Topic 8

Software Testing

Part 2

Structural Testing

- aka Path Analysis
- a white-box or glass-box testing technique
  - we are looking at the code
- Is the program (ie. test suite) “hitting” every path of execution

Structural Testing

- Levels of code coverage
  - C_0 →→ Statement/Line coverage
  - C_1 →→ Branch coverage
  - C_2 →→ Condition coverage
  - C_3 →→ Multiple condition coverage
  - C_4 →→ Path coverage

Statement Coverage

```
public static String classify(int x) {
   boolean pos = true;
   boolean even = true;
   if (x > 0)
      pos = true;
   if (x % 2 == 0)
      even = true;
   return String.format("Number:%1$d,
            Positive:%2$b,Even:%3$b", x, pos, even);
}
```

String returned should contain:

- "Positve:true" if the number is positve, ":false" otherwise
- "Even:true" if the number is even, ":false" otherwise

Statement Coverage

```
@Test
public void testClassify() {
   String result = Ex1.classify(2);
   assertThat(result,
               containsString("Positive:true"));
   assertThat(result, containsString("Even:true"));
}
```

Choose test case for value 2

Will this pass? Yes.

What about for value 3?

Statement Coverage

$$C_0 = \frac{|\text{Statements exercised}|}{|\text{Statements}|}$$

Strategy: find paths that cover all statements; write test cases to exercise those paths

- Test suite should have $C_0 = 1$ (ie. 100%)
- Least restrictive of the coverage criteria
  - Some branches may be missed
- Helps measure correctness of code written
  - Better than nothing

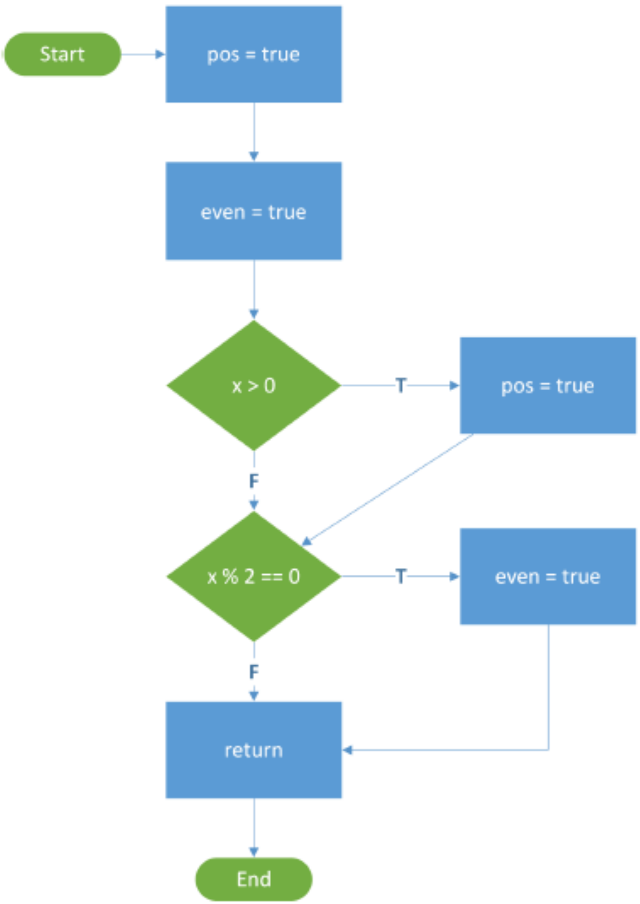
Structural Testing

- But, structural testing is not functional testing...
- The method is not correct
  - still need to test correct functionality

Program Flowcharts

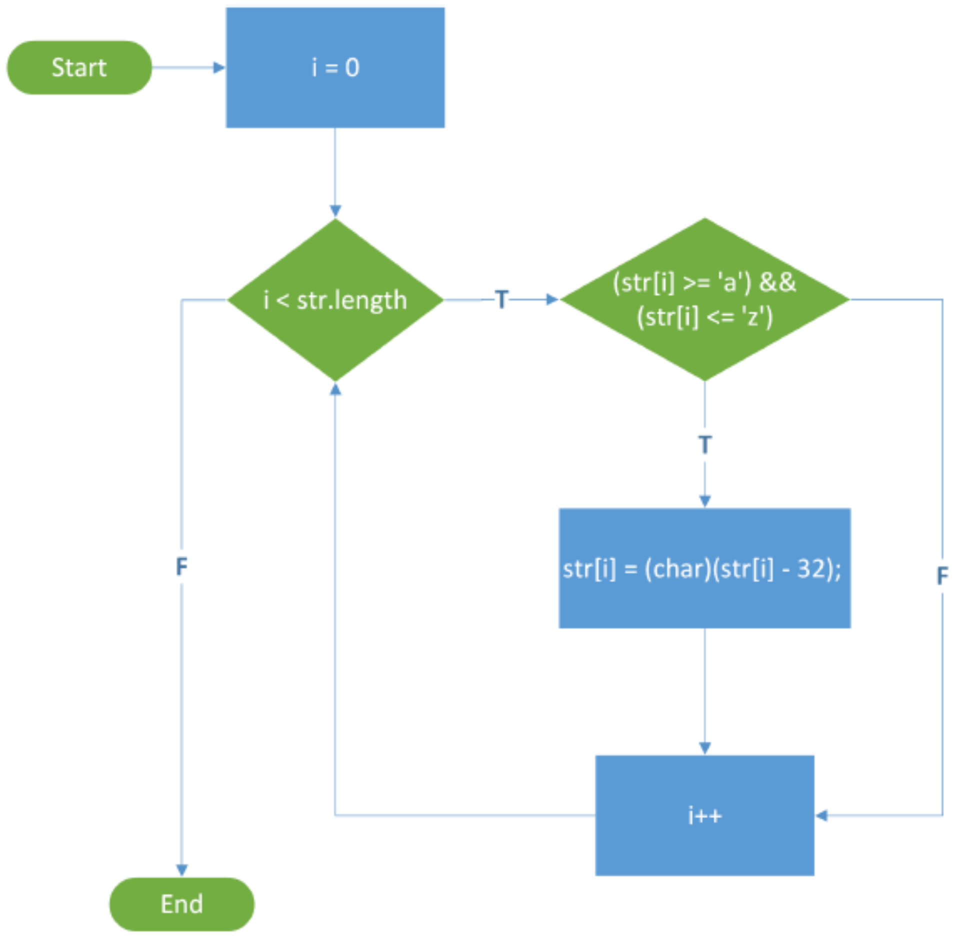
- Labelled, directed graph, where
  - statements are represented by rectangle nodes
  - decisions are represented by diamond nodes

Program Flowcharts



- Nodes
  - statements
- Edges
  - indicate parts of paths which may be followed through the code
- Labels on edges
  - indicate flow direction when a condition is true(T) vs false(F)

Program Flowcharts



Structural Testing

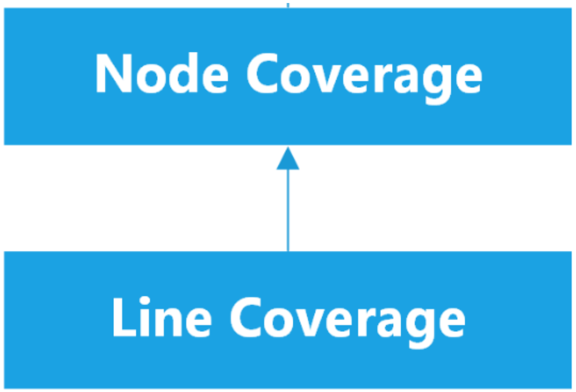
- If our test suite executes all statements, has visited every node in the flowchart
  - test suite achieves full/100% node coverage or covered every node
    - (equivalent to statement coverage)
- Note: Statement = Line Coverage
  - Any line containing code is measured
    - Considered covered if any code on the line is executed
  - Not exactly the same thing as statement coverage
    - Can have more than one statement per line
  - Most people mean statement coverage when they say line coverage

