Software Engineering

Software Complexity

Topics

- Measuring Software Complexity
- Cyclomatic Complexity

Measuring Software Complexity

- Software complexity is difficult to operationalize complexity so that it can be measured
- Computational complexity measure big O (or big Oh), O(n)
  - Measures software complexity from the *machine's viewpoint* in terms of how the size of the input data affects an algorithm's usage of computational resources (usually running time or memory)
- Complexity measure in software engineering should measure complexity from the *viewpoint of human developers*
  - Computer time is cheap; human time is expensive

Desirable Properties of Complexity Metrics

- **Monotonicity**: adding responsibilities to a module cannot decrease its complexity
    - If a responsibility is added to a module, the modified module will exhibit a complexity value that is the same as or higher than the complexity value of the original module
- **Ordering** ("representation condition" of measurement theory):
    - Metric produces the same ordering of values as intuition would
    - Cognitively more difficult should be measured as greater complexity
- **Discriminative power** (sensitivity): modifying responsibilities should change the complexity
  - Discriminability is expected to increase as:
    - 1) the number of distinct complexity values increases and
    - 2) the number of classes with equal complexity values decreases
- **Normalization**: allows for easy comparison of the complexity of different classes

Cyclomatic Complexity

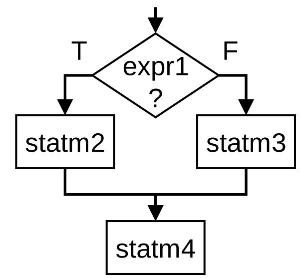
- Invented by Thomas McCabe (1974) to measure the complexity of a program's conditional logic
  - Counts the number of decisions in the program, under the assumption that decisions are difficult for people
  - Makes assumptions about decision-counting rules and linear dependence of the total count to complexity
- Cyclomatic complexity of graph G equals #edges - #nodes + 2
  - $V(G) = e – n + 2$
- Also corresponds to the number of linearly independent paths in a program (described later)

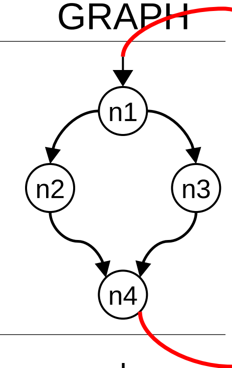
Converting Code to Graph

CODE | **FLOWCHART** | **GRAPH**

(a)

```
if expression1 then
    statement2
else
    statement3
end if
statement4
```





For a strongly connected graph:

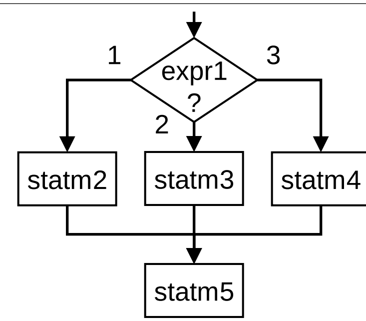
Create a ***virtual edge***

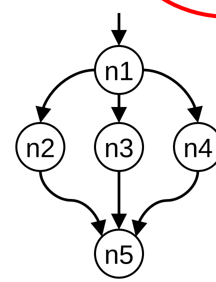
to connect the END node

to the BEGIN node

(b)

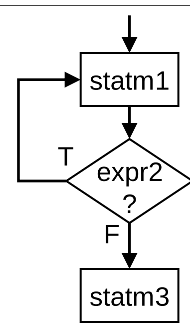
```
switch expr1
    case 1:
        statement2
    case 2:
        statm3
    case 3:
        statm4
end switch
statm5
```

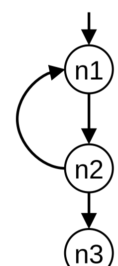




(c)

```
do
    statement1
  while expr2
end do
statement3
```





Paths in Graphs (1)

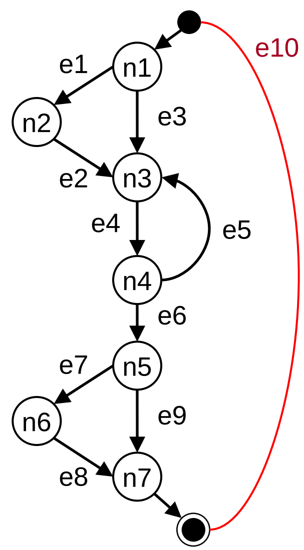
- *A* graph is **strongly connected** if for any two nodes *x*, *y* there is a path from *x* to *y* and vice versa
- *A* **path** is represented as an *n*-element vector where *n* is the number of edges
  <□, □, ..., □>
- The *i*-th position in the vector is the number of occurrences of edge *i* in the path

