


# ECE 322 Lab Report #1

 Benjamin Kong  
1573684

October 5, 2021

## Introduction

The purpose of this lab was to introduce **B**lack-box testing techniques by applying various **B**lack-box testing methods such as dirty testing, error guessing, and partition testing.

**B**lack-box testing is a method of software testing that examines a program without knowledge of the internal workings. In this lab, we focus on three **B**lack-box testing methods:

- Dirty testing: Testing a component or system in a way that it was not designed to be used,
- Error guessing: Deriving test cases from experience (such as divide by zero) to find errors in the program, and
- Partition testing: Divide the input data into partitions from which we can derive test cases to test each partition in order to cover as many scenarios with as few test cases as possible.

## Part 1 — Calculator Program

### Q1

In part 1, we are testing a simple calculator program. This calculator has basic functions such as addition, subtraction, multiplication, and division. We will be testing the correctness of the calculator (i.e. testing if we get the expected output for a variety of inputs), testing invalid inputs, extremely large numbers, and incorrect syntax.

### Q2

To test the calculator, we will be using dirty testing and error guessing in order to try and find errors in the calculator program (see the table in Q4 to see the test cases). Dirty testing

involves testing a program in a way it was not designed to be used. It's good to see if the program handles errors correctly. For example,

- Huge number: In test 3, we enter an enormous number into the calculator. The calculator errors. This is likely because the calculator cannot handle large numbers correctly.
- Invalid syntax: In test case 7, we expect that  $5^3$  returns NaN. However, it returns 1. This may be because the program assumes we want to do  $5^{0^3}$ , which would return 1.
- Invalid syntax: In test case 8, we have a space between the numbers and the operator. We would expect this to return 495; however, the program returns NaN. This may be because the program doesn't handle spaces correctly.
- Invalid syntax: In test case 10, we have  $5(2)$ . The calculator seems to not interpret this as us wanting to do  $5 \times 2$ .
- Invalid syntax: In test case 11, we have two negative signs between two numbers. Normally, we would expect this to mean  $1-(-1)$ . However, the calculator errors. This may be because the calculator doesn't correctly handle two operators between two numbers.

Error guessing involves trying to guess where an error will occur based on experience as a programmer. This is good to see if the program handles edge cases correctly. For example,

- Order of operations: we expect the program to follow order of operations. As seen in test 18, however, the calculator subtracts 8 from 64 before squaring the result. This is incorrect, as the calculator should square 8 and subtract there result from 64. This may be because the calculator doesn't handle all **B**EDMAS cases correctly.

### Q3

While we were able to find many errors with our program, we certainly did not find all the errors of the program. This is because it is hard to cover all scenarios with **B**lack-box testing as we don't know the inner workings of the program. Dirty testing and error

guessing depend on the tester's experience and creativity; this can result in missed scenarios as it requires a lot of experience and effort to come up with unusual test cases that may break the program.

## Q4

Test case table. The rows highlighted in red represent test cases that failed.

TestID	Input Desc.	Expected	Actual
1	0+1	1	1
2	1+1	2	2
3	3^400	7.06E+190	NaN
4	NaN+NaN	NaN	NaN
5	NaN+2	NaN	NaN
6	Entering nothing		0
7	5^^3	NaN	1
8	500 - 5	495	NaN
9	5 ^2	25	25
10	5(2)	10	52
11	1--1	2	NaN
12	5*(2)	10	10
13	2/5	0.4	0.4
14	10+10	20	20
15	3+3/10	3.3	3.3
16	9+9.9	18.9	18.9
17	8^(1+1)	64	64
18	64-8^2	0	3136
19	2+2+2	6	6
20	(2*(3+5))	16	16
21	16^0.5	4	4
22	100*100/100	100	100
23	5/6-	NaN	NaN
24	0.1+0.2	0.3	0.3
25	9/10	0.9	0.9

## Part 2 — Triangle Classification Program

### Q1

The program takes in three inputs,  $a$ ,  $b$ , and  $c$ . These represent the side lengths of a triangle. The program takes in these inputs and returns the type of triangle that would be formed from a triangle with the given side lengths: isosceles, scalene, or equilateral. If invalid input is entered, the program should return an error message.

For this program, we will be employing partition testing. Partition testing involves partitioning the input space in order to reduce the number of test cases. Test cases are selected in such a way that all equivalence classes are covered. Equivalence classes are set of inputs we believe will cause the system to behave identically (ex. for an addition program, two positive numbers would form an equivalence class).

### Q2


#### Triangle Equivalence Classes

Input Condition	Valid Equiv. Classes	Invalid Equiv. Classes
Triangle type	Isosceles (1) Scalene (2) Equalateral(3)	N/A
Valid triangle	$a + b > c$ (4)	$a + b = c$ (7) $a + b < c$ (8)
Number of input arguemnts	3 (5)	No arguments (9) Less than 3 (10) Greater than 3 (11)
Argument type	Positive integer (6)	Negative integer (12) Non-integer (13) Non-numeric (14)

### Q3

Refer to Q5 for the test cases. As seen in the test case table, for test case 13, we get an unexpected result when  $a + b = c$ . In reality, this is not a possible triangle (as this would just be a line). However, the triangle program returns that this is an isosceles triangle. This may be because the program doesn't check the edge case where  $a + b = c$  when determining if the triangle is valid.

### Q4

The test method employed here (partition testing) was able to effectively find an error in the program. There were significantly less test cases than dirty testing/error guessing as we only checked equivalence classes. However, since we are doing lack-box testing and don't know the inner workings of the program, there may be unknown behavior that might be uncovered via other testing methods such as error guessing.


### Q5

Test case table. The rows highlighted in red represent test cases that failed.

TestID	Input Desc.	a	b	c	Expected	Actual
1	Isosceles	5	5	3	Isosceles	Isosceles
2	Scalene	13	9	14	Scalene	Scalene
3	Equilateral	2	2	2	Equilateral	Equilateral
4	No arguments				ERROR: Not enough arguments	ERROR: Not enough arguments
5	Not enough arguments	1			ERROR: Not enough arguments	ERROR: Not enough arguments
6	Too many arguments	1	1	1 1	ERROR: Too many arguments	ERROR: Too many arguments
7	Argument with one negative	-1	1	1	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value

8	Argument with two negatives	-1	-1	1	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
9	Argument with three negatives	-1	-1	-1	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
10	Argument with decimal	1.1	1.1	1.1	ERROR: Invalid argument - non integer	ERROR: Invalid argument - non integer
11	Argument with zero	0	1	1	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
12	All arguments zero	0	0	0	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
13	$a + b = c$	3	3	6	ERROR: Invalid triangle	Isosceles
14	$a + b < c$	3	3	7	ERROR: Invalid triangle	ERROR: Invalid triangle
15	String input	a	b	c	ERROR: Invalid argument - non integer	ERROR: Invalid argument - non integer

## Conclusion

In this lab, we were introduced to three  lack-box testing techniques: dirty testing, error guessing, and partition based testing. We performed dirty testing and error guessing on a simple calculator program and found some errors. We also performed partition based testing on a triangle program and found an error for an edge case. We saw that each technique has its advantages and disadvantages:

- Dirty testing can handle unexpected inputs, but may require a lot of creativity, experience, and test cases.

- 
- Error guessing can help reveal random errors, but again requires creativity, experience, and lots of test cases.
  - Partition testing reduces the number of test cases, but without knowing the inner workings of the program, there may still be errors lurking.