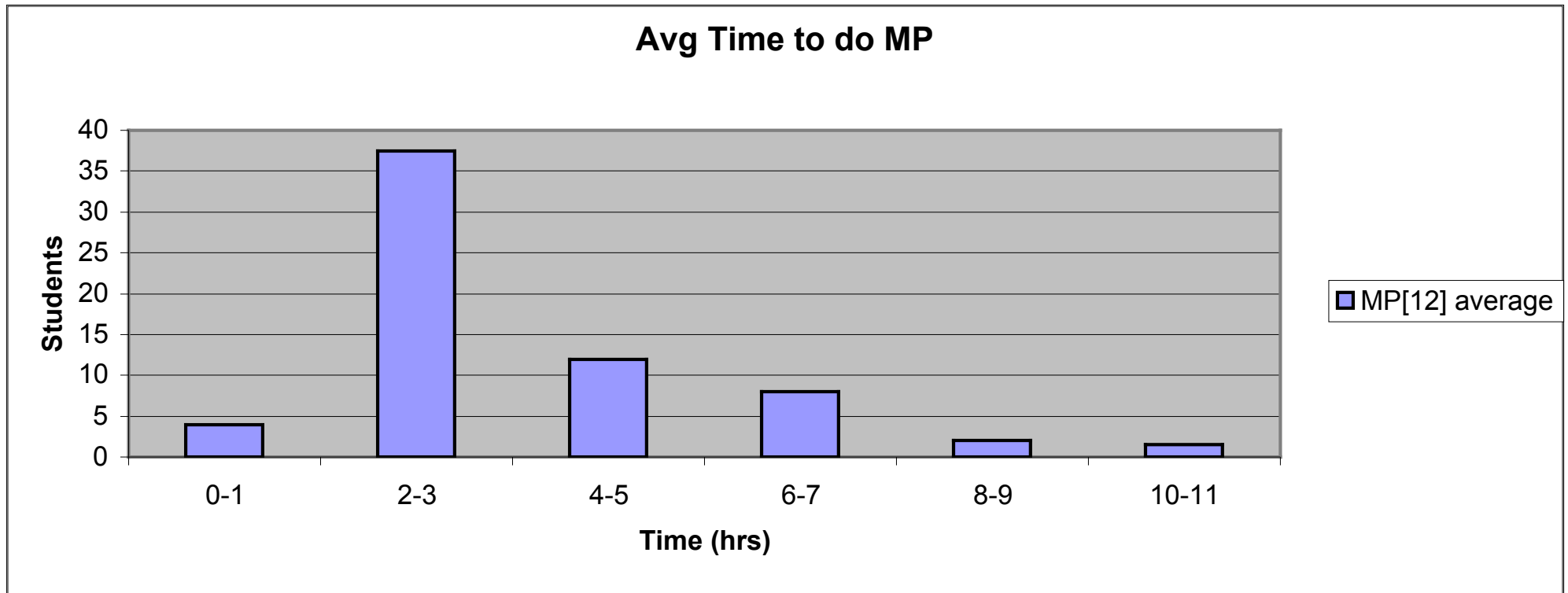# Debugging

Some data from a previous semester

March 6, 2013

What is responsible for distribution?

- Have you ever monitored the time you spent on doing your MPs?
    - And account for time for each type of activity?

- If so, what do you spend the bulk of your time doing?
    - (If not, you should; metacognition is important in problem solving)

(Approximate) Problem solving steps

1. Learn necessary material (i.e., attend lecture/section, read book)
   — May need to revisit after the steps below
2. Read (and understand) problem statement
3. Plan solution
4. Write solution
5. Debug solution

Debugging is a skill (that can be learned)

- Debugging **isn't** some magical innate characteristic
- It is a set of tricks that can be learned, practiced and refined.
  - There are good and bad methodologies
  - Structured vs. Ad-hoc approaches

- What follows is a list of tricks from a book entitled:

  "DEBUGGING: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems," David J. Agans
  - It is a light read (takes just a couple hours).
  - If you are spending many hours debugging your MP's it is probably a very good investment.

☞ DEBUGGING RULES ☜

Understand the System

Make It Fail

Quit Thinking and Look

Divide and Conquer

Change One Thing at a Time

Keep an Audit Trail

Check the Plug

Get a Fresh View

If You Didn't Fix It, It Ain't Fixed

© 2001 David Agans   www.debuggingrules.com

Rule #1: Understand the System

- **Know the fundamentals:** That is what I try to give you in lecture, but that is meant to ease not replace RTFMing.

- **RTFM:** Read the documentation for the system that you are working with. What you might think is a bug, is actually the correct behavior of the system (you might be using it improperly because you don't know what it is supposed to do).
  - Read and understand the code we hand out.

- **Look up the details:** when you need to use "mul" look it up to see exactly how it works.

- **Know your tools:** learn how to use a debugger.

Rule #2: Make it Fail

- **Do it again:** You need to be able to reproduce the bug to study it (find when it occurs) and to be able verify that you fixed it.

- **Stimulate the failure:** Spray a hose on a leaky window so you can see where it leaks.

- **Start at the beginning:** Understand everything that happened that lead to the bug.

Rule #3: Quit Thinking and Look

- **See the failure:** Don't guess at the source of a bug, actually inspect the system to see the bug happening.

- **Build in Instrumentation:** Add code in that spits out intermediate states so you can see when bad things happen.

- **Guess only to focus the search:** Go ahead and guess to make the search faster, but see the bug before you fix the bug.

Rule #4: Divide and Conquer

- **Use a binary search:** Guess a number between 1 and 100 with 7 guesses.

- **Determine which side of the bug you are on:** If the state is bad where you are checking, the bug is upstream.

- **Fix the bugs you know about:** Bugs defend and hide one another. Take' em out as soon as you find them.

Rule #5: Change one Thing at a Time

- **Be Scientific:** Change one thing and observe the change in the output. If you change many things at once, you can't isolate each change's impact.

- **Grab the Brass Bar:** Fully understand the bug, before you change anything. Random changes are more likely to break things than fix them.

- **Change one thing at a time:** Undo the first change before making the second change, so that you have only one change at a time.

- **What did you change since the last time it worked:** If something worked before, the bug is probably in what has changed since it worked.

Rule #6: Keep an Audit Trail

- Write down: what you did, in what order, and what happened as a result.

Rule #7: Check the Plug

- **Question your assumptions:** Make sure that your divide-and-conquer is testing the whole space.

- **Start at the beginning:** Did you turn it on? Did you initialize everything? Are the inputs good?

- **Test the tool:** Are you running the right compiler?

Rule #8: Get a Fresh View

- **Ask for fresh insights:** Sometimes just explaining a problem to someone else can help you identify hidden assumptions and possible bug sources.

- **Tap expertise:** Go to office hours. Work with fellow students (this is why we let you work in groups on the MPs).

- **Report symptoms, not theories:** Don't drag a crowd into your rut. Also, your symptoms need to be more detailed than "it doesn't work."

- **Nothing is too unimportant to be mentioned:** When explaining the problem try not to filter out what you deemed as unimportant; the fact that you think it is unimportant might be why you haven't found the bug.

Rule #9: If you didn't fix it, it ain't fixed.

- **Check that it is really fixed:** If you followed "Make it fail" you can use your reproducible test case to verify that it is working. Do so!

- **Fix the process:** If you notice that when you write code that you often get a certain kind of bug, think if there is a different approach that avoids that kind of bug.

Learn how to use a Debugger.

- (Insert instrumentation)
- Setting break points
- Observe execution state
- Watch code execute (and make sure it does what you think it should)
- Conditional break points