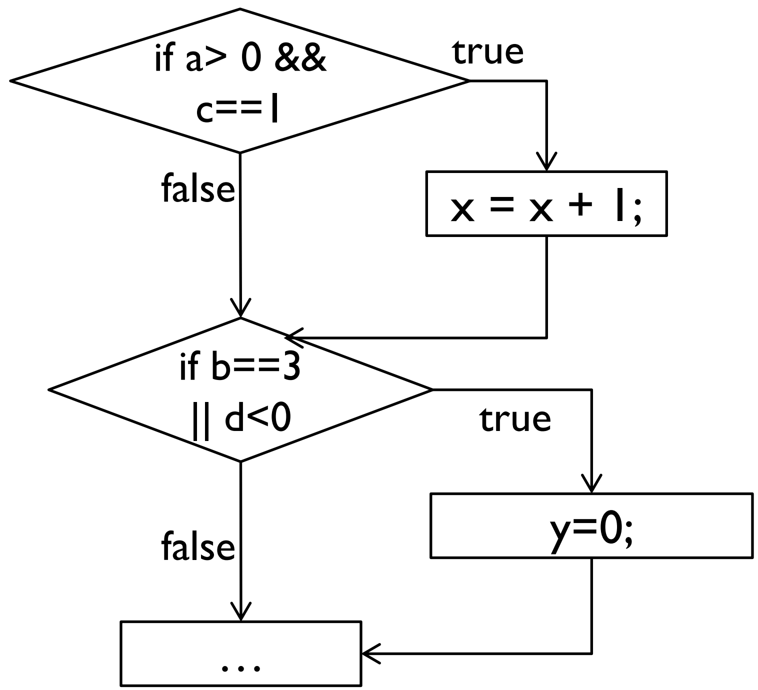
100% Condition Coverage may not achieve decision coverage!

With these choices of data values the conditionedStatement will never be executed (i.e., decision x&&y never be true)

Level 4 – 100% Multiple Condition Coverage

- Use the knowledge of how the compiler actually evaluates the multiple conditions to create test cases

```
if (a>0 && c==1) {
  x=x+1;
}
if (b==3 || d<0) {
   y=0;
}
…
```



Can you "compile" it into a more detailed CFG?

Basis Path Testing

Also known as Structured Testing

Motivation

- Find a minimum set of basis paths, all other execution paths are linear combinations of basis paths

- If the basis paths are okay, we could hope that everything that can be expressed in terms of the basis paths is also okay

Guarantee 100% statement and branch coverage

Basis Paths

- For a CFG viewed as a vector space V of all paths, find a subset B of V such that every element in V can be represented as a linear combination of elements in B
  - Vector space: rows – paths, columns – CFG edges, cell - #times an edge is traversed by a path
  - Elements in B are basis paths
  - Linear combination – linear vector operations
    - Addition/subtraction: path (un)concatenation, e.g., P3 = P1 + P2
    - Multiplication by a number: path repetition, e.g., P4 = P1 * 2

	E1	E2	E3	E4	E5	E6	E7
P1	1	0	2	0	0	1	1
P2	0	0	1	2	2	0	1
P3	1	0	3	2	2	1	2
P4	2	0	4	0	0	2	2

Basis Path Testing Process

1. Derive the control flow graph from the software
2. Compute the graph's Cyclomatic Complexity (C)
3. Select a set of C basis paths

4. Inputs are chosen to cause the SUT to execute the selected paths. This is called path sensitization.
5. Expected results for those inputs are determined.

6. Tests are run
7. Actual outputs are compared with the expected outputs
8. A determination is made as to the proper functioning of the SUT

Cyclomatic Complexity of a CFG

- Measures software complexity and testability in terms of the number of decision points (i.e., if/while/for/switch) in the CFG

- The more decision points (i.e., if/while/for/switch) a program has → The higher the CC → The more basis paths you need to test

Cyclomatic Complexity - Implications

- Cyclomatic complexity determines the number of basis paths to be tested

Cyclomatic number	Type	Risk
1-10	A simple, well structured program	Low
11-20	A reasonably complex program	Moderate
21-50	A complex program	High
>50	An extremely complex, untestable program	Very high

