# Graph-based test coverage (part 2)

- Data flow graph coverage
- All-Defs, All-Uses, All-Du-Paths coverage
- Last-def, First-use call analysis

Ciências ULisboa

slides: Eduardo Marques, Vasco Vasconcelos, Francisco Martins, João Neto

# Data flow coverage criteria for graphs

- **Assumption**: To test a program adequately we must focus on the flows of data values, to ensure that created values are used correctly
- Given a program and a variable $v$ of that program:
  - A **definition** of $v$ is a program location that assigns (writes to) $v$.
  - An **use** of $v$ is a program location that accesses (reads from) $v$.
- Given a graph $G = (N, N_0, N_f, E)$ and $n \in N$, $e \in E$
  - $def(n)$ and $def(e)$: set of variables defined by $n$ or $e$
  - $use(n)$ and $use(e)$: set of variables used by $n$ or $e$

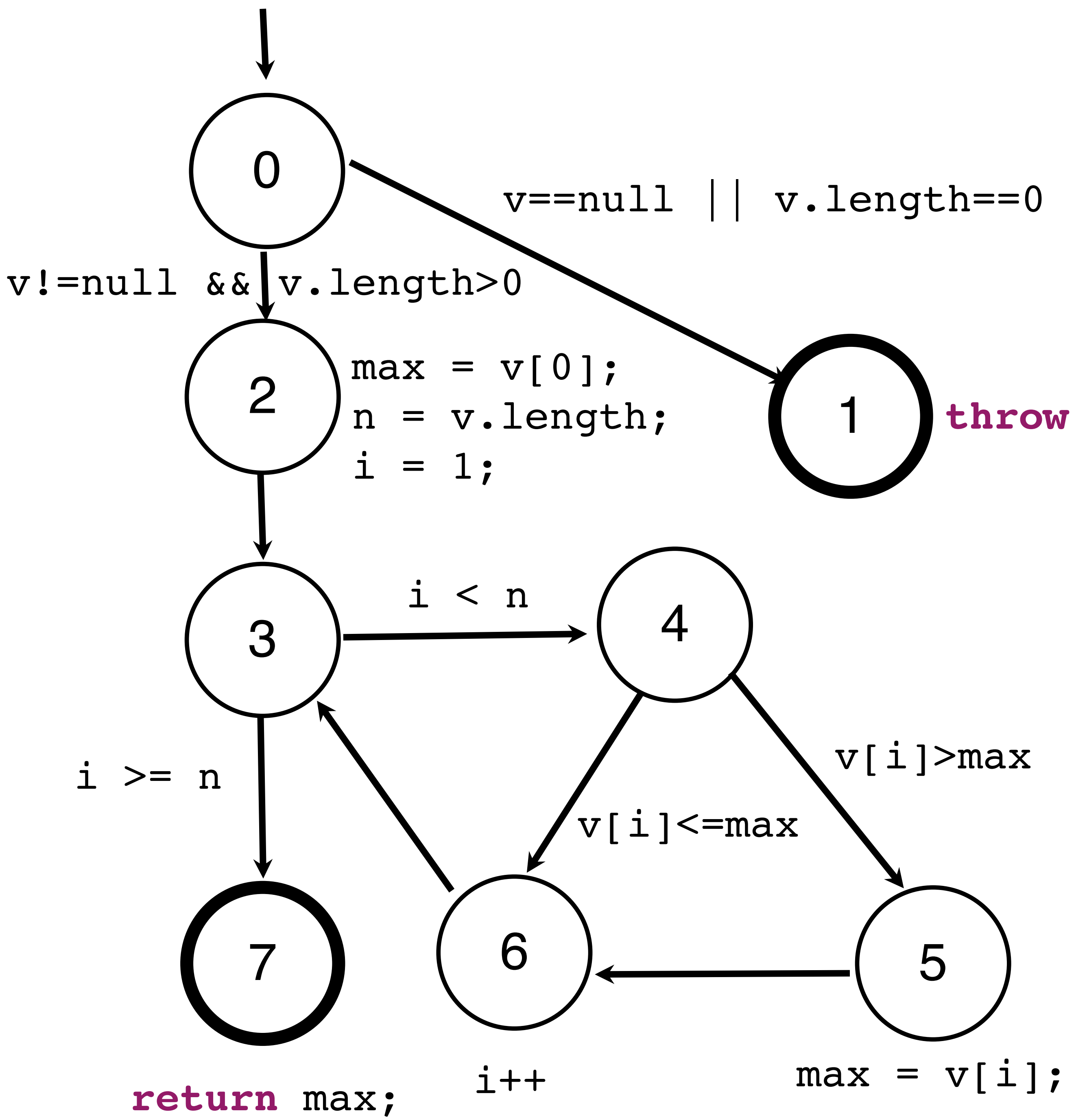
- **Data flow coverage criteria**: based on definitions and uses of data.

# Example

```java
public static int max(int[] v) {
  if (v == null || v.length == 0)
    throw new IllegalArgumentException();
  int max = v[0];
  int n = v.length;
  for (int i = 1; i < n; i++)
    if (v[i] > max)
      max = v[i];
  return max;
}
```

Let us derive the CFG for this method and compute the definitions/uses for each variable at each node and edge.

# Example: definitions and uses in max




| nodes & edges: l | def(l) | use(l) |
|---|---|---|
| 0 | {v} | {} |
| (0,1), (0,2) | {} | {v} |
| 2 | {max, n, i} | {v} |
| (2,3) | {} | {} |
| 3 | {} | {} |
| (3,4), (3,7) | {} | {i, n} |
| 4 | {} | {} |
| (4,5), (4,6) | {} | {v, i, max} |
| 5 | {max} | {v, i} |
| 6 | {i} | {i} |
| (5,6), (6,3) | {} | {} |
| 7 | {} | {max} |

All method parameters are defined at the CFG entry node.

