Masking and Introduction to Data Flow Analysis

CSCE 747 - Lecture 8 - 02/02/2017

Previously...

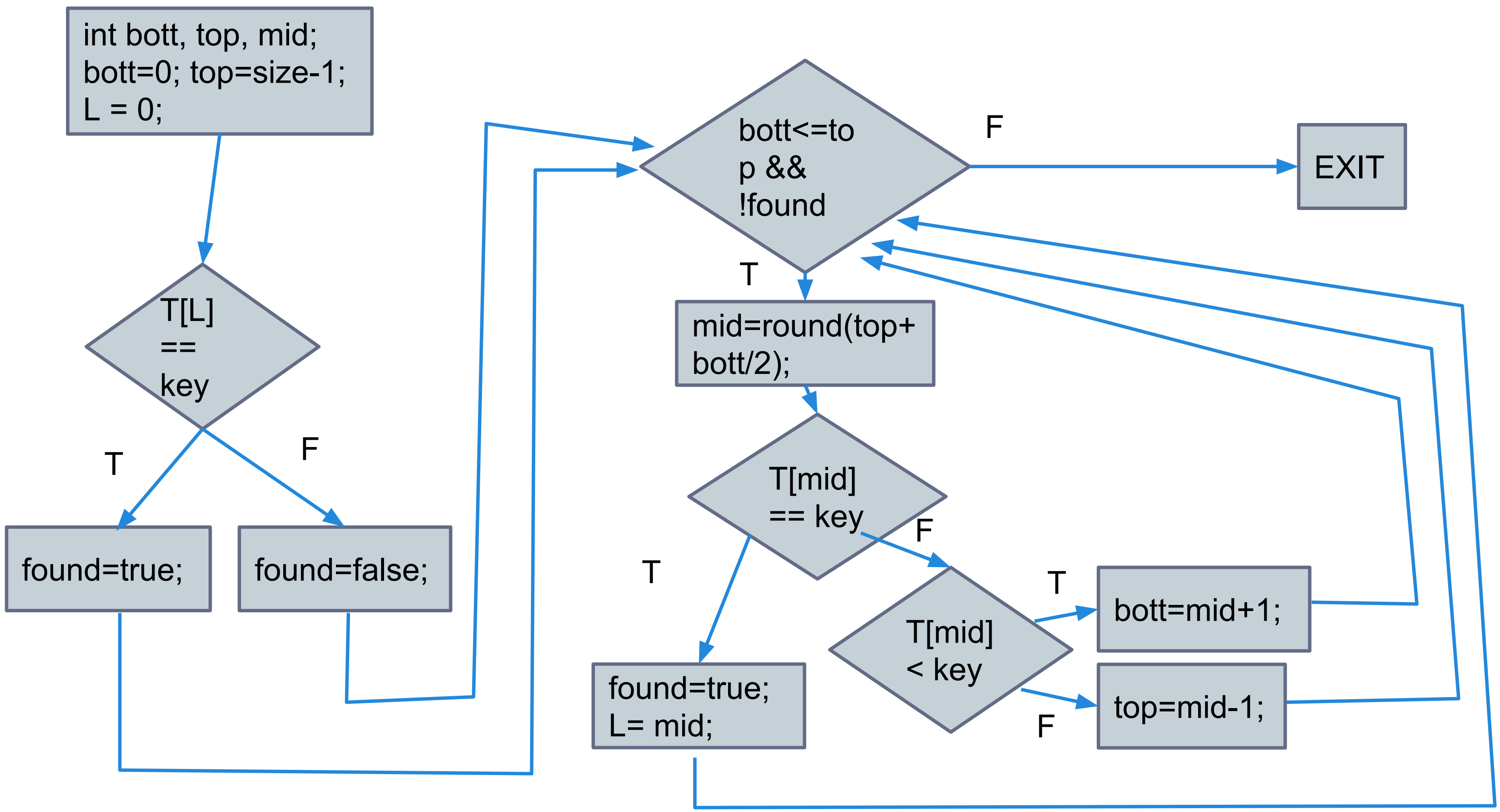
- Test adequacy can be assessed through **adequacy metrics**.
- Many are based on elements from the **program structure**.
  - Statements, branches, conditions, procedure calls.
- Others are based on control paths.
  - Sequences of edges in the CFG.
  - Path coverage, boundary interior coverage, loop coverage.