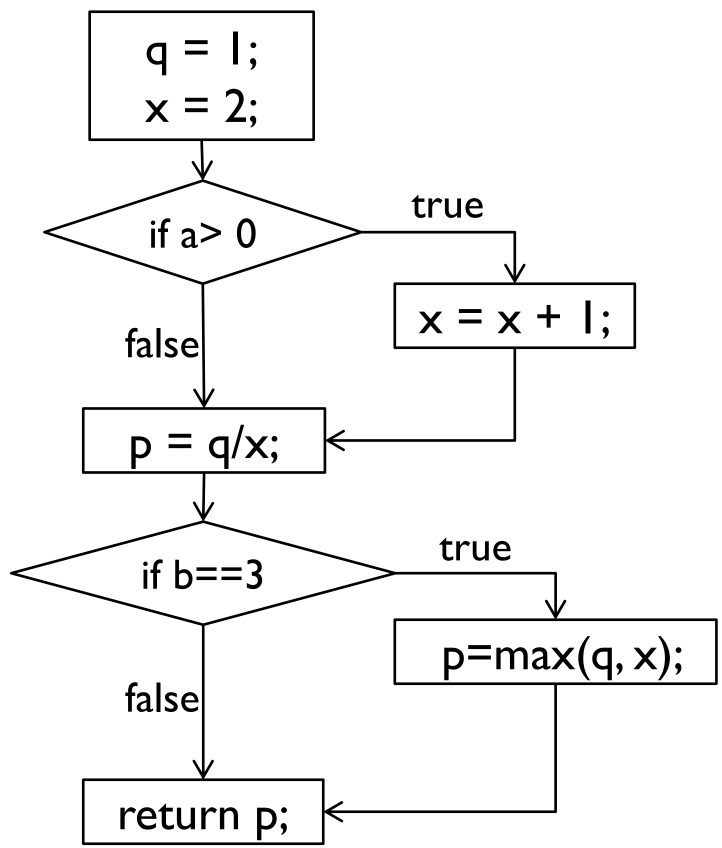
Compute Cyclomatic Complexity

- C = |edges| - |nodes| + 2
- C = |decisionpoint| + 1, if all decision points are binary (i.e., one true branch + one false branch)

The two equations compute the same value if all decision points are binary

Cyclomatic Complexity – Example

```
int computeP(int a, int b) {
    int q, x, p;
    q = 1;
    x = 2;
    if(a > 0) {
      x = x + 1;
    }
    p = q/x;
    if(b == 3) {
      p = max(q, x);
    }
    return p;
}
```



|edges| = 8
|nodes| = 7
CC = 8 – 7 + 2 = 3

Or
|decisionpoint| = 2
CC = 2 + 1 = 3

Recall:
1 path for statement coverage
2 paths for branch coverage
3 basis paths
4 paths for path coverage

How to Determine a Set of Basis Paths?

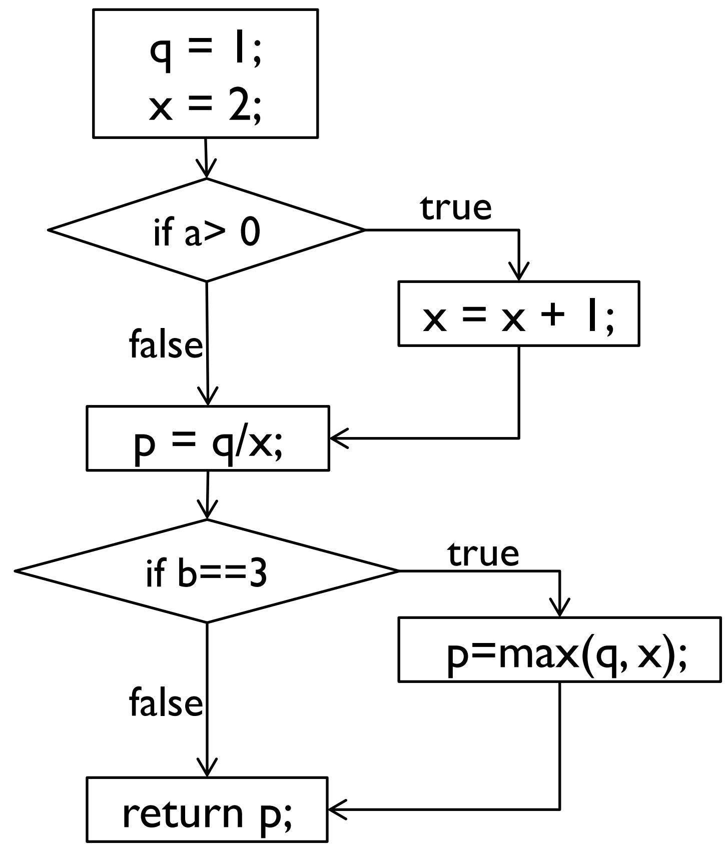
- The McCabe's baseline method
  1. Choose a baseline path (e.g., some "normal case" program execution)
  2. Retrace the baseline path and flip previous decisions one at a time
  3. Repeat until all decisions have been flipped

The first step can be somewhat arbitrary; McCabe advises choosing a path with as many decision points as possible

"flip" means when a node of outdegree ≥ 2 (i.e., a decision point) is reached, a different edge must be taken