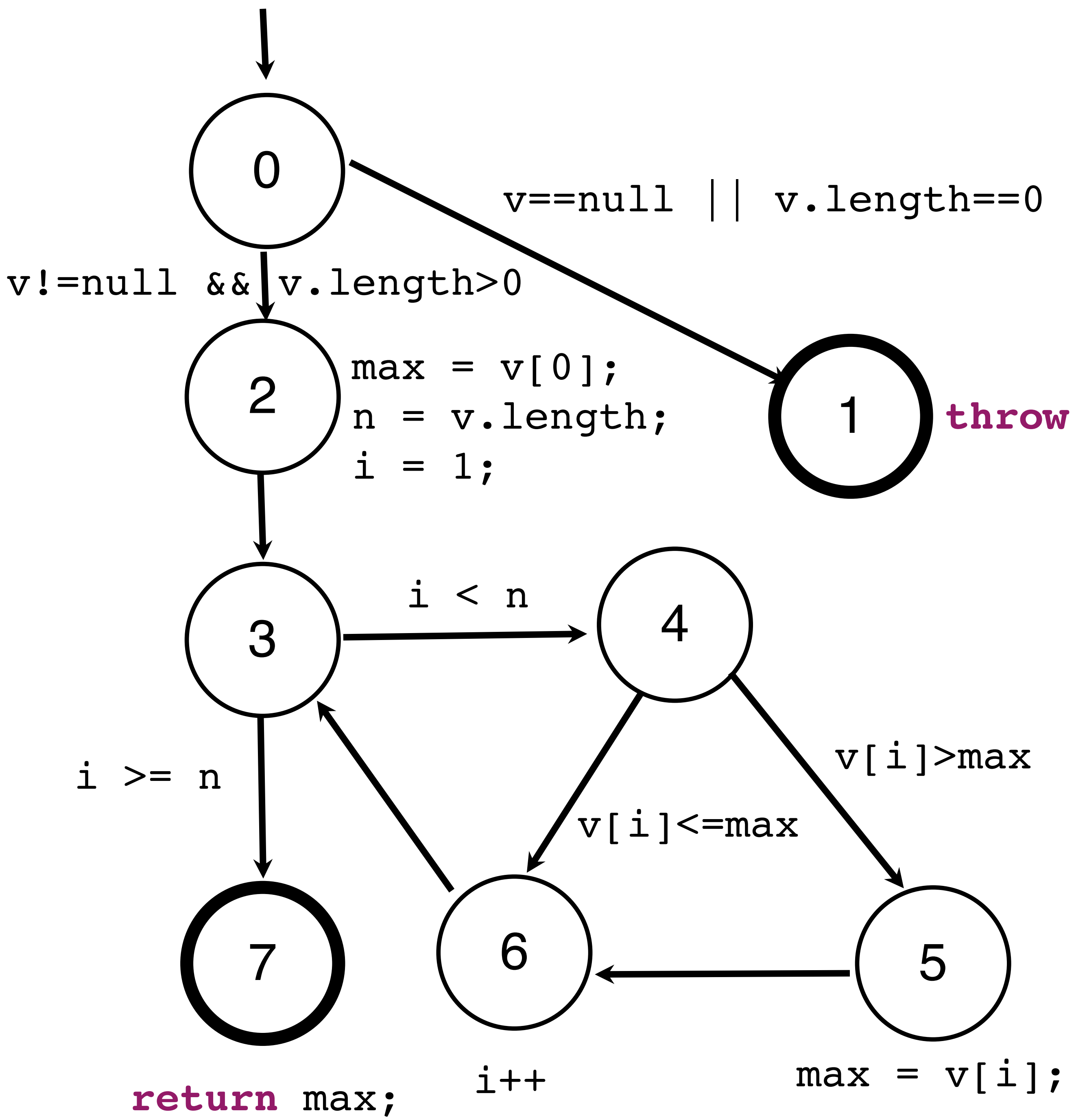
# Def-clear paths

- For simplicity we will consider only graphs where edges have no definitions (all definitions occur in nodes).
  - This is the case with CFG but not with Finite State Machines.
- A def of a variable may or may not reach a particular use:
  - There is no path from the def to the use, or
  - The value of the variable is changed by another def before it reaches the use
- A path from location (node or edge) l to location l’ is **def-clear** with respect to variable v if v is not in def(n) or def(e), for all nodes n or edges e in the path (except in l and l’)

# du-path and def-path set

- A **du-path** with respect to a variable $v$ is a simple (no inner-loops), def-clear path, from a node $n$ to a node $n'$ such that $v \in def(n)$ and $v \in use(n')$
- Note:
  - du-path are always associated with a variable
  - there may be intervening uses on the path
- Test criteria for data flow are defined as sets of du-paths
- We first categorise du-paths in different groups
- The first grouping is according to definitions
- A **def-path set** $du(n,v)$ is the set of du-paths wrt to variable $v$ that originate in node $n$

# Example: def-path sets in `max`

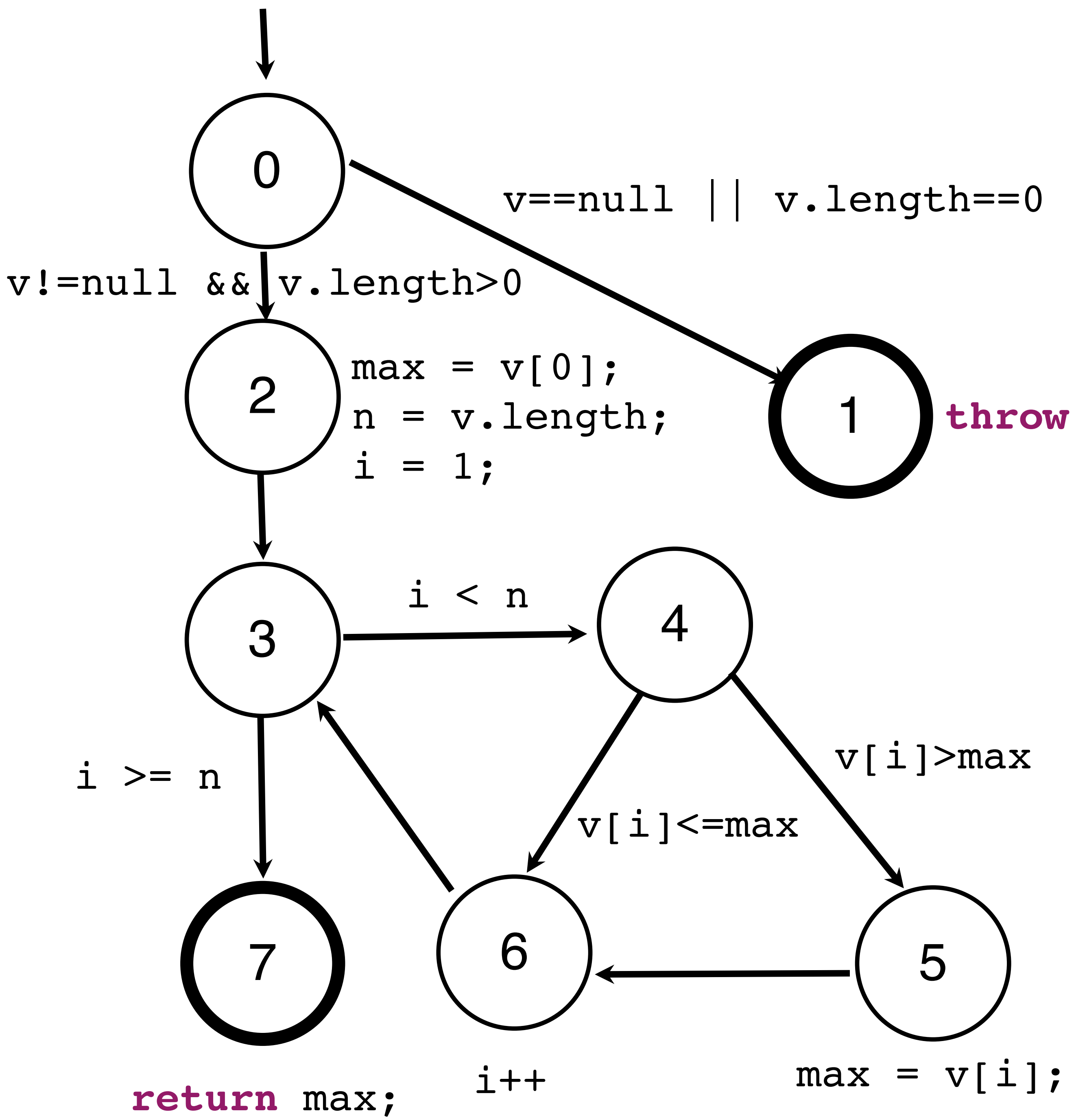



| n | v | du(n,v) |
|---|---|---|
| 0 | v | {[0,1], [0,2], [0,2,3,4,5], [0,2,3,4,6]} |
| 2 | n | {[2,3,4], [2,3,7]} |
| 2 | i | {[2,3,4], [2,3,7], [2,3,4,5], [2,3,4,6], [2,3,4,5,6]} |
| 6 | | {[6,3,4], [6,3,7], [6,3,4,6], [6,3,4,5], [6,3,4,5,6]} |
| 2 | max | {[2,3,7], [2,3,4,5], [2,3,4,6]} |
| 5 | | {[5,6,3,7], [5,6,3,4,5]} |

