CMPT 473
Software Testing, Reliability and Security

Graph Coverage

Nick Sumner

Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers/is adequate for our quality objectives.
- So far: Input & Requirements based

Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers/is adequate for our quality objectives.
- So far: Input & Requirements based

Efficiently sort a provided list in under X seconds

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers/is adequate for our quality objectives.
- So far: Input & Requirements based

Efficiently sort a provided list in under X seconds

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

How well can input based techniques test this?

Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers/is adequate for our quality objectives.
- So far: Input & Requirements based

Efficiently sort a provided list in under X seconds

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

How well can input based techniques test this?

How might we do better?

White Box / Black Blox

- Considering only the requirements or input is a ***black box*** approach
  - Treats the program like an opaque box
  - No deep knowledge of the program's structure

White Box / Black Blox

- Considering only the requirements or input is a ***black box*** approach
  - Treats the program like an opaque box
  - No deep knowledge of the program's structure
- Techniques that use artifacts of the program structure are ***white box*** approaches
  - They can 'see into' the program's implementation

White Box Testing

- What is a simple approach that solves our problem here?

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

White Box Testing

- What is a simple approach that solves our problem here?
- Statement Coverage
  - How many of the statements did the suite test? (%)

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

White Box Testing

- What is a simple approach that solves our problem here?
- Statement Coverage
  - How many of the statements did the suite test? (%)
- Branch Coverage
  - How many of the condition outcomes were tested? (%)

```
void sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

White Box Testing

- In this course, we'll mostly look at ***graph coverage*** based techniques
  - Most commonly used metrics in the real world

White Box Testing

- In this course, we'll mostly look at ***graph coverage*** based techniques
  - Most commonly used metrics in the real world
  - Most concepts can be modeled through graphs
    e.g. programs, protocols, use patterns, designs, …

White Box Testing

- In this course, we'll mostly look at ***graph coverage*** based techniques
  - Most commonly used metrics in the real world
  - Most concepts can be modeled through graphs
    e.g. programs, protocols, use patterns, designs, …

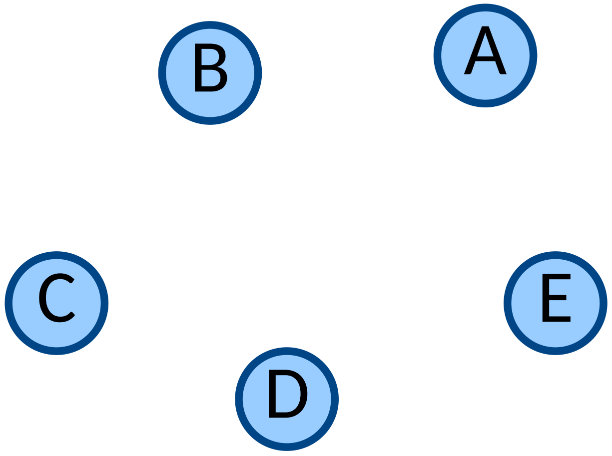
So a bit of review…

Graphs

- What is a *graph* G?

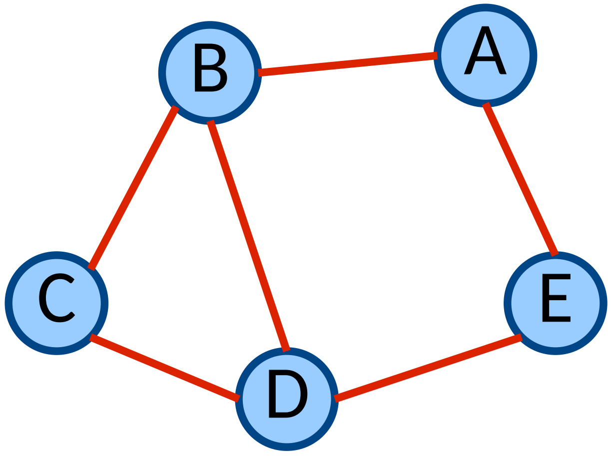
Graphs

- What is a *graph* G?
  - A set N of nodes



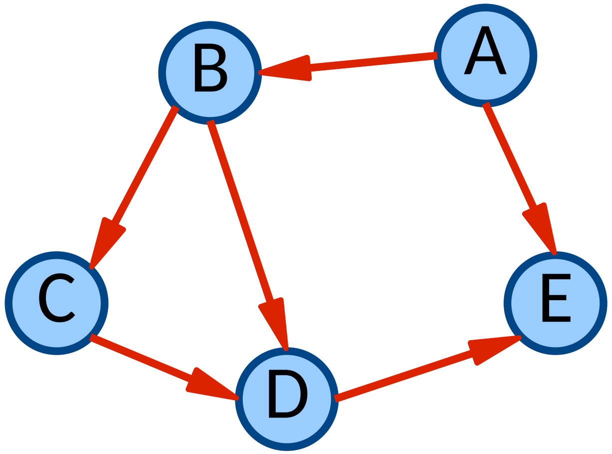
Graphs

- What is a *graph* G?
  - A set N of nodes
  - A set E of edges



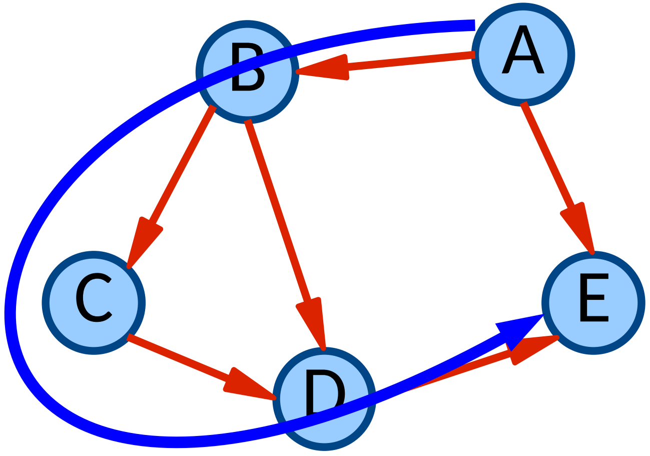
Graphs

- What is a *graph* G?
  - A set N of nodes
  - A set E of edges
- When edges are directed from one node to another, the graph is a *directed graph*



Graphs

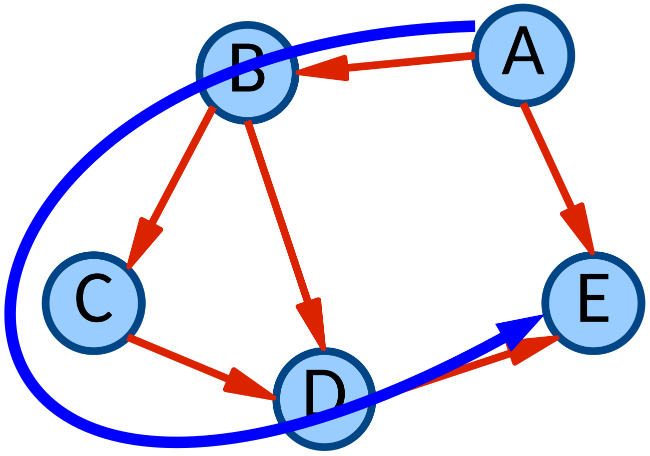
- What is a *graph* G?
  - A set N of nodes
  - A set E of edges
- When edges are directed from one node to another, the graph is a *directed graph*
- A *path* is a list of pairwise connected nodes



Graphs

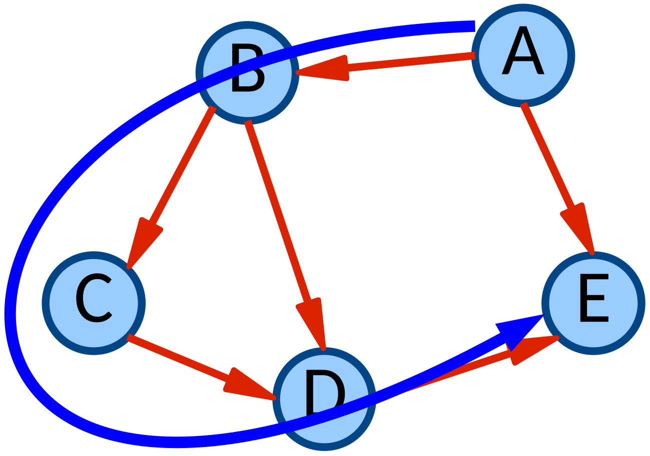
- What is a *graph* G?
  - A set N of nodes
  - A set E of edges
- When edges are directed from one node to another, the graph is a *directed graph*
- A *path* is a list of pairwise connected nodes

ABCDE



Graphs

- What is a *graph* G?
  - A set N of nodes
  - A set E of edges
- When edges are directed from one node to another, the graph is a *directed graph*
- A *path* is a list of pairwise connected nodes



ABCDE
ABDE
AE
BCDE
BD

Control Flow Graphs (CFGs)

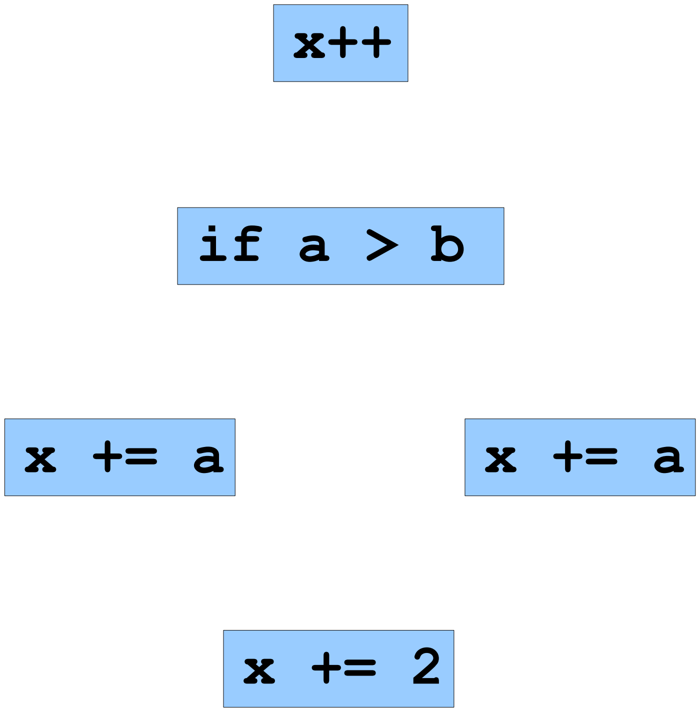
- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!

Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!
- *Control Flow Graphs*

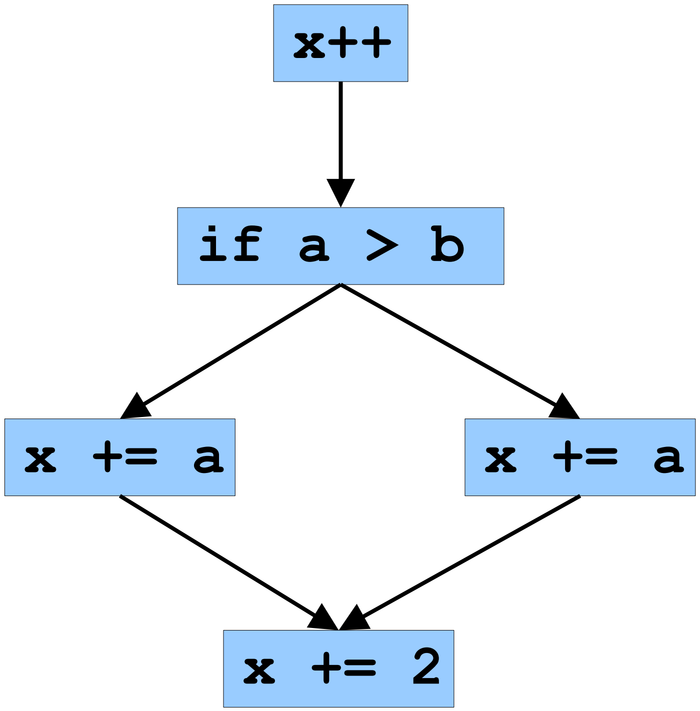
Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!
- *Control Flow Graphs*
  - Nodes comprise the code of a program



Control Flow Graphs (CFGs)

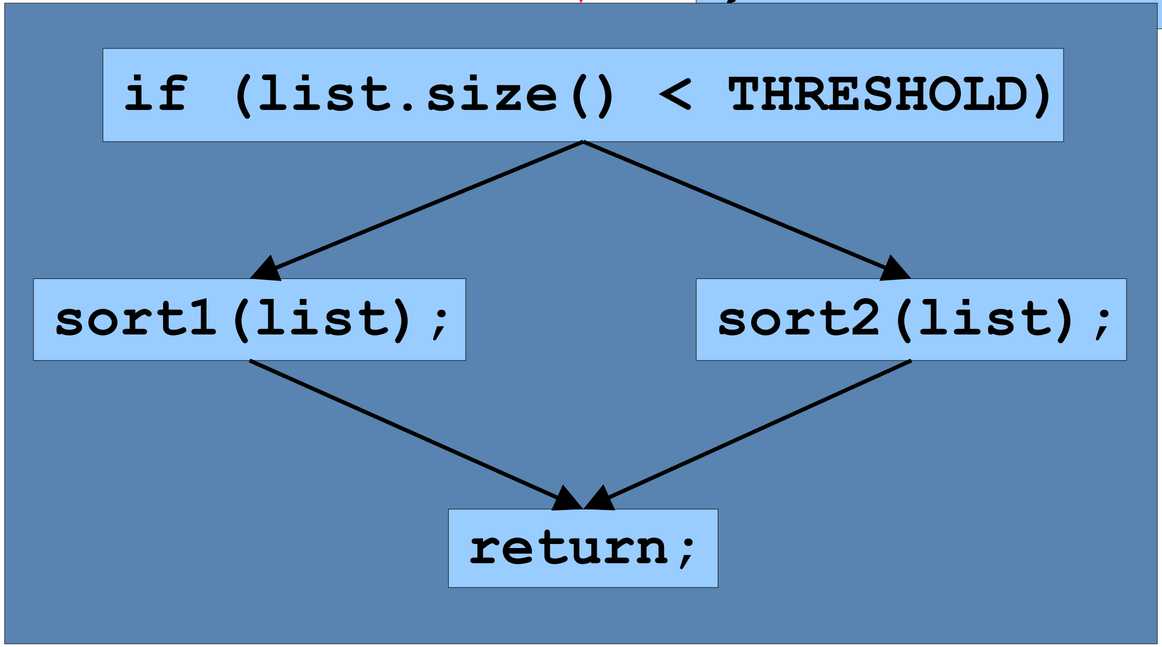
- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!
- *Control Flow Graphs*
  - Nodes comprise the code of a program
  - Edges show the paths that an execution may take through a program



Control Flow Graphs

Example:

```
void
sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```



Control Flow Graphs

Example:

```
void
sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

```
if (list.size() < THRESHOLD)
```

```
sort1(list);
```

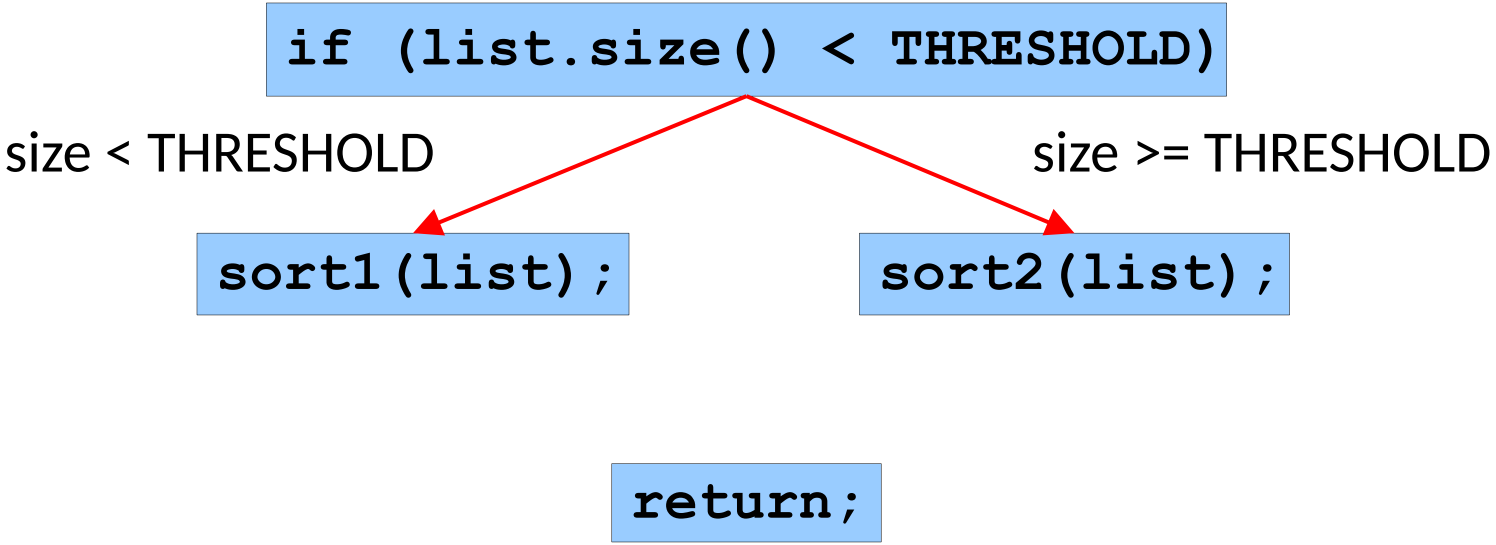
```
sort2(list);
```

```
return;
```

Control Flow Graphs

```
void
sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

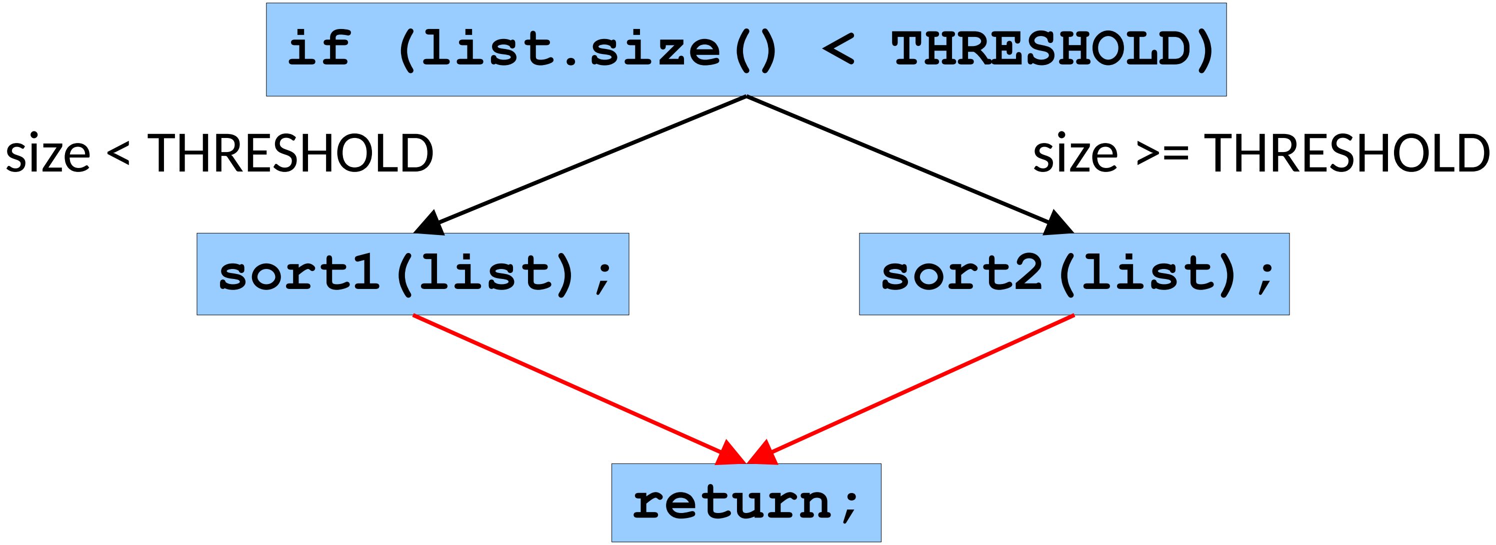
Example:



Control Flow Graphs

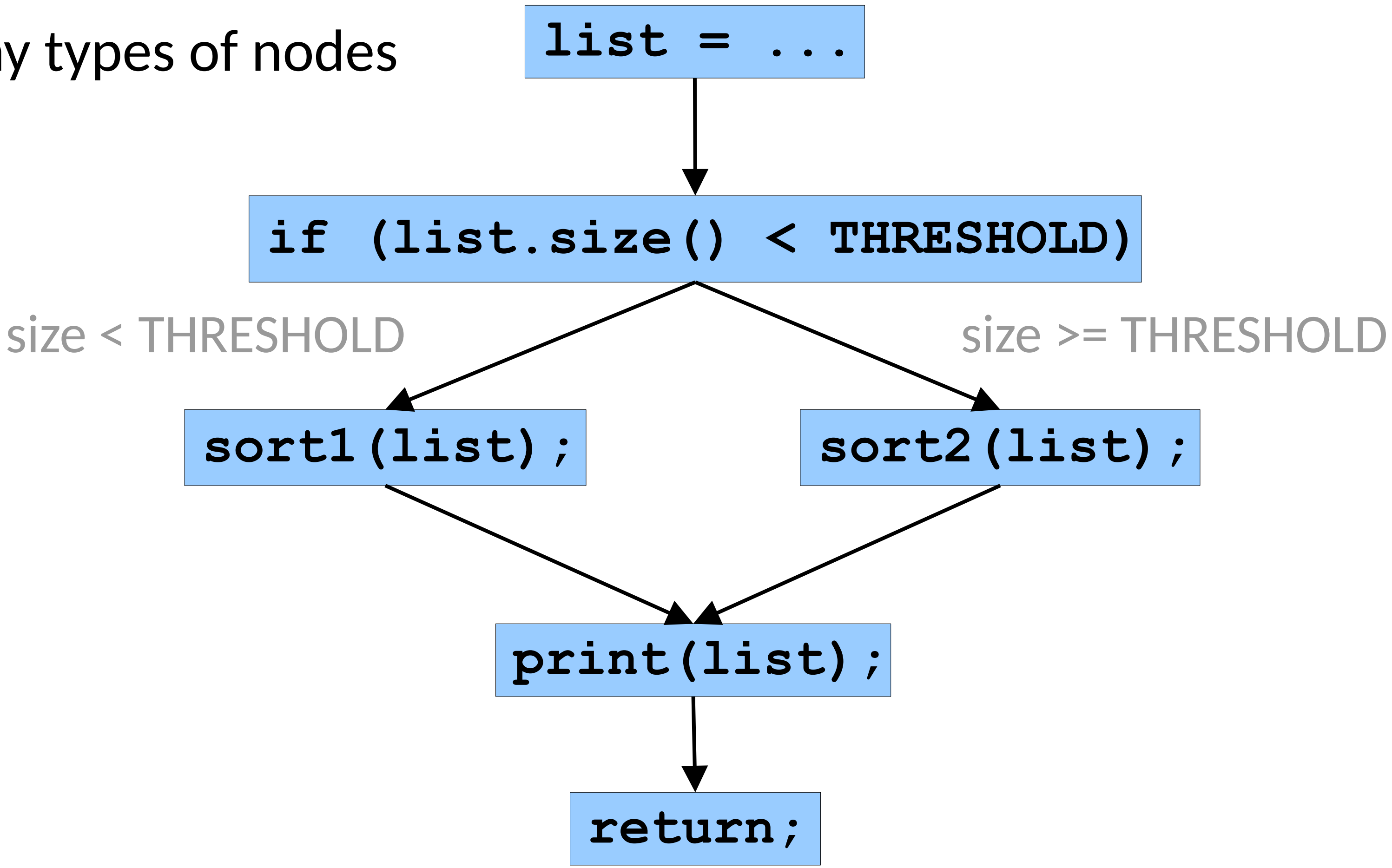
```
void
sortEfficiently(List list) {
  if (list.size() < THRESHOLD) {
    sort1(list);
  } else {
    sort2(list);
  }
}
```

Example:



