



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Lecture 9: Test Adequacy and Structural Testing

Gregory Gay
DIT635 - February 17, 2021

We Will Cover

- Test Adequacy Criteria
- Structural Testing:
 - Use coverage of the program structure to judge existing tests and create new tests.
 - Common structural coverage metrics:
 - Statement Coverage, Branch Coverage, Condition Coverage, Path Coverage

**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.

November 2020

| Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
|--------|--------|---------|-----------|----------|--------|----------|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | | | |
| 29 | 30 | | | | | |

www.a-print



Test Adequacy Criteria

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



- Can we measure “good testing”?
- **Test adequacy criteria** “score” testing efforts by measuring the completion of **test obligations**.
 - Checklists of properties that must be met by test cases.

(In)Adequacy Criteria

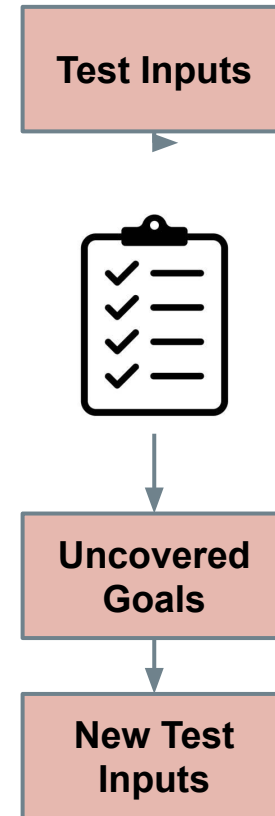
- Criteria identify **inadequacies** in the tests.
 - If no test reaches a statement, test suite is inadequate for finding faults in that statement.
 - If we plant a fake fault and no test exposes it, our tests are inadequate at detecting that fault.
- Tests may still miss faults, but maximizing criteria shows that tests at least meet certain goals.

Adequacy Criteria

- Adequacy Criteria based on coverage of factors correlated to finding faults (hopefully).
 - Exercising elements of source code (**structural testing**).
 - Detection of planted faults (**fault-based testing**)
- Widely used in industry - easy to understand, cheap to calculate, offer a checklist.
 - Enable tracking of “testing completion”
 - Can be measured in IntelliJ, Eclipse, etc.

Use of Criteria

- Measure adequacy of existing tests
 - Create additional tests to cover missed obligations.
- Create tests directly
 - Choose specific obligations, create tests to cover those.
 - Targets for automated test generation.



Structural Testing

Structural Testing

- The structure of the software itself is a valuable source of information.
- Prescribe how code elements should be executed, and measure coverage of those elements.
 - If-statements, Boolean expressions, loops, switches, paths between statements...

```
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 A;
}
```

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 covering gaps in the source code.

Structural Does Not Replace Functional

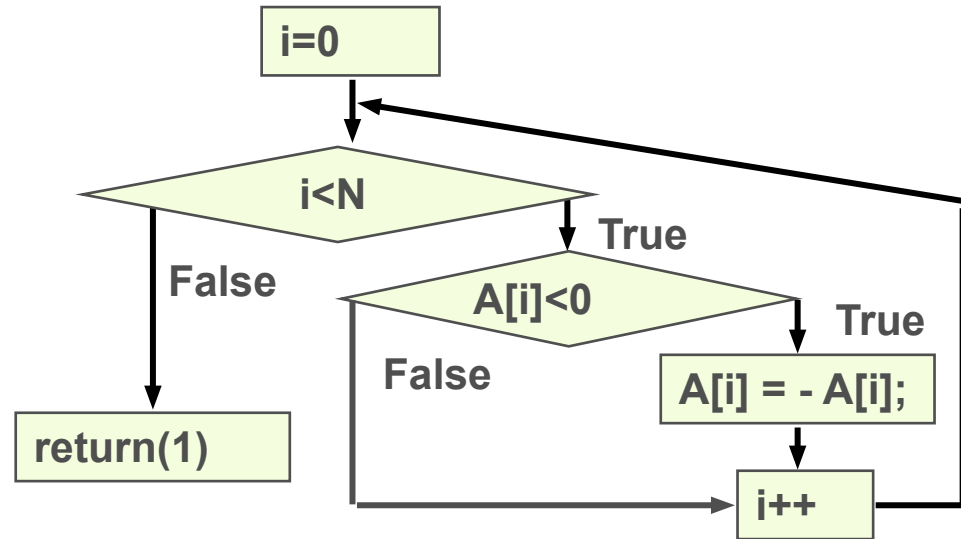
- **Should not be the basis for all test cases**
- Harder to make verification argument.
 - Do not map as directly to requirements.
- Does not expose missing functionality.
- Useful for supplementing functional tests.
 - Functional tests good at exposing conceptual faults.
 - Structural tests good at exposing coding mistakes.