# def-pair set

- The second grouping is according to pairs of def and uses
- Consider all du-paths wrt a given variable that are defined in one node and used in another (possibly identical) node
- A **def-pair set** du(n,n′,v) is the set of du-paths wrt to variable v that originate in node n and end in node n′
- Collects all simple ways to get from a given definition to a given use
  - $du(n,v) = \bigcup_{n'} du(n,n'v)$

# Example: def-pair sets in `max`




| n | v | du(n,v) |
|---|---|---|
| 2 | i | {[2,3,4], [2,3,7], [2,3,4,5], [2,3,4,6], [2,3,4,5,6]} |

| n | n’ | v | du(n,n’,v) |
|---|---|---|---|
| 2 | 4 | i | {[2,3,4]} |
| | 5 | | {[2,3,4,5]} |
| | 6 | | {[2,3,4,6], [2,3,4,5,6]} |
| | 7 | | {[2,3,7]} |

# Data flow coverage criteria

**All-Defs Coverage (ADC)**

Each def reaches **at least one use**

for each *def-path* set S=du(n,v), TR contains *at least one* path in S

**All-Uses Coverage (AUC)**

Each def reaches **all possible uses**

For each *def-pair* set S=du(n,n',v), TR contains *at least one* path in S

AUC subsumes ADC

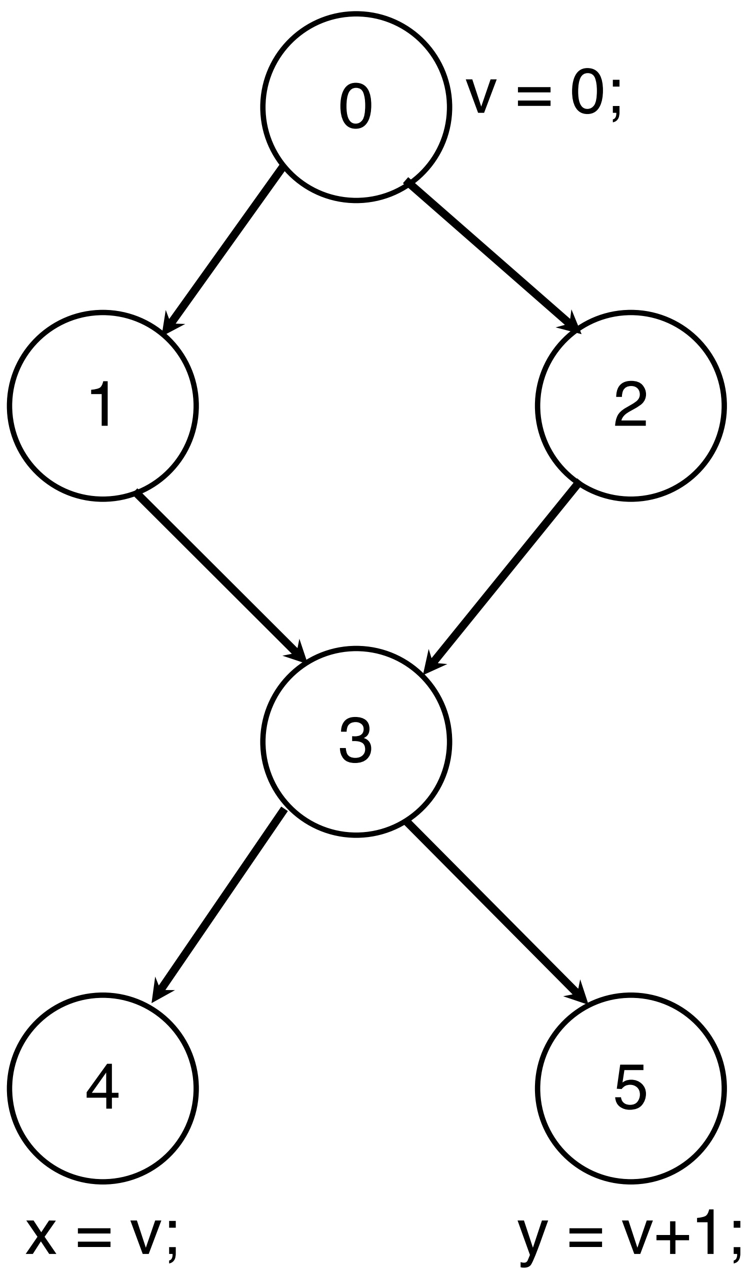
**All-Du-Paths Coverage (ADUPC)**

Each def reaches **all possible du-paths**