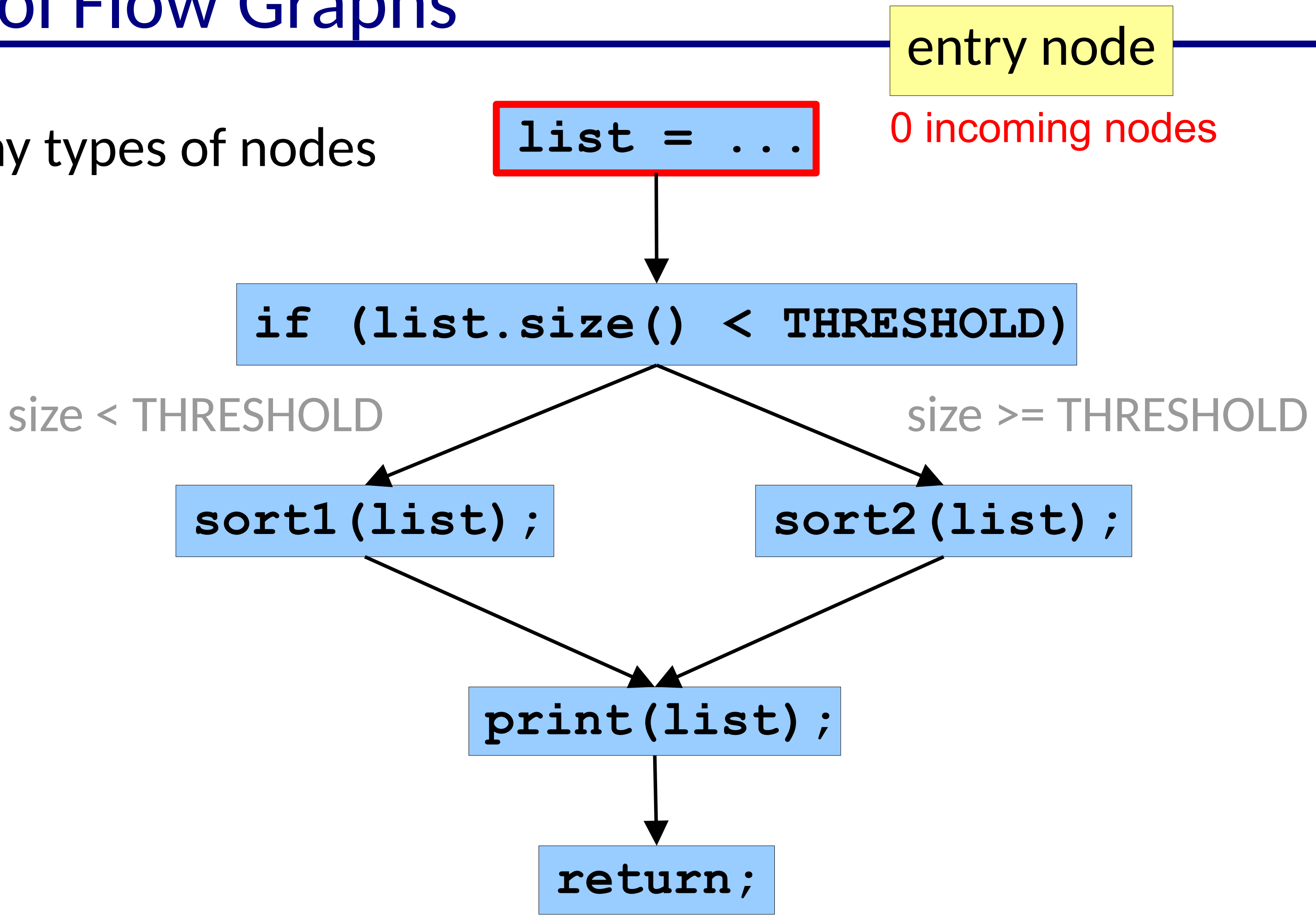
Control Flow Graphs

- Many types of nodes



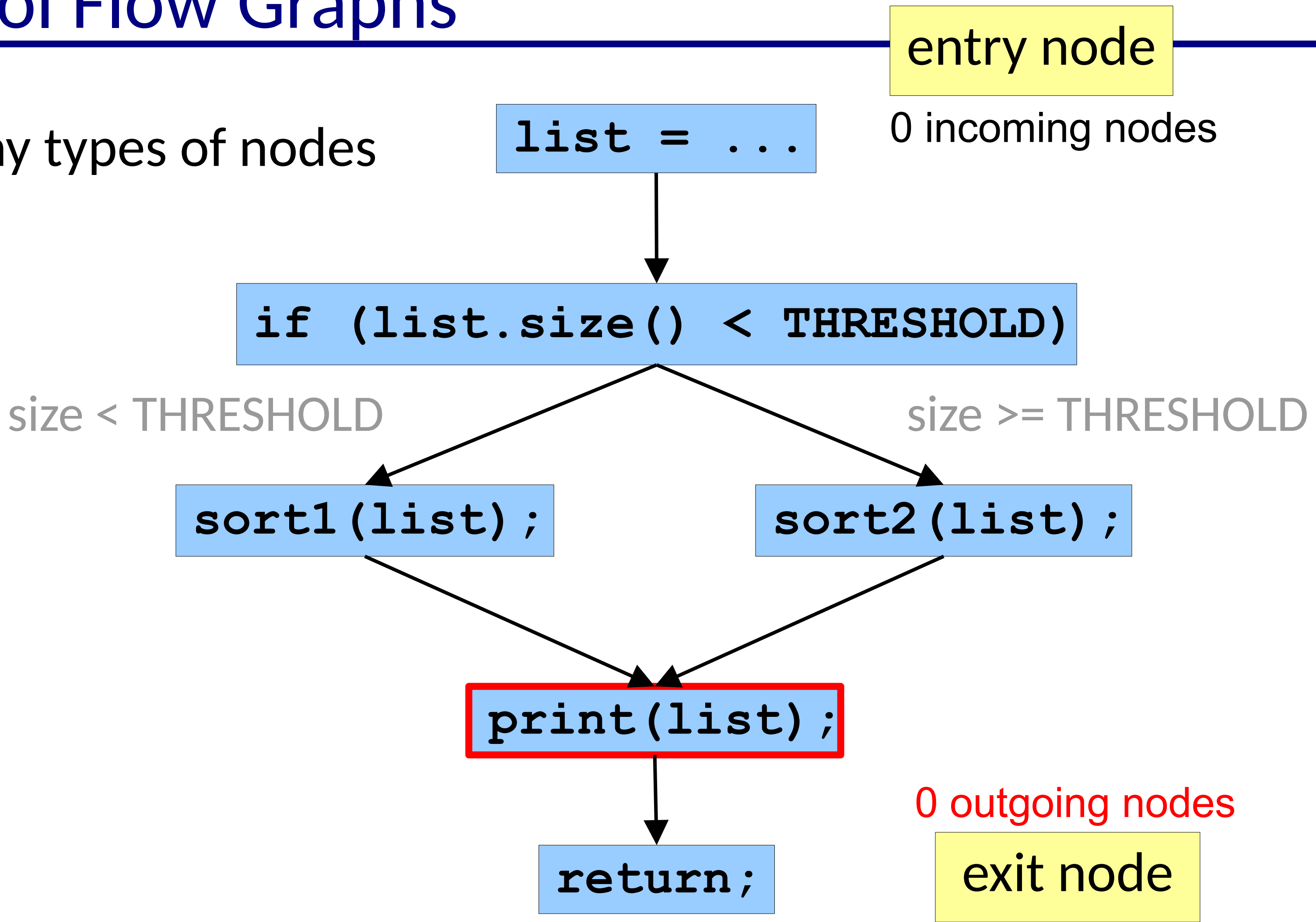
Control Flow Graphs

- Many types of nodes



Control Flow Graphs

- Many types of nodes

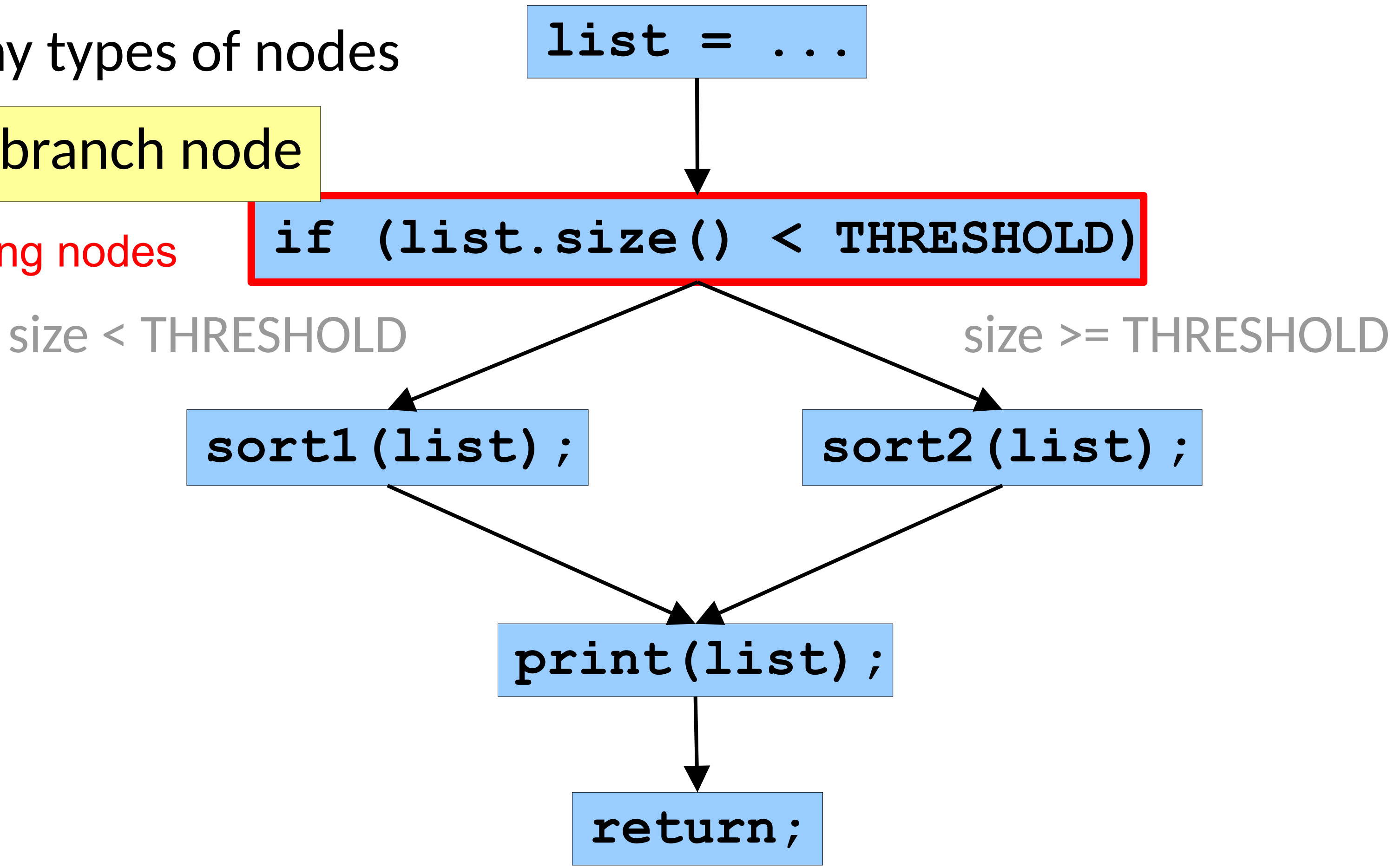


Control Flow Graphs

- Many types of nodes

decision / branch node

>1 outgoing nodes



Control Flow Graphs

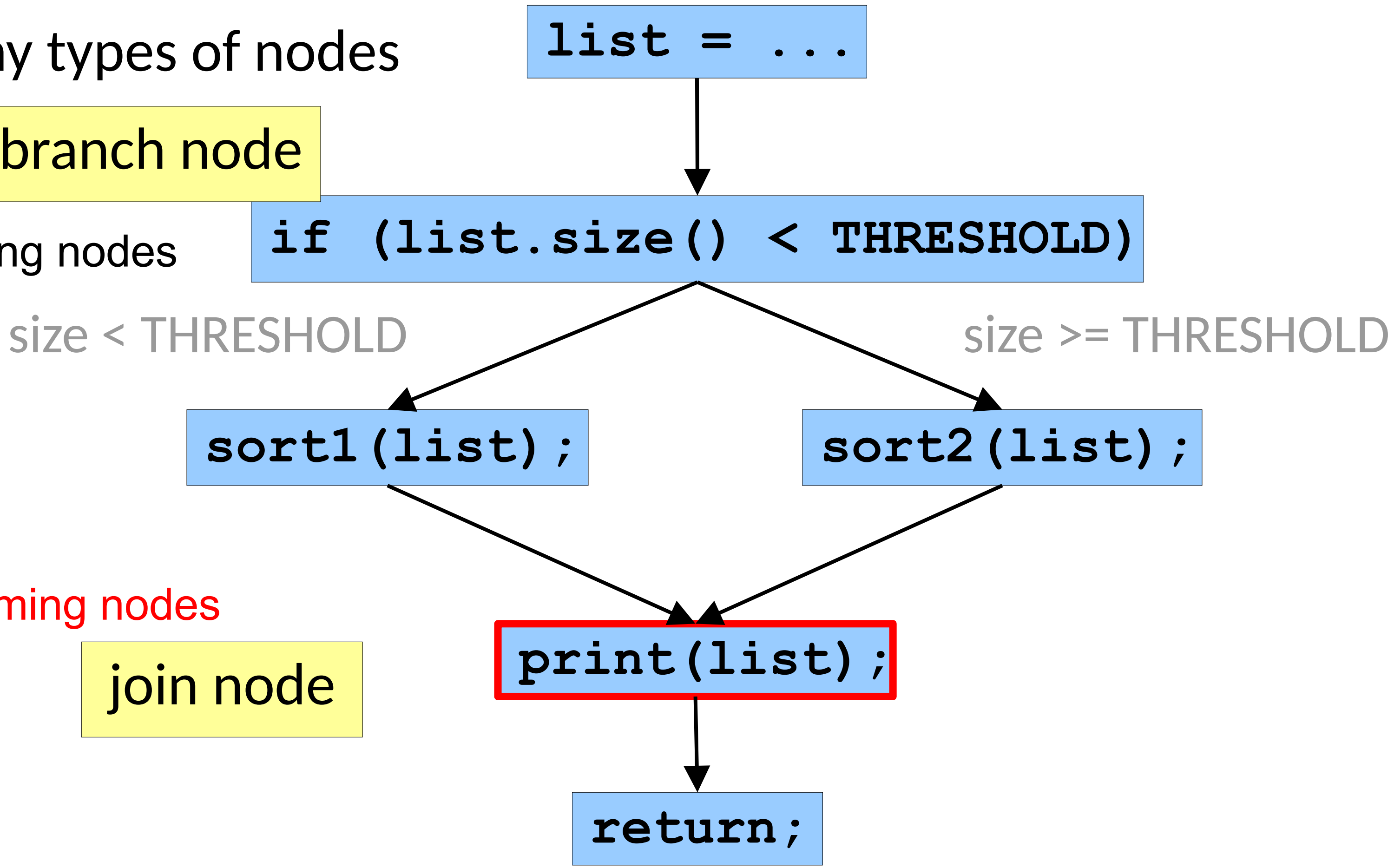
- Many types of nodes

decision / branch node

>1 outgoing nodes

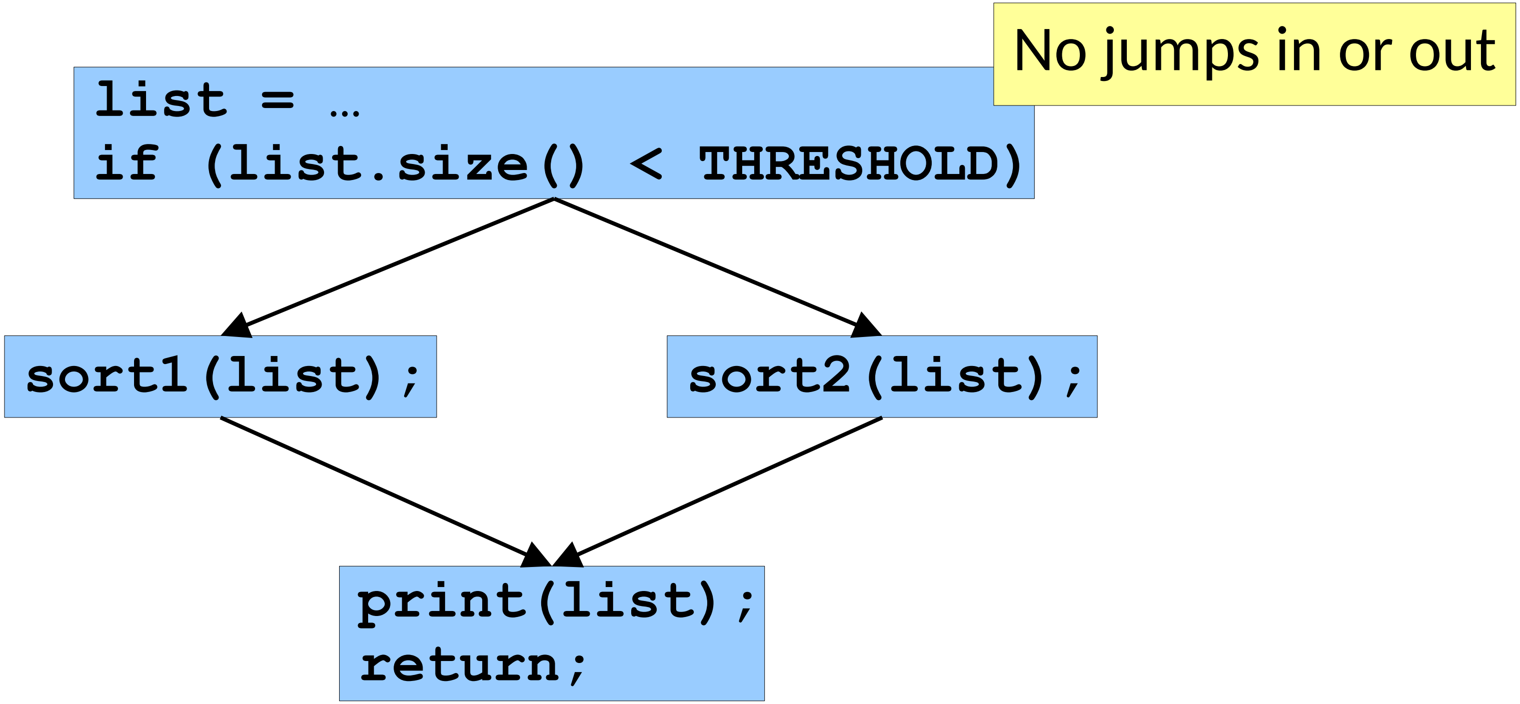
>1 incoming nodes

join node



Control Flow Graphs

- Straight line code is grouped into ***basic blocks***



Control Flow Graphs

- Many patterns arise from common constructs

```
q = 0;
r = x;
while r >= y {
  r = r - y;
  q = q + 1;
}
```

Control Flow Graphs

- Many patterns arise from common constructs

```
q = 0;
r = x;
while r >= y {
  r = r - y;
  q = q + 1;
}
```

```
for (i = 0; i < n; i++) {
   foo(i);
}
```

Control Flow Graphs

- Many patterns arise from common constructs

```
q = 0;
r = x;
while r >= y {
   r = r - y;
   q = q + 1;
}
```

```
switch  (x) {
   case a: foo(x); break;
   case b: bar(x);
   case c: baz(x); break;
}
```

```
for (i = 0; i < n; i++) {
   foo(i);
}
```

Control Flow Graphs

- Many patterns arise from common constructs

```
q = 0;
r = x;
while r >= y {
  r = r - y;
  q = q + 1;
}
```

```
switch  (x) {
   case a: foo(x); break;
   case b: bar(x);
   case c: baz(x); break;
}
```

```
for (i = 0; i < n; i++) {
   foo(i);
}
```

```
if (x == 0 || y/x > 1) {
    foo(x, y);
}
```

Control Flow Graphs

```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```

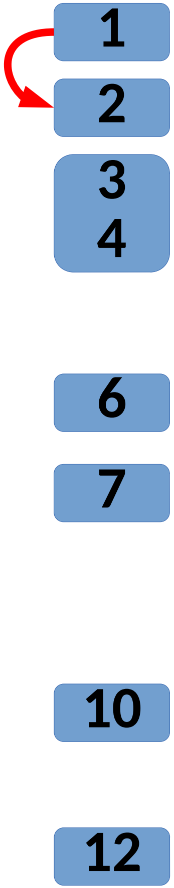

From: Ammann & Offutt

Control Flow Graphs