Control and Data Flow

- We need to understand how system executes.
 - Conditional statements result in branches in execution, jumping between blocks of code.
- Control flow: how control passes through code.
 - Which code is executed, and when.
- Data flow: how data passes through code.
 - How variables are used in different expressions.

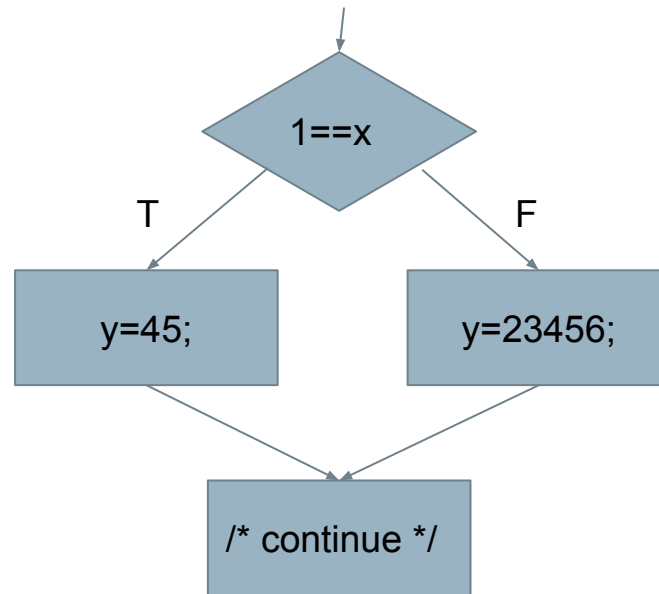
Control-Flow Graphs

- Directed graph representing flow of control.
- Nodes represent blocks of sequential statements.
- Edges connect nodes in the sequence they are executed.
 - Multiple edges indicate conditional statements.



