Western

UNIVERSITY · CANADA

Topic 11

Structural Testing

Computer Science 2212b
Introduction to Software Engineering
Winter 2014

Jeff Shantz
jeff@csd.uwo

Agenda

- **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 (C_0)

Consider the following code:

```
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:
  - `"Positive: true"` if the number is positive; `false` otherwise
  - `"Even: true"` if the number is even; `false` otherwise

Statement Coverage (C_0)

We'll choose `x = 2` as a test case

```
@Test
public void testClassify() {

  String result = Ex1.classify(2);
  assertThat(result, containsString("Positive: true"));
  assertThat(result, containsString("Even: true"));

}
```

Our test passes!

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running Ex1Test
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.132 sec
```

Statement Coverage (C_0)

JaCoCo reports 100% statement coverage from the test case

Element	Missed Instructions	Cov.	Missed	Lines
classify(int)		100%	0	7

But the `classify` method is still wrong. Why?

```java
  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);
  }

}
```

Statement Coverage (C_0)

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

- **Statement Coverage:**
  - *Has each statement in the method been executed?*
- **Strategy**: find paths that cover all statements; write test cases to exercise those paths
- **Least restrictive of the coverage criteria**
  - Some branches may be missed
- **Can help to measure correctness of code written**
  - *Better than nothing*

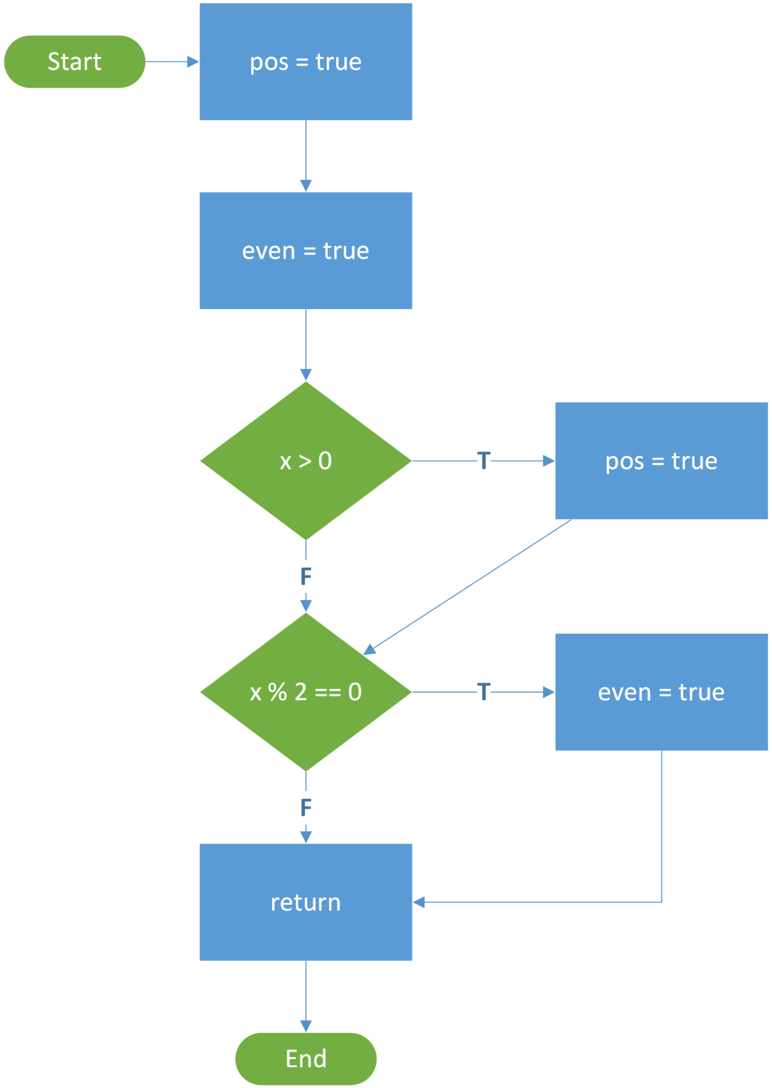
Program Flowcharts

```
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);
}
```

Program flowchart: a labelled, directed graph in which

- statements are represented by *rectangles*
- decisions are represented by *diamonds*

Program Flowcharts



Nodes: statements

Edges: indicate parts of paths which may be followed through the code

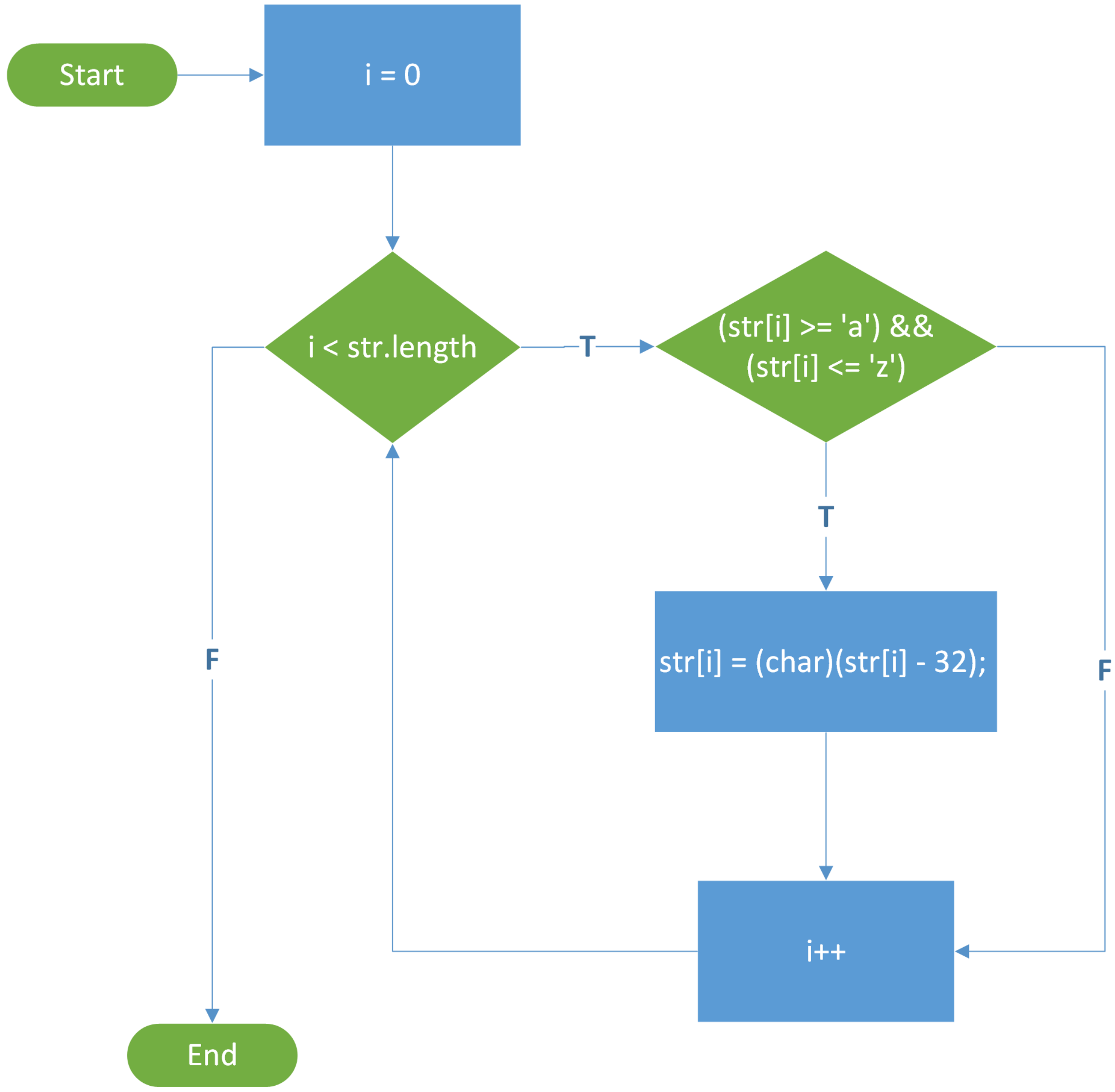
Labels on edges: indicate what happens when a condition is true/false

Program Flowcharts: Example 2

Consider the following code:

```
public static void toUppercase(char[] str) {

  int i = 0;

  while (i < str.length) {

    if ((str[i] >= 'a') && (str[i] <= 'z'))
      str[i] = (char)(str[i] - 32);

    i++;
  }

}
```

