1 Introduction

The purpose of this lab was to serve as a practical introduction to rudimentary black-box testing techniques. Although there are an extensive number of black-box testing techniques, in this lab, we were only introduced to generic methods such as dirty testing, error guessing, and partition testing. Black-box testing is testing with no knowledge of the internal structure or implementation details of the program.

2 Part 1 – Calculator Program

Q1

The application being tested is a simple calculator that implements a number of standard calculator operations such as addition, subtraction, multiplication,and division. In general, we are trying to write test cases to check if the calculator function as expected. This may involve testing if the buttons are functional, testing character input, testing small/large numbers, testing incorrect syntax, and testing order of operations.

Q2

We are using dirty testing and error guessing to test the simple calculator. Dirty testing and error guessing usually involves testing out of the box. Sometimes using the tester's previous experience of testing similar products may be helpful. These kinds of tests aim to break the software to show that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests. Q4 contains the full list of test cases where the highlighted test cases are the ones with unexpected results. From experience, I believe the following could be reasons for the failures:

1. 2^{150} is an extremely large number. The calculator perhaps was not designed to support such large numbers.
2. It's likely that numbers separated by a space, such as 1 1, simply get concatenated to 11. Because of this, when numbers are encased in brackets (1)(1), it also concatenates them instead of multiplying them.
3. When there's a bracket followed by a minus sign, such as $(2^3) - 1$ the program may get confused and think that there's two numbers (2^3) and -1 with no operation between them. This could be why the output is NaN.
4. When there's several addition symbols back to back such as $2 + + + 2$, it seems like the program assumes every plus after the first plus is used to describe a positive number. This is why the actual result we got was 4.

Q3

The testing methods employed in this section were able to find several errors in the calculator program, but was not able to identify all errors. It is possible that there are many more errors in the program that we were not able to identify. This testing method requires creativity and experience to come up with unconventional test cases and is limited by the tester's creativity. This can be unreliable and may result in lots of unnecessary test cases.

Q4

Test ID	Description	Expected	Actual
1	9+10	19	19
2	1-1	0	0
3	2^2	4	4
4	2^150	1.81E+75	NaN
5	2(2)	4	22
6	2*2	4	4
7	3+2*2	7	7
8	2*2+3	7	7
9	(3+2) *2	10	10
10	(2^3)+1	9	9
11	(2^3)-1	7	NaN
12	(2^3)/2	4	4
13	2^-1	0.5	0.5
14	2^(-1)	0.5	0.5
15	(2)^(-1)	0.5	0.5
16	2-(-2)	4	NaN
17	2(-2)	-4	NaN
18	3+2	5	5
19	2+2+2	6	6
20	2*2*2	8	8
21	2+++2	NaN	4
22	(2+2	NaN	NaN
23	2^2^2	16	16
24	2+()	NaN	NaN
25	2+	NaN	NaN

3 Part 2 – Triangle Classification Program

Q1

The application being tested is a command line triangle classification program. The triangle application reads in three space separated positive integers (a, b, c) representing the length of each side of the triangle, and outputs either the type of triangle (scalene, isosceles, or equilateral) or prints an error message to the console. In this section, we will be using partition testing. In partition testing, we try to categorize test cases into equivalence classes. Through partition testing, we try to cover as many possibilities as possible with as few test cases as possible. This usually results in a lower amount of test cases.

Q2

Triangle Equivalence Classes

Input Condition	Valid Input Class	Invalid Input Class
# of input args	3 input args	<3 input args >3 input args
Type of args	Positive ints	Neg ints Decimal 0
Type of triangle	Equilateral Isosceles Scalene	a + b = c a + b <c

Q3

In Q5, we can see a list of our test cases. The highlighted test cases are the test cases that had an unexpected outcome. There was one test case that failed our test. This was for the test case where $a = 2, b = 2, c = 4$. We expected the program to output "ERROR: Not a valid triangle", but it actually said that it was an isosceles triangle. This is likely because the program only checks if $a + b > c$ instead of $a + b \geq c$.

Q4

The testing methods employed in this section were able to find some important errors in the program. Partition-based testing resulted in having significantly less tests than dirty testing and still was effective in finding the more obvious errors; however, it seems to be less effective at testing unique cases.

Q5

Test ID	a	b	c	Description	Expected	Actual
1	2	2	2	Equilateral Test	Equilateral	Equilateral
2	2	3	2	Isosceles Test	Isosceles	Isosceles
3	2	3	4	Scalene Test	Scaelene	Scalene
4	2	3		Missing an agument	ERROR: Not enough arguments	ERROR: Not enough arguments
5	2	3	0	0 as an argument	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
6	2 2	3	4	Too many arguments	ERROR: Too many arguments	ERROR: Too many arguments
7	!	2	3	Special character	ERROR: Invalid argument - non integer	ERROR: Invalid argument - non integer
8	0.1	2	3	Decimal	ERROR: Invalid argument - non integer	ERROR: Invalid argument - non integer
9	-1	2	3	Negative number	ERROR: Invalid argument - non positive value	ERROR: Invalid argument - non positive value
10				No input	ERROR: Not enough arguments	ERROR: Not enough arguments
11	2	2	4	a + b = c	ERROR: Not a valid triangle	Isosceles
12	3	4	9	a + b < c	ERROR: Not a valid triangle	ERROR: Not a valid triangle

4 Conclusion

In this lab, we were introduced to black-box testing. Specifically, dirty testing, error guessing, and partition based testing. We learned that dirty testing and error guessing involves testing every possible input/action a user can perform on the software. It requires testers to be creative and look for unique errors by thinking outside of the box. With more experience in testing similar software, these types of testing may become more effective as the tests are only as good as the creativity of the tester. For partition based testing, we were able to cover as many possibilities as possible with as little test cases as possible, but were unable to test unique cases. In the end, one testing method doesn't seem to be better than the other and the tester should choose the appropriate method for the program they want to test.