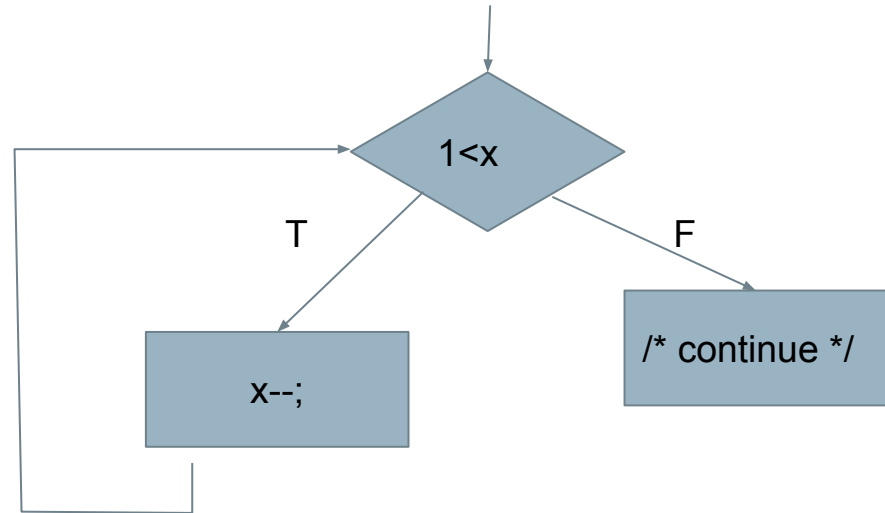
Control Flow: If-then-else

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



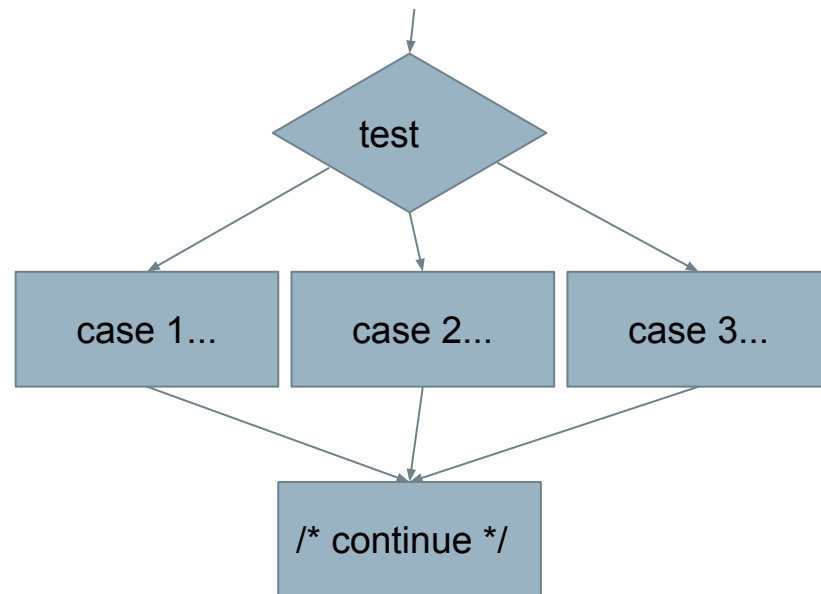
Loop

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



Case

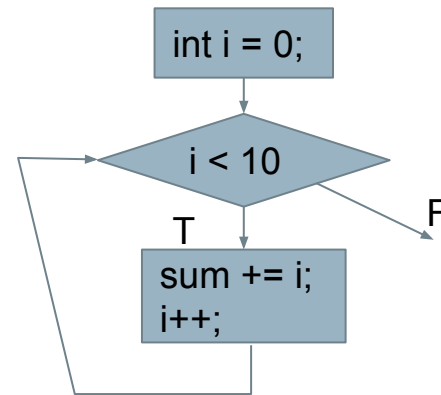
```
1 switch (test) {  
2     case 1 : ...  
3     case 2 : ...  
4     case 3 : ...  
5 }  
6 /* continue */
```



Basic Blocks

- Nodes are basic blocks.
 - Set of sequential instructions with single entry and exit point.
- Typically adjacent statements, but one statement might be broken up to model control flow in the statement.

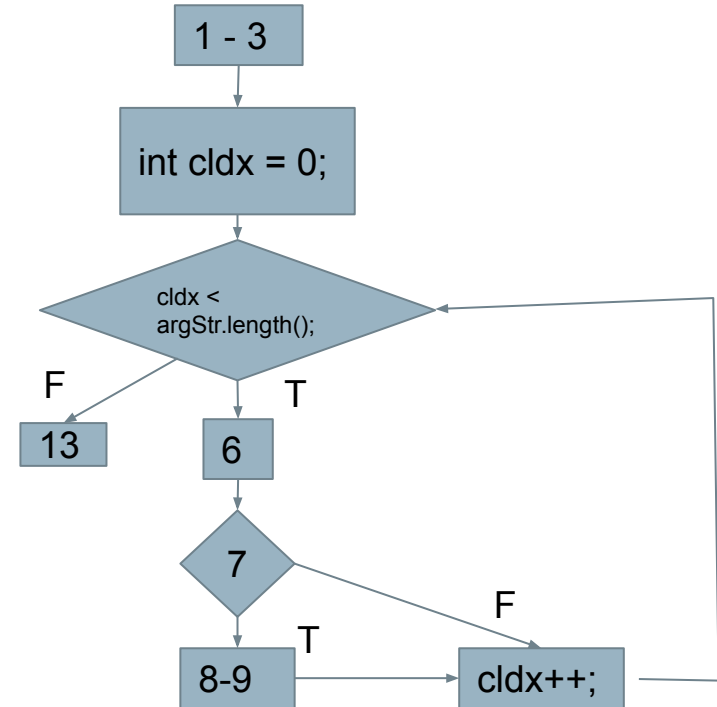
```
for(int i=0; i < 10; i++){  
    sum += i;  
}
```



Control Flow Graph Example

```

1. public static String collapseNewlines(String argSt){
2.     char last = argStr.charAt(0);
3.     StringBuffer argBuf = new StringBuffer();
4.
5.     for(int cldx = 0; cldx < argStr.length(); cldx++){
6.         char ch = argStr.charAt(cldx);
7.         if (ch != '\n' || last != '\n'){
8.             argBuf.append(ch);
9.             last = ch;
10.        }
11.    }
12.
13.    return argBuf.toString();
14. }
  
