Structural Testing

CSCE 247 - Lecture 22 - 04/15/2019

Every developer must answer:

Are our tests are any good?

More importantly… Are they good enough to stop writing new tests?

Have We Done a Good Job?

What we want:

- We've found all the faults.
  - Impossible.

What we (usually) get:

- We compiled and it worked.
- We run out of time or budget.
  - Inadequate.

Test Adequacy Metrics

Instead - can we **compromise between the impossible and the inadequate**?

- Can we measure “good testing”?
  - Test adequacy metrics “score” testing efforts by measuring the completion of a set of **test obligations**.
    - Properties that must be met by our test cases.

(In)Adequacy Metrics

- We do not know what faults exist before testing, so we rely on an approximation of “we found all of the faults”.
- Criteria identify **in**adequacies in the tests.
  - If the test does reach a statement, it is *inadequate* for finding faults in that statement.
  - If the requirements discuss two outcomes of a function, but the tests only cover one, then the tests are *inadequate* for verifying that requirement.

Adequacy Metrics

- Adequacy Metrics based on coverage of factors correlated to finding faults.
  - (hopefully)
  - Widely used in industry - easy to understand, cheap to calculate, offer a checklist.
  - Some metrics based on coverage of requirement statements, used for verification.
  - Majority based on exercising elements of the source code in ways that might trigger faults.
    - This is the basis of *structural testing*.

We Will Cover

- Structural Testing:
  - Derive tests from the program structure, directed by a chosen adequacy metric.
- Common structural coverage metrics:
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage

Structural Testing

- The structure of the software itself is a valuable source of information.
- Structural testing is the practice of using that structure to derive test cases.
- Sometime called white-box testing
  - Functional = black-box.

Structural Testing

- Uses a family of metrics that define how and what code is to be executed.
- Goal is to exercise a certain percentage of the code.
  - Why??

```
while (*eptr){
    char c;
    c = *eptr;
    if(c == '+'){
        *dptr = ' ';
    } else{
        *dptr = *eptr;
    }
}
```

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
  - Requirements missing outcomes
- Structural testing compliments functional testing by requiring that code elements are exercised in prescribed ways.

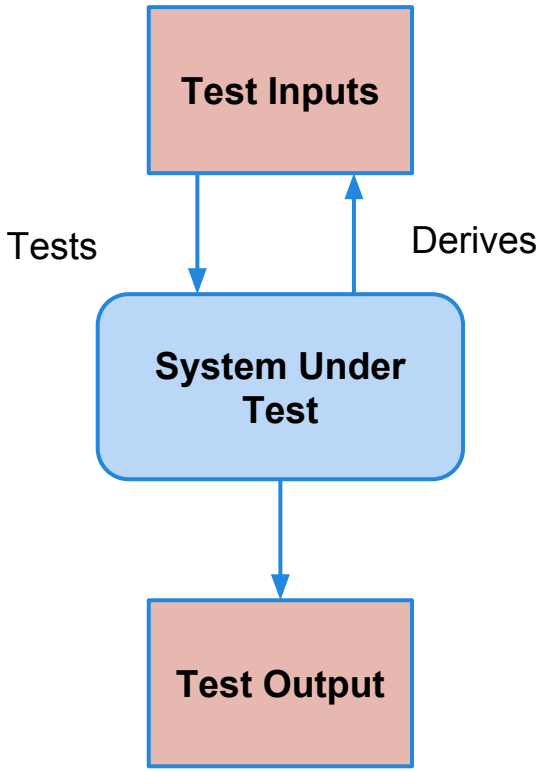
White Box Does Not Replace Black Box

- Structural testing should not be the basis for "How do I choose tests?"
  - Structure-based tests do not directly make an argument for verification and cannot expose "missing path" faults - where the implementation does not include items in the specification.
  - Structural testing is useful for supplementing functional tests to help reveal faults.
    - Functional 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.