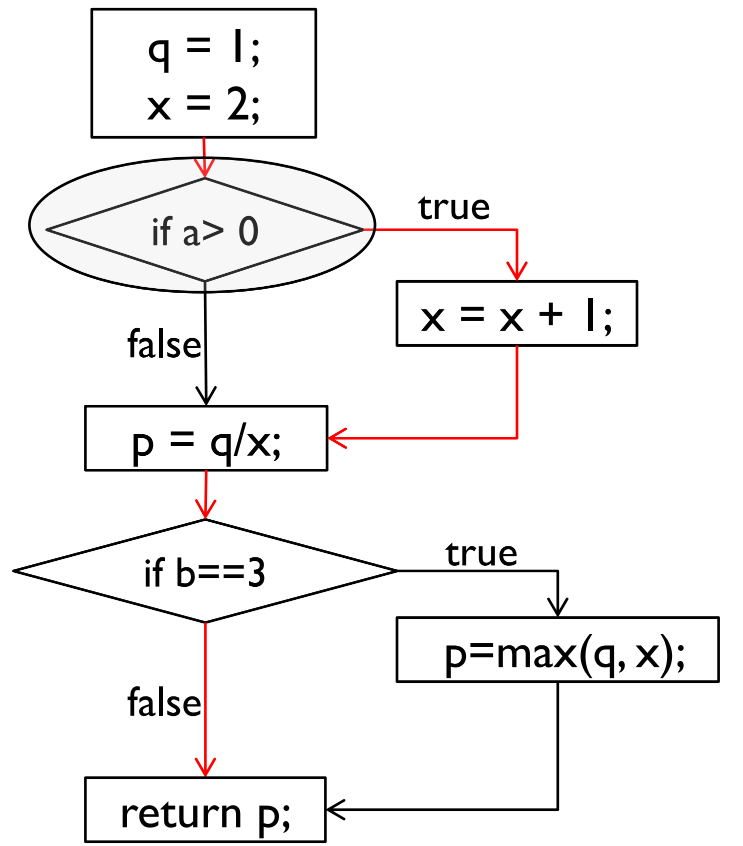
Choose the Baseline Path

- "Typical" path of execution
- Most important path from the tester's view



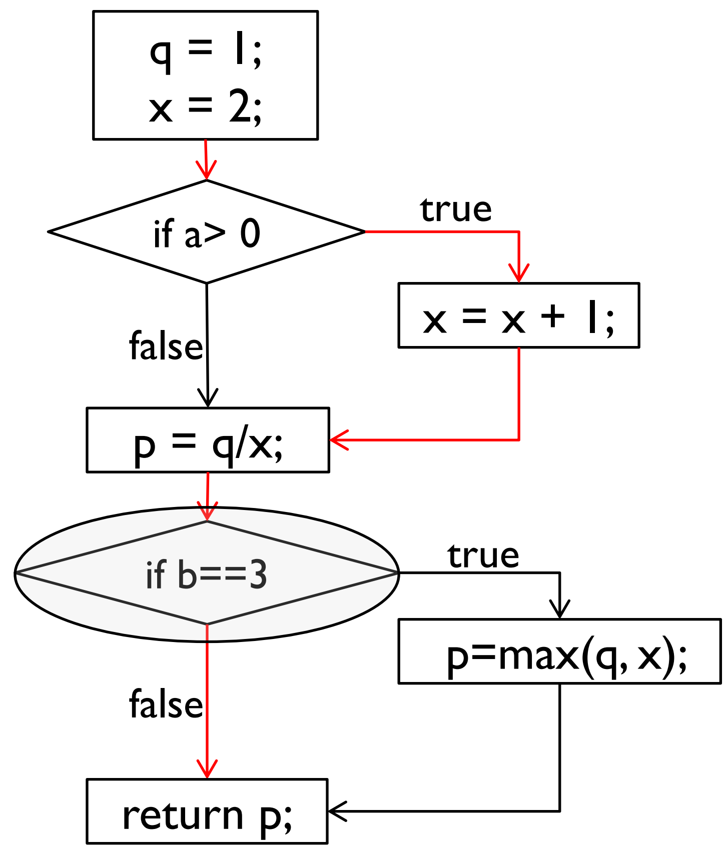
Retrace and Flip Previous Decisions

- Flip the outcome of the first decision point
- Keep the maximum number of other decision points the same