```



Structural Coverage Criteria

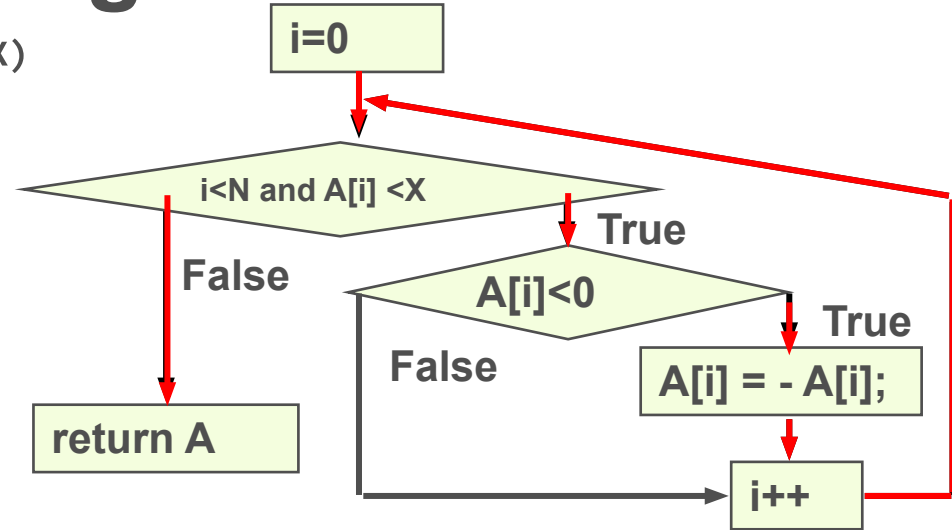
- Criteria based on exercising:
 - Statements (nodes of CFG)
 - Branches (edges of CFG)
 - Conditions
 - Paths
 - ... and many more
- Measurements used as adequacy criteria

Statement Coverage

- 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 A;
}
```



Can cover in one test: [-1], 1, 10

A Note on Test Suite Size

- Coverage not 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 often more important than ***whether*** it was executed.

Test Suite Size

- Favor large number of targeted tests over small number of tests that cover many statements.
 - If a test targets a small number of obligations, it is easier to tell where a fault is.
 - If a test covers a large number of obligations, we get higher coverage, but at the cost of being able to identify and fix faults.
 - The exception - 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

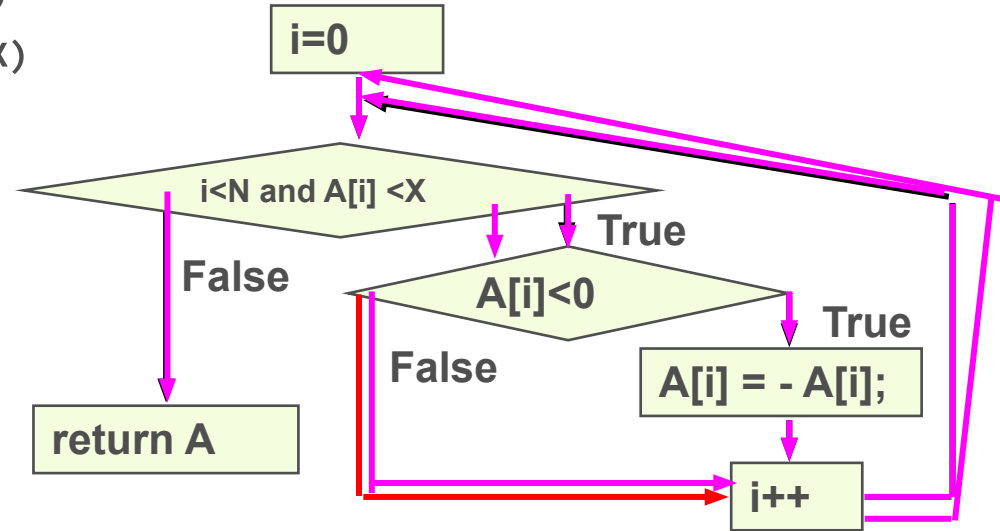
- Criterion A subsumes another Criterion B if, for every program P , every test suite satisfying A also satisfies B on P .
 - If we satisfy A, there is no point in measuring B.
- Branch coverage subsumes statement coverage.
 - Covering all edges in CFG requires covering all nodes in a graph.

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 A;
}
```



- $([-1], 1, 10)$ leaves one edge uncovered.
- $([-1, 1], 2, 10)$ achieves Branch Coverage.

Let's take a break.

