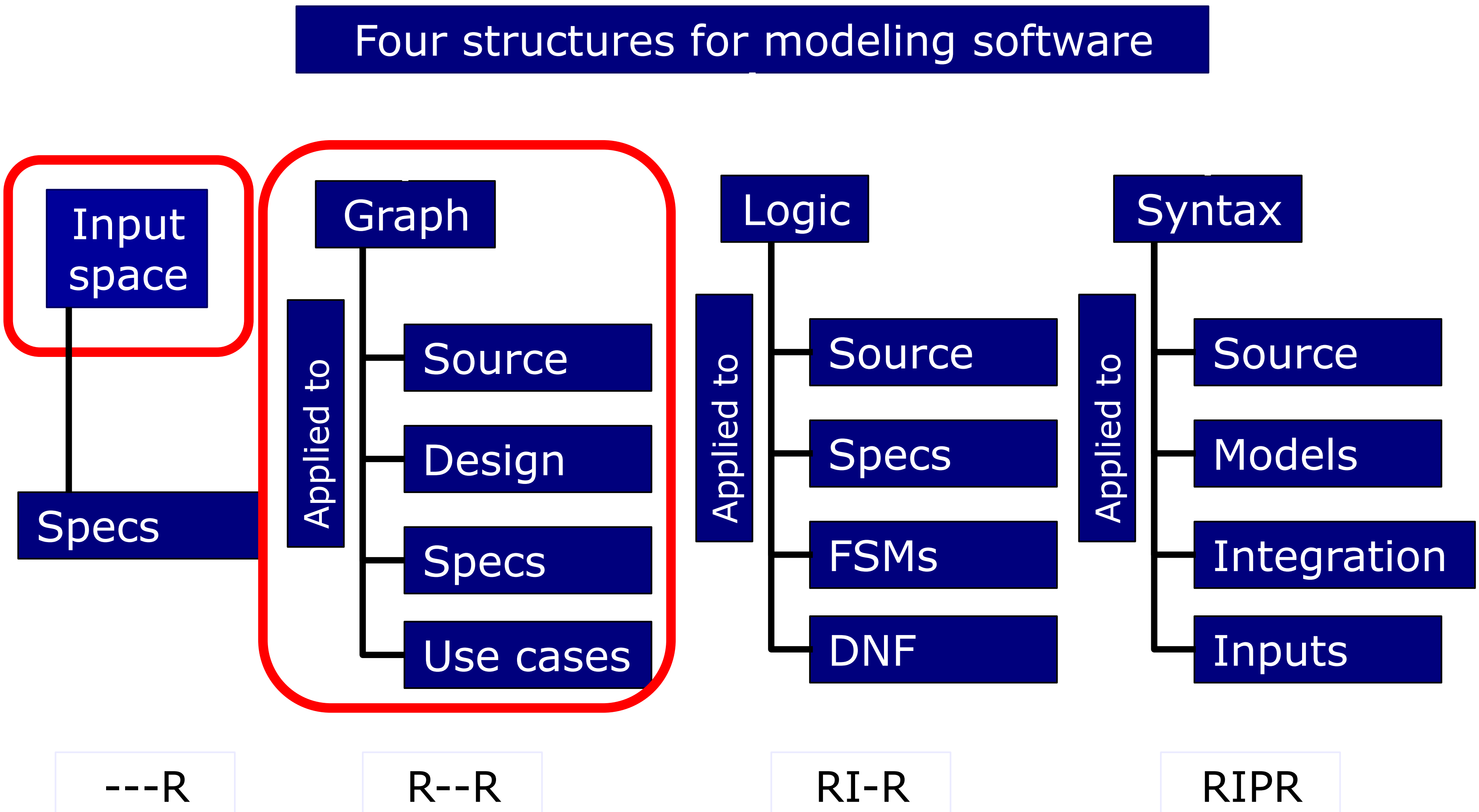
CSc 179 – Graph Coverage for Source Code

Credits:

AO – Ammann and Offutt, “Introduction to Software Testing,” Ch. 7

University of Virginia (CS 4501 / 6501)

Structures for Criteria-Based Testing



Overview

- Graph coverage criteria are widely used on source code
- Define graph, then apply coverage criterion
- Control flow graph (CFG): the most common graph for source code
- Node coverage: execute every statement
- Edge coverage: execute every branch
- Data flow coverage: augment the CFG with
  - defs: statements that assign values to variables
  - uses: statements that use variables

Control Flow Graph (CFG)

- Represent the control flow of a piece of source code
  - Nodes represent basic blocks
    - Basic blocks represent sequences of instructions / statements that always execute together in sequence
  - Edges represent control flow (branch) between basic blocks
    - Transfer of control
  - Initial nodes correspond to a method's entry points
  - Final nodes correspond to a method's exit points
    - `Return` or `throw` in Java
  - Decision nodes represent choices in control flow
    - **`if`** or **`switch-case`** blocks or **condition for loops** in Java
- Can be annotated with extra information such as branch predicates, defs, and uses

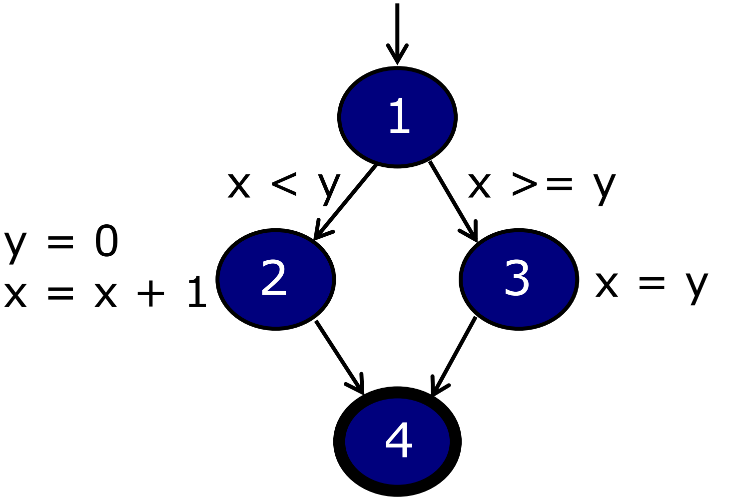
Example: CFG for *if-else*

```
if (x < y)
{
    y = 0;
    x = x+1;
}
else
{
    x = y;
}
```

- Basic blocks (nodes)
  - 1: if (x < y)
  - 2: y=0; x = x+1;
  - 3: x = y;
- Entry node
  - 1
- Decision nodes
  - 1
- Junction nodes
  - 4
- Exit nodes
  - 4

Control flow (edges)

1 → 2
1 → 3
2 → 4
3 → 4



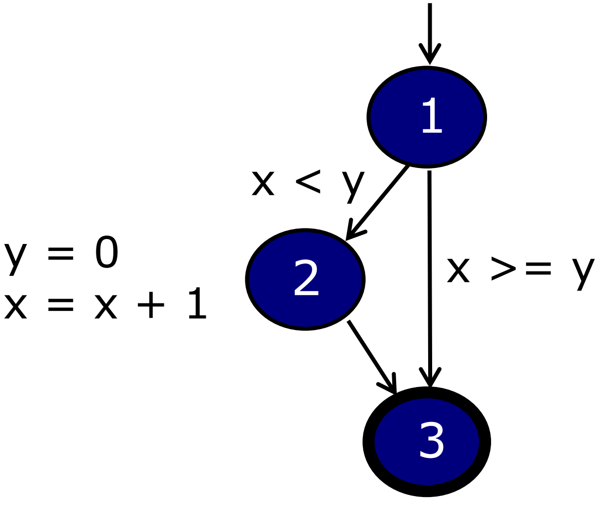
Example: CFG for *If* without *else*

```
if (x < y)
{
    y = 0;
    x = x+1;
}
```

- Basic blocks (nodes)
  1: if (x < y)
  2: y=0; x = x+1;
- Entry node
  1
- Decision nodes
  1
- Junction nodes
  3
- Exit nodes
  3

Control flow (edges)