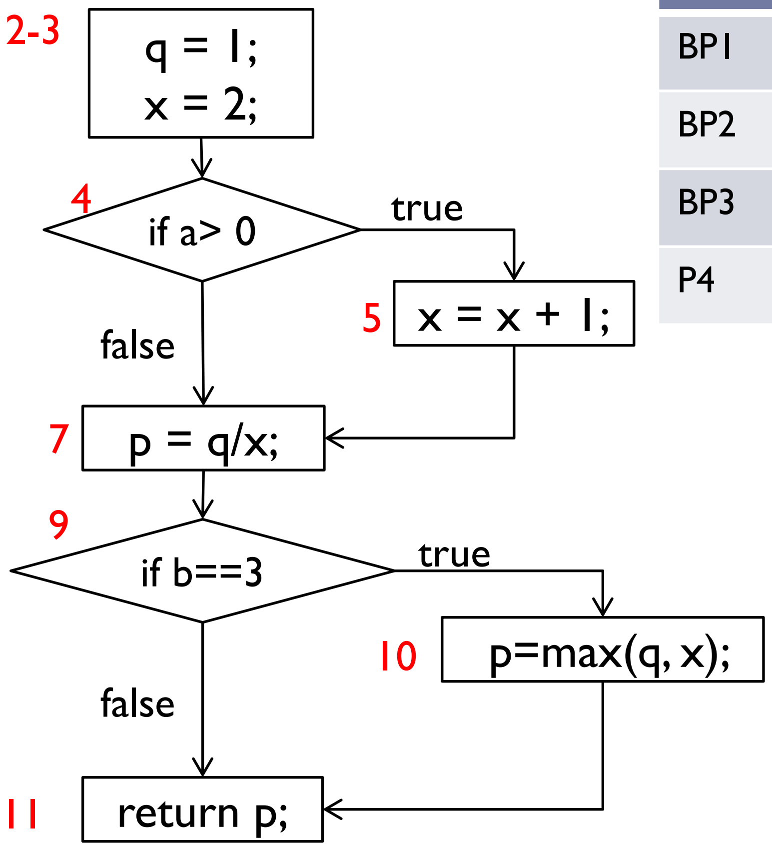


Retrace and Flip Previous Decisions

- Flip the outcome of the second decision point
- Keep the maximum number of other decision points the same



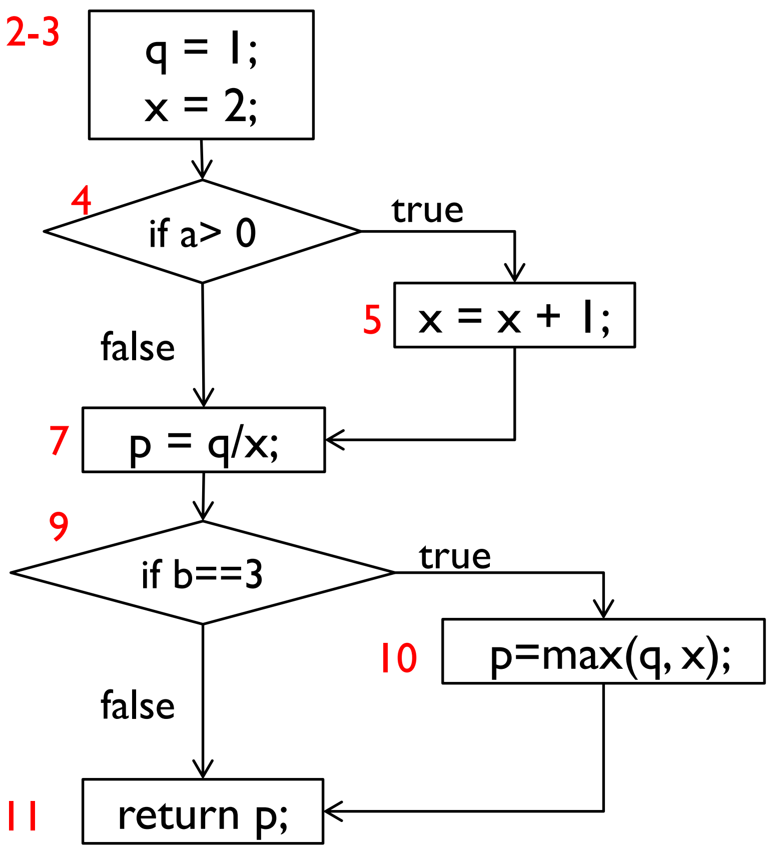
Basis Paths in Vector Space



	2-3-4	4-5-7	4-7	7-9	9-10-11	9-11
BP1	1	1	0	1	0	1
BP2	1	0	**1**	1	0	1
BP3	1	1	0	1	**1**	0
P4	1	0	1	1	1	0

- The red bold entries show edges that appear in exactly one basis path, so paths BP2, BP3 must be independent.
- Path BP1 is independent of all of these, because any attempt to express BP1 in terms of the others introduces unwanted edges
- The fourth path P4 can be generated by linear combination BP2 + BP3 – BP1