Activity:
Writing Loop-Covering Tests

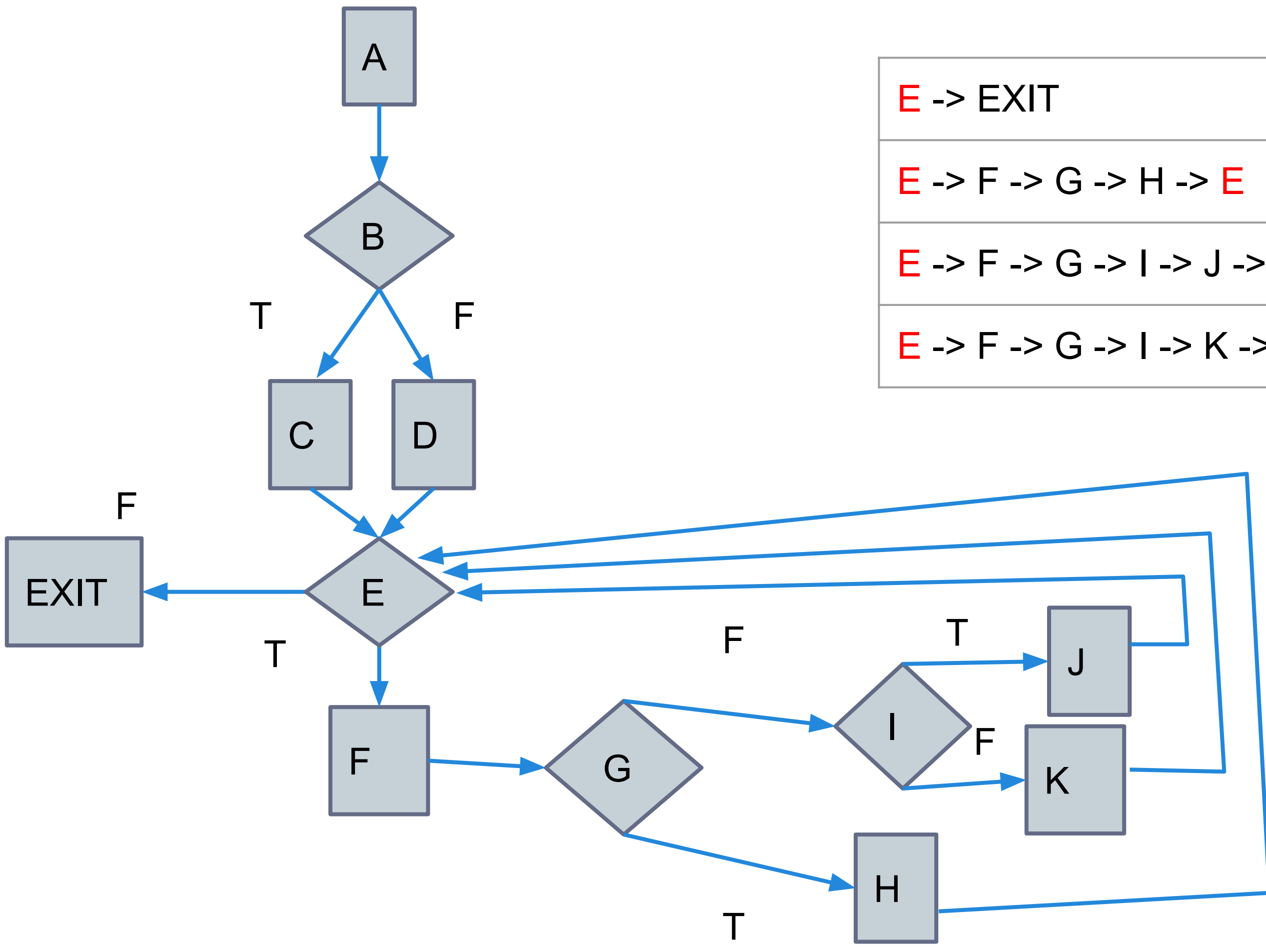
For the binary-search code:

1. Draw the control-flow graph for the method.
2. Identify the subpaths through the loop and draw the unfolded CFG for boundary interior testing.
3. Develop a test suite that achieves loop boundary coverage.