Example Paths

```
if expression1 then
     statement2
end if

do
     statement3
  while expr4
end do

if expression5 then
     statement6
end if
statement7
```



Paths:	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1 = e1, e2, e4, e6, e7, e8	1	1	0	1	0	1	1	1	0	0
P2 = e1, e2, e4, e5, e4, e6, e7, e8	1	1	0	2	1	1	1	1	0	0
P3 = e3, e4, e6, e7, e8, e10	0	0	1	1	0	1	1	1	0	1
P4 = e6, e7, e8, e10, e3, e4	0	0	1	1	0	1	1	1	0	1
P5 = e1, e2, e4, e6, e9, e10	1	1	0	1	0	1	0	0	1	1
P6 = e4, e5	0	0	0	1	1	0	0	0	0	0
P7 = e3, e4, e6, e9, e10	0	0	1	1	0	1	0	0	1	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1	1	0	2	1	1	0	0	1	1

Paths P3 and P4 are the same, but with different start and endpoints

NOTE: A path does not need to start in node n1 and does not need to begin and end at the same node.
E.g.,

- Path P4 starts (and ends) at node n4
- Path P1 starts at node n1 and ends at node n7

Paths in Graphs (2)

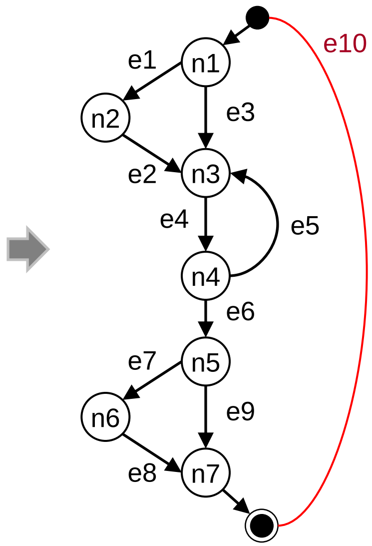
- *A* **circuit** is a path that begins and ends at the same node
  - e.g., P3 = <e3, e4, e6, e7, e8, e10> begins and ends at node n1
  - P6 = <e4, e5> begins and ends at node n3
- *A* **cycle** is a circuit with no node (other than the starting node) included more than once

Example Circuits & Cycles

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



Circuits:	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P3 = e3, e4, e6, e7, e8, e10	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 = e6, e7, e8, e10, e3, e4	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 = e1, e2, e4, e6, e9, e10	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 = e4, e5	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 = e3, e4, e6, e9, 10	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1,	1,	0,	2,	1,	1,	0,	0,	1,	1
P9 = e3, e4, e5, e4, e6, e9, 10	0,	0,	1,	2,	1,	1,	0,	0,	1,	1

Cycles:

P3 = e3, e4, e6, e7, e8, e10

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

P7 = e3, e4, e6, e9, 10

P4, P8, P9 are not cycles

Linearly Independent Paths

- A path p is said to be a **linear combination** of paths $p_1, \ldots, p_n$ if there are integers $a_1, \ldots, a_n$ such that $p = \sum a_i \cdot p_i$ (a_i could be negative, zero, or positive)
- A set of paths in a strongly connected graph is **linearly independent** if no path in the set is a linear combination of any other paths in the set
  - A **linearly independent path** is any path *through* the program ("complete path") that introduces at least one *new edge* that is not included in any other linearly independent paths.
    - A path that is subpath of another path is not considered to be a linearly independent path.
- A **basis set** of *cycles* is a maximal linearly independent set of cycles
  - In a graph with e edges and n nodes, the basis has $e - n + 1$ cycles
    - +1 is for the virtual edge, introduced to obtain a strongly connected graph
- Every path is a linear combination of basis cycles

Baseline method for finding the basis set of cycles

- Start at the source node
  (the first statement of the program/module)
- Follow the leftmost path until the sink node is reached
- Repeatedly retrace this path from the source node, but change decisions at every node with out-degree ≥2, starting with the decision node earliest in the path

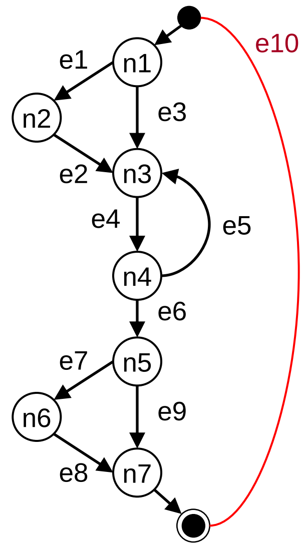
T.J. McCabe & A.H. Watson, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, 1996.