For each *def-pair* set S, TR contains *every* path in S

ADUPC subsumes AUC

# Example of the difference among the three criteria

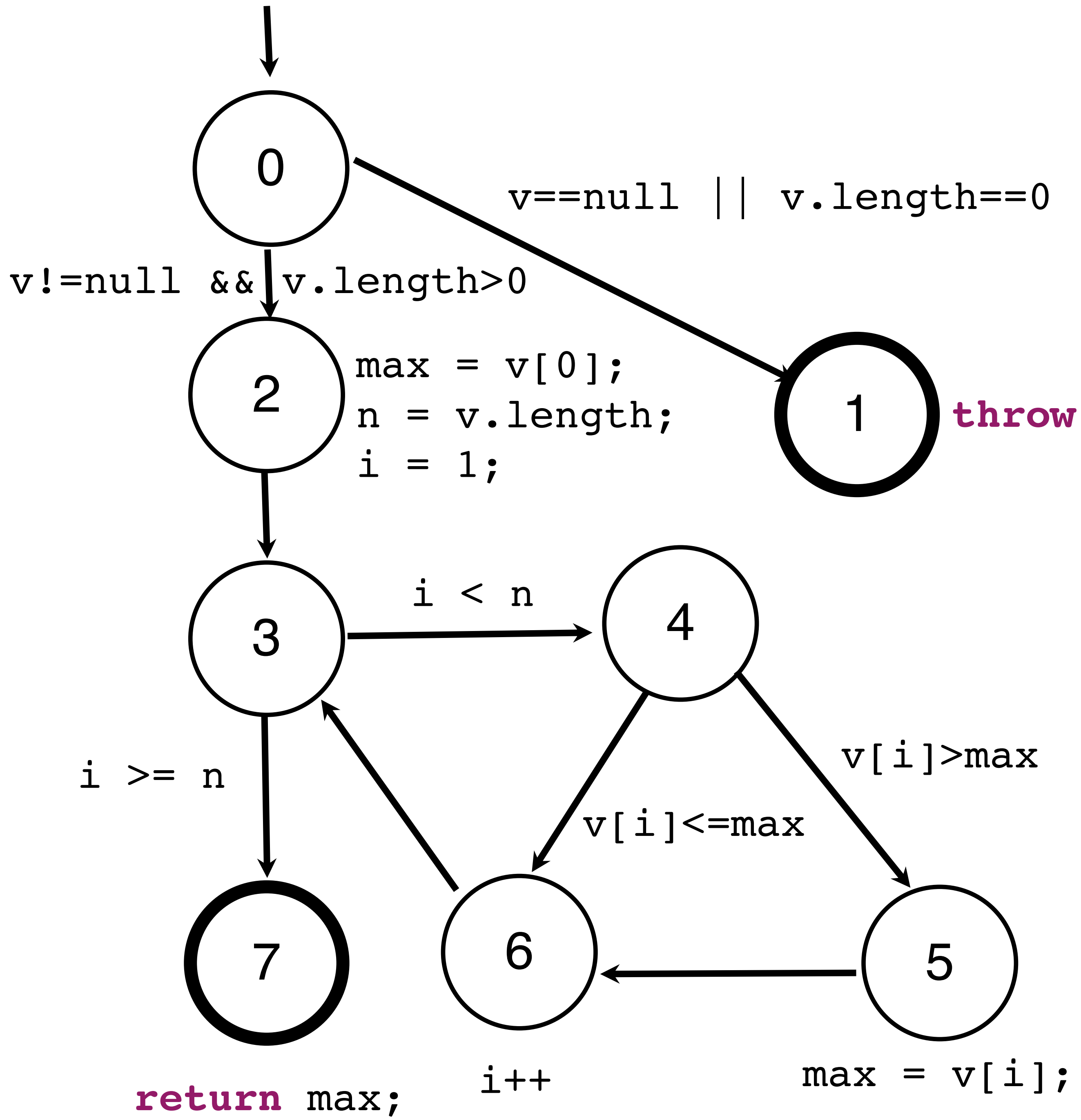



Assume that the only definitions and uses of **v** are as shown: **def at node 0, uses at nodes 4 and 5**.

**ADC** - *only one du-path* needs to be covered for the def. of **v**, for instance {[0,1,3,4]}.

**AUC -** we need to cover *one du-path per use* of **v**, i.e., one du-path ending at node **4** and another one ending at node **5**, for instance {[0,1,3,4], [0,1,3,5]}.

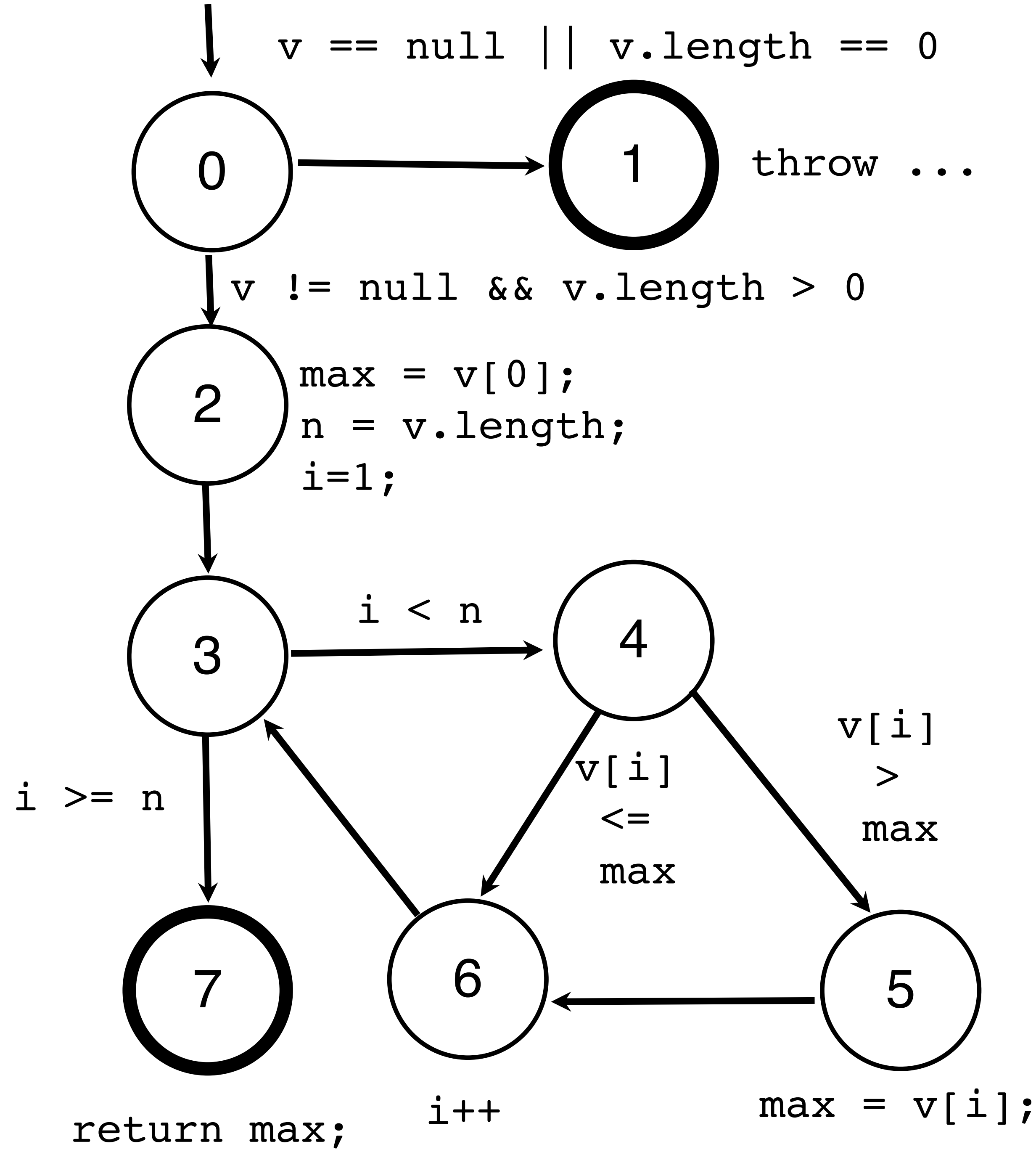
**ADUPC -** *all du-paths* must be covered: {[0,1,3,4], [0,1,3,5], [0,2,3,4], [0,2,3,5]}




| n | v | du(n,v) |
|---|---|---|
| 0 | v | [0,1] **[0,2]** **[0,2,3,4,5]** [0,2,3,4,6] |
| 2 | n | **[2,3,4]** [2,3,7] |
| 2 | i | **[2,3,4]** [2,3,7] **[2,3,4,5]** [2,3,4,6] **[2,3,4,5,6]** |
| 6 | i | [6,3,4] **[6,3,7]** [6,3,4,6] [6,3,4,5] [6,3,4,5,6] |
| 2 | max | [2,3,7] **[2,3,4,5]** [2,3,4,6] |
| 5 | max | **[5,6,3,7]** [5,6,3,4,5] |

| Test | (input, expected) | test path |
|---|---|---|
| t1 | ({1,2}, 2) | **[0,2,3,4,5,6,3,7]** |

