SENG3320 Assignment 2:
Automated Test Data Generation

Group 10

Kyle Beattie C3303374
Ni Zeng C3238805
Brandon Allen C3279505
Austin Baxter C3356468

Table of Contents

1. Introduction

This report details the testing of a provided copy of the KWIC and Triangle programs. The usage of Fuzz Testing and Automated Testing Techniques were recorded below along with analysis on the effectiveness of these techniques. Provided is also a comparison between Fuzz testing, Mutation Testing and Symbolic Execution.

2. Question 1: Fuzz Testing

2.1. Test Tool Design

The Test Tool for Fuzz Testing the KWIC program is separated into three classes consisting of KWICTester, RandomData and ExceptionHandler. The KWICTester classes implements the main method which takes three command line inputs. These inputs in order describe the number of testing files created, max number of lines in each file and max number of ASCII characters in each line. These values are then utilised in the RandomData class which creates the required number of text files which contain a random number of random book titles that are limited by the arguments used when running the program. The text files are then saved in the InputTextFolder location. KWICTester will then run the KWIC program for each text file that had just been created, catching all exceptions that occur. These exceptions are then sent to the ExceptionHandler which utilises a LinkedHashSet so that only unique exceptions are saved. An exception is considered unique if everything after the first “at” isn’t already in a LinkedHashSet which automatically discards duplicates. Once all input texts are tested all exceptions and their input are listed in a text folder in the ExceptionTextFolder location.

2.2. Test Environment

The program was tested using java version "11.0.10". To run the program Command Prompt or PowerShell can be used. After changing the directory to inside the Q1 folder and compiling using “javac *.java” the program can be run by entering the command “java -XX:-OmitStackTraceInFastThrow KWICTester numberOfFiles numberOfLines numberOfCharacters” with the three required arguments being positive integers. Including -XX:-OmitStackTraceInFastThrow is required as java optimises exceptions that are frequent which removes the stack trace and only shows the exception that is caught.

2.3. Test Cases

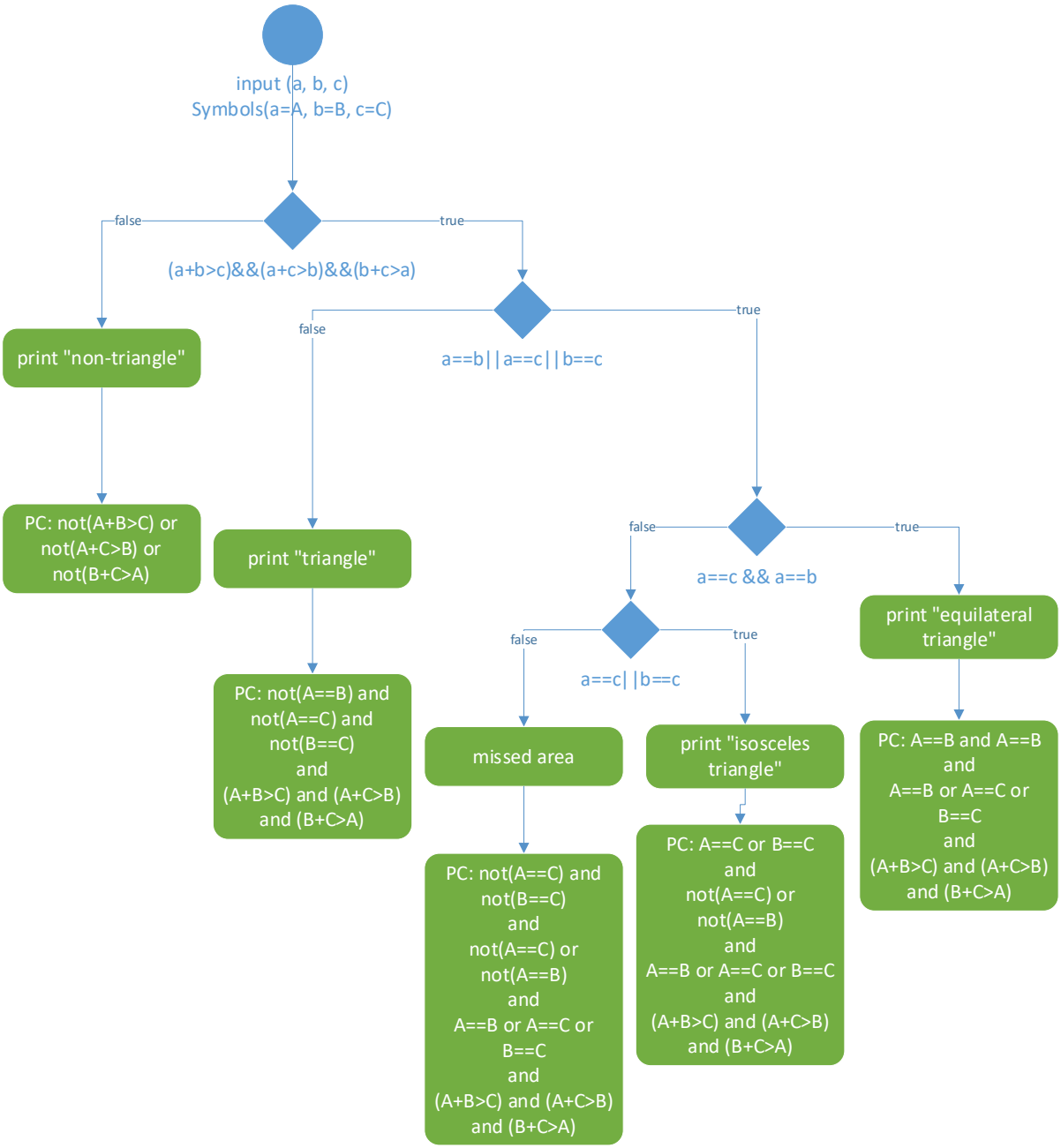
Each time the program is executed a new set of test inputs is generated that contain a random list of book names. This is due to the number of lines in each text test file being randomly selected from [1: numberOfLines] and the number of characters in each line also being randomly chosen from [1: numberOfCharacters]. The characters used for the book titles are also randomly selected to consist of the printable ASCII characters which range from character code [32:126]. Space characters are weighted to have a higher percentage of being selected as numbers between [20:31] are changed to 32 which represents a space. This is done to allow better demonstration of the KWIC program. Examples of test inputs can be found in Appendix A: Q1 Example Inputs.

