Debugging

CSE 331
Spring 2010

Ways to get your code right

Validation

Purpose is to uncover problems and increase confidence

Combination of reasoning and test

Debugging

Finding out why a program is not functioning as intended

Defensive programming

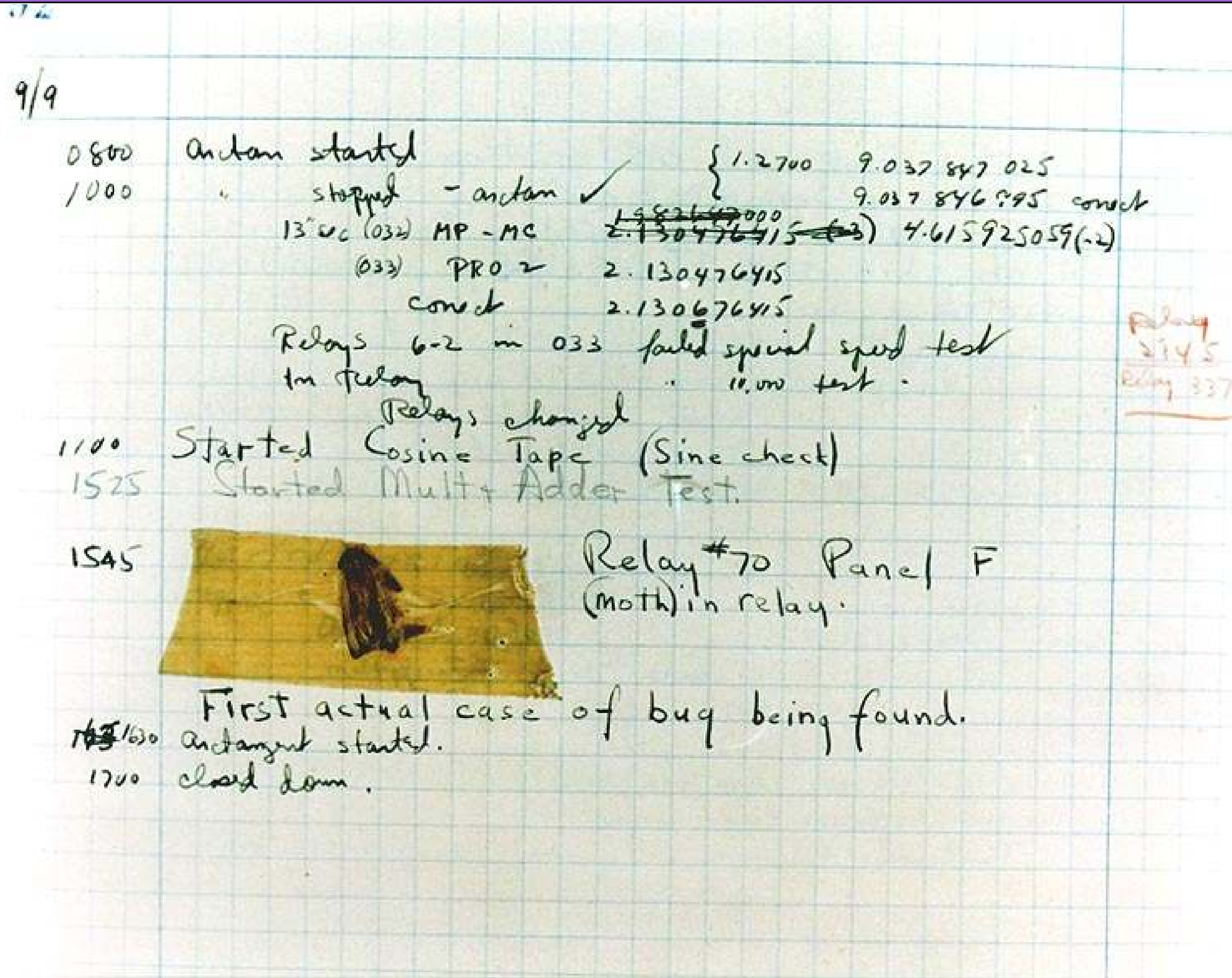
Programming with validation and debugging in mind

Testing ≠ debugging

test: reveals existence of problem

debug: pinpoint location+cause of problem

A bug – September 9, 1947



A Bug's Life

Defect – mistake committed by a human

Error – incorrect computation

Failure – visible error: program violates its specification

Debugging starts when a failure is observed

- Unit testing
- Integration testing
- In the field

Defense in depth

1. Make errors impossible

   Java makes memory overwrite bugs impossible

2. Don't introduce defects

   Correctness: get things right the first time

3. Make errors immediately visible

   Local visibility of errors: best to fail immediately

   Example: checkRep() routine to check representation invariants

4. Last resort is debugging

   Needed when effect of bug is distant from cause

   Design experiments to gain information about bug

   - Fairly easy in a program with good modularity, representation hiding, specs, unit tests etc.
   - Much harder and more painstaking with a poor design, e.g., with rampant rep exposure