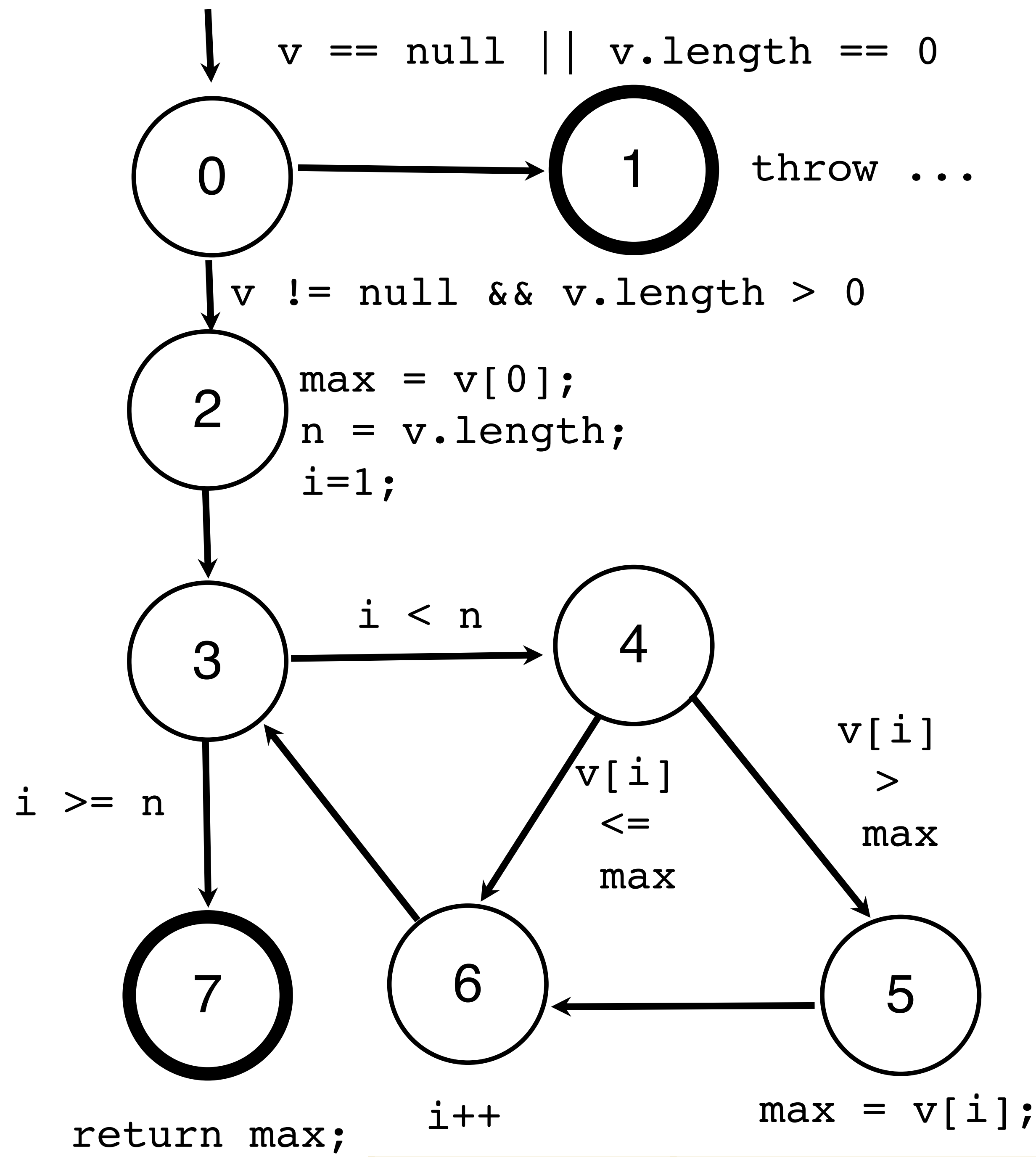
Test t1 enough to satisfy ADC.




| v | n | du(n,v) |
|---|---|---|
| v | 0 | [0,1] [0,2] [0,2,3,4,5] [0,2,3,4,6] |
| n | 2 | [2,3,4] [2,3,7] |
| i | 2 | [2,3,4] [2,3,7] [2,3,4,5] [2,3,4,6] [2,3,4,5,6] |
| | 6 | [6,3,4] [6,3,7] [6,3,4,6] [6,3,4,5] [6,3,4,5,6] |
| max | 2 | [2,3,7] [2,3,4,5] [2,3,4,6] |
| | 5 | [5,6,3,7] [5,6,3,4,5] |

Test set {t1,t2,t3,t4} satisfies AUC, but du-path [6,3,4,6] not covered.

| Test | (input, expected) | test path |
|---|---|---|
| t1 | ({1,2}, 2) | **[0,2,3,4,5,6,3,7]** |
| t2 | (null, IllegalA. ) | **[0,1]** |
| t3 | ({1,0,2,3}, 3) | **[0,2,3,4,6,3,4,5,6,3,4,5,6,3,7]** |
| t4 | ({1}, 1) | **[0,2,3,7]** |