Program Flowcharts: Example 2



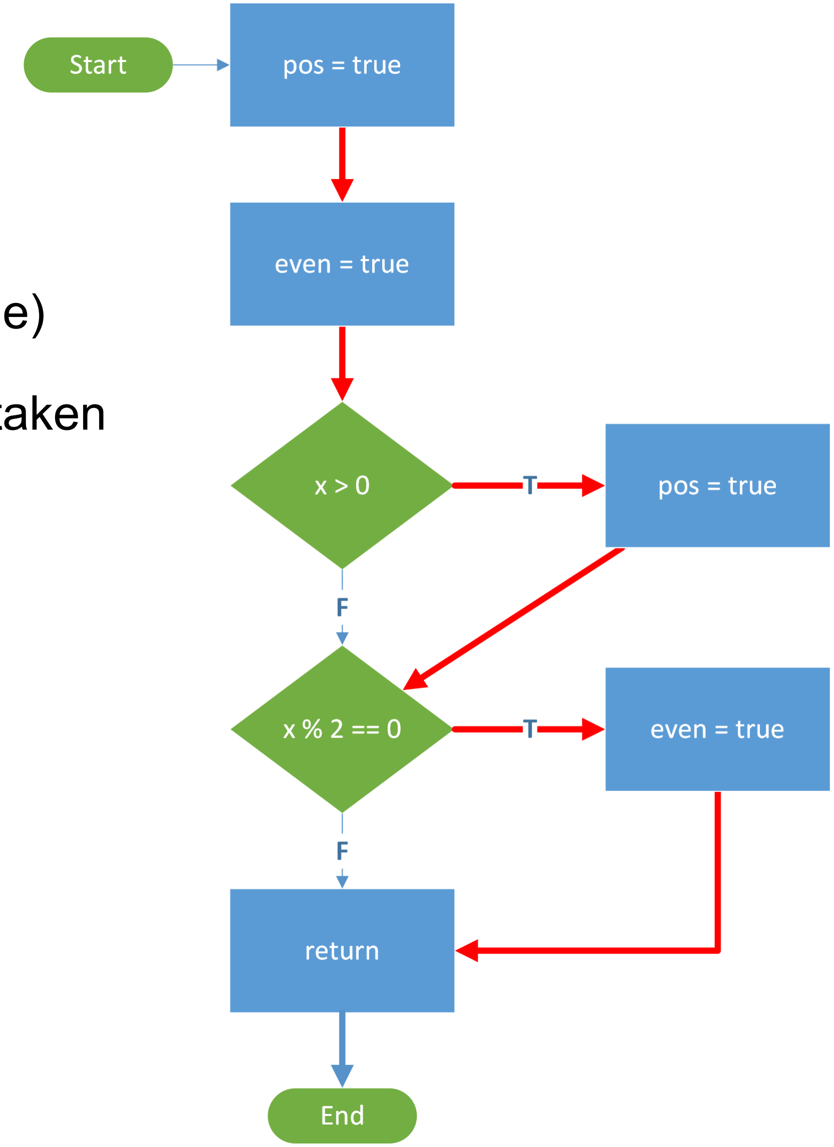
Structural Testing and Flowcharts

- **If our test suite executes all statements, then it visits every node in the flowchart**
  - We say that the test suite *covers every node*
  - Since nodes represent statements, we can say that it *covers every statement*
  - **This does not mean that every edge in the flowchart has been followed**

Example: Statement Coverage

Consider the first example flowchart

- Test case: `x = 2`
- All nodes visited (100% statement coverage)
- However, neither **F** branch has ever been taken



Example: Statement Coverage

JaCoCo reports that our test case has executed only 50% of the branches

Element	Missed Instructions	Cov.	Missed Branches	Cov.
classify(int)		100%		50%

```java
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);
}
```

Edge Coverage (C_1)

- **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)*

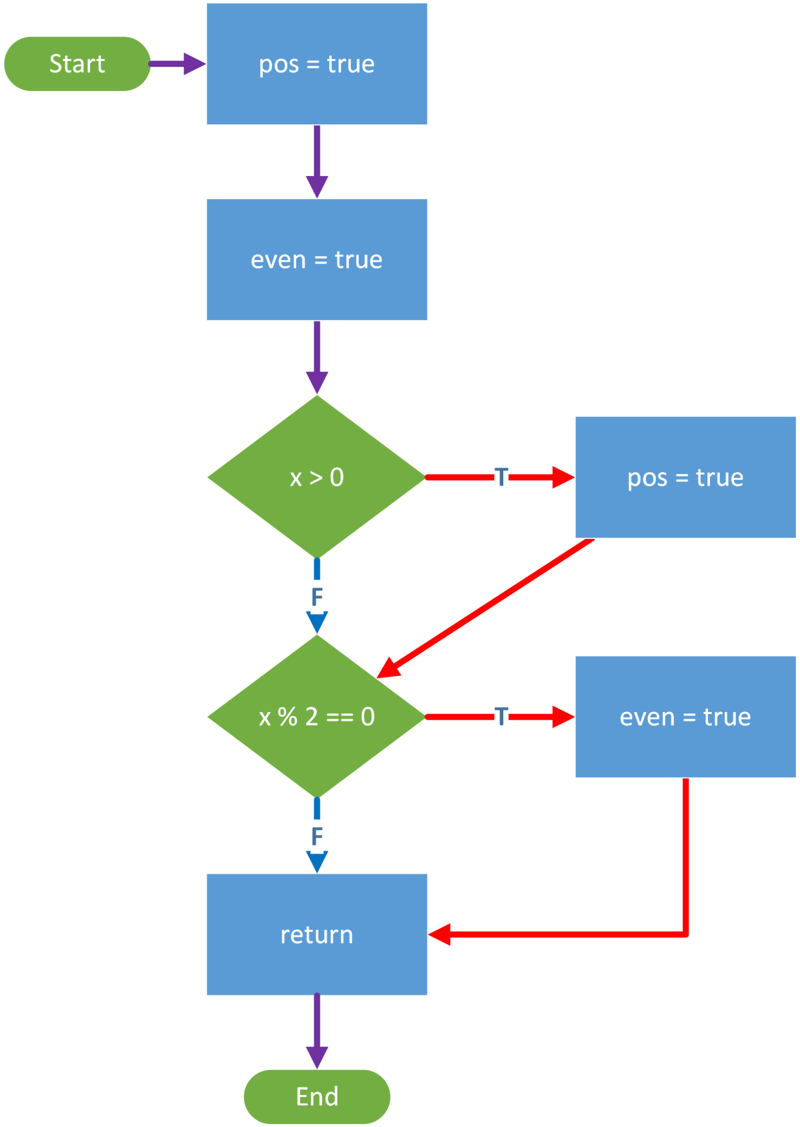
- **Branch coverage:**
  - *Has every branch of each control structure (if, switch) been executed?*
  - Equivalently, *has every edge in the program flowchart been executed?*
    - Caution: JaCoCo's branch coverage only considers edges from decision nodes – not every edge in the flowchart

Edge Coverage (C_1)

- **More thorough than statement coverage (C_0)**
  - 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 (C_1)



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.

Edge Coverage (C_1)

```
@Test
public void testClassifyPositiveEven() {

    String result = Ex3.classify(2);
    assertThat(result, containsString("Positive: true"));
    assertThat(result, containsString("Even: true"));

}

@Test
public void testClassifyNegativeOdd() {

    String result = Ex3.classify(-1);
    assertThat(result, containsString("Positive: false"));
    assertThat(result, containsString("Even: false"));

}
```

Edge Coverage (C_1)

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.222 sec <<<
FAILURE!
testClassifyNegativeOdd(Ex3Test)  Time elapsed: 0.011 sec  <<< FAILURE!
java.lang.AssertionError:
Expected: a string containing "Positive: false"
     but: was "Number: -1, Positive: true, Even: true"
```

In striving for 100% edge coverage, we caught a bug that our previous test – despite giving us 100% node coverage – could not.

Edge coverage can reveal failures where node coverage cannot.

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*
- **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*

Example: Line Coverage

```
public class ComputerScienceStudent {

  private boolean zombie;

  public String getMood(int hoursOfSleep) {

    if (hoursOfSleep < 2) { this.zombie = true; }

    return (hoursOfSleep >= 2) ? "Ready to code" :
                                 "More Red Bull please";

  }

}
```

Strength of Coverage Measures

100% node coverage implies 100% line coverage

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

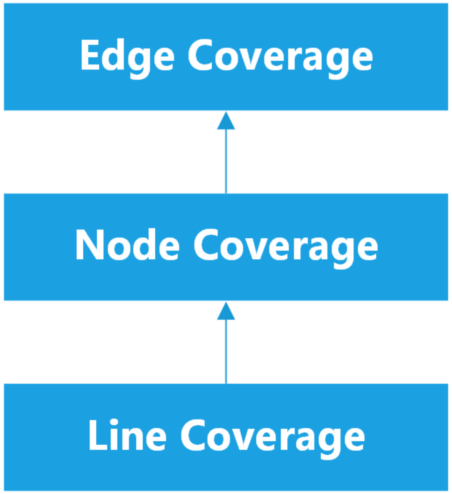
100% edge coverage implies 100% node coverage

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

Strength of Coverage Measures

Thus, we can say

- Node (statement) coverage is stronger than line coverage
- Edge (decision) coverage is stronger than node coverage



Question

Draw a flowchart for the following C code, and find the minimum number of test cases to achieve:

- 100% line coverage
- 100% node coverage
- 100% edge coverage

