Challenge: Building Test Cases for merge()

Help Center

As part of testing your implementation of `merge()` from week one, you ran your implementation on a small collection of example inputs such as `[0, 0, 2, 2]` to determine whether it computed correct answers. During OwlTest, your implementation of `merge` was also tested using a much larger set of input examples designed to more thoroughly test your code. These input examples are referred to as *test cases*. For your tests, you computed the correct answer for each test case by hand and compared this answer to the answer returned by your implementation of `merge`. In OwlTest, we compute the correct answer for each test case using a reference implementation that we created.

During the course of implementing `merge`, a common experience for many students was having their implementation run correctly on their own test cases, but then fail unexpectedly on OwlTest. This experience is due to the fact that most students create far too few test cases. Intermediate programmers often suffer from a case of wishful thinking when their program runs on their own small set of test cases and think "There is no way my program can have any more bugs." In practice, more bugs are often present.

To avoid this experience, the solution is to choose "better" sets of test cases. In this activity, we will share some strategies for choosing test cases and then offer a challenge that involves building "good" sets of test cases for `merge()`.

Strategies for choosing test cases

Before discussing specific strategies, we first discuss what "better" and "good" mean when referring to sets of test cases. Here, "better" means that one set of test cases is more likely to detect errors in your programs than another. Defining what is meant by a "good" set of test cases is very subjective and often depends on the consequences of allowing an incorrect implementation to be used in practice. For example, a set of test cases that has 99.9% chance of catching an error in a student's implementation of `merge` for this class is a "good" set of test cases. However, if a program tested in this manner was going to be used as part of the flight control system for a commercial airliner, a 0.1% chance of failure wouldn't make many airline passengers happy.

The task of choosing "good" sets of test cases is a tough one even for experienced programmers. For this activity, we suggest four rules to follow when creating test cases:

1. Create more test cases than you think is necessary. Our experience in examining incorrect implementations of `merge` is that chance of an incorrect program passing a set of test cases drops quickly as the size of the set grows. While you may feel silly adding more test cases when you are sure that your program is correct, experience shows that adding more test cases always improves the likelihood that you program is correct when it passes these test.
2. Create lots of small test cases and a generous selection of larger test cases. Small test cases are easier to create. Take advantage of this fact and create lots of them. If the cost of failure is high for your application, consider testing it on all small inputs. Augment these tests with a collection of larger test cases whose size may be beyond what you expect to run in practice. For example, we have seen several cases where student implementations of `merge` run on all list of up to length four, but fail on a list of length five such as `[2, 2, 4, 4, 2]`.
3. Choose sets of test cases that execute all of the logical components of your code. Code that is never executed is likely to have a bug. Choose your test cases with the idea that every part of

your program should be executed during the evaluation of at least one test case in the set. For example, if your program includes an `if` statement, build multiple test cases with the goal of having each case test one clause of the `if`. For loops, choose test cases such the body of the loop is executed at least once. If possible, add test cases that cause the body be executed zero times as well. Make sure that all helper functions are tested.

4. Add test cases that arose during the debugging of your program. Finally, you will encounter inputs that expose bugs in your code. Always, add these inputs as test cases. One common trait of human nature is to repeat your errors. Adding the test case that caused an error will help prevent an undetected repeat of this error.

Challenge: Building "good" sets of test cases for merge()

To provide you with an opportunity to hone your skills as a tester, we have designed an activity that involves you creating a set of test cases for `merge` and then submitting this set of test cases to OwlTest for an assessment of the quality of these test cases. The Owltest is available here. Your submission to OwlTest should consist of two Python definitions:

- A definition for the function `merge(line)` from 2048.
- A list `TEST_CASES` containing test cases for the function `merge()`.

First, Owltest will check the correctness of your implementation of `merge` using the test cases in the list `TEST_CASES`. If your implementation of `merge` fails on one of your own test cases, OwlTest will report this fact and terminate with a score of zero. You should fix your implementation so that it runs on your test cases and resubmit.

Next, OwlTest will assess the test cases in `TEST_CASES` versus a collection of implementations of `merge` that return an incorrect answer on some input. (To simplify this activity, OwlTest ignores the issue of mutating the input list.) In particular, OwlTest will run each incorrect program on the test cases in `TEST_CASES`. If one of these incorrect implementations produces correct results on all test cases in `TEST_CASES`, the set of test cases in `TEST_CASES` needs to be augmented. In this case, OwlTest will display the implementation for this incorrect version of `merge()` so you can add new test cases to `TEST_CASES` that cause this program to fail.

Your score for the activity will be the percentage of incorrect implementations that your set of test cases detects. As your set of test data grows, you may find that you need to debug the provided incorrect version of `merge` to determine which test cases to add to `TEST_CASES`. Expect this part of the activity to be challenging. Debugging another person's code is usually a tough task. If every incorrect implementation of `merge()` fails on at least one test case in `TEST_CASES`, your set of test cases is deemed complete and receives a score of 100.

We will post a thread in the forums where everyone can report their score as well as the number of test cases used to generate that score. Good luck and have fun!

Created Thu 19 Feb 2015 1:20 PM PST
Last Modified Sat 21 Feb 2015 2:12 AM PST