CSE P 504

Advanced Topics in Software Systems:
Testing and Debugging

Spring 2024

Michael Ernst & James Yoo

Course introduction

Key questions:
What does your program do?
How do you know?

Today

- Course overview
- What is software engineering?
- Static vs. dynamic program analysis
- Small-group brainstorming:
  software testing and debugging challenges

Course overview

Logistics

- Lectures, discussions, in-class exercises, homework
- Course material, schedule, policies, etc. on website: https://courses.cs.washington.edu/courses/csep504/24sp/
- Submission of assignments via Canvas: https://canvas.uw.edu

Course schedule by weeks

- Course introduction
- Best practices and version control
- Coverage-based testing
- Mutation-based testing
- Delta debugging
- Invariants and partial oracles
- Statistical fault localization
- Static analysis
- Abstract interpretation
- Automated theorem proving

Background

Dynamic analysis

Static analysis

One homework per week

Homework and in-class activities

Each class meeting has two parts:

1. Lecture & discussion
2. Activity: use a state-of-the-art tool
   - In-class part: small-group work
   - Take-home part: reflection and submission of answers (graded)

Grading

- **20%** Homeworks (2 homeworks)
- **70%** In-class activities (7 sessions)
- **10%** Participation

Questions?

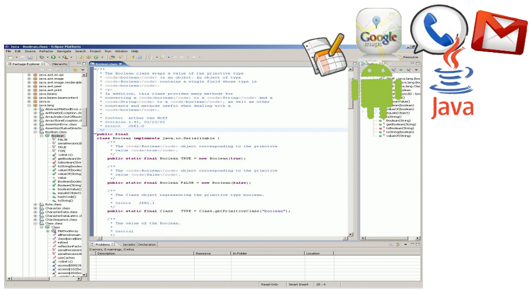
Expectations

- Prepare for lecture by reading (research papers, etc.)
- Participate in discussions
- Try new tools and techniques
- Have fun!

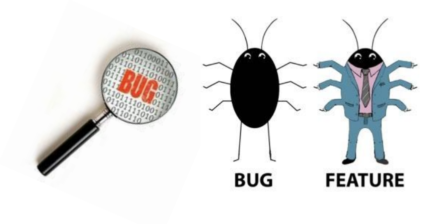
What is Software Engineering?

What is software engineering?

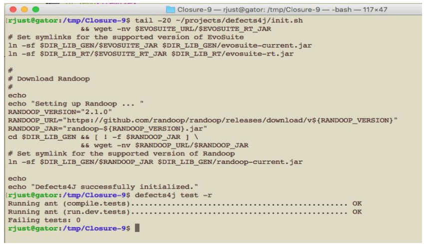
- Developing in an IDE
  and software ecosystem?



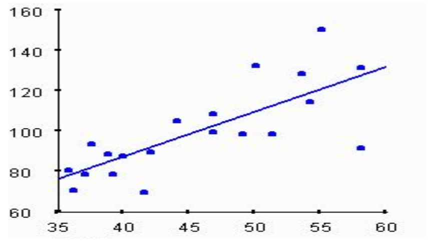
- Testing and debugging a software system?



- Deploying and running
  a software system?



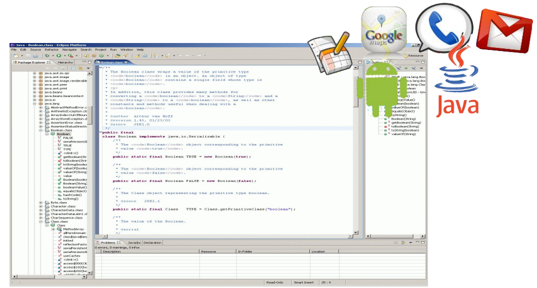
- Empirically evaluating a software system?



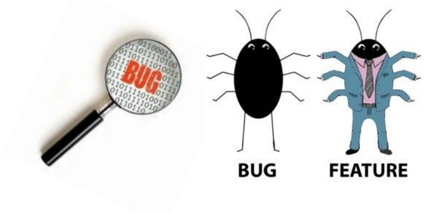
- Writing (design) docs?