2.4. Summary

The number of unique exceptions generated can differ each time the program is run. This is due to random data being generated to perform the fuzz testing. Generally increasing the number of test inputs, possible number of lines and characters results in more unique exceptions being found. Two exceptions are thrown which include ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException. Running the program multiple times with different command line argument inputs has shown that only one unique StringIndexOutOfBoundsException is caught however several unique ArrayIndexOutOfBoundsException are caught. These ArrayIndexOutOfBoundsExceptions differ by the number of times KWIC.quickSort(KWIC.java:778) is included in the stack trace of the exception. Using command line arguments of 5000 20 30 and running multiple times produced a maximum of 11 total unique exceptions which can be found in Result.txt in the Q1 folder. Due to the randomness of the inputs, it may be possible for more unique exceptions to be generated if the command line arguments are significantly increased although this will result in a longer execution time and larger storage requirements.

3. Question 2: Automated Testing Techniques

3.1. Control Flow Analysis


input (a, b, c)
Symbols(a=A, b=B, c=C)
(a+b>c)&&(a+c>b)&&(b+c>a)
false
true
print "non-triangle"
PC: not(A+B>C) or not(A+C>B) or not(B+C>A)
a==b||a==c||b==c
false
true
print "triangle"
PC: not(A==B) and not(A==C) and not(B==C) and (A+B>C) and (A+C>B) and (B+C>A)
a==c && a==b
false
true
print "equilateral triangle"
PC: A==B and A==B and A==B or A==C or B==C and (A+B>C) and (A+C>B) and (B+C>A)
a==c||b==c
false
true
missed area
PC: not(A==C) and not(B==C) and not(A==C) or not(A==B) and A==B or A==C or B==C and (A+B>C) and (A+C>B) and (B+C>A)
print "isosceles triangle"
PC: A==C or B==C and not(A==C) or not(A==B) and A==B or A==C or B==C and (A+B>C) and (A+C>B) and (B+C>A)

3.2. Symbolic Execution

KLEE was used in this section through a browser interface. The function was small enough that a main function could be placed below it. This function setup and ran KLEE providing an output.

KLEE Output

```
KLEE: output directory is "/tmp/code/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING: undefined reference to function: printf
KLEE: WARNING ONCE: calling external: printf(20726432) at /tmp/code/tmp/code/code.c:3 0
non-triangle.
non-triangle.
non-triangle.
triangle.
equilateral triangle .
isosceles triangle.
isosceles triangle.

KLEE: done: total instructions = 136
KLEE: done: completed paths = 8
KLEE: done: generated tests = 8
```

Interpretation of Results

Firstly there is a discrepancy between the number of lines output and the number of paths completed.

To give greater context a missing a print statement will be added to make up the last output. In the deepest nested if statement, no statement is presented if only a==b is satisfied. A print statement is placed here (else printf("missed area.\n");). This reveals the eighth test condition. This results in the output sequence:

```
KLEE: output directory is "/tmp/code/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING: undefined reference to function: printf
KLEE: WARNING ONCE: calling external: printf(17122720) at /tmp/code/tmp/code/code.c:3 0
non-triangle.
non-triangle.
non-triangle.
triangle.
equilateral triangle .
missed area.
isosceles triangle.
isosceles triangle.

KLEE: done: total instructions = 138
KLEE: done: completed paths = 8
KLEE: done: generated tests = 8
```

First the run fails each of the first decision's conditions. That is (a+b<=c) then (a+c<=b) then (b+c<=a). This results in the three "non-triangle." Outputs.