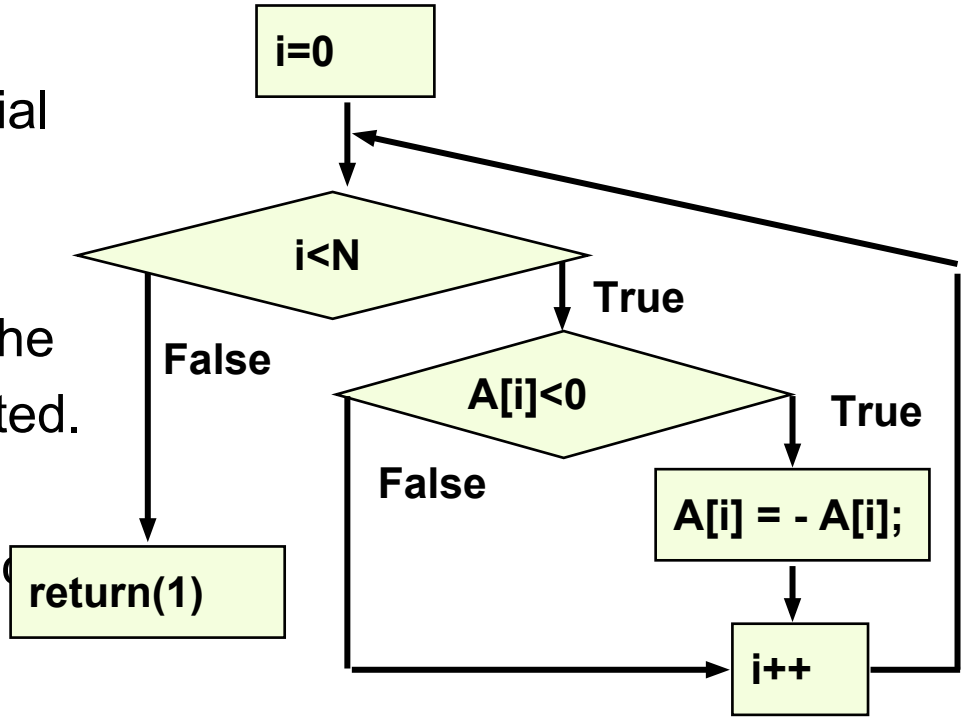


Control and Data Flow

- We need context on how 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.

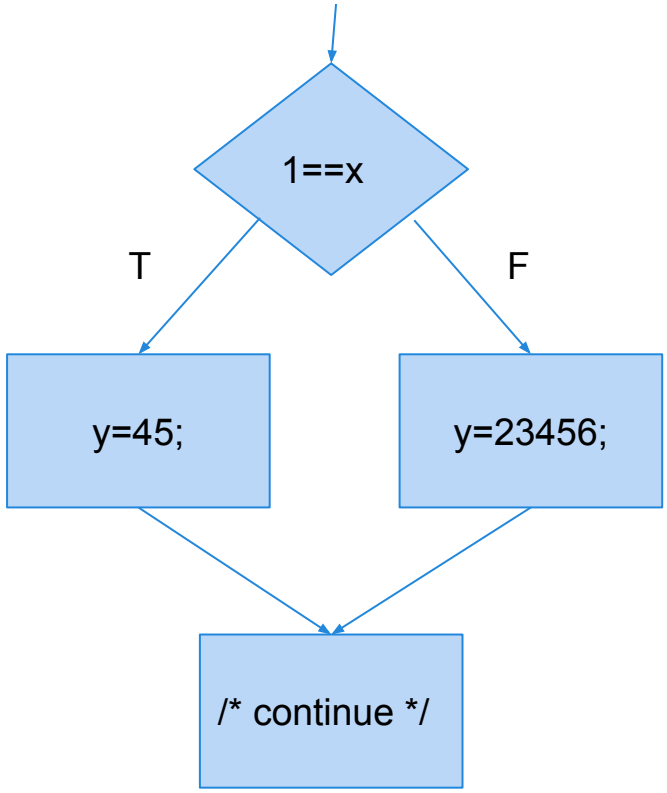
Control-Flow Graphs

- 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 (lo if statements, switches).