What is software engineering?

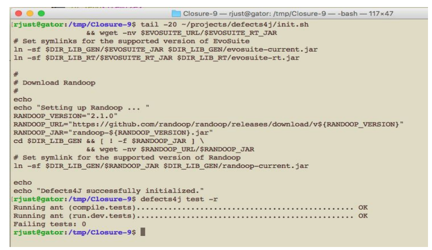
- Developing in an IDE and software ecosystem?



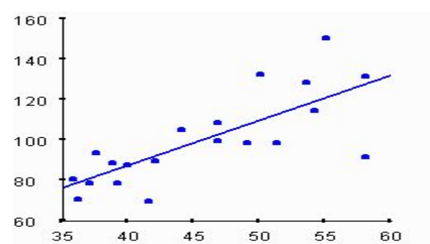
- Testing and debugging a software system?



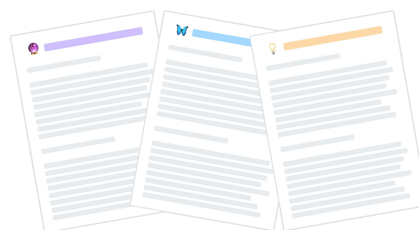
- Deploying and running a software system?



- Empirically evaluating a software system?



- Writing (design) docs?



All of the above and much more!

What is software engineering?

Software Engineering is the complete process of
specifying,
requirements engineering, specifications, documentation
designing, software architecture and design, UI
developing, programming (just one of many important tasks)
analyzing,
testing, debugging, linting, verification, performance engineering
deploying,
DevOps, CI, packaging, operation, remote diagnostics, documentation, websites
& maintaining refactoring, extensions, adaptation, issue tracking
a software system.

Static and dynamic program analysis

What is program analysis?

Analyze the behavior of a program; examples:

- optimize the program
- check program's behavior (against its specification)

Concerned with properties such as

- Correctness
- Performance
- Safety
- Liveness

Can be static or dynamic, which affects

- Computational cost
- Accuracy and precision

Why do we need program analysis?

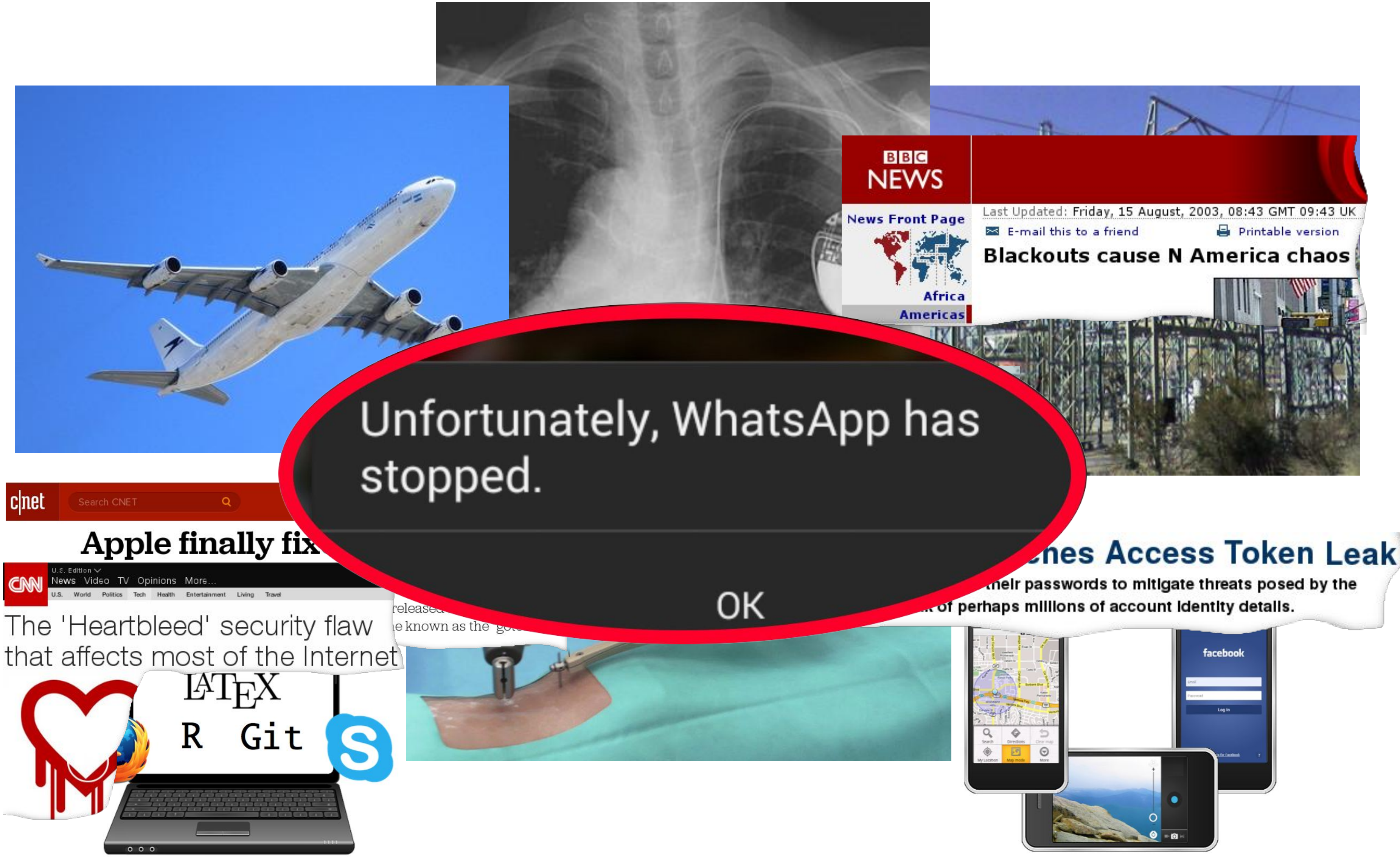
Why do we need program analysis?