The rest of the run only concerns the equality of the different values. The following table may be used.

a == b	a == c	b == c
false	false	false
true	true	true
true	false	false
false	true	false
false	false	true

Note: the above conditions cannot be true two at a time.

The first-row results in the "triangle." Output, no two values are equal. The second-row results in the "equilateral triangle." Result, all values are equal. The third through fifth results are each condition being true one at a time. The first condition being true triggers the missed area statement that should result in an isosceles. The last two correctly result in isosceles outputs.

This method was useful due to its speed and showing a mismatch between number of outputs and paths used. Ideally all paths should produce outputs so when there is a discrepancy between the two a bug is obvious.

A downfall to this method is the inability to print the value of the symbolic value used for the execution of the program. For instance triangle(-1,-1,-1) would print "equilateral." If this is not an acceptable output. There is no determining these inputs.

Control Flow Analysis:

Decision Coverage:

a	b	c	Output	Valid
1	1	5	Non-triangle	True
2	3	4	Triangle	True
2	2	2	Equilateral	True
2	2	3	No output	False, should output isosceles.
2	3	2	Isosceles	True

Condition Coverage:

Conditions:

Dentoted by	Condition
C1	a+b>c
C2	a+c>b
C3	b+c>a
C4	a==b
C5	a==c
C6	b==c
C7	a==c (double nested)
C8	a==b (double nested)
C9	a==c (triple nested)
C10	b==c (triple nested)

a	b	c	Conditions executed and result	Output/Decisions	valid
1	1	5	C1=false, C2=true, C3=true	Non-triangle	True
1	5	1	C1=true, C2=false, C3=true	Non-triangle	True
5	1	1	C1=true, C2=true, C3=false	Non-triangle	True
2	3	4	C1=true, C2=true, C3=true, C4=false, C5=false, C6=false	Triangle	True
2	2	2	C1=true, C2=true, C3=true, C4=true, C5=true, C6=true, C7=true, C8=true	Equilateral	True
2	2	3	C1=true, C2=true, C3=true, C4=true, C5=false, C6=false, C7=false, C8=true, C9=false, C10=false	No output	False, should output isosceles.
2	3	2	C1=true, C2=true, C3=true, C4=false, C5=true, C6=false, C7=true, C8=false, C9=true, C10=false	Isosceles	True
3	2	2	C1=true, C2=true, C3=true, C4=false, C5=false, C6=true, C7=false, C8=false, C9=false, C10=true	Isosceles	True

Condition / Decision Coverage:

a	b	c	Conditions executed and result	Output/Decisions	valid
1	1	5	C1=false, C2=true, C3=true	Non-triangle	True
1	5	1	C1=true, C2=false, C3=true	Non-triangle	True
5	1	1	C1=true, C2=true, C3=false	Non-triangle	True
2	3	4	C1=true, C2=true, C3=true, C4=false, C5=false, C6=false	Triangle	True
2	2	2	C1=true, C2=true, C3=true, C4=true, C5=true, C6=true, C7=true, C8=true	Equilateral	True
2	2	3	C1=true, C2=true, C3=true, C4=true, C5=false, C6=false, C7=false, C8=true, C9=false, C10=false	No output	False, should output isosceles.
2	3	2	C1=true, C2=true, C3=true, C4=false, C5=true, C6=false, C7=true, C8=false, C9=true, C10=false	Isosceles	True
3	2	2	C1=true, C2=true, C3=true, C4=false, C5=false, C6=true, C7=false, C8=false, C9=false, C10=true	Isosceles	True

Multiple Condition Coverage:

a	b	c	Conditions executed and result	Output/Decisions	valid
1	1	5	C1=false, C2=true, C3=true	Non-triangle	True
1	5	1	C1=true, C2=false, C3=true	Non-triangle	True
5	1	1	C1=true, C2=true, C3=false	Non-triangle	True
2	3	4	C1=true, C2=true, C3=true, C4=false, C5=false, C6=false	Triangle	True
2	2	2	C1=true, C2=true, C3=true, C4=true, C5=true, C6=true, C7=true, C8=true	Equilateral	True
2	2	3	C1=true, C2=true, C3=true, C4=true, C5=false, C6=false, C7=false, C8=true, C9=false, C10=false	No output	False, should output isosceles.
2	3	2	C1=true, C2=true, C3=true, C4=false, C5=true, C6=false, C7=true, C8=false, C9=true, C10=false	Isosceles	True
3	2	2	C1=true, C2=true, C3=true, C4=false, C5=false, C6=true, C7=false, C8=false, C9=false, C10=true	Isosceles	True

Note: CC, C/DC and MCC are equal because conditions C1, C2, C3 and (C4 and C8), (C5,C7 and C9), (C6 andC10) can only occur one at a time (the conditions in brackets are the same conditions used in different decisions).

3.3. Fuzz Testing

The idea of Fuzz Testing on this question is to apply random integers for variables a, b, and c to examine the outcome of the triangle (int a, int b, int c).