1 → 2
1 → 3
2 → 3



Example: CFG for *If* with *return*

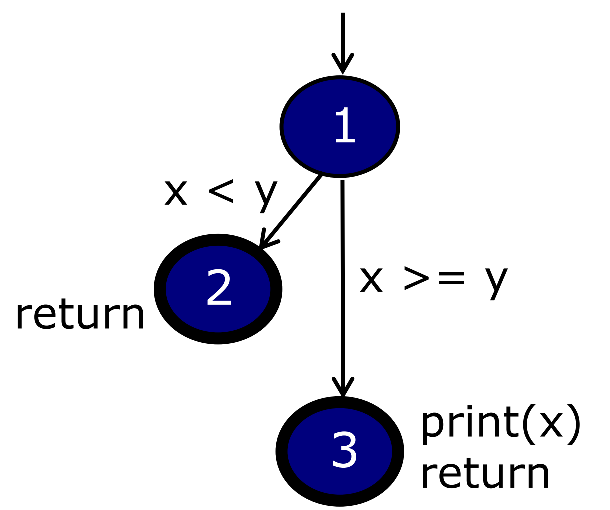
```
if (x < y)
{
    return;
}
print(x);
return;
```

- Basic blocks (nodes)
  - 1: if (x < y)
  - 2: return;
  - 3: print(x); return;
- Entry node
  - 1
- Decision nodes
  - 1
- Junction nodes
  - -
- Exit nodes
  - 2, 3

Control flow (edges)

1 → 2

1 → 3



Loops

- Loops require extra nodes ("dummy" node)
  - Not directly derived from program statements
- Looping structures: *while* loop, *for* loop, *do-while* loop
- Common mistake
  - Try to have the edge go to the entry node

Example: CFG for a *while* loop

```
x = 0;
while (x < y)
{
    y = f(x,y);
    x = x + 1;
}
```

- Basic blocks (nodes)
  - 1: x = 0;
  - 2: while(x < y)
  - 3: y = f(x,y); x = x+1;

Control flow (edges)

1 → 2

2 → 3

2 → 4

3 → 2

Entry node

1

Decision nodes

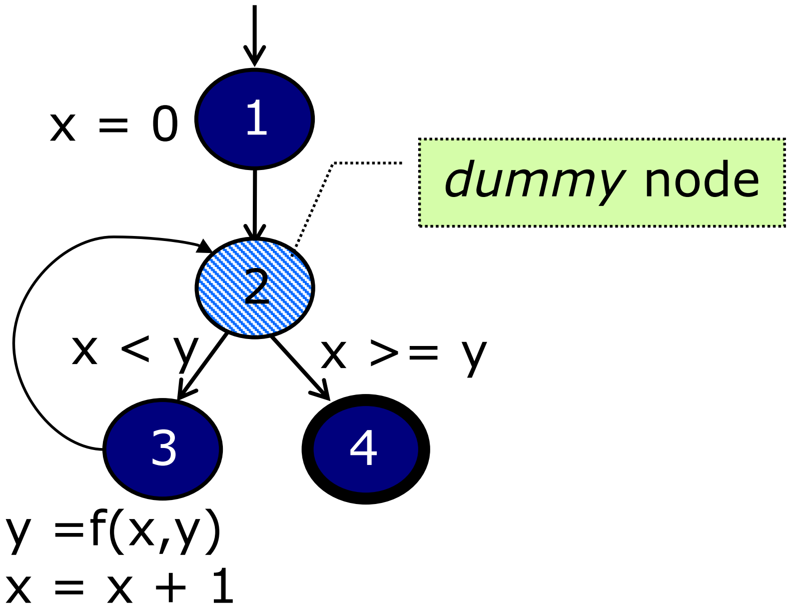
2

Junction nodes

-

Exit nodes

4



Example: CFG for a *for* loop

```
for (x=0; x<y; x++)
{
    y = f(x,y);
}
```

- Basic blocks (nodes)
  - 1: x = 0;
  - 2: x < y
  - 3: y = f(x,y);
  - 4: x++;

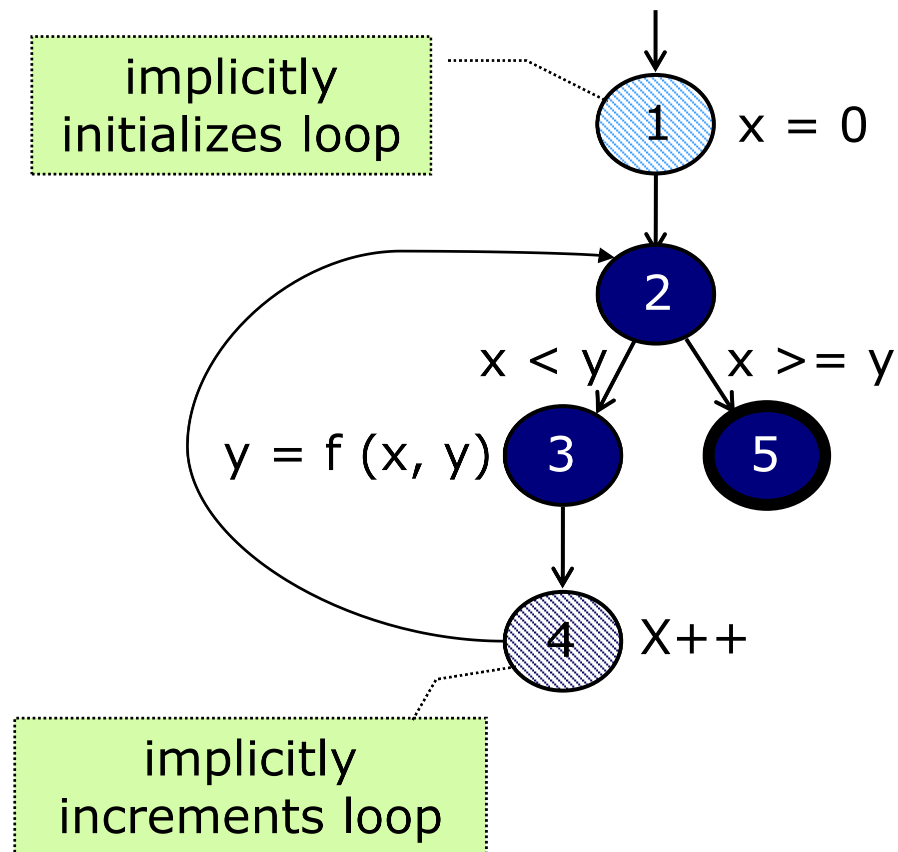
Entry node
1

Decision nodes
2

Junction nodes
-

Exit nodes
5

Control flow (edges)
1 → 2, 2 → 3, 2 → 5, 3 → 4, 4 → 2



Example: CFG for a *do-while* loop

```
x = 0;
do
{
      y = f(x,y);
      x = x + 1;
} while (x < y)
println(y);
```

- Basic blocks (nodes)
  - 1: x = 0; do
  - 2: y = f(x,y); x = x+1;
    while(x < y)
  - 3: print(y);

Control flow (edges)