CFG

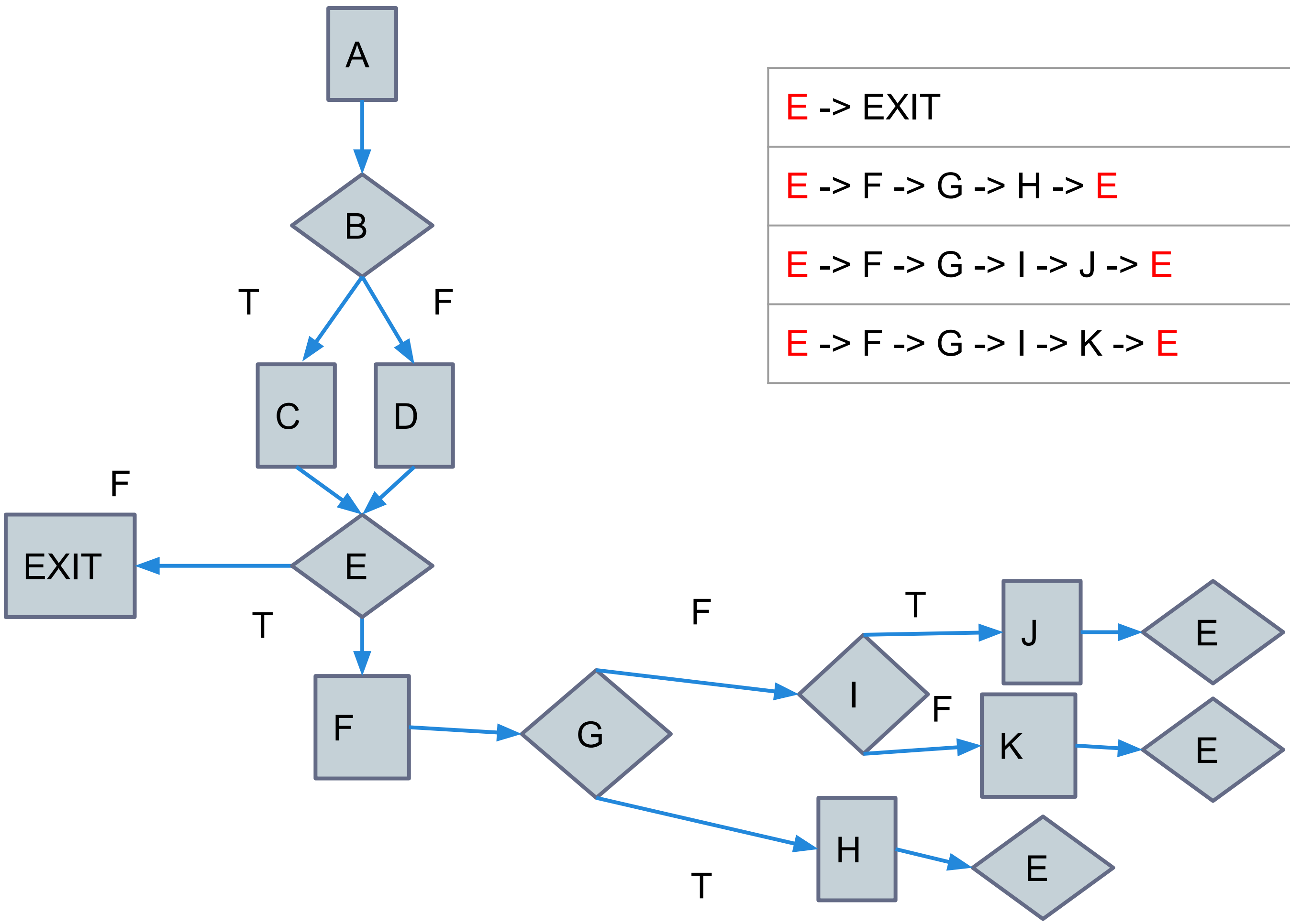

int bott, top, mid;
bott=0; top=size-1;
L = 0;
T[L]
==
key
T
F
found=true;
found=false;
bott<=to
p &&
!found
F
EXIT
T
mid=round(top+
bott/2);
T[mid]
== key
F
T
found=true;
L= mid;
T[mid]
< key
T
F
bott=mid+1;
top=mid-1;

CFG



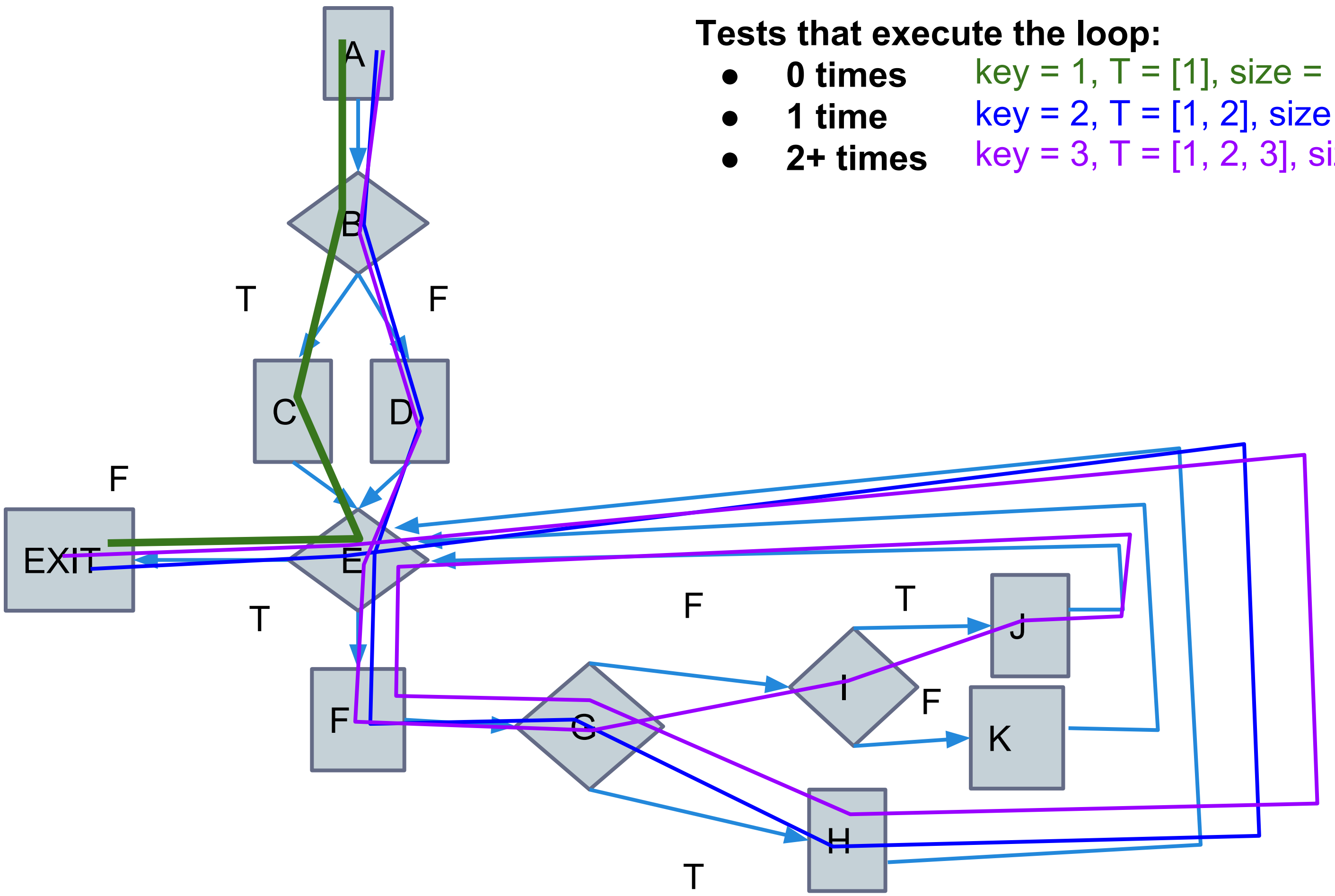
E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



Tests that execute the loop:

- **0 times** key = 1, T = [1], size = 1
- **1 time** key = 2, T = [1, 2], size = 2
- **2+ times** key = 3, T = [1, 2, 3], size = 3

The Infeasibility Problem

Sometimes, **no** test can satisfy an obligation.

- Impossible combinations of conditions.
- Unreachable statements as part of defensive programming.
  - Error-handling code for conditions that can't actually occur in practice.
- Dead code in legacy applications.
- Inaccessible portions of off-the-shelf systems.

The Infeasibility Problem

Stronger criteria call for potentially infeasible combinations of elements.

```
(a > 0 && a < 10)
```

It is not possible for both conditions to be false.

Problem compounded for path-based coverage criteria.