```c
int main()
{
  int a, b;
  a = 6;
  printf("\n\nEnter a value for a: ");
  scanf ("%d",&a);
  printf("\n\nEnter a value for b: ");
  scanf ("%d", &b);
  if ((a < 10) || (a > 15))
     if (b ==9) printf ("good\n");
     else
        printf("evening\n");
  else
     printf("madam");
  printf("sir\n");
  return 0;
}
```

Your Answer

Minimal Test Suites

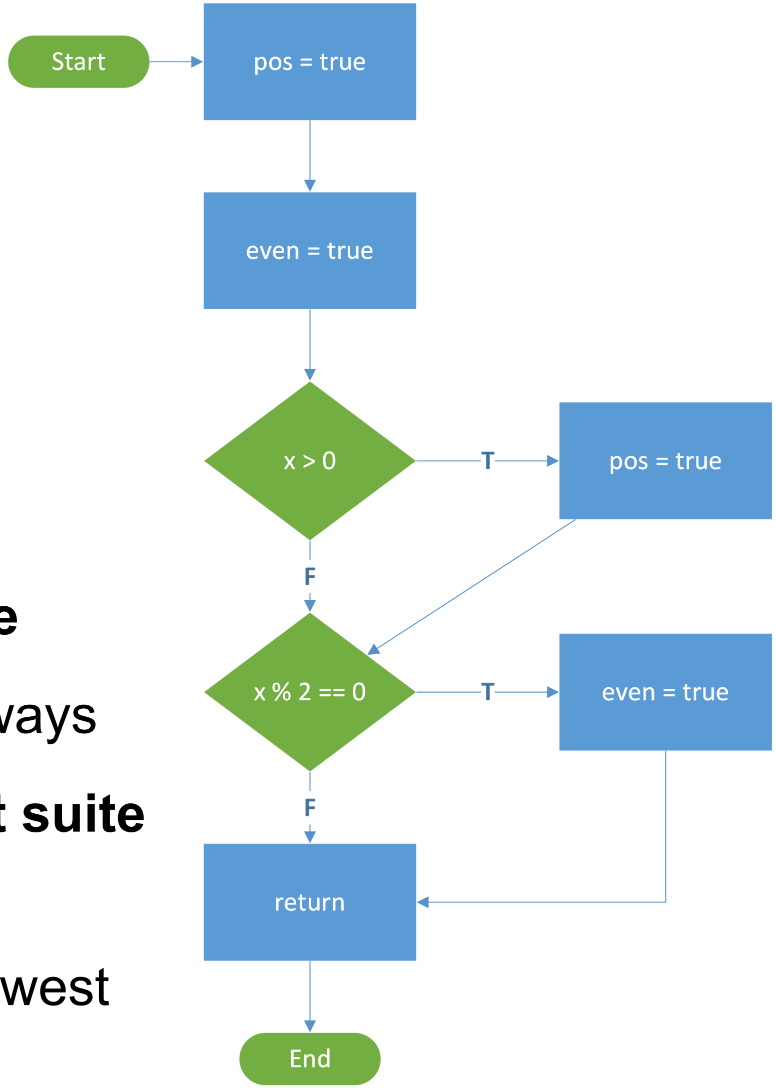
By adding more test cases to a test suite, we may achieve higher coverage

- Larger test suites take longer to run
- Often want to find some *minimal* test suite
- Import 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
- **Cannot have a minimal test suite which covers all edges with only one test case**
  - Have to execute both decisions both ways
- **Hence, `x = 2`, `x = -1` is a minimal test suite for 100% edge coverage**
  - Minimal in the sense of needing the fewest test cases



Minimal Test Suites: Another Example

Can achieve 100% node coverage with a one-character test case

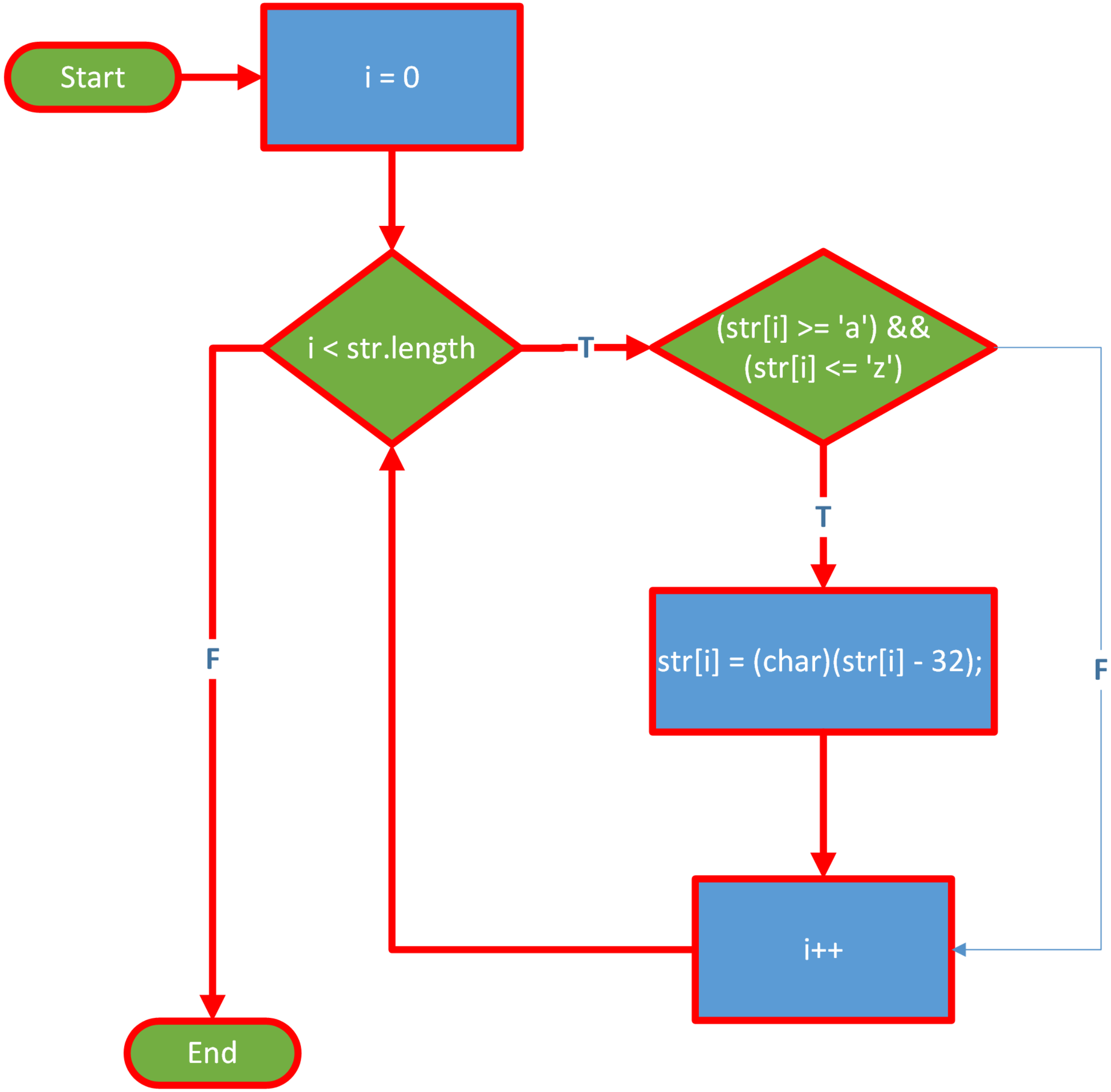
- e.g. `"a"`

However, **F** branch of the inner condition is not taken.

To cover all edges, we need a test suite like

A. `"a"`, `"X"`; or

B. `"aX"`



Minimal Test Suites: Another Example

Which of (A) and (B) is *minimal?*

- Depends on how we define minimality
- (A) has 2 test cases, but each has just 1 character
- (B) has just 1 test case, but it has 2 characters

Minimal Test Suites: Another Example

We should always precisely define *minimality* for a test suite

- Here, it makes more sense to accept (B) as minimal
- Less test cases = faster execution of test suite

Hence, for this problem, we would say that:

- Test suite **X** will be considered smaller 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 having to accept `"aaaaaaXXXXX"` as minimal

Stronger Coverage Measures

Decision: everything in parentheses after the `if`, `while`, etc.

- 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

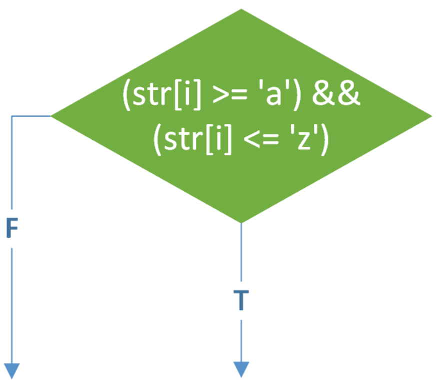
So far, we've written each decision in a single diamond.

If we divide the *conditions* within each decision into separate diamonds, we can get a better reflection of what the program does.

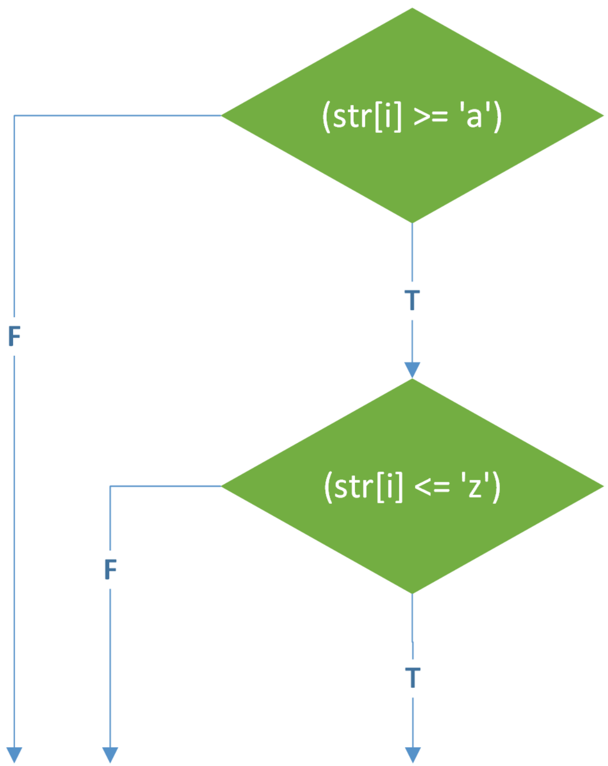
Flowcharts: Splitting Up Decisions

Example:

Instead of



We write



Recall that languages like Java use *short-circuit evaluation*

- The evaluation of a Boolean expression is cut short as soon as it can be determined to be true of false

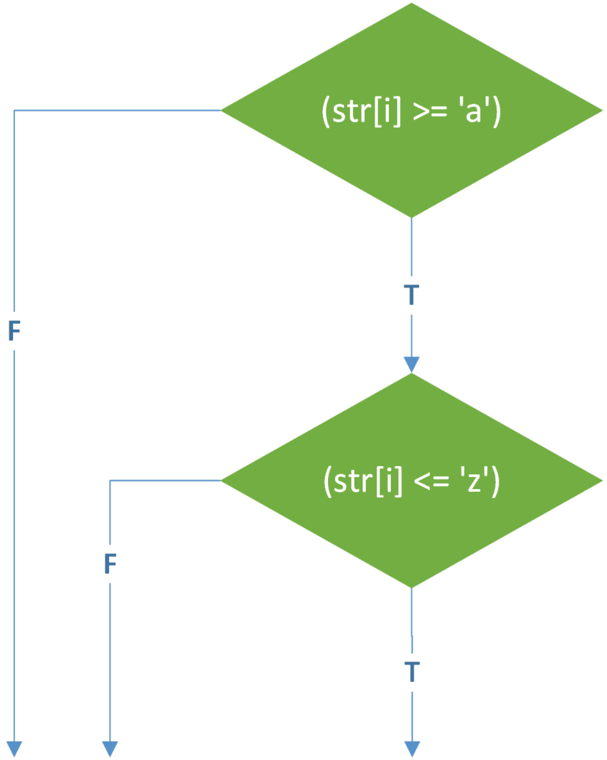
We now have two new edges that we'll need to cover

Condition Coverage (C_2)

If we split up all diamonds and *then* cover all edges

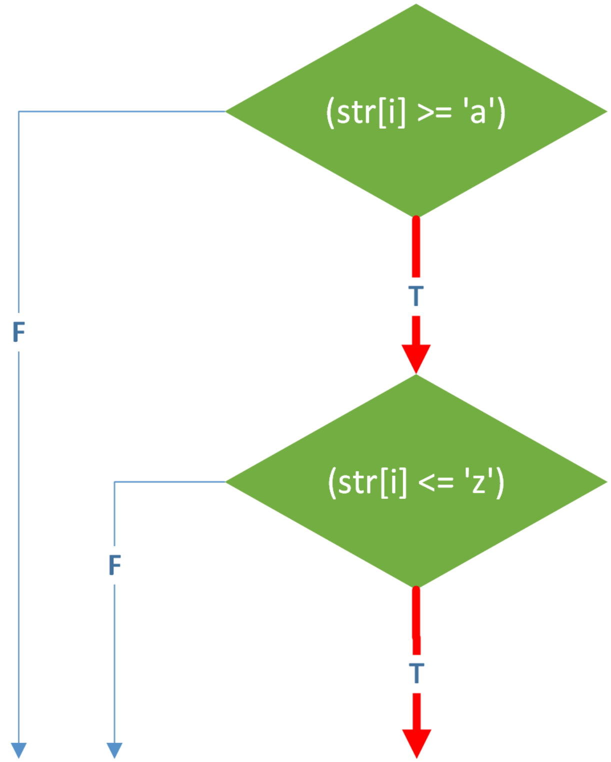
- We achieve more thorough coverage than C_1 coverage gives us
- **This is called *condition coverage***

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



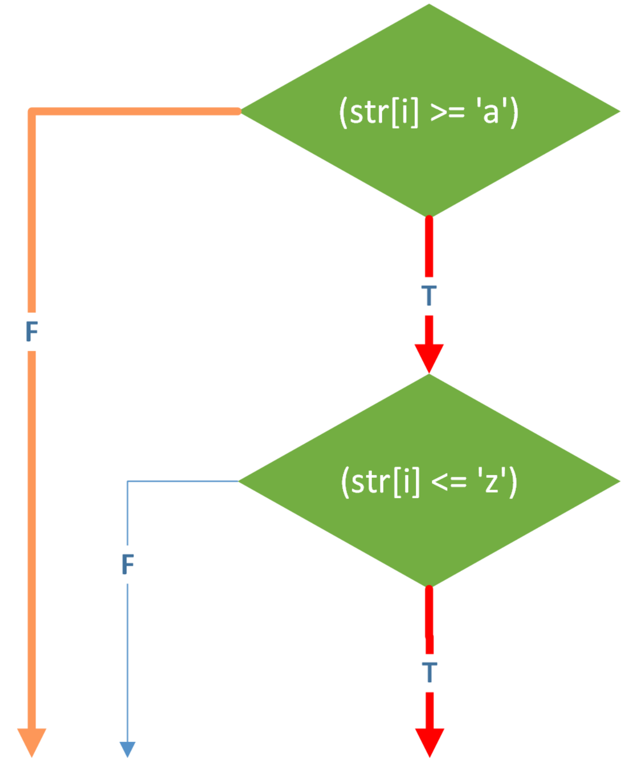
Condition Coverage (C_2)

Test Case: "a"



Condition Coverage (C_2)

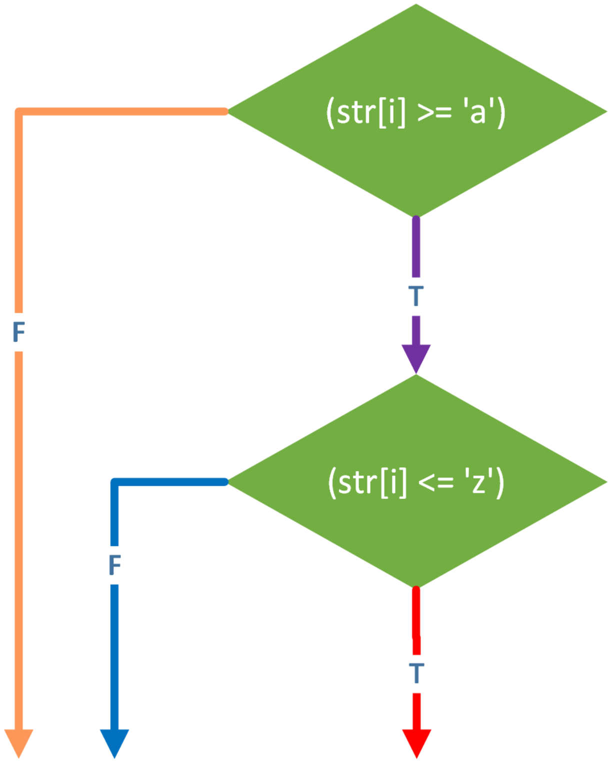
Test Case: "aX"



Condition Coverage (C_2)

(to evaluate the second F edge, we simply need a character that is `>=` `a` and also `>` `z`)

Test Case: `"aX~"`

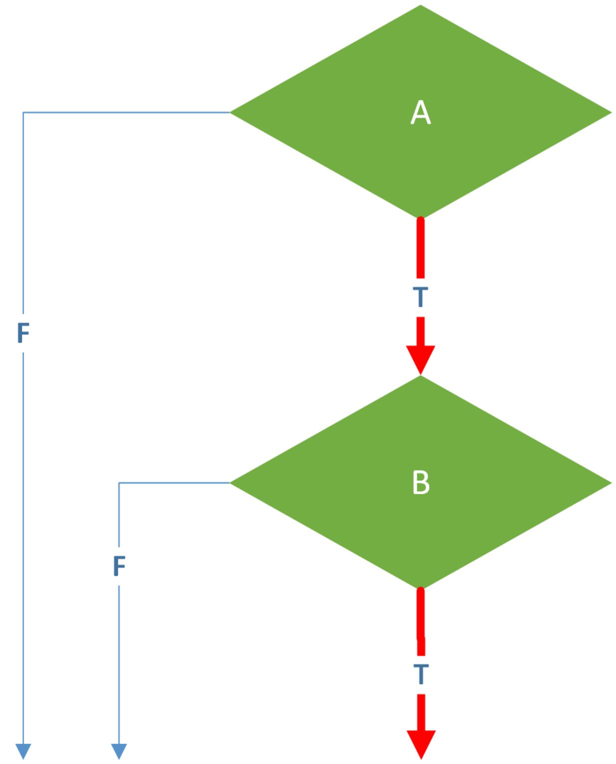


Hence, test case `"aX~"` is sufficient to achieve 100% condition coverage.

Condition Coverage (C_2): `A && B`

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

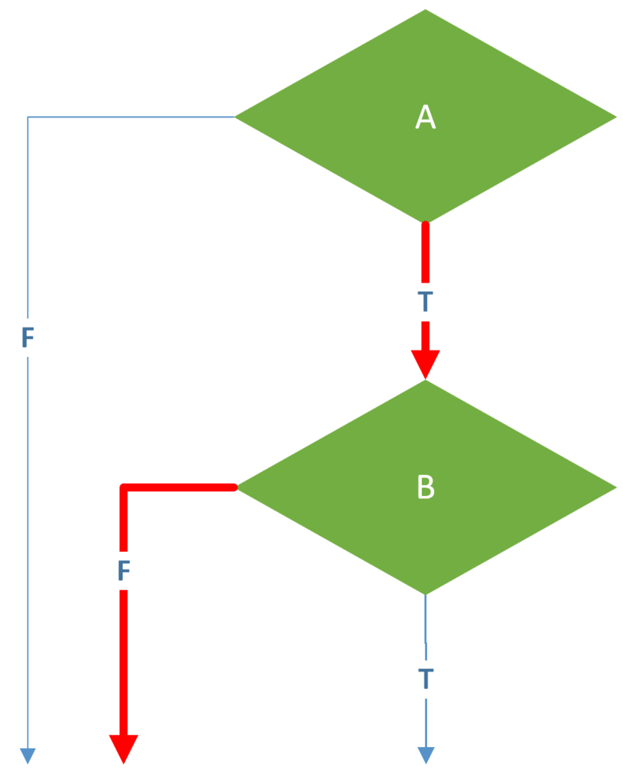
- **A** = `true`, **B** = `true`