```
1     x = 0;
2     while (x < y) {
3        y = f (x, y);
4        if (y == 0) {
            break;
6        } else if (y < 0) {
7           y = y*2;
            continue;
         }
10       x = x + 1;
      }
12    print (y);
```


From: Ammann & Offutt

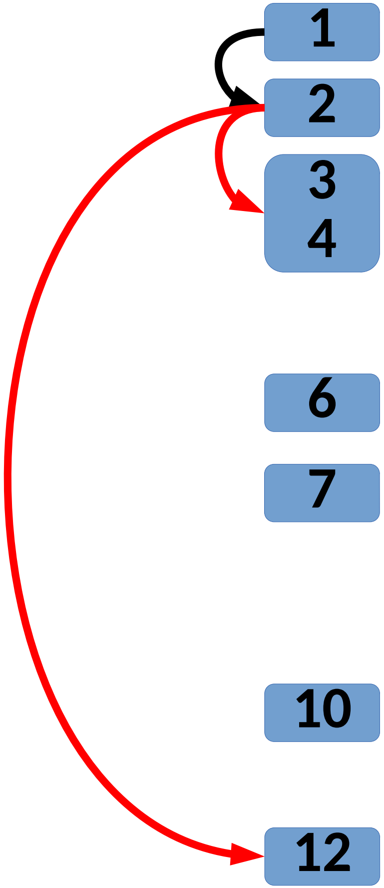
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

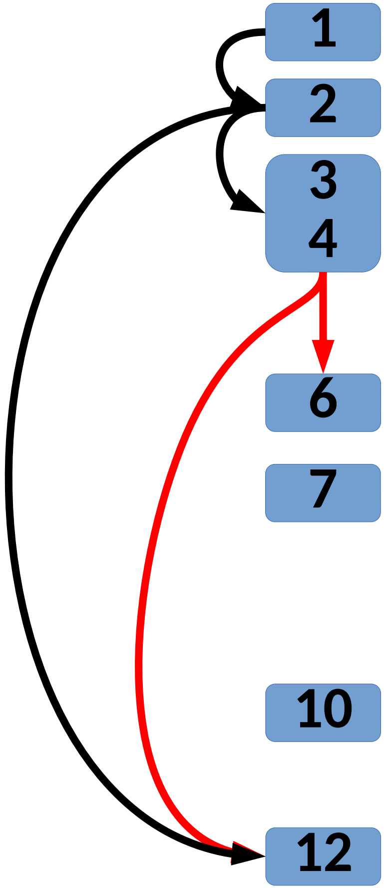
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

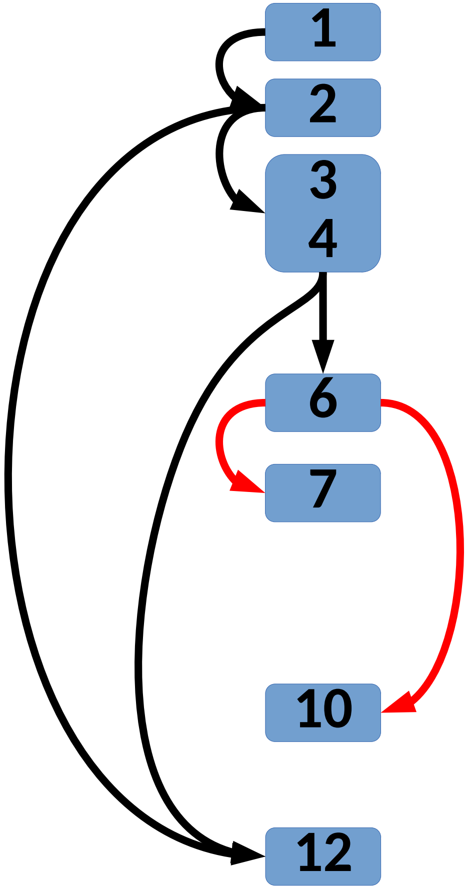
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

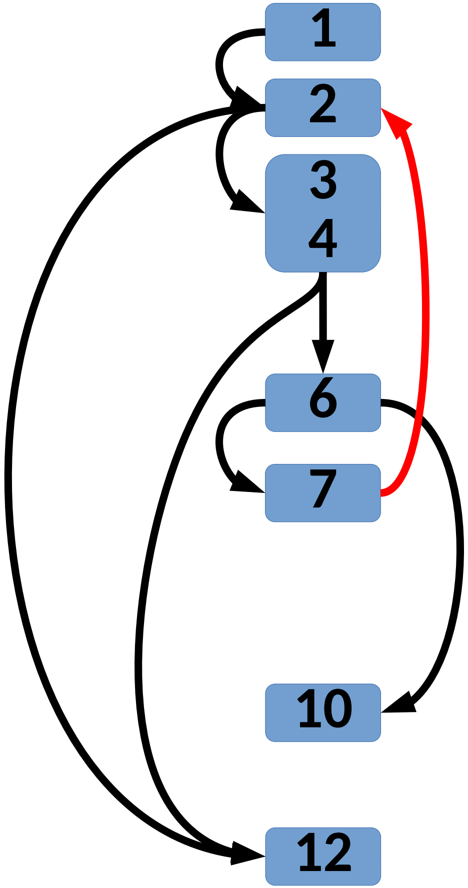
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

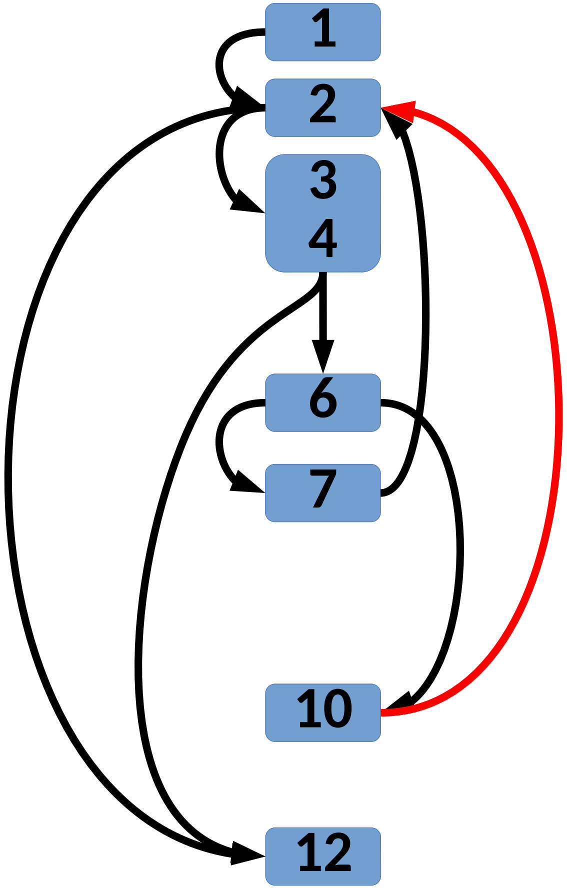
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

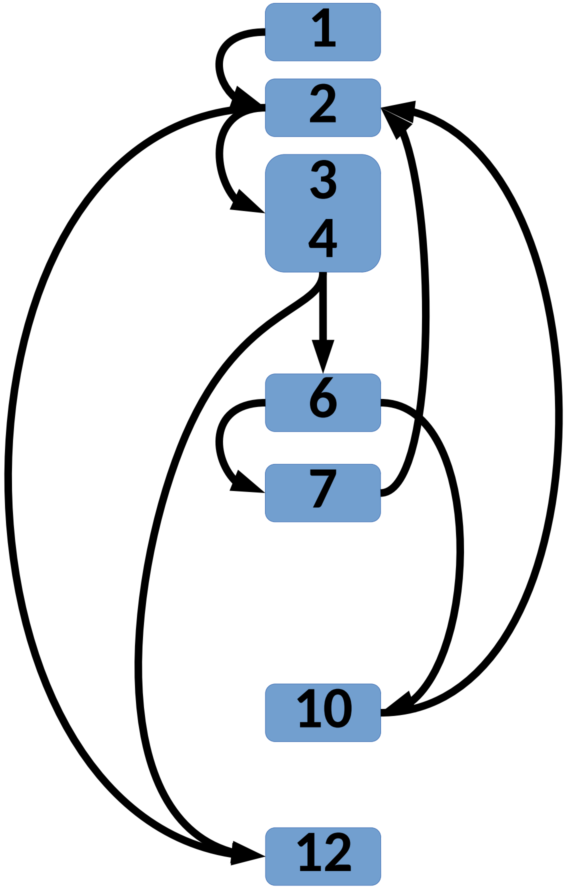
Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

Control Flow Graphs




```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```


From: Ammann & Offutt

CFG Coverage

- Statement Coverage → Node Coverage
  - Try to cover all *reachable* basic blocks

CFG Coverage

- Statement Coverage → Node Coverage
  - Try to cover all reachable basic blocks

Thinking in terms of node coverage can be more efficient. **Why**?

CFG Coverage

- Statement Coverage → Node Coverage
  - Try to cover all reachable basic blocks
- Branch Coverage → Edge Coverage
  - Try to cover all *reachable* paths of length ≤ 1

CFG Coverage

- Statement Coverage → Node Coverage
  - Try to cover all reachable basic blocks
- Branch Coverage → Edge Coverage
  - Try to cover all reachable paths of length ≤ 1

How do node & edge coverage compare? **Why**?

Pragmatic Concerns

- The *goal* is full coverage (of whatever criteria)

Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*

Pragmatic Concerns

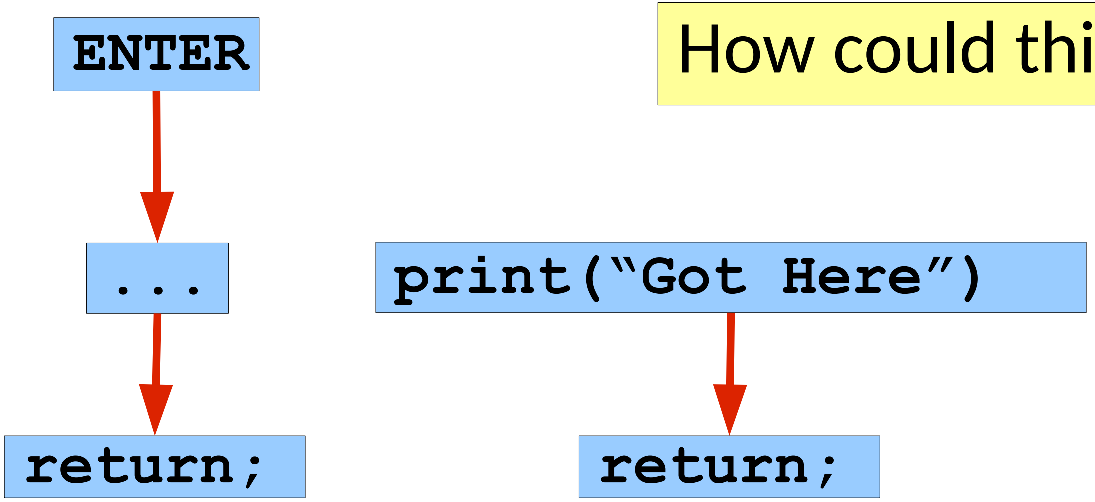
- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code

How could this happen?

Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code

How could this happen?

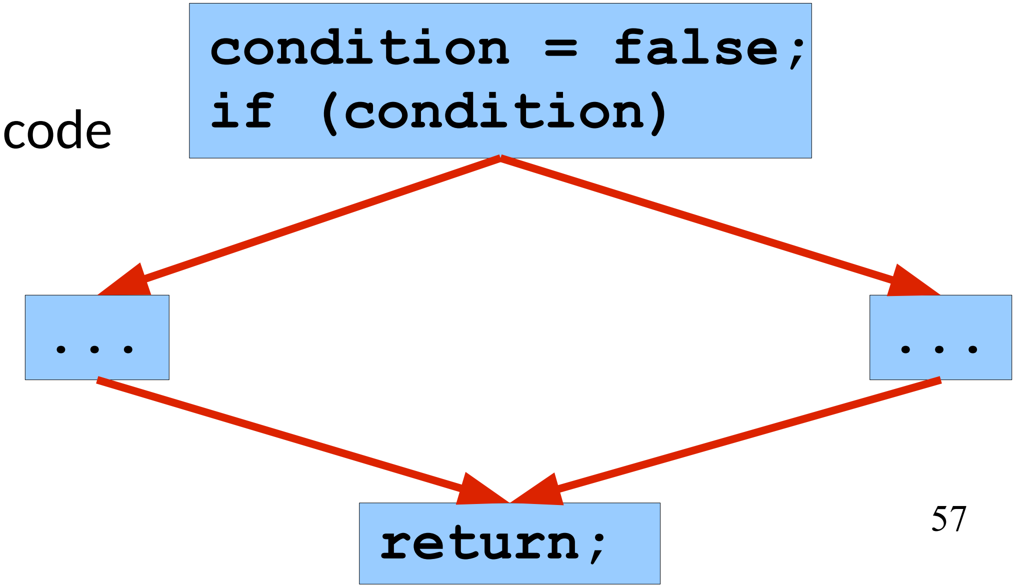


Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code

Pragmatic Concerns

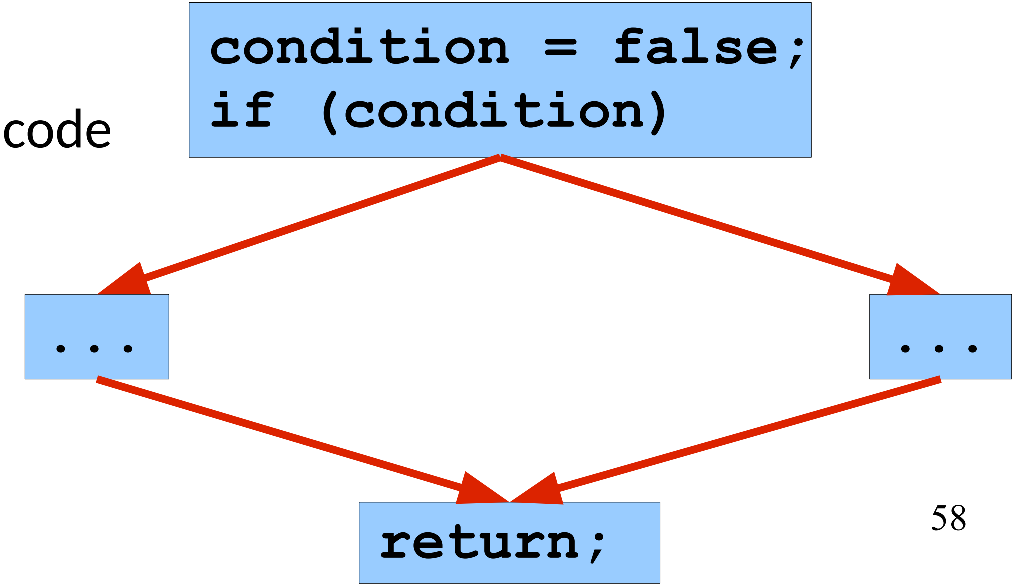
- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code



Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code

This can be *undecidable*!



Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code
- So what do you do in practice?
  - No, really. What have you done in practice?

Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider *reachability*
  - *Syntactic* Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code
- So what do you do in practice?
  - No, really. What have you done in practice?
  - Relative degrees of coverage matter (40%? 80%?)