Not possible to traverse the path where both if-statements evaluate to true.

```
if (a < 0)  a = 0;
if (a > 10) a = 10;
```

The Infeasibility Problem

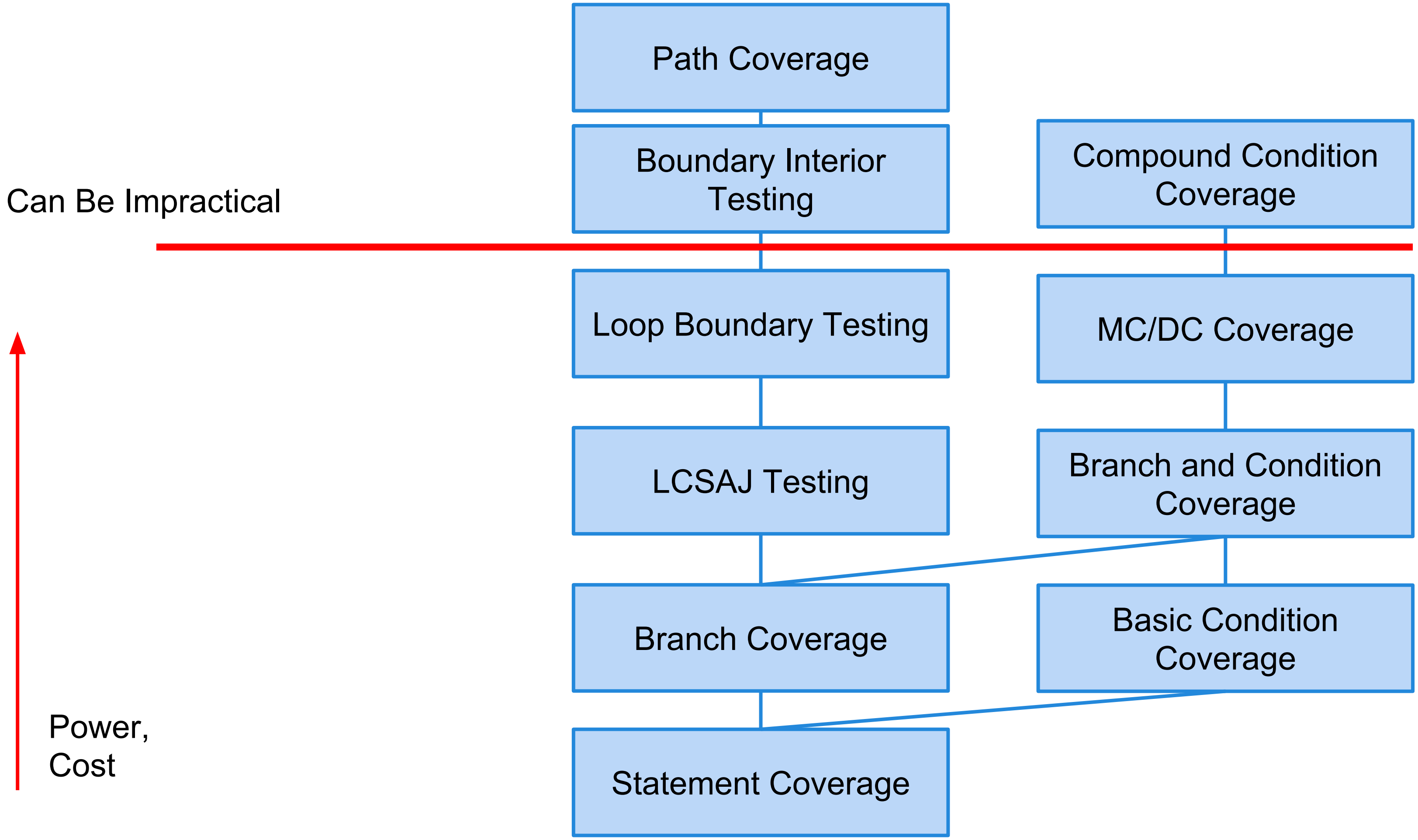
How this is usually addressed:

- Adequacy “scores” based on coverage.
  - 95% branch coverage, 80% MC/DC coverage, etc.
  - Decide to stop once a threshold is reached.
  - Unsatisfactory solution - elements are not equally important for fault-finding.
- Manual justification for omitting each impossible test obligation.
  - Required for safety certification in avionic systems.
  - Helps refine code and testing efforts.
  - … but **very** time-consuming.

In Practice.. The Budget Coverage Criterion

- Industry's answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1:
    - Adequacy criteria answer the wrong question. Selection is more important.
  - Implication 2:
    - Practical comparison of approaches must consider the cost of test case selection

Which Coverage Metric Should I Use?



Where Coverage Goes Wrong...

- Testing can only reveal a fault when execution of the faulty element causes a failure, but…
- Execution of a line containing a fault does not guarantee a failure.
    - (a <= b) accidentally written as (a >= b) - the fault will not manifest as a failure if a==b in the test case.
- Merely executing code does not guarantee that we will find all faults.

Don’t Rely on Metrics

- There is benefit from using coverage as a stopping criterion.
- But, auto-generating tests with coverage as the goal produces poor tests.
- Two key problems - sensitivity to how code is written, and whether infected program state is noticed by oracle.

Masking

- One statement can mask the effect of another statement.
    - X = (A && B)
    - Y = (X || Z)
    - If Z is false, then X is **masked**.
        - It doesn't matter what the value of X is.
- Coverage metrics focus on one element at a time (one statement, one branch, one path).
    - MC/DC can ensure that X influences Y, but not that A influences Y.
    - This could **mask** a fault in A.