- ~15 million lines of code

Let’s say 50 lines per page (0.05 mm)

- 300000 pages
- 15 m (49 ft)

Why do we need program analysis?



Example analysis: code review

Different types of reviews

- Code/design review
- Informal walkthrough
- Formal inspection

A requirement for many (safety-critical) systems.

Example analysis: code review

Different types of reviews

- Code/design review
- Informal walkthrough
- Formal inspection

```
double foo(double[] d) {
  int n = d.length;
  double s = 0;
  int i = 0;
  while (i<n)
    s = s + d[i];
    i = i + 1;
  double a = s / n;
  return a;
}
```

Let’s do an informal code review.

Can this Java code be improved?

Example analysis: code review

Different types of reviews

- Code/design review
- Informal walkthrough
- Formal inspection

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n) {
    sum = sum + nums[i];
    i = i + 1;
  }

  double avg = sum / n;
  return avg;
}
```

```
static OSStatus
SSLVerifySignedServerKeyExchange(...) {
    OSStatus err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify returned %d\n", (int)err);
        goto fail;
    }
    fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

Anything wrong with that code?

Apple's "goto fail" bug:
a security vulnerability for 2 years!

```
static OSStatus
SSLVerifySignedServerKeyExchange(...) {
	OSStatus err;
	...
	if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
		goto fail;
		goto fail;
	if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
		goto fail;
	err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
	if(err) {
		sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify returned %d\n", (int)err);
		goto fail;
	}
	fail:
		SSLFreeBuffer(&signedHashes);
		SSLFreeBuffer(&hashCtx);
		return err;
}
```

Anything wrong with that code?

Code review

Pros

- Can be applied at any step in the development process
- Does not require an executable program
- Improves confidence and communication

Cons

- Time-consuming
- Mostly informal
- Not replicable

Static and dynamic analysis

Static and dynamic analysis Outline

⇒ Definition of static and dynamic analysis

Synergy: combining static and dynamic analysis

- Aggregation
- Analogies

Duality: subsets of behavior

Static analysis

Examples: compiler optimizations, linters, program verifiers

Examine program text (no execution)

Build a model of program state

- An abstraction of the run-time state

Reason over possible behaviors

- "run" the program over the abstract state

Abstract interpretation

Typically implemented via dataflow analysis

Each program statement's *transfer function* indicates how it transforms state

Example: What is the transfer function for

```
y = x++;
```

?