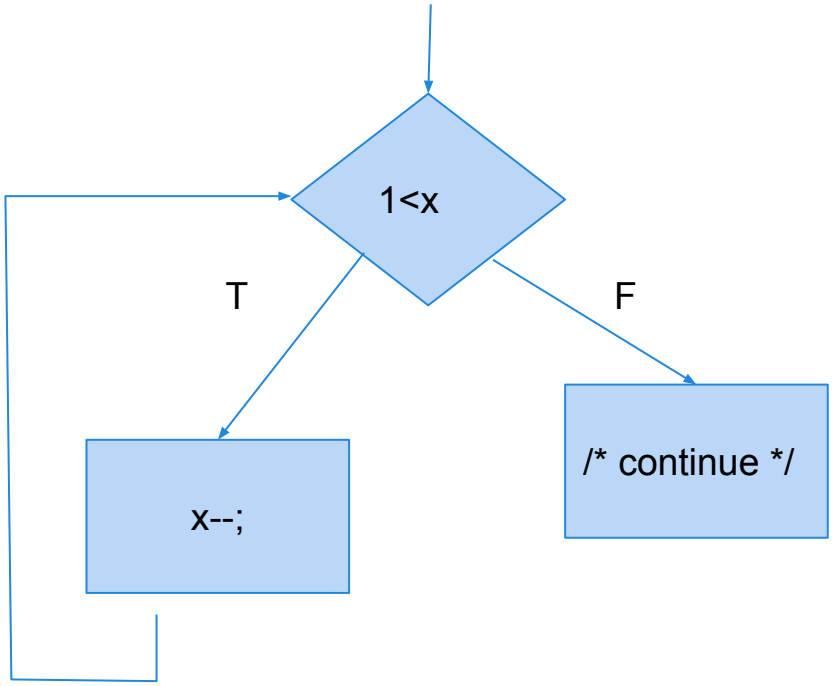
If-then-else

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



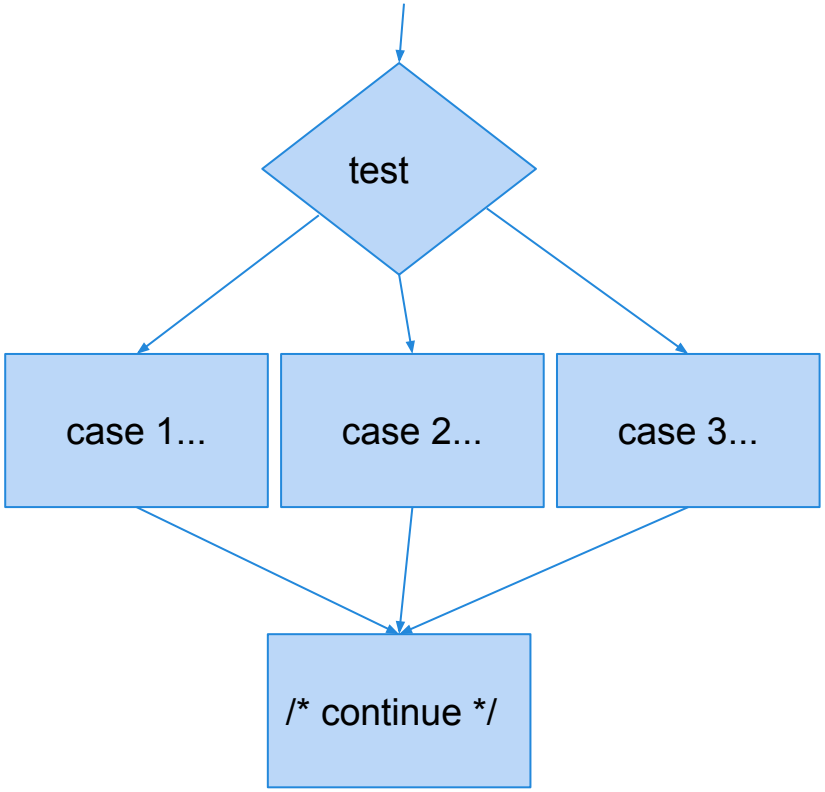
Loop

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



Case

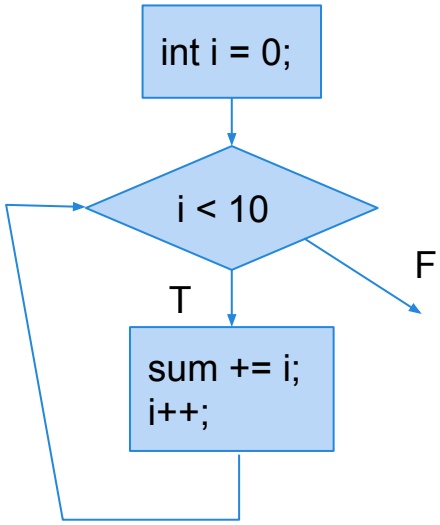
```
1 switch (test) {
2       case 1 : ...
3       case 2 : ...
4     case 3 : ...
5 }
6 /* continue */
```



Basic Blocks

- Nodes represent basic blocks - a set of sequentially executed instructions with a single entry and exit point.
- Typically a set of adjacent statements, but a statement might be broken up into multiple blocks to model control flow in the statement.

```
for(int i=0; i < 10; i++){
    sum += i;
}
```