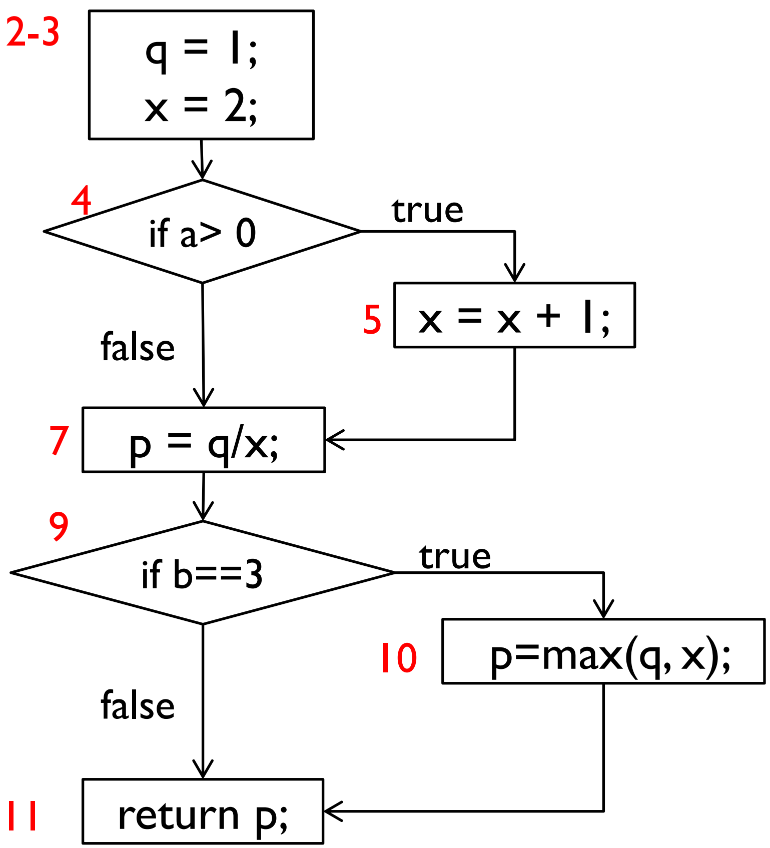
Create a Test Case for Each Basis Path



- One set of basis paths
  I. 2-3, 4, 5, 7, 9, 11
  II. 2-3, 4, 7, 9, 11
  III. 2-3, 4, 5, 7, 9, 10, 11

- Three test cases
  I. a = 4; b = 2
  II. a = 0; b = 5
  III. a = 7; b = 3

Different Sets of Basis Paths for the Same Program



- One set of basis paths
  I. 2-3, 4, 5, 7, 9, 11
  II. 2-3, 4, 7, 9, 11
  III. 2-3, 4, 5, 7, 9, 10, 11

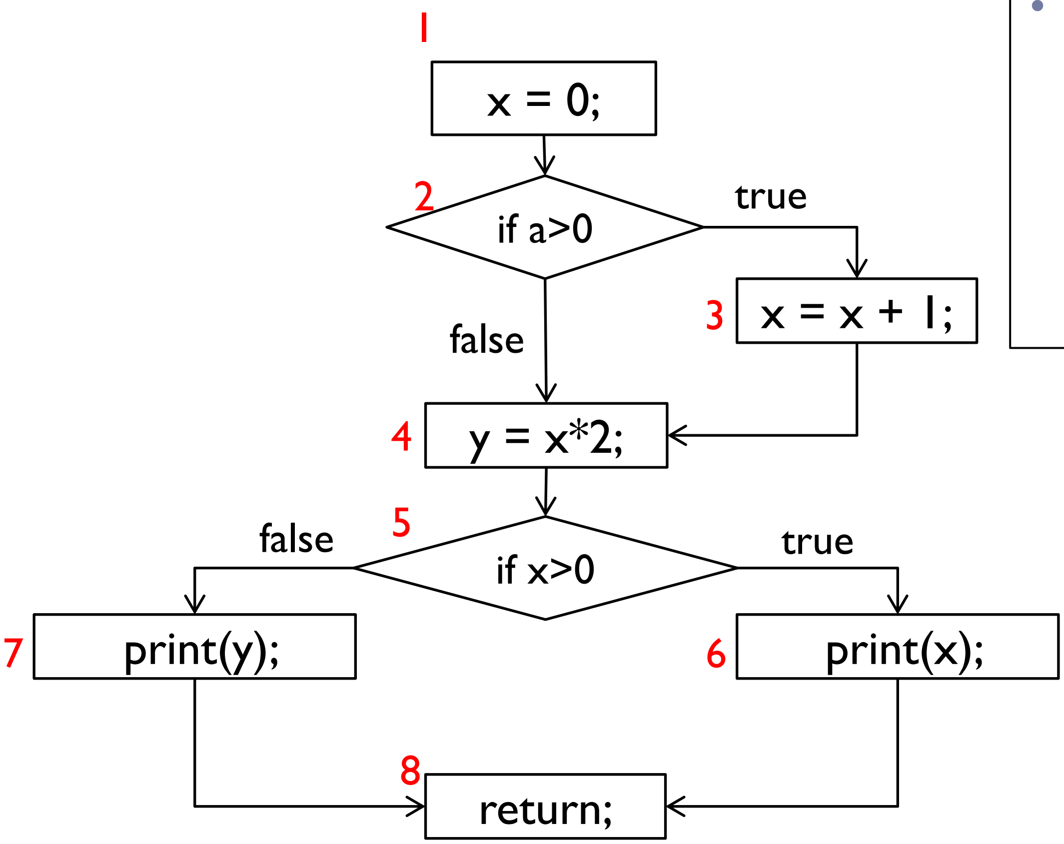
- Another set of basis paths
  I. 2-3, 4, 7, 9, 11
  II. 2-3, 4, 5, 7, 9, 11
  III. 2-3, 4, 7, 9, 10, 11

