Generating Test Cases

/run/media/lijunjie/资料/Learning Video/writing-running-fixing-code/03_testing-and-debugging/01_testing/06_generating-test-cases_instructions.html

One difficulty with testing arises from the fact that you want to test the cases you did not think of—but if you do not think of these cases, then how do you know to write tests for them? One approach to such a problem is to generate the test cases according to some algorithm. If the function we are testing takes a single integer as input, we might iterate over some large range (say -1,000,000 to 1,000,000) and use each integer in that range as a test case.

Another possibility is to generate the test cases (pseudo-)randomly (called, unsurprisingly, *random testing*). Note that *pseudo-random* means that the numbers look random (no “obvious” pattern) to a human, but are generated by an algorithm which will produce the same answer each time if they start from the same initial state (called a “seed”). Random testing can be appealing as it can hit a wide range of cases quickly that you might not think of at all. For example, if your algorithm has 6 parameters, and you decide you want to test 100,000 possibilities for each parameter in all combinations, you will need 100,000^6 = 10^30 test cases—even if you can do 10 trillion test cases per second (which would be beyond “fast” by modern standards), they will take about 3 million years to run! With random testing, you *could* run a few thousands or millions of cases, and rely on the Law of Large Numbers to make it likely that you encounter a lot of varieties of relationships between the parameters.

One tricky part about generating test cases algorithmically is that we need some way to verify the answer—and the function we are trying to test is what computes that answer. We obviously cannot trust our function to check itself, leaving us a seeming “chicken and egg” problem. In a very few cases, it may be appealing to write two versions of the function which can be used to check each other. This approach is appropriate when you are writing a complex implementation in order to achieve high performance, but you could also write a simpler (but slower) implementation whose correctness you would more readily be sure of. Here, it makes sense to implement both, and test the complex/optimized algorithm against the simpler/slower algorithm on many test cases.

We often test properties of the answer to see that it correct. For example, if we are testing an algorithm to compute the square root of a number, we can check that the answer times itself produces the original input (that is that the square root of *n* -squared equals *n* —or is within the expected error given the precision of the numbers involved.) Testing this property of the answer assures us that it was correct without requiring us to know (or be able to compute by other means) what it is. The previous technique works well for invertible functions (that is, where we can apply some inverse operation that should result in the original input), but many programming problems do not fit into this case.

We may, however, be able to test other properties of the system to increase our confidence that it is correct. For example, imagine that we are writing software to simulate a network, which routes messages from sources to destinations (the details of how to implement this are beyond the skills we have learned so far, but that is not important for the example). Even without knowing the right answer, we can still test that certain properties of the system are obeyed: every message we send should be delivered to the proper destination, that delivery should happen exactly one time, no improper destinations should receive the messages, the delivery should occur in some reasonable time bound, an so on.

Checking these properties does not check that the program gave the right answer (*e.g* ., it may have routed the message by an incorrect but workable path), but it checks for certain classes of errors on those test cases. As with all test cases, this increases our confidence that the program is right, but does not prove that it is right. Of course, we would need to test the other aspects of the answer in other ways—which may involve looking at the details of fewer answers by hand to ensure all details are right.

This approach requires development of code which is not part of the main application—a program which not only sends the requests into the network model, but also tracks the status of those requests and checks for the proper outcomes. This additional program is called a *test harness* —it is a program you write to run and test the main parts of your code. Developing such infrastructure can be time consuming, but is often a good investment of time, especially for large projects.