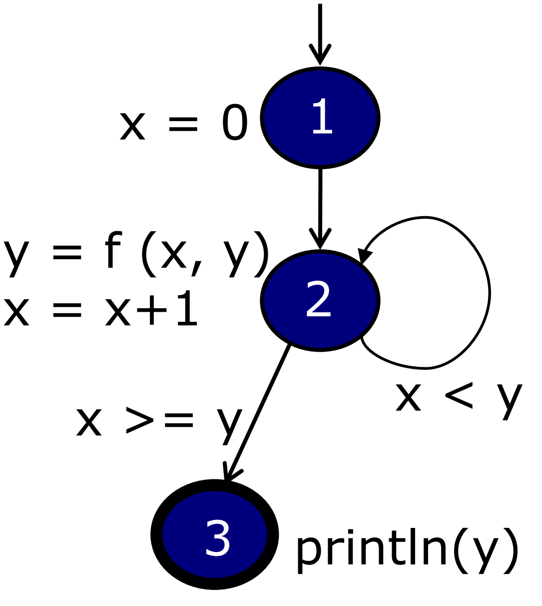
1 → 2
2 → 2
2 → 3

Entry node
  1

Decision nodes
  2

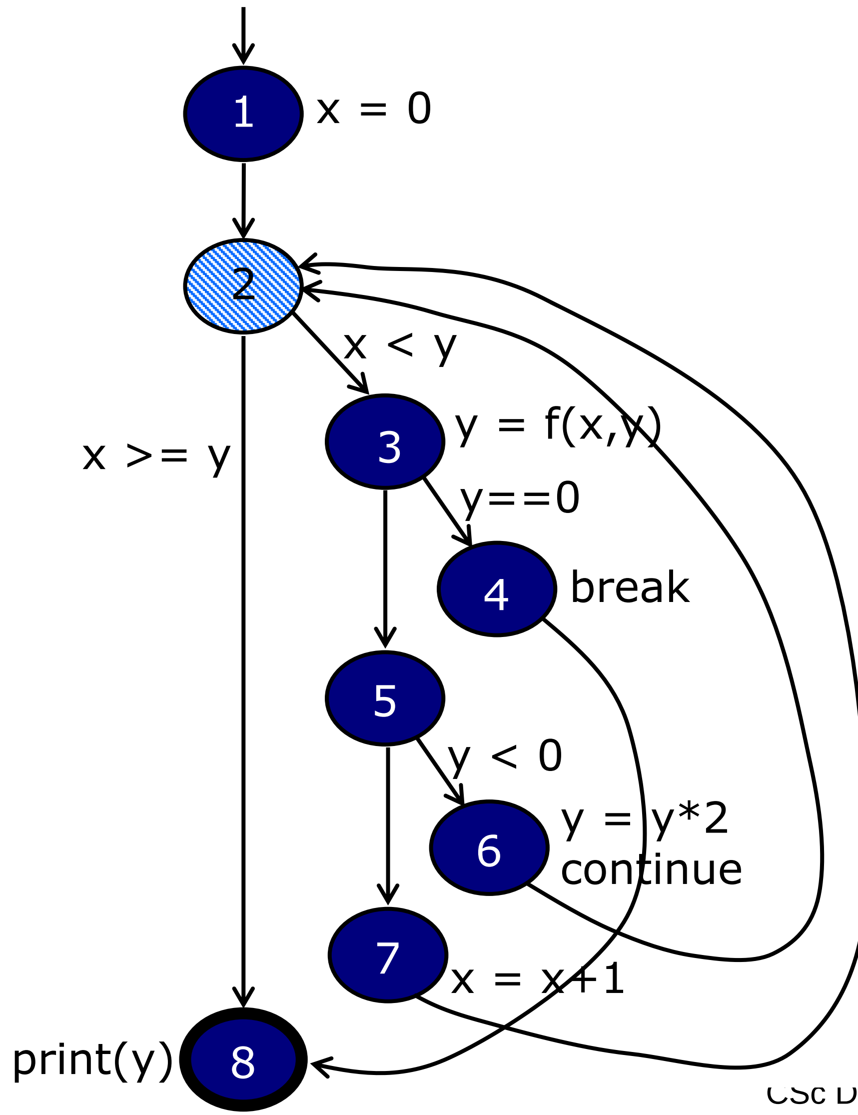
Junction nodes
  -

Exit nodes
  3



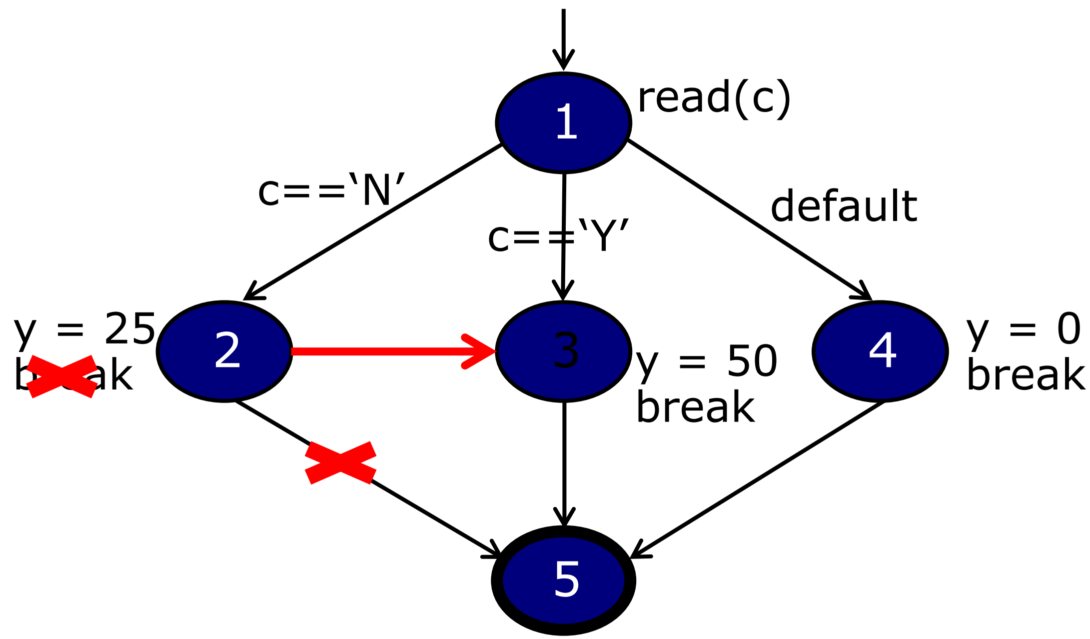
Example: CFG for a loop with *break* and *continue*

```
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(x);
```



Example: CFG for (*switch*) *case*

```
read(c);
switch(c)
{
case 'N':
      y = 25;
      break;
case 'Y':
      y = 50;
      break;
default:
      y = 0;
      break;
}
```

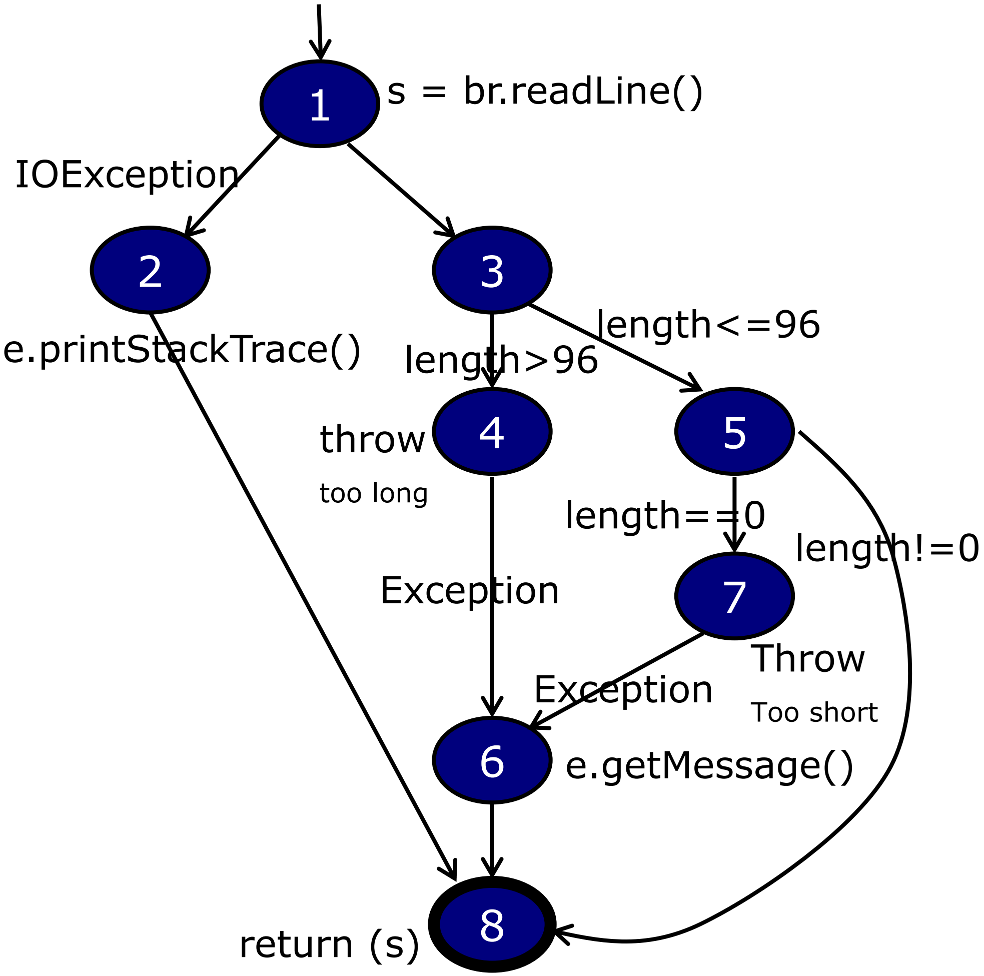


Cases without break?