Sensitivity to Structure

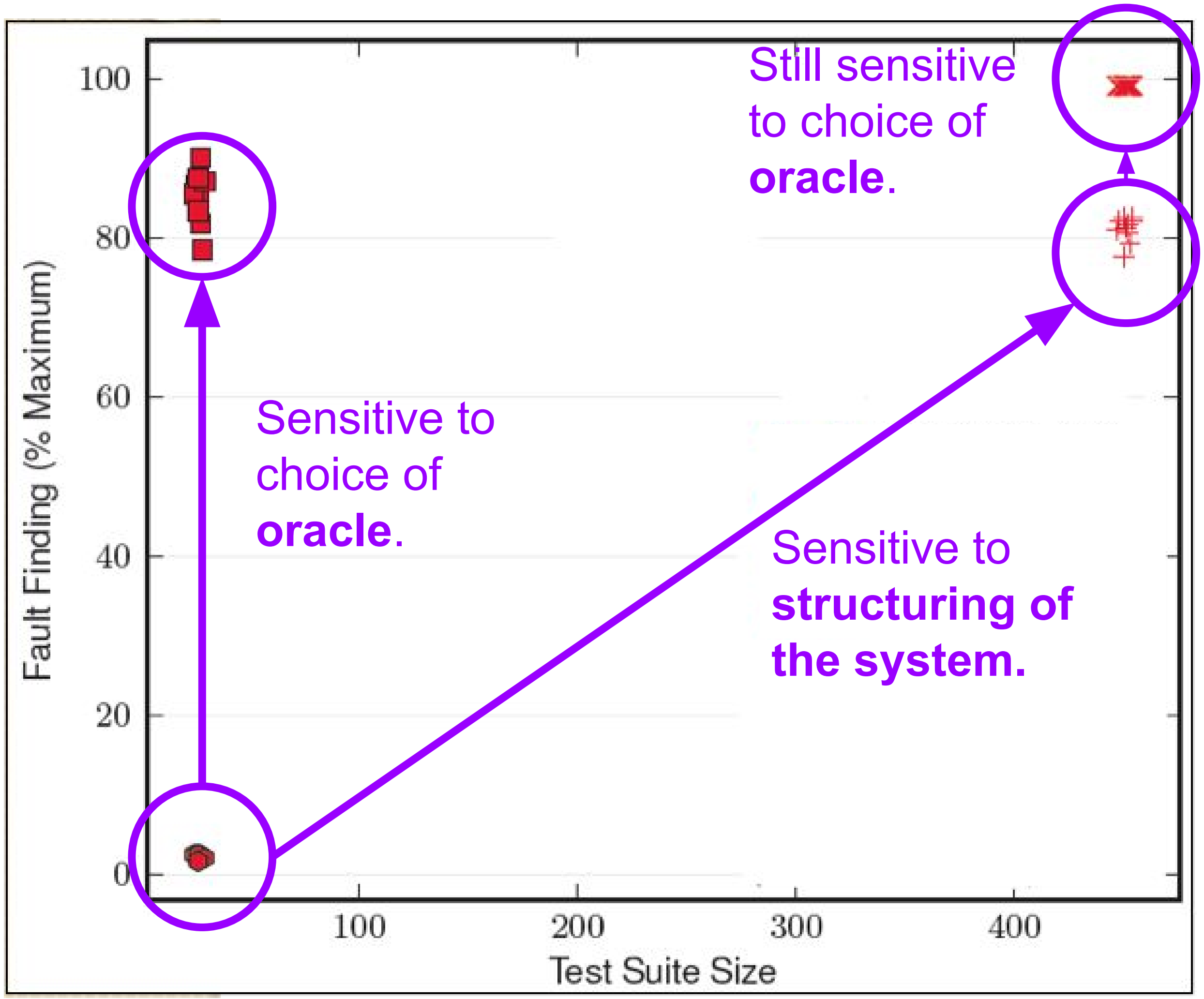
```
expr_1 = in_1 || in_2;
out_1 = expr_1 && in_3;

out_1 = (in_1 || in_2) && in_3;
```

- Both pieces of code do the same thing.
- How code is written impacts the number and type of tests needed.
- Simpler statements result in simpler tests.
    - And introduce risk of **masking**.

Sensitivity to Oracle

- The oracle judges test correctness.
  - We need to choose what results we check when writing an oracle.
- Typically, we check certain output variables.
  - However, **masking** can prevent us from noticing a fault if we do not check the right variables.
  - We can’t monitor and check all variables.
  - But, we can carefully choose a small number of bottleneck points and check those.
    - Some techniques for choosing these, but still more research to be done.

Coverage Effectiveness



Masking

Why do we care about faults in masked expressions?

- Effect of fault is only masked out for *this* test.
- It is still a fault. In another execution scenario, it might not be masked.
  - We just haven't noticed it yet.
  - The fault isn't gone, we just have bad tests.
- One solution - ensure that there is a path from assignment to output where we will **notice the fault**.

Path Conditions

- Most coverage criteria impose constraints on a single element.
- However, test obligations can also impose constraints on the path taken.
    - I.e., path coverage, boundary interior coverage
- Existing coverage criteria can be strengthened with path constraints.

Observability

- MC/DC eliminates masking in individual statements by requiring independent impact.
- However, that statement's effect can be masked by another statement.
- Observability measures ability to infer internal system activity from information we monitor.
  - Can increase by using a larger oracle.
  - This is expensive. Instead, build it into the coverage metric.

Observable MC/DC

- Assessing “independent impact” requires showing that a change in a condition’s value affects the value of an expression.
- Same idea can be applied to the path.
- Observability requires showing that a change to a targeted element affects a monitored variable.
- Observable MC/DC adds observability constraints to MC/DC.

Tagging Semantics

Assign each condition a **tag set**:

`(ID, Boolean Outcome)`

Evaluation determines tag propagation:

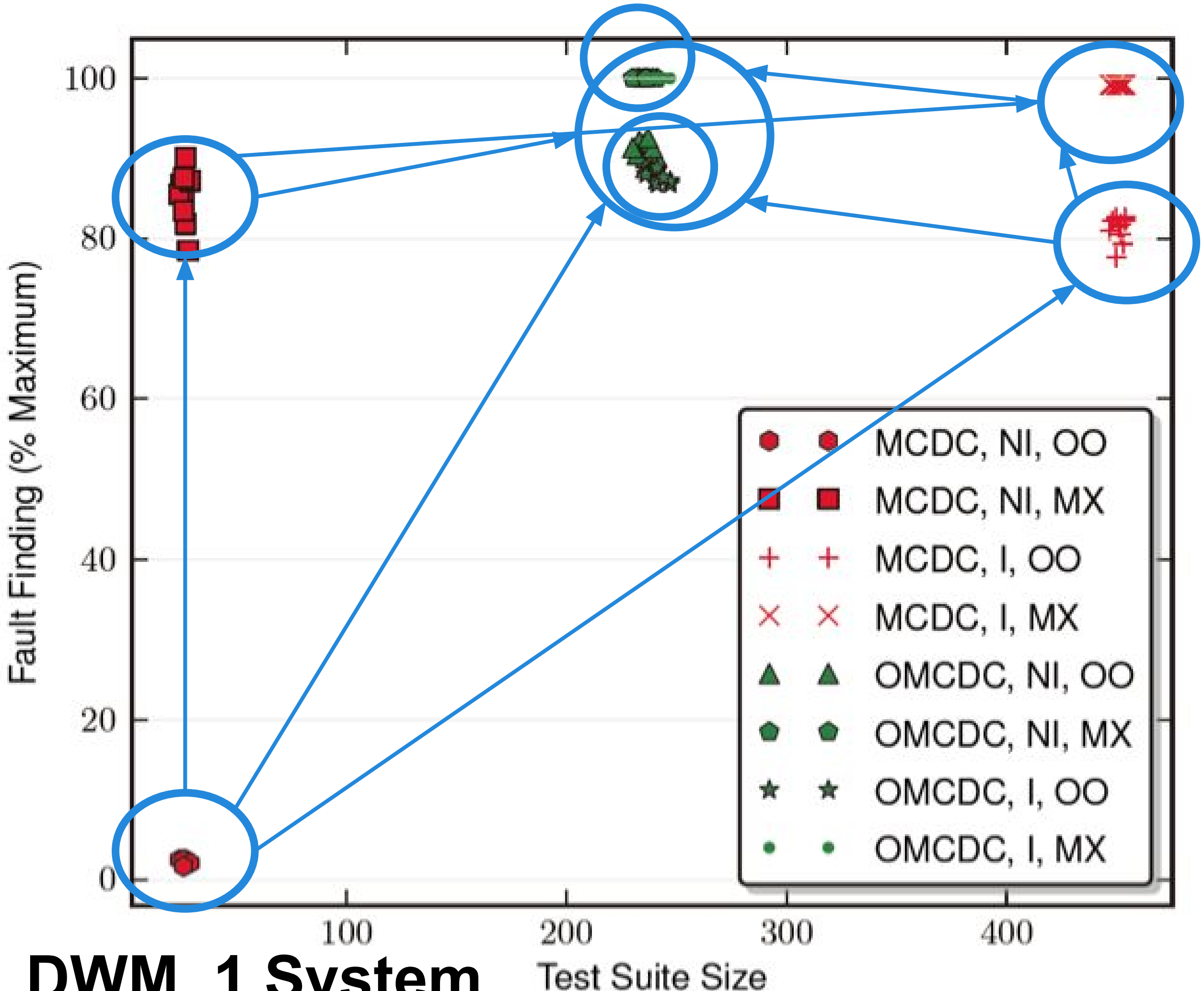
`exp1=c1 && c2;`	[~~(c1,true)~~, (c2,false)]
`exp2=c3 || c4;`	[(c3,true), ~~(c4,false)~~]
`out=if (c5) then exp1 else exp2;`	[(c5,true),(c2, false), ~~<exp2>~~ <exp1>,]<exp2>]

Benefits of Observability

Observability should improve test effectiveness by accounting for **program structure** and **oracle composition**:

- We select what points the oracle monitors, OMC/DC requires propagation path to those points.
- No sensitivity to structure because impact must be propagated at monitoring points.
  - i.e., we place conditions on the path taken.

Evaluation - Results



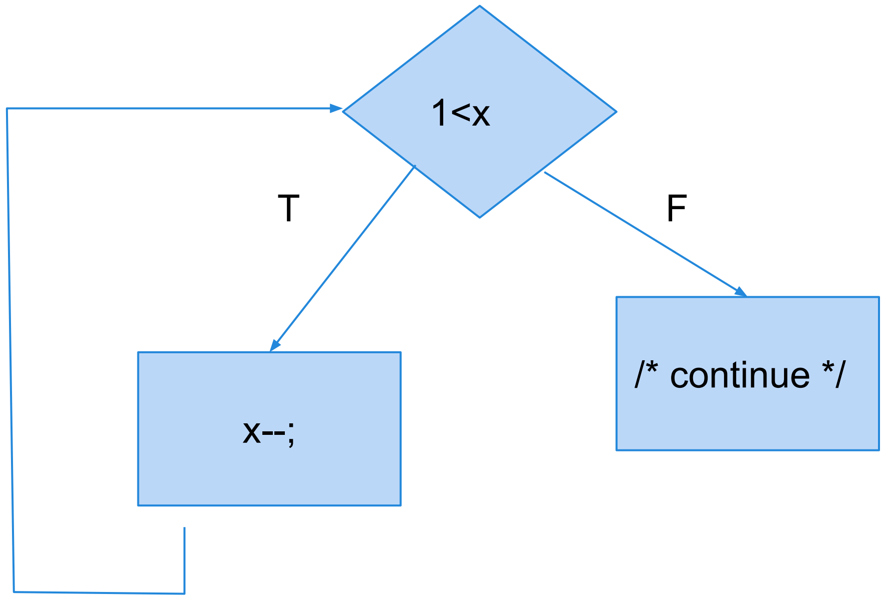
DWM_1 System

Still Not a Solved Problem

- OMC/DC often prescribes a large number of infeasible obligations.
- Tests can be difficult to derive.
- Often results in better fault-finding, but not 100% fault-finding (especially in complex systems).
- Points to our next topic - **the importance of how code executes**.

Control Flow

- Capture dependencies in terms of how control passes between parts of a program.
- We care about the effect of a statement when it affects the path taken.
    - but deemphasize the information being transmitted.