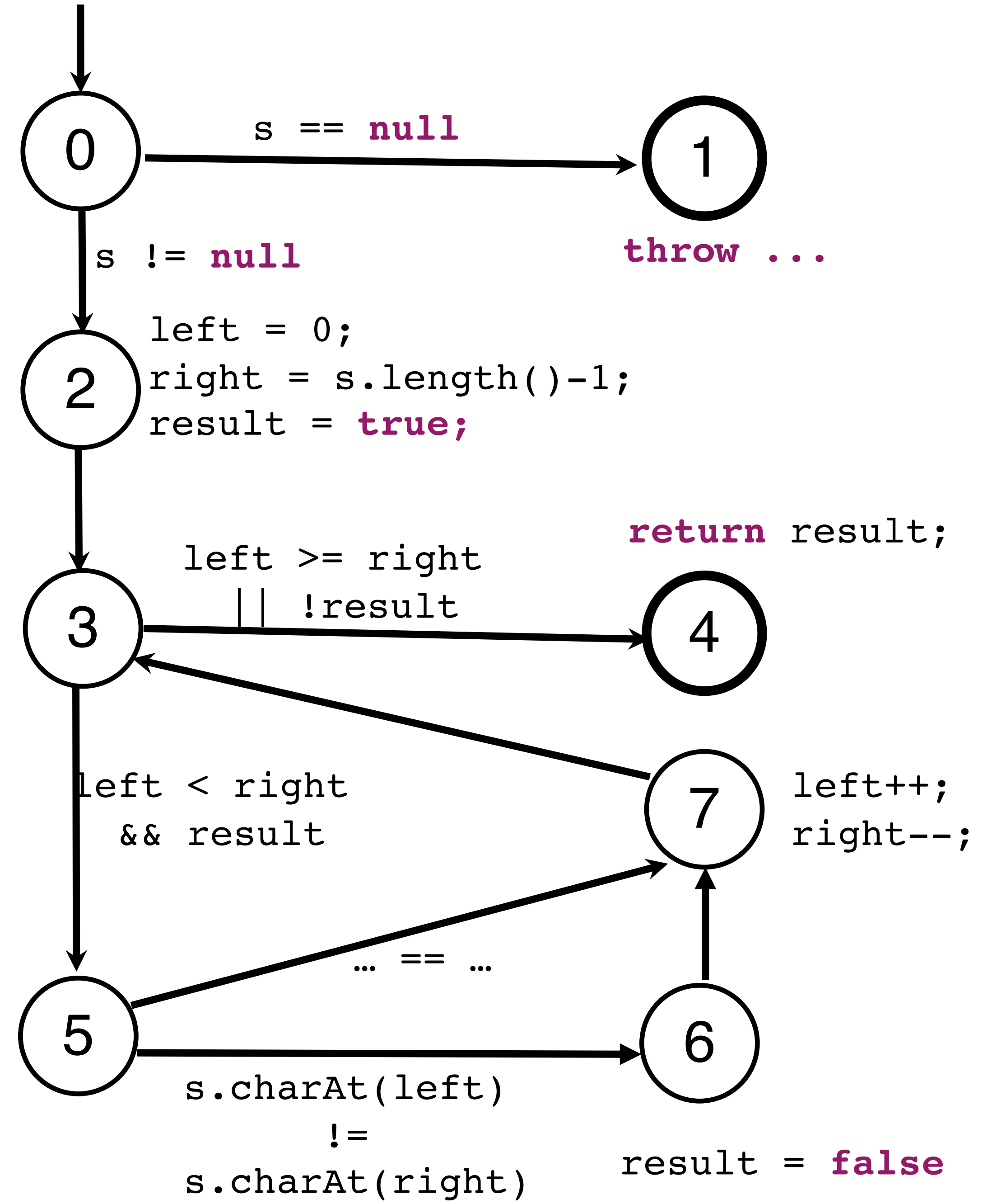

| v | n | du(n,v) |
|---|---|---|
| v | 0 | [0,1] [0,2] [0,2,3,4,5] [0,2,3,4,6] |
| n | 2 | [2,3,4] [2,3,7] |
| i | 2 | [2,3,4] [2,3,7] [2,3,4,5] [2,3,4,6] [2,3,4,5,6] |
| | 6 | [6,3,4] [6,3,7] [6,3,4,6] [6,3,4,5] [6,3,4,5,6] |
| max | 2 | [2,3,7] [2,3,4,5] [2,3,4,6] |
| | 5 | [5,6,3,7] [5,6,3,4,5] |

Test set {t1,t2,t3,t4,t5} satisfies ADUPC.

| Test | (input, expected) | test path |
|---|---|---|
| t1 | ({1,2}, 2) | **[0,2,3,4,5,6,3,7]** |
| t2 | (null, IAE) | **[0,1]** |
| t3 | ({1,0,2,3}, 3) | **[0,2,3,4,6,3,4,5,6,3,4,5,6,3,7]** |
| t4 | ({1}, 1) | **[0,2,3,7]** |
| t5 | ({1,0,0}, 1) | **[0,2,3,4,6,3,4,6,3,7]** |

# Exercise 1

```java
boolean isPalindrome(String s) {
  if (s == null)
    throw new IllegalArgumentException();
  int left = 0;
  int right = s.length() - 1;
  boolean result = true;
  while (left < right && result) {
    if (s.charAt(left) != s.charAt(right))
      result = false;
    left++;
    right--;
  }
  return result;
}
```




1. Write a table with all definitions and uses.
2. Write a table identifying all du-paths.
3. Identify test cases and write corresponding JUnit test methods that satisfy (if possible)
    - **a)** ADC but not AUC, **b)** AUC but not ADUPC, and **c)** ADUPC.

# Exercise 2

```
1 public static int bSearch(int[] array, int value){
2   int left = 0;
3   int right = array.length - 1;
4   while(left <= right) {
5     int middle = (left + right) / 2;
6     if (array[middle] == value)
7       return middle;
8     if (array[middle] < value)
9       left = middle + 1;
10    else
11      right = middle - 1;
12  }
13  return -1;
14 }
```

1. Draw the CFG of bSearch. Don't forget the possible NullPointerException in line 3.
**2.** Identify the definitions and uses (in tabular form as exemplified before).
3. Identify all du-paths table (in tabular form as exemplified before).
**4.** Identify test cases that satisfy (if possible) **a)** ADC, **b)** AUC, and **c)** ADUPC

# Data flow coverage for call sites

- Control connections between method calls are simple and straightforward ☞ not very effective at finding faults
- Data flow connections are usually complex and difficult to analyse ☞ rich source for software faults

- A ***caller*** is a unit that invokes another unit, the ***callee***
- **The instruction that makes the call is the *callsite***
- Idea: ensure that variables defined at caller are appropriately used in callee
- Concentrate on:
  - **last def** of variables just before calls to (and returns from) callee
  - **first use** of variables just after calls to (and returns from) callee

# Data flow couplings

- Consider couplings between caller and callee units:
  - **parameter coupling**, when parameters are passed in caller→callee
  - **return value coupling**, when a return value is passed in callee→caller
  - **shared data coupling**, when there are shared variables by caller & callee
  - **external device coupling**, when two units access the same external media (e.g., a file or a socket)
- For variable x expressing a coupling between caller and callee:
  - **last-def**: the set of nodes that *define* x from which there is a def-clear path to the call (callee) or the return site (caller)
  - **first-use**: the set of nodes that *use* x for which there is a def-clear & use-clear path from the entry point (callee) or call site (caller) to the nodes