The third paths in the two sets are different

Some Criticisms on Basis Path Analysis

- Vector operations may not always make practical sense
  - This P4 = BP2 + BP3 – BP1 is fine
- Do not take data dependency into consideration
  - May result in topologically possible but logical infeasible paths

Infeasible Basis Path – Example

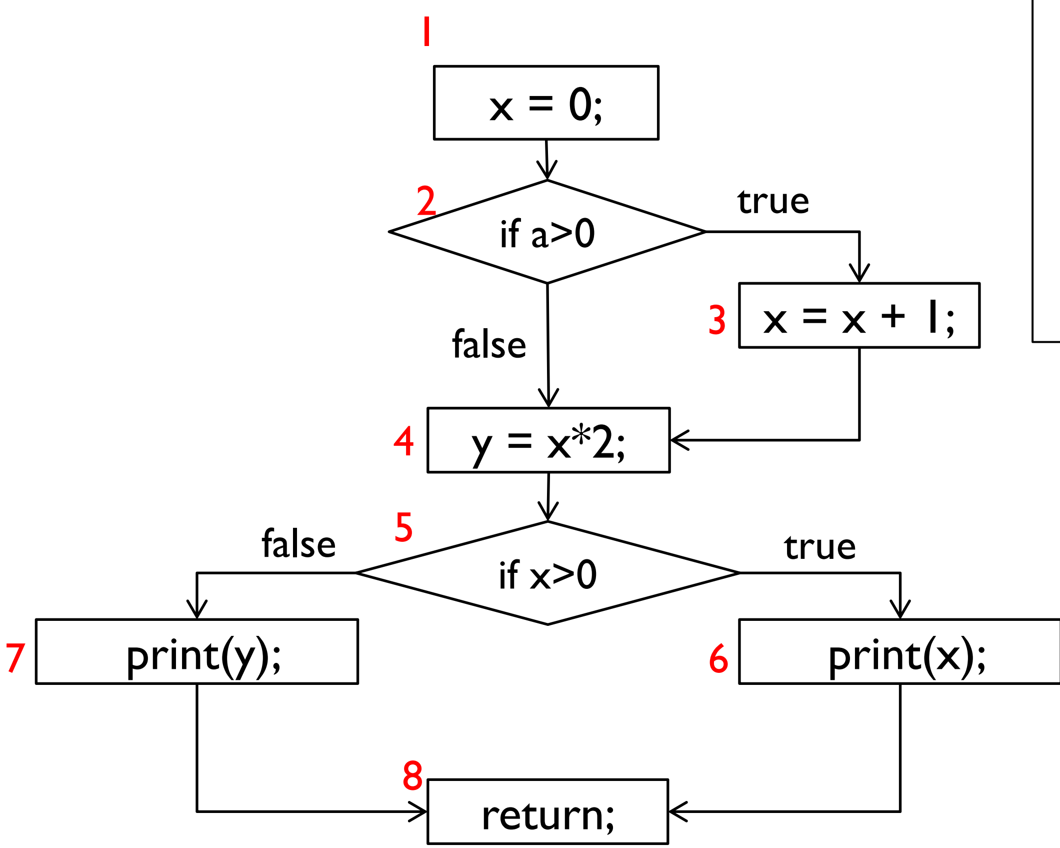


- One set of basis paths
  1. 1, 2, 3, 4, 5, 6, 8
  2. ~~1, 2, 4, 5, 6, 8~~ (infeasible)
  3. ~~1, 2, 3, 4, 5, 7, 8 (infeasible)~~

