Structural Testing

CSCE 740 - Lecture 21 - 11/11/2015

We Will Cover

- A second approach to testing, which is geared to expose program faults
- The use of program structure analysis in testing
  - Statement Coverage
  - Branch coverage
  - Path coverage
- The concept of program complexity

Structural Testing

- Sometime called white-box testing
  - Requirements-based = black-box. We don't care what is going on inside the program.
- Derivation of test cases according to program structure
  - Knowledge of the program is used to identify test cases.

Structural Testing

- Uses a family of metrics that measure “how much” code is executed - individually executed statements/branches/paths, combinations of conditional expressions.
- Goal is to exercise a certain percentage of the code.
  - Why??

The basic idea:
You can’t find all of the faults without exercising all of the code.

Structural Testing - Motivation

- Requirements-based tests should execute *most* code, but will rarely execute all of it.
  - Helper functions
  - Error-handling code
  - Missing outcomes
- Structural testing compliments requirements-based testing by requiring that code elements are exercised in different ways.

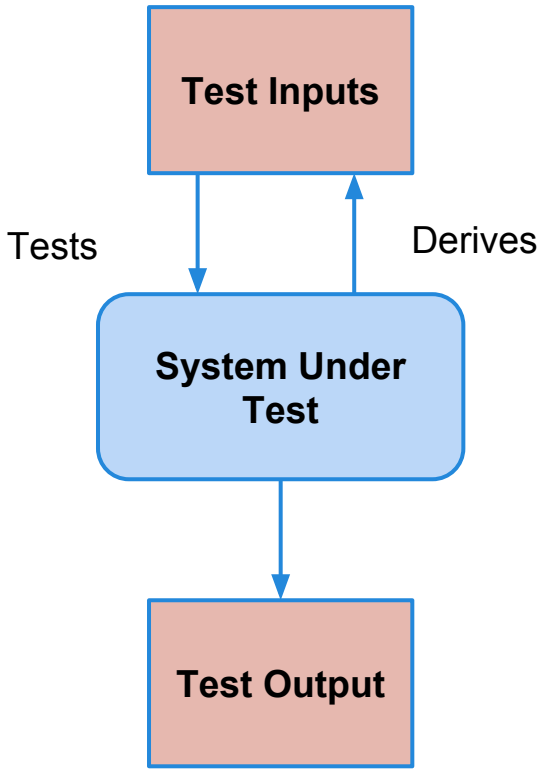
White Box Does Not Replace Black Box

- Structural tests do not replace requirements-based tests.
  - Structure-based tests do not directly make an argument for verification.
  - They cannot expose “missing path” faults - where the implementation does not include items in the specification.
  - Black box tests are good at exposing conceptual faults. White box tests are good at exposing coding mistakes.

Structural Testing Usage

Take code, derive information about structure, use test obligation information to:

- Create Tests
  - Design tests that satisfy obligations.
- Measure Adequacy of Existing Tests
  - Measure coverage of existing tests, fill in gaps.



Control and Data Flow

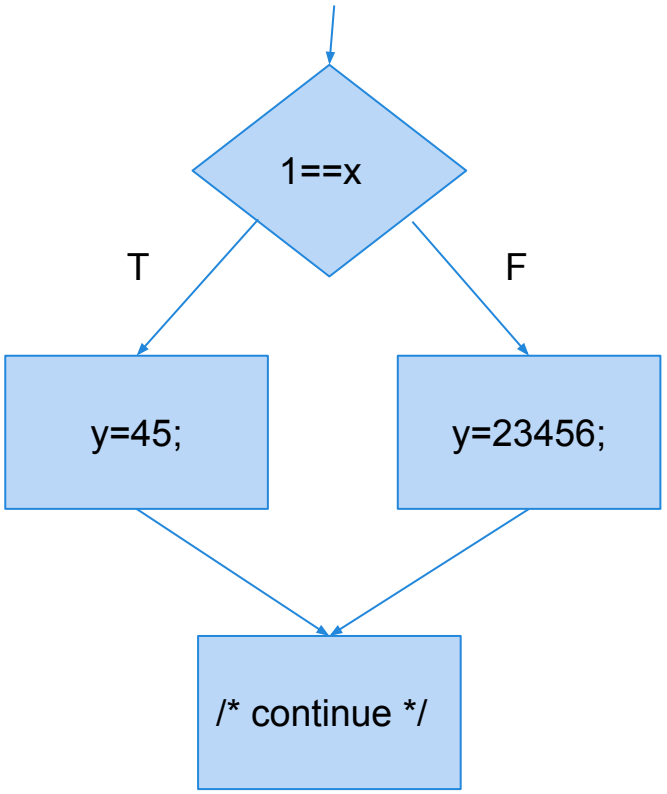
- We need context on how the system executes.
- Code is rarely sequential - conditional statements result in branches in execution, jumping between blocks of code.
  - Control flow is information on how control passes between blocks of code.
- Data flow is information on how variables are used in other expressions.

Control-Flow Graphs

- A directed graph representing the flow of control through the program.
- Nodes represent sequential blocks of program commands.
- Edges connect nodes in the sequence they are executed. Multiple edges indicate conditional statements (loops, if statements, switches).