Linearly Independent Paths (1)

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



$\boxed{V(G) = e - n + 2 = 9 - 7 + 2 = 4}$

Or, if we count e10, then e – n + 1 = 10 – 7 + 1 = 4

Example paths:

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1 = e1, e2, e4, e6, e7, e8	1,	1,	0,	1,	0,	1,	1,	1,	0,	0
P2 = e1, e2, e4, e5, e4, e6, e7, e8	1,	1,	0,	2,	1,	1,	1,	1,	0,	0
P3 = e3, e4, e6, e7, e8, e10	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 = e6, e7, e8, e10, e3, e4	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 = e1, e2, e4, e6, e9, e10	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 = e4, e5	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 = e3, e4, e6, e9, 10	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1,	1,	0,	2,	1,	1,	0,	0,	1,	1

Cycles:

P3 = e3, e4, e6, e7, e8, e10

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

P7 = e3, e4, e6, e9, 10

EXAMPLE #1: P5 + P6 = P8

P5	{1, 1, 0, 1, 0, 1, 0, 0, 1, 1}
+ P6	{0, 0, 0, 1, 1, 0, 0, 0, 0, 0}
= P8	{1, 1, 0, 2, 1, 1, 0, 0, 1, 1}

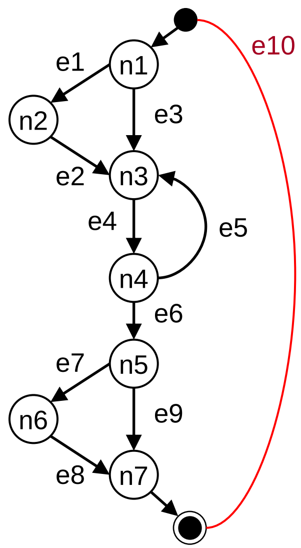
EXAMPLE #2: 2×P3 – P5 + P6 =

2×P3	{ 0, 0, 2, 2, 0, 2, 2, 2, 0, 2}
– P5	{ 1, 1, 0, 1, 0, 1, 0, 0, 1, 1}
	{-1,-1, 2, 1, 0, 1, 2, 2,-1, 1}
+ P6	{ 0, 0, 0, 1, 1, 0, 0, 0, 0, 0}
~~= P?~~	~~{-1,-1, 2, 2, 1, 1, 2, 2,-1, 1}~~

➔ **Problem**: The arithmetic doesn't work for *any* paths

— it works *always* only for *linearly independent paths*!

Linearly Independent Paths (2)



V(G) = e – n + 2 = 9 – 7 + 2 = 4

Linearly Independent Paths:

(by enumeration)

P1' = e1, e2, e4, e6, e7, e8, e10

P2' = e1, e2, e4, e5, e4, e6, e7, e8, e10

P3' = e3, e4, e6, e7, e8, e10

P4' = e1, e2, e4, e6, e9, e10

(P4 same as P3)

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1 =	1,	1,	0,	1,	0,	1,	1,	1,	0,	0
P2 =	1,	1,	0,	2,	1,	1,	1,	1,	0,	0
P3 =	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 =	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 =	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 =	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 =	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 =	1,	1,	0,	2,	1,	1,	0,	0,	1,	1

EXAMPLE #3: P6 = P2' – P1'

P2'	{1, 1, 0, 2, 1, 1, 1, 1, 0, 0}
– P1'	{1, 1, 0, 1, 0, 1, 1, 1, 0, 0}
= P6	{0, 0, 0, 1, 1, 0, 0, 0, 0, 0}

EXAMPLE #4: P7 = P3' + P4' – P1'

P3'	{0, 0, 1, 1, 0, 1, 1, 1, 0, 1}
+ P4'	{0, 0, 1, 1, 0, 1, 1, 1, 0, 1}
– P1'	{1, 1, 0, 1, 0, 1, 1, 1, 0, 0}
= P7	{0, 0, 1, 1, 0, 1, 0, 0, 1, 1}

EXAMPLE #5: P8 = P2' – P1' + P4'

P2'	{1, 1, 0, 2, 1, 1, 1, 1, 0, 0}
– P1'	{1, 1, 0, 1, 0, 1, 1, 1, 0, 0}
+ P4'	{0, 0, 1, 1, 0, 1, 1, 1, 0, 1}
= P8	{1, 1, 0, 2, 1, 1, 0, 0, 1, 1}

Q: Note that P2' = P1' + P6, so why not use P1' and P6 instead of P2'?

