Introduction

Bugs: Why do we care

- Numerous
- Costly (~ $60B/yr on U.S.)

Recent

- log4shell
  - Java logging allow RCE
- Heartbleed
  - No mem. bound check
- AMZN Prime Day 2018
  - Scalability issue

Prevention

- testing
  - unit
  - integration
- code reviews
- Clean code
- Decrease scope
- Log. IDE
- Defensive pros.
  - Offensive pros. (throw errors at user if criteria not met)
- Reverse point

Prevention only work for 95% ⇒ 5% need mitigation strats.

How to test

- Compilation
  - Warnings
  - Type error
- Run on input
  - Fuzzy testing
  - GUI
  - Static testing

Non-func. requirements need testing. → load/perf. testing

- Scalability
- Availability

Faults, Errors, Failures

1. Failure: observable, unacceptable behavior
2. Fault: incorrect step/defect
   - Bug
   - Not always leading to failure ← non-observable, right ans. still given.
3. Error: incorrect state ∈ fault
   - Not always to failure.
   - Fault doesn't always lead to error: fault executes but state is fine.
   - Always has an expected state

Ex 11

```
int countZeros (int[] x) {
    if (x == null) {
        throw new Exception ();
    }
    int count = 0;
    for (int i=1; i < x.length; i++) {
        if (x[i] == 0) {
            count ++;
        }
    }
    return count;
}
```

1. Fault?

   $i = 1$ ⇒ $i = 0$

2. Input: no fault/error/failure.

   x = null ⇒ Exception is expected.

3. Input: fault, no error/failure.

   No ⇒ we always expect $i=0$ after first iteration

4. Input: fault, error, no fail.

   x s.t. $x[0] \neq 0$

   Error state: PC = first itr, $i = 1$

   Expected: PC = first, $i = 0$

5. Failure:

   x s.t. $x[0] = 0$

   ∴ count is wrong

Ex 11

```
int findLast (int[] x, int y):
    for (int i = x.length-1; i > 0; --i) {
        if (x[i] == y)
            return i
    return -1
```

1. Fault

   $i > 0$ ⇒ $i \geq 0$

2. Input that doesn't exec. fault

   $x = []$, $y =$ ___ ⇒ Never entr for loop

3. Fault, no error/failure.

   Error state ⇒ suppose to check 0 but didn't

   Don't check 1st

   ∴ $x = [3, 4, 5]$, $y = 5$

4. Fault, error, no failure:

   Get all down to 1st elem., but doesn't cause fail

   ∴ $x = [1, 2, 3, 4]$, $y = 10$

Trace the program + identify 1st error state:

① List out all vars.

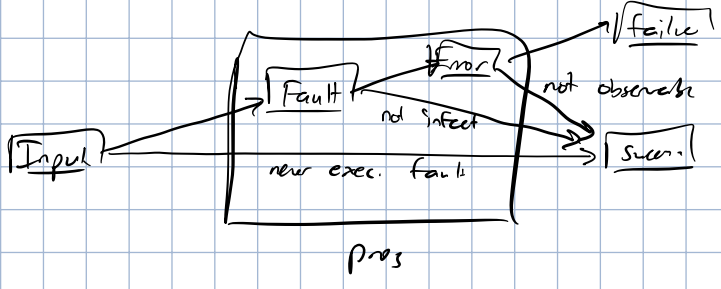
②

PC	var, . . .
0	
1	
2	
⋮	

Exhausive

RIP model of failure: critria for failure:

1. Reachable fault
2. Infection: fault → error
3. Propagate: observe error

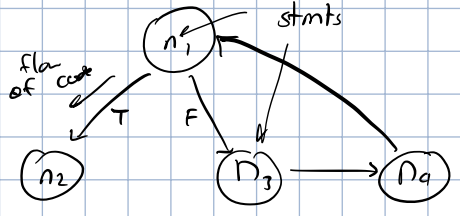


How to address fault

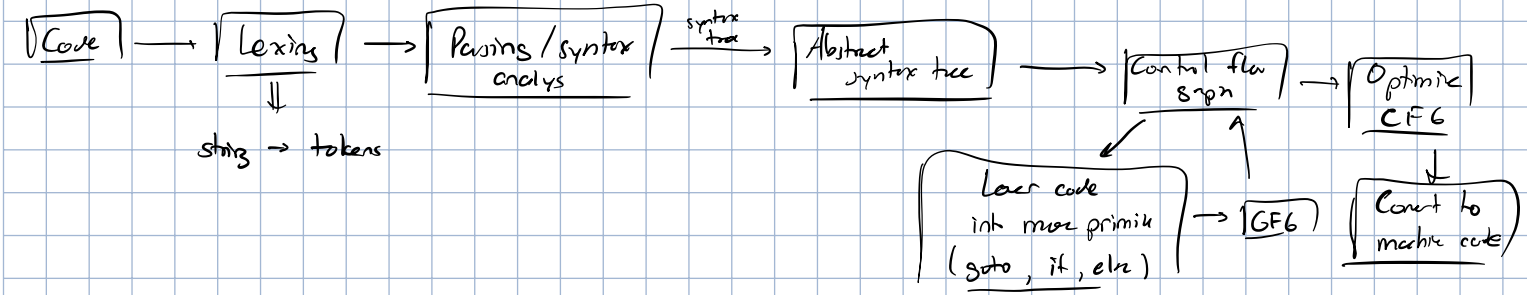
- Avoid
  - Design / code
- Detect
  - Testing (show presence of bug, not absence)
    - Depends on test case
  - Verification (show absence of bugs)
- Tolerate
  - Minimize impact of bug.