Structural Testing

100% node coverage implies 100% line coverage

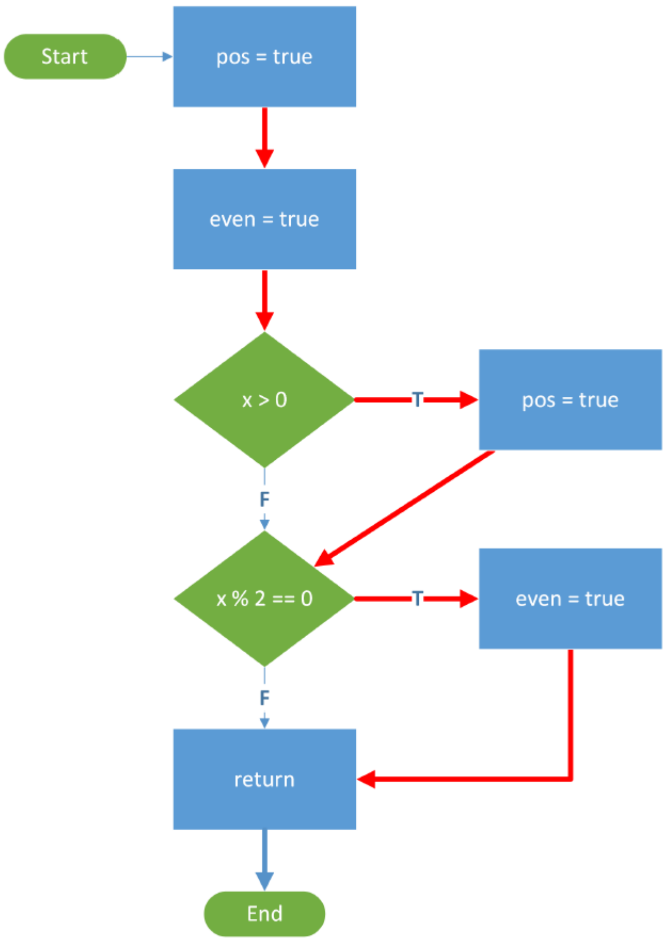
- The converse is not true:
  - 100% line coverage does not imply 100% node coverage

Node (statement) coverage is stronger than line coverage



Example

- Consider the first example flowchart
  - Test case: x = 2
  - All nodes visited
    - 100% statement coverage
  - However, neither F branch has ever been taken



Edge Coverage

- If our test suite follows every edge in the flowchart
  - We say that the test suite covers every edge
  - We can also say that it covers every decision (or branch)
- Branch coverage:
  - Has every branch of each control structure (if, switch) been executed?
  - Equivalently, has every edge in the program flowchart been executed?

Edge Coverage

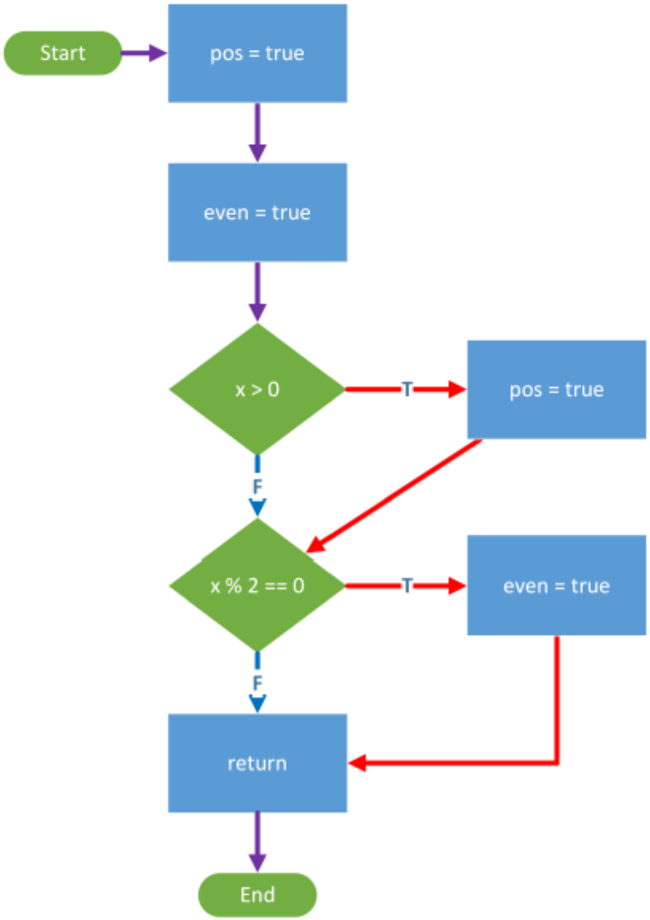
- More thorough than statement coverage
  - Catches more problems
- Example:
  - We had 100% statement coverage for the first flowchart, but the bug was not detected
  - Would we have detected the bug if we followed every edge? Let’s write some tests and see.

Edge Coverage

```
@Test
public void testClassifyPositiveEven() {
   String result = Ex1.classify(2);
   assertThat(result,
                  containsString("Positive:true"));
   assertThat(result,
                  containsString("Even:true"));
}

@Test
public void testClassifyNegativeOdd() {
   String result = Ex1.classify(-1);
   assertThat(result,
                  containsString("Positive:false"));
   assertThat(result,
                  containsString("Even:false"));
}
```

Edge Coverage



- We'll test with the following test cases:
  - x = 2
  - x = -1
  - A purple edge indicates an edge followed by both test cases
- This test suite will give us 100% edge coverage.

Coverage Terminology

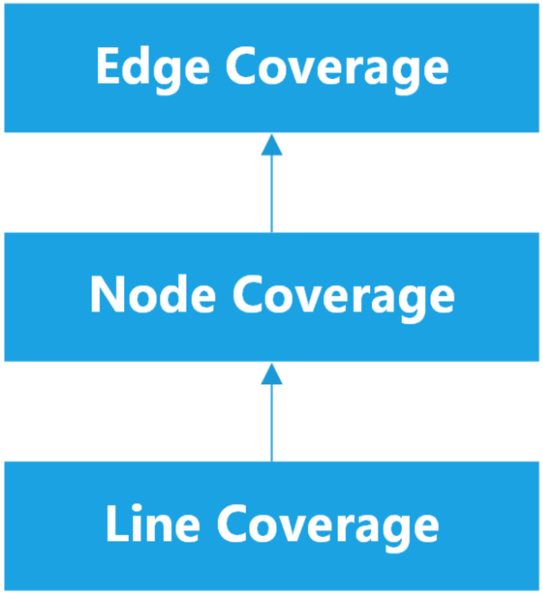
- If a test suite executes 100% of all statements, we say it achieves 100% node coverage (or statement coverage)
- If a test suite follows 90% of all edges, we say it achieves 90% edge coverage

Edge Coverage

100% edge coverage implies 100% node coverage

- If we followed every edge, we must have visited every node
- The converse is not true:
  - 100% node coverage does not imply 100% edge coverage

Edge coverage is stronger than node coverage



Minimal Test Suites

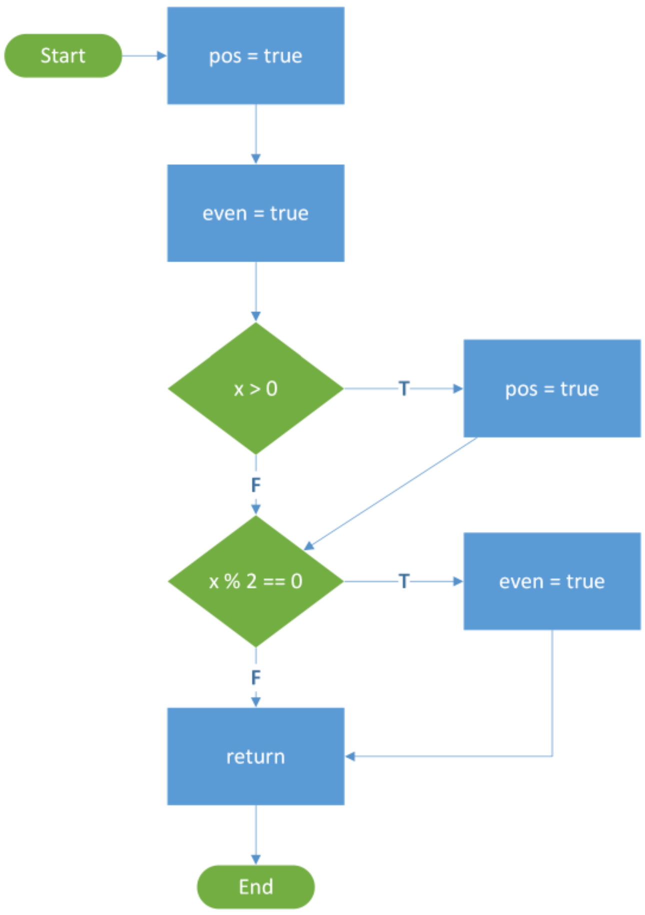
- More test cases = higher coverage
  - Larger test suites take longer to run
  - Often want to find some minimal test suite
    - Which still achieves 100% coverage
  - Important to define what minimal means
    - Minimal test cases?
    - Minimal number of inputs used?
    - Minimal time to run?

