How to Test a Program

- **Test small chunks.**

  As much as possible, test each function or class method as you write it and test after each logical unit has been written (even if it is just part of a larger function or method). This may seem tedious, but it is much easier to track down a bug in a small section of code than in a large one. As you gain experience, you may feel confident enough in your ability to write correct code and track down bugs and so your chunks may become a little larger – but you still shouldn't write a large program without testing as you go.

- **Be thorough with your tests.**

  With experience, you'll build a repertoire of common special cases that need to be tested for typical problems.

  Augment these tests (or come up with tests in the first place) by considering various aspects of the program:

  - Look at function comments. Make sure you test "normal" cases (where you expect the function to carry out its task) as well as error cases (where the function cannot carry out its task). The comments should identify what happens if the function gets certain input that it can't work with – make sure you test those inputs, and verify the function's behavior. (If you wrote the function and it doesn't have these comments, write them!)
  - Look at the flow of control in each section of code, and make sure that every line of code is executed by at least one test. For example, an 'if' statement should have a test which causes the test condition to be true, and one which causes it to be false. For loops it is a good idea to test cases where the test condition is false the first time and where the loop body executes at least once. If you have a boolean condition involving || or &&, make sure your tests address every possible way in which the condition can be true or false. For example, if a test condition a || b, have tests where a is false and b is true, where a is true and b is false, where both are true, and where both are false. (Sometimes this isn't possible, but test as many cases as you can.)

- **Always have in mind what the output *should* be before looking at what it *is*.** It is all too easy to fall into the trap of assuming the output is right because the computer produced it – the computer, however, is only as correct as the program it is running.

  Sometimes you don't know the correct answer (after all, the point of the program was to compute it!). In these cases, think about what sort of properties a correct answer should have – while you might not be able to prove the answer is correct just by looking at the output, you can still spot incorrect answers.

  A real example: two students were working on a program which carries out a simulation of customers in line at a bank – customers arrive at the bank and go to the back of the line; eventually a customer moves to the beginning of the line, is served by a bank teller, and leaves the bank. The simulation was run for 500 time units, and a customer arrived at the bank once every 100 time units (on average). About how many customers would you expect to be served in one run of the simulation? Roughly 5? (It might be a few more or less because the every 100 time units is an average, so customers may arrive closer together or farther apart.) You don't know the exact answer, but you can tell that you've got a problem if the program says 147 customers were served. (Even 10 customers might be enough to make you suspicious, or at least warrant further checking.)

How to Avoid Debugging a Program

The best strategy is not to have to debug in the first place. Of course, this is easier said than done – even experienced programmers have to debug. There are, however, some strategies which can help reduce the likelihood of errors (and thus having to debug). For example:

- **Plan out your algorithm – and test it by working through some examples – before implementing the program.** (Working through examples can also help you develop the algorithm.) If you are not clear in your head about what you are trying to implement, it'll never work correctly.

- **Double-check the logic of common cases immediately after writing a section of code.** For example, it is easy to be off-by-one in loop conditions and cause a loop to execute one too many times or too few – so identify the last time you expect the loop condition to be true and the first time it should be false, and verify that the code reflects that. Another example – it is easy to mix up || and &&, particularly because English lets you be a little sloppy with which you use. So, after you write that boolean condition, ask yourself when you want it to be true (do all conditions need to be true, or just one) or, conversely, when you want it to be false (all conditions false or just one) and make sure the code reflects that.

- **Write comments – for classes, functions, and variables – as you write the code, not after.** Writing the comment forces you to think clearly about what a class, function, or variable is for (to help cement it in your mind) and provides a reference for later should you need to clarify the role of that class, function, or variable.

- **Write comments internally in the code as you write the code, not after.** Note what a section within a function is for, an assumption you make, etc. - writing the comment again helps to cement concepts in your head (as well as making it easier for someone else to figure out later).

- **Check frequently that the way you are using a class, function, variable is consistent with the comments you wrote for it.** This is most commonly a problem with variables – for example, attempting to use the same variable for two different purposes, or using the wrong variable somewhere.

- **Make use of tools like assertions.**

  This doesn't prevent you from having to debug, but it can greatly speed the debugging – when you think "I expect this value to be such-and-such here" then use an assertion to programmatically make that check. Even if your program works correctly the first time, thinking about putting in assertions helps you cement in your mind how you expect the program to behave.

How to Debug a Program

Even with the best of intentions, programs still need debugging. A methodical approach can decrease the time it takes you to find the bug, and can help provide inspiration if you get stuck.

1. **Observe the behavior of the program.**
   What are some inputs for which it doesn't work correctly? - try to find the shortest/simplest case for which the program doesn't work correctly (if such a thing can be identified)
   What are some inputs (if any) for which it *does* work correctly? - perhaps you can identify a property that makes the cases that do work different from the ones that don't
   Does it always do the same thing given the same inputs? - programs with a random element may not always do the same thing, and this needs to be considered when trying to find a bug (and when testing)

2. **Formulate one or more theories about what could be happening in the program to explain the behavior you've observed.**
   Theories may take several forms. One strategy is to make theories about the location of the bug e.g. "I think the bug is in function A" or "I think the bug in somewhere between these two lines of code". Another is to make theories about the program's flow of control or variable values e.g. "I think the 'else' part of this 'if' statement is being executed instead of the 'if' part" or "I think variable x is getting a bad value".
   Include as many theories as occur to you – just one theory is enough to get started, but if several possibilities occur to you, include them in your list. However, don't feel like you have to come up with every possible theory – there's time for that if the ones you come up with first don't work out.
   Include all the theories that you come up with – do not exclude a theory based on whether or not you think it is likely. (For example, you might think there's no way the test condition of your 'if' statement could be false, but don't exclude the theory that the 'else' part is being executed mistakenly if, in fact, somehow executing the 'else' part would explain the behavior you've observed. It could be you're wrong about the impossibility of the test condition being false.) Instead, use likelihood as one criteria for choosing which theory to tackle first.
   Always come up with at least one theory – at the very least, you can theorize that the problem occurs somewhere between the beginning of the program and the end, but it is often possible to make a better guess than that. For example:
   - If the program never exits, it may be waiting for input or be stuck in an infinite loop.
   - If the wrong alternative is chosen, it may be a problem with an 'if' statement.
   - If the first part of the program ran successfully, then the bug is likely after that point.
   - Segmentation faults are typically memory-access problems e.g. pointer errors or array-indexing problems. A segmentation fault which occurs at the very end of a program (or function) may be due to errors in the destructor(s) for objects which go out of scope at that point.
   - If you have just added a new section of code – and the program worked before – the bug may be in the newly-added section.

   As you debug more programs, think about how you can expand this list.

3. **Figure out how to rule out (or support) your theories.**
   View this process as designing a set of experiments to test your theories. The goal is to identify what kinds of things you'll do to test the theory (e.g. identify where you'll print out information and what you'll print) *and* what you expect to see if your theory is correct and if it isn't. Having what you expect to see in mind before running the program again helps keep you out of the trap of believing whatever the computer produces because it is a computer, and helps you think about whether the test you are considering will really test what you want.
   Tackle your theories one at a time, starting with the most likely and/or the easiest to test. You don't need to test them all at once! Some theories may be mutually exclusive, so one experiment can potentially disprove several theories at once – these might be good theories to tackle first. Don't spend too much time trying to decide which theory to test first – the only cost of picking the "wrong" one is taking more time to locate the bug.
   There are many strategies for testing theories – which you choose depends on the theory, the tools available, and what you feel most comfortable with. Possibilities include:
   - Trace through the code by hand, either mentally or on paper carrying out the tasks that the program is carrying out. This can work well if the bug is not too subtle, and if you feel confident in the semantics of the programming language (so you can do what the code says and not what you think it does).
   - Insert line(s) to print out information. You can print any identifiable information if the goal is just to determine the flow of control in the program (did the computer get to a particular point or choose a particular alternative). Often it is useful to print out the value of variables, to see if they are correct or to try to diagnose why the flow of control is what it is (e.g. print out the variable(s) involved in the test condition if you are trying to figure out why the 'else' case of an 'if' is being executed when you don't expect it to be).
   - Insert breakpoints or watchpoints in a debugger to let you stop the program and examine variables at key points.
   - Remove (comment out) code. This can help you locate a bug in several ways. If removing the code causes the program to work correctly or makes the bug go away, the bug may be in the code that was removed. Alternatively, removing code can help you find the smallest set of circumstances in which the bug still occurs – again helping you to narrow down where the bug is.

4. **Run the experiments and refine the set of theories.**
   Implement your experiment(s) and re-run the program.
   Based on the results of your experiment(s), you may be able to rule out some theories (e.g. if the 'else' part of the 'if' in fact isn't being executed, the problem isn't being caused by the 'else' part being executed when it shouldn't be) or refine others (e.g. by determining the value in question is still OK at a certain point of the code, you can narrow down the region where the value goes bad). You may also come up with new theories (sometimes this is necessary, if you've ruled out all of the theories you had).

5. **Repeat steps 2-4 as needed until you've identified both the location *and* the root cause of the problem.**
   In many cases, you're done when you've found the location where something goes wrong.
   Sometimes, however, you need to dig deeper to make sure you've found the root cause of the problem and not just a symptom. Having in mind what values ought to be helps with this.
   Example #1: You've traced a segmentation fault to a line where a NULL pointer is dereferenced. At this point, you need to determine whether or not NULL is a legal value for that pointer – if so, you've found the bug and the cause (you didn't handle that case properly). If not, the question becomes how it got a NULL value (so back to the theories and the testing since the actual bug – where the pointer got its NULL value – is somewhere else). If you don't know whether or not NULL is legal, you need to understand the design of the program before you can continue.
   Example #2 (real example): The bank simulation program, again. Among other things, the program computed the average time customers wait in line. The students found that the waiting times being computed were negative, traced the negative value to the line which computed waiting time = departure time – arrival time, and changed it to waiting time = arrival time – departure time to fix the bug. However, the problem is not the order of subtraction (thinking about how the waiting time *should* be computed addresses that – departure times ought to be later than arrival times, so departure – arrival is the correct calculation), but rather in something else which caused the departure to be before the arrival. (It turned out this was due to a bug in the scheduling of the various events – like customer arrivals and departures – in the simulation.)

6. **Fix the bug, and resume testing.**
   Run three sets of tests: the test that you just fixed, any tests you ran previously that worked (to make sure that you didn't break anything by fixing what you just fixed), and new tests to test cases you haven't already tested.