- **A** = `true`, **B** = `false`
- **A** = `false`



Condition Coverage (C_2): `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`

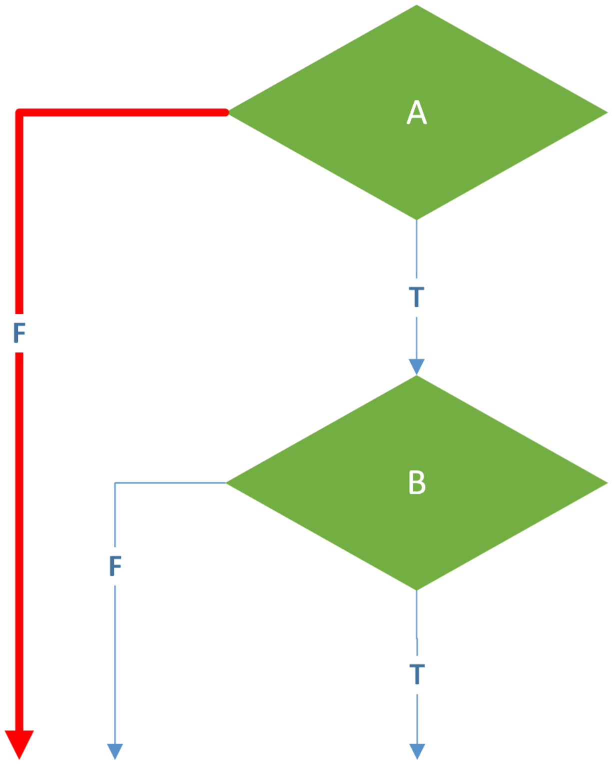


Condition Coverage (C_2): `A && B`

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`

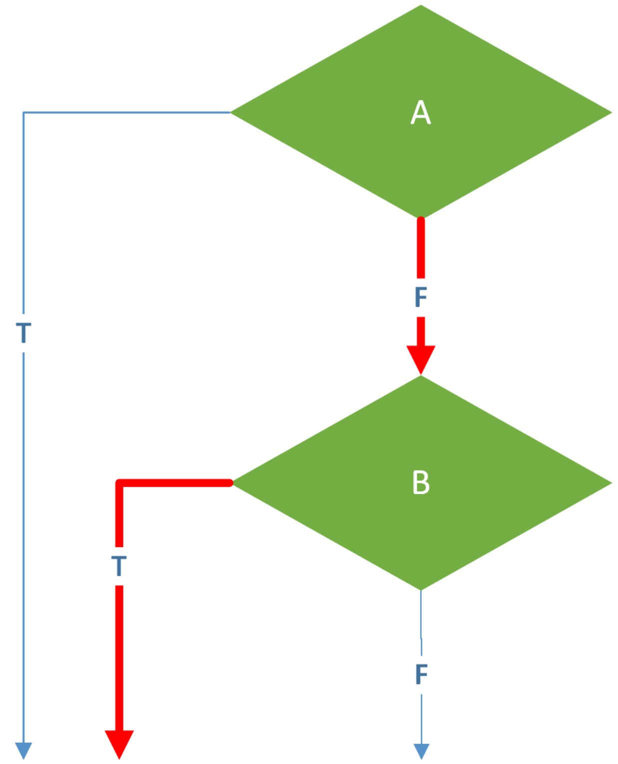
We will not even be able to get to the evaluation of **B** unless **A** is `true`, as in the first two cases, due to short-circuit evaluation.



Condition Coverage (C_2): `A || B`

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

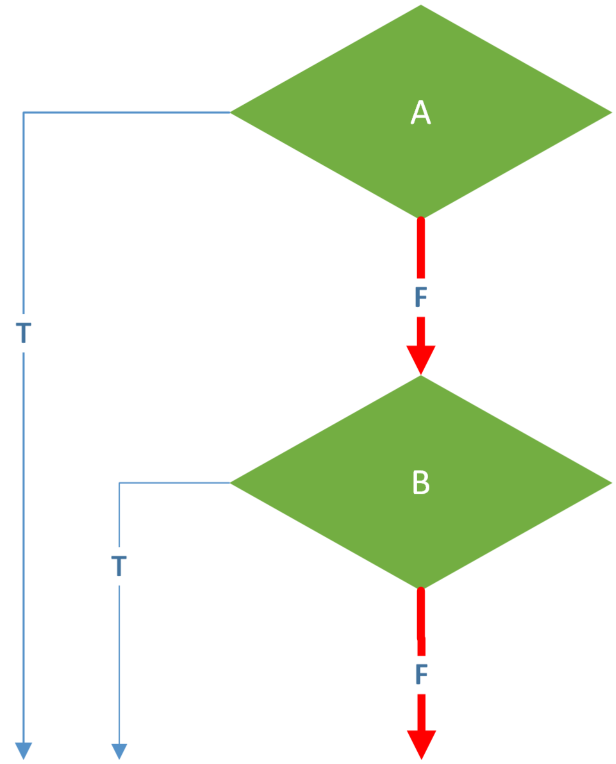
- **A** = `false`, **B** = `true`
- **A** = `false`, **B** = `false`
- **A** = `true`



Condition Coverage (C_2): `A || B`

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

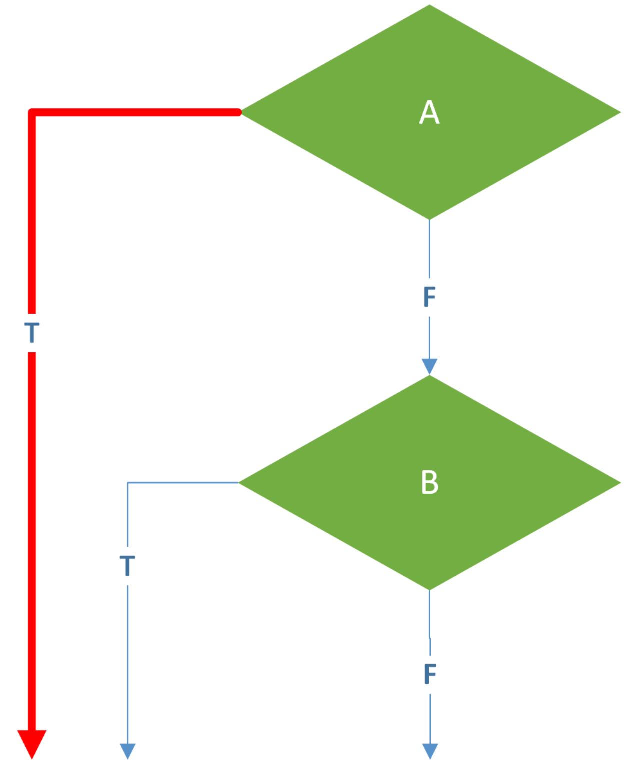
- **A** = `false`, **B** = `true`
- **A** = `false`, **B** = `false`
- **A** = `true`



Condition Coverage (C_2): `A || B`

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

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



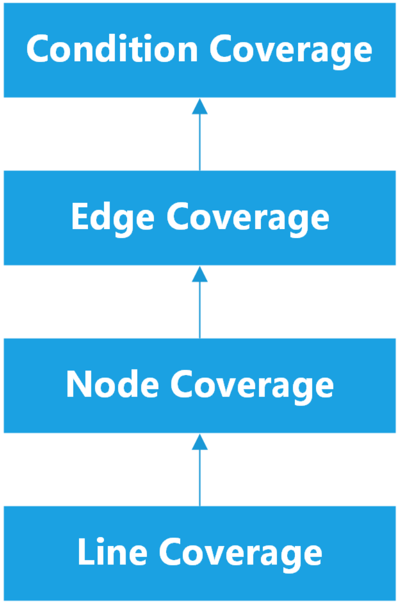
We will not even be able to get to the evaluation of **B** unless **A** is `false`, as in the first two cases, due to short-circuit evaluation.

Condition Coverage (C_2)

If we have achieved condition coverage

- We must have evaluated each condition on 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 (C_2)

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)`

Thus, the expression is equivalent to:

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

Example: Condition Coverage (C_2)

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 (C_2)

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 (C_2)