- One test case
  - a = 4
  - Infeasible basis path
  - Infeasible basis path

Fail to test … 2, 4, … and … 5, 7, 8, … branches

Minimize Infeasible Basis Paths



- Another set of basis paths
    I. 1, 2, 3, 4, 5, 6, 8
    II. ~~1, 2, 4, 5, 6, 8~~ (infeasible)
    III. 1, 2, 4, 5, 7, 8 (change both decision points at the same time)

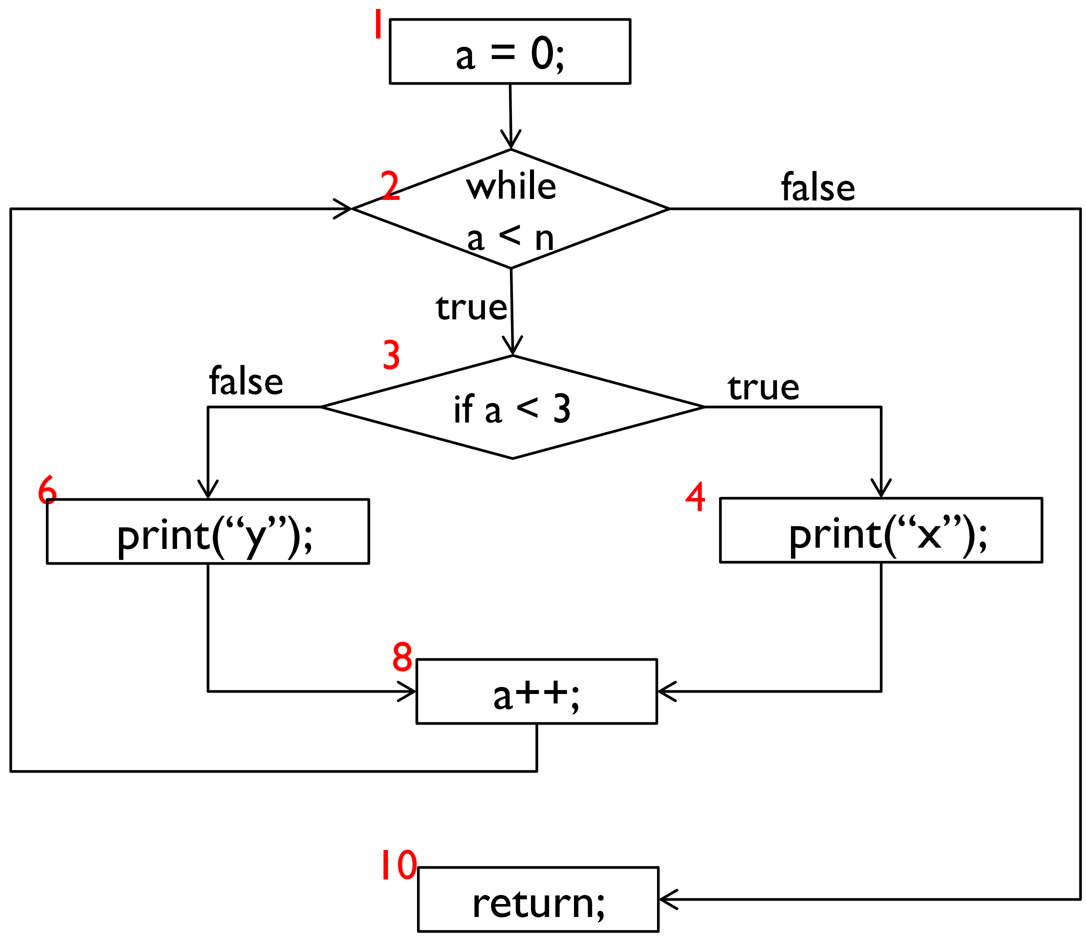
- Two test cases
    - a = 4
    - Infeasible basis path
    - a = 0