Data Flow

- Another view - program statements compute and transform data…
  - So, look at how that data is passed through the program.
- Reason about **data** dependence
  - A variable is used here - where does its value from?
  - Is this value ever used?
  - Is this variable properly initialized?
  - If the expression assigned to a variable is cha what else would be affected?

Data Flow

- Basis of the optimization performed by compilers.
- Used to derive test cases.
  - Have we covered the dependencies?
- Used to detect faults and other anomalies.
  - Is this string tainted by a fault in the expression that calculates its value?

Definition-Use Pairs

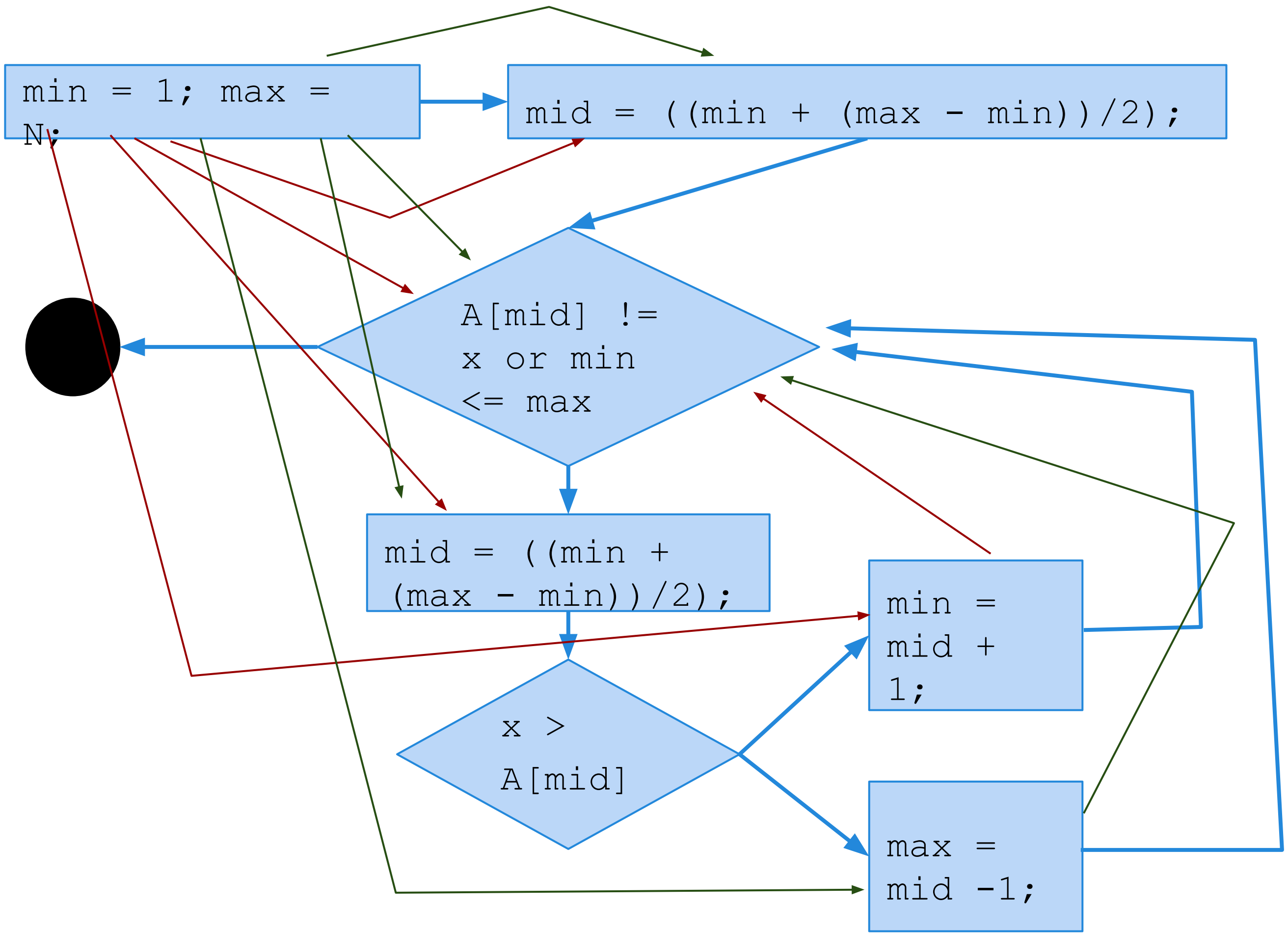
- Data is defined.
  - Variables are declared and assigned values.
- … and data is used.
  - Those variables are used to perform computations.
- Associations of definitions and uses capture the flow of information through the program.
  - Definitions occur when variables are declared, initialized, assigned values, or received as parameters.
  - Uses occur in expressions, conditional statements, parameter passing, return statements.

Example - Definition-Use Pairs

```
1.  min = 1;
2.  max = N;
3.  mid = ((min + (max - min))/2);
4.  while (A[mid] != x or min <= max){
5.      mid = ((min + (max - min))/2);
6.      if (x > A[mid]){
7.          min = mid + 1
8.      } else {
9.          max = mid - 1;
10.     }
11. }
```

1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use -** x, A[mid], mid
7. **def -** min, **use** - mid
8. -
9. **def -** max, **use -** mid

Example - Definition-Use Pairs



1. **def** - min
2. **def** - max, **use** - N
3. **def** - mid, **use** - min, max
4. **use** - A[mid], mid, x, min, max
5. **def** - mid, **use** - min, max
6. **use** - x, A[mid], mid
7. **def** - min, **use** - mid
8. -
9. **def** - max, **use** - mid

Def-Use Pairs

- We can say there is a def-use pair when:
  - There is a *def* (definition) of variable *x* at location A.
  - Variable *x* is *use*d at location B.
  - A control-flow path exists from A to B.
  - and the path is *definition-clear* for x.
    - If a variable is redefined, the original def is *killed* and the pairing is between the new definition and its associated use.