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 (C_2)

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 (C_2)

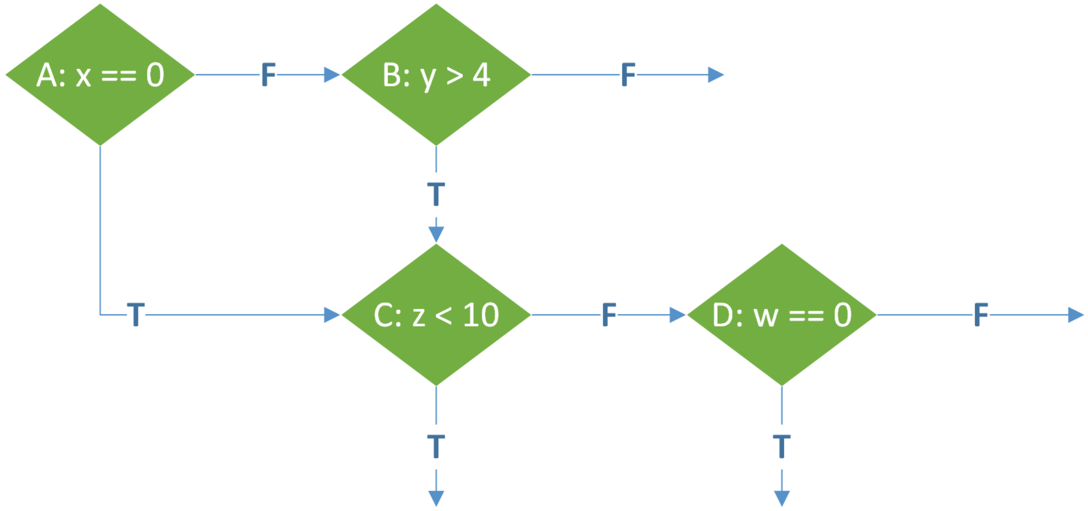
Test	`A: (x == 0)`	`B: (y > 4)`	`C: (z < 10)`	`D: (w == 0)`	`(A || B) && (C || D)`
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F
8	F	F	-	-	F
9	F	F	-	-	F

Example: Condition Coverage (C_2)

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

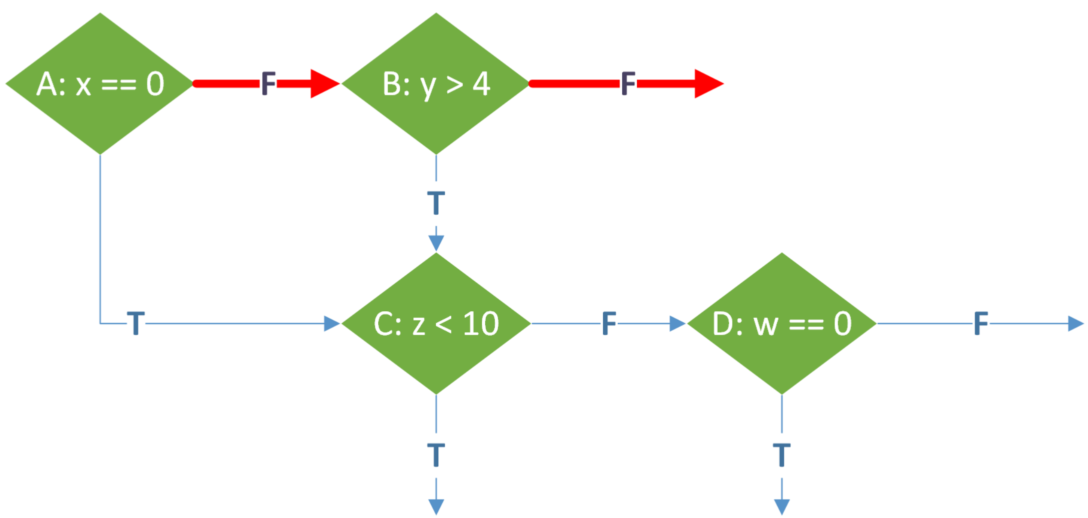
Example: Condition Coverage (C_2)

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



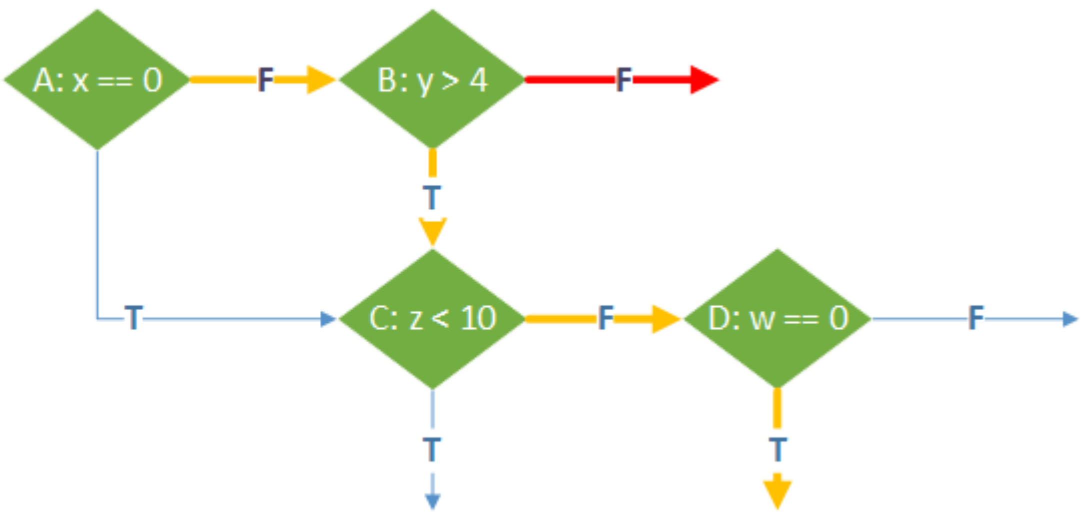
Example: Condition Coverage (C_2)

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



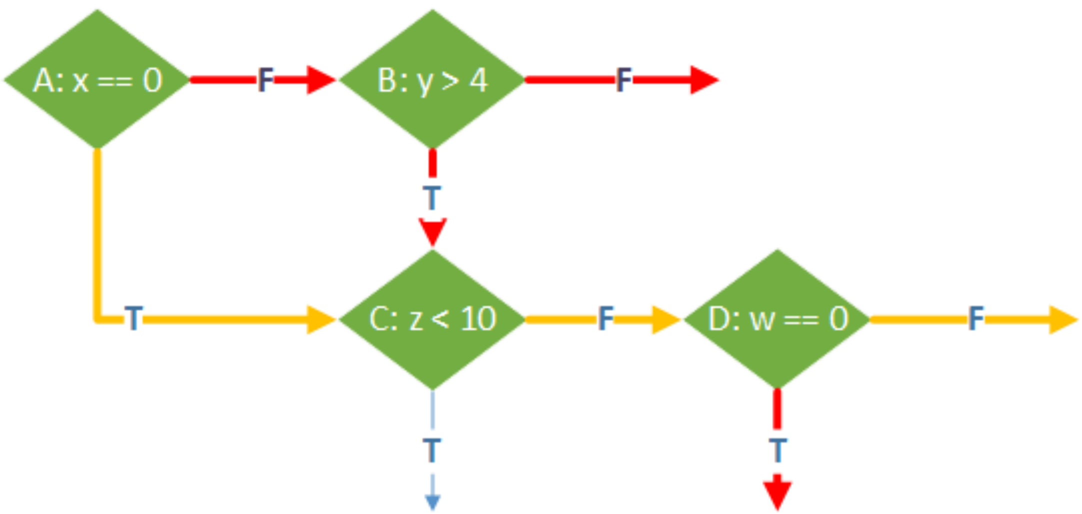
Example: Condition Coverage (C_2)

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



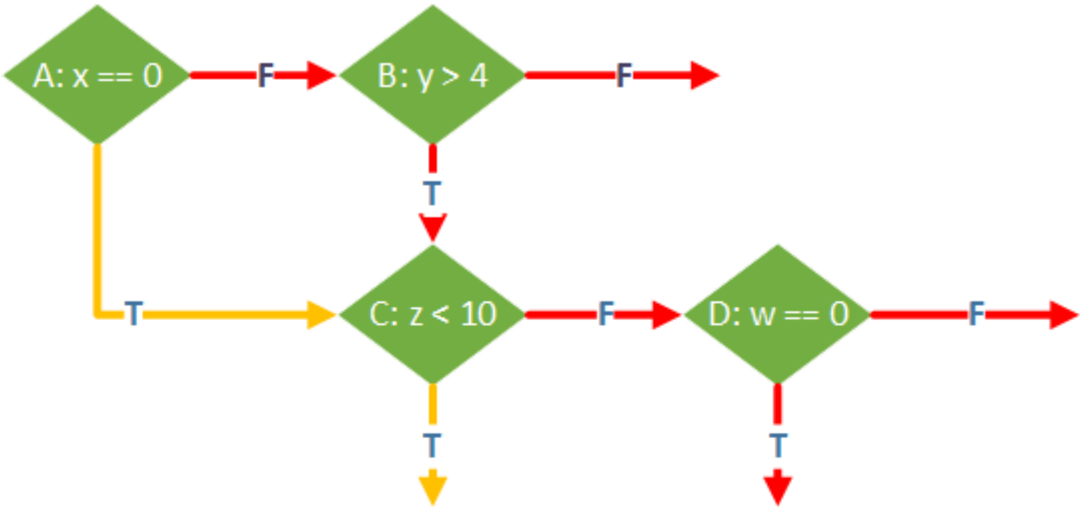
Example: Condition Coverage (C_2)

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 (C_2)

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 (C_2)

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 (C_3)

Condition coverage 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`*

Example: Condition Coverage (C_2)

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: Multiple Condition Coverage (C_3)

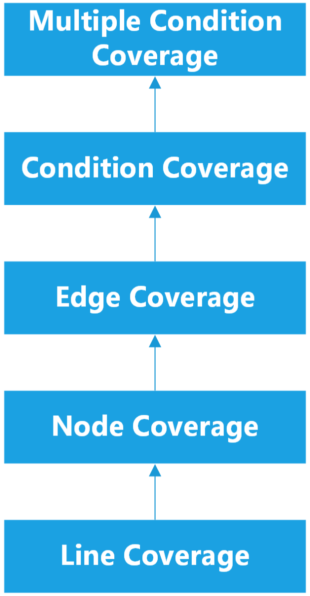
Test	`A: (x == 0)`	`B: (y > 4)`	`C: (z < 10)`	`D: (w == 0)`	`(A || B) && (C || D)`
1	T	-	T	-	T
2	T	-	F	T	T
3	T	-	F	F	F
4	F	T	T	-	T
5	F	T	F	T	T
6	F	T	F	F	F
7	F	F	-	-	F

Multiple Condition Coverage (C_3)

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

Thus, multiple condition coverage is stronger than condition coverage



Path Coverage (C_4)

Path coverage is the strongest possible coverage measure

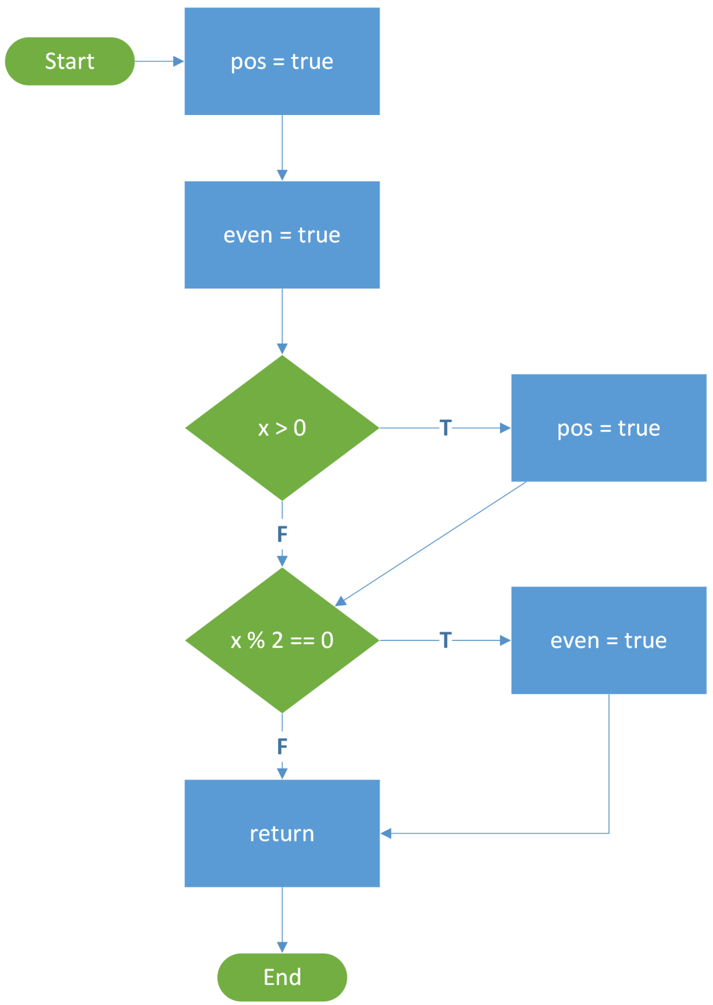