- Fall through to the next case

Example: CFG for Exceptions (*try-catch*)

```
try
{
    s = br.readLine();
    if (s.length() > 96)
        throw new Exception("too long");
    if (s.length() ==0)
        throw new Exception("too short");
} catch (IOException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.getMessage();
}
return(s);
```



Exercise

```
public static int numberOccurrences(char[] v, char c)
{
    if (v == null)
        throw new NullPointerException();
    int n = 0;
    for (int i=0; i<v.length; i++)
    {
        if (v[i] == c)
            n++;
    }
    return n;
}
```

Entry node

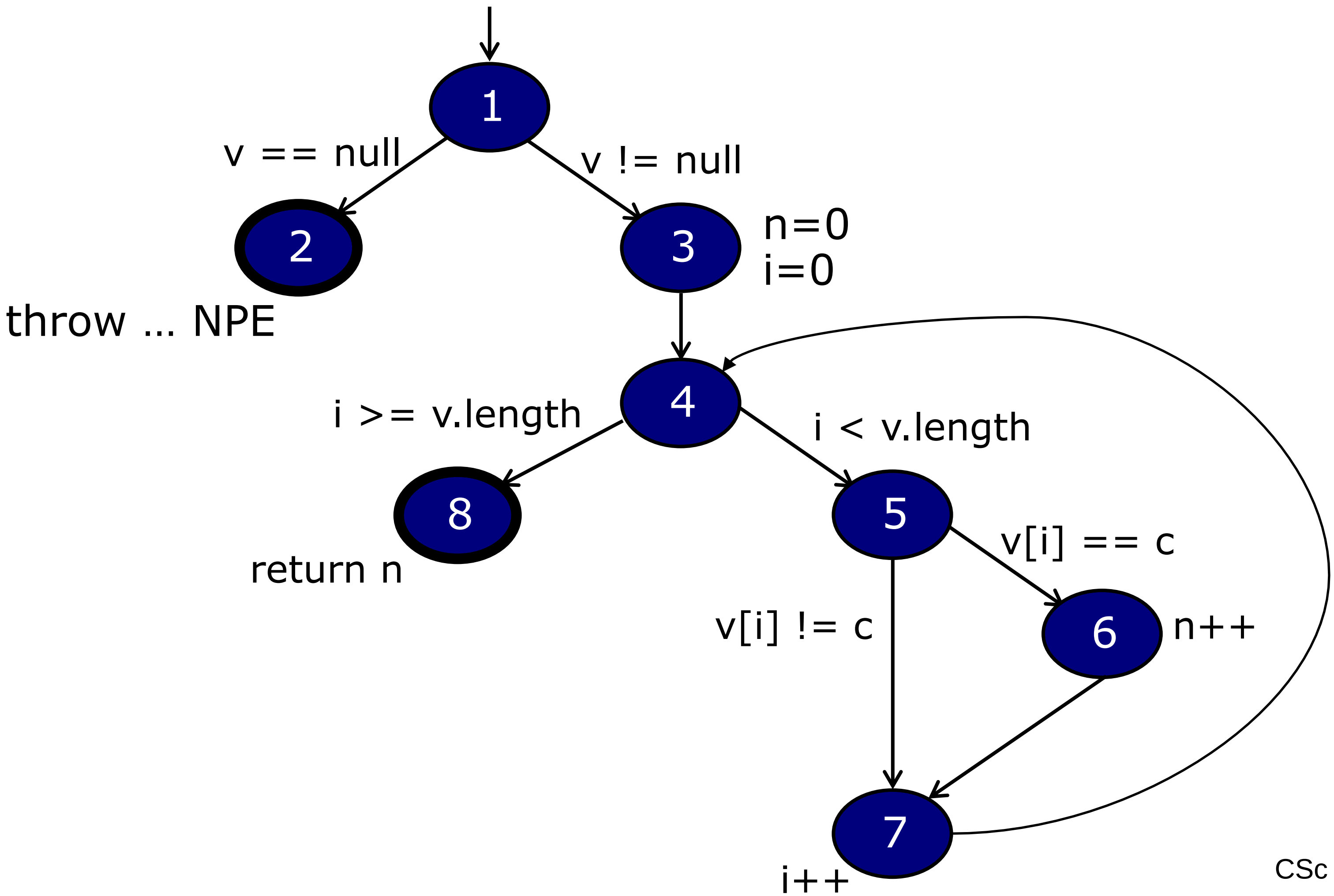
1

Exit nodes

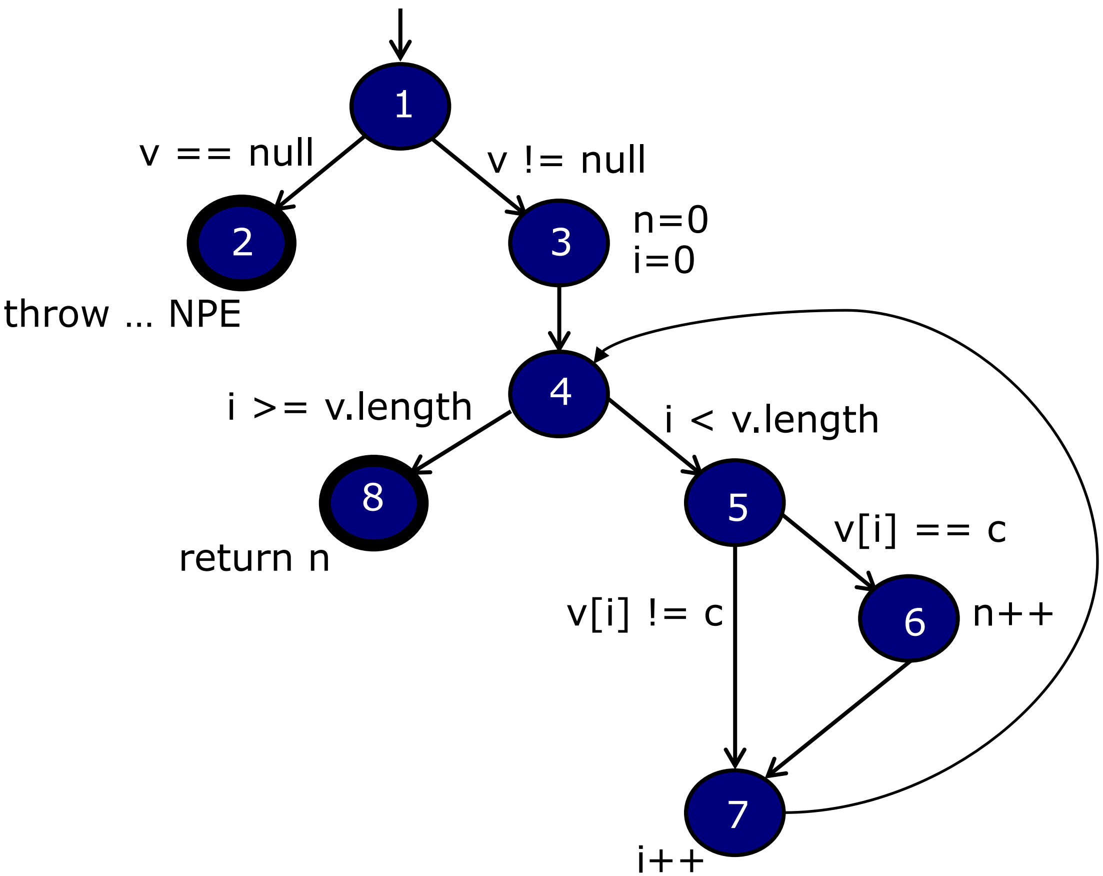
2, 8

- Basic blocks (nodes)
  - 1: if (v == null)
  - 2: throw .. NPE;
  - 3: n=0; i=0;
  - 4: i < v.length;
  - 5: if (v[i] == c)
  - 6: n++;
  - 7: i++;
  - 8: return n;
- Control flow (edges)
  - 1→2, 1→3
  - 3→4
  - 4→5, 4→8
  - 5→6, 5→7
  - 6→7
  - 7→4

CFG for *numberOccurrences()*



Applying Node Coverage (NC)



Test requirements

TR = {1,2,3,4,5,6,7,8}

Test paths

t1 = [1,2]

t2 = [1,3,4,5,6,7,4,8]