Deal with Loop in Basis Path Selection

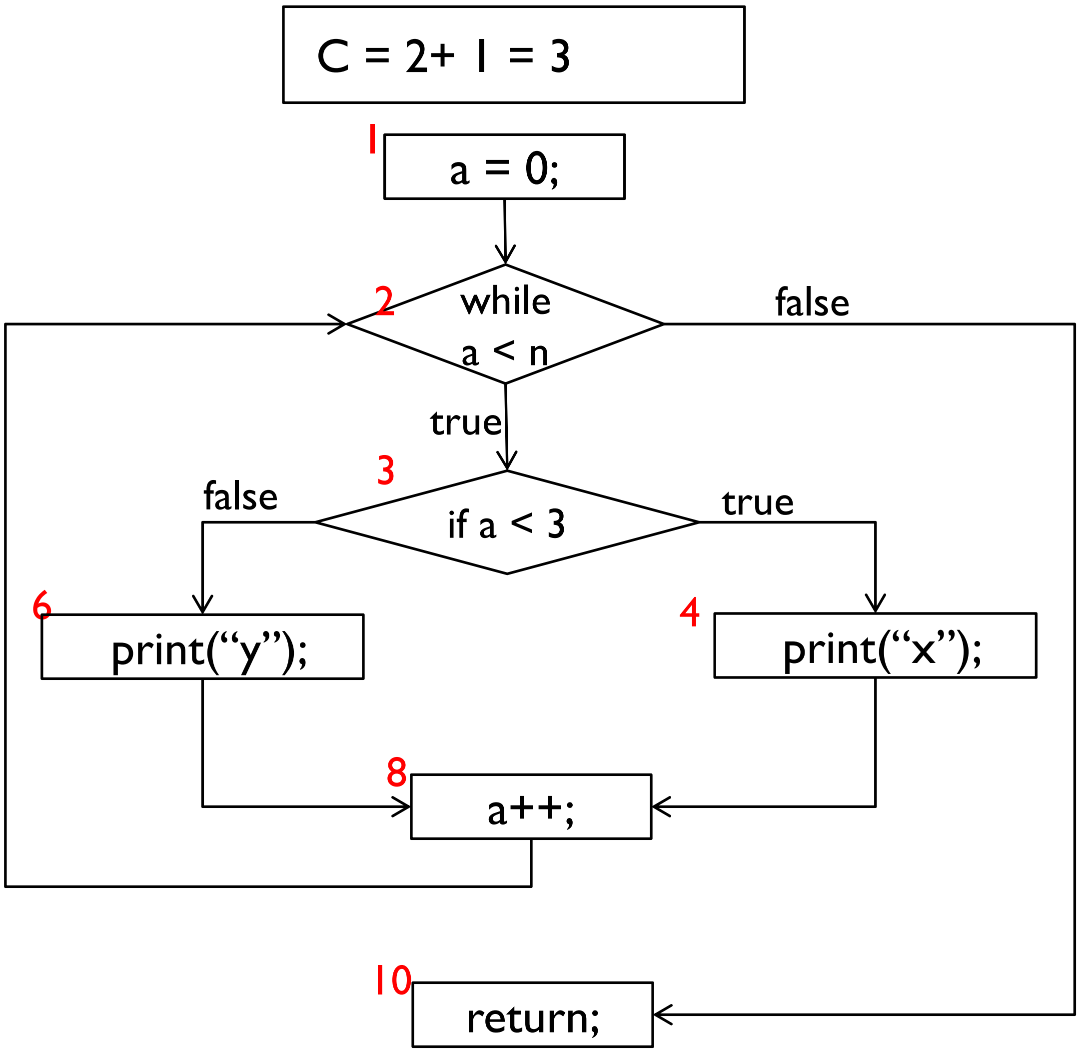
- When selecting basis paths, test the loop only zero and once (no need to consider iteration)
- When selecting baseline path, select false branch first at loop decision point if possible (i.e., do not enter the loop)
- When choose input values to execute the path, the real execution path may execute loop several times

Exercise – Basis Path Testing

```
doSomething(int n) {
1.  int a = 0;
2.  while(a < n) {
3.    if(a < 3) {
4.      print("X");
5.    } else {
6.      print("Y");
7.    }
8.    a++;
9.  }
10. return;
}
```



Exercise – Basis Path Testing



- Three basis paths
  1. 1, 2, 10
  2. 1, 2, 3, 4, 8, 2, 10
  3. 1, 2, 3, 6, 8, 2, 10

- Three test cases
  1. n = 0
  2. n = 1
  3. n = 4

- Real execution paths
  1. 1, 2, 10
  2. 1, 2, 3, 4, 8, 2, 10
  3. 1, 2, 3, 4, 8, 2, 3, 4, 8, 2, , 3, 4, 8, 2, 3, 6, 8, 2, 10

Guidelines and Observations

Testing Efficiency

- Pros
  - Be sure that every path through the SUT has been identified and tested according to certain test coverage criteria
- Cons
  - Testing all execution paths is generally infeasible
    - e.g., loop, #decision points
  - May not detect data sensitivity bugs
    - e.g., a = a – 1 // should be a = a + 1; or a/b // b cannot be zero
  - Never find paths that are not implemented
    - e.g., paths in specification may simply be missing in implementation

Applicability

- A narrow view of white box testing
  - Code testing performed by developers
- But white box testing is more than code testing – it is path testing
- We can apply the same techniques to test paths between modules within subsystems and between subsystems within systems