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);`

Decisions and Conditions

- A *decision* is a complex Boolean expression.
 - Made up of *conditions*
 - Connected with Boolean operators (and, or, xor, not):
 - Boolean variables: `Boolean b = false;`
 - Subexpressions that evaluate to true/false involving (<, >, <=, >=, ==, and !=): `Boolean x = (y < 12);`

Decision Coverage

- Branch Coverage deals with a subset of decisions.
 - Branching decisions that decide how control is routed through the program.
- Decision coverage requires that all boolean decisions evaluate to true and false.
- Coverage =
$$\frac{\text{Number of Decisions Covered}}{\text{Number of Total Decisions}}$$

Basic Condition Coverage

- Several coverage metrics examine the individual *conditions* that make up a *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 =
$$\frac{\text{Number of Truth Values for All Conditions}}{2 \times \text{Number of Conditions}}$$

Basic Condition Coverage

- Make each condition both True and False

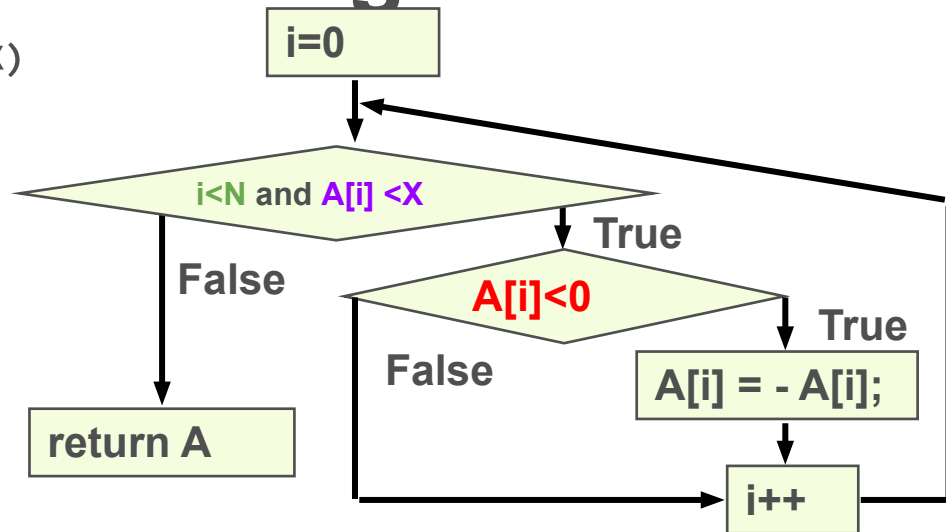
(A and B)

| Test Case | A | B |
|-----------|-------|-------|
| 1 | True | False |
| 2 | False | True |