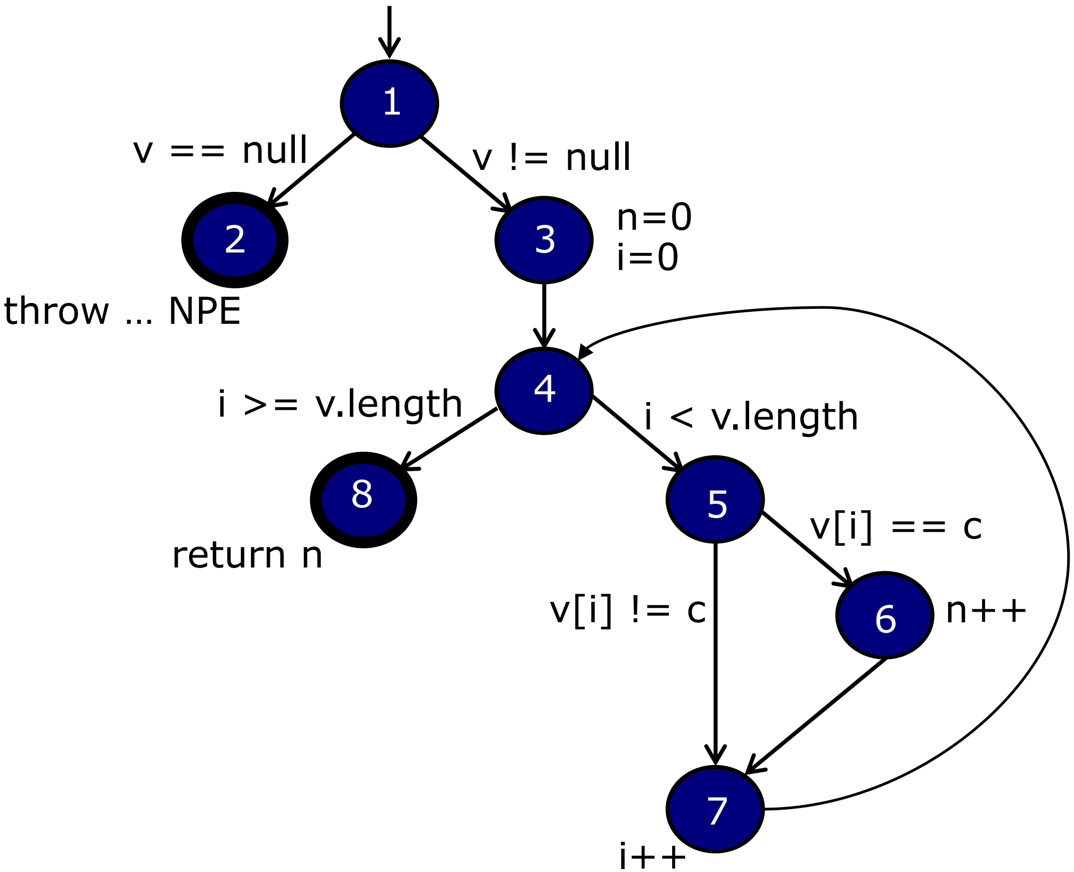
NC satisfied by {t1, t2}

Test case values (v,c)

t1 = (null, ‘a’), expected NPE

t2 = ({‘a’}, ‘a’), expected 1

Applying Edge Coverage (EC)



Test requirements

TR = {(1,2), (1,3), (3,4), (4,5), (4,8), (5,6), (5,7), (6,7), (7,4)}

Test paths

t1 = [1,2]

t2 = [1,3,4,5,6,7,4,8]

t3 = [1,3,4,5,7,4,8]

EC satisfied by {t1, t2, t3}

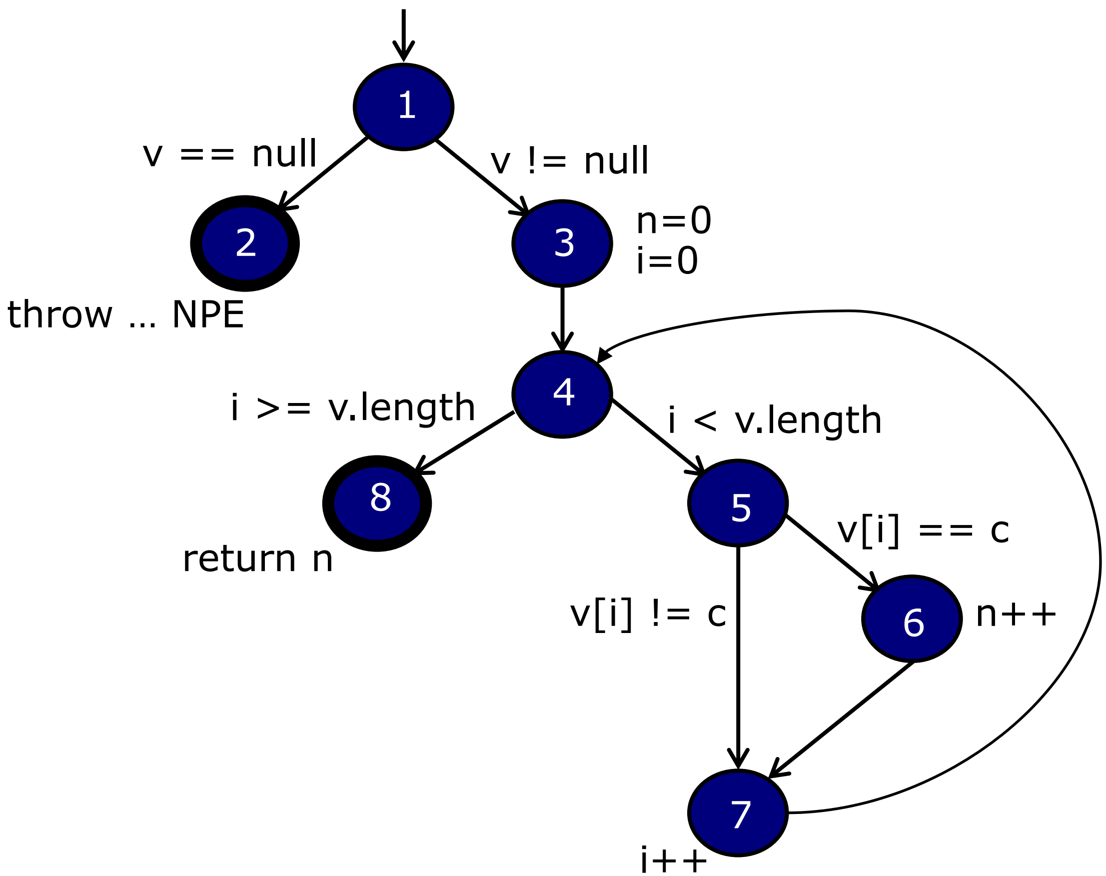
Test case values (v,c)

t1 = (null, ‘a’), expected NPE

t2 = ({‘a’}, ‘a’), expected 1

t3 = ({‘x’}, ‘a’), expected 0

Applying Edge-Pair Coverage (EPC)



Test requirements

TR = {(1,2), (1,3,4), (3,4,8), (3,4,5), (4,5,6), (4,5,7), (5,6,7), (5,7,4), (6,7,4), (7,4,5), (7,4,8)}

Test paths

t1 = [1,2]

t2 = [1,3,4,8]

t3 = [1,3,4,5,7,4,5,6,7,4,8]

EPC satisfied by {t1, t2, t3}

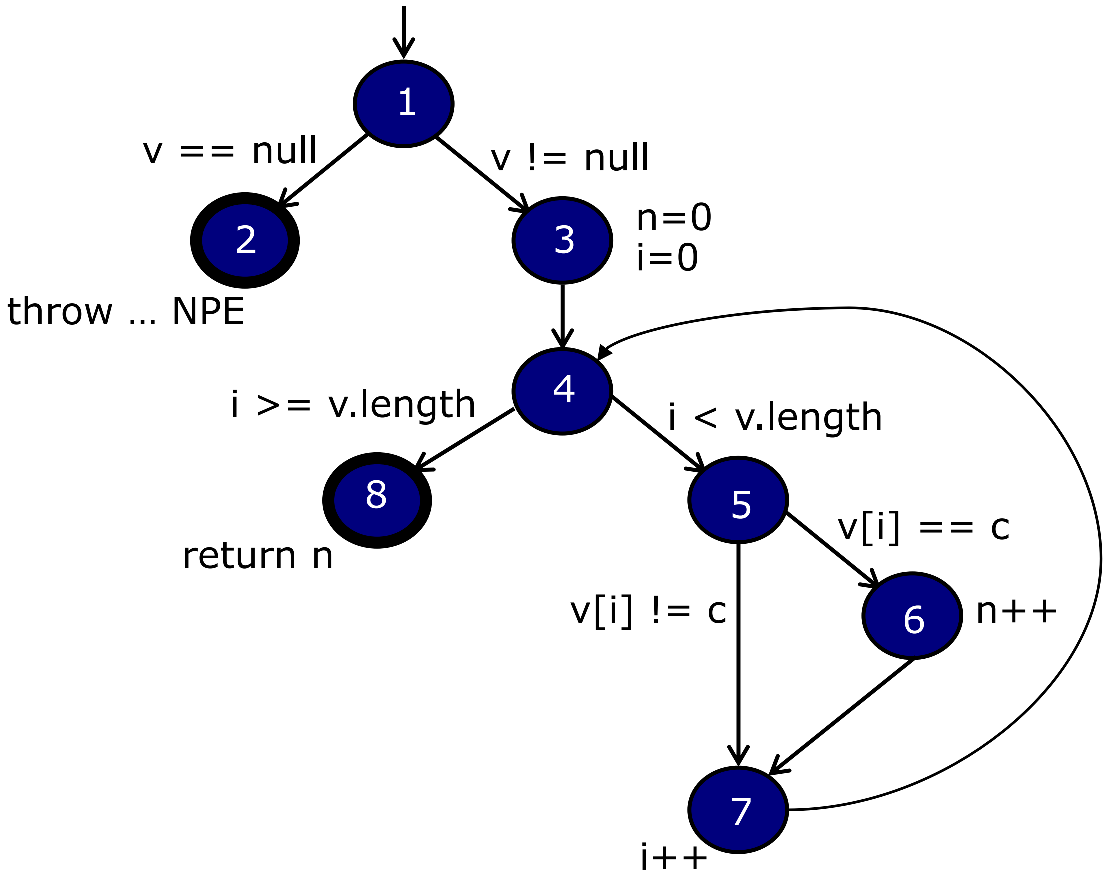
Test case values (v,c)