# Parameter & return value coupling

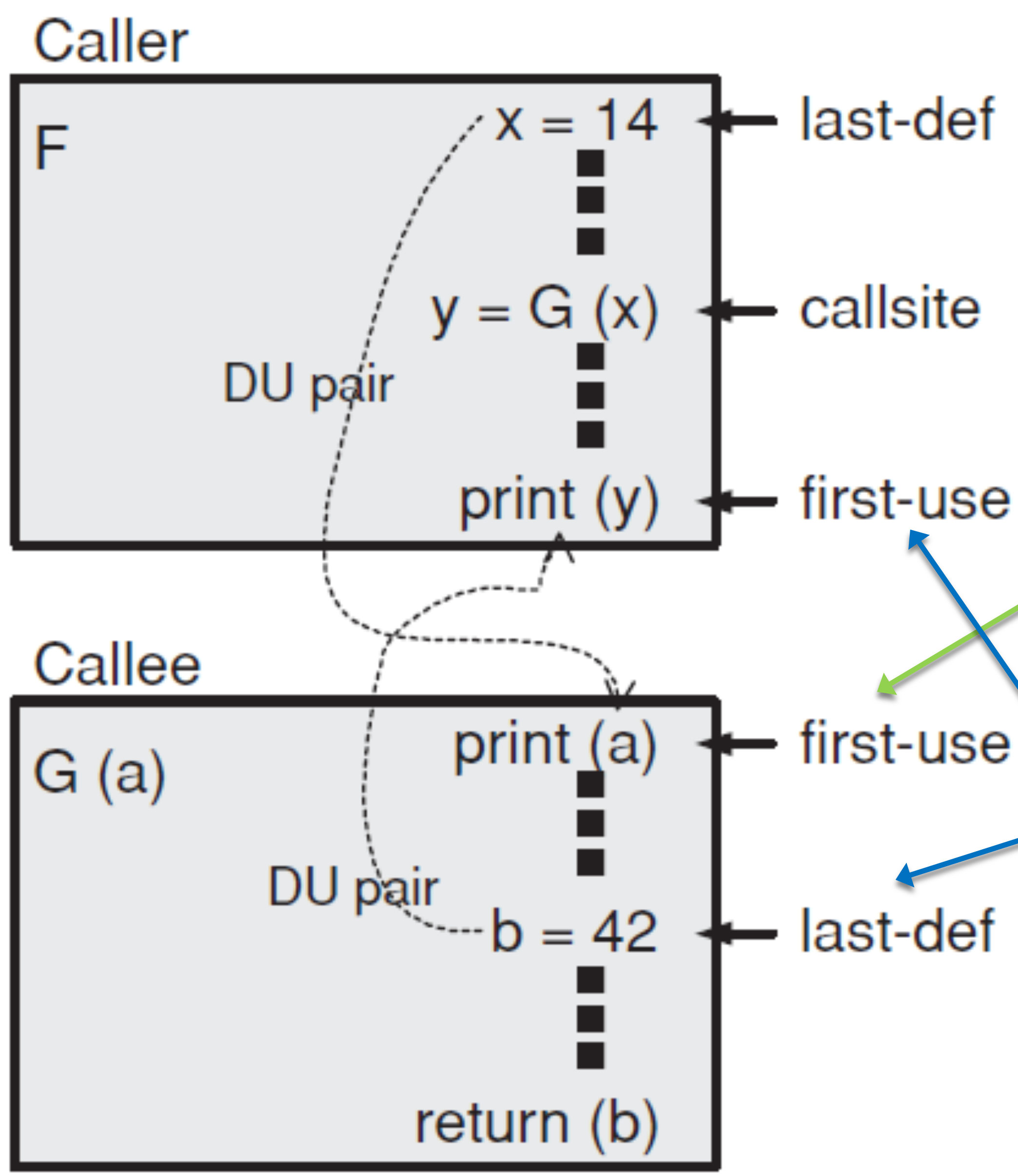
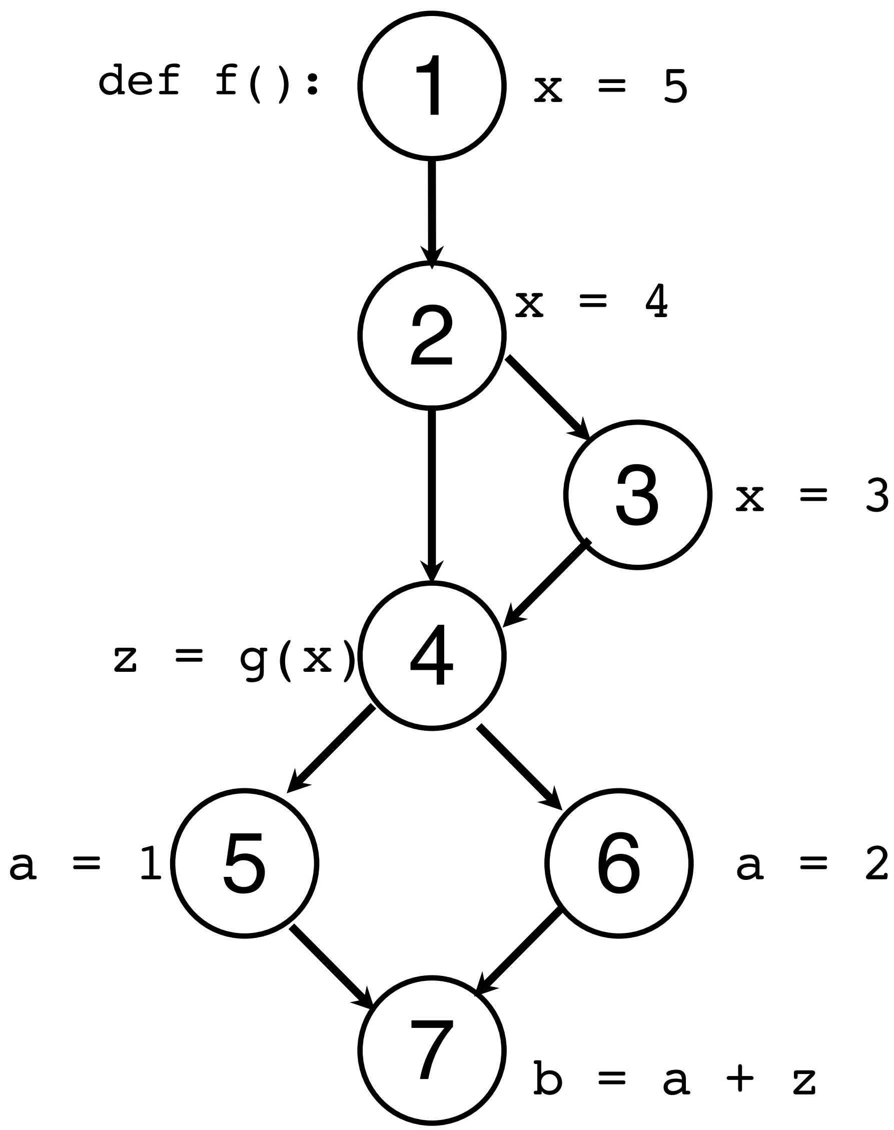


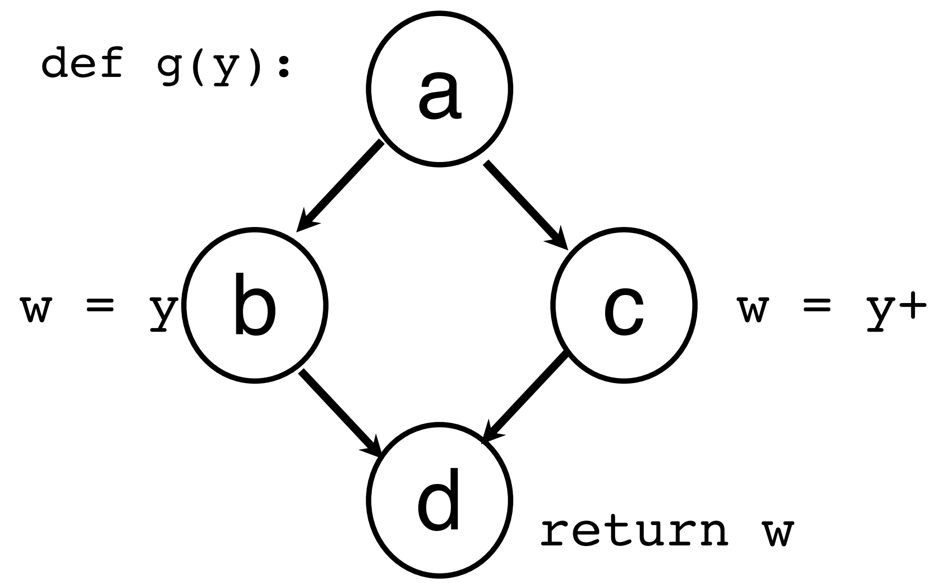

**Figure 2.26.** Coupling du-pairs.

[Amman & Offutt, p. 69]