Minimal Test Suites Example

Suppose we want 100% edge coverage

- Test suite: x = 2, x = 1, x = -2
  - Requires 3 test cases
- Test suite: x = 2, x = -1
  - Requires only 2 test cases

- Not possible to have smaller test suite, with only 1 test case
  - Must cover both decisions T and F edges
- x = 2, x = -1 is a minimal test suite for 100% edge coverage
  - in the sense of needing the fewest test cases



Minimal Test Suites Example

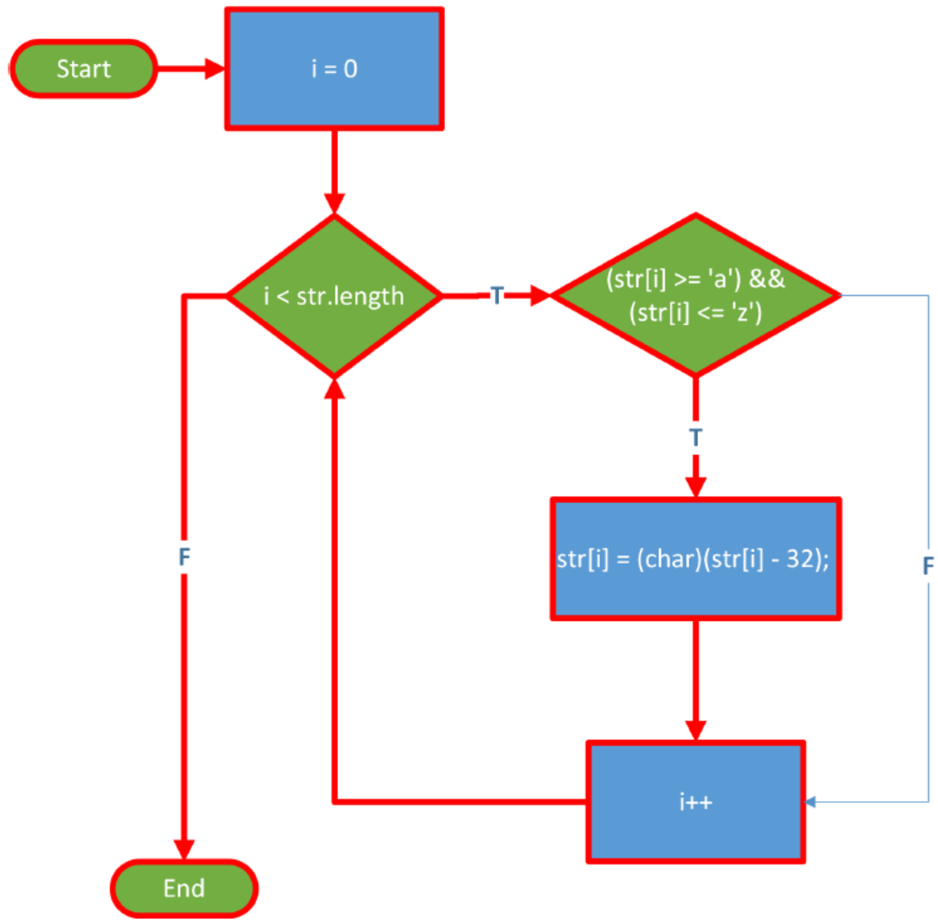
100% node coverage:

1) {‘a’}

- but F branch of the inner condition is not taken

100% edge coverage:

1) {‘a’} {‘X’}
2) {‘a’, ‘X’}



Minimal Test Suites

- Which of (1) and (2) is minimal for edge coverage?
  - Depends on how we define minimality
    - (1) has 2 tests, but each has just 1 character
    - (2) has just 1 test, but it has 2 characters

Minimal Test Suites

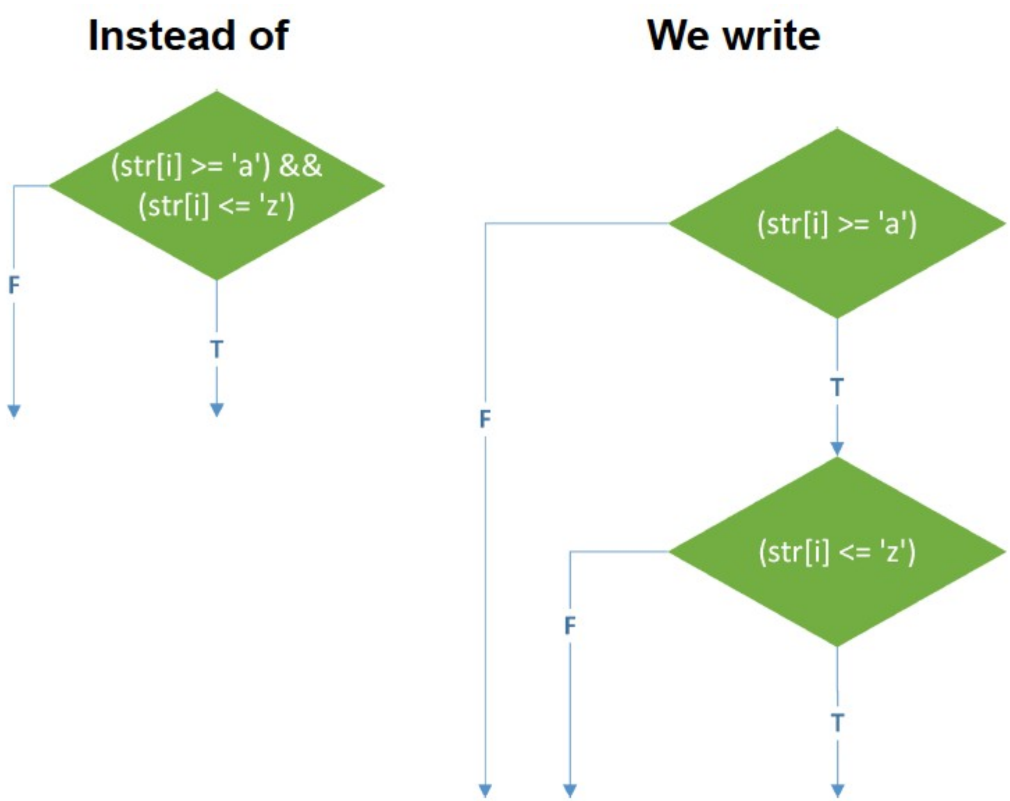
- minimality for a test suite should always be defined
  - Here, it makes more sense to accept (2) as minimal
  - Fewer test cases = faster execution of the test suite

- For this problem, we could say that:
  - Test suite X is more minimal than test suite Y if either:
    - X has fewer test cases than Y; or
    - X has the same number of test cases as Y, but the total number of characters in X is less than in Y
      - Saves us from accepting {‘a’, ‘a’, ‘a’, ‘X’, ‘X’, ‘X’} as minimal

Condition Coverage

- Decision: everything in parentheses after the if, while
  - e.g. `((str[i] >= 'a') && (str[i] <= 'z'))`
- Condition: the individual terms of the decision
  - e.g `(str[i] >= 'a'), (str[i] <= 'z')`
    - are the two conditions
- So far, we've written each decision in a single diamond.
- If we divide the conditions within each decision into separate diamonds, we can get a better reflection of what the program does.

Flowcharts: Splitting up Decisions



Recall: Java shortcircuits logical operators && and ||

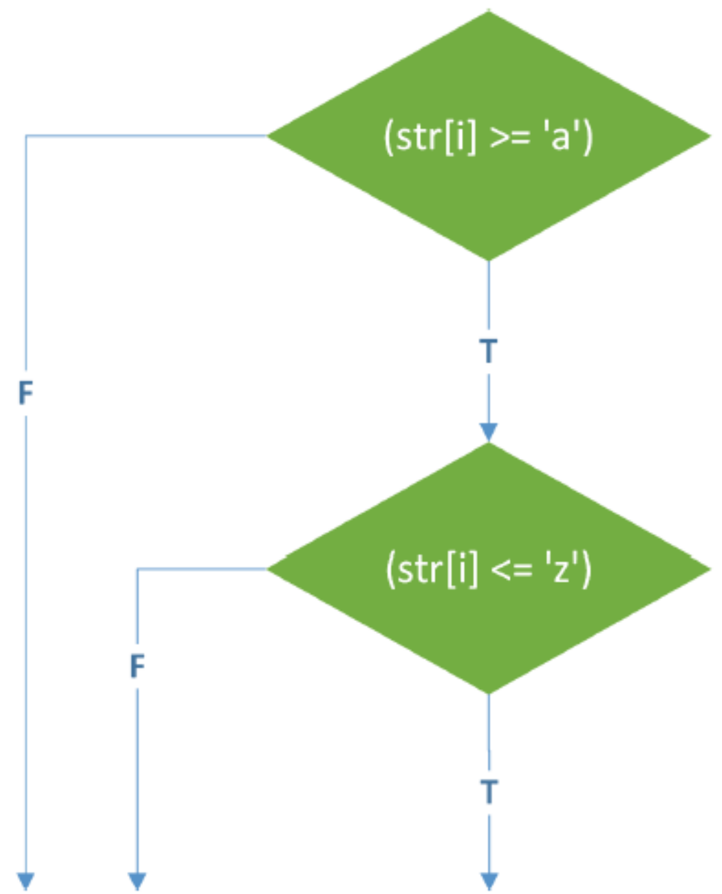
- Boolean expression evaluation stops as soon as final result can be determined