CONTROL FLOW GRAPHS

Defn: graph for code



Compilation:



CFG construction

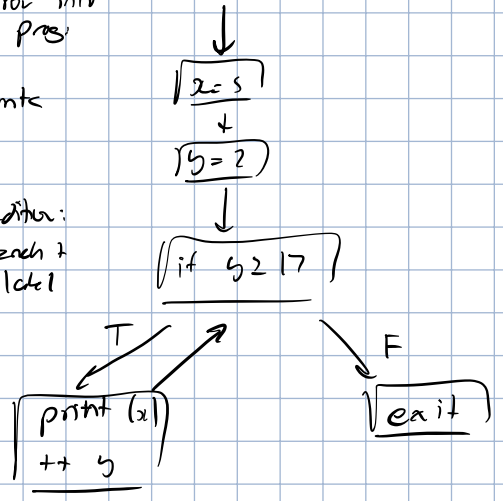
```
    x = 5
    y = 2
L0: if (y >= 17) goto L1
    print(x)
    ++ y
    goto L0
L1: nop
```

For loop

① Turn into pros

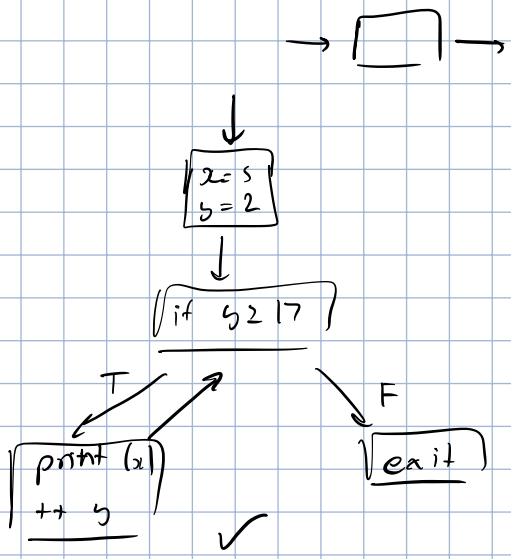
② Stmts

③ Condition: branch + label

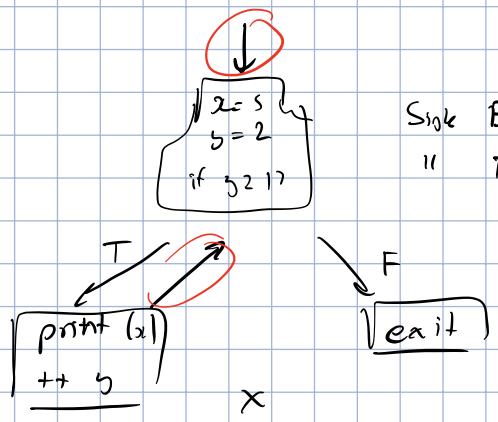


Basic block: simplification:

- Stmts that always execute together
- Follow single exit, single entry policy:
  - ↳ 1 stmt that leads to exit, only 1 stmt of entering block



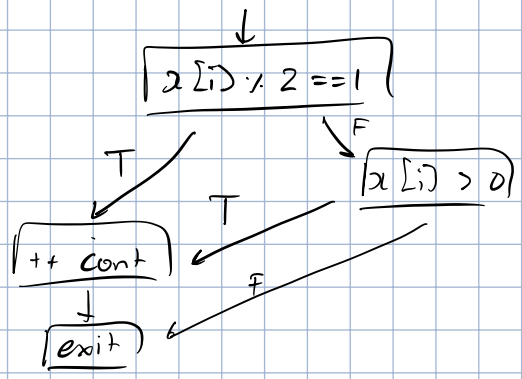
✓



Single Exit: ✓
" Entry: ✗

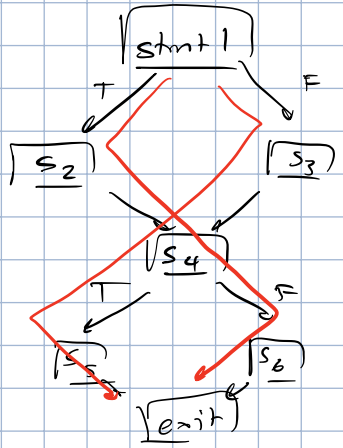
✗

Ex:// `if (a[i] % 2 == 1 || a[i] > 0)`
`++ count`



- Note: designer complex code to make CFG simple
- Don't include func. calls
- Be able to reduce CFG via blocks