Pragmatic Concerns

- Many branch coverage tools work only at `if` granularity

Pragmatic Concerns

- Many branch coverage tools work only at `if` granularity

```
def misleading(x: int, y: int) -> bool:
  if x > 0 or y > 0:
      return x + y >
  return x * y < 20
```

```
def test_misleading():
    assert not misleading(0,0)
    assert misleading(1,0)
```

pytest-cov → 100% branch coverage

Pragmatic Concerns

- Many branch coverage tools work only at **`if`** granularity

```
def misleading(x: int, y: int) -> bool:
```

```
ding():
isleading(0,0)
```

```
function misleading(x, y) {
  if (x > 0 || y > 0) {
    return x + y > 0;
  }
  return x * y < 20;
}
```

```
test('misleading branch coverage', () => {
  expect(misleading(0, 0)).toBe(true);
  expect(misleading(1, 0)).toBe(true);
});
```

jest → 100% branch coverage

Pragmatic Concerns

- Many branch coverage tools work only at `if` granularity
  - "*Condition coverage*" can complement this but is also misleading (more later)

Pragmatic Concerns

- Many branch coverage tools work only at `if` granularity
  - "Condition coverage" can complement this but is also misleading
- Other tools consider short-circuits to be branches
  - Common in native languages, Java, …
  - Recommended practice by the FAA….

Pragmatic Concerns

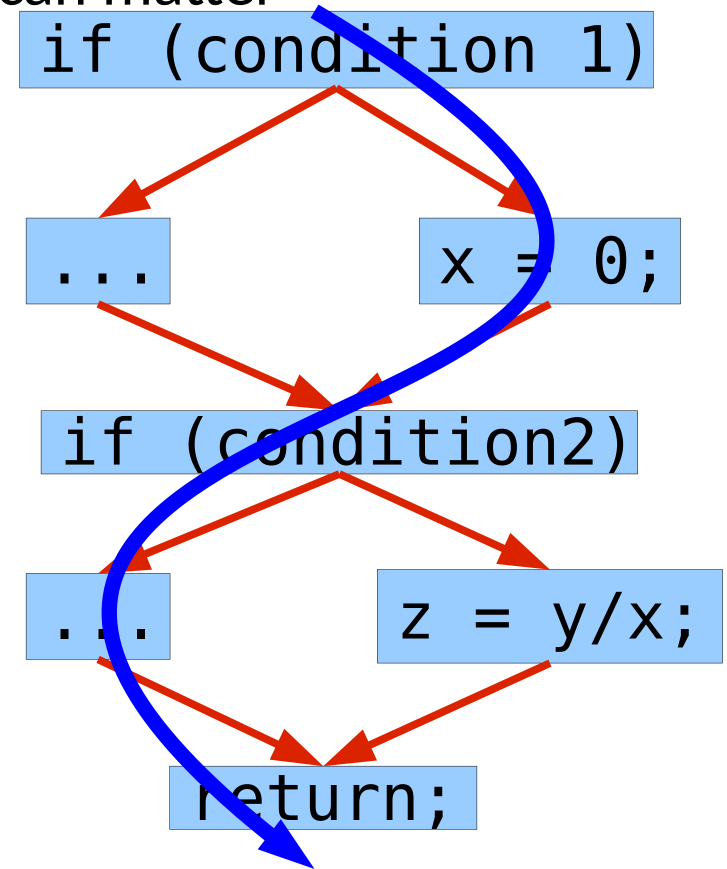
- Many branch coverage tools work only at `if` granularity
  - “Condition coverage” can complement this but is also misleading
- Stronger graph coverage criteria can help

CFG Coverage

- The path taken by each test can matter

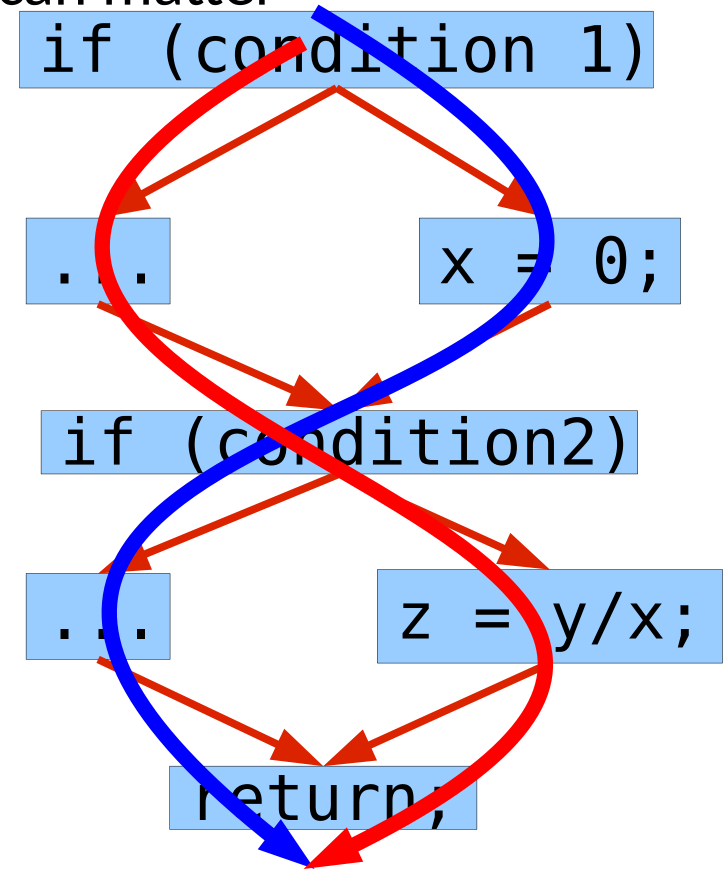
CFG Coverage

- The path taken by each test can matter



CFG Coverage

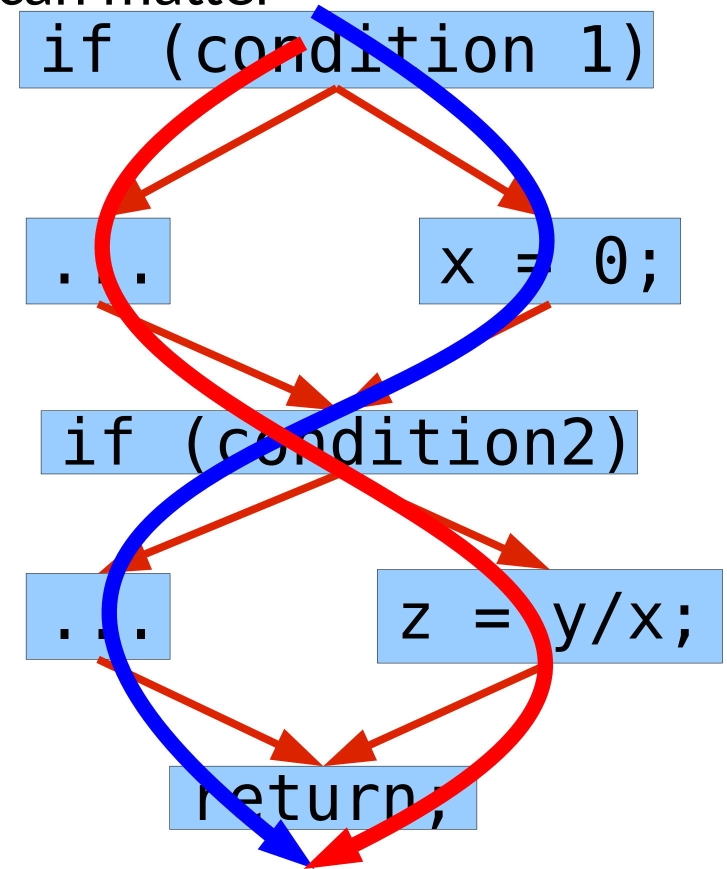
- The path taken by each test can matter



CFG Coverage

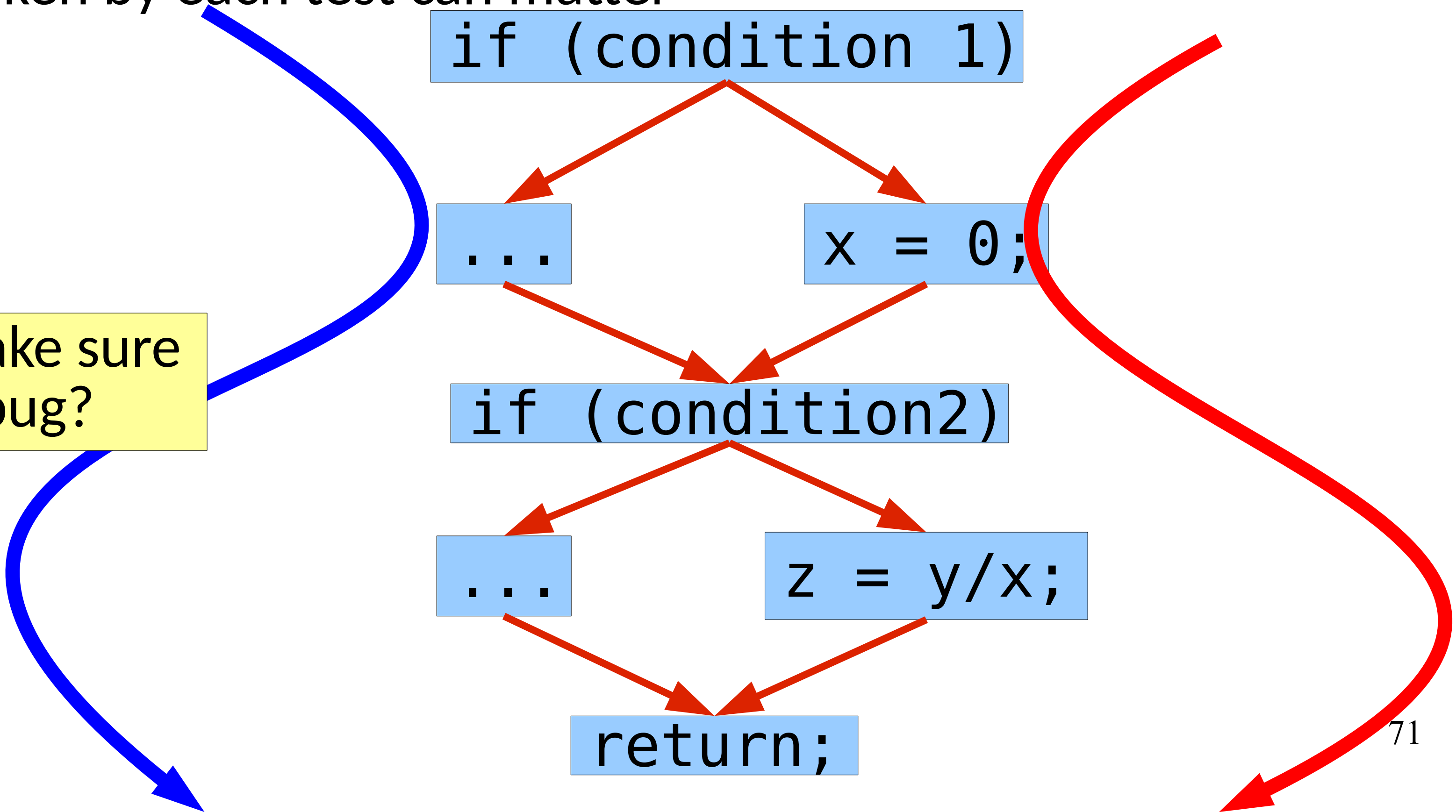
- The path taken by each test can matter

Full edge coverage & no bugs found



CFG Coverage

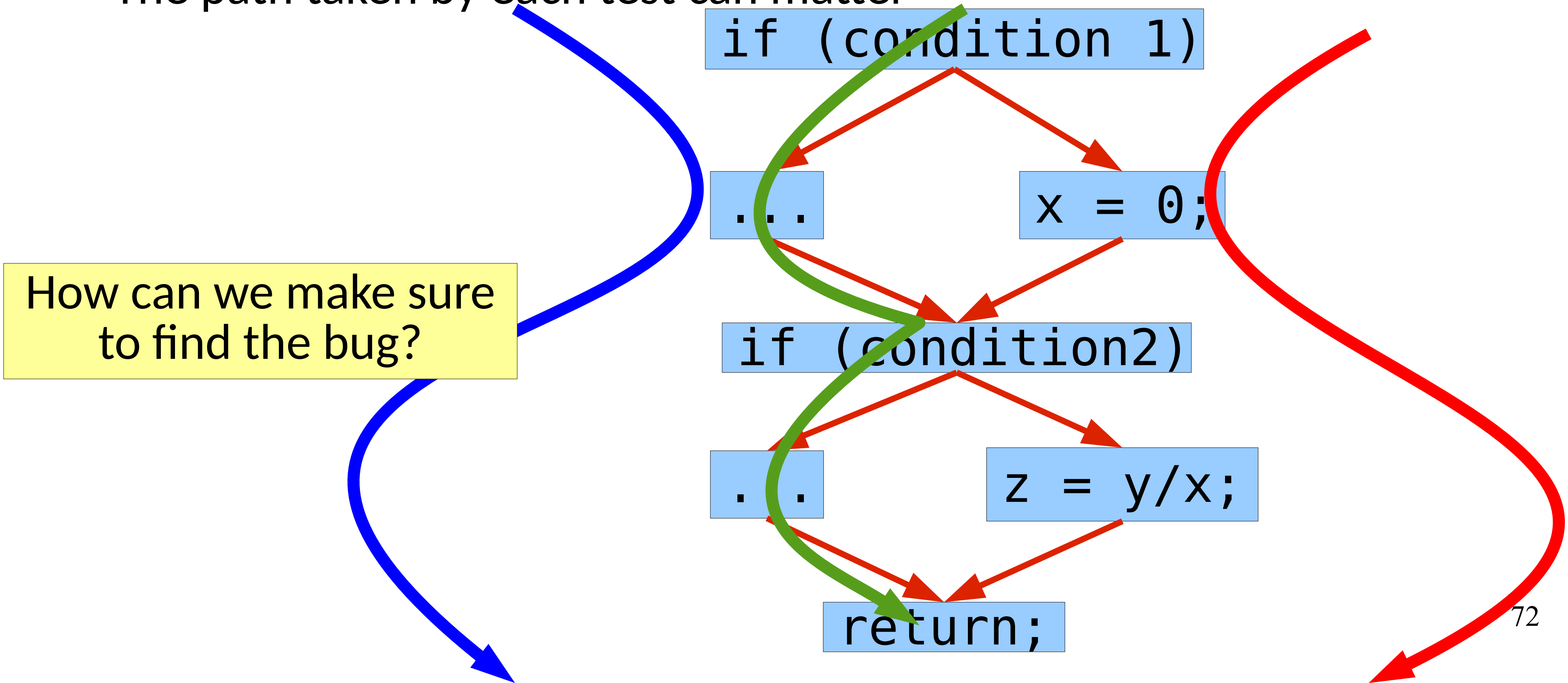
- The path taken by each test can matter



How can we make sure to find the bug?