t1 = (null, ‘a’), expected NPE

t2 = ({}, ‘a’), expected 0

t3 = ({‘x’,‘a’}, ‘a’), expected 1

Applying Prime Path Coverage



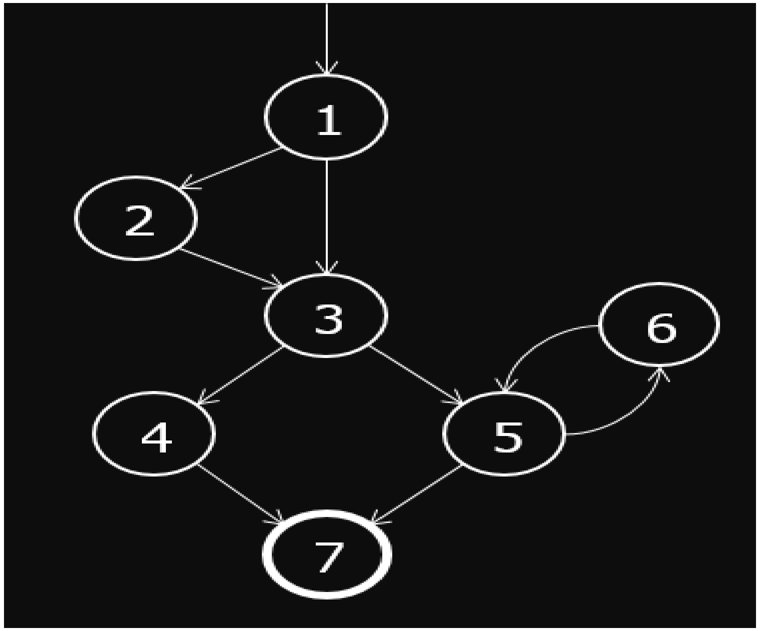
Deriving prime paths

- Enumerate all simple paths of length 0, 1, 2, 3, … until no more simple paths are found
- Pick the prime paths among all derived simple paths
- Notes:(1) Simple path: Path from node n_i to n_j that is **no internal loops.** (2) Prime Path is simple path that is **<u>not</u>** **<u>subpath</u>** of any other simple path

Recap: Simple Paths

Path from node n_i to n_j that is **no internal loops**

- A path from ni to nj is simple if no node appears more than once in the path (first and last nodes may be identical)
- A loop is a simple path



List simple paths: 31 simple paths

[1,2,3,4,7], [1,2,3,5,7], [1,2,3,5,6],

[1,2,3,4], [1,2,3,5],

[1,3,4,7], [1,3,5,7], [1,3,5,6],

[2,3,4,7], [2,3,5,7], [2,3,5,6],

[1,2,3], [1,3,4], [1,3,5],

[2,3,4], [2,3,5],

[3,4,7], [3,5,7], [3,5,6],

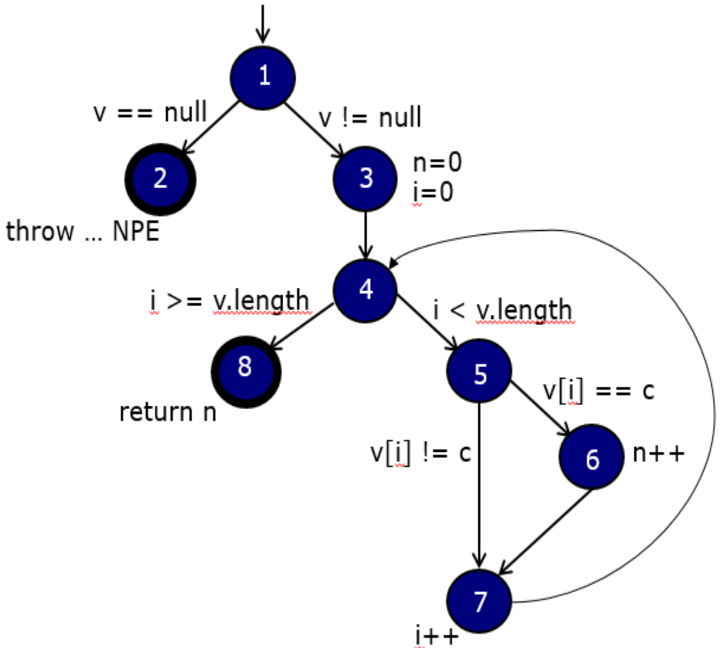
[5,6,5],

[6,5,6], [6,5,7],

[1,2], [1,3], [2,3], [3,4], [3,5],
[4,7], [5,7], [5,6], [6,5]

Subpaths of other simple paths → avoid these

Applying Prime Path Coverage



[1]
[2]!
[3]
[4]
[5]
[6]
[7]
[8]!

[1,2]!
[1,3]
[3,4]
[4,5]
[4,8]!
[5,6]
[5,7]
[6,7]
[7,4]

[1,3,4]
[3,4,5]
[3,4,8]!
[4,5,6]
[4,5,7]
[5,6,7]
[5,7,4]
[6,7,4]
[7,4,5]
[7,4,8]!

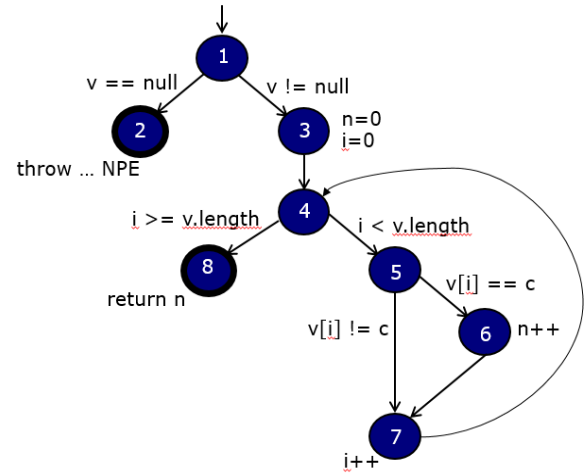
[1,3,4,5]
[1,3,4,8]!
[3,4,5,6]
[3,4,5,7]!
[4,5,6,7]
[4,5,7,4]*
[5,6,7,4]
[5,7,4,5]*
[5,7,4,8]!
[6,7,4,5]
[6,7,4,8]!
[7,4,5,6]
[7,4,5,7]*

[1,3,4,5,6]
[1,3,4,5,7]!
[3,4,5,6,7]!
[4,5,6,7,4]*
[5,6,7,4,5]*
[5,6,7,4,8]!
[6,7,4,5,6]*
[7,4,5,6,7]*

! – cannot be extended
* – is a cycle

[1,3,4,5,6,7]!