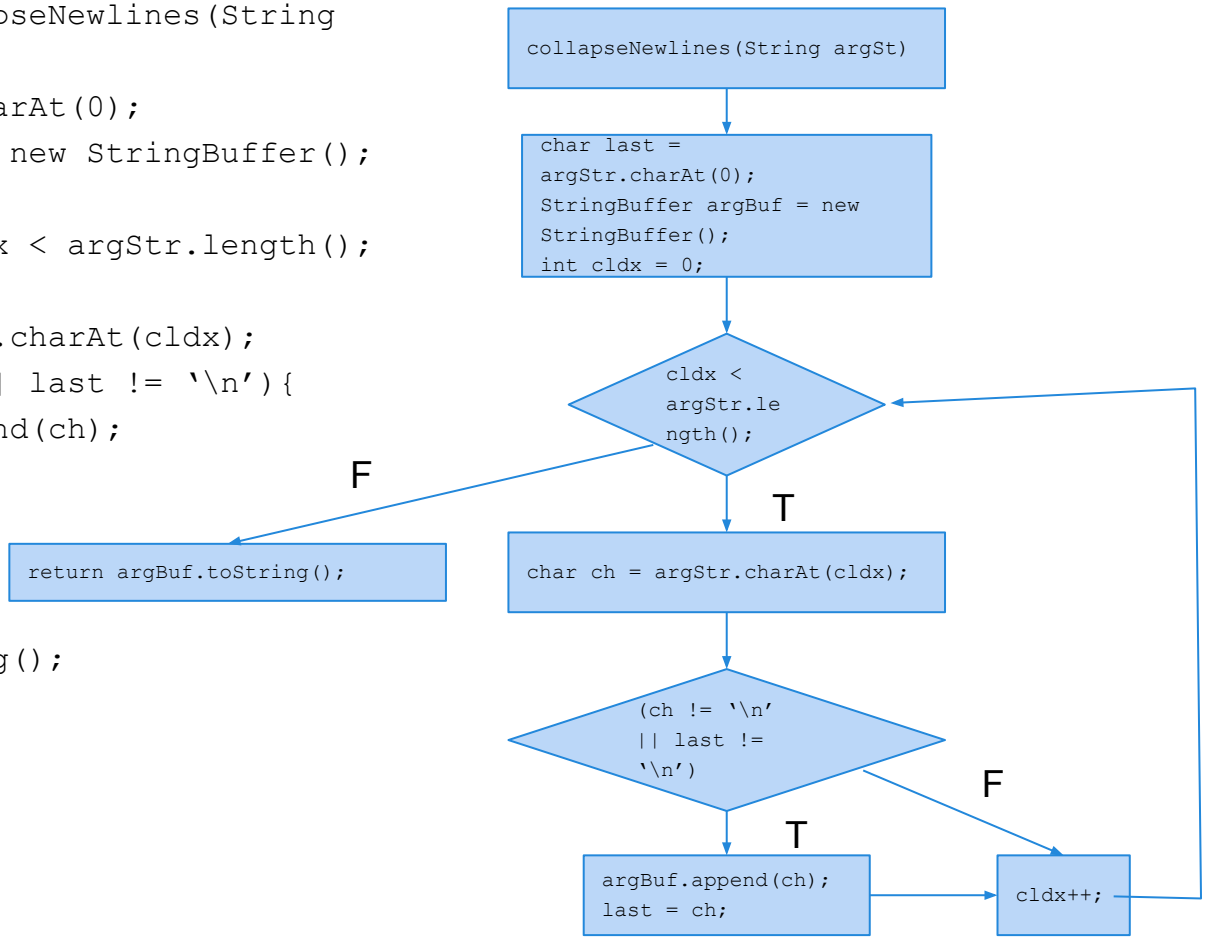


Control Flow Graph Example

```
public static String collapseNewlines(String
argSt){
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for(int cldx = 0; cldx < argStr.length();
      cldx++){
            char ch = argStr.charAt(cldx);
            if (ch != '\n' || last != '\n'){
                  argBuf.append(ch);
                  last = ch;
            {
      }

      return argBuf.toString();
}
```



Structural Coverage Criteria

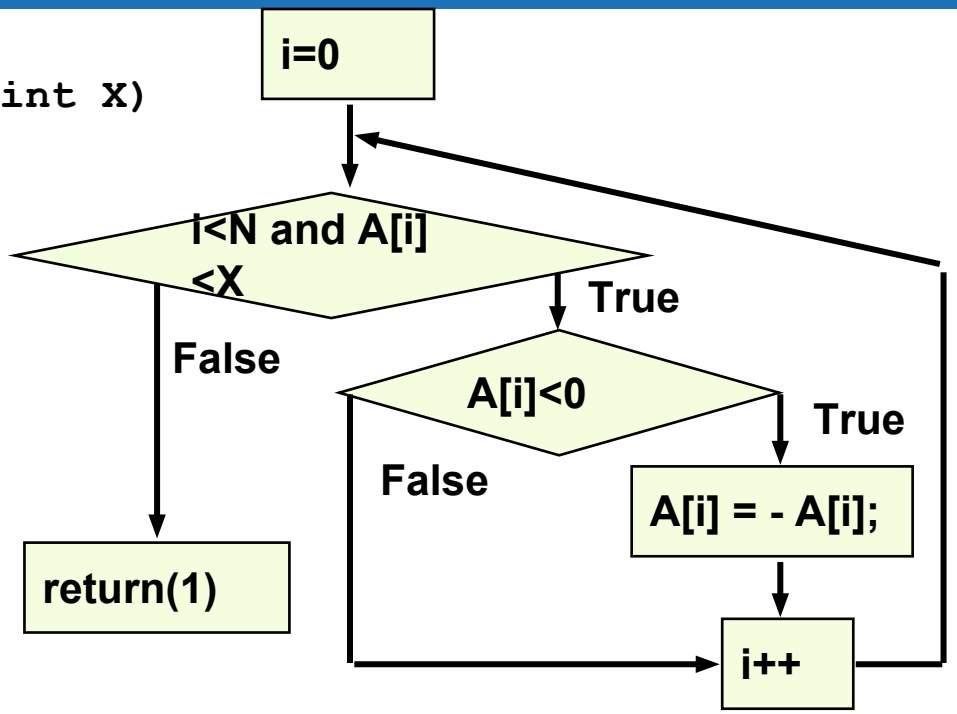
- Criteria based on exercising of:
  - Statements (nodes of CFG)
  - Branches (edges of CFG)
  - Conditions
  - Paths
  - … and many more
- Measurements used as (in)adequacy criteria
  - If significant parts of the program are not tested, testing is surely inadequate.

Statement Coverage

- The most intuitive criteria. Did we execute every statement at least once?
  - Cover each node of the CFG.
- The idea: a fault in a statement 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?

A Note on Test Suite Size

- Level of coverage is not strictly correlated to test suite size.
  - Coverage depends on whether obligations are met. Some tests might not cover new code.
- However, larger suites often find more faults.
  - They exercise the code more thoroughly.
  - *How* code is executed is often more important than *whether* it was executed.

Test Suite Size

- Generally, we favor a large number of *targeted* tests over a smaller number of tests that exercise a lot of statements.
  - If a test targets a smaller number of obligations, it is easier to tell where a fault is.
  - If a test executes everything and covers a large number of obligations, we get higher coverage, but at the cost of being able to identify and fix faults.
  - The exception - if the cost to execute each test is high.