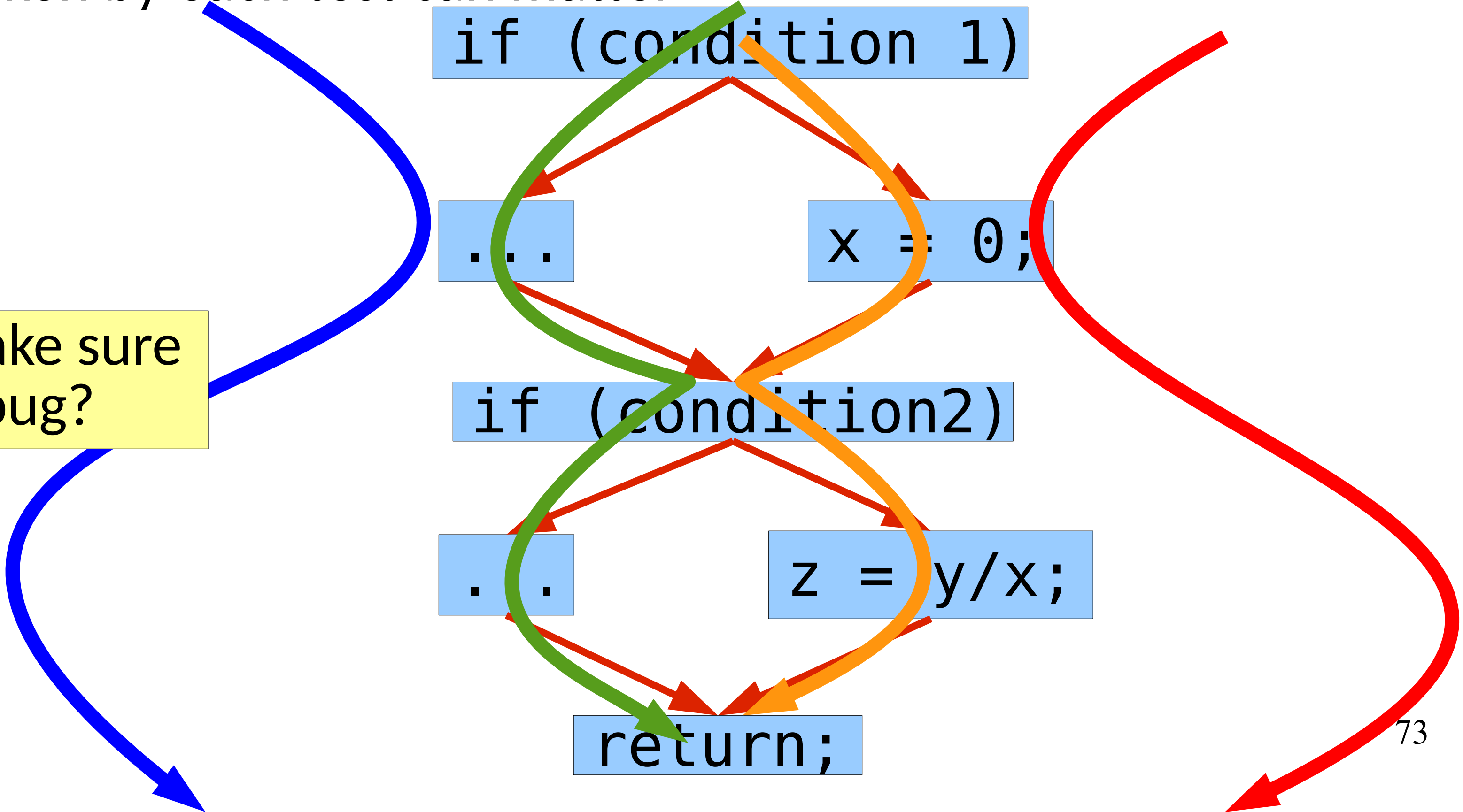
CFG Coverage

- The path taken by each test can matter



CFG Coverage

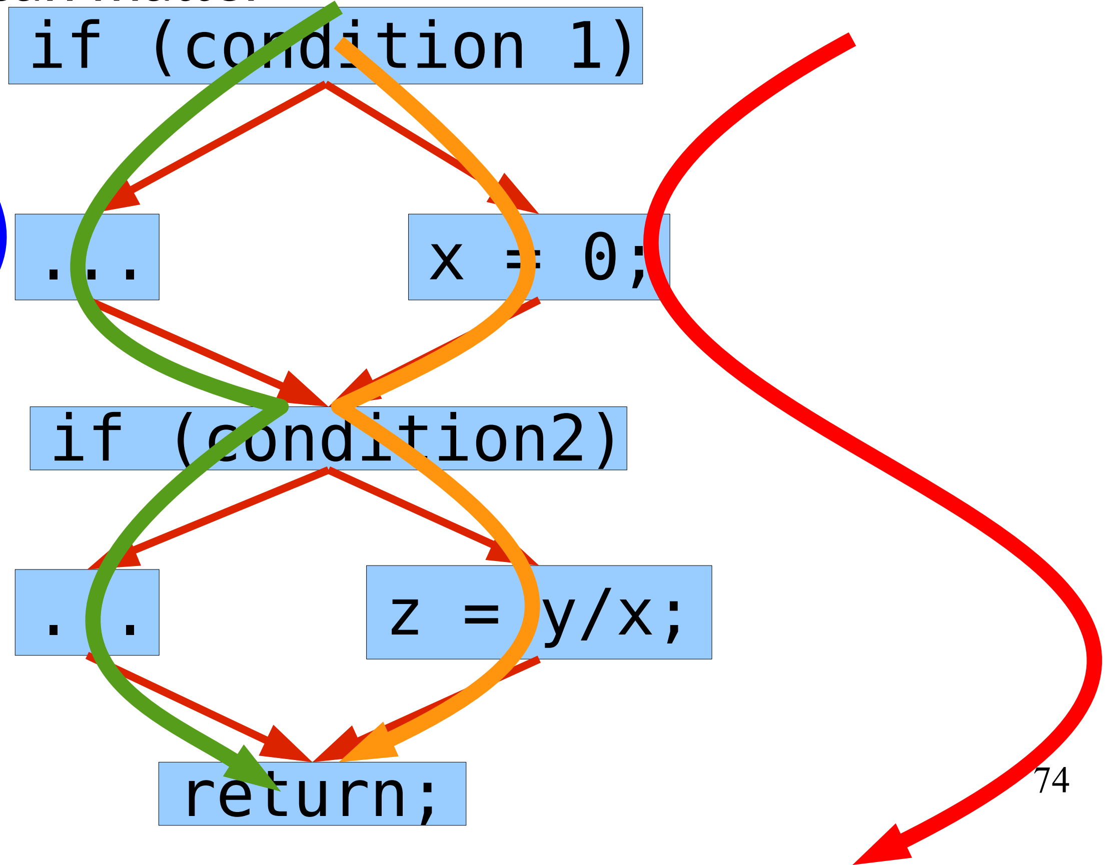
- The path taken by each test can matter



How can we make sure to find the bug?

CFG Coverage

- The path taken by each test can matter



How can we make sure to find the bug?

Testing *all paths* exposes the bug.

Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Path Coverage

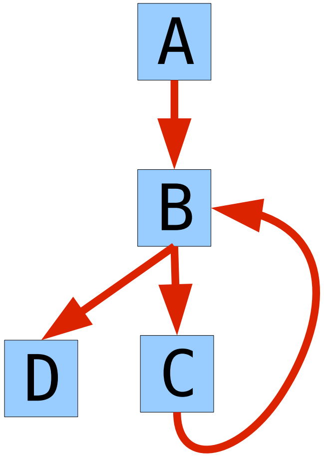
- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?

Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?

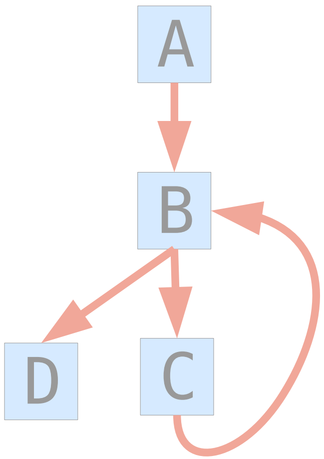


Infeasibility

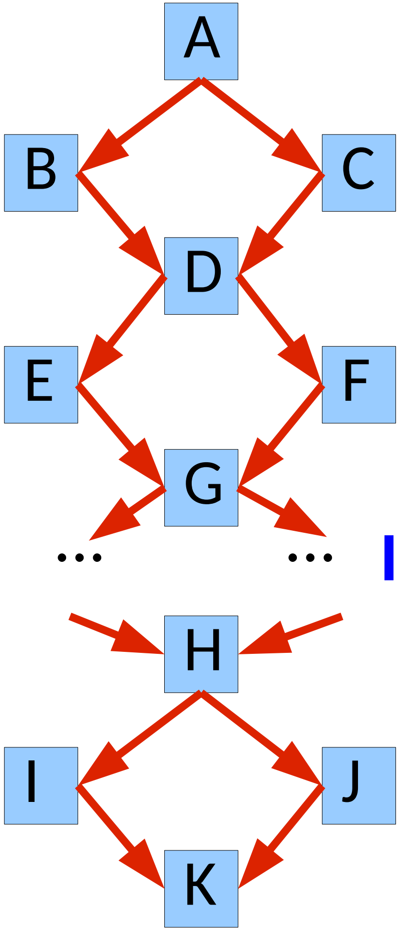
Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?



Infeasibility



Intractability

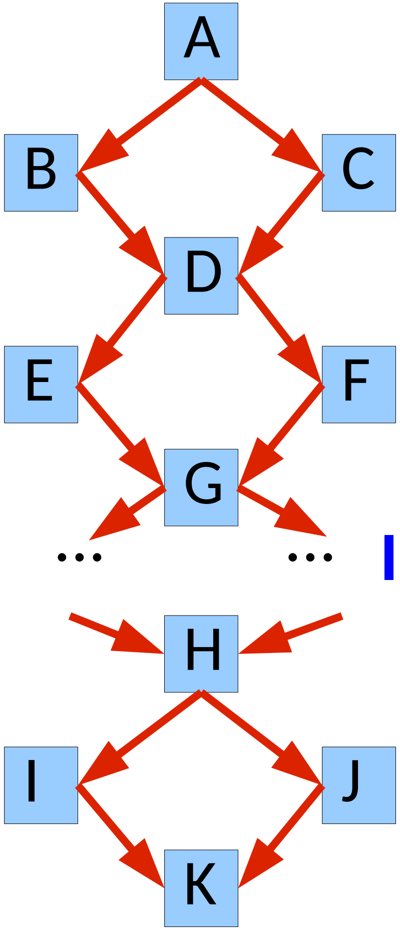
Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?

How many paths?

How does this relate to input based approaches?

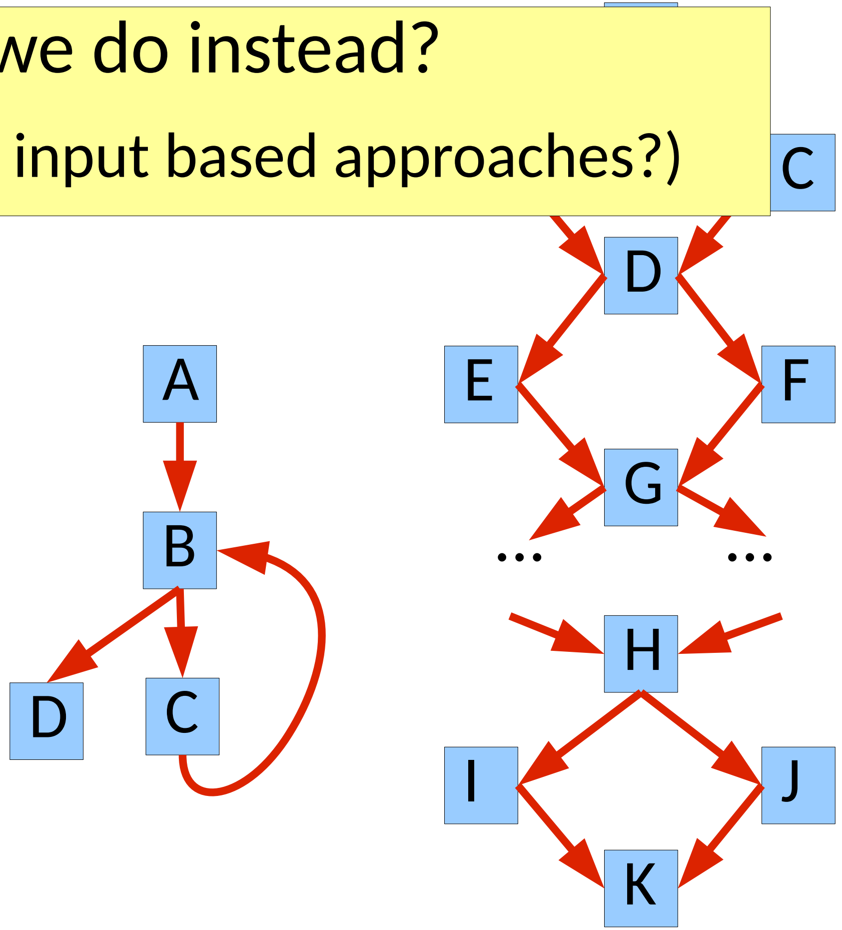


Intractability

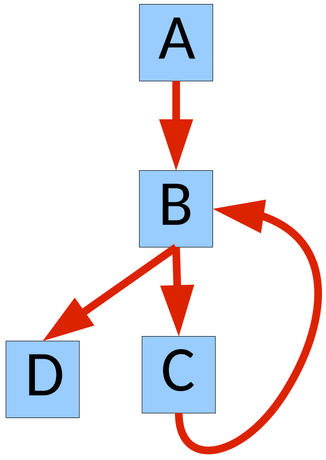
Compromises?

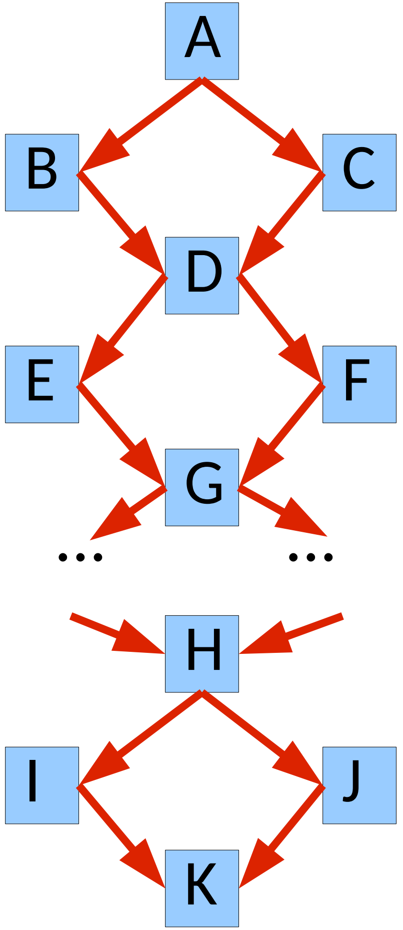
What could we do instead?

(How did we handle the input based approaches?)