Applying Prime Path Coverage



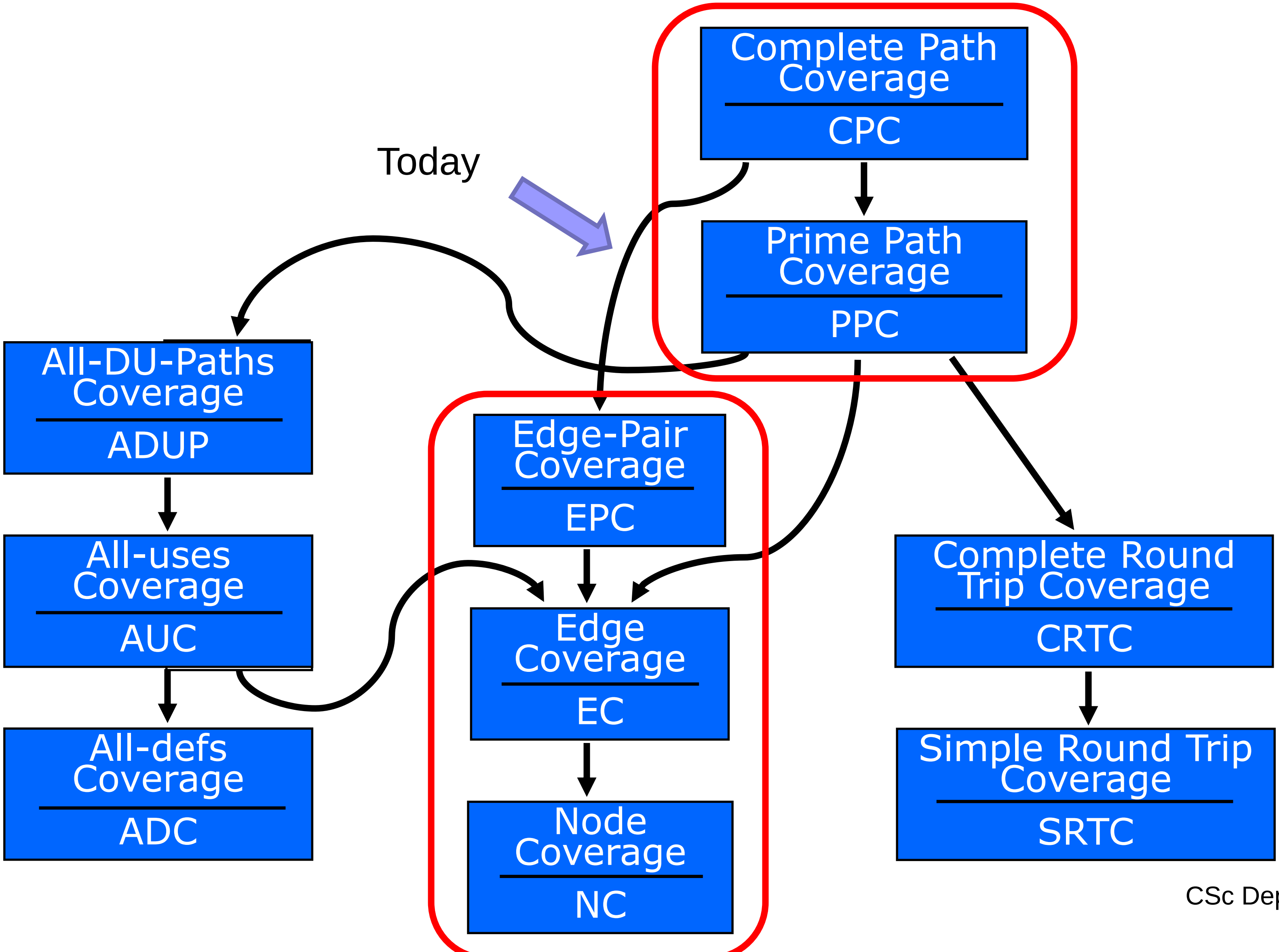
Prime path	Covered by
[1,2]	t1
[1,3,4,8]	t2
[1,3,4,5,6,7]	t4, t5
[1,3,4,5,7]	t3
[4,5,7,4]	t3, t4
[4,5,6,7,4]	t3, t4, t5
[5,7,4,5]	t3
[5,7,4,8]	t4
[5,6,7,4,5]	t4, t5
[5,6,7,4,8]	t3, t5
[6,7,4,5,6]	t5
[7,4,5,7]	t4
[7,4,5,6,7]	t3, t5

	Test paths	Test case values (v,c)	Expected values
t1	[**1**,**2**]	(null, ‘a’)	NPE
t2	[**1**,3,4,**8**]	({}, ‘a’)	0
t3	[**1**,3,4,5,7,4,5,6,7,4,**8**]	({‘x’, ‘a’}, ‘a’)	1
t4	[**1**,3,4,5,6,7,4,5,7,4,**8**]	({‘a’, ‘x’}, ‘a’)	1
t5	[**1**,3,4,5,6,7,4,5,6,7,4,**8**]	({‘a’, ‘a’}, ‘a’)	2

PPC satisfied by {t1, t2, t3, t4, t5}

PPC: TR contains each prime path in graph G

Recap:Graph Coverage Criteria Subsumption



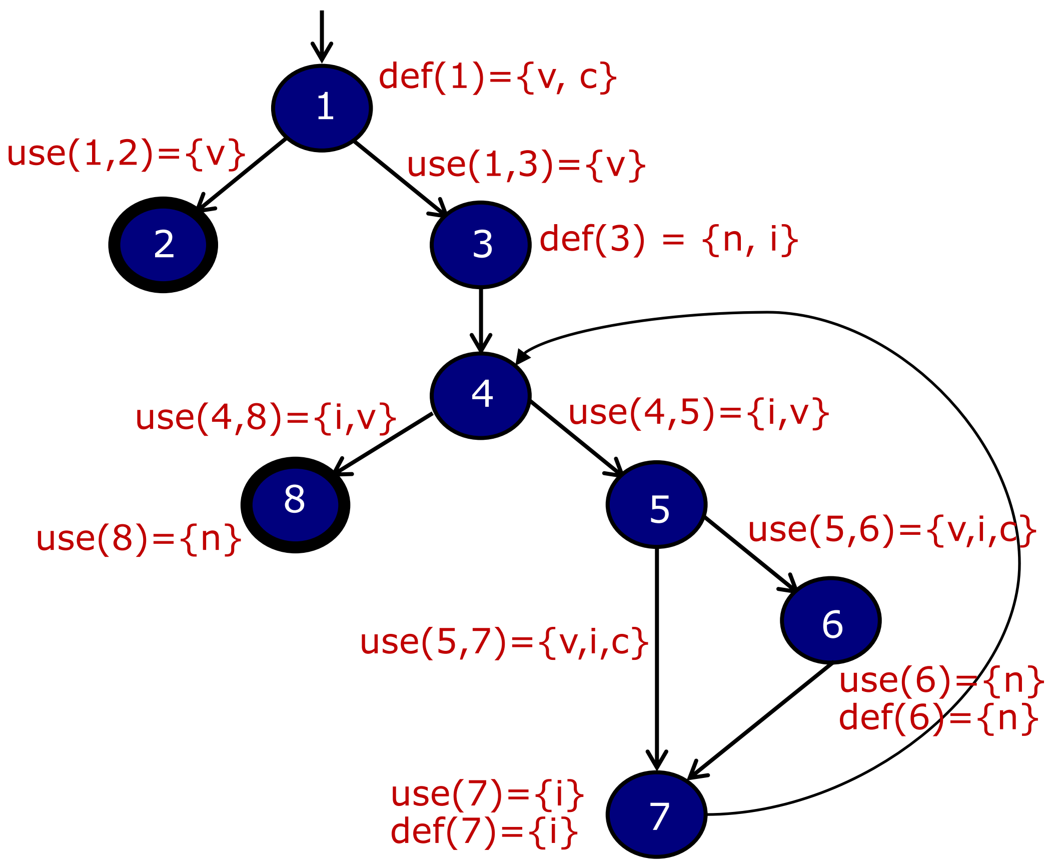
Recap: Handling Loops in Graphs

Attempts to deal with loops:

- 1970s: Execute cycles once ([5, 6, 5] in previous example)
- 1980s: Execute each loop, exactly once
- 1990s: Execute loops 0 times, once, more than once
- 2000s: Prime paths (touring, sidetrips, and detours)

Applying Data Flow Coverage

v and c are forwarded parameters



Deriving test requirements

- List all du-pairs
- Based on du-pairs, derive du-paths
- Must be def-clear paths
- All Defs Coverage (ADC)
  - For each def, at least one use must be reached
- All Uses Coverage (AUC)
  - For each def, all uses must be reached
- All DU-Paths Coverage (ADUPC)
  - For each def-use pair, all paths between defs and uses must be covered