Fuzz Testing structure:
>FuzzTesting (Q2 Fuzz Testing task folder)

FuzzInput_Output.txt (contain the input & output test case result)

FuzzTesting.c (Fuzz test case generator)

FuzzTesting.exe (Fuzz test case generator executor)

triangle. c (Given c program for Fuzz Testing)

When run the FuzzTesting.exe, the Fuzz generator will first ask for user input for the number of test cases that need to be generated. After the input, the generator will generate 3 random integer numbers (a,b,c) from the range 0 to 9 for each of the test cases. The generated integer number(a,b,c) will execute using triangle.c and record the input and output result in a text file Name 'Fuzzinput_Output.txt'.

In the text file "Fuzzinput_Output.txt", the text file will first list individual values a,b,c and results of each test case. The total number of test cases, total time spent fuzz testing, number of non-triangles, triangles, isosceles triangles and equilateral triangles and number of errors that occurred.

In terms of control-flow coverage covered by Fuzz Testing is very depends on the number of test cases to be generated. The more test cases to be generated, the more coverage will be covered.
During the Fuzz Testing on triangle.c , it discovered that there are some test cases with no output type of triangle , it is certainly a bug in trianle.c. the total number of errors has been recorded in the "Fuzzinput_Output.txt" File.
The main problem that causes this no out for a test case is that in triangle.c , there is one more condition which is not been added to the code.
the missing condition: (a == b)
the original code in triangle.c :

```
else if (a==c||b==c)
    printf("isosceles triangle.\n");
```

it should be :

else if (a ==c || b== c || a ==b) printf("isosceles triangle.\n");

In order to compare the test results of Fuzz testing and symbolic execution, the number of generated test cases should remain the same as the previous symbolic execution which is 8.