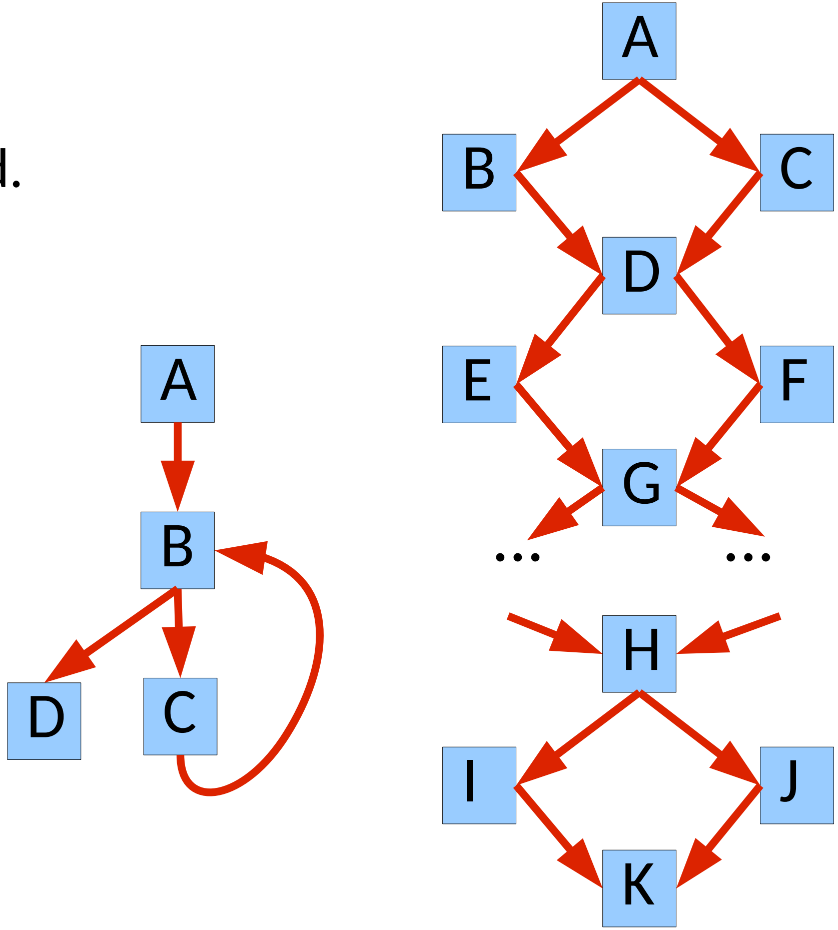
Compromises?





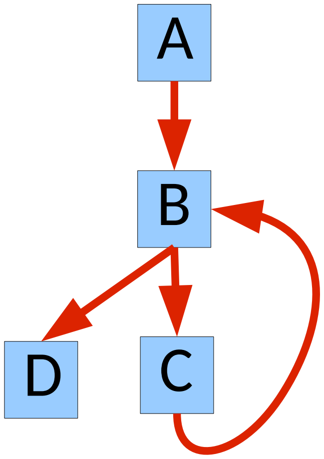
Compromises?

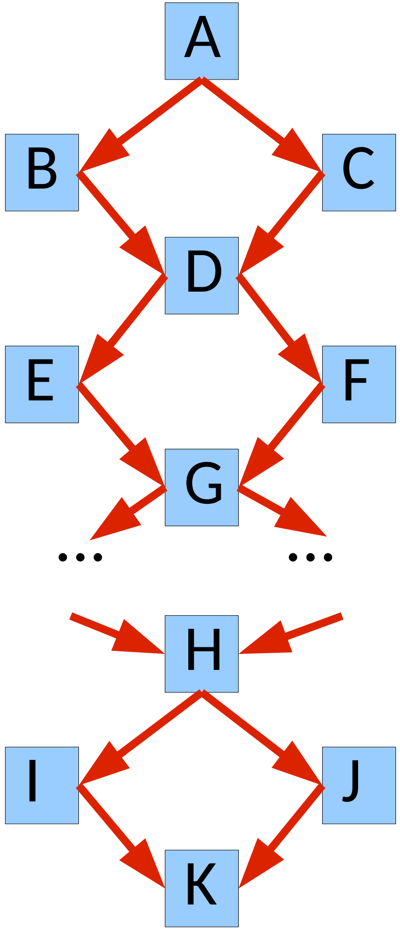
- Edge Pair Coverage
  - Each path of length <= 2 is tested.



Compromises?

- Edge Pair Coverage
  - Each path of length <= 2 is tested.
- Specified Path Coverage
  - Given a number k, test k paths



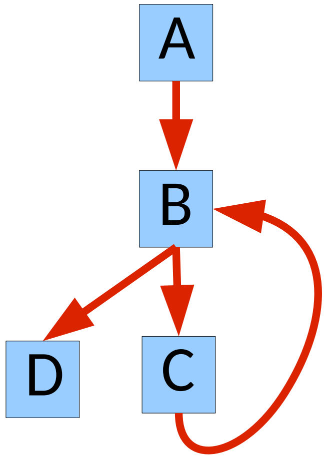


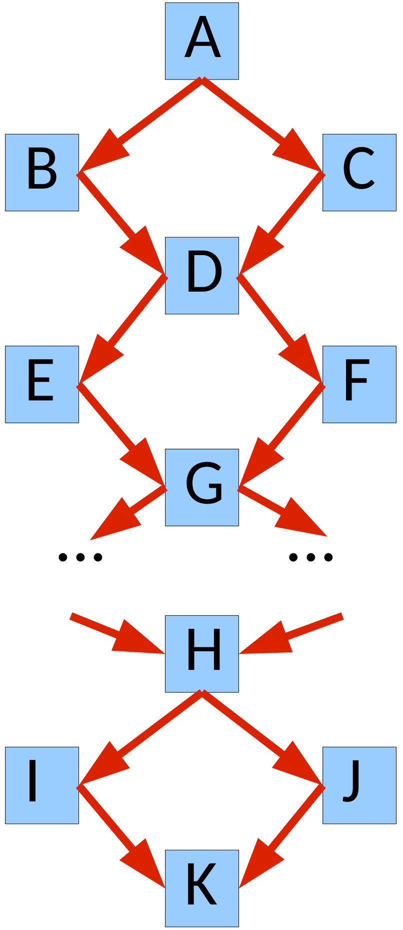
Compromises?

- Edge Pair Coverage
  - Each path of length <= 2 is tested.
- Specified Path Coverage
  - Given a number k, test k paths

What do these look like?

Are they good?



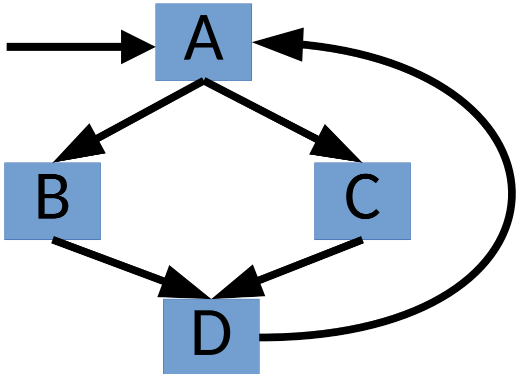


Coping With Loops

- What criteria do you use when testing loops?

Coping With Loops

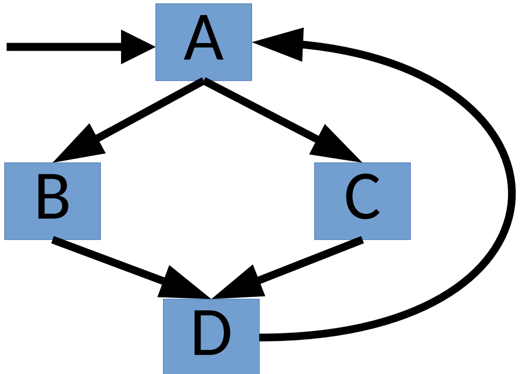
- What criteria do you use when testing loops?
- Simple Paths
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)



AB
CD
BDA***B***
~~***A***BD***A***C~~

Coping With Loops

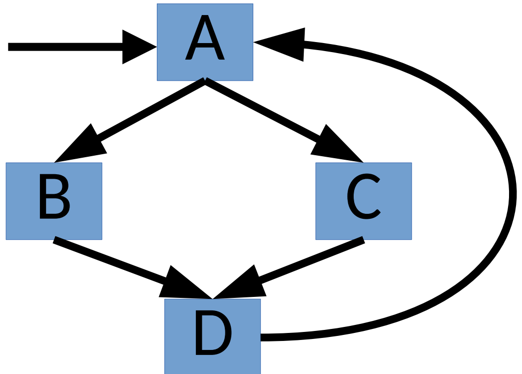
- What criteria do you use when testing loops?
- Simple Paths
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the *acyclic* behaviors of a program



AB
CD
BDA***B***
~~***A***BD*A*C~~

Coping With Loops

- What criteria do you use when testing loops?
- Simple Paths
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program



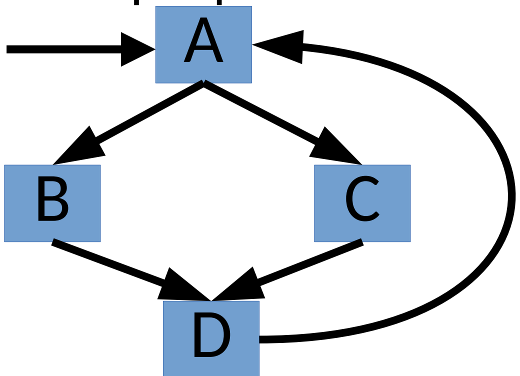
How many may there be?

Coping With Loops

- What criteria do you use when testing loops?
- Simple Paths
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program
- Prime Paths
  - A simple path that is not a subpath of any other simple path

Coping With Loops

- What criteria do you use when testing loops?
- Simple Paths
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program
- Prime Paths
  - A simple path that is not a subpath of any other simple path



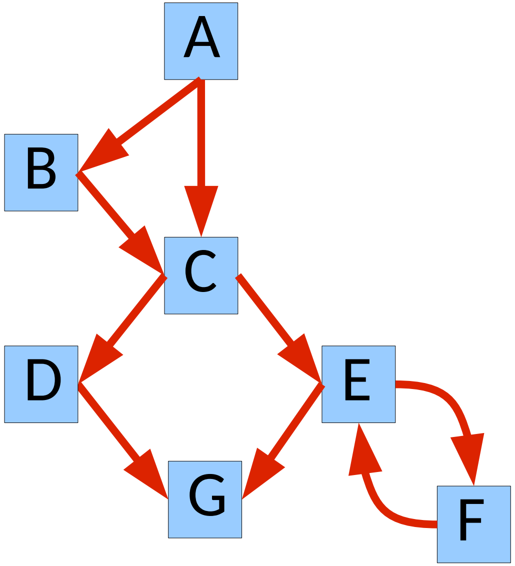
What does this provide?
What do they look like?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths

Coping With Loops

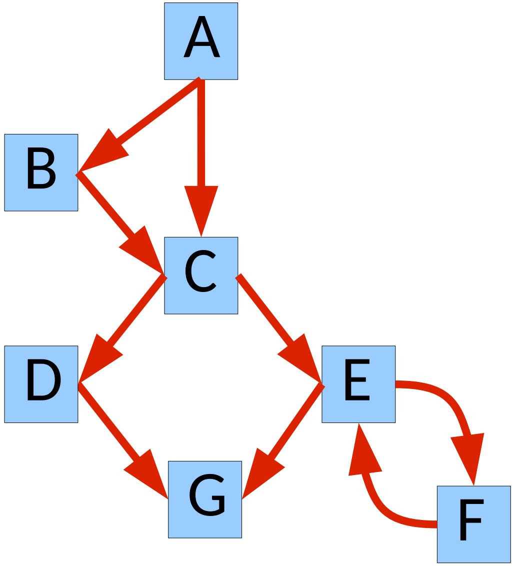
- Prime Path Coverage
  - Cover all prime paths



Example from Ammann & Offutt

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths

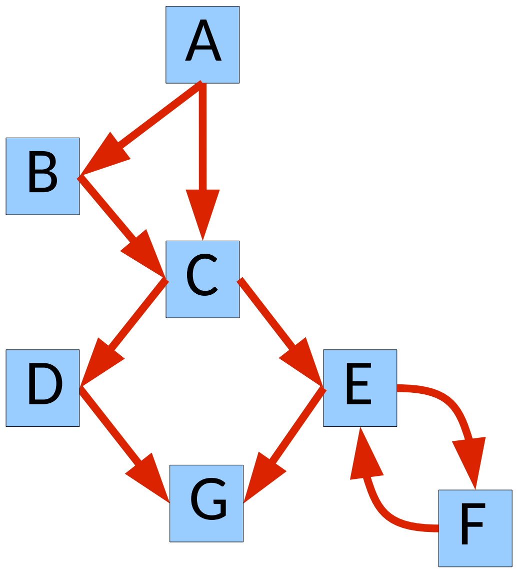


Example from Ammann & Offutt

What are the prime paths?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths



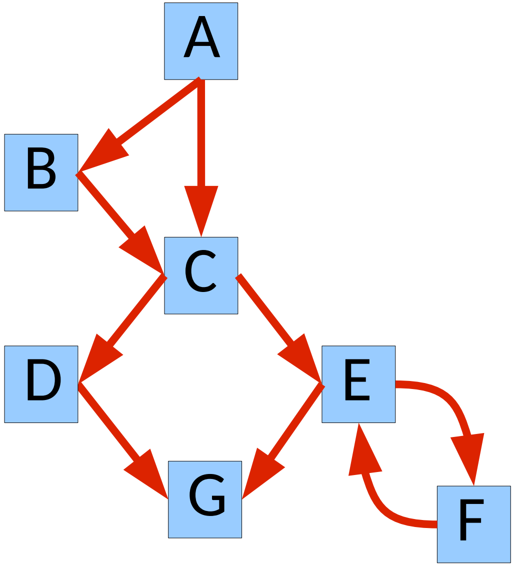
Example from Ammann & Offutt

What are the prime paths?

How many simple paths?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths



Example from Ammann & Offutt

What are the prime paths?

How many simple paths?

Can you intuitively explain what prime paths capture?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths

Are these tests good or bad?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths

Are these tests good or bad?

Do they address all of the problems with path coverage?

Coping With Loops

- Prime Path Coverage
  - Cover all prime paths

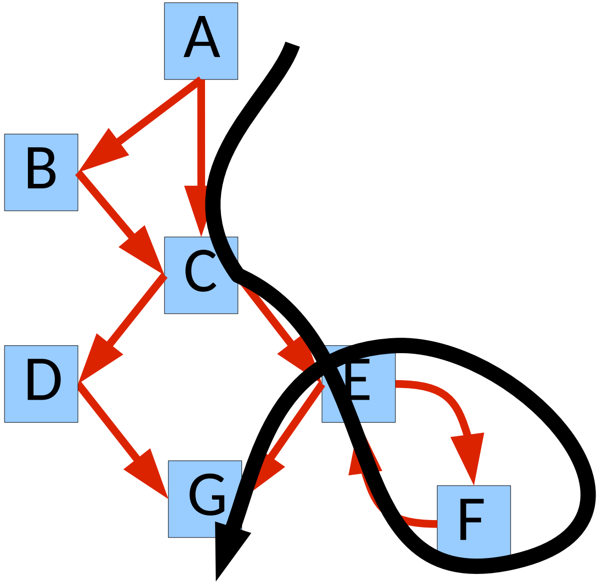
Are these tests good or bad?

Do they address all of the problems with path coverage?

Can you think of things they miss?