We now have two new edges that we'll need to cover

Condition Coverage

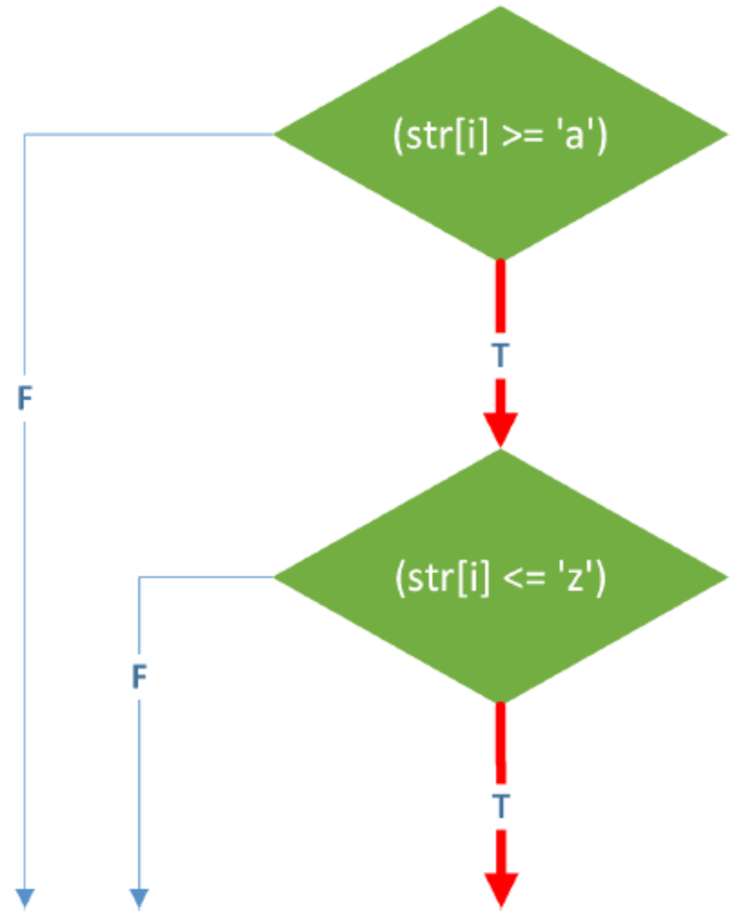
We split up all diamonds and then cover all edges

How many characters will be needed in our test input to achieve 100% condition coverage?



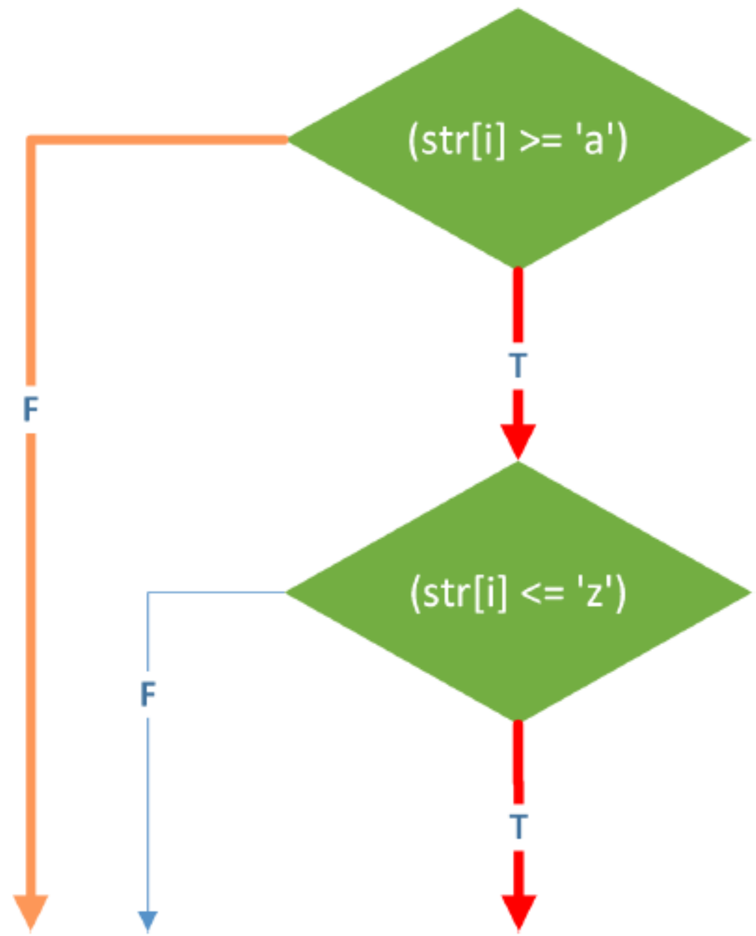
Condition Coverage

- Test Case
  – {‘a’}



Condition Coverage

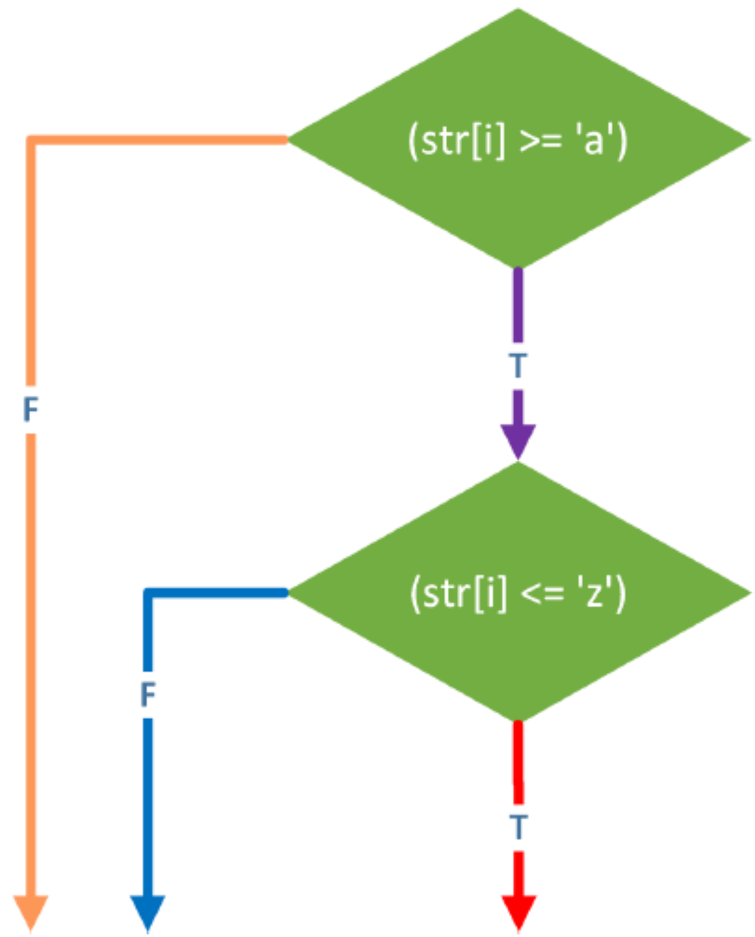
- Test Case
  - {‘a’, ‘X’}



Condition Coverage

To evaluate the second F edge, we simply need a character that is >= ‘a’ and also > ‘z’

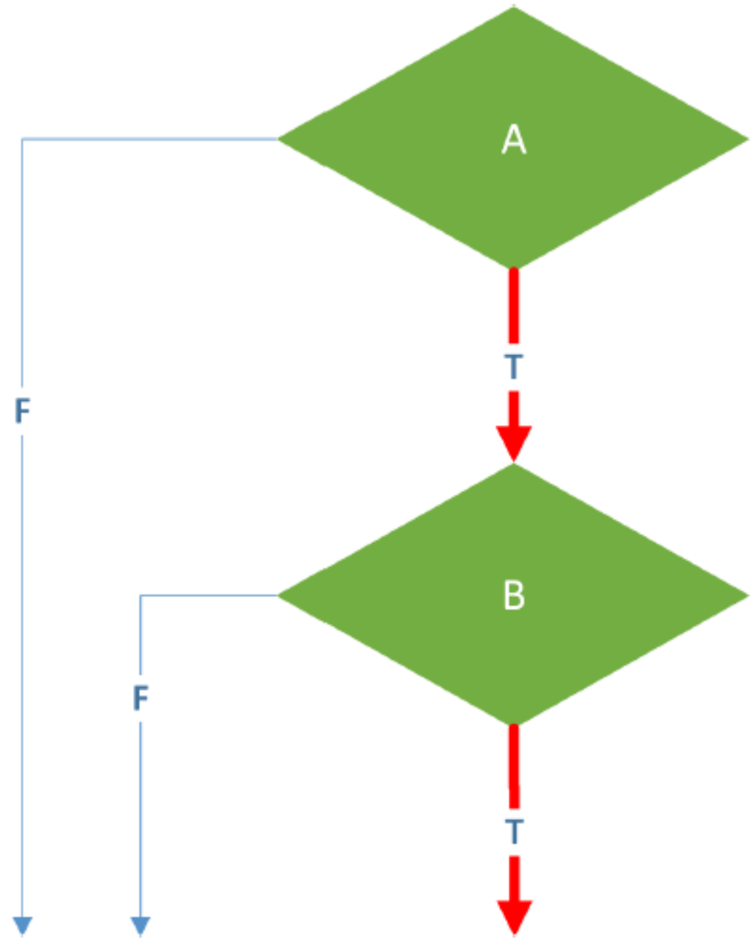
- Test Case
  - {‘a’, ‘X’, ‘~’}
- This test case achieves 100% condition coverage