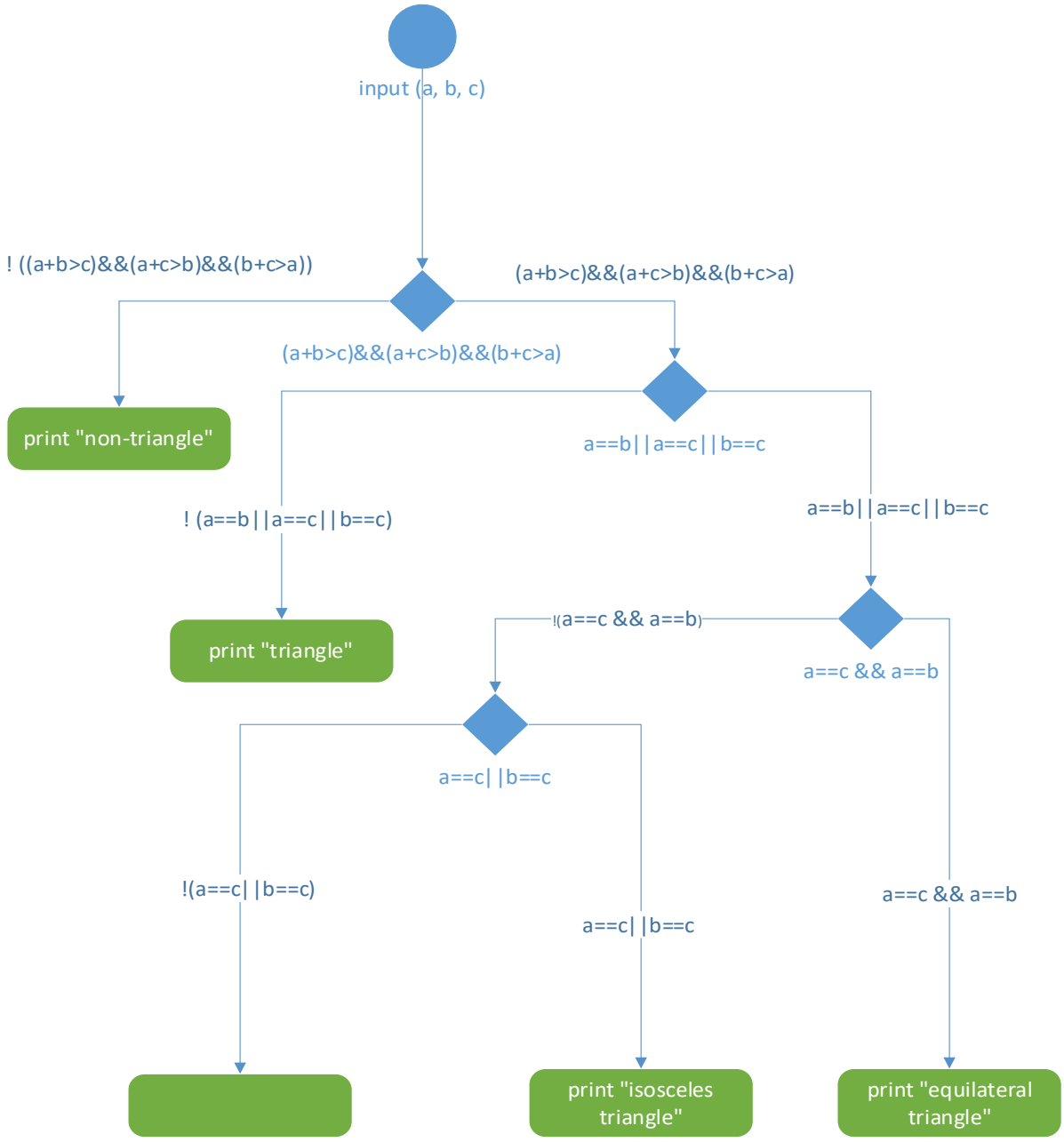
Q2 > FuzzTesting > FuzzInput_Output.txt

You, 18 seconds ago | 1 author (You)

```
7  6  7  isosceles triangle.
2  1  2  isosceles triangle.
6  3  7  triangle.
1  0  0  non-triangle.
2  9  9  isosceles triangle.
0  7  6  non-triangle.
7  4  5  triangle.
5  8  8  isosceles triangle.

Total Test case: 8
Total time spent: 0.015000 s
non_triangle: 2
triangle: 2
isosceles_triangle: 4
equilateral_triangle: 0
error (with no output triangle type): 0
========================================
```

Control Flow Diagram:



Test Cases (Sort by triangle type):

Test cases:	a	b	c	Output
1 2	1 0	0 7	0 6	Non-triangle
3 4	6 7	3 4	7 5	Triangle
5 6 7 8	7 2 2 5	6 1 9 8	7 2 9 8	Isosceles

Control flow analysis:

Decision Coverage (based on the 8 test cases):

Test cases	(a+b>c)&&(a+c>b)&&(b+c>a)	(a==b || a==c || b==c)	(a==c && a==b)	(a==c||b==c)	Decision
1	F	T	F	T	T
2	F	F	F	F	T
3	T	F	F	F	T
4	T	F	F	F	T
5	T	T	F	T	T
6	T	T	F	T	T
7	T	T	F	T	T
8	T	T	F	T	T

Based on the 8 test cases generated by the Fuzz testing program, it cannot achieve 100 % Decision Coverage at this point. (a==c && a==b) has not yet been covered. This only achieves 80% decision coverage.

Condition Coverage (based on the 8 test cases):

Conditions:

Test case	a+b>c	a+c>b	b+c>a	a==c	b==c	a==b
1	T	T	F	F	T	F
2	T	F	T	F	F	F
3	T	T	T	F	F	F
4	T	T	T	F	F	F
5	T	T	T	T	F	F
6	T	T	T	T	F	F
7	T	T	T	F	T	F
8	T	T	T	F	T	F

Based on the 8 test cases generated by the Fuzz testing program, it cannot achieve 100 %

Conditions Coverage at this point. Condition like “a == b” has not been covered at this point. the automatic generate Fuzz testing program can only achieve 83.33%.

Multiple Condition Coverage:

a+b>c	a+c>b	b+c>a	Test case number		a==c	a==b	Test case number		b==c	Test case number
T	T	T	3,4,5,6,7,8		T	T	null		T	7,8
T	T	F	1		T	F	5,6		F	1,2,3,4,5,6
T	F	F	null		F	T	null			
T	F	T	2		F	F	1,2,3,4,7,8			
F	T	T	null							
F	T	F	null							
F	F	T	null							
F	F	F	null							

Multiple condition Coverage 50%.

Based on the generated test case results:
Due to the previous symbolic execution-only generating 8 test cases, to compare with the Fuzz testing on triangle.c. The total number of test cases must remain the same.

8 test cases for Fuzz testing are not enough for Fuzz testing to achieve full control flow coverage.

Control-flow coverage achieved:	
Fuzz Testing: Because of the low number of test cases randomly generated by the Fuzz Testing program. For Fuzz testing program can only achieve 80% decision coverage,83.3% condition coverage, 50% Multiple condition coverage.	Symbolic testing: Because of the relationship between conditions achieving D, C, D/C and MC coverage are all covered 100% by the symbolic execution.
Time spent:	
Fuzz Testing: 0.015 s	Symbolic testing: using the browser version of KLEE could have had the effect of increasing the time taken to execute. This would related to ping and queuing the job approximately 6 seconds

Summery, in order to get close to a 100% control flow coverage of the Fuzz Testing program on triangle.c , it required the user to enter a larger number of test cases that need to be generated by the Fuzz Testing program. 100 test cases might be able to achieve 100% Fuzz

Testing depending on the random integer generated by the program.