CFG paths ≥ actual code paths
↳ Infeasible paths:

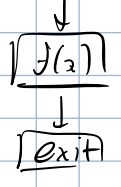


If $S_1 == S_4$
then
$S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_6$ is impossible

Recursive function:
1 CFG ⟺ 1 func

```
f(x):
  f(x)
```

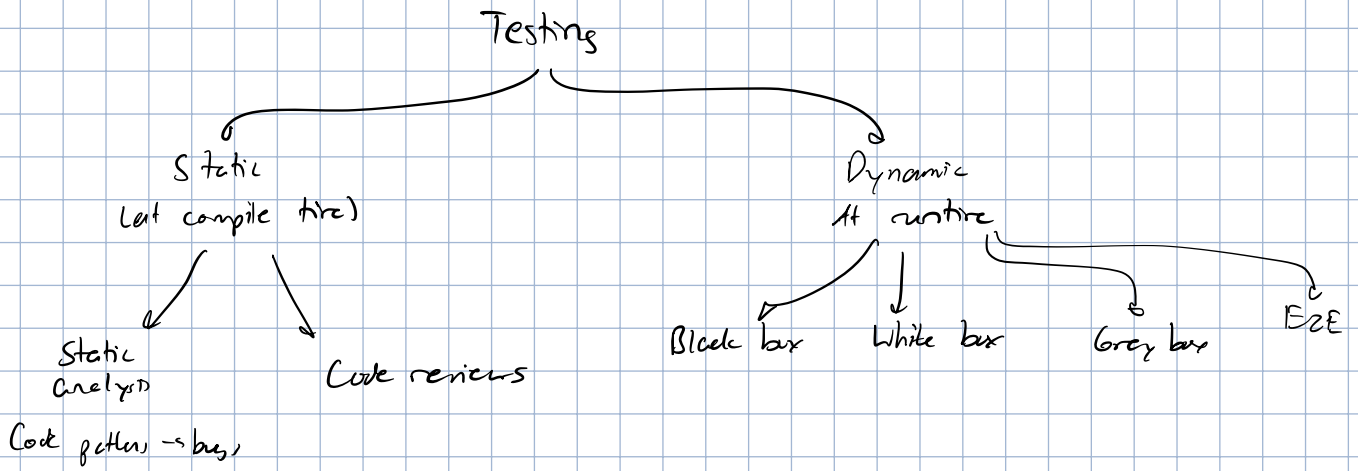
⇒



✓

Structural Coverage

Testing



Ideal: complete testing (all inputs)

- Since impossible → best set of rep. test cases

Test cases: (input, exp. output

Test case: { Prefix ⇒ set up of test case
Test ⇒ runs code + test
Postfix ⇒ clean up + verification

Want a testable system: ability to run test cases

1. Observability
2. Controllability

{test case 1, test case 2, ...} = test set

JUnit example:

```
@RunWith (Junit 4.class)
public class ___ {
    @Before
    pub. void setUp() {...}
    @After
    pub. void takeDown() {...}
    @test
    pub. void test_____ {...}
}
```

Test script:

```
TestClass test1 = new DemoTest
test1. setUp()
     . test__()
     . takeDown()
```

Another obj for another test
↳ Isolation.

Assertion in JUnit: assertTrue(- -), assertFalse(- -)

Test requirements: smth. to satisfy in testing

Coverage

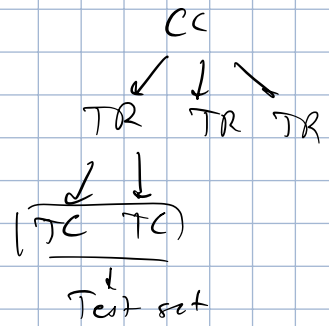
Coverage criterion: generate test req. in systematic way

Ex.// Ice cream machine w/ diff. flav.

Coverage criterion: test all flavors.

TRs = { choc., mint, vanilla, ... }

TC1 TC2 ...



Q: how to tell if 1 test set > another?

Which one satisfies most test req ⇒ coverage.

Test set satisfy CC ⇔ ∀ test req, exists a test in test set that satisfies test req.

↳ Not always possible b/c code

Coverage level:

$$\frac{\#\text{ of satisfied TR}}{\#\text{ of total TR}} \Rightarrow \text{comp. metric}$$

Coverage criterion

Line coverage:

% of src code lines executed by tests

Issue:

L1 fun1; fun2; w...; ....