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.
- Coverage = $\frac{\text{Number of Branches Covered}}{\text{Number of Total Branches}}$

Subsumption

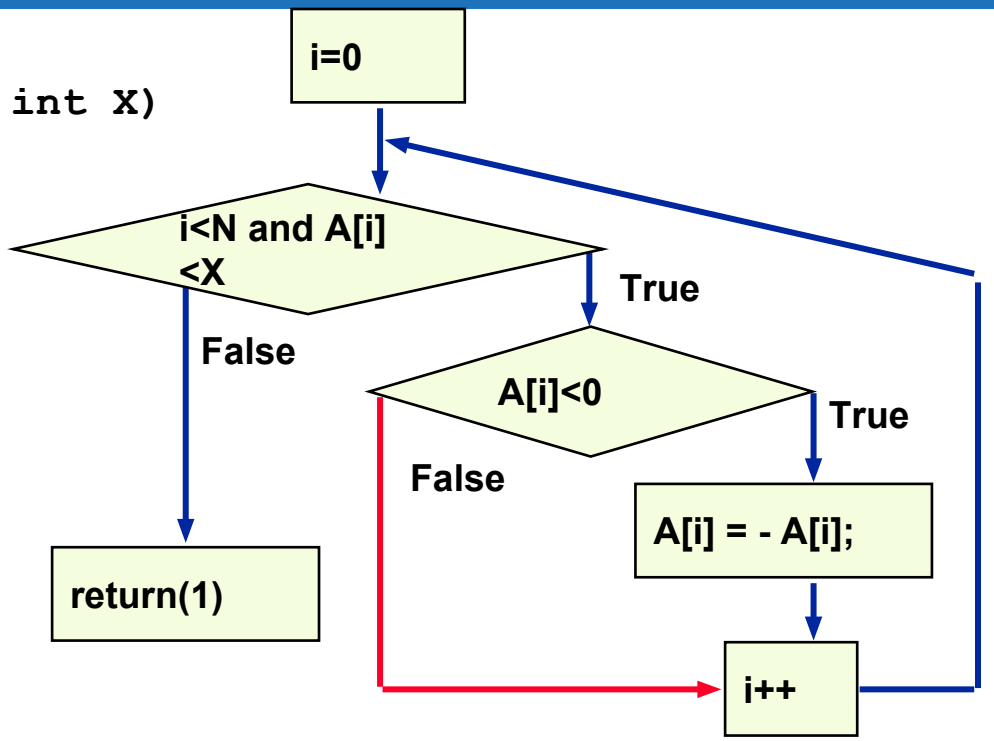
- Coverage metric (*A*) *subsumes* another metric (*B*) if, for every program *P*, every test suite satisfying *A* also satisfies *B* with respect to *P*.
  - If we satisfy A, there is no point in measuring B.
  - Branch coverage subsumes statement coverage.
    - Covering all edges requires covering all nodes in a control-flow graph.
  - Covering all 2-way parameter interactions (combinatorial-interaction testing) subsumes covering all parameter partitions individually.

Subsumption

- Shouldn’t we always choose the stronger metric?
  - Not always…
    - Typically require more obligations (so, you have to come up with more tests)
      - Or, at least, tougher obligations - making it harder to come up with the test cases.
    - May end up with a large number of *unsatisfiable* obligations

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);
}
```



What test obligations must be covered?
How does fault detection potential change?
Where would we want to use branch coverage?

Decisions and Conditions

- A *decision* is a complex Boolean expression.
  - Often cause control-flow branching:
    - `if ((a && b) || !c) { ...`
  - But not always:
    - `Boolean x = ((a && b) || !c);`
  - Made up of *conditions* connected with Boolean operators (and, or, xor, not):
    - Simple Boolean connectives.
      - Boolean variables: `Boolean b = false;`
      - Subexpressions that evaluate to true/false involving (<, >, <=, >=, ==, and !=): `Boolean x = (y < 12);`

Basic Condition Coverage

- Several coverage metrics that examine the individual *conditions* that make up a control-flow *decision*.
- Identify faults in decision statements.

`(a == 1 || b == -1)` instead of `(a == -1 || b == -1)`

- Most basic form: make each condition T/F.
- 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

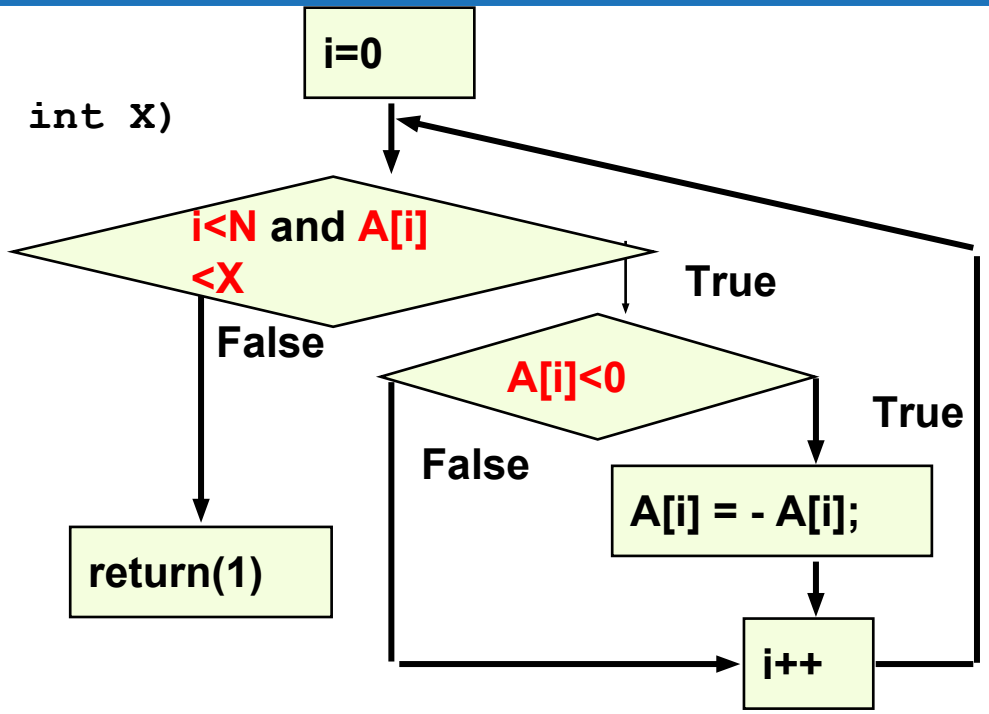
(A and B)

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);
}
```



What test obligations must be covered?
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

Short-Circuit Evaluation

- In many languages, if the first condition determines the result of the entire decision, then fewer tests are required.
  - If A is false, B is never evaluated.