A: Because P6 is not a “complete path”, so it cannot be a linearly independent path

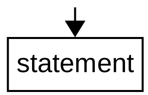
Unit Testing: Path Coverage

  – Finds the number of distinct paths through the program to be traversed at least once

- Minimum number of tests necessary to cover all edges is equal to the **number of independent paths** through the control-flow graph
- (Recall the lecture on Unit Testing)

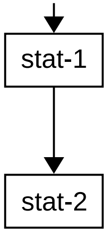
Issues (1)

Single statement:



= CC =

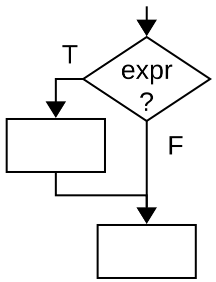
Two (or more) statements:



Cyclomatic complexity (CC) remains the same for a linear sequence of statements regardless of the sequence length
—insensitive to complexity contributed by the multitude of statements
(Recall that discriminative power (sensitivity) is a desirable property of a metric)

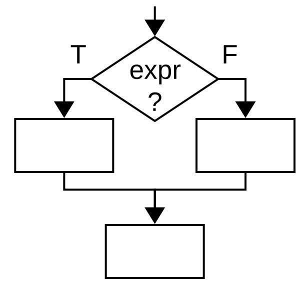
Issues (2)

Optional action:



= CC =

Alternative choices:

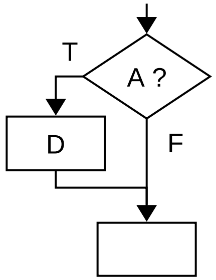


Optional action versus alternative choices —
the latter is psychologically more difficult

Issues (3)

Simple condition:

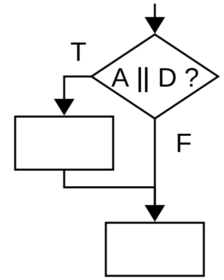
```
if (A) then D;
```



= CC =

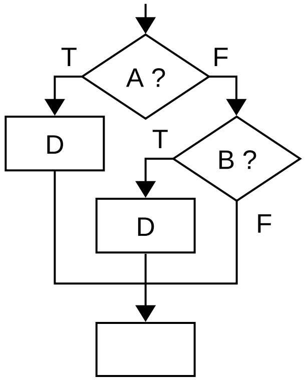
Compound condition:

```
if (A OR B) then D;
```



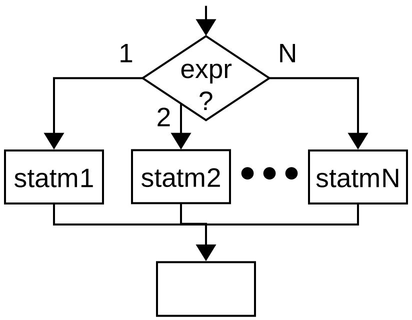
BUT, compound condition can be written as a nested IF:

```
if (A) then D;
else if (B) then D;
```



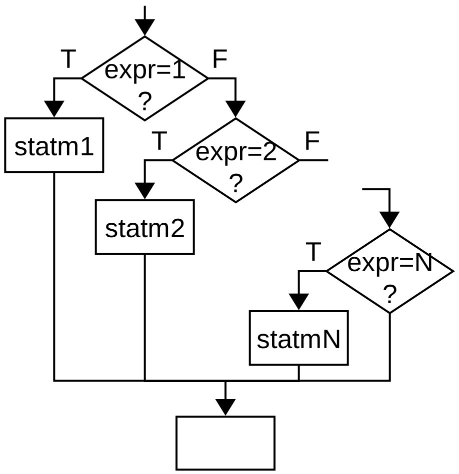
Issues (4)

Switch/Case statement:



= CC =

N–1 predicates:



<u>Counting a **switch** statement</u>:

—as a single decision

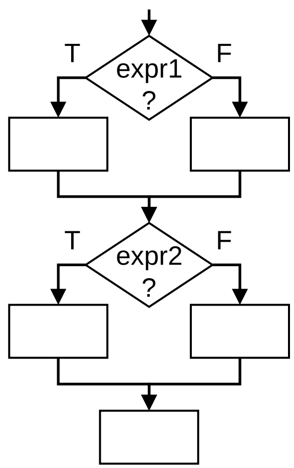
proposed by W. J. Hansen, "Measurement of program complexity by the pair (cyclomatic number, operator count)," *SIGPLAN Notices*, vol.13, no.3, pp.29-33, March 1978.