- 100% path coverage means that every possible path through the program flowchart has been followed
- **Path:** sequence of nodes visited during an execution
- **For code with no loops, this is achievable**
- **For code with non-deterministic loops:**
  - Each iteration of the loop adds an additional path
  - For some code, we can iterate any number of times
    - The number of iterations might depend on the size of an input
  - Hence, *for some code*, **it is not possible to achieve 100% path coverage**

Path Coverage (C_4): Example 1

Recall the integer classification algorithm from earlier.

Four paths through the code are possible:

- `x = 2`
- `x = 1`
- `x = -2`
- `x = -1`

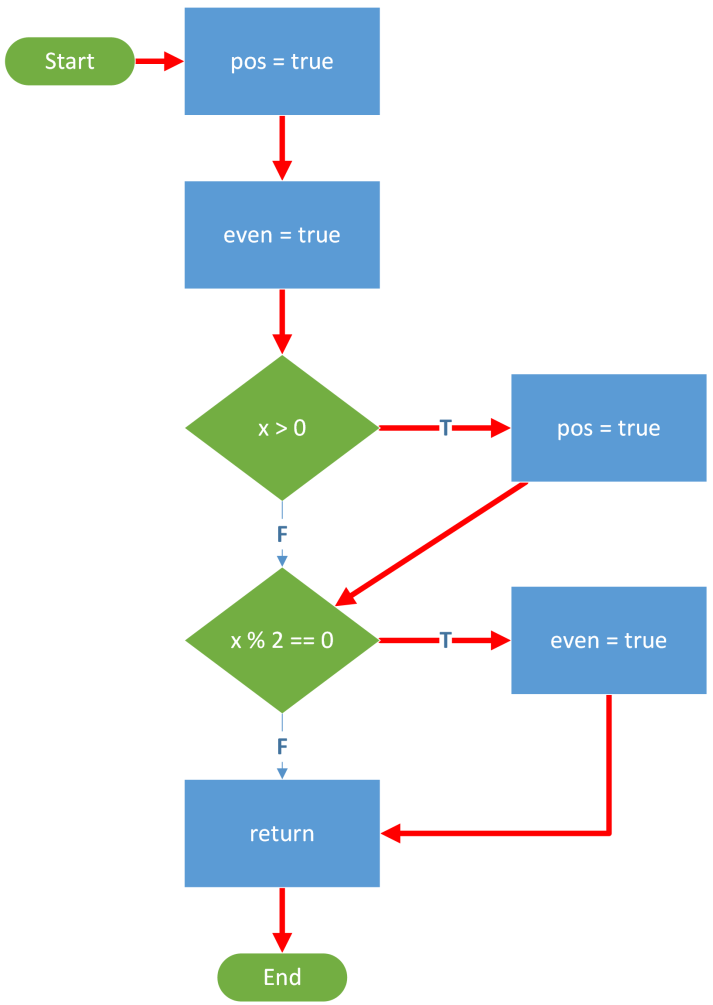


Path Coverage (C_4): Example 1

Recall the integer classification algorithm from earlier.

Four paths through the code are possible:

- `x = 2`
- `x = 1`
- `x = -2`
- `x = -1`

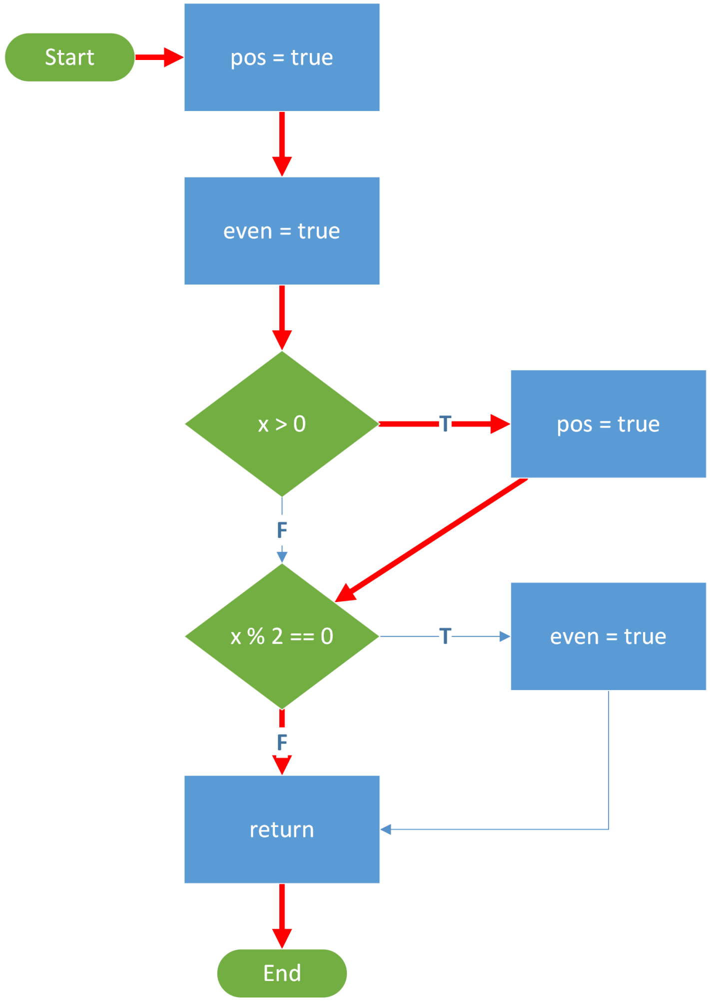


Path Coverage (C_4): Example 1

Recall the integer classification algorithm from earlier.

Four paths through the code are possible:

- `x = 2`
- `x = 1`
- `x = -2`
- `x = -1`

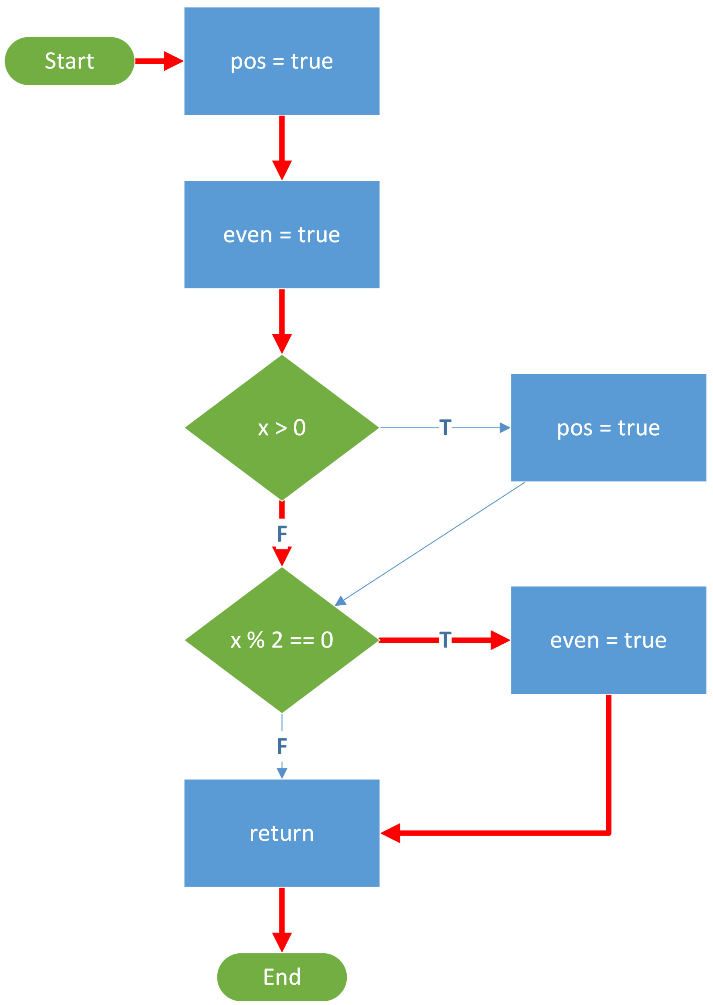


Path Coverage (C_4): Example 1

Recall the integer classification algorithm from earlier.

Four paths through the code are possible:

- `x = 2`
- `x = 1`
- `x = -2`
- `x = -1`

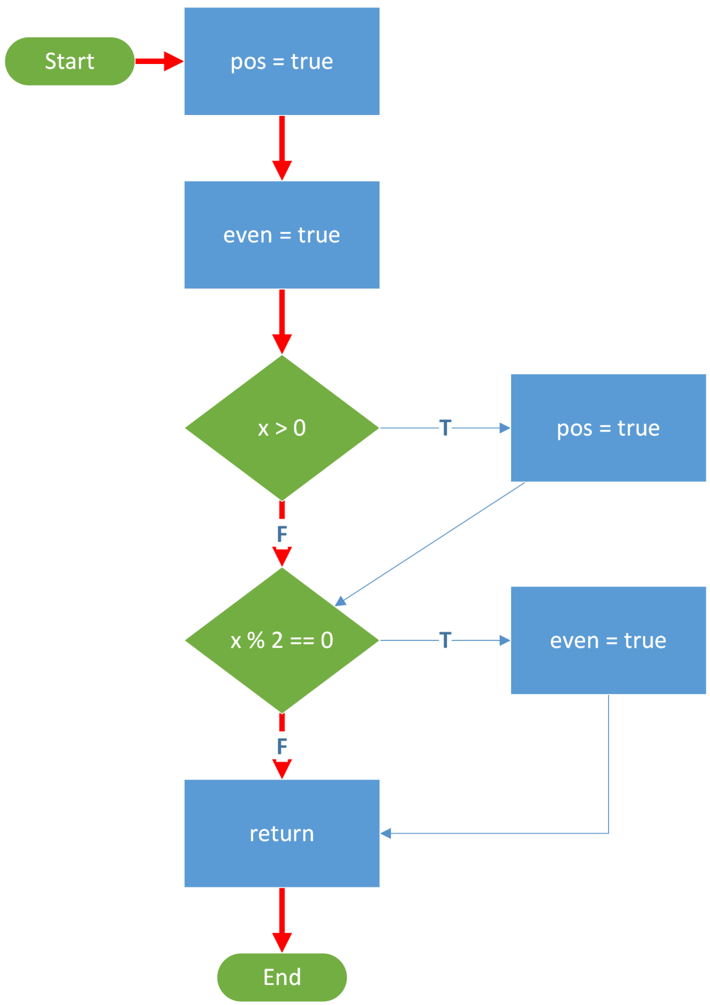


Path Coverage (C_4): Example 1

Recall the integer classification algorithm from earlier.

Four paths through the code are possible:

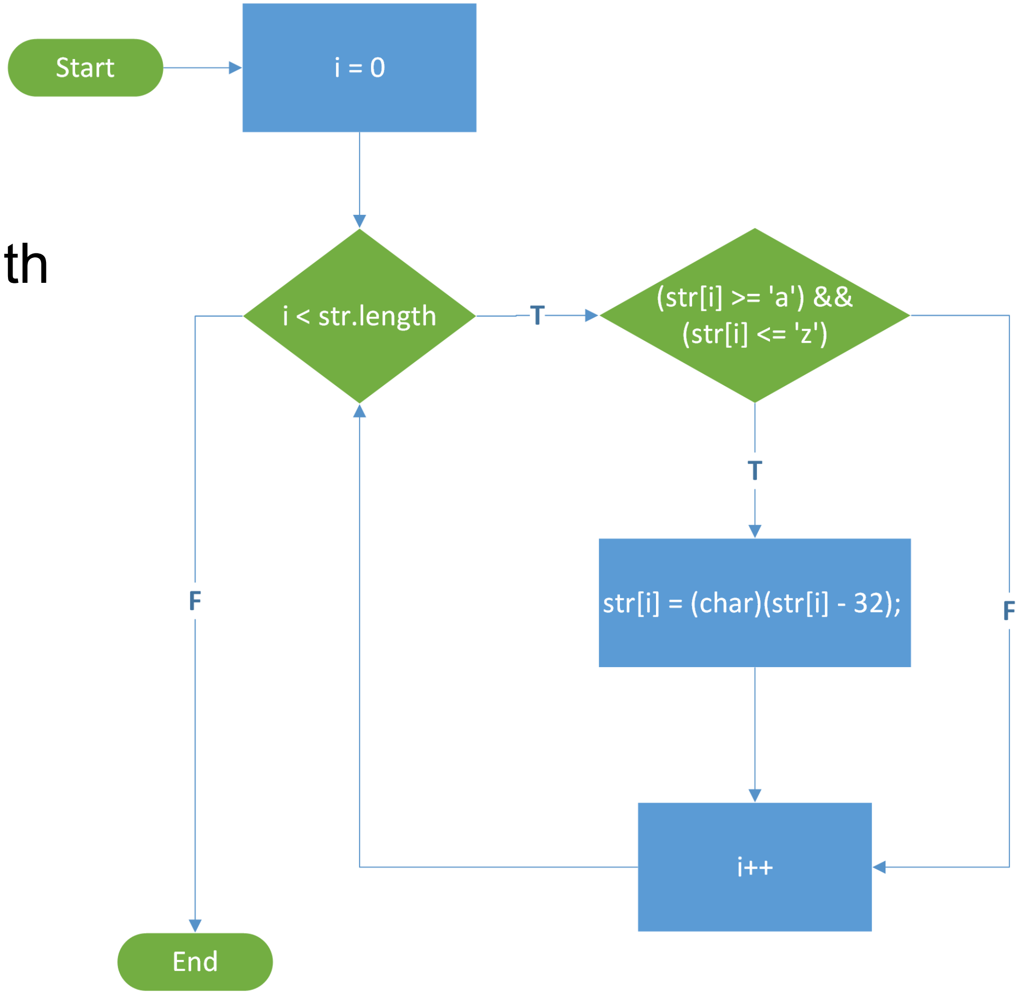
- `x = 2`
- `x = 1`
- `x = -2`
- `x = -1`



Path Coverage (C_4): Example 2

Recall the case conversion algorithm from earlier.

