# Practical Tips for Designing Test Cases



Writing good tests cases requires a lot of thought: you need to think about a wide variety of things that can go wrong. Here are some suggestions to guide your thinking, as well as important lessons. First, some thoughts for testing error cases:

- Make sure your tests cover every error case. Think about all the inputs that the program cannot handle, i.e., ones where you would expect to receive an error message. If the program requires a number, give it “xyz”, which is not a valid number. An even better test case might be “123xyz” which starts out as a number but isn’t entirely a number. How the program should handle this depends on the requirements of the program, but you want to make sure it handles it correctly.

- Be sure to test “too many” as well as “too few”. If a program requires exactly N things, test it with at least one case greater than N and at least one case with fewer than N. For example, if a program requires a line of input with exactly 10 characters, test it with 9 and with 11.

- Any given test case can only test one “error message and exit” condition. This means that if you want to test two different error conditions, you need two different test cases: one for each error condition. Suppose that our program requires a three letter string of lower-case letters. We might be tempted to test with “aBcD” to test two things at once, but this will only test one (and believing you tested both is problematic!) To see why this rule exists, think about the algorithm to check for these errors:

Check if the input string does not have exactly three letters

If it does not then:

*print “Error: the input needs to be 3 letters”*

exit the program

Check if the input string is made up entirely of lowercase letters

If it does not then:

*print “Error: the input needs to be all lowercase letters”*

exit the program

Violating one input condition will cause the program to exit! This is also true of other situations where the program rejects the input, even if does not exit

Test exactly at the boundary of validity. If a program requires between 7 and 18 things, you should have test cases with 6, 7,8, 17, 18, and 19 things. You need to make sure that 6 and 19 are rejected while 7, 8, 17, and 18 are accepted. Testing exactly at the boundaries is important because of the common "off by one" mistake---maybe the programmer wrote < when he should have written <= or >= when he should have written > or something similar. If you test with values that are right at the boundary, you will find these mistakes.

However, testing is not just about checking error handling. You want to make sure that the algorithm correctly handles valid inputs too. Here are some suggestions:

- Think carefully about whether or not there are any special cases where one particular input value (or set of values has to be treated unusually). For example, in poker an Ace is usually ranked the highest, however, it can have the lowest ranking in an "Ace Low Straight" (5 4 3 2 A). If you are testing code related to poker hands, you would want to explicitly test this case, since it requires treating an input value differently from normal

- Think carefully about the requirements, and consider whether something could be misinterpreted, easily mis-implemented, or have variations which could seem correct. Suppose your algorithm works with sequences of decreasing numbers. You should test with a sequence like 7 6 6 5 4, which has two equal numbers in it. Checking equal numbers is a good idea here, since people might have misunderstood whether the sequence is strictly decreasing (equal numbers don't count as continuing to decrease) or non-increasing (equal numbers do count).

- Think about types. What would happen if the programmer used the wrong type in a particular place? This could mean that the programmer used a type which was too small to hold the required answer (such as a 32-bit integer when a 64-bit integer is required), used an integer type when a floating point type is required, or used a type of the wrong signedness (signed when unsigned is required or vice versa).

- Consider any kind of off-by-one error that the programmer might have been able to make. Does the algorithm seem like it could involve counting? What if the programmer was off by one at either end of the range she counted over? Does it involve numerical comparison? What if < and <= (or > and >=) were mixed up?

- Whenever you have a particular type of problem in mind, think about how that mistake would affect the answer relative to the correct behavior, and make sure they are different. For example, suppose you are writing a program that takes two sequences of integers and determine which one has more even numbers in it. You are considered that the programmer might have an off-by-one error where he accidentally misses the last element of the sequence. Would this be a good test case?

Sequence 1: 2 3 5 6 9 8

Sequence 2: 1 4 2 8 7 6

This would not be a good test case for this particular problem. If the program is correct, it will answer “Sequence 2” (which has 4 compared to 3). However, if the algorithm mistakenly skips the last element, it will still answer “Sequence 2” (because it will count 3 elements in Sequence 2, and 2 elements in Sequence 1). A good test case to cover this off-by-one-error would be

Sequence 1: 2 3 5 6 9 8

Sequence 2: 1 4 2 8 7 3

Now a correct program will report a tie (3 + 3) and a program with this particular bug will report Sequence 2 as having more even numbers.

**Consider all major categories of inputs, and be sure you cover them.**

- For numerical inputs, these would generally be negative, zero, and positive. One is also usually a good number to be sure you cover
- For sequences of data, your tests should cover an empty sequence, a single element sequence, and a sequence with many elements.
- For characters: lowercase letters, uppercase letters, digits, punctuation, spaces, non-printable characters
- For many algorithms, there are problem-specific categories that you should consider. For example, if you are testing a function related to prime numbers (e.g., isPrime), then you should consider prime and composite (not prime) numbers as input categories to cover.
- When you combine two ways to categorize data, cover all the combinations. For example, if you have a sequence of numbers, you should test with an empty list, a one element sequence with 0, a one element sequence with a negative number, a one element sequence with a positive number, and have each of negative, zero, and positive numbers appearing in your many-element sequences.
- An important corollary of the previous rules is that if your algorithm gives a set of answers where you can list all possible ones (true/false, values from an enum, a string from a particular set, etc), then your test cases should ensure that you get every answer at least once. Furthermore, if there are other conditions that you think are important, you should be sure that you get all possible answers for each of these conditions. For example, if you are getting a yes/no answer, for a numeric input, you should test with a negative number that gives yes, a negative number that gives no, a positive number that gives yes, a positive number that gives no, and zero [zero being only one input, will have one answer].

- All of this advice is a great starting point, but the most important thing for testing is to think carefully---imagine all the things that could go wrong, think carefully about how to test them, and make sure your test cases are actually testing what you think they are testing.