Condition Coverage - A && B

In general, to achieve condition coverage for a decision A && B, we need to design test cases so that:

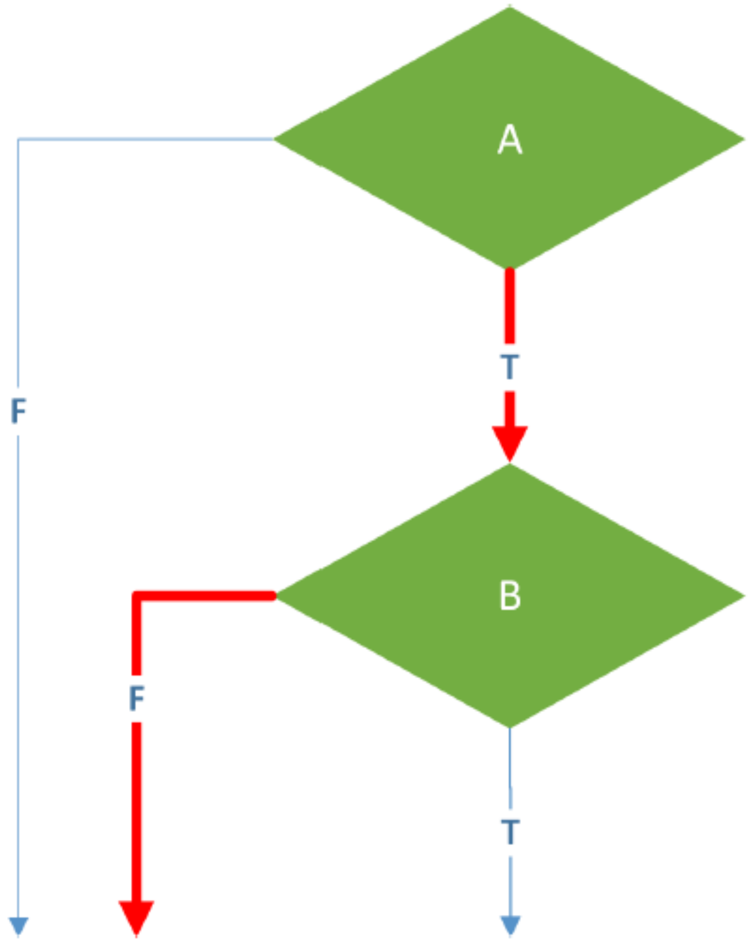
- `A = true, B = true`
- `A = true, B = false`
- `A = false`



Condition Coverage - A && B

In general, to achieve condition coverage for a decision A && B, we need to design test cases so that:

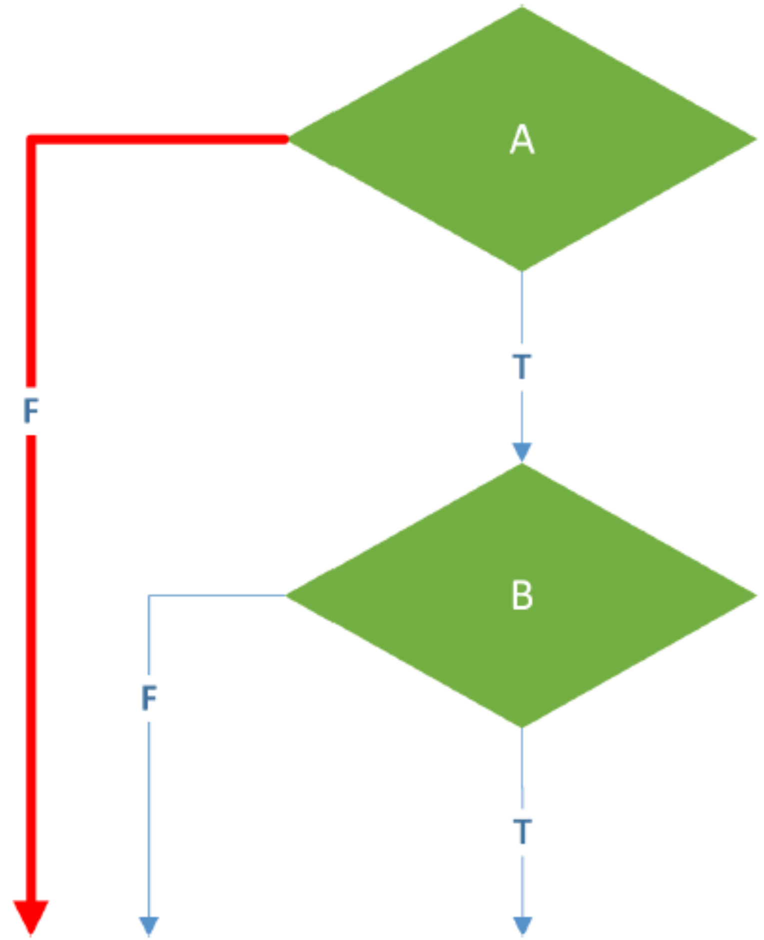
- `A = true, B = true`
- `A = true, B = false`
- `A = false`



Condition Coverage - A && B

In general, to achieve condition coverage for a decision A && B, we need to design test cases so that:

- `A = true, B = true`
- `A = true, B = false`
- `A = false`

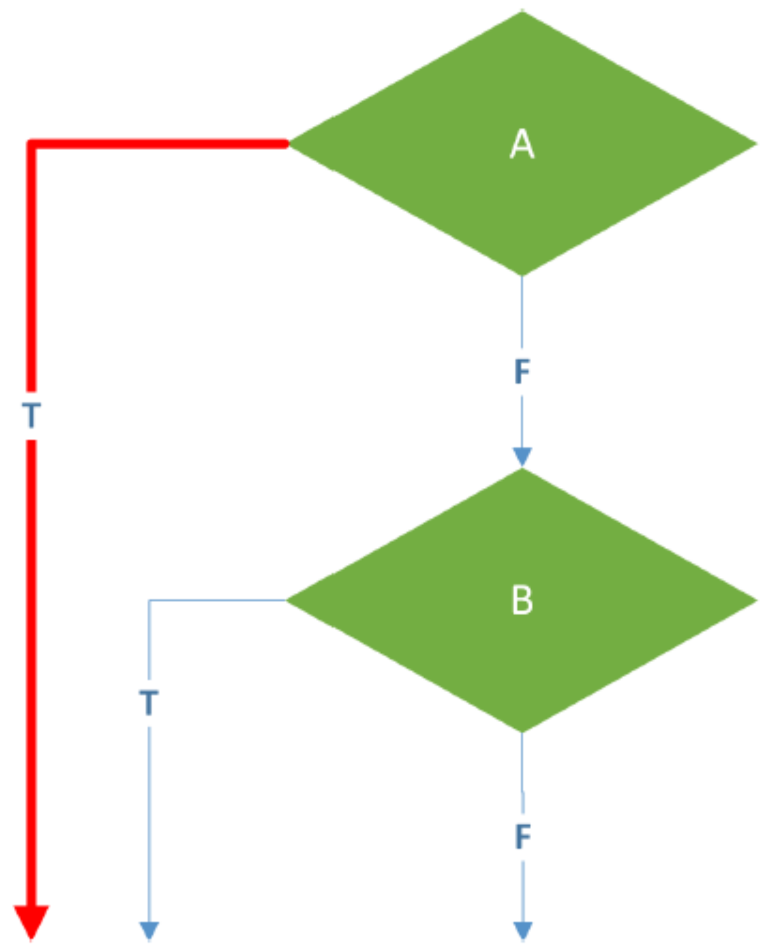


Unless A is true, B won't be evaluated, due to short-circuit evaluation

Condition Coverage - A || B

In general, to achieve condition coverage for a decision A || B, we need to design test cases so that:

- `A = true`
- `A = false, B = true`
- `A = false, B = false`

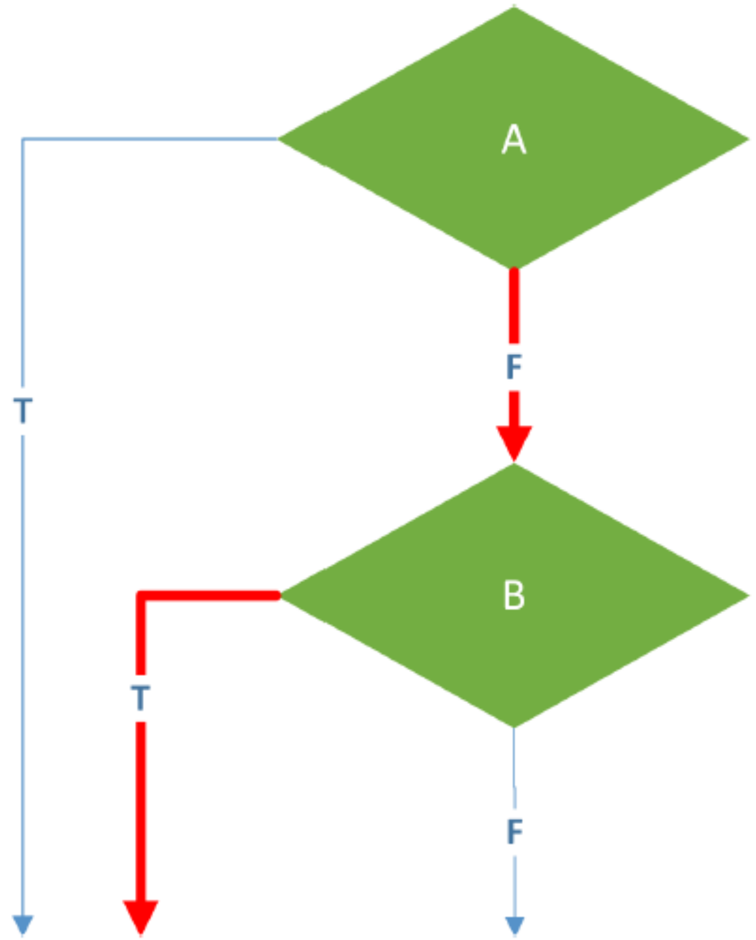


Unless A is false, B won't be evaluated, due to short-circuit evaluation

Condition Coverage - A || B

In general, to achieve condition coverage for a decision A || B, we need to design test cases so that:

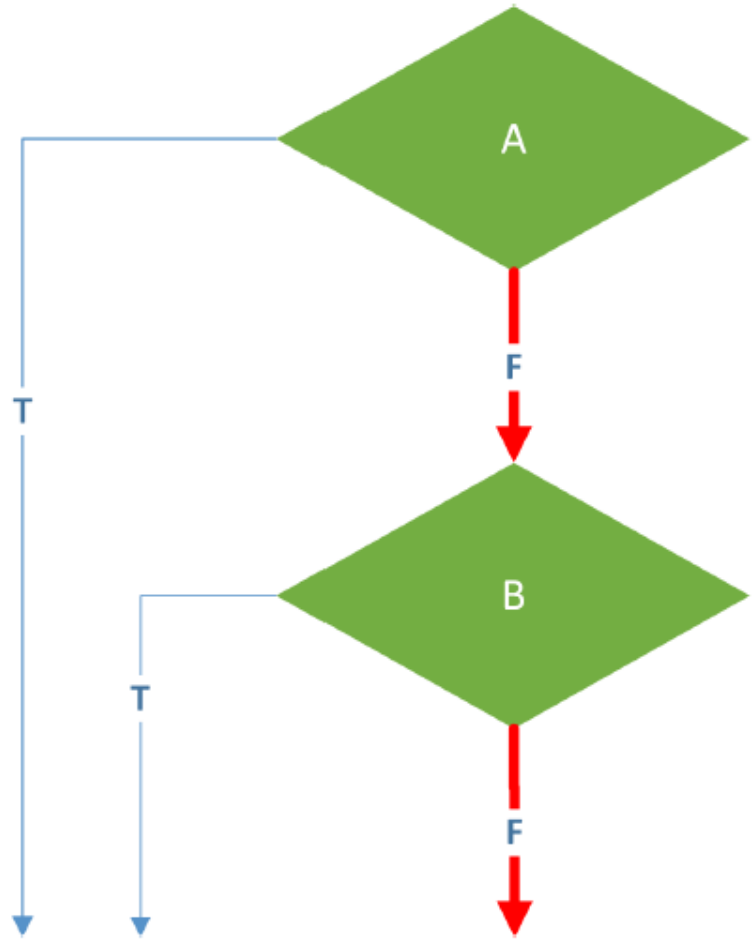
- `A = true`
- `A = false, B = true`
- `A = false, B = false`



Condition Coverage - A || B

In general, to achieve condition coverage for a decision A || B, we need to design test cases so that:

- `A = true`
- `A = false, B = true`
- `A = false, B = false`

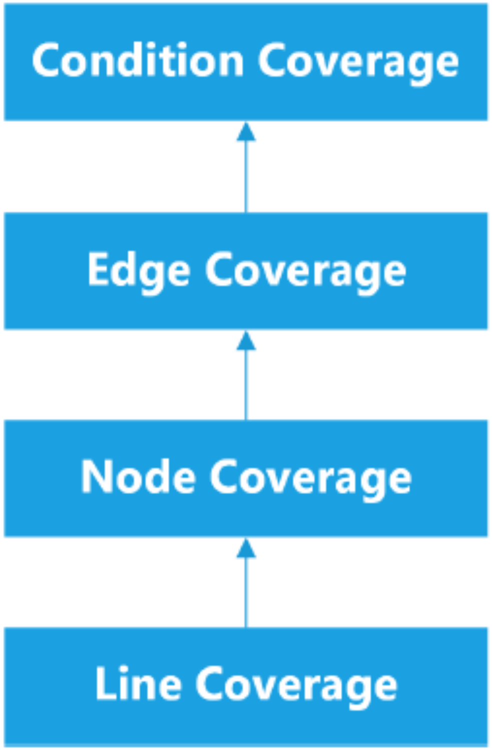


Condition Coverage

If we have 100% condition coverage

- We must have evaluated each condition of an if, while, etc. both ways
- Therefore, we must have evaluated each decision both ways

Thus, condition coverage is stronger than edge (decision) coverage



Example: Condition Coverage

Consider the following Boolean expression:

```
( ((x == 0) || (y > 4)) && ((z < 10) || (w == 0)) )
```

For brevity, let

```
A : (x == 0)
B : (y > 4)
C : (z < 10)
D : (w == 0)
```

Expression above is equivalent to:

```
(  ( A || B ) && ( C || D )  )
```

Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	T	T	T	T
2	T	T	T	F	T
3	T	T	F	T	T
4	T	T	F	F	F
5	T	F	T	T	T
6	T	F	T	F	T
7	T	F	F	T	T
8	T	F	F	F	F
9	F	T	T	T	T
10	F	T	T	F	T
11	F	T	F	T	T
12	F	T	F	F	F
13	F	F	T	T	F
14	F	F	T	F	F
15	F	F	F	T	F
16	F	F	F	F	F

Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	T	-	T
3	T	-	F	T	T
4	T	-	F	F	F
5	T	-	T	-	T
6	T	-	T	-	T
7	T	-	F	T	T
8	T	-	F	F	F
9	F	T	T	-	T
10	F	T	T	-	T
11	F	T	F	T	T
12	F	T	F	F	F
13	F	F	T	-	F
14	F	F	T	-	F
15	F	F	F	T	F
16	F	F	F	F	F

Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	T	-	F
8	F	F	F	T	F
9	F	F	F	F	F

Example: Condition Coverage

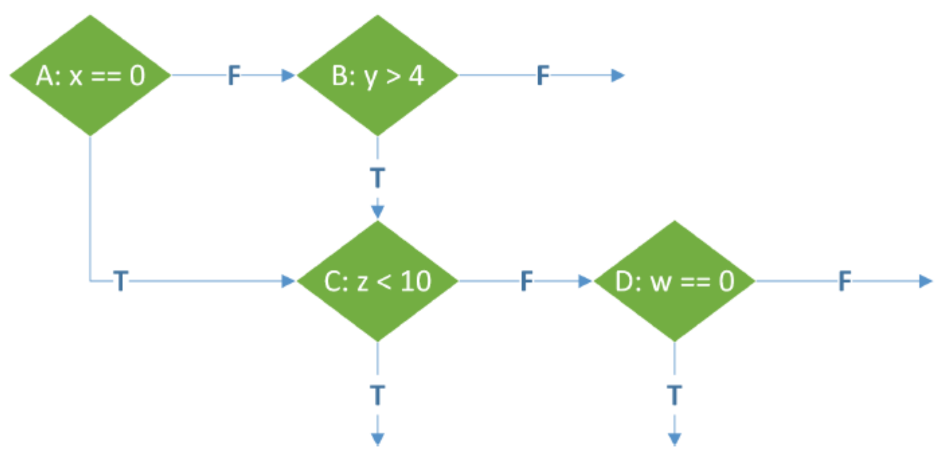
Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F
8	F	F	-	-	F
9	F	F	-	-	F