Every string of a different length will follow a different path.

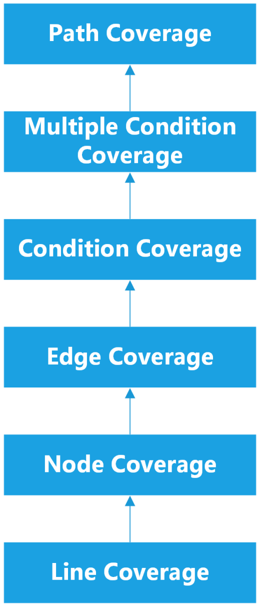


Path Coverage (C_4)

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

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

Testing Loops

Even if we can't test all possible paths in programs with non-deterministic loops, we can't rely solely on C_3 coverage

- Might not identify certain kinds of errors
- Often code is wrong because the programmer did not consider what would happen if
  - The loop decision is **false** right from the start
  - The loop decision is **true** once, and then **false**

Testing Loops

Even if we can't test all possible paths in programs with non-deterministic loops, we can't rely solely on C_3 coverage

- Sometimes, there is a maximum number of possible iterations for a loop (e.g. the loop might stop at the end of an array).
- Code may be wrong if programmer did 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)

Testing Loops: Exercise

Provide adequate loop coverage for the following code that finds the rightmost period in a string (used to find a file's extension).

```
i = strlen(fname) - 1;

while (i > 0 && fname[i] != '.')
{
   i--;
}
```

0 times	
1 time	
More than once	
`max-1` times	
`max` times	

Testing Loops: Exercise

The code in the loop in almost certainly correct. However, the problem **may not be with the loop itself**, but the **code that comes after**.

```
i = strlen(fname) - 1;

while (i > 0 && fname[i] != '.')
{
   i--;
}
```

For instance, it might

- Expect a non-empty extension
- Expect a non-empty filename before the extension
- Be unable to handle the case of no extension

Loop testing will help to identify these problems as well.

Western

UNIVERSITY · CANADA