- Does not require hitting both branches.
 - 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 A;
}
```



- `([-1, 1], 2, 10)`
 - Negative value in array
 - Positive value (but < X)
- `([11], 1, 10)`
 - Positive, but > X
- Both eventually cause `i < N` to be false.

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.
 - 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 |

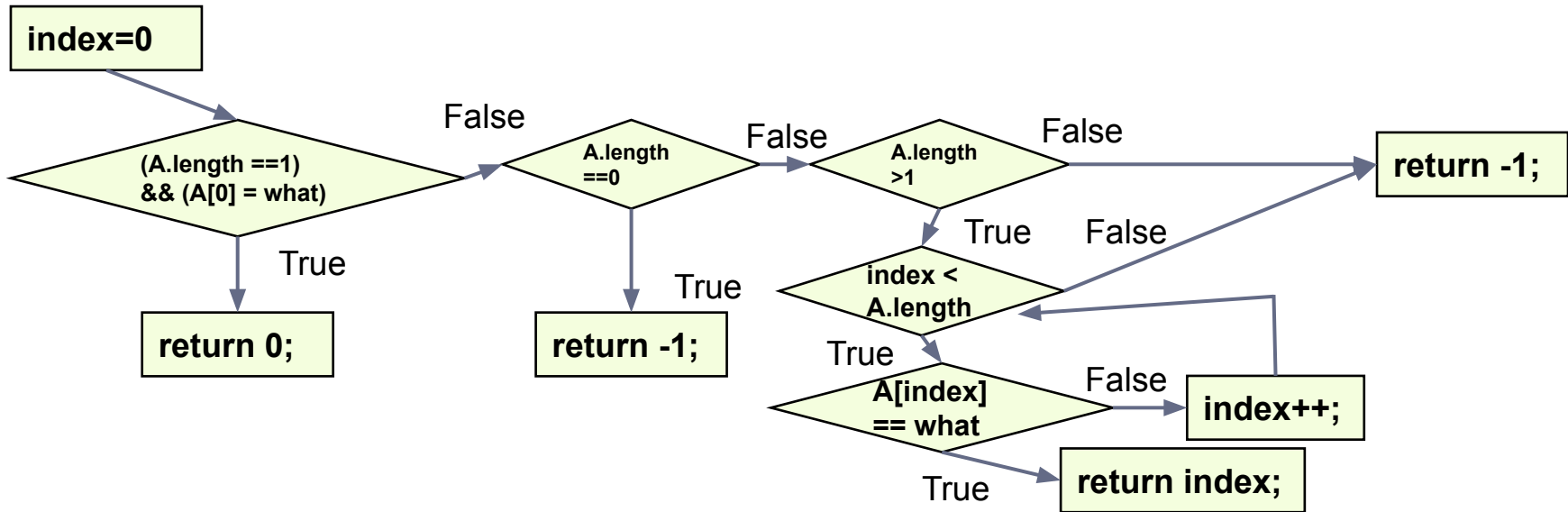
Activity

<https://bit.ly/3rL8s7t>

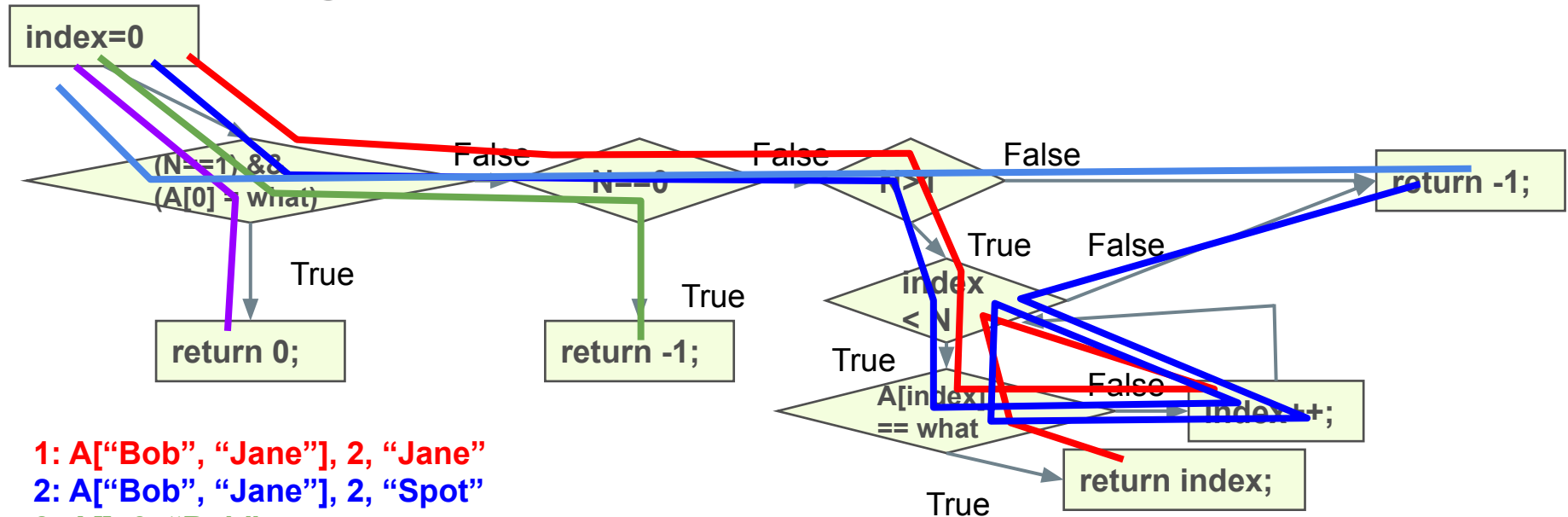
Draw the CFG and write tests that provide statement, branch, and basic condition coverage over the following code:

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

Activity - Control Flow Graph



Activity - Possible Solution



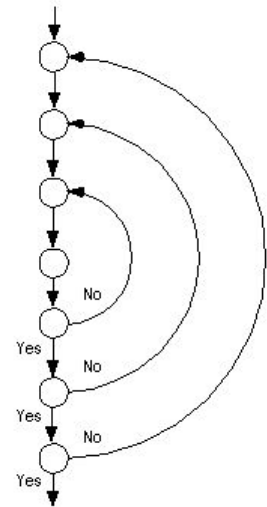
- 1: A["Bob", "Jane"], 2, "Jane"
- 2: A["Bob", "Jane"], 2, "Spot"
- 3: A[], 0, "Bob"
- 4: A["Bob"], 1, "Bob"
- 5: A["Bob"], 1, "Spot"

Loop Boundary Coverage

- Focus on problems related to loops.
 - Cover scenarios representative of how loops might be executed.
- For each loop, write tests that:
 - Skip the loop entirely.
 - Take exactly one pass through the loop.
 - Take two or more passes through the loop.

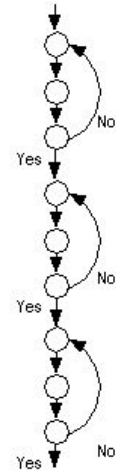
Nested Loops

- Often, loops are nested within other loops.
 - For each level, execute 0, 1, 2+ times
- In addition:
 - Test innermost loop first with outer loops executed minimum number of times.