Selecting an abstract domain

$\langle\, x = 2; y = 5 \,\rangle$

```
y = x++;
```

$\langle\, x = 3; y = 2 \,\rangle$

$\langle\, x = \{ 3, 5, 7 \}; y = \{ 9, 11, 13 \} \,\rangle$

```
y = x++;
```

$\langle\, x = \{ 4, 6, 8 \}; y = \{ 3, 5, 7 \} \,\rangle$

$\langle$ x is odd; y is odd $\rangle$

```
y = x++;
```

$\langle$ x is even; y is odd $\rangle$

$\langle x{=}3, y{=}11\rangle, \langle x{=}5, y{=}9\rangle, \langle x{=}7, y{=}13\rangle$

```
y = x++;
```

$\langle x{=}4, y{=}3\rangle, \langle x{=}6, y{=}5\rangle, \langle x{=}8, y{=}7\rangle$

$\langle$ x is prime; y is prime $\rangle$

```
y = x++;
```

$\langle$ x is anything; y is prime $\rangle$

$\langle\, x_n = f(a_{n-1},\ldots,z_{n-1}); y_n = f(a_{n-1},\ldots,z_{n-1}) \,\rangle$

```
y = x++;
```

$\langle\, x_{n+1} = x_n + 1; y_{n+1} = x_n \,\rangle$

Research challenge: Choose good abstractions

The abstraction determines the expense (in time and space)

The abstraction determines the accuracy (what information is lost)

- Less accurate results are poor for applications that require precision
- Cannot conclude all true properties in the grammar

Static analysis recap

- Slow to analyze large models of state, so use abstraction
- Conservative: account for abstracted-away state
- Sound*: (weak) properties are guaranteed to be true
  - "`f` returns a non-negative value"
    is weaker (but easier to establish) than
    "`f` returns the absolute value of its argument"

*Some static analyses are not sound

Dynamic analysis

Examples: profiling, testing, debugging

Execute program (over some inputs)

- No abstraction: semantics from runtime system

Observe executions

- Requires instrumentation infrastructure

2 research challenges:

- what to measure
- what test runs

Research challenge: What to measure?

Coverage or frequency

- Statements, branches, paths, procedure calls, types, method dispatch

Values computed

- Formal parameters, array indices

Run time, memory usage

Test oracle results

Similarities among runs [Podgurski 99, Reps 97]

Like abstraction, determines what is reported

Research challenge: Choose good tests

The test suite determines the expense (in time and space)

The test suite determines the accuracy (what executions are never seen)

- Less accurate results are poor for applications that require correctness
- Many domains do not require correctness!

*What information is being collected also matters

Dynamic analysis recap

- Can be as fast as execution (over a test suite, and allowing for data collection)
  - Example: aliasing
- Precise: no abstraction or approximation
- Unsound: results may not generalize to future executions
  - Describes execution environment or test suite

Static analysis	Dynamic analysis
Abstract domain	Concrete execution
slow if precise	slow if exhaustive
Conservative	Precise
due to abstraction	no approximation
Sound	Unsound
due to conservatism	does not generalize

Outline

Definition of static and dynamic analysis

⇒ Synergy: combining static and dynamic analysis

- Aggregation
- Analogies

Duality: subsets of behavior

Combining static and dynamic analysis

1. Aggregation:
   Pre- or post-processing
2. Inspiring analogous analyses:
   Same problem, different domain

1. Aggregation: Pre- or post-processing

Use output of one analysis as input to another

Dynamic then static

- Profile-directed compilation: unroll loops, inline, reorder dispatch, …
- Verify properties observed at run time

Static then dynamic

- Reduce instrumentation requirements
  - Efficient branch/path profiling
  - Discharge obligations statically (type/array checks)
- Type checking (e.g., Java, including generics and casts)
- Indicate suspicious code to test more thoroughly