3.4. Mutation Testing

For the mutation testing, 11 triangle variants were tested, with one representing the original triangle.c and 10 mutants that were configured using Arithmetic Operator Replacement, Relational Operator Replacement or Conditional Operator Replacement.

```
#include <stdio.h>


void triangle (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 1, the base unmutated variant. Triangle code were all edited with Notepad++ v8.3.3

```
#include <stdio.h>


void triangle2 (int a, int b, int c){
if ((a-b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
```

Triangle 2, with a+b>c becoming a-b>c

```c
#include <stdio.h>


void triangle3 (int a, int b, int c){
if ((a+b>c)&&(a-c>b)&&(b+c>a)) {
    if (a==b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 3, with a+c>b becoming a-c>b

```c
#include <stdio.h>


void triangle4 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b-c>a)) {
    if (a==b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 4, with b+c>a becoming b-c>a

```c
#include <stdio.h>


void triangle5 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a>=b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 5, with a==b becoming a>=b

```c
#include <stdio.h>


void triangle6 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b || a>=c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 6, with a==c becoming a>=c

```c
#include <stdio.h>


void triangle7 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b || a==c || b>=c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 7, with b==c becoming b>=c

```c
#include <stdio.h>


void triangle8 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b && a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 8, with a==b ||a==c becoming a==b && a==c

```c
#include <stdio.h>


void triangle9 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b || a!=c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 9, with a==c becoming a!=c

```c
#include <stdio.h>


void triangle10 (int a, int b, int c){
if ((a+b>c)&&(a+c>b)&&(b+c>a)) {
    if (a==b && a==c && b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 10, with a==b || a==c || b==c becoming a==b && a==c && b==c

```c
#include <stdio.h>


void triangle11 (int a, int b, int c){
if ((a+b>c)||(a+c>b)&&(b+c>a)) {
    if (a==b || a==c || b==c) {
        if (a==c && a==b)
            printf("equilateral triangle.\n");
        else if (a==c||b==c)
            printf("isosceles triangle.\n");
    }
    else
        printf("triangle.\n");
}
else
    printf("non-triangle.\n");

return;
}
```

Triangle 11, with (a+b>c)&&(a+c>b) becoming (a+b>c)||(a+c>b)

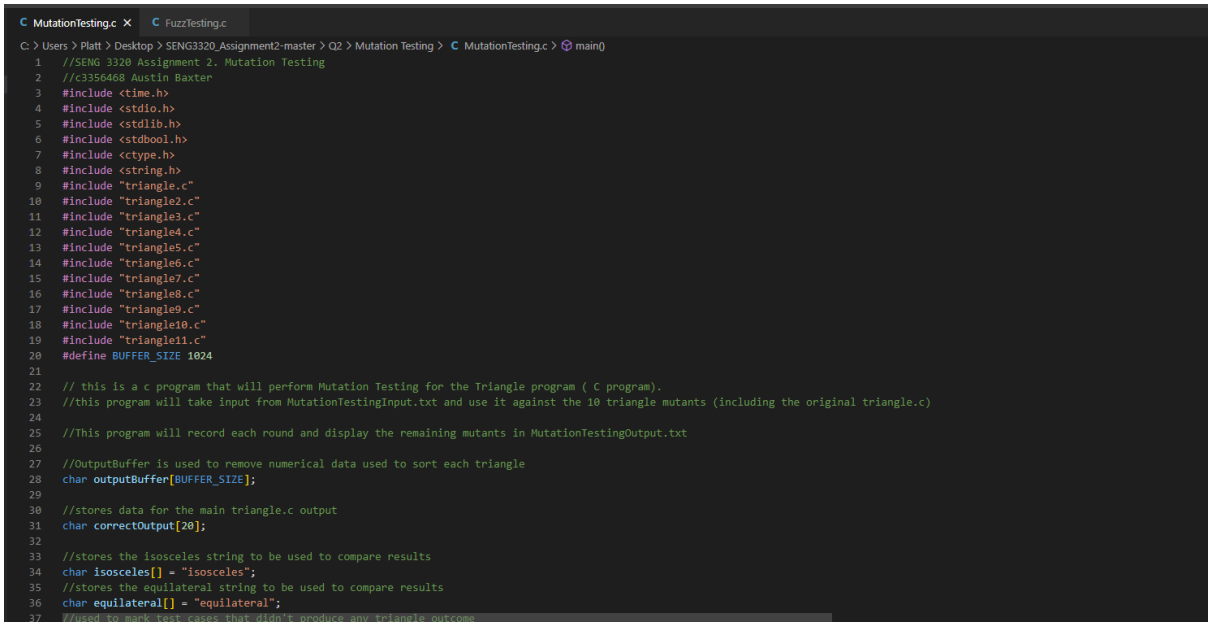
These mutants were tested against using the test cases below. The initial test cases were taken from Brandon Allen's section, who developed the KLEE solution in this assignment, however the test cases were not enough to eliminate all mutants so Ni Zeng's fuzz testing and some specialised test cases of my own design were used to supplement this list.

```
5 1 2
5 1 1
1 2 5
1 1 5
1 5 2
1 5 1
2 3 4
2 2 3
2 3 2
3 2 2
2 2 2
5 8 8
7 4 5
0 7 6
2 9 9
1 3 2
2 4 3
```

Each column represented one of the sides of the triangle, a,b and c respectively

Each test case had its own round, and mutants were eliminated if they provided differing results to Triangle 1, which was the unmodified variant. The testing program was coded in c

and the functions can be found in MutationTesting.c.



```c
//SENG 3320 Assignment 2. Mutation Testing
//c3356468 Austin Baxter
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>
#include "triangle.c"
#include "triangle2.c"
#include "triangle3.c"
#include "triangle4.c"
#include "triangle5.c"
#include "triangle6.c"
#include "triangle7.c"
#include "triangle8.c"
#include "triangle9.c"
#include "triangle10.c"
#include "triangle11.c"
#define BUFFER_SIZE 1024

// this is a c program that will perform Mutation Testing for the Triangle program ( C program).
//this program will take input from MutationTestingInput.txt and use it against the 10 triangle mutants (including the original triangle.c)

//This program will record each round and display the remaining mutants in MutationTestingOutput.txt

//OutputBuffer is used to remove numerical data used to sort each triangle
char outputBuffer[BUFFER_SIZE];

//stores data for the main triangle.c output
char correctOutput[20];

//stores the isosceles string to be used to compare results
char isosceles[] = "isosceles";
//stores the equilateral string to be used to compare results
char equilateral[] = "equilateral";
//used to mark test cases that didn't produce any triangle outcome
```

This program was designed using Visual Studio Code 1.67.2, code compilation was done using native visual studio tools.

```
Round: 0
Test Data: 5 1 2
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:triangle.
```

The initial round used data that could not form a triangle; however Triangle 11 considered the data a valid triangle and was therefore eliminated. This test case however didn't detect any anomalies for 9/10 mutants, and therefore was not an effective test case

```
Round: 1
Test Data: 5 1 1
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:dead in round 0
```

Round 1 was also not a triangle, and no mutants were caught with this test case

```
Round: 2
Test Data: 1 2 5
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:dead in round 0
```

Round 2 followed Round 1, with no mutants being eliminated

```
Round: 3
Test Data: 1 1 5
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:dead in round 0
```

Round 3 followed Round 2, no mutants were also eliminated

```
Round: 4
Test Data: 1 5 2
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:dead in round 0
```

```
Round: 5
Test Data: 1 5 1
Triangle 1: non-triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:non-triangle.
 Triangle 6:non-triangle.
 Triangle 7:non-triangle.
 Triangle 8:non-triangle.
 Triangle 9:non-triangle.
 Triangle 10:non-triangle.
 Triangle 11:dead in round 0
```

Round 4 and Round 5 both could not eliminate any mutants

```
Round: 6
Test Data: 2 3 4
Triangle 1: triangle.
Triangle 2:non-triangle.
 Triangle 3:non-triangle.
 Triangle 4:non-triangle.
 Triangle 5:triangle.
 Triangle 6:triangle.
 Triangle 7:triangle.
 Triangle 8:triangle.
 Triangle 9:failed
 Triangle 10:triangle.
 Triangle 11:dead in round 0
```

Round 6 was a major success in the removal of the mutant triangles. Whilst the original class did classify the test data as forming a triangle, triangles 2,3 and 4 both did not consider it a triangle, and Triangle 9 failed to provide a conclusion on the matter. This test case eliminated 4/9 remaining mutants.

```
Round: 7
Test Data: 2 2 3
Triangle 1: failed
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:failed
 Triangle 6:failed
 Triangle 7:failed
 Triangle 8:triangle.
 Triangle 9:dead in round 6
 Triangle 10:triangle.
 Triangle 11:dead in round 0
```

Round 7 was valuable in not only showing that the base triangle file has a bug, where an isosceles triangle has not been noticed, but also showed that Triangle 8 and 10 were not showing the same outcomes 2/5 remaining mutants were discovered and eliminated, making this test case very valuable.

```
Round: 8
Test Data: 2 3 2
Triangle 1: isosceles
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:isosceles
 Triangle 6:isosceles
 Triangle 7:isosceles
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

```
Round: 9
Test Data: 3 2 2
Triangle 1: isosceles
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:isosceles
 Triangle 6:isosceles
 Triangle 7:isosceles
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Round 8 and 9 showed an isosceles triangle that was caught by the program, unfortunately all mutants showed identical results, with no eliminations these rounds

```
Round: 10
Test Data: 2 2 2
Triangle 1: equilateral
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:equilateral
 Triangle 6:equilateral
 Triangle 7:equilateral
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

```
Round: 11
Test Data: 5 8 8
Triangle 1: isosceles
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:isosceles
 Triangle 6:isosceles
 Triangle 7:isosceles
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Rounds 10 and 11 also failed to detect any mutants. Round 11 and beyond were part of the Fuzz testing results

```
Round: 12
Test Data: 7 4 5
Triangle 1: triangle.
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:failed
 Triangle 6:failed
 Triangle 7:triangle.
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Round 12 managed to detect 2/3 mutants, with Triangle 5 and 6 failing to give a response.

```
Round: 13
Test Data: 0 7 6
Triangle 1: non-triangle.
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:dead in round 12
 Triangle 6:dead in round 12
 Triangle 7:non-triangle.
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

```
Round: 14
Test Data: 2 9 9
Triangle 1: isosceles
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:dead in round 12
 Triangle 6:dead in round 12
 Triangle 7:isosceles
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

```
Round: 15
Test Data: 1 3 2
Triangle 1: non-triangle.
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:dead in round 12
 Triangle 6:dead in round 12
 Triangle 7:non-triangle.
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Rounds 13,14 and 15 were unsuccessful in eliminating Triangle 7. This mutant's mutation is very minor, resulting in it clearing most rounds without any issue.