 - Move one loops out, keep the inner loop at “typical” iteration numbers, and test this layer as you did the previous layer.
 - Continue until the outermost loop tested.



Concatenated Loops

- One loop executes. Next line of code starts a new loop. These are generally independent.
- If not, follow a similar strategy to nested loops.
 - Start with bottom loop, hold higher loops at minimal iteration numbers.
 - Work up towards the top, holding lower loops at “typical” iteration numbers.



Why These Loop Strategies?

- If proving correctness, we establish preconditions, postconditions, and invariants that are true on each execution of loop.
 - The loop executes zero times when the postconditions are true in advance.
 - The loop invariant is true on loop entry (one), then each loop iteration maintains the invariant (many).
 - (invariant and $\neg(\text{loop condition})$) implies postconditions are met
- Loop testing strategies echo these cases.

Activity: Binary Search

<https://bit.ly/3rL8s7t>

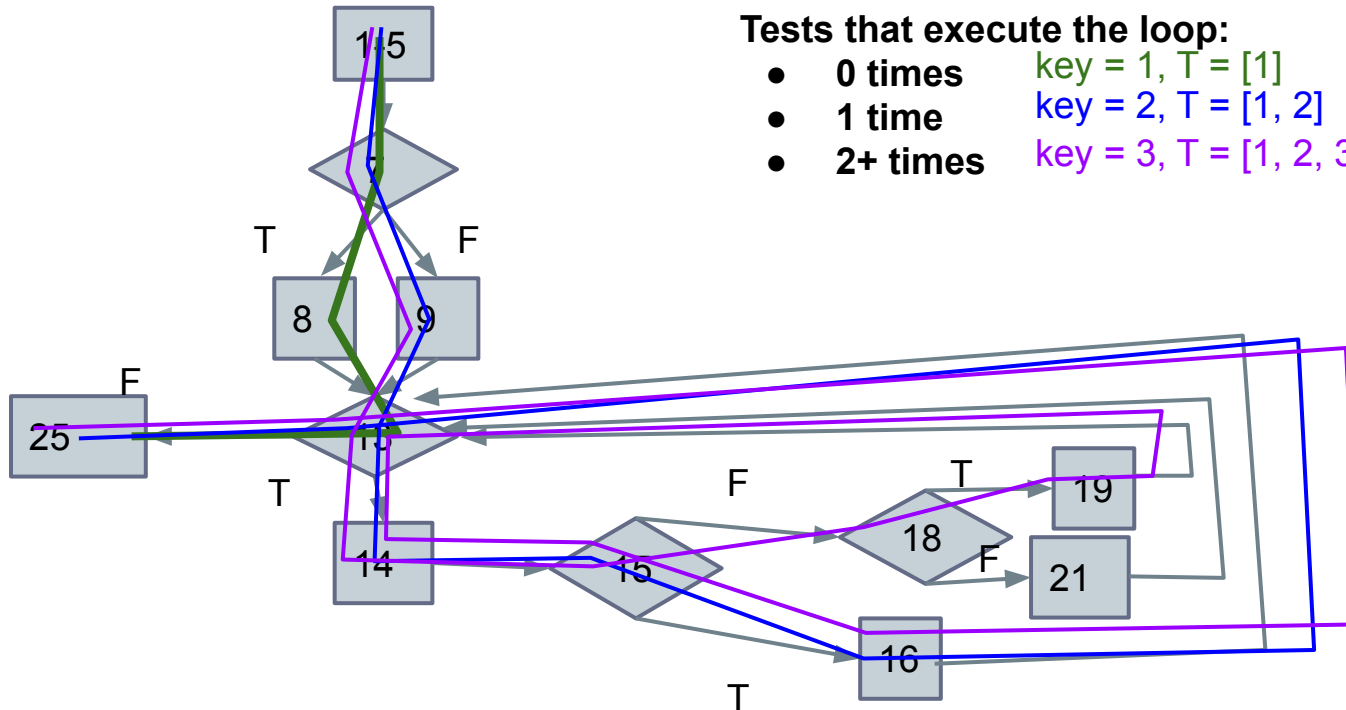
For the binary-search code:

1. Draw the control-flow graph for the method.
2. Develop a test suite that achieves loop boundary coverage (executes while loop 0, 1, 2+ times).

Activity: Binary Search

Tests that execute the loop:

- 0 times $\text{key} = 1, T = [1]$
- 1 time $\text{key} = 2, T = [1, 2]$
- 2+ times $\text{key} = 3, T = [1, 2, 3]$



The Infeasibility Problem

Sometimes, no test can satisfy an obligation.

- Impossible combinations of conditions.
- Unreachable statements as part of defensive programming.