First defense: Impossible by design

In the language

- Java makes memory overwrite bugs impossible

In the protocols/libraries/modules

- TCP/IP guarantees that data is not reordered
- BigInteger guarantees that there is no overflow

In self-imposed conventions

- Hierarchical locking makes deadlock bugs impossible
- Banning recursion prevents infinite recursion/insufficient stack
- Immutable data structures guarantees behavioral equality
- Caution: You must maintain the discipline

Second defense: Correctness

Get things right the first time

Don’t code before you think! Think before you code.

If you're making lots of easy-to-find bugs, you're also making hard-to-find bugs – don't use the compiler as crutch

Especially true, when debugging is going to be hard

Concurrency

Real-time environment

Other difficult test and instrumentation environments

Simplicity is key

Modularity

- Divide program into chunks that are easy to understand
- Use abstract data types with well-defined interfaces
- Use defensive programming; avoid rep exposure

Specification

- Write specs for all modules, so that an explicit, well-defined contract exists between each module and its clients

Third defense: Immediate visibility

If we can't prevent bugs, we can try to localize them to a small part of the program

- Assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation
- Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (unless it's in the test driver)
- Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed

When localized to a single method or small module, bugs can be found simply by studying the program text

Benefits of immediate visibility

- Key difficulty of debugging is to find the code fragment responsible for an observed problem
  - A method may return an erroneous result, but be itself error free, if there is prior corruption of representation
- The earlier a problem is observed, the easier it is to fix
  - For example, frequently checking the rep invariant helps the above problem
- General approach: fail-fast
  - Check invariants, don't just assume them
  - Don't try to recover from bugs – this just obscures them

Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (true) {
    if (a[i]==k) break;
    i++;
}
```

This code fragment searches an array `a` for a value `k`.

- Value is guaranteed to be in the array.
- What if that guarantee is broken (by a bug)?

Temptation: make code more “robust” by not failing

Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
```

Now at least the loop will always terminate

- But no longer guaranteed that `a[i]==k`
- If rest of code relies on this, then problems arise later
- *This obscures the link between the bug's origin and the eventual erroneous behavior it causes.*

Don't hide bugs

```
// k is guaranteed to be present in a
int i = 0;
while (i<a.length) {
    if (a[i]==k) break;
    i++;
}
assert (i<a.length) : "key not found";
```

Assertions let us document and check invariants

Abort/debug program as soon as problem is detected

- Turn an error into a failure
- But, assertion not checked until we use the data
- Might be a long time after original error

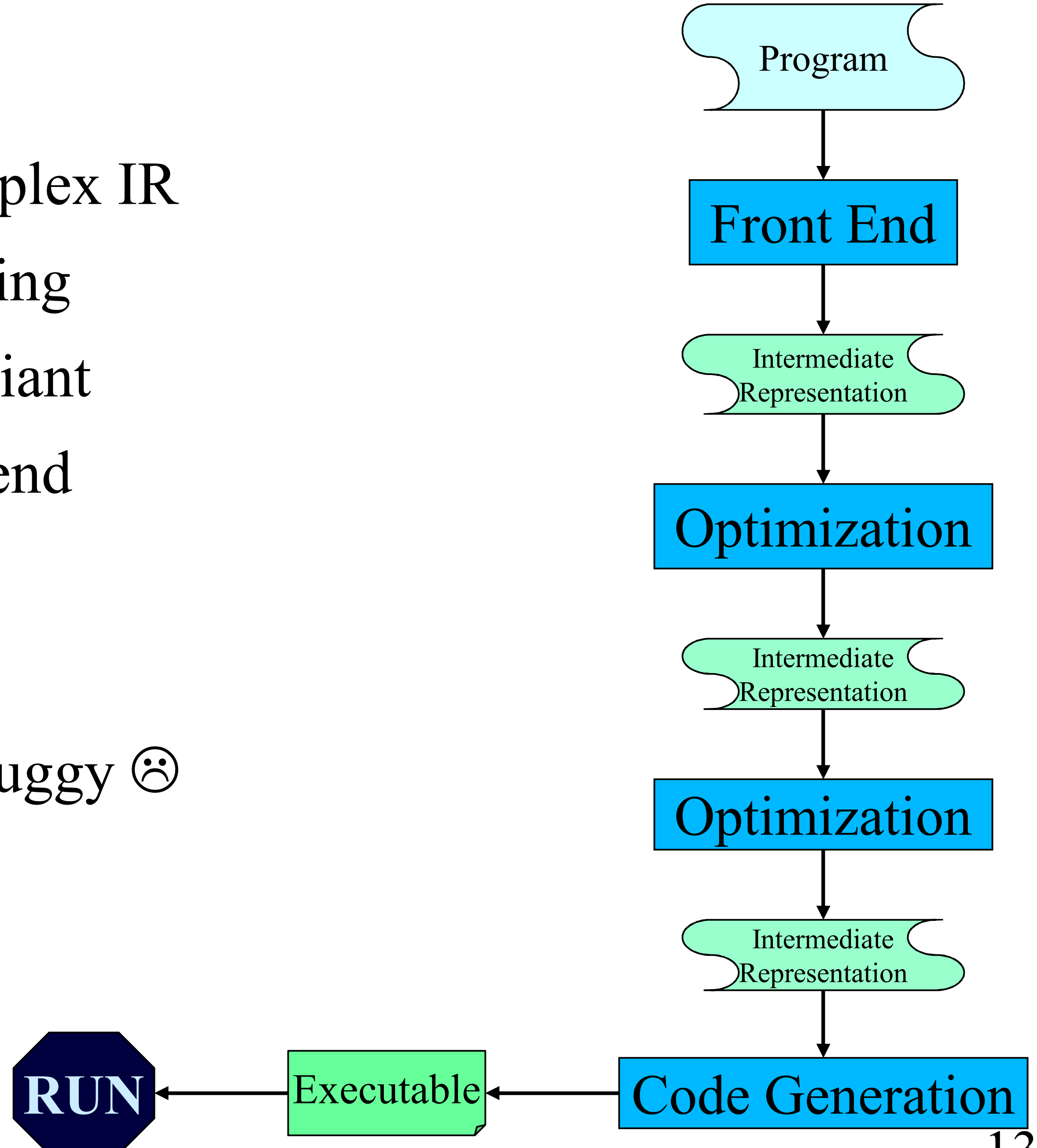
How to debug a compiler

Multiple passes

- Each operates on a complex IR
- Lot of information passing
- Very complex rep invariant
- Code generation at the end

Bugs

- Compiler crashes ☺
- Generated program is buggy ☹



Bug-Specific Checks

```
static void check(Integer a[], List<Integer> index) {
   for (e:index) {
   assert(e != 1234, "Inconsistent Data Structure");
   }
}
```

Bug shows up as 1234 in list

Check for that specific condition

It's usually better to do this as a conditional breakpoint in a debugger

Checks In Production Code

Should you include assertions and checks in production code?

Yes: stop program if check fails - don't want to take chance program will do something wrong

No: may need program to keep going, maybe bug does not have such bad consequences

Correct answer depends on context!

Ariane 5 – program halted because of overflow in unused value, exception thrown but not handled until top level, rocket crashes…

Regression testing

- Whenever you find and fix a bug
  - Add a test for it
  - Re-run all your tests
- Why is this a good idea?
  - Often reintroduce old bugs while fixing new ones
  - Helps to populate test suite with good tests
  - If a bug happened once, it could well happen again
- Run regression tests as frequently as you can afford to
  - Automate the process
  - Make concise test suites, with few superfluous tests

Last resort: debugging

Bugs happen

- Industry average: 10 bugs per 1000 lines of code ("kloc")

Bugs that are not immediately localizable happen

- Found during integration testing
- Or reported by user

step 1 – Clarify symptom (simplify input)

step 2 – Find and understand cause, create test

step 3 – Fix

step 4 – Rerun all tests

the debugging process

step 1 – find a small, repeatable test case that produces the failure (may take effort, but helps clarify the bug, and also gives you something for regression)

- *don't move on to next step until you have a repeatable test*

step 2 – narrow down location and proximate cause

- study the data / hypothesize / experiment / repeat
- may change the code to get more information
- *don't move on to next step until you understand the cause*

step 3 – fix the bug

- Is it a simple typo, or design flaw? Does it occur elsewhere?

step 4 – add test case to regression suite

- Is this bug fixed? Are any other new bugs introduced?