Example: Condition Coverage

Test	`A: (x == 0)`	`B: (y > 4)`	`C: (z < 10)`	`D: (w == 0)`	`(A || B) && (C || D)`
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F
8	F	F	-	-	F
9	F	F	-	-	F

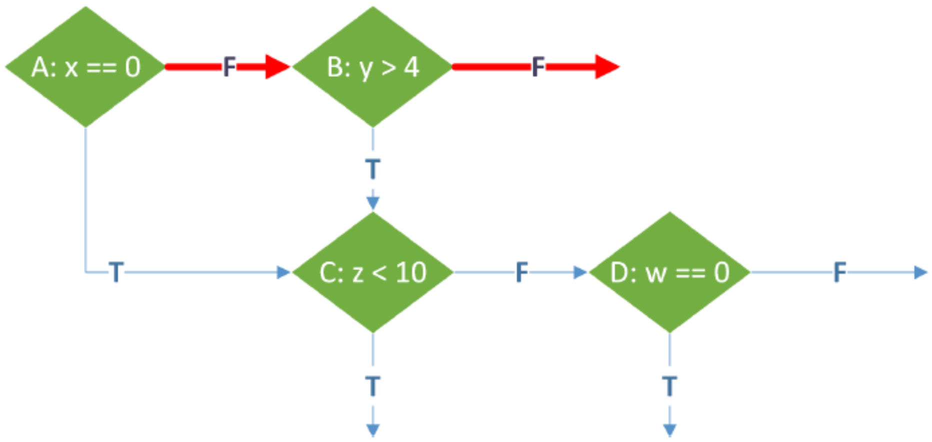
Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F



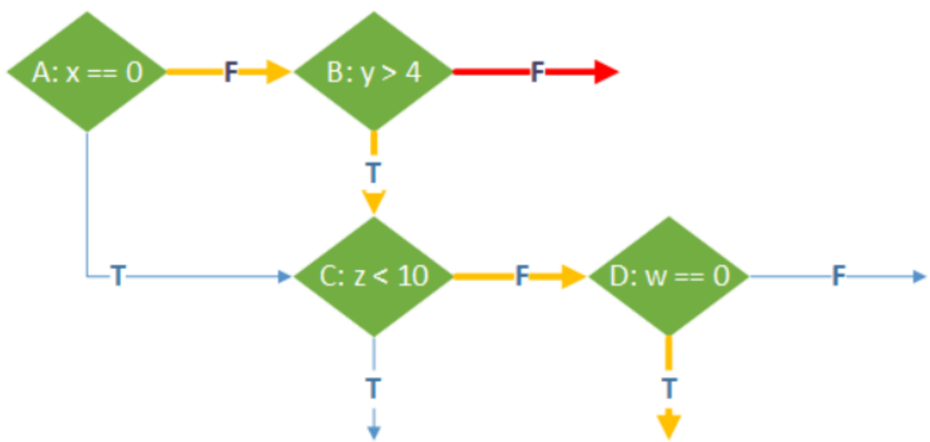
Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F



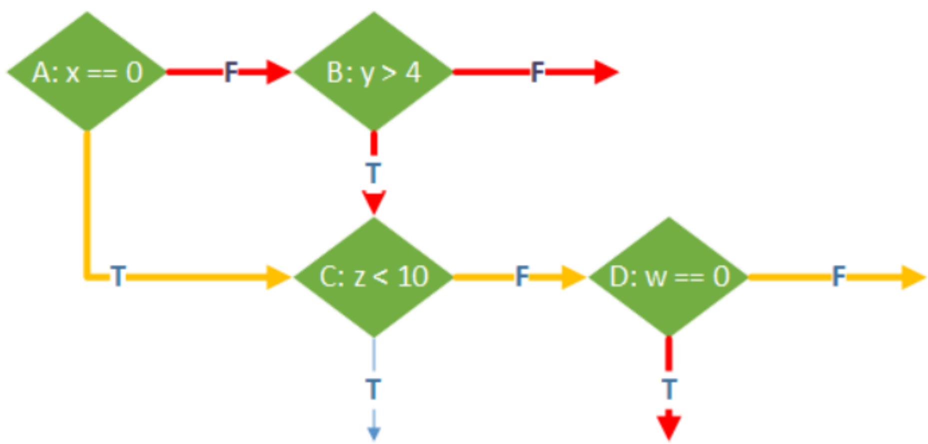
Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F



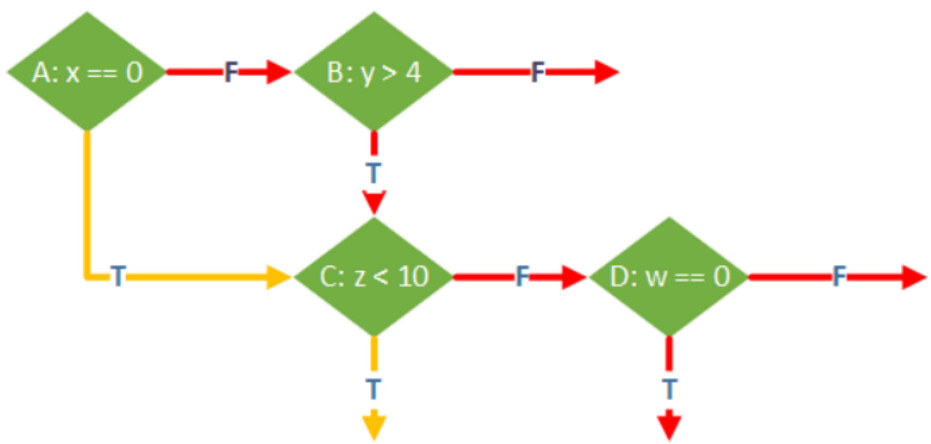
Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F



Example: Condition Coverage

Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F



Example: Condition Coverage

Test	`A: (x == 0)`	`B: (y > 4)`	`C: (z < 10)`	`D: (w == 0)`	`(A || B) && (C || D)`
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F

Tests 1, 3, 5, 7 are sufficient for 100% condition coverage.

Hence, we might select the following test cases:

- **Test 1**: `x = 0, y = 0, z =  0, w = 0`
- **Test 3**: `x = 0, y = 0, z = 10, w = 1`
- **Test 5**: `x = 1, y = 5, z = 10, w = 0`
- **Test 7**: `x = 1, y = 0, z =  0, w = 0`

Multiple Condition Coverage

- Condition coverage(C_2) says
  - Every atomic condition must evaluate once to true and once to false
- Multiple condition coverage (C_3) says
  - Every possible combination of atomic and composed predicates must evaluate once to true and once to false

Ex: Condition Coverage

<table>
<tr><th>Test</th><th>A: (x == 0)</th><th>B: (y > 4)</th><th>C: (z < 10)</th><th>D: (w == 0)</th><th>(A || B) && (C || D)</th></tr>
<tr><td>1</td><td>T</td><td>-</td><td>T</td><td>-</td><td>T</td></tr>
<tr><td>2</td><td>T</td><td>-</td><td>F</td><td>T</td><td>T</td></tr>
<tr><td>3</td><td>T</td><td>-</td><td>F</td><td>F</td><td>F</td></tr>
<tr><td>4</td><td>F</td><td>T</td><td>T</td><td>-</td><td>T</td></tr>
<tr><td>5</td><td>F</td><td>T</td><td>F</td><td>T</td><td>T</td></tr>
<tr><td>6</td><td>F</td><td>T</td><td>F</td><td>F</td><td>F</td></tr>
<tr><td>7</td><td>F</td><td>F</td><td>-</td><td>-</td><td>F</td></tr>
</table>

Ex: Multiple Condition Coverage

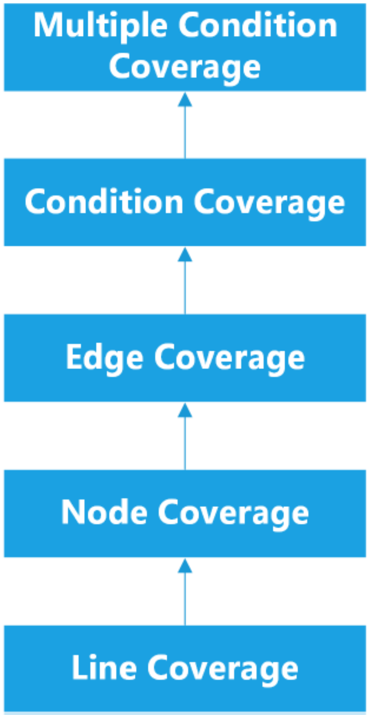
Test	A: (x == 0)	B: (y > 4)	C: (z < 10)	D: (w == 0)	(A || B) && (C || D)
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F