—as $\log_2(N)$ relationship

proposed by V. Basili and R. Reiter, "Evaluating automatable measures for software development,"•*Proceedings of the IEEE Workshop on Quantitative Software Models for Reliability, Complexity and Cost*, pp.107-116, October 1979.

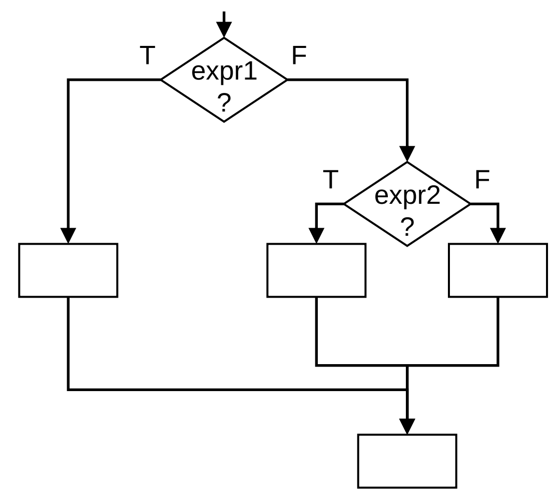
Issues (5)

Two sequential decisions:



= CC =

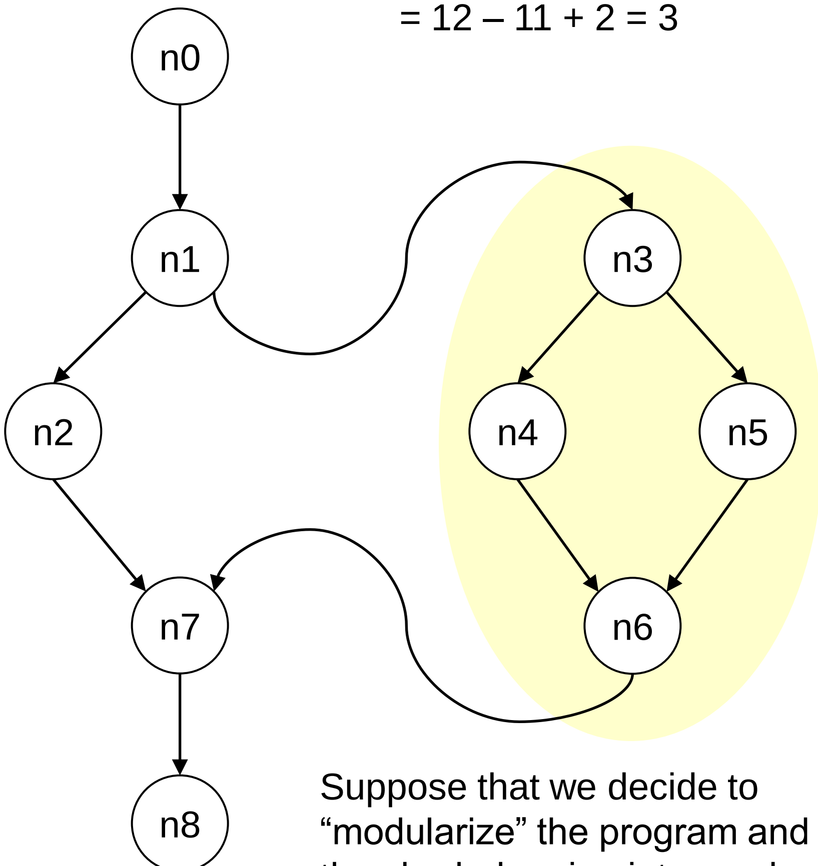
Two nested decisions:



But, it is known that people find nested decisions more difficult …

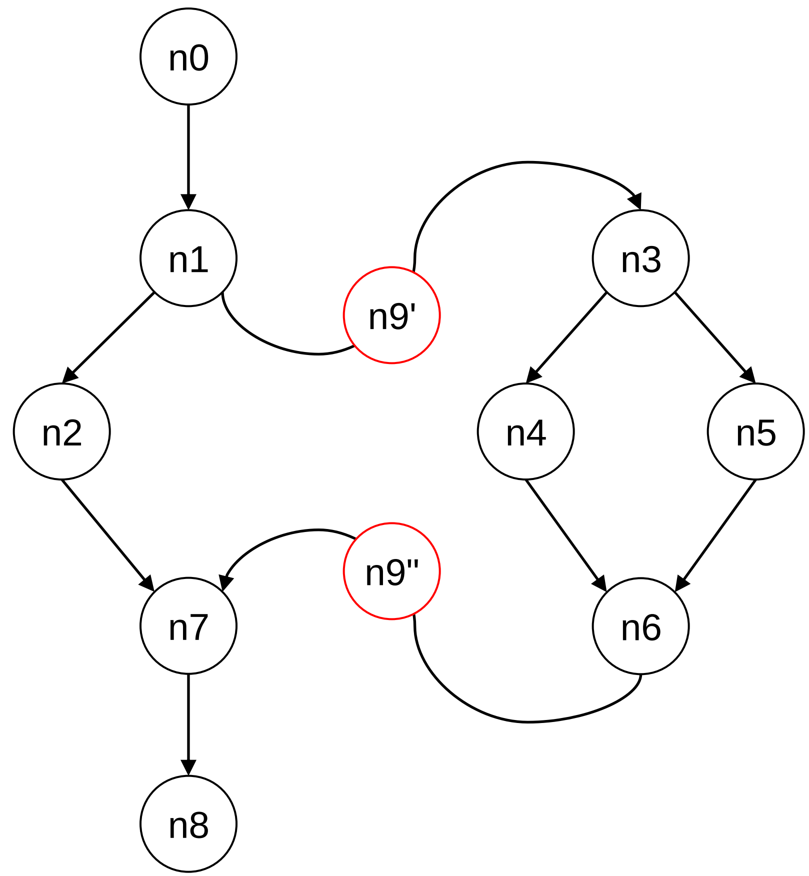
CC for Modular Programs (1)

V = e – n + 2
= 12 – 11 + 2 = 3



Suppose that we decide to "modularize" the program and make the shaded region into a subroutine

Adding a sequential node does not change CC:

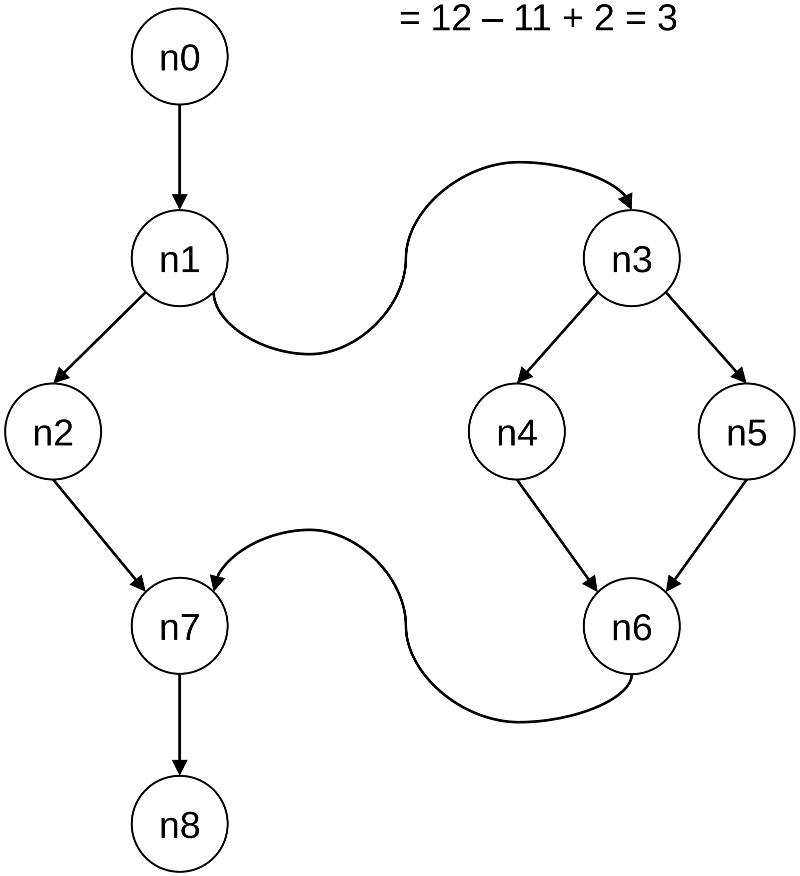


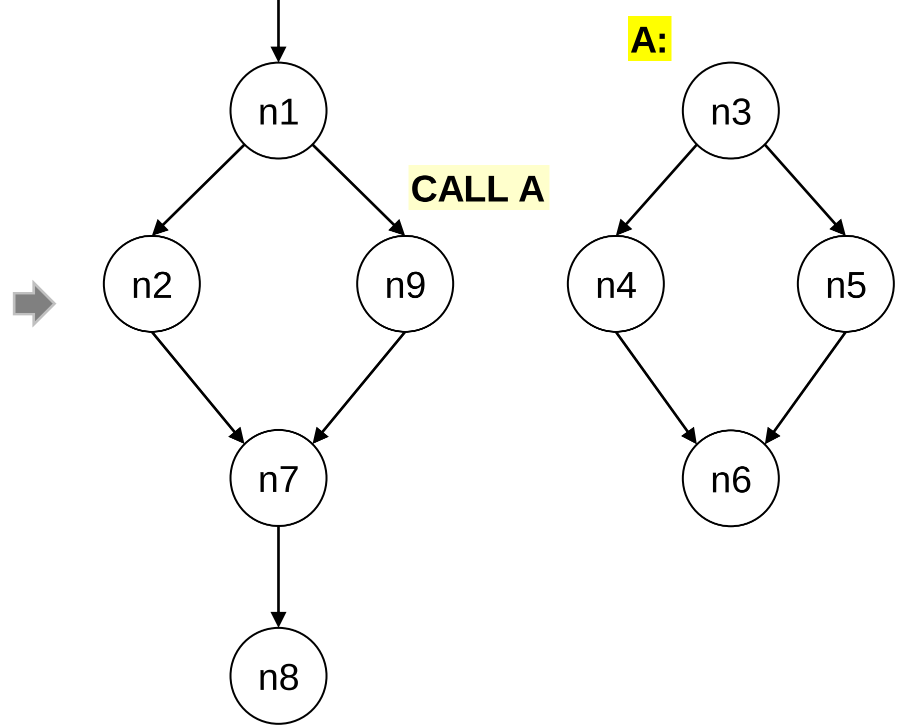
CC for Modular Programs (2)