 - Error-handling code for conditions that can't occur.
- Dead code in legacy applications.

The Infeasibility Problem

- Stronger criteria call for potentially infeasible combinations of elements.

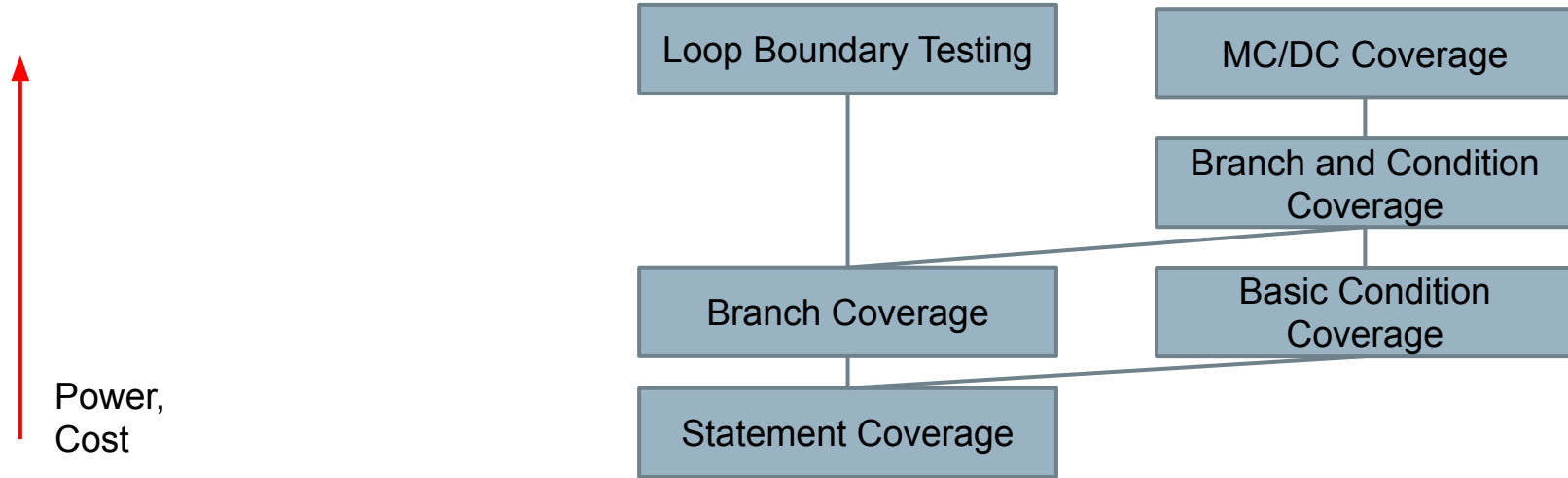
$$(a > 0 \ \&\& \ a < 10)$$

- It is not possible for both conditions to be false.
 - A is negative and greater than 10
- Loop boundary coverage - loop can't be skipped.

The Infeasibility Problem

- Adequacy “scores” based on coverage.
 - 95% branch coverage, 80% MC/DC coverage, etc.
 - Stop once a threshold is reached.
 - Unsatisfactory solution - elements are not equally important for fault-finding.
- Manual justification for omitting each impossible test obligation.
 - Helps refine code and testing efforts.
 - ... but very time-consuming.

Which Coverage Criterion Should I Use?



We Have Learned

- Test adequacy metrics let us “measure” how good our testing efforts are.
 - Prescribe test obligations that can be used to remove inadequacies from test suites.
- Code structure is used in many adequacy criteria.
- Many different criteria, based on:
 - Statements, branches, conditions, loops, etc.

Next Time

- Next class: More examples of structural coverage and data-flow criteria
 - Optional Reading - Pezze and Young, Chapters 6 and 13
- Friday Exercise Session: Structural Coverage
 - Using Meeting Planner code
- Homework - Assignment 2 due Feb 28



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY