Debugging

This handout has some strategies and tools that can be helpful to use while debugging, especially for CSE 332 projects because there is a considerably larger amount of code and complexity compared to previous UW CSE classes. There are also a few strategies to avoid because they are not effective in these large projects.

Debugging "Non-Strategies"

This section describes strategies that you should *absolutely avoid* when debugging. They lead to extra frustration, often don't help you find the bug, and won't work at all as the programs get larger and more complicated.

Stare and Hope

When you have a complex program... spread out over multiple files... with some code that you didn't write, it's impossible to keep it all in your head. A common attempt at debugging is to *stare at the code* and wait for the bug to "pop" out at you. In CSE 143 level programs, this often works, but it's not a good approach from now on. The intuition on why this doesn't work well is that you're not looking at the states the program gets in–computers are better at being computers than we are. So, you should always be running your program to debug.

"Shotgun" Debugging

A "slightly more advanced" strategy is to make a change that "feels right", then run the program to see if it worked. Then, make a change, then run the program, etc. Although this time you're running the program, The major concern with this strategy is that every time you make a change, you're not *learning anything about where or why the program went wrong*. Every time you run the program, you should be ruling out some potential reason the program is broken.

Debugging Tools

While the approach you take to debugging is the most important thing, the tool you use can also make-or-break your debugging session. We recommend you try both and use whichever one makes more sense in the situation.

`println`

One way of examining state is to... print out the state. One "gotcha" with our testing framework is that you have to use `System.err.println` (rather than `System.out.println`). Whenever you print out state while debugging, you should "mark" it with some string; this way, you know where/when in the program the output is from. A really common way of debugging with `println`s is to surround the potential problem area with a "begin" and an "end" and attempt to narrow down the area that the bug occurs in (more on this later in the handout).

IDE Debugger

Another way of examining state is to use the built-in debugger in whatever IDE you have. If you're going to do this, please make sure to read the tutorial before using it. Make sure you understand the difference between "stepping over" and "stepping in". The debugger can be used in much the same way as the `println` statements, but the approach is different. Instead of surrounding the area that might be broken, you step through until you hit the place it is broken. Unfortunately, you will often have to step through the program multiple times even after you've found the error. The reason for this is that you'll need to step more carefully to figure out *exactly* where things went wrong.

Debugging Strategies

There are a wide variety of approaches to debugging, and you should learn any and all of them. You'll likely end up having a "favorite" approach, but we recommend varying the approach based on the type of bug you've run into.

Tell A Story

A bug is nothing more than a divergence between your expectations (the "story" of what your program is supposed to do) and reality. Debugging is reconciling *the exact place that the stories diverge*. If you can find the place where they diverge, then you can understand what went wrong by examining state right before and after the divergence.

To find the moment of divergence, you should attempt to narrow down the possible scope of the bug. To put it another way, before you start debugging, the error could be anywhere from the first line of `main` to the last line of the program. You can usually very quickly narrow down the scope to a much more reasonable piece of the program. The hard part is narrowing it enough to see exactly where the bug is.

For this, we recommend doing (effectively) a binary search. That is, see if "around the middle of the program" shows the buggy behavior: if it does, then search later; if not, then you can move the end of your scope to the middle. Then, you keep on narrowing until you've found exactly where the bug is.

Run An "Experiment"

Debugging can be treated like a scientific experiment. That is, before running the program, you should make a hypothesis as to what the bug is. Then, design a way to confirm or deny your hypothesis. Finally, run your program and actually confirm or deny your idea. If you weren't right, then come up with a new hypothesis. The important idea here is that you should never edit your program without some sort of idea of where or what the actual bug is.

Fail Fast

Some of the most difficult bugs happen because there is an error early in the program that doesn't show up until significantly later in the program. For example, if you `insert` lots of things into a data structure, you don't actually notice any errors until you do a `find`. Debugging is significantly easier if you make your programs "fail fast". All this entails is checking that all the invariants are met at the beginning and the end of every method. In other words, you should write a `private` method that "checks" the internal structure and if it ever returns false, throw an exception and end your program.

Take A Break

If you've been debugging for a while and you're stuck, *stop debugging*. Take a break. Do something else. Anything else. The more frustrated you are, the less productive and efficient you will be.

Ask For Help

If you have a particularly thorny bug, you've tried all of the above, and you have no idea what's going on, *come talk to us*! We will help! We can walk you through debugging strategies. (We won't do it for you, but we'll do it with you if you've made an attempt.)