Intuitive expectation:
Modularization should not increase complexity

$V = e - n + 2$
$= 12 - 11 + 2 = 3$

$V = e - n + 2p$
$= 10 - 10 + 2 \times 2 = 4$





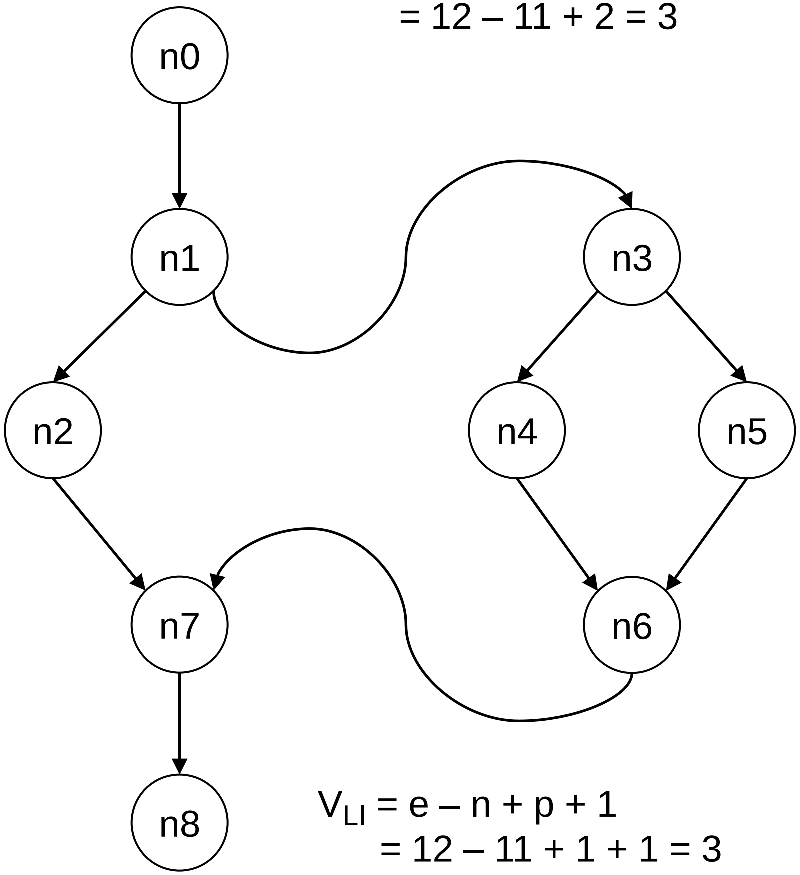
Modified CC Measures

- Given ***p*** connected components of a graph:
  - $V(G) = e - n + 2p$ (1)
  - $V_{LI}(G) = e - n + p + 1$ (2)
  - Eq. (2) is known as *linearly-independent* cyclomatic complexity
  - V_{LI} does not change when program is modularized into *p* modules

CC for Modular Programs (3)

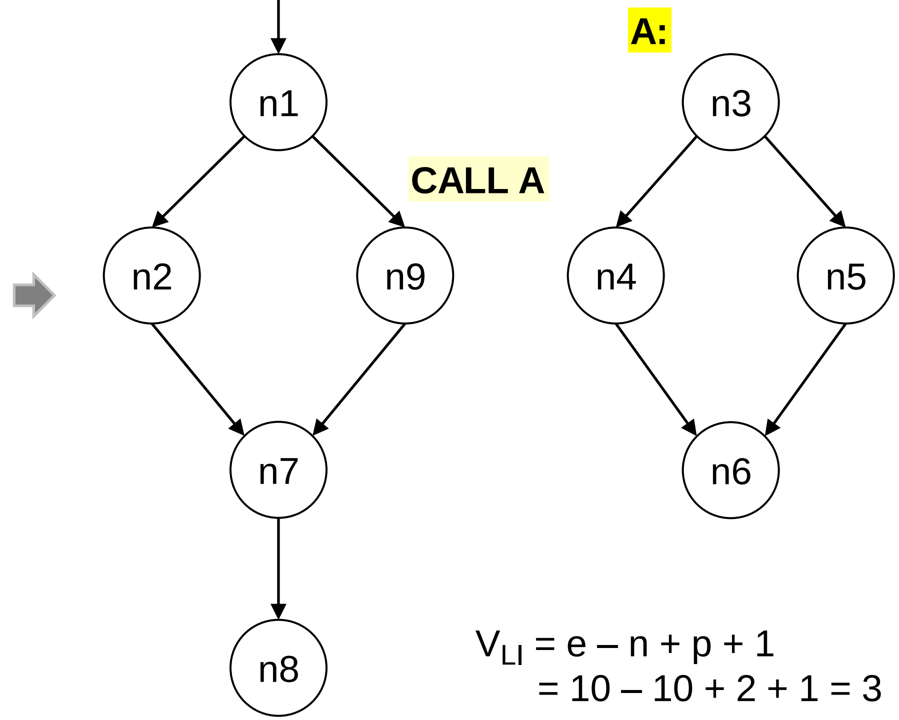
Intuitive expectation:
Modularization should not increase complexity

$$V = e - n + 2 = 12 - 11 + 2 = 3$$



$$V_{LI} = e - n + p + 1 = 12 - 11 + 1 + 1 = 3$$

$$V = e - n + 2p = 10 - 10 + 2 \times 2 = 4$$



$$V_{LI} = e - n + p + 1 = 10 - 10 + 2 + 1 = 3$$

Practical SW Quality Issues (1)

- No program module should exceed a cyclomatic complexity of 10
    - Originally suggested by McCabe
    - P. Jorgensen, *Software Testing: A Craftman's Approach, 2nd Edition*, CRC Press Inc., pp.137-156, 2002 .
- Software refactorings are aimed at reducing the complexity of a program's conditional logic

♦ *Refactoring: Improving the Design of Existing Code*
by Martin Fowler, et al.; Addison-Wesley Professional, 1999.
♦ *Effective Java* (2nd Edition)
by Joshua Bloch; Addison-Wesley, 2008.

Practical SW Quality Issues (2)

- Cyclomatic complexity is a screening method, to check for potentially problematic code.
- As any screening method, it may turn false positives and false negatives
- Will learn about more screening methods (cohesion, coupling, …)