(A and B)

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

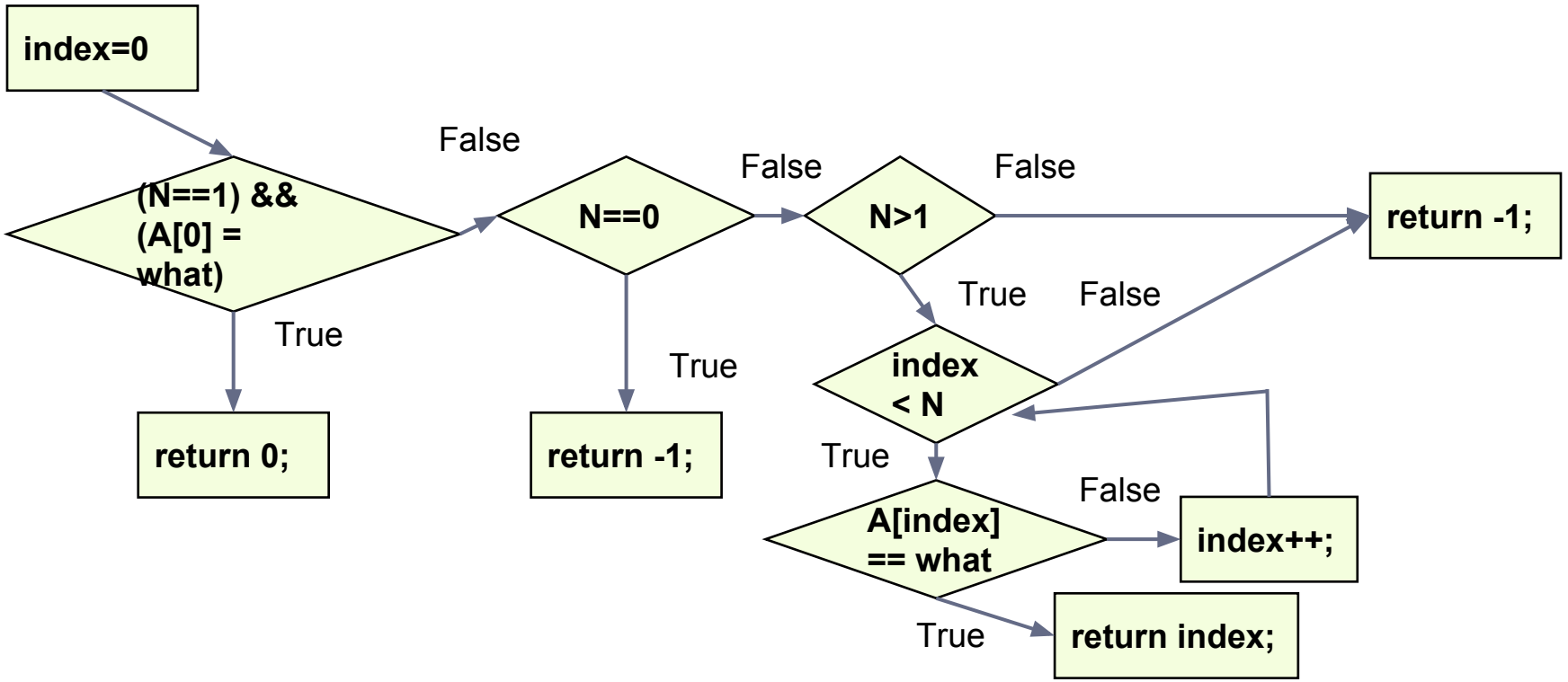
Test Case	A	B	(A and B)
1	True	True	True
2	True	False	False
3	False	True	False
4	False	False	False

Activity

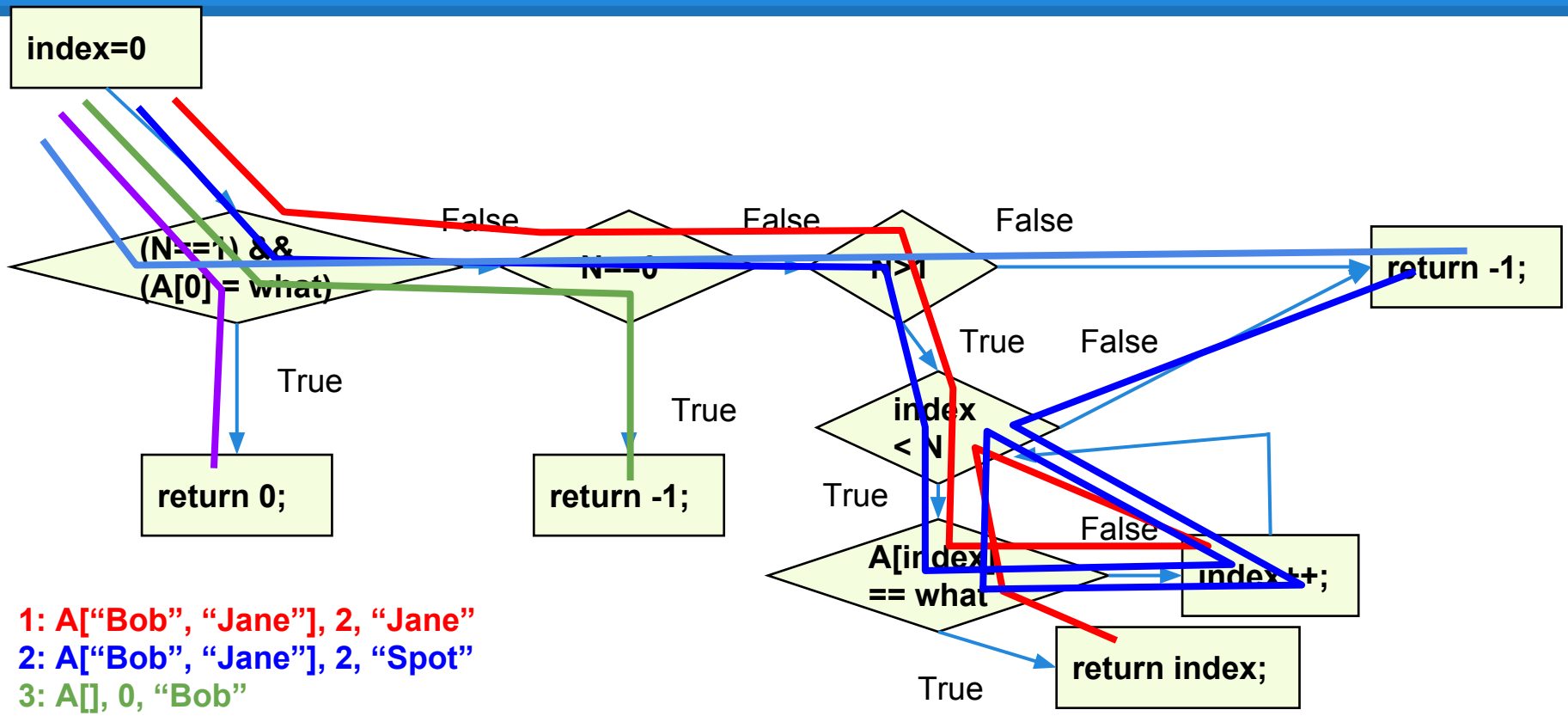
Draw the CFG and write tests that provide statement, branch, and basic condition coverage over the following code:

```
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;
}
```

Activity


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

Activity - Possible Solution



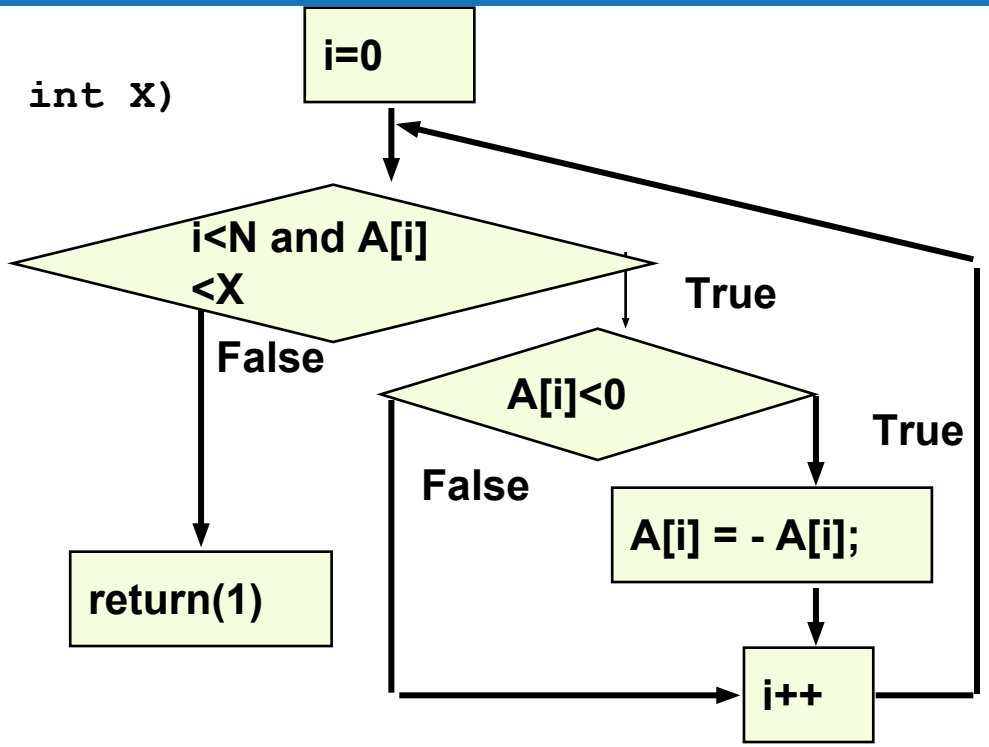
1: A[“Bob”, “Jane”], 2, “Jane”
2: A[“Bob”, “Jane”], 2, “Spot”
3: A[], 0, “Bob”
4. A[“Bob”], 1, “Bob”
5. A[“Bob”], 1, “Spot”

Path Coverage

- Other criteria focus on single elements.
  - However, all tests execute a sequence of elements - a path through the program.
  - Combination of elements matters - interaction sequences are the root of many faults.
- Path coverage requires that all paths through the CFG are covered.
- Coverage = $\frac{\text{Number of Paths Covered}}{\text{Number of Total Paths}}$

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);
}
```



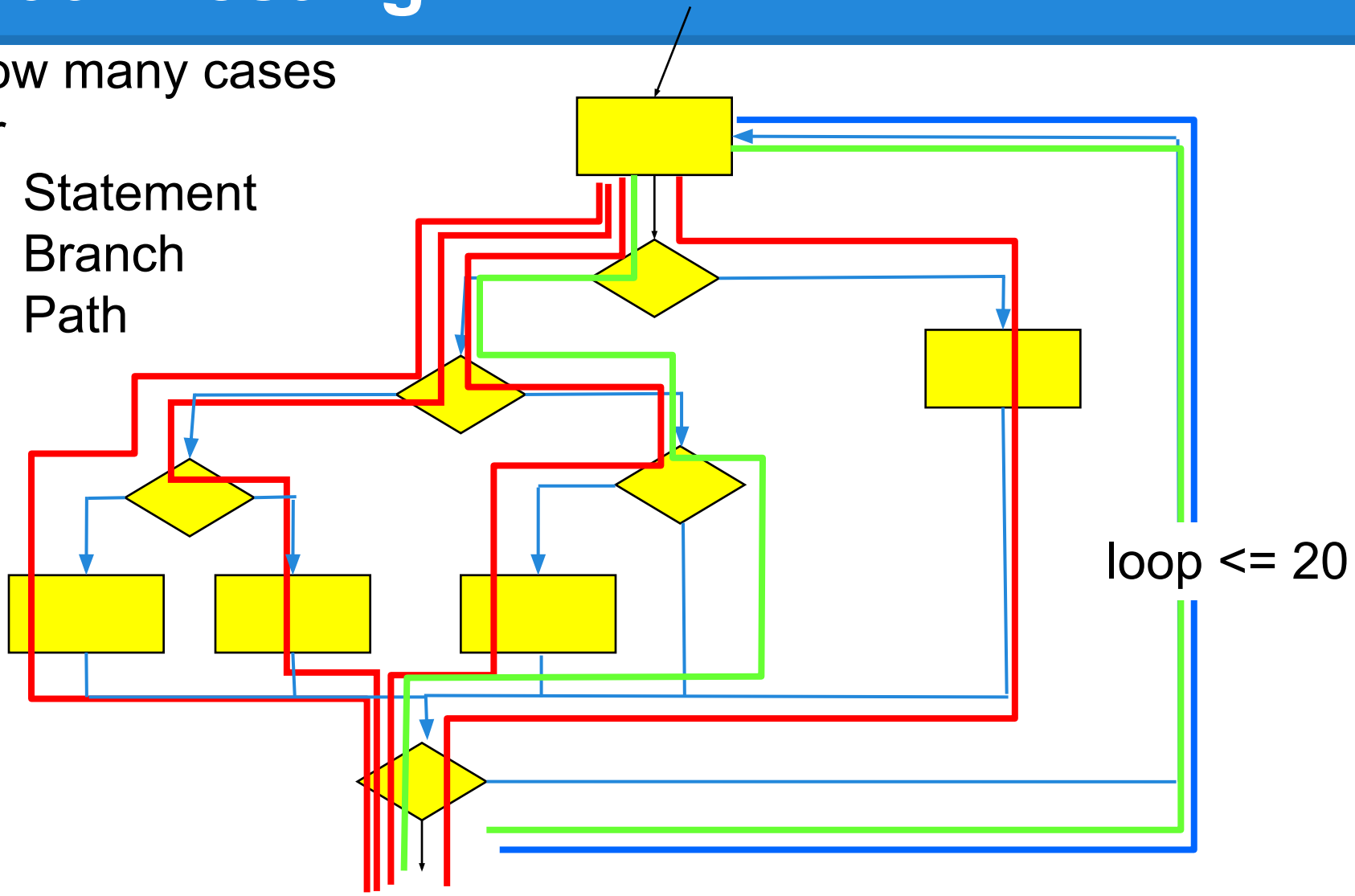
In theory, path coverage is the ultimate coverage metric.
In practice, it is impractical.

- **How many paths does this program have?**

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**.

However, there are ways to get some of the benefits of path coverage without the cost...

We Have Learned

- Test adequacy metrics let us “measure” how good our testing efforts are.
  - They prescribe test obligations that can be used to remove inadequacies from test suites.
- Code structure is used in many adequacy metrics. Many different criteria, based on:
  - Statements, branches, conditions, paths, etc.
- Coverage metrics tuned towards particular types of faults. Some are theoretically stronger than others, but are also more expensive and difficult to satisfy.

Next Time

- Next time - more on structural coverage
  - Path-based Metrics
  - Procedure Coverage
  - The Infeasibility Problem
  - Limitations of Coverage Metrics

- Homework 4
  - Any questions?