MUTATION TESTING

a type of fault-based testing

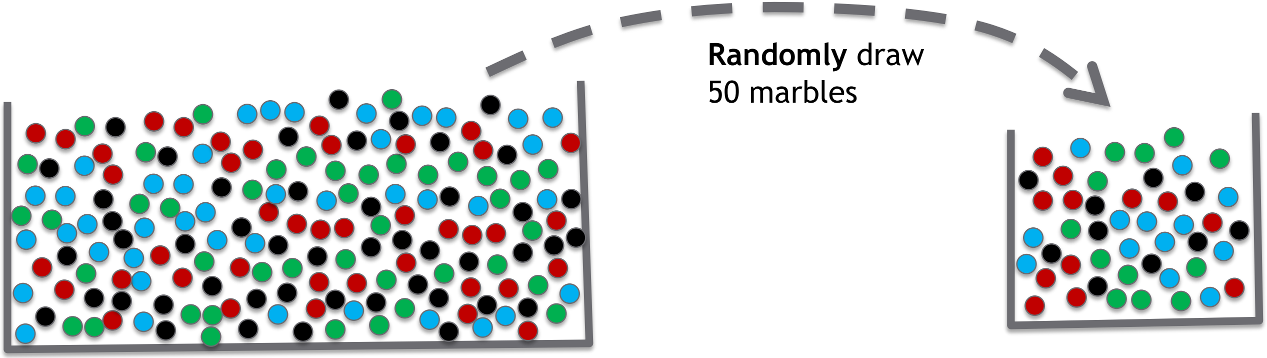
… using bugs to find bugs …

Motivating example: let's count marbles ... a lot of marbles

- Suppose we have a big bowl of marbles (several hundreds of different colors, except black)
- How can we estimate how many?
  - I don't want to count every marble individually
  - I have a bag of 100 black marbles of the same size
  - What if I mix them in?

Estimating marbles

- I mix 100 black marbles into the big bowl: stir well ...
- I draw out 50 marbles at random
- 10 of them are black
- How many colored marbles were in the big bowl to begin with?



Population: 100 black + ? colored

Small sample: 10 black, 40 colored

Estimating marbles

- I mix 100 black marbles into the big bowl
  - stir well …
- I draw out 50 marbles at random (this is called a sample)
- 10 of them are black

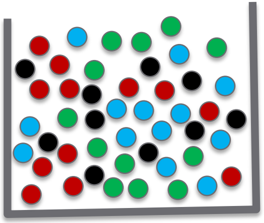
- How many colored marbles were in the big bowl to begin with?

$$\text{P[drawing a black marble]} = \overset{\text{sample}}{10/50} \cong \overset{\text{population}}{100/(100 + N)}$$

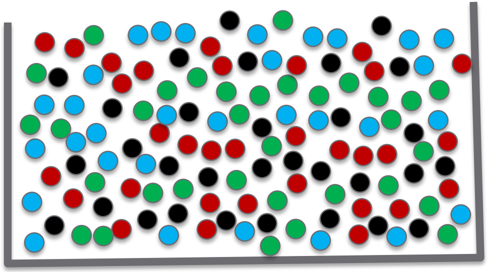
If truly random, should be same in the population

Solve for N → 400

Example of estimating a population from a small sample using a technique called capture-recapture



Random sample: 10 black + 40 colored



Population: 100 black + ? colored

Another example

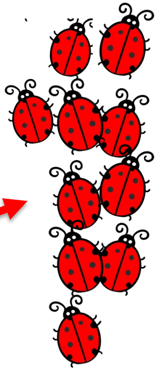
Estimating wildlife population

- Capture 100 impalas in ecological area
- Tag them and release them back
- Capture 200 more impalas randomly, count tagged and untagged
  - 10 tagged (re-captured) and 190 untagged
- How many impalas in the area?

  $10/200 = 100/N \rightarrow N = 2000$ impalas

Estimating test suite and code quality

- Now, instead of impalas, I have a program with bugs
- I add (*seed*) 100 new bugs deliberately
  - *assume they are similar to real bugs*
- I run my test suite on the program with seeded bugs ...
  - the tests fail on 10 of the seeded bugs
  - plus reveal 5 other native bugs
- How effective was my test suite?
- What can I say about remaining bugs?