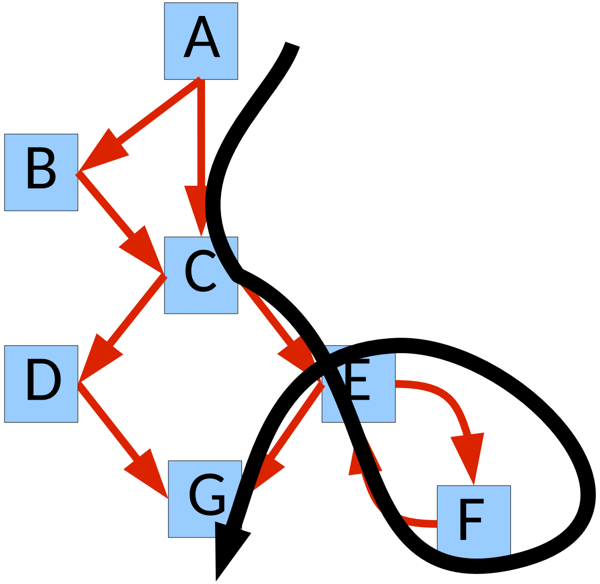
Turning Them Into Tests

- Reconsider



Turning Them Into Tests

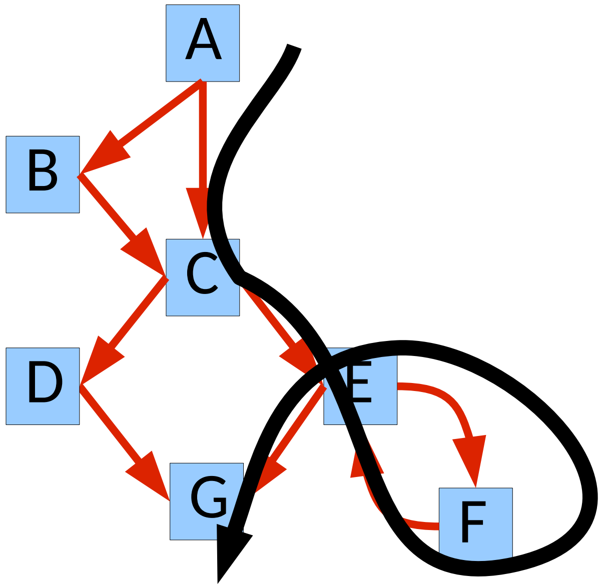
- Reconsider



Is this path prime?

Turning Them Into Tests

- Reconsider

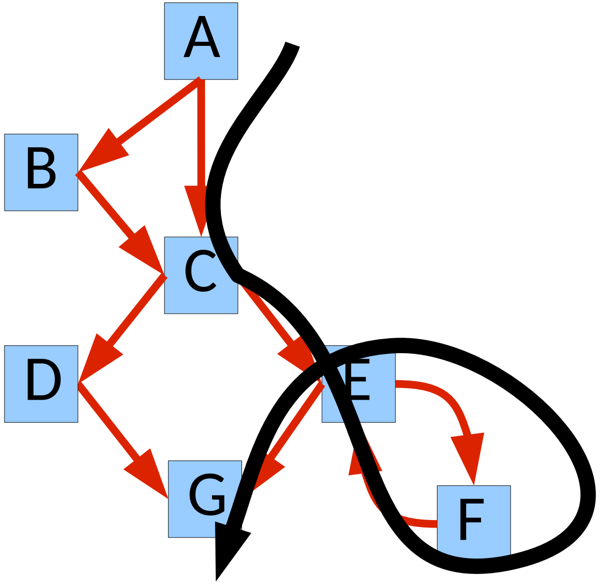


Is this path prime?

Is it still useful?

Turning Them Into Tests

- Reconsider



Is this path prime?

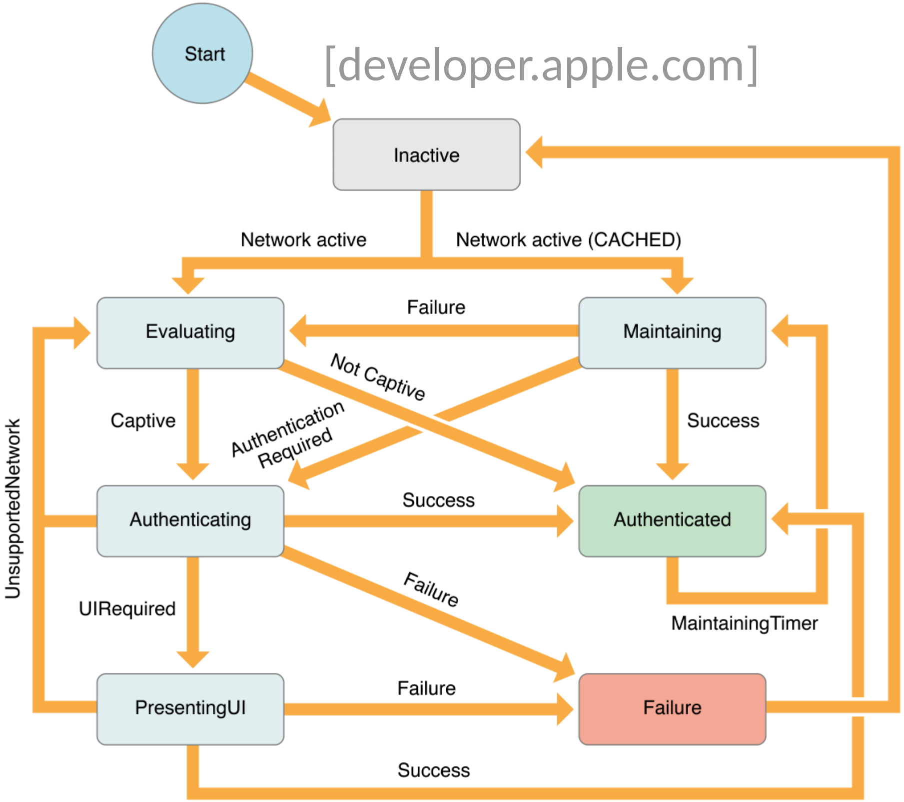
Is it still useful?

One test may cover multiple prime paths!
Requirements ≠ Tests

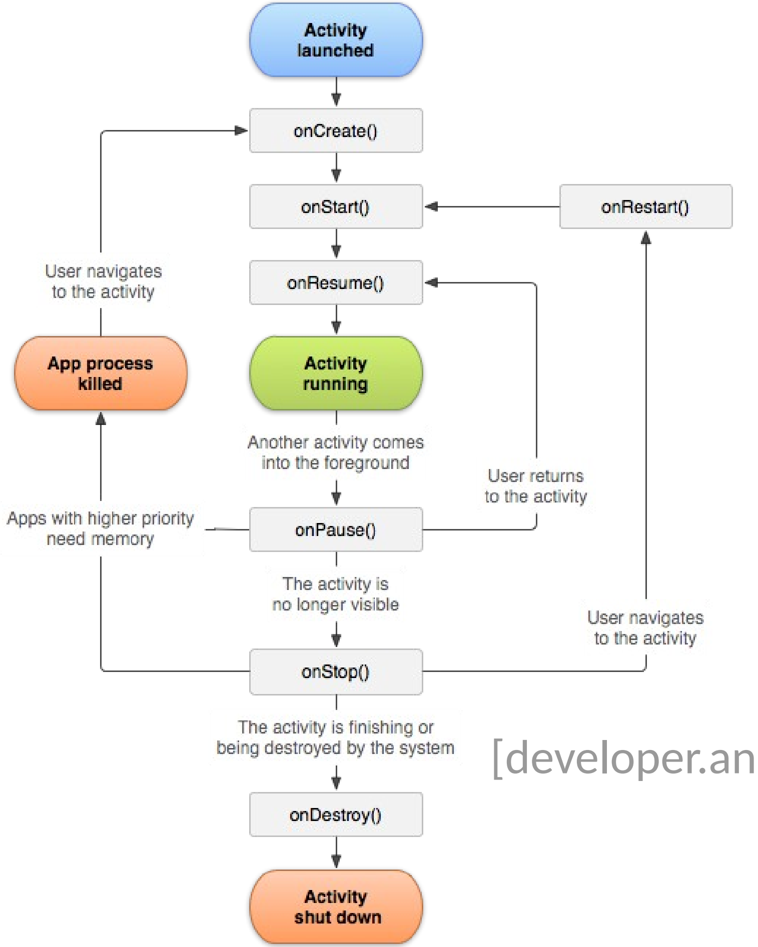
Remember that graphs are everywhere

Remember that graphs are everywhere

- Protocols and lifecycles



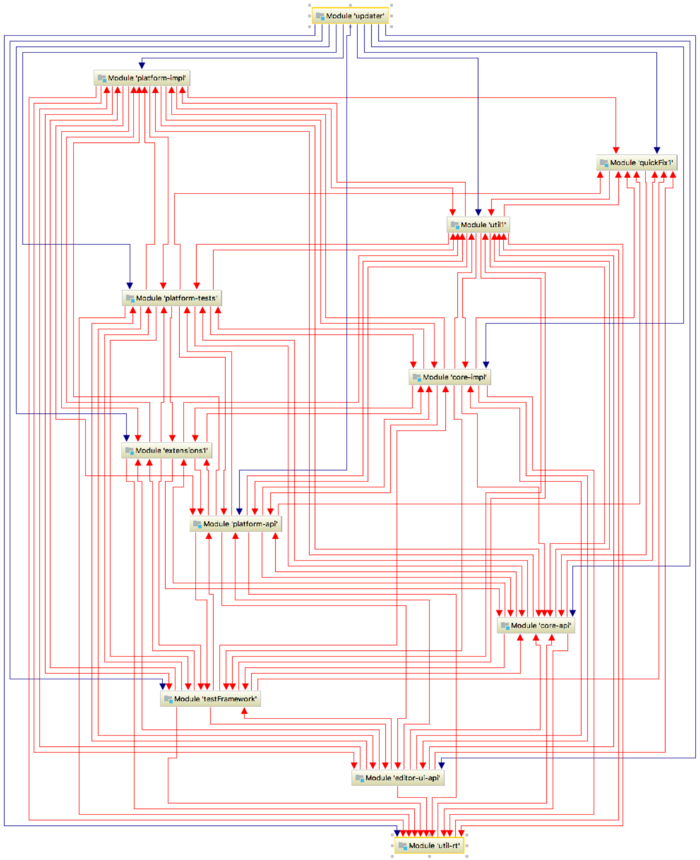
[developer.apple.com]



[developer.android.com]

Remember that graphs are everywhere

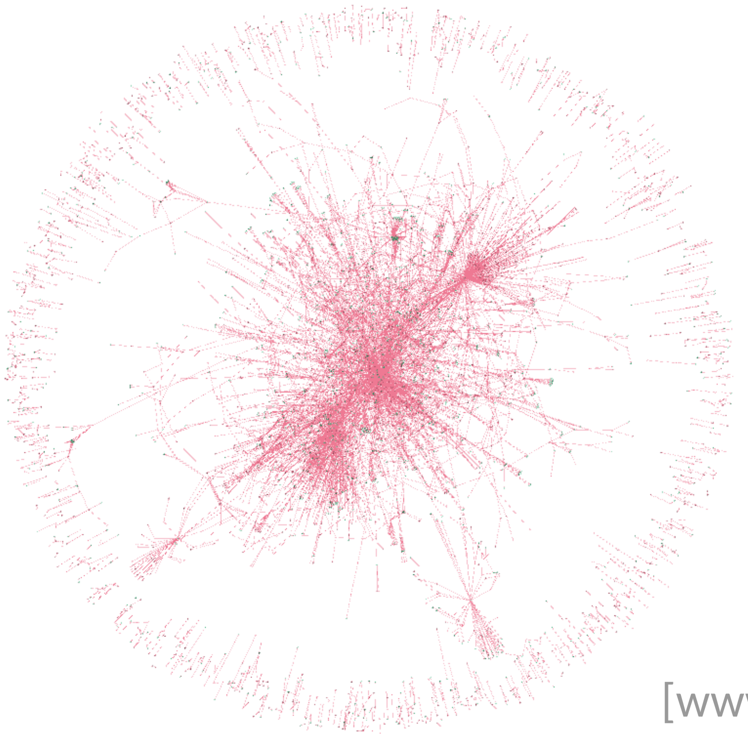
- Protocols and lifecycles
- Dependencies between components



[www.jetbrains.com]

Remember that graphs are everywhere

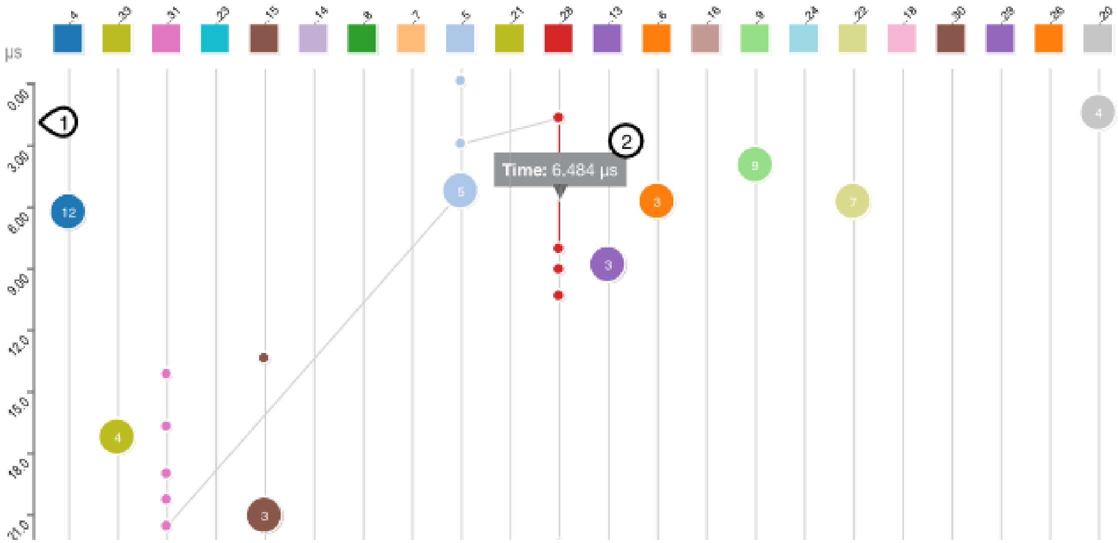
- Protocols and lifecycles
- Dependencies between components
- Callgraphs



[www.androguard.com]

Remember that graphs are everywhere

- Protocols and lifecycles
- Dependencies between components
- Callgraphs
- (Micro)Services and distributed systems



[Beschastnikh et al.]

Summary

- Graph coverage is a common basis for measuring test suite adequacy
  - Branch coverage is the most common “cost-effective” approach

Summary

- Graph coverage is a common basis for measuring test suite adequacy
  - Branch coverage is the most common “cost-effective” approach
- Path coverage criteria can provide deeper insight

Summary

- Graph coverage is a common basis for measuring test suite adequacy
  - Branch coverage is the most common “cost-effective” approach
- Path coverage criteria can provide deeper insight
  - Subtle logic interactions
  - Subtle loop behaviors

Summary

- Graph coverage is a common basis for measuring test suite adequacy
  - Branch coverage is the most common “cost-effective” approach
- Path coverage criteria can provide deeper insight
  - Subtle logic interactions
  - Subtle loop behaviors
  - But managing costs can require care

Summary

- Graph coverage is a common basis for measuring test suite adequacy
  - Branch coverage is the most common “cost-effective” approach
- Path coverage criteria can provide deeper insight
  - Subtle logic interactions
  - Subtle loop behaviors
  - But managing costs can require care
- Graphs. Are. Everywhere.