2. Analogous analyses: Same problem, different domain

Any analysis problem can be solved in either domain

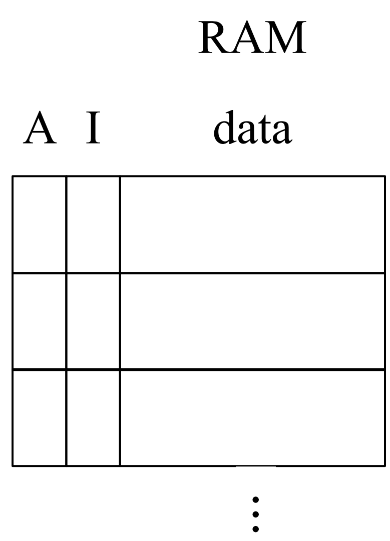
- Type safety: no memory corruption or operations on wrong types of values
  - Static type-checking
  - Dynamic type-checking
- Slicing: what computations could affect a value
  - Static: reachability over dependence graph
  - Dynamic: tracing

Memory checking

Goal: find array bound violations, uses of uninit. memory

Purify [Hastings 92], Valgrind: run-time instrumentation

- Tagged memory: 2 bits (allocated, initialized) per byte
- Each instruction checks/updates the tags
  - Allocate: set “A” bit, clear “I” bit
  - Write: require “A” bit, set “I” bit
  - Read: require “I” bit
  - Deallocate: clear “A” bit



LCLint [Evans 96]: compile-time dataflow analysis

- Abstract state contains allocated and initialized bits
- Each transfer function checks/updates the state

Identical analyses!

Another example: atomicity checking [Flanagan 2003]

Specifications

- Specification checking
  - Statically: theorem-proving
  - Dynamically: `assert` statement
- Specification generation
  - Statically: by hand or abstract interpretation [Cousot 77]
  - Dynamically: by invariant detection [Ernst 99], reporting unfalsified properties

Your analogous analyses here

Look for gaps with no analogous analyses!

Try using the same analysis

- But be open to completely different approaches

There is still low-hanging fruit to be harvested

Outline

Definition of static and dynamic analysis

Synergy: combining static and dynamic analysis

- Aggregation
- Analogies

⇒ Duality: subsets of behavior

Static analysis	Dynamic analysis
Abstract domain	Concrete execution
slow if precise	slow if exhaustive
Conservative	Precise
due to abstraction	no approximation
Sound	Unsound
due to conservatism	does not generalize

Sound dynamic analysis

Observe every possible execution!

Problem: infinite number of executions

Solution: test case selection and generation

- Efficiency tweaks to an algorithm that works perfectly in theory but exhausts resources in practice

Precise static analysis

Reason over full program state!

Problem: infinite number of executions

Solution: data or execution abstraction

- Efficiency tweaks to an algorithm that works perfectly in theory [Cousot 77] but exhausts resources in practice

Dynamic analysis focuses on a subset of executions

The executions in the test suite

- Easy to enumerate
- Characterizes program use

Typically optimistic for other executions

Static analysis focuses on a subset of data structures

More precise for data or control described by the abstraction

- Concise logical description
- Typically conservative elsewhere (safety net)