DU-Pairs → DU-Paths

DU Pairs	
[1, (1, 2)]	
[1, (1, 3)]	
[1, (4, 8)]	
[1, (4, 5)]	
[1, (5, 6)]	
[1, (5, 7)]	Variable v
[1, (5, 7)]	Variable c
[1, (5, 6)]	
[3,8]	
[3,6]	Variable n
[6,8]	
[6,6]	
[3,7]	
[7,7]	
[3, (4, 8)]	
[3, (4, 5)]	
[3, (5, 6)]	Variable i
[3, (5, 7)]	
[7, (4, 8)]	
[7, (4, 5)]	
[7, (5, 6)]	
[7, (5, 7)]	

defs <u>after</u> uses, these are valid DU pairs

DU Paths	
[1,3]	
[1,2]	
[1,3,4,8]	
[1,3,4,5]	
[1,3,4,5,7]	
[1,3,4,5,6]	Variable v
[1,3,4,5,6]	Variable c
[1,3,4,5,7]	
[3,4,8]	
[3,4,5,6]	Variable n
[6,7,4,8]	
[6,7,4,5,6]	
[3,4,5]	
[3,4,8]	
[3,4,5,6]	
[3,4,5,7]	
[3,4,5,6,7]	Variable i
[7,4,5]	
[7,4,8]	
[7,4,5,6]	
[7,4,5,7]	
[7,4,5,6,7]	

ADC: DU-Paths → Test Paths

DU Paths	
[1,3] [1,2] [1,3,4,8] [1,3,4,5] [1,3,4,5,7] [1,3,4,5,6]	Variable v
[1,3,4,5,6] [1,3,4,5,7]	Variable c
[3,4,8] [3,4,5,6] [6,7,4,8] [6,7,4,5,6]	Variable n
[3,4,5] [3,4,8] [3,4,5,6] [3,4,5,7] [3,4,5,6,7] [7,4,5] [7,4,8] [7,4,5,6] [7,4,5,7] [7,4,5,6,7]	Variable i

Test paths that satisfy All Defs Coverage

Variable	All Def Coverage
v	[1,3,4,8]
c	[1,3,4,5,6,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,5,7,4,5,7,4,8]

AUC: DU-Paths → Test Paths

DU Paths	
[1,3] [1,2] [1,3,4,8] [1,3,4,5] [1,3,4,5,7] [1,3,4,5,6]	Variable v
[1,3,4,5,6] [1,3,4,5,7]	Variable c
[3,4,8] [3,4,5,6] [6,7,4,8] [6,7,4,5,6]	Variable n
[3,4,5] [3,4,8] [3,4,5,6] [3,4,5,7] [3,4,5,6,7] [7,4,5] [7,4,8] [7,4,5,6] [7,4,5,7] [7,4,5,6,7]	Variable i

Test paths that satisfy All Uses Coverage

Variable	All Use Coverage
v	[1,2] [1,3,4,8] [1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
c	[1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,6,7,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,5,7,4,5,7,4,8] [1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,7,4,8] [1,3,4,5,7,4,5,6,7,4,8]

ADUPC: DU-Paths → Test Paths

DU Paths	
[1,3] [1,2] [1,3,4,8] [1,3,4,5] [1,3,4,5,7] [1,3,4,5,6]	Variable v
[1,3,4,5,6] [1,3,4,5,7]	Variable c
[3,4,8] [3,4,5,6] [6,7,4,8] [6,7,4,5,6]	Variable n
[3,4,5] [3,4,8] [3,4,5,6] [3,4,5,7] [3,4,5,6,7] [7,4,5] [7,4,8] [7,4,5,6] [7,4,5,7] [7,4,5,6,7]	Variable i

Test paths that satisfy All DU-Paths Coverage

Variable	All DU Path Coverage
v	[1,3,4,8] [1,2] [1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
c	[1,3,4,5,6,7,4,8] [1,3,4,5,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,6,7,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,7,4,5,7,4,8] [1,3,4,5,7,4,8] [1,3,4,5,7,4,5,6,7,4,8]

Summary

- A common application of graph coverage criteria is to program source – control flow graph (CFG)
- Applying graph coverage criteria to control flow graphs is relatively straightforward
- A few decisions must be made to translate control structures into the graph
- We use basic blocks when assigning program statements to nodes while some tools assign each statement to a unique node.
  - Coverage is the same, although the bookkeeping will differ

McCabe's Cyclomatic Complexity & Code Coverage

McCabe’s Cyclomatic Complexity

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

Source: https://en.wikipedia.org/wiki/Cyclomatic_complexity

McCabe's Cyclomatic Complexity

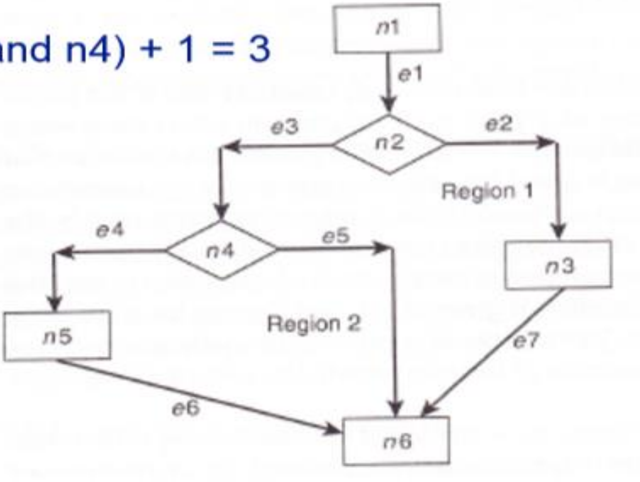
- Basic idea: program quality is directly related to the complexity of the control flow (branching)

- Computed from a control flow diagram
  - Cyclomatic complexity = E - N + 2p
  - E = number of edges of the graph
  - N = number of nodes of the graph
  - p = number of connected components (usually 1)

- Alternate computations:
  - number of binary decision + 1
  - number of closed regions +1

McCabe's Cyclomatic Complexity Example

- Using the different computations:
  - 7 edges - 6 nodes + 2*1 = 3
  - 2 regions + 1 = 3
  - 2 binary decisions (n2 and n4) + 1 = 3



McCabe's Cyclomatic Complexity

• What does the number mean?

• Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand

• It's the maximum number of linearly independent paths through the flow diagram - used to determine the number of test cases needed to cover each path through the system

• The higher the number, the more risk exists (and more testing is needed)

- 1-10 is considered low risk
- greater than 50 is considered high risk

Structural Analysis … Providing Actionable Metrics

The higher the complexity the more bugs. The more bugs the more security flaws

Cyclomatic Complexity & Reliability Risk

- **1 – 10** **Simple procedure, little risk**
- **11- 20** **More Complex, moderate risk**
- **21 – 50** **Complex , high risk**
- **>50** **Untestable, VERY HIGH RISK**

Cyclomatic Complexity & Bad Fix Probability

- **1 – 10** **5%**
- **20 –30** **20%**
- **> 50** **40%**
- **Approaching 100** **60%**

Essential Complexity (Unstructuredness) &
Maintainability (future Reliability) Risk

- **1 – 4** **Structured, little risk**
- **> 4** **Unstructured, High Risk**

Code Coverage

- Code coverage is a measure to describe the degree to which the source code of a program is executed when a particular test suite runs

- Code Coverage is classified as a White box testing

Benefits

- Identify untested part of codebase
- Identify testing gaps or missing tests
- Identify the redundant/dead Code

Types of Coverage

Method coverage

```
public void method(a, b, c){
    if( a && b ){
        call();
    }
    call();
    if( c || call()){
        call();
    }
}
```

Branch coverage

Statement coverage

Path coverage

Code Coverage

- ECLEmmea Code Coverage offers covered instruction and missed instructions

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CSC179A2Test	56.6 %	5,252	4,030	9,282
src	56.6 %	5,252	4,030	9,282
(default package)	56.6 %	5,252	4,030	9,282
Main.java	0.0 %	0	3,115	3,115
TestTriangleGetPerimeter.java	75.9 %	466	148	614
TestTriangleGetSideLengths.java	79.1 %	559	148	707
TestTriangleSetSideLengths.java	80.1 %	595	148	743
TestTriangleIsScalene.java	79.2 %	506	133	639
TestTriangleGetArea.java	78.8 %	464	125	589
TestTriangleIsImpossible.java	90.2 %	495	54	549
TestTriangleClassify.java	93.8 %	587	39	626
TestTriangleIsEquilateral.java	93.2 %	535	39	574
TestTriangleIsIsosceles.java	93.5 %	560	39	599
TestTriangleIsRightAngled.java	92.6 %	485	39	524
TriangleTestSuite.java	0.0 %	0	3	3