```
Round: 16
Test Data: 2 4 3
Triangle 1: triangle.
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:dead in round 12
 Triangle 6:dead in round 12
 Triangle 7:failed
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Round 16 used test data specifically designed to eliminate Triangle 7 and was therefore successful.

```
Final Results
Triangle 1: triangle.
Triangle 2:dead in round 6
 Triangle 3:dead in round 6
 Triangle 4:dead in round 6
 Triangle 5:dead in round 12
 Triangle 6:dead in round 12
 Triangle 7:dead in round 16
 Triangle 8:dead in round 7
 Triangle 9:dead in round 6
 Triangle 10:dead in round 7
 Triangle 11:dead in round 0
```

Results show that Round 6 was the most effective test case in identifying Triangle variants, however Round 7 was effective in showing a structural bug within the original triangle program.

3.5. Comparison

Symbolic execution generated 8 tests for 8 paths and presented with 7 outputs. Three paths for breaking the triangle inequality in three different ways. That is one value was too high to form a triangle with the two other values. One for failing all equality of values conditions, being a plain triangle. One for passing all equality of values conditions. Two for the most nest “else if” statement correctly defining isosceles. And one path that in the original method made no output. An anomaly is present in the results.

Fuzz testing stored the print data in an output buffer. Reading this output buffer allowed for reading the methods outputs. If the function allows for is and is not a triangle, then no output should be defined as an invalid output. Logic in the testing functions tests for this. Upon finding an instance of an empty output buffer and error has been found. This occurs with approximately 4% (44/1000) of inputs. An anomaly is possibly found in the results with a probability dependent upon the number of test cases run.

Mutation Testing used ten similar but different versions of the triangle function. Different operators are used in each mutant version of the function. Test values are run through all the mutants and the mutants with outputs not matching the original function are determined to be killed. Killing all the mutants determines that all the logic and operations of the original function are necessary. This determines redundancy in the original triangle function. This method does not find any errors in its run as it compares the mutants to the original function being tested as though it has already been determined to be completely valid.

4. Appendix A: Q1 Example Inputs

4.1. Example 1: No Exception

Input Text:
nw- Kf
:# \ief_*Q] T
l0(f{q^:TqlDbK: , [
+u t> AF;'V&=i%x!I

Output:
+u t> AF;'V&=i%x!I
, [l0(f{q^:TqlDbK:
:# \ief_*Q] T
AF;'V&=i%x!I +u t>
Kf nw-
T :# \ief_*Q]
[l0(f{q^:TqlDbK: ,
\ief_*Q] T :#
] T :# \ief_*Q
l0(f{q^:TqlDbK: , [
nw- Kf
t> AF;'V&=i%x!I +u

4.2. Example 2: ArrayIndexOutOfBoundsException

Input Text:
v,1" \Rv a c{7E3%
? 2[ml[j({%_WFDJ \V
6a q K &xoTs(r
!M/.'9c

Exception:
java.lang.ArrayIndexOutOfBoundsException: Index 12 out of bounds for length 12 at KWIC.partition(KWIC.java:790) at KWIC.quickSort(KWIC.java:774) at KWIC.newAlphabetizing(KWIC.java:759) at KWIC.main(KWIC.java:858) at KWICTester.main(KWICTester.java:29)

4.3. Example 3: StringIndexOutOfBoundsException

Input Text:
AtVbW
9
;& ZG?
  d4hN] VC)
m{2"Dm>5

Output:
;
AtVbW
VC) d4hN]
ZG? ;&
d4hN]

Exception:
java.lang.StringIndexOutOfBoundsException: offset 36, count 9, length 44 at java.base/java.lang.String.checkBoundsOffCount(String.java:3304) at java.base/java.lang.String.rangeCheck(String.java:280) at java.base/java.lang.String.<init>(String.java:276) at java.base/java.lang.String.valueOf(String.java:2989) at KWIC.newOutPut(KWIC.java:842) at KWIC.main(KWIC.java:860) at KWICTester.main(KWICTester.java:29)

5. Group Member Contributions

Task	Member
Section 1- Fuzz testing	Kyle Beattie
Section 2- Fuzz Testing	Ni Zeng
Section 2-Symbolic Execution	Brandon Allen
Mutation Testing	Austin Baxter
Report	All