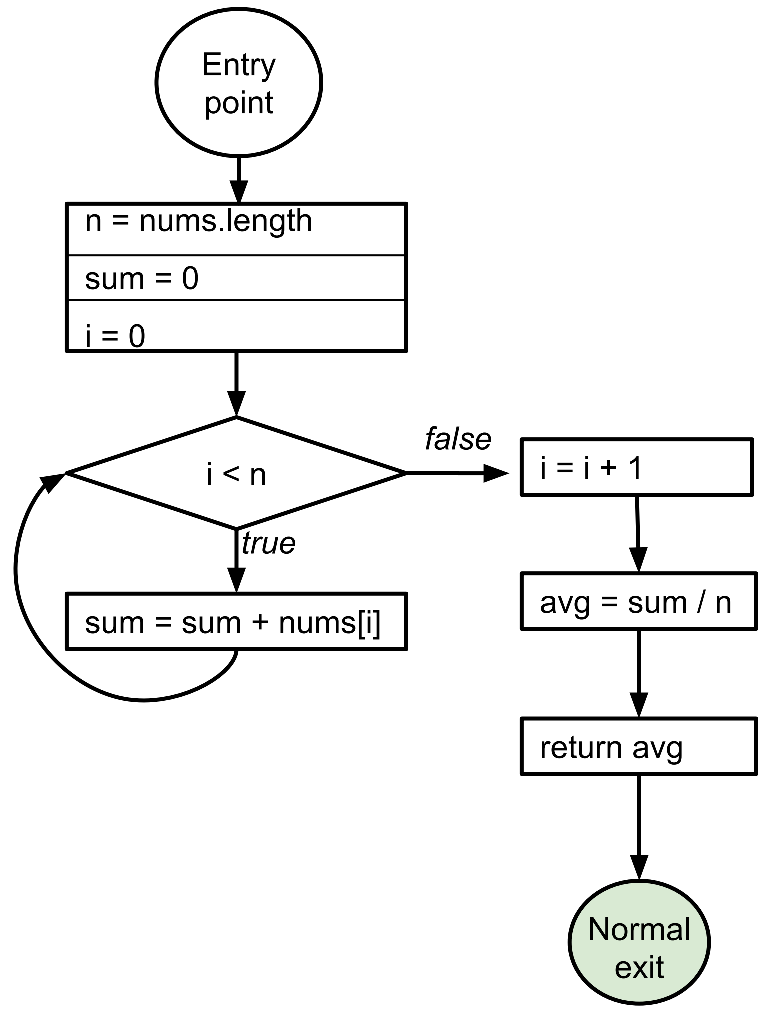
Example: *k*-limiting [Jones 81]

- Represents each object reachable by $\leq k$ pointers
- Groups together (approximates) more distant objects

Static analysis: the control flow graph

Control-flow and data-flow analysis

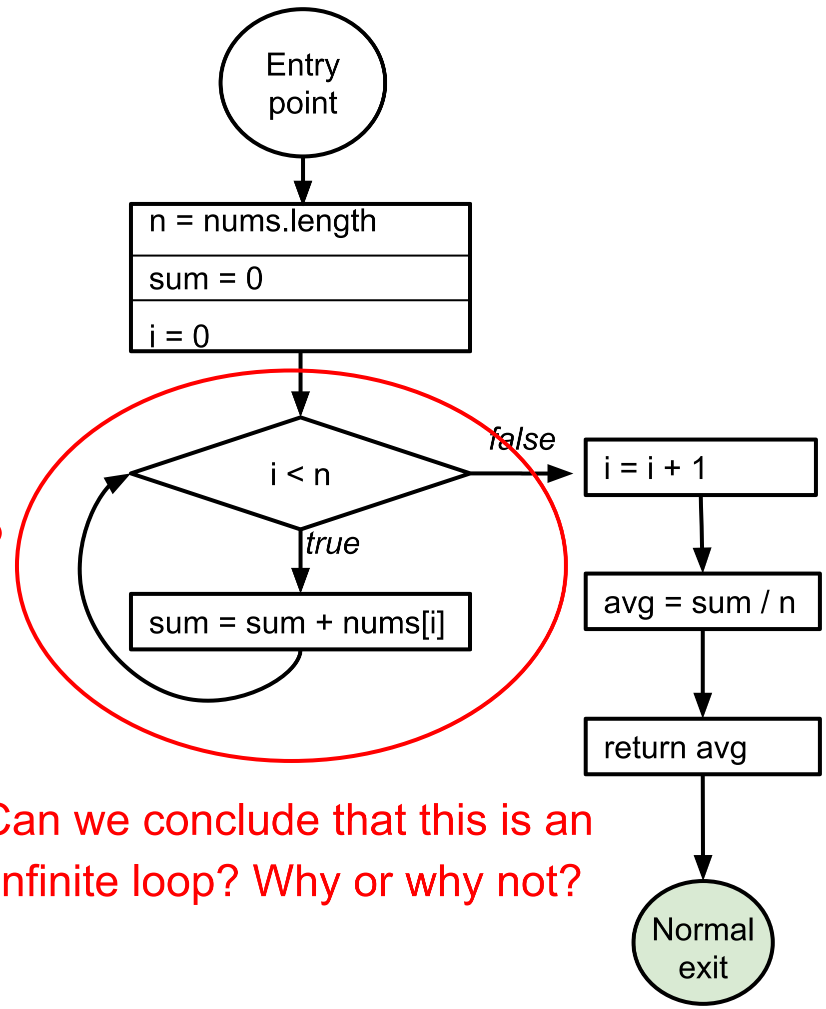
```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n)
    sum = sum + nums[i];
    i = i + 1;

  double avg = sum / n;

  return avg;
}
```