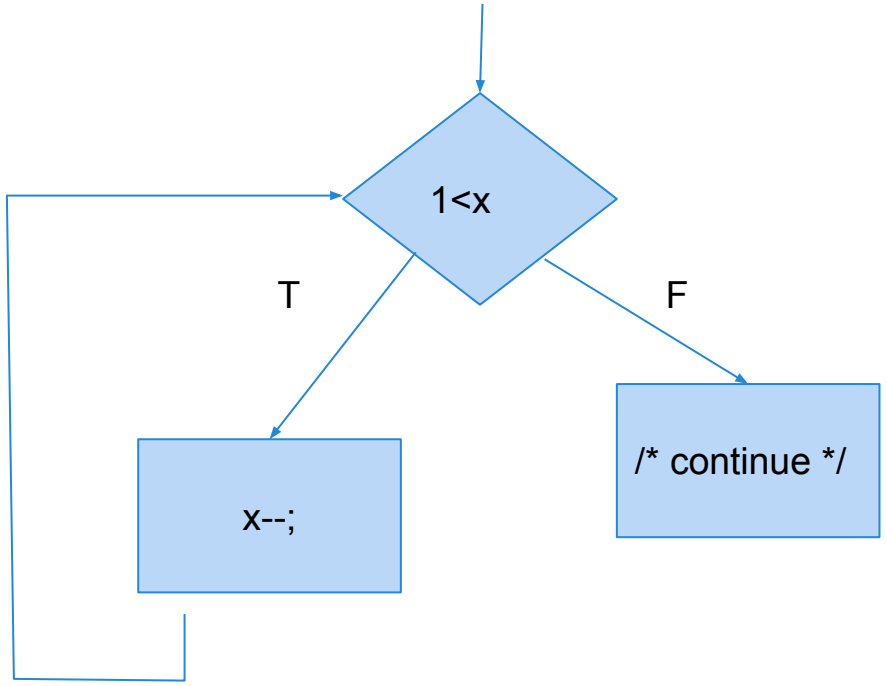
If-then-else

```
1 if (1==x) {
2     y=45;
3 }
4 else {
5     y=23456;
6 }
7 /* continue */
```



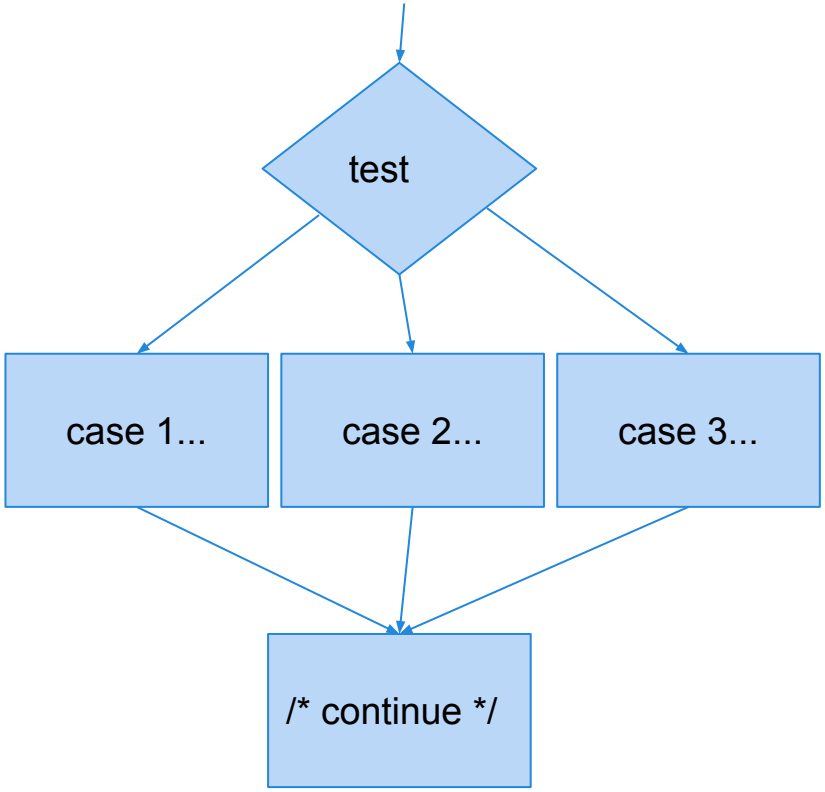
Loop

```
1 while (1<x) {
2       x--;
3 }
4 /* continue */
```



Case

```
1 switch (test) {
2       case 1 : ...
3       case 2 : ...
4       case 3 : ...
5 }
6 /* continue */
```