Debugging and the scientific method

- Debugging should be systematic
  - Carefully decide what to do
  - Keep a record of everything that you do
  - Don't get sucked into fruitless avenues

1. Formulate a hypothesis
2. Design an experiment
3. Perform the experiment
4. Adjust your hypothesis and continue

Reducing input size example

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A,B s.t. full=A+sub+B)
boolean contains(String full, String sub);
```

User bug report:

It can't find the string `"very happy"` within:

```
"Fáilte, you are very welcome! Hi Seán! I am
very very happy to see you all."
```

Wrong responses:

1. See accented characters, panic about not having thought about unicode, and go diving for your Java texts to see how that is handled.
2. Try to trace the execution of this example.

Right response: simplify/clarify the symptom

Reducing absolute input size

Find a simple test case by divide-and-conquer

Pare test down – can't find **`"very happy"`** within:

- **`"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`**
- **`"I am very very happy to see you all."`**
- **`"very very happy"`**

Can find **`"very happy"`** within:

- **`"very happy"`**

Can't find **`"ab"`** within **`"aab"`**

(We saw what might cause this bug earlier in the quarter!)

Reducing relative input size

Sometimes it is helpful to find two almost identical test cases where one gives the correct answer and the other does not

Can't find **`"very happy"`** within:

- **`"I am very very happy to see you all."`**

Can find **`"very happy"`** within:

- **`"I am very happy to see you all."`**

General strategy: simplify

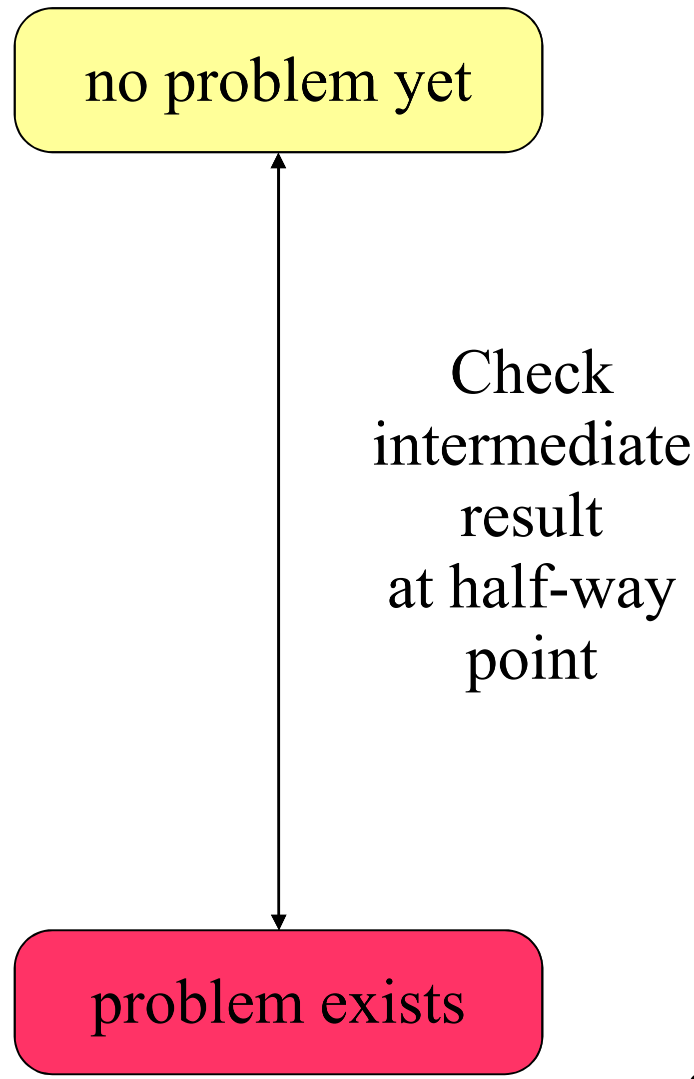
- In general: find simplest input that will provoke bug
  - Usually not the input that revealed existence of the bug
- Start with data that revealed bug
  - Keep paring it down (binary search can help)
  - Often leads directly to an understanding of the cause
- When not dealing with simple method calls
  - The "test input" is the set of steps that reliably trigger the bug
  - Same basic idea

Localizing a bug

- Take advantage of modularity
  - Start with everything, take away pieces until bug goes
  - Start with nothing, add pieces back in until bug appears
- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results
- Can use binary search to speed things up
  - Bug happens somewhere between first and last statement
  - So can do binary search on that ordered set of statements

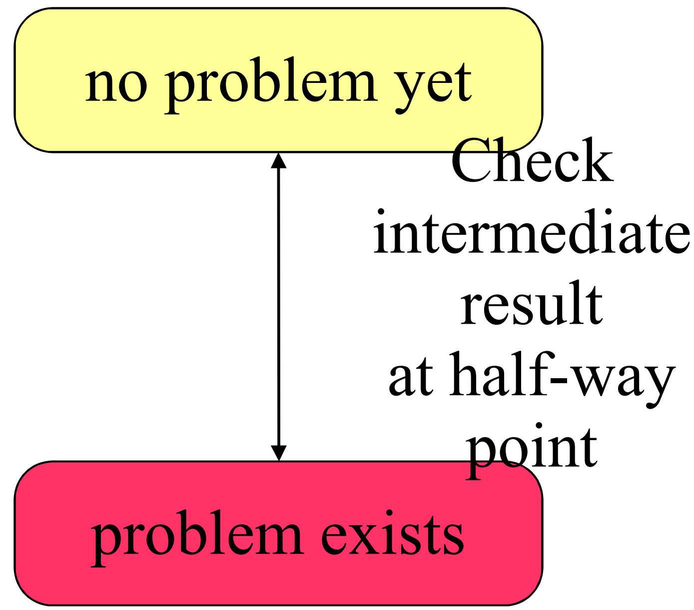
binary search on buggy code