Static analysis: example

Control-flow and data-flow analysis

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n)
    sum = sum + nums[i];
    i = i + 1;

  double avg = sum / n;

  return avg;
}
```



Can we conclude that this is an infinite loop? Why or why not?

Dynamic analysis: example

Software testing (also monitoring and profiling)

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n)
    sum = sum + nums[i];
    i = i + 1;

  double avg = sum / n;

  return avg;
}
```

A test for the avg function:

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = Math.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```

Testing sqrt

```
// throws: IllegalArgumentException if x<0
// returns: approximation to square root of x
double sqrt(double x) {...}
```

What are some values or ranges of *x* to test?

- $x < 0$ (exception thrown)
- $x \geq 0$ (returns normally)
- around $x = 0$ (boundary condition)
- perfect squares (sqrt(x) an integer), non-perfect squares
- x<sqrt(x) and x>sqrt(x) – that's x<1 and x>1 (and x=1)
- *Specific tests: say x = -1, 0, 0.5, 1, 4*

What's so hard about testing?

"Just try it and see if it works..."

```
// requires: 1 ≤ x,y,z ≤ 10000
// effects:  computes some f(x,y,z)
int proc(int x, int y, int z)
```

Exhaustive testing would require 1 trillion runs!

Sounds totally impractical – and this is a trivially small problem

Key problem: choosing test suite (partitioning of inputs)

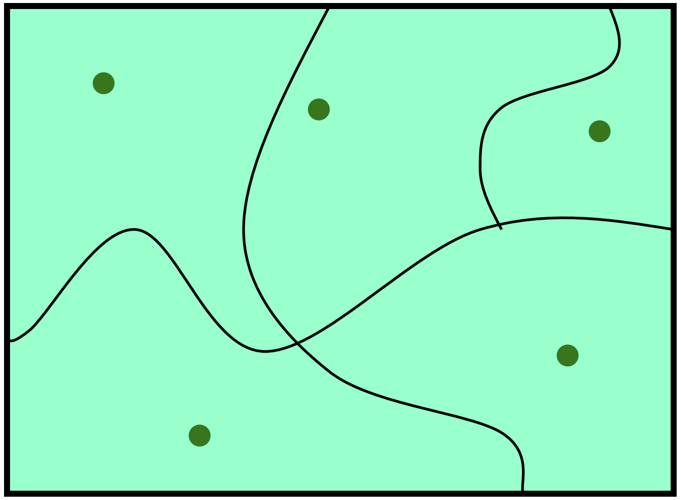
Small enough to finish quickly

Large enough to validate the program

What are heuristics for writing tests?

When are you done testing?

Approach: Partition the Input Space



Ideal test suite:

Identify sets with same behavior

Try one input from each set

Two problems

1. Notion of the same behavior is subtle
   - Naive approach: execution equivalence
   - Better approach: revealing subdomains
2. Discovering the sets requires perfect knowledge
   - Use heuristics to approximate cheaply

Naive approach: Execution equivalence

```
// returns:  if x < 0   ⇒ returns -x
//           otherwise  ⇒ returns x
int abs(int x) {
   if (x < 0) return -x;
   else       return x;
}
```

All $x < 0$ are execution equivalent:

program takes same sequence of steps for any $x < 0$

All $x \geq 0$ are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite

Execution equivalence is not enough

Consider the following buggy code:

```
// returns:  if x < 0      ⇒ returns -x
//           otherwise ⇒ returns x
int abs(int x) {
   if (x < -2) return -x;
   else        return x;
}
```

Two execution behaviors:

$x < -2$ $\quad$ $x \geq -2$

Three behaviors:

$x < -2$ (OK) $\quad$ $x = -2$ or -1 (bad) $\quad$ $x \geq 0$ (OK)

{-3, 3} does not reveal the error!