Estimating test suite and code quality

- Found 10 out of 100 seeded bugs
- Plus 5 native bugs

Test suite was 10% (10/100) effective in finding seeded bugs

$$\frac{\#\text{ Seeded Bugs Found}}{\#\text{ Native Bugs Found}} \cong \frac{\text{Total }\#\text{ Seeded Bugs}}{\text{Total }\#\text{ Native Bugs}}$$

45 x

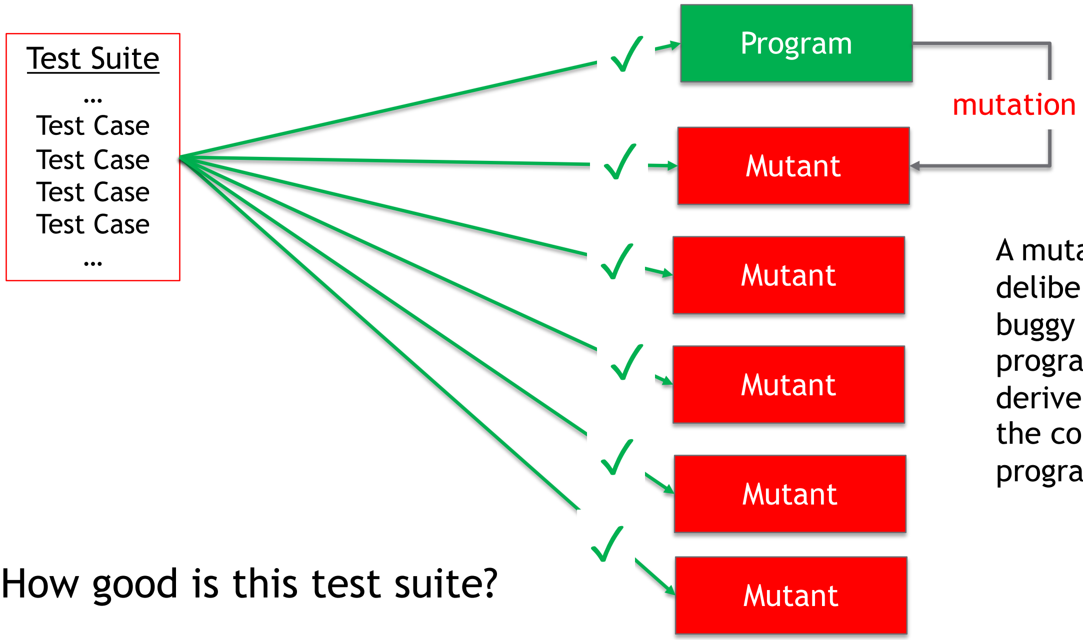
10/5 = 100/N → *N* = 50 Total # Native Bugs

45 Native Bugs could be remaining ← just *an estimate*

We can use the idea to assess the effectiveness of a test suite!

We can also use seeded faults to estimate native (actual) faults, or external code quality!

