Debugging

Michael Ernst

CSE 190p

University of Washington

Two key ideas

1. The scientific method
2. Divide and conquer

If you master those, you will find debugging easy, and possibly enjoyable

The scientific method

- Create a hypothesis
- Design an experiment to test that hypothesis
  - Ensure that it yields insight
- Understand the result of your experiment
  - If you don’t understand, then possibly suspend your main line of work to understand that

Tips:

- Be systematic
  - Never do anything if you don't have a reason
  - Don’t just flail
    - Random guessing is likely to dig you into a deeper hole
- Don’t make assumptions (verify them)

Example experiments

1. An alternate implementation of a function
   – Run all your test cases afterward
2. A new, simpler test case
   – Examples: smaller input, or test a function in isolation
   – Can help you understand the reason for a failure

Your scientific notebook

Record everything you do

- Specific inputs and outputs (both expected and actual)
- Specific versions of the program
  - If you get stuck, you can return to something that works
  - You can write multiple implementations of a function
- What you have already tried
- What you are in the middle of doing now
  - This may look like a stack!
- What you are sure of, and why

Your notebook also helps if you need to get help or reproduce your results

Divide and conquer

- Where is the defect (or “bug”)?
- Your goal is to find the one place that it is
- Finding a defect is often harder than fixing it

- Initially, the defect might be anywhere in your program
  - It is impractical to find it if you have to look everywhere
- Idea: bit by bit reduce the scope of your search
- Eventually, the defect is localized to a few lines or one line; then you can fix it

Divide and conquer in the program

- Localize the defect to part of the program (e.g., one function, or one part of a function)
- Code that isn't executed cannot contain the defect

- Test one function at a time
- Add assertions or print statements
  - The defect is executed before the failing assertion (and maybe after a succeeding assertion)
- Split complex expressions into simpler ones

- Example: Failure in

```
a = set({graph.neighbors(user)})
```

  Change it to

```
x = graph.neighbors(user)
y = {x}
a = set(x)
```

  (but with better variable names).
  - The error occurs on the "y = {x}" line

Debugging via print (or "logging") statements

- A sequence of print statements is a record of the execution of your program
- The print statements let you see and search multiple moments in time
- Print statements are a useful technique, in moderation
- Be disciplined
  - Too much output is overwhelming rather than informative
  - Remember the scientific method: have a reason (a hypothesis to be tested) for each print statement
  - Don't *only* use print statements

Divide and conquer in time

- The code used to work (for some test case)
- The code now fails
- The defect is related to some line you changed

- This is useful only if you kept a version of the code that worked (use good names!)
- This is most useful if you have made few changes
- Moral: test often!
  - Fewer lines to compare
  - You remember what you were thinking/doing recently

Divide and conquer in test cases

- Your program fails when run on some large input
  - It's hard to comprehend the error message
  - The log of print statement output is overwhelming
- Try a smaller input
  - Choose an input with some but not all characteristics of the large input
  - Example: Unicode characters, zeroes in data, …