- **parameter coupling**
  - **last-def of x**: set of locations in caller that last define a call parameter x just before the call site;
  - **first-use of x**: set of locations in callee that first use a parameter x after the entry point;
- **return value coupling**
  - **last-def**: set of locations in callee that last define the return result;
  - **first-use**: set of locations in caller that first use the result of the call after the call site;
- **last-defs** and **first-uses** define coupling du-pairs

# Last-defs and first-uses




- parameter coupling
  - last-def(x) = {2, 3}
  - first-use(y) = {b,c}
- return value coupling
  - last-def(w) = {b,c}
  - first-use(z) = {7}

# Coupling du-paths and coverage criteria

- A **coupling du-path** for x is a path from a last-def of x to a first-use of x.
- Data flow coverage criteria can be applied to coupling du-paths:
  - **All-Coupling-Def coverage** (cf. All-Defs-Coverage): cover at least one coupling du-path for every last-def of x
  - **All-Coupling-Use coverage** (cf. All-Uses-Coverage): cover at least one coupling du-path from every last-def to first-use of x
  - **All-Coupling-Du-Paths** (cf. All-DU-Paths-Coverage): cover every coupling du-path from every last-def to first-use of x

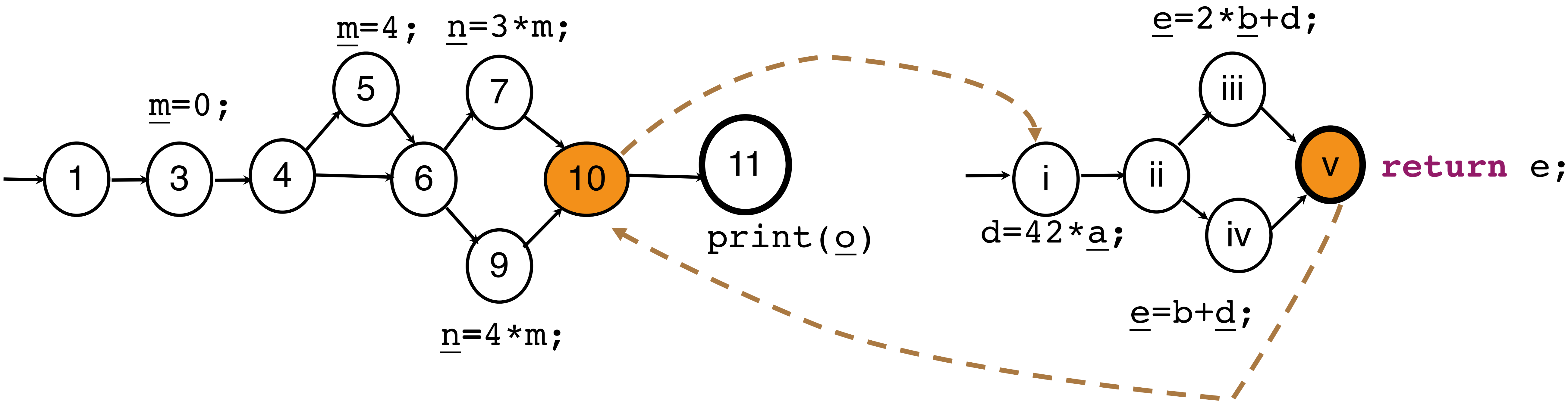
# Example

```java
void trash(int x) {
  int m, n;
  m = 0;
  if (x > 0)
    m = 4;
  if (x > 5)
    n = 3 * m;
  else
    n = 4 * m;
  int o = takeOut(m, n);
  System.out.println(o);
}
```

```java
int takeOut(int a, int b) {
  int d, e;
  d = 42 * a;
  if (a > 0)
    e = 2 * b + d;
  else
    e = b + d;
  return e;
}
```

| last-def | first-use |
|---|---|
| m: {3,5} | a: {i} |
| n: {7,9} | b: {iii,iv} |
| e: {iii,iv} | o: {11} |

expect 8 du-pairs




[Example from A & O, p. 74]

# Example (cont'd)

```
1   void trash(int x) {
2     int m, n;
3     m = 0;
4     if (x > 0)
5       m = 4;
6     if (x > 5)
7       n = 3 * m;
8     else
9       n = 4 * m;
10    int o = takeOut(m, n);
11    System.out.println(o);
12  }
```

```
1   int takeOut(int a, int b) {
2     int d, e;
3     d = 42 * a;
4     if (a > 0)
5       e = 2 * b + d;
6     else
7       e = b + d;
8     return e;
9   }
```

The coupling du-pairs (last-def → first-use) are:

(a) (trash, m, line 3) → (takeOut, a, line 3)

(b) (trash, m, line 5) → (takeOut, a, line 3)

(c) (trash, n, line 7) → (takeOut, b, line 5)

(d) (trash, n, line 7) → (takeOut, b, line 7)

(e) (trash, n, line 9) → (takeOut, b, line 5)

(f) (trash, n, line 9) → (takeOut, b, line 7)

(g) (takeout, e, line 5) → (trash, o, line 11)

(h) (takeout, e, line 7) → (trash, o, line 11)

**trash(0)** covers **(a) (f) (h)**
**trash(1)** covers **(b) (e) (g)**
**trash(6)** covers **(b) (c) (g)**

**(d)** is infeasible since **m** will have value 4 if line 7 of **trash** is executed. In that case, so will **a** in **takeOut**, and line 7 in that method will not be executed.

# Exercise 3

```java
public class RootFinder {
  private static int a, b, c;
  private static int roots;
  private static double r1, r2;
  public static void main(String[] args) {
    if (args.length == 3) {
      a = Integer.parseInt(args[0]);
      b = Integer.parseInt(args[1]);
      c = Integer.parseInt(args[2]);
    } else {
      a = 1;
      b = 2;
      c = 1;
    }
    findRoots();
    System.out.printf("%d solutions\n", roots);
    if (roots == 1)
      System.out.printf("x=%f\n", r1);
    else if (roots == 2)
      System.out.printf("x=%f or x=%f\n", r1, r2);
  }
--> (continued right)
```

```java
  private static void findRoots() {
    int delta = b * b - 4 * a * c;
    if (delta < 0) {
      roots = 0;
      return;
    }
    double x = - (double) b / (2 * a);
    if (delta == 0) {
      roots = 1;
      r1 = x;
    } else {
      double y = Math.sqrt(delta) / (2 * a);
      roots = 2;
      r1 =  x - y;
      r2 =  x + y;
    }
  }
}
```

Consider the call from `main` to `findRoots`. Observe that the data coupling is defined by means of static fields (`a, b, c, roots, r1, and r2`) .

Find all coupling du-pairs for the call. Identify infeasible coupling du-pairs and characterise tests to cover all others.