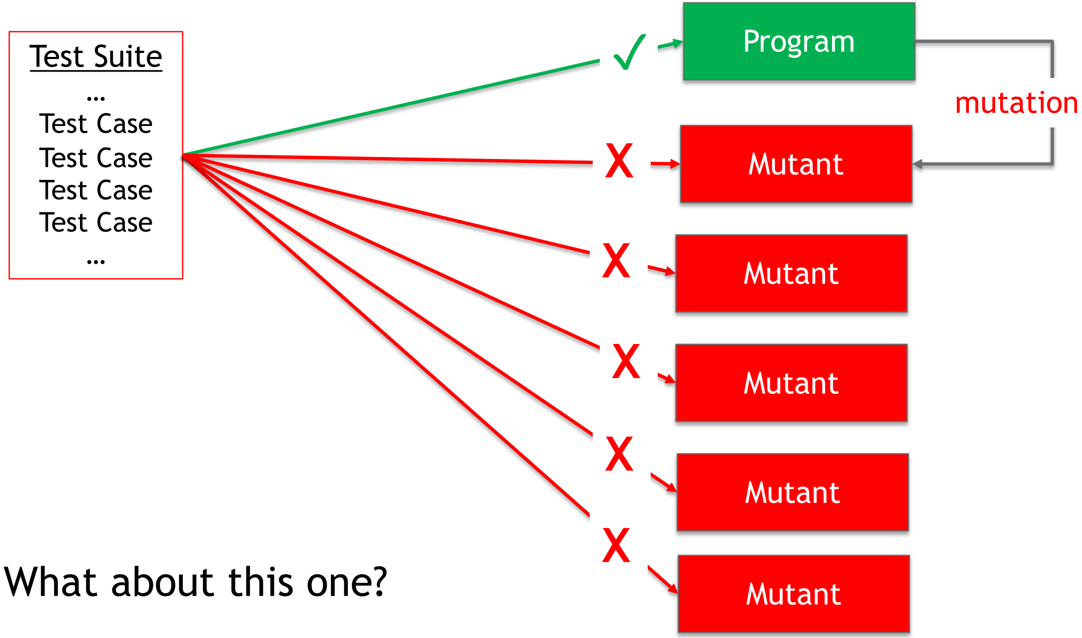
And this brings us to mutation testing…



A mutant is a deliberately buggy program derived from the correct program!

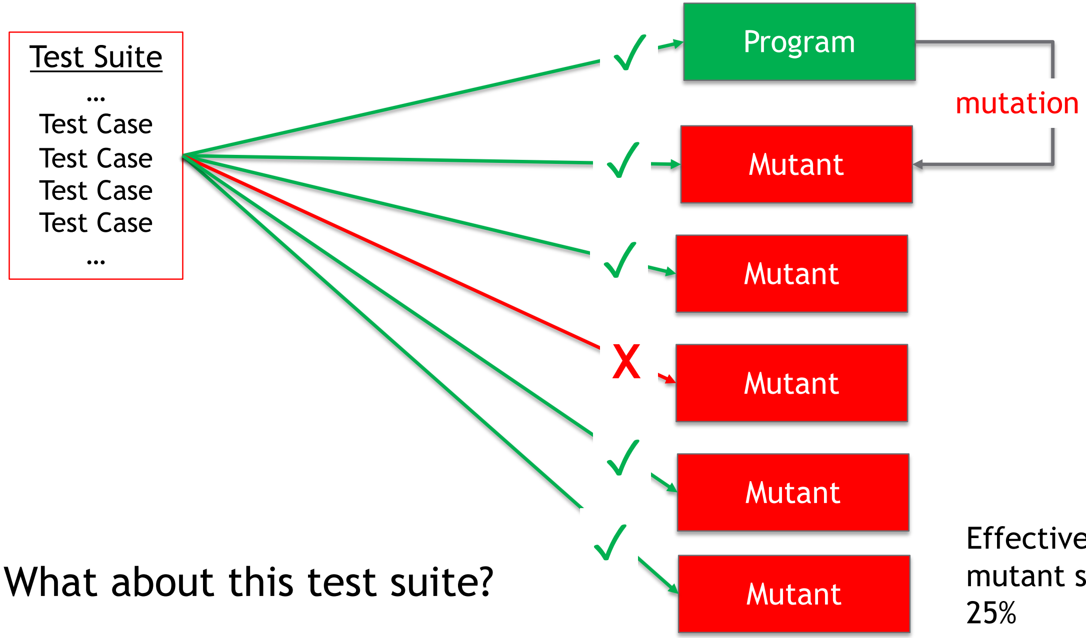
How good is this test suite?

And this brings us to mutation testing…



What about this one?

And this brings us to mutation testing…



What about this test suite?

Effectiveness on the mutant set: 1/5 = 25%

Mutation testing…

is judging the effectiveness of a test suite in finding real faults, by measuring how well it finds seeded, fake faults (mutants)

A *mutant* is a copy of a program with a *mutation*

- A *mutation* is a syntactic change (hopefully, seeding a bug)
  - Example: change “i < 0” to “i <= 0”

- Run test suite on all the mutant programs: one mutation per mutant
- A mutant is *killed* if it fails on at least one test case

- The more mutants are killed, the more effective test suite would be in finding real bugs

What do I need to believe?