Multiple Condition Coverage

If we have achieved multiple condition coverage

- We must have evaluated every possible combination of each condition at least once to true and once to false
- Therefore, we must have evaluated each condition both ways

Multiple condition coverage is stronger than condition coverage



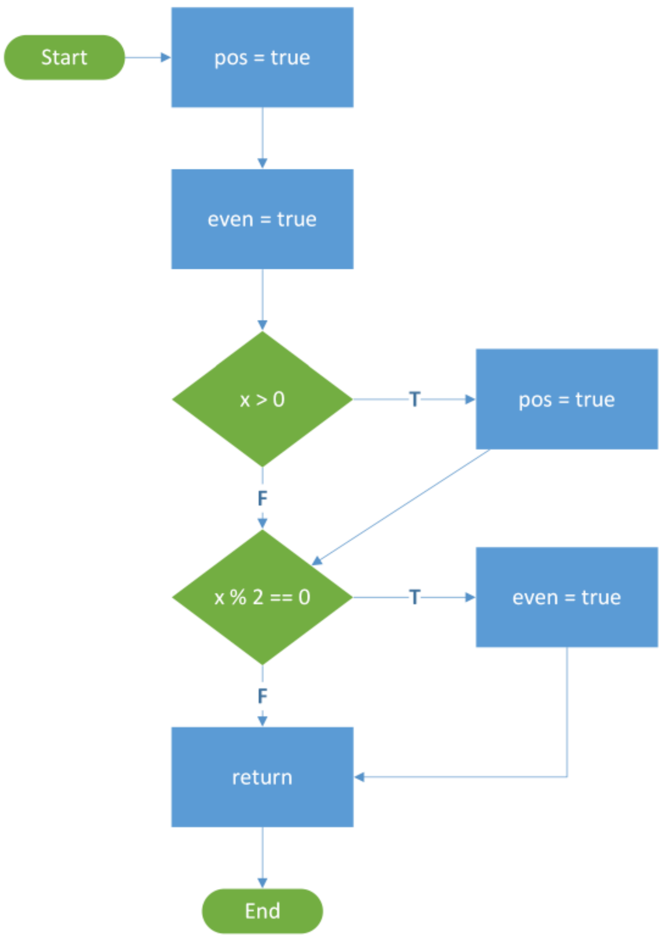
Path Coverage

- Path: sequence of nodes visited during an execution
- Path coverage is the strongest possible coverage measure
  - 100% path coverage means that every possible path through the program flowchart has been followed
  - For (non recursive) code without loops, this is achievable
  - For code with non-deterministic loops:
    - Each iteration of the loop adds an additional path
    - For some code, we can iterate any number of times
      - Eg. The number of iterations might depend on the size of an input
- For some code, 100% path coverage is not possible

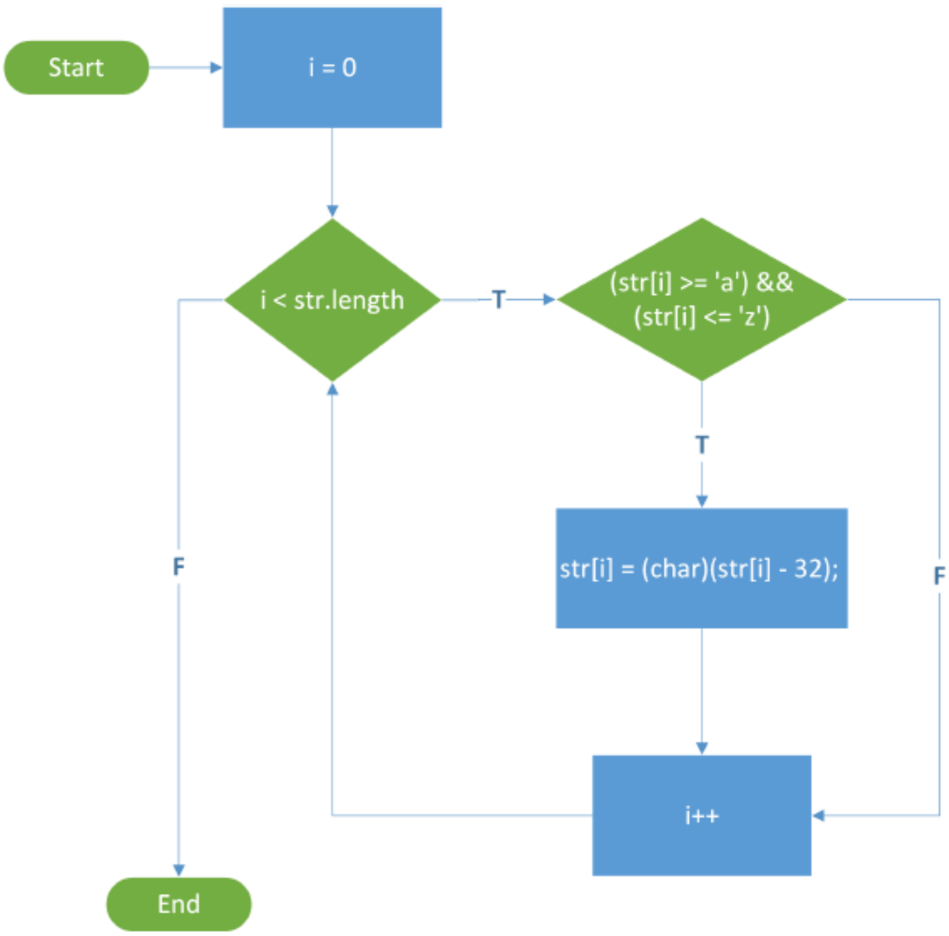
Path Coverage - Example 1

- Recall the integer classification algorithm

- Four paths through the code are possible, covered by Test Suite:
  1) x = 2
  2) x = 1
  3) x = -2
  4) x = -1



Path Coverage - Example 2



Recall case converter

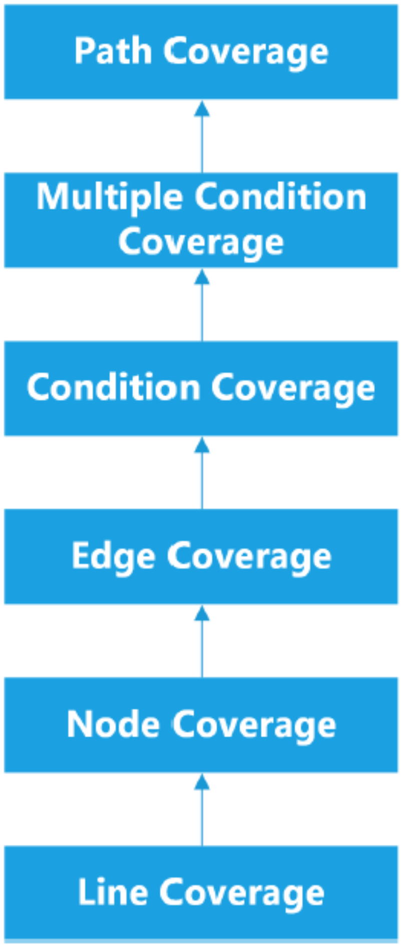
- Every char array evaluated may have different length
  - Will follow different paths

Path Coverage

If we have achieved 100% path coverage

- We must have evaluated every possible path through the program
- Therefore, we must have evaluated every possible combination of each condition at least once to true and once to false

Path coverage is stronger than multiple condition coverage



Testing Loops

- Path coverage for non-deterministic loops is impossible
  - Most programs have non-deterministic loops with an arbitrary number of iterations
    - e.g. a top-level loop in which we read a command and execute it

- For entire programs, we don't generally attempt 100% path coverage
- However, path coverage is still useful for small sections of code

Testing Loops

- We can't test all paths in programs with nondeterministic loops
  - But, we want to do better than multiple condition coverage
    - or might miss certain kinds of errors

Testing Loops

- Programmer may not consider what would happen if:
  - The loop decision is false right from the start
  - The loop decision is true once, and then false

- Sometimes, there is a maximum number of possible iterations for a loop (e.g. the loop might stop at the end of an array).

- Programmer may not consider what would happen if
  - Loop decision is true max times
  - Loop decision is true max-1 times

Testing Loops

It is therefore useful to write test cases which execute the loop:

- 0 times
- 1 time
- More than once
- max times (if applicable)
- max-1 times (if applicable)