Activity 1 - Control-Flow Graph

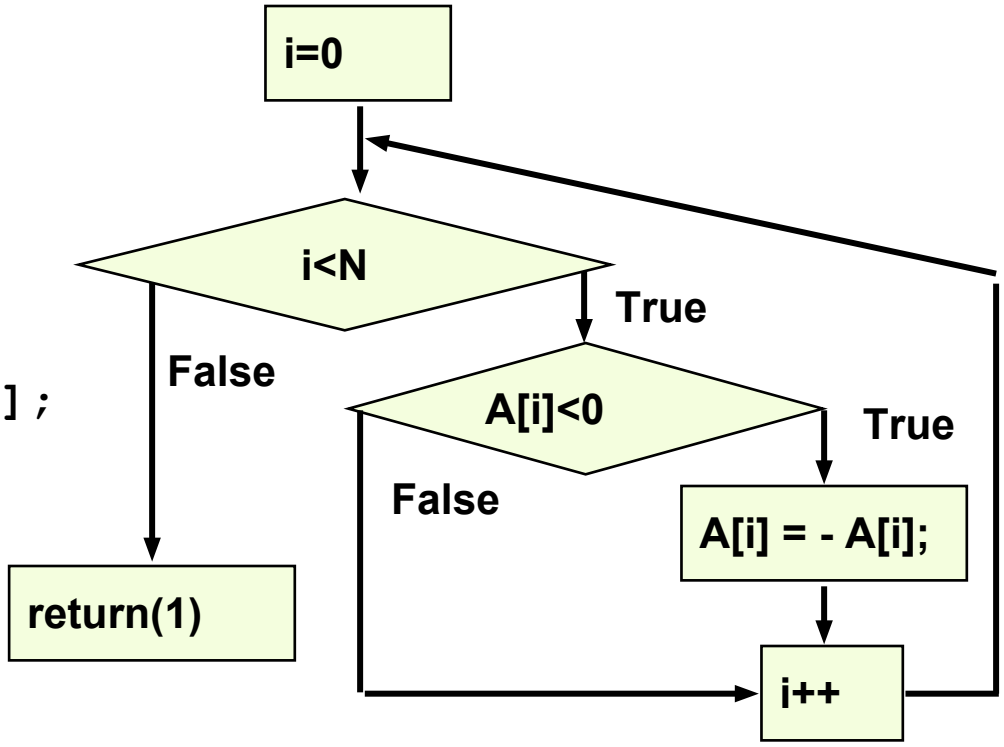
Draw a control-flow graph for the following code:

```
1. int abs(int A[], int N)
2. {
3.      int i=0;
4.     while (i< N)
5.     {
6.         if (A[i]<0)
7.            A[i] = - A[i];
8.         i++;
9.     }
10.    return(1);
11.}
```

Activity 1 - Solution

Draw a control-flow graph for the following code:

```
1. int abs(int A[], int N)
2. {
3.      int i=0;
4.     while (i< N)
5.     {
6.          if (A[i]<0)
7.              A[i] = - A[i];
8.          i++;
9.     }
10.    return(1);
11.}
```



Structural Coverage Criteria

- Criteria based on exercising of:
  - Statements (nodes of CFG)
  - Branches (edges of CFG)
  - Conditions
  - Paths
  - … and many more
- Measurements used as (in)adequacy criteria
  - If significant parts of the program are not tested, testing is surely inadequate.

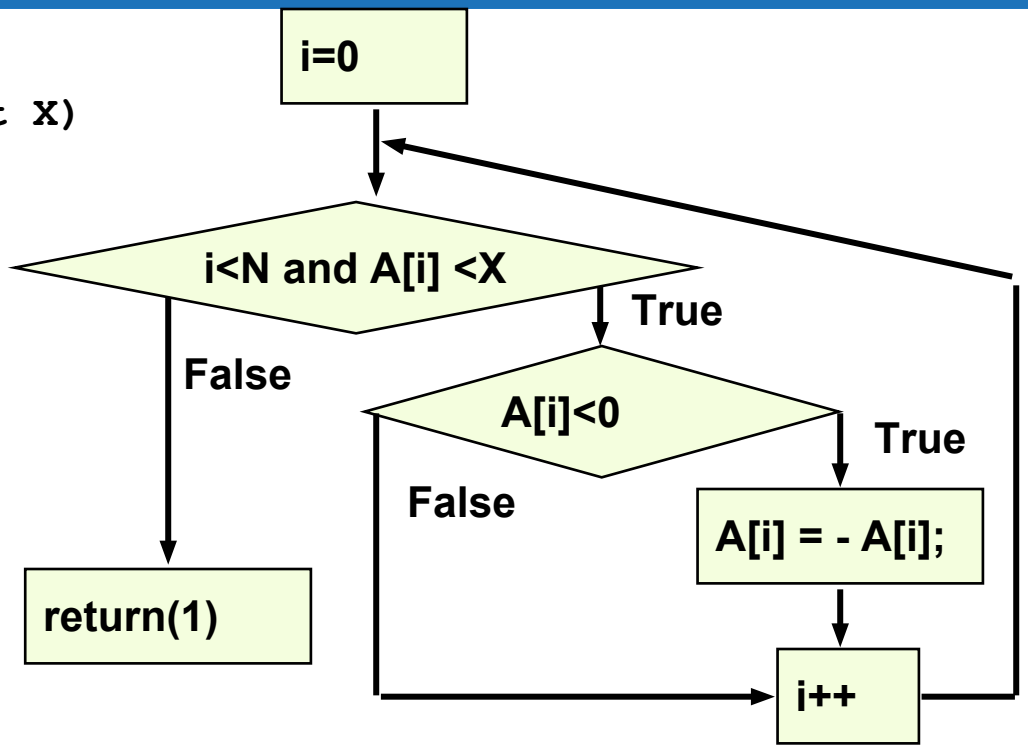
Structural coverage is an attempt to compromise between the **impossible** and the **inadequate**.

Statement Coverage

- The most intuitive criteria. Did we execute every statement at least once?
  - Cover each node of the CFG.
- The idea: faults cannot be revealed unless we execute the statement.
- Coverage = $\frac{\text{Number of Statements Covered}}{\text{Number of Total Statements}}$

Statement Coverage

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```



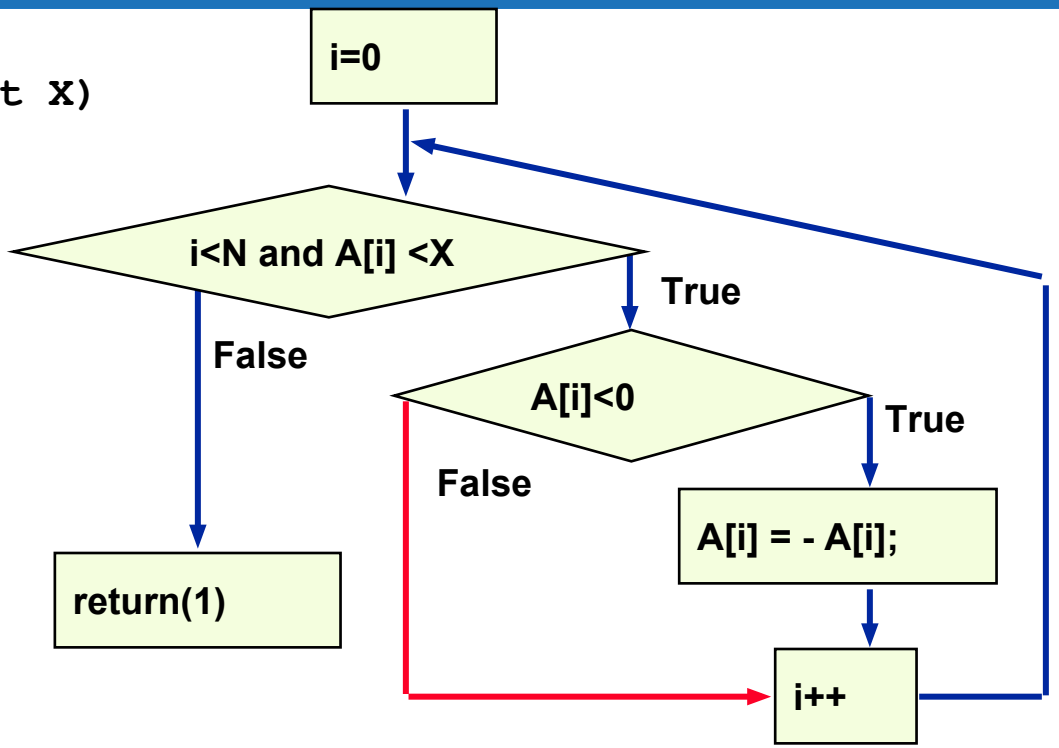
How many tests do we need to provide coverage?
What kind of faults could we miss?
Where would we want to use statement coverage?

Branch Coverage

- Do we have tests that take all of the control branches at some point?
  - Cover each edge of the CFG.
- Helps identify faults in decision statements.
- If branch coverage is achieved, than statement coverage is also achieved.
  - branch coverage *subsumes* statement coverage.
- Coverage = $\frac{\text{Number of Branches Covered}}{\text{Number of Total Branches}}$

Branch Coverage

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```



How many tests do we need to provide coverage?
How does fault detection potential change?
Where would we want to use branch coverage?

Condition Coverage

- Condition coverage examines the individual conditions that make up a control-flow decision
- Helps identify faults in conditional statements.

  `(a == 1 || b == -1)` instead of `(a == -1 || b == -1)`
- Several different forms.
- Coverage = $\frac{\text{Number of Truth Values for All Conditions}}{\text{2x Number of Conditions}}$

Basic Condition Coverage

- Make each condition both True and False

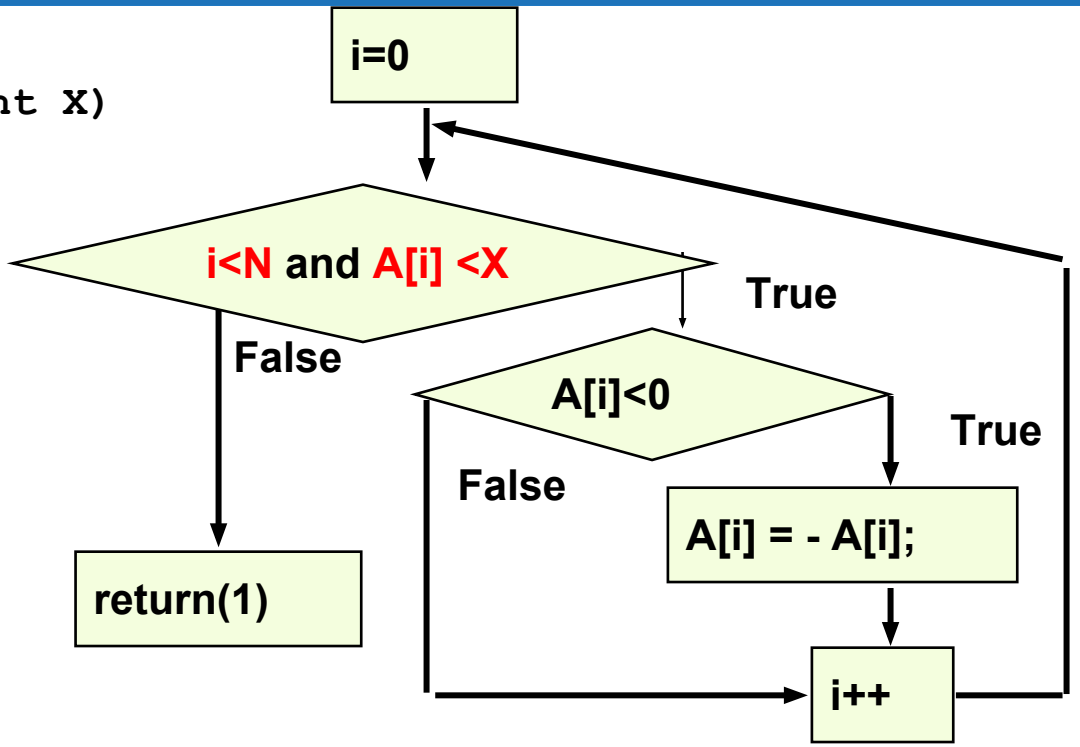
(A and B)

Test Case	A	B
1	True	False
2	False	True

- Can be satisfied without hitting both branches, so does not subsume branch coverage.
  - In this case, false branch is taken for both tests

Basic Condition Coverage

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```



How many tests do we need to provide coverage?
How does fault detection potential change?
Where would we want to use condition coverage?

Compound Condition Coverage

- Evaluate every combination of the conditions

(A and B)

Test Case	A	B
1	True	True
2	True	False
3	False	True
4	False	False

- Subsumes branch coverage, as all outcomes are now tried.
- Can be expensive in practice.

Compound Condition Coverage

- Requires **many** test cases.

(A and
(B and
(C and D))))

Test Case	A	B	C	D
1	True	True	True	True
2	True	True	True	False
3	True	True	False	True
4	True	True	False	False
5	True	False	True	True
6	True	False	True	False
7	True	False	False	True
8	True	False	False	False
9	False	True	True	True
10	False	True	True	False
11	False	True	False	True
12	False	True	False	False
13	False	False	True	True
14	False	False	True	False
15	False	False	False	True
16	False	False	False	False

Modified Condition/Decision Coverage (MC/DC)

- Requires:
  - Each **condition** evaluates to true/false
  - Each **decision** evaluates to true/false
  - Each condition shown to **independently affect outcome** of each decision it appears in.

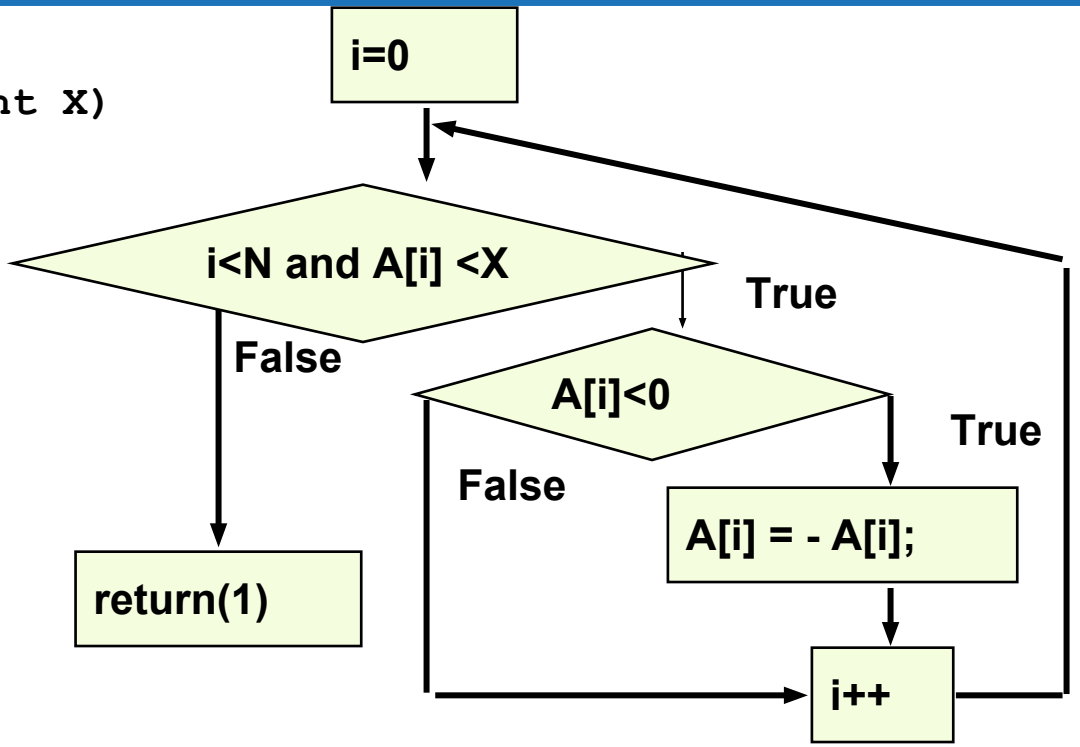
Test Case	A	B	(A and B)
1	True	True	True
2	True	False	False
3	False	True	False
4	False	False	False

Path Coverage

- Other criteria focus on one element at a time. Combination of elements matters - interaction sequences are the root of many faults.
- Path coverage requires that all paths through the CFG are covered.
- $$\text{Coverage} = \frac{\text{Number of Paths Covered}}{\text{Number of Total Paths}}$$
- Impractical in practice.

Path Coverage

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```

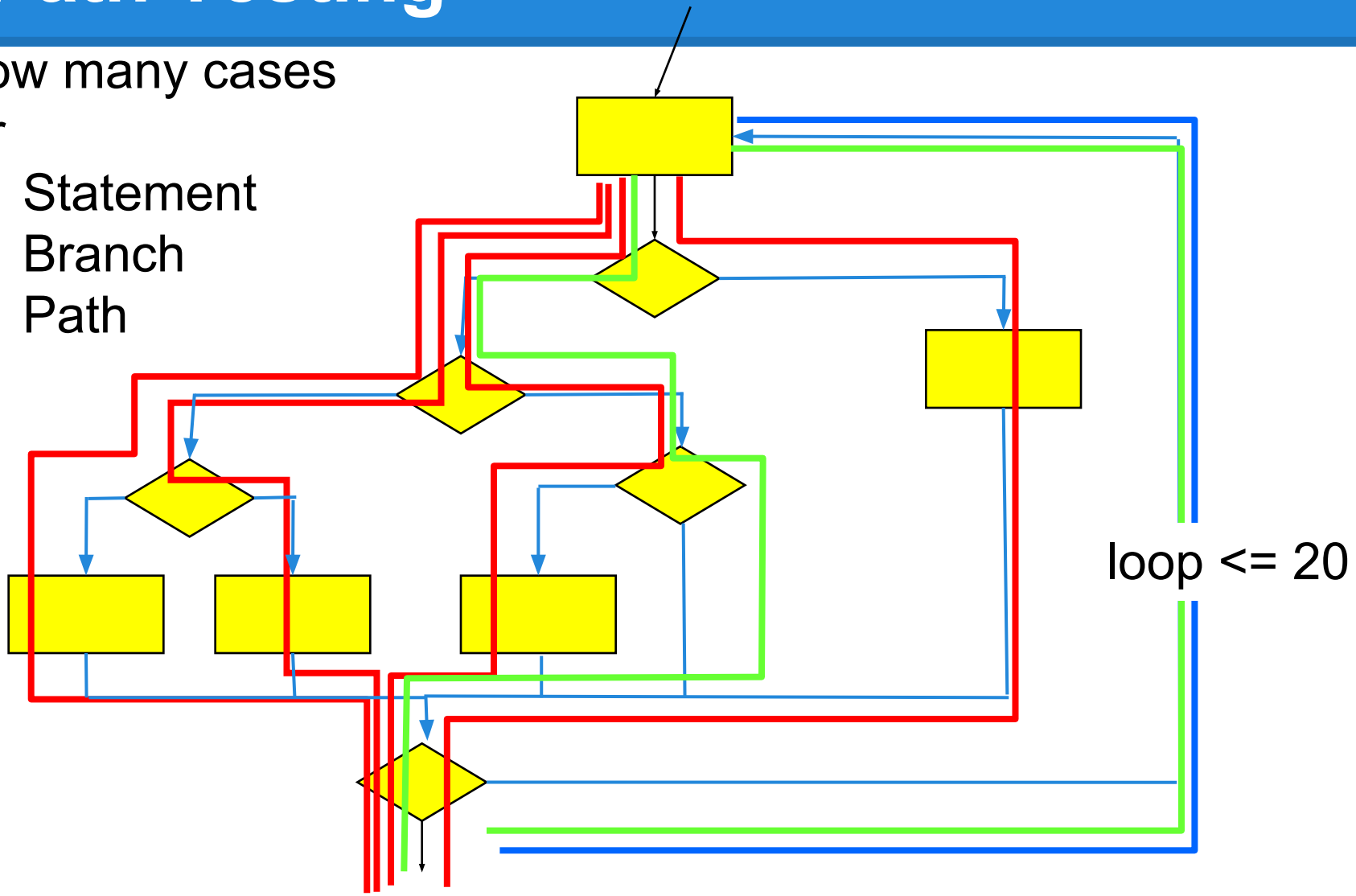


How many tests do we need to provide coverage?
How do we deal with loops?

Path Testing

How many cases for

- Statement
- Branch
- Path



Number of Tests

Path coverage for that loop bound requires:
3,656,158,440,062,976 test cases

If you run 1000 tests per second, this will take **116,000 years**.

Making Path Coverage Feasible

- Impose limitations on loops.
  - Try 0,1,...,n times.
- Cover all independent subpaths.
  - Which can be combined to form all paths.
- Data-Flow Coverage Criteria
  - Cover all pairings of definition and use.

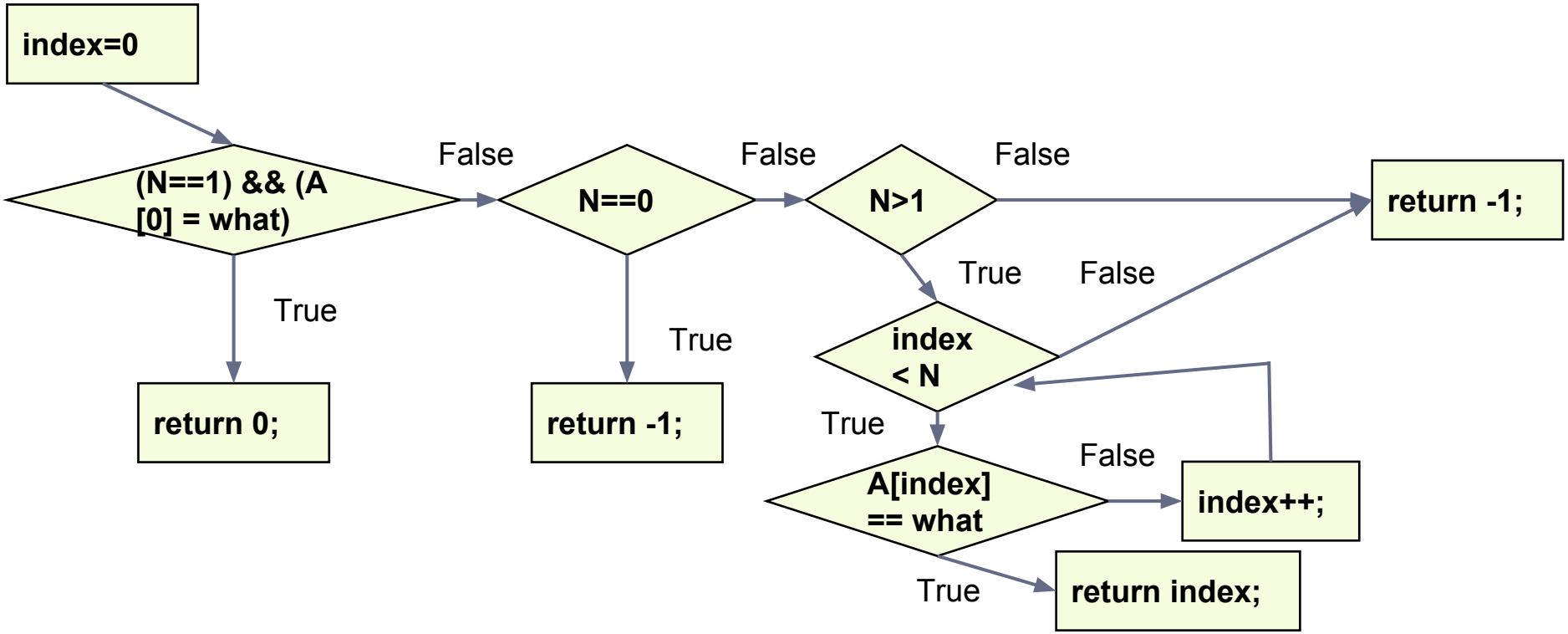
Activity 2

Write tests that provide statement, branch, and basic condition coverage over the following code:

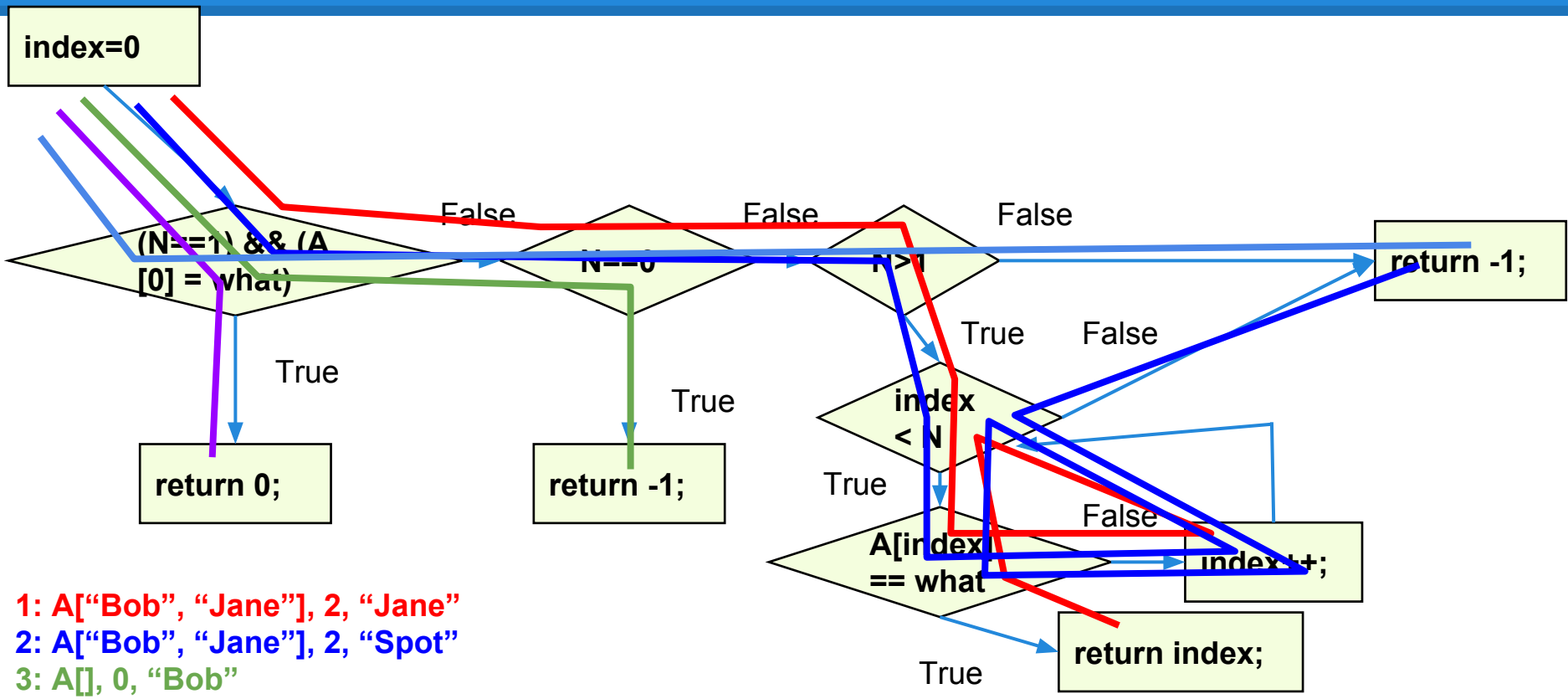
```
int search(string A[], int N, string what){
    int index = 0;
    if ((N == 1) && (A[0] == what)){
         return 0;
    } else if (N == 0){
        return -1;
    } else if (N > 1){
        while(index < N){
            if (A[index] == what)
               return index;
            else
                index++;
        }
    }
    return -1;
}
```

(Hint - draw the CFG first to help check coverage)

Activity 2


index=0
(N==1) && (A [0] = what)
False
N==0
False
N>1
False
return -1;
True
return 0;
True
return -1;
True
False
index < N
True
False
A[index] == what
index++;
True
return index;

Activity 2 - Possible Solution



1: A[“Bob”, “Jane”], 2, “Jane”
2: A[“Bob”, “Jane”], 2, “Spot”
3: A[], 0, “Bob”
4. A[“Bob”], 1, “Bob”
5. A[“Bob”], 1, “Spot”

Where Coverage Goes Wrong...

- Testing can only reveal a fault when execution of the faulty element causes a failure, but…
- Execution of a line containing a fault does not guarantee a failure.
  - If (a <= b) was written as (a >= b) accidentally, the fault will not manifest as a failure if a==b in the test case.
- Merely executing code does not guarantee that we will find all faults.

The Infeasibility Problem

Sometimes, **no** set of test cases can satisfy a coverage criterion because there are elements that no test can execute in the required manner.

- Impossible combinations of conditions.
- Unreachable statements as part of defensive programming.
  - Error-handling code for conditions that can't actually occur in practice.
- Dead code in legacy applications.
- Inaccessible portions of off-the-shelf systems.

The Infeasibility Problem

Stronger criteria call for potentially infeasible combinations of elements.

```
(a > 0 && a < 10)
```

It is not possible for both conditions to be false.

Problem compounded for path-based coverage criteria.

Not possible to traverse the path where both if-statements evaluate to true.

```
if (a < 0)  a = 0;
if (a > 10) a = 10;
```

The Infeasibility Problem

How this is usually addressed:

- Adequacy "scores" based on coverage.
  - 95% statement coverage, 80% MC/DC coverage, etc.
  - Decide to stop once a threshold is reached.
  - Unsatisfactory solution - elements are not equally important for fault-finding.
- Manual justification for omitting each impossible test obligation.
  - Required for safety certification in avionic systems.
  - Helps refine code and testing efforts.
  - … but **very** time-consuming.

In Practice.. The Budget Coverage Criterion

- Industry's answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1:
    - Adequacy criteria answer the wrong question. Selection is more important.
  - Implication 2:
    - Practical comparison of approaches must consider the cost of test case selection

We Have Learned

- Code structure can be used to derive test cases or as a measurement of test adequacy.
- Many different criteria, based on:
  - Statements, branches, conditions, paths, etc.
- Infeasibility problem - not always possible to cover all elements.
- Coverage should not be used for the sake of coverage, but should factor in budget, time, project type, and problem domain.

Next Time

- The Test Oracle - how we judge the behavior of the system under test.
- Homework 4
  - Started coding?
  - Any questions?