- Mutation testing uses seeded faults as black marbles (or tagged impalas)
- Does it make sense? What must I assume?

  What must be true of black marbles, if they are to be useful in counting a bowl of colored marbles?

  What must be true of mutant, if they are to be useful in detecting real faults?

Mutation testing makes two assumptions: one about <u>programs</u>…

- **Competent programmer hypothesis (about <u>programs</u>)**
  - Programs are nearly correct (reasonable assumption)

Real faults are small variations from the correct program
→ *Mutants are reasonable models of real buggy programs*

Mutation testing makes two assumptions: and another about <u>tests</u>…

- **Coupling effect hypothesis (about <u>tests</u>)**
  – Tests that find simple faults also find more complex faults

Complex (e.g., missing logic) and simple faults (e.g., mutation) have similar consequences

→ *Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is also likely to be good at failing with real faults*

Mutation operators

- Syntactic change from *legal program* to *legal program (legal program compiles)*
- Examples
  - Constant-for-constant replacement
    - e.g.: from “x < 5” to “x < 12”
    - select from constants found somewhere in program text
  - Relational operator replacement
    - e.g.: from “x <= 5” to “x < 5”
  - Variable initialization elimination
    - e.g.: from “int x = 5;” to “int x;”

Alive mutants

- Scenario
  - We create 100 mutants from our program
  - We run our test suite on all 100 mutants, plus the original program
  - The original program passes all tests
  - 94 mutants are killed (fail at least one test)
  - 6 mutants remain *alive*
- What can we learn from the living mutants?
  - Do we do nothing?
  - Do we create more mutants?
  - Do we add extra tests to test suite to kill all mutants?
  - Do we re-evaluate our testing strategy?

Alive mutants

- Scenario
  - We create 100 mutants from our program
  - We run our test suite on all 100 mutants, plus the original program
  - The original program passes all tests
  - 94 mutants are killed (fail at least one test)
  - 6 mutants remain *alive*
- What can we learn from the living mutants?
  - ~~Do we do nothing?~~
  - ~~Do we create more mutants?~~
  - ~~Do we add extra tests to test suite to kill all mutants?~~ (*not unless we can generate them automatically*)
  - Do we re-evaluate our testing strategy?

How mutants survive

- A surviving mutant may be equivalent to the original program
  - Maybe changing “x < 0” to “x <= 0” didn't’ change the output at all! The seeded “bug” is superfluous
  - Determining whether a mutant is equivalent to original program or to another mutant may be easy or hard; in the general case, it is undecidable
- Or the test suite could be inadequate
  - If the mutant could have been killed, but was not, it may indicate a weakness in the test suite (need more tests?)
  - *But unlike other adequacy criteria, adding a test case for just this mutant is a bad idea: we already know that particular bug is not there!*

Ineffective test suite

Program

```
f(x, y) {
   if (x < 0 || y < 0) return 0;
   …
   … z = x + 2*y
   …
}
```

Mutant

```
f(x, y) {
   if (x < 0 || y < 0) return 0;
   …
   … z = y + 2*y
   …
}
```

Test Cases:

x = -1, y = 1 (invalid value of x)	alive
x = 1, y = -1 (invalid value of y)	alive
x = 1, y = 1 (normal values)	alive

Same representative value is chosen for the valid block of two characteristics!

More effective test suite

Program

```
f(x, y) {
   if (x < 0 || y < 0) return 0;
   …
   … z = x + 2*y
   …
}
```

Mutant

```
f(x, y) {
   if (x < 0 || y < 0) return 0;
   …
   … z = y + 2*y
   …
}
```

Test Cases:

x = -1, y = 1 (invalid value of x) alive

x = 1, y = -1 (invalid value of y) alive

x = 1, y = 2 (normal values) killed

The mutant was not killed not because we needed an extra test, but because we didn't follow a good strategy in selecting test inputs:

Good idea to perturb input values when choosing representative values!

Mutation testing: key messages

- …is *not* a testing-technique per se
- …is a technique for assessing the “goodness,” or effectiveness, of a test suite
- …can help estimate the defect rate
- …relies on certain assumptions
- …may not be very effective in giving actionable feedback

Sources

- Pezze and Young, *Software Testing & Analysis, Ch 16*