A Testing Exercise:

(From Glenford Myers: The Art of Software Testing)

A program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isoceles, or equilateral.

On a sheet of paper, write a set of test cases (i.e., specific sets of data) that you feel would adequately test this program.

Basic Testing Guidelines

- A test case has two parts:
    1. Description of input data
    2. Precise description of correct output for that input
- A programmer should avoid testing his or her programs.
- A programming organization should not test its own programs.
- The results of each test should be thoroughly inspected (lots of errors are missed).
- Test cases must be written for invalid and unexpected as well as valid and expected input conditions.

Basic Testing Guidelines (2)

- Examining a program to see if it does not do what it is supposed to do is only half the battle. The other half is seeing whether the program does what it is not supposed to do (i.e., must examine for unintended function and side effects.

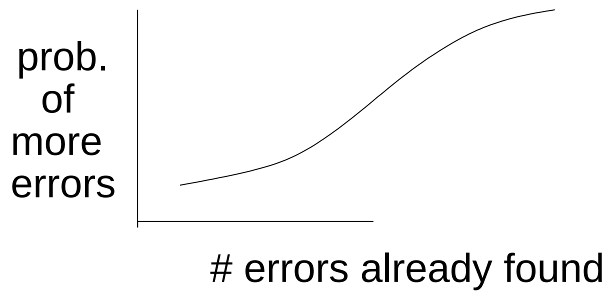
- Avoid throw-away test cases unless the program is a throw-away program.

  Test cases are a valuable investment -- regression testing.

- Do not plan a testing effort under the tacit assumption that no errors will be found.

Basic Testing Guidelines (3)

- The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in it.



- Testing is an extremely creative and challenging task.
  - Exceeds creativity required in designing program.
  - Don't put your worst or newest people here.

Building Assurance (Confidence)

- Dynamic Analysis
- Static Analysis
- Quality Assurance (conformance to standards)
- V&V of non-software lifecycle products (e.g., user manual)
- Acceptance (user) testing

Dynamic Analysis

- Testing
  - Black Box
  - White Box
- Monitoring run-time behavior
- Automated test case generation
- Coverage analysis
- Assertions

Black Box Testing

Test data derived solely from specification (i.e., without knowledge of internal structure of program).

- Need to test every possible input

```
x := y * 2
if x = 5 then y := 3
```

(since black box, only way to be sure to detect this is to try every input condition)

  - Valid inputs up to max size of machine (not astronomical)
  - Also all invalid input (e.g., testing Ada compiler requires all valid and invalid programs)
  - If program has ‘‘memory’’, need to test all possible unique valid and invalid sequences.

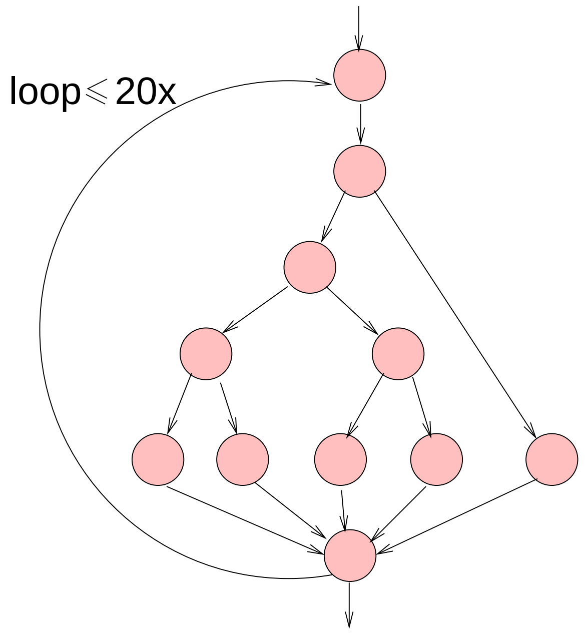
- So for most programs, exhaustive input testing is impractical.

White Box Testing

Derive test data by examining program's logic.

Exhaustic path testing: Two flaws

1) Number of unique paths through program is astronomical.



(control-flow graph)

$5^{20} + 5^{19} + 5^{18} + \ldots + 5 = 10^{14}$

$= 100$ trillion

If could develop/execute/verify one test case every five minutes = 1 billion years

If had magic test processor that could develop/execute/evaluate one test per msec = 3170 years.

White Box Testing (con't)

2) Could test every path and program may still have errors!

- Does not guarantee program matches specification,
  i.e., wrong program.

- Missing paths: would not detect absence of necessary paths

- Could still have data-sensitivity errors.

  e.g. program has to compare two numbers for convergence

  ```
  if (A - B) < epsilon ...
  ```

  is wrong because should compare to abs(A - B)

  Detection of this error dependent on values used for A and B and would not necessarily be found by executing every path through program.

Static Analysis

- Syntax checks
- Look for error-prone constructions
  (enforce standards)
- Program structure checks

  Generate graphs and look for structural flaws
- Module interface checks

  Detect inconsistencies in declarations of data structures and improper linkages between modules
- Human Reviews

  Checklists (inspections)

  Walkthroughs (reviews)

Static Analysis (con't)

- Event sequence checking

  Compare event sequences in program with specification of legal sequences

- Symbolic execution

```
A := X + 5
B := 2 * A
If B>0 then C := |B| - 1
        else  C := |B| + 1
```

```
B = 2 * (X + 5)
if 2 * (X + 5) > 0
     then |2 * (X + 5)| - 1
     else  |2 * (X + 5)| + 1
```

- Formal verification

  Use theorem proving methods to show equivalence of code and a formal specification of required behavior.