```java
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```



binary search on buggy code

```java
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```



Quickly home in on bug in O(log n) time by repeated subdivision

Binary Search in a Compiler

	Class								
	A	B	C	D	E	F	G	H	I
Front end									
Optimization 1									
Optimization 2									
Optimization 3									
Optimization 4									
Optimization 5									
Optimization 6									
Code generation									
Link and Run									
Test									

Binary Search

Binary search

Binary Search in a Compiler

	Class								
	A	B	C	D	E	F	G	H	I
Front end									
Optimization 1									
Optimization 2									
Optimization 3									
Optimization 4									
Optimization 5									
Optimization 6									
Code generation									
Link and Run									
Test									

Detecting Bugs in the Real World

- Real Systems:
  - Large and complex (duh!)
  - Collection of modules, written by multiple people
  - Complex input
  - Many external interactions
  - Non-deterministic
- Replication can be an issue
  - Infrequent bug
  - Instrumentation eliminates the bug
- Bugs cross abstraction barriers
- Large time lag from corruption to detection

Heisenbugs

Sequential, deterministic program – bug is repeatable

But the real world is not that nice…

- Continuous input/environment changes
- Timing dependencies
- Concurrency and parallelism

Bug occurs randomly

Hard to reproduce

- Use of debugger or assertions → bug goes away
- Only happens when under heavy load
- Only happens once in a while

Debugging In Harsh Environments

- Harsh environments
  - Bug is nondeterministic, difficult to reproduce
  - Can’t print or use debugger
  - Can’t change timing of program (or bug has to do with timing)
- Build an event log (circular buffer)
- Log events during execution of program as it runs at speed
- When detect error, stop program and examine logs

Logging Events

- Helps you reconstruct the past
  - Example: Script file output format
- The log may be all you know about a customer’s environment
  - It should enable you to reproduce the bug
- Advanced topics:
  - To reduce overhead, may store in memory, not on disk
  - Circular logs to avoid resource exhaustion

Tricks for Hard Bugs

- Rebuild system from scratch and reboot
- Explain bug to a friend
- Make sure it is a bug – program may be working correctly and you don't realize it!
- Minimize input required to exercise bug
- Add checks to program
  - Minimize distance between error and detection
  - Use binary search to narrow down possible locations
- Use logs to record events in history

Where is the bug?

- The bug is <u>not</u> where you think it is
  - Ask yourself where it cannot be; explain why
- Look for stupid mistakes first, e.g.,
  - Reversed order of arguments: Collections.copy(src,dest)
  - Spelling of identifiers: int hash<u>c</u>ode()
    - `@Override` can help catch method name typos
  - Same object vs. equal: a == b versus a.equals(b)
  - Failure to reinitialize a variable
  - Deep vs. shallow copy
- Make sure that you have correct source code
  - Recompile everything

When the going gets tough

- Reconsider assumptions
  - E.g., has the OS changed? Is there room on the hard drive?
  - Debug the code, not the comments
- Start documenting your system
  - Gives a fresh angle, and highlights area of confusion
- Get help
  - We all develop blind spots
  - Explaining the problem often helps
- Walk away
  - Trade latency for efficiency – **sleep**!
  - One good reason to start early

Key Concepts in Review

- Testing and debugging are different
  - Testing reveals existence of bugs
  - Debugging pinpoints location of bugs
- Goal is to get program to work
  - Not to find bugs
- Debugging should be a systematic process
  - Use the "scientific method"
- It's important to understand source of bugs
  - To decide on appropriate repair