Heuristic: Revealing Subdomains

A subdomain is a subset of possible inputs.

A subdomain is revealing for error E if either:

- Every input in that subdomain triggers error E, or
- No input in that subdomain triggers error E

Need to test only one input from each subdomain.

If

- subdomains cover the entire input space, and
- subdomains are revealing, and
- test oracles are sufficiently strong to detect E

then we are guaranteed to detect every error E.

The trick is to guess these revealing subdomains.

Example

For buggy **`abs`**, what are revealing subdomains?

```
// returns:  x < 0      ⇒ returns -x
//           otherwise ⇒ returns x
int abs(int x) {
    if (x < -2) return -x;
    else        return x;
}
```

Example sets of subdomains:

Which is best?

… {-2} {-1} {0} {1} …

{…, -4, -3} {-2, -1} {0, 1, …}

… ~~{-6, -5, -4} {-3, -2, -1}~~ {0, 1, 2}

…

Heuristics for designing test suites
= heuristics for choosing inputs
= heuristics for dividing the domain

A good heuristic gives:

- few subdomains
- For all errors E in some class of errors,
  high probability that some subdomain is revealing for E

Different heuristics target different classes of errors

- In practice, combine multiple heuristics

Black Box Testing

Heuristic: Explore each case/path in the specification

Procedure is a black box: interface visible, internals hidden but its spec is like an implementation you can test

Example:

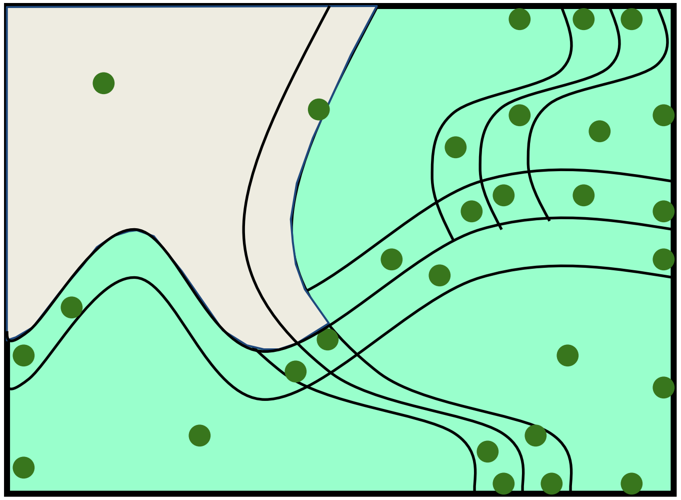
```
// returns:   a > b ⇒ returns a
//            a < b ⇒ returns b
//            a = b ⇒ returns a

int max(int a, int b)
```

3 cases, so 3 tests:

(4, 3) => 4 *(i.e. any input in the subdomain a > b)*
(3, 4) => 4 *(i.e. any input in the subdomain a < b)*
(3, 3) => 3 *(i.e. any input in the subdomain a = b)*

Heuristic: Boundary Testing

You might have misdrawn the boundaries ⇒ the subdomains are not revealing



Small subdomains at the edges of the "main" subdomains can reveal common errors:

- off-by-one bugs
- empty container
- null
- arithmetic overflow
- aliasing

In practice:
Create tests at the edges of subdomains

Boundary Testing

To define the boundary, need a *metric space*

A distance metric that defines adjacent inputs

One approach: operations define the metric space

Two values are adjacent if one operation apart

Point is on a boundary if either:

- There exists an adjacent point in a different subdomain, or
- Some basic operation cannot be applied to the point

Example: list of integers

Basic operations: **`create`**, **`insert`**, **`remove`**, …

Adjacent values: <[2,3],[2,3,4]>, <[2,3],[2]>

Boundary value: [] (can’t apply **`remove`**)

Small-group brainstorming:
software testing and debugging challenges