Example - Definition-Use Pairs

```
1.   min = 1;
2.   max = N;
3.   mid = ((min + (max - min))/2);
4.   while (A[mid] != x or min <= max){
5.        mid = ((min + (max - min))/2);
6.        if (x > A[mid]){
7.             min = mid + 1
8.        } else {
9.             max = mid - 1;
10.       }
11.  }
```

DU Pairs
min: (1, 3), (1, 4), (1, 5), (7, 4), (7, 5)
max: (2, 3), (2, 4), (1, 5), (9, 4), (9, 5)
N: (0, 2)
mid: (3, 4), (5, 6), (5, 7), (5, 9), (5, 4)
x: (0, 4), (0, 6)
A: (0, 4), (0, 6)

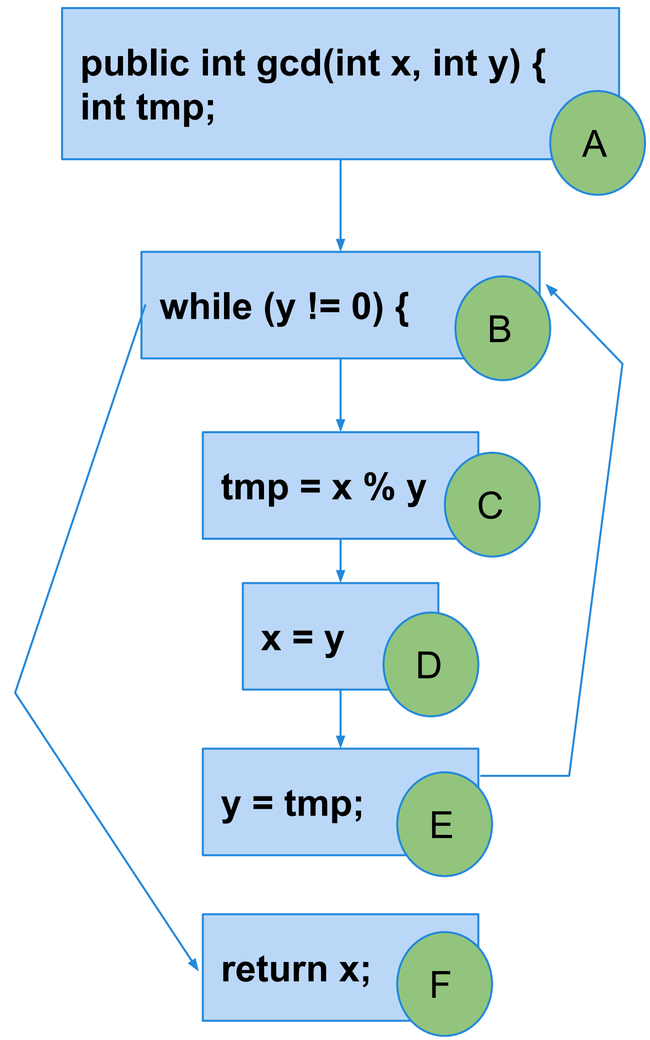
Example - GCD

```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```

1. def: x, y
2. def: tmp
3. use: y
4. use: x, y
   def: tmp
5. use: y
   def: x
6. use: tmp
   def: y
7. -
8. use: x

Example - GCD

```
1.  public int gcd(int x, int y){
2.      int tmp;
3.      while(y!=0){
4.          tmp = x % y;
5.          x = y;
6.          y = tmp;
7.      }
8.      return x;
9.  }
```



Def-Use Pairs
x: (1, 4), (5, 4), (5, 8),
y: (1, 3), (1, 4), (1, 5), (6, 3), (6, 4), (6, 5)
tmp: (4, 6)

Activity - DU Pairs

- For the provided code, identify all DU pairs.
  - Hint - first, find all definitions and uses, then link them.
  - DU Pair = there exists a *definition-clear path* between the definition of x and a use of x.
    - If x is redefined on the path, the original definition is *killed* and replaced.
  - Remember that there is a loop.

Activity Solution - Defs and Uses

```
7. public static String collapseNewlines(String argStr)
8. {
9.     char last = argStr.charAt(0);
10.    StringBuffer argBuf = new StringBuffer();
11.
12.    for(int cldx = 0; cldx < argStr.length(); cldx++)
13.    {
14.          char ch = argStr.charAt(cldx);
15.          if(ch != ‘\n’ || last != ‘\n’)
16.          {
17.                argBuf.append(ch);
18.                last = ch;
19.          }
20.    }
21.
22.    return argBuf.toString();
23. }
```

Variable	Definitions	Uses
argStr	7	9, 12, 14
last	9, 18	15
argBuf	10, 17	22
cldx	12	12, 14
ch	14	15, 17, 18

Activity Solution - Def-Use Pairs

```
7. public static String collapseNewlines(String argStr)
8. {
9.     char last = argStr.charAt(0);
10.    StringBuffer argBuf = new StringBuffer();
11.
12.    for(int cldx = 0; cldx < argStr.length(); cldx++)
13.    {
14.          char ch = argStr.charAt(cldx);
15.          if(ch != '\n' || last != '\n')
16.          {
17.                argBuf.append(ch);
18.                last = ch;
19.          }
20.    }
21.
22.    return argBuf.toString();
23. }
```

Variable	D-U Pairs
argStr	(7, 9), (7,12), (7, 14)
last	(9, 15), (18, 15)
argBuf	(17, 22)
cldx	(12, 12), (12, 14)
ch	(14, 15), (14, 17), (14

We Have Learned

- Control-flow and data-flow both capture important paths in program execution.
- Analysis of how variables are defined and then used and the dependencies between definitions and usages can help us reveal important faults.
- Many forms of analysis can be performed using data flow information.

Next Class

- Data flow analysis.
    - Using Def-Use pairs to understand how programs work.

- Reading: Chapter 6
- Homework 1 due tonight.
- Reading assignment 2 out. Due February 9th.