General Debugging Practices

1 Intro

You may have been told that programming is the process of writing code that performs a task. That's not quite the case. Programming is the process of writing some code that attempts to perform a task, then discovering that your code doesn't work, figuring out why, fixing it, discovering that it's broken in some other way, figuring out why, fixing that problem, and so on.

It is *normal* that your code doesn't work at first *and you should expect it*. One of the important things we hope to teach you in 122 is how to debug your code. Here are some steps to take to get started down the debugging path, which is where the majority of a programmer's time is spent. **Please try these things before posting on Ed or asking for help at office hours.**

2 My Code Doesn't Work

First, identify what you expected to happen, and what actually happened. (example: I expected `f` to return `10`, but instead it returned `13`). Once you have identified this failure, your next goal is to find the "smallest possible" test case that exhibits this same failure.

What do we mean by "smallest possible"? The end goal is that the least number of lines of code run before the failure. If your function takes in an array of values, generally speaking this will happen when the array contains as few numbers as possible. For an arbitrary data structure, this generally means the simplest data structure possible. Be aware that sometimes the behavior of a function actually ends up being simpler when you pass in a more complex data structure than when you pass in a simpler one.

There is no fool-proof way to identify the smallest possible test case — generally speaking it is simply a guess-and-check process. However, trying to remove parts of the data structure until you can remove no more is a useful heuristic.

Once you have reduced your test case to the smallest possible, your next goal is to narrow down the problem by continuing the "I expected... but instead..." process for smaller and smaller sections of code. One great way to do this is to trace through your test case on paper, try to figure out where your paper function diverges from the one you implemented.

One way to determine where your paper function diverges from the one you implemented is by adding **print statements**. If you're unsure on where the best place to put print statements is, it may be worth reading the Debugging with Print Statements guide. As a rule of thumb, good places to put print statements are:

- at the beginning of the function (especially if it is recursive!)
- right before a function returns
- inside a loop so that it prints each iteration

Another way to determine where your paper function diverges from the one you implemented is by **adding assertions**. The advantage of assertions over print statements is that once you have fixed your bug, you can leave those asserts in, while it is often a good idea to clean up debugging prints once you figure out a bug. The Debugging with Contracts guide goes more in depth on

techniques for using assertions and contracts for debugging! As a good rule of thumb, generally speaking assertions can be put anywhere you can put a print statement.

A third way to determine where your paper function diverges from the one you implemented is by **commenting out sections of code**. This can be useful to help reduce the mental load of trying to keep in mind many moving pieces (when only some are necessary).

2.1 Additional Things to do

While you are debugging your code, make sure that you fully read and try to understand any error messages that appear. Error messages are one of your best windows into what is going on so take the time to read them! If you are unsure as to what an error message is saying, feel free to post on Ed or ask a TA at office hours!

Furthermore, a great tool for debugging is Talking through the code with yourself. Although it seems a bit silly at first, sometimes just walking through the code with yourself is enough to see an obvious flaw. While you are doing this, go through the writeup and make sure you fully understand the requirements and that you are meeting the requirements.

Lastly, if you have been working on a bug for a long time, **take a break**. Step away from the problem for an hour or two. Go eat, go take a walk through Schenley park and *most importantly*, don't think about the problem. When you come back you will be in a much better position to figure out your bug.

3 My code fails on autolab

There are really 3 steps you should follow in order to go from code failing on autolab to code being fixed. These are as follows:

1. Read the Autolab error message. These often provide helpful messages or hints. Read through the Interpreting Autolab Output guide if you are unsure how to interpret the error message
2. Write local test cases until one fails. Read through the Writing a Test File guide if you are unsure about how to write test cases.
3. Congratulations! You now have a failing test case! This is (seriously) a good thing — go to Section My Code Doesn't Work and start from there.

4 How do I test my code

A great tool for figuring out how to test your code is the Writing a Test File guide. A short list of some important and useful concepts in that guide for how to write good test cases is as follows:

- Often we give you examples in the writeup. These are excellent test cases for you to write. Often Autolab runs these exact tests on your code. Win!
- For each conditional in your code, write one test case that makes the conditional true and one that makes it false.
- Test **edge cases**: 0 or 1 element arrays, inputs that are identical as well as completely dissimilar to each other, numbers that are large, 0, positive and negative numbers, etc. You shouldn't test things that violate the function's preconditions, and very large arrays make c0 unhappy, so stay away from those.

Note: It is hard to test every conditional exhaustively — with 10 independent conditionals, there are over 1000 unique configurations of which conditionals evaluate to true and which evaluate to false. As a result, the second bulletpoint is more of a guide rather than a direct recommendation.

5 How do I do this task?

Obviously there is no magic bullet to help you figure out how to do a task, but generally speaking, one of the following ideas will help you figure out how to do a task:

- Read the statement for the task in its entirety and **underline** anything that strikes you as important
- Draw an example
- Find a pattern — break the task up into different cases or separate tasks
- Walk through an example on paper and figure out how you (as a human) are doing the task
- Search the C0 Library Reference for a function that can help
- Go back through the write-up **carefully** and make sure you haven't missed any hints or comments that might help.