Statement / node coverage

$$\frac{\#\text{ of exec statements}}{\text{total \# of statements}}$$

or

$$\frac{\#\text{ of nodes in CFG}}{\text{total \# of nodes}}$$

Edge coverage

$$\frac{\#\text{ of edges}}{\text{total \# of edges}}$$ } CFG

Subsumes: C_1 subsumes C_2 iff all tests in C_2 satisfy C_1 as well

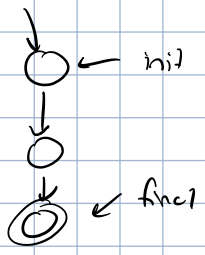
↳ C_1 is stronger than C_2 ⇒ find more bugs.

Graph Coverage

Graphs

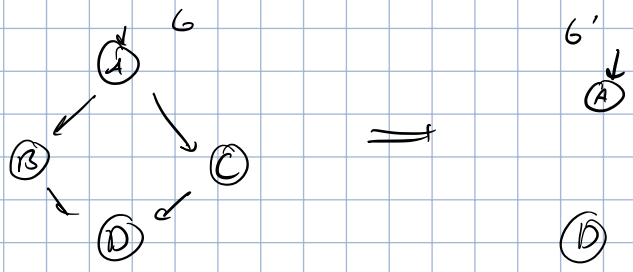
Defn of graph:

1. Nodes
   - N_0: initial $\neq \emptyset$
   - N_f: final $\neq \emptyset$
2. Edges (directed)

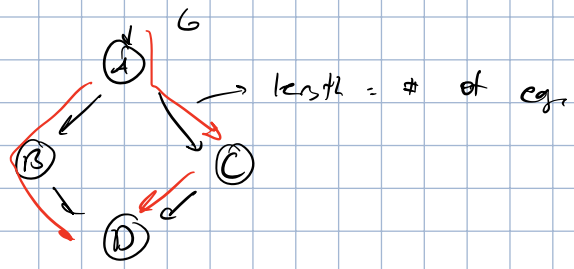


Subgraph: subset of N_0, N_f of graph

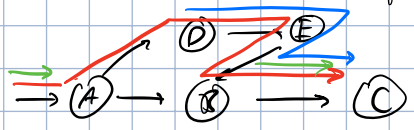
↳ Edges: all edges in original graph that connect subgraph nodes.



Path: set of nodes connected by edges



- Sub-path: subset of path that is still a path



ξ : invalid ξ : valid

Test path: starts at N_0, ends at N_f

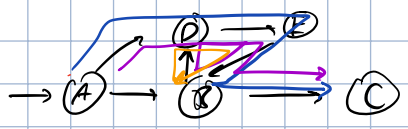
Reachability: reach(x) ← node, edge, combo

- Subgraph that can be reached by x (in that path) } Syntactically reachable

Semantically reachable ⇒ nodes that can be reached in code

SESE graphs: $|N_0| = |N_f| = 1$

Simple path: no internal cycles, except entire path



2: simple path 2: non-simple path

2: simple path

Prime path: maximal simple path ⇒ not a subpath

① All simple paths of length 0

② Grow by 1 if possible for all

③ Kill all paths in prev level that could grow

④ If ending at end node → !. Cycle → *

⑤ Remove all existing subpaths

Graphs + Testing

Test case (T) ↔ test paths

↑ $path_6(T)$

CFG

Deterministic

1 test case → 1 path

Non-det

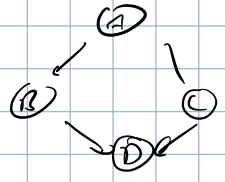
1 test case could have multiple paths

→ Dependence on memory / randomness

Node coverage:

All TR have requirement to reach every node reachable by N_0

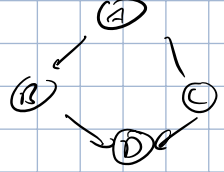
↳ Every reachable node:



→ $TR = \{B, C, D\}$

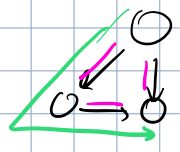
Edge coverage:

$TR = \{$ all paths of length 1 $\}$



$TR = \{AC, AB, BD, CD\}$

Diff. if

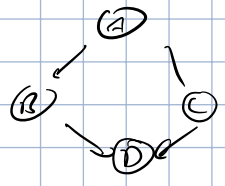


2: node

2: edge

Edge-pair cov:

TR = { all paths of length 2 }



TR = { ABD, ACD }

Prime path cov. : TR contains all prime path

↳ Issue: infeasible prime paths ⇐ feasible subpaths

Complete path cov. : cover all paths

↳ Issue: not possible if loop.