COSC 3P95 Assignment 1

Brett Terpstra
bt19ex@brocku.ca - 692021

October 12, 2023

Contents

1 Question 1 1
1.1 Soundness vs Completeness 1
1.2 Positive = Find bug 2
1.3 Positive = Not find bug 2
2 Question 2 3
2.1 A 3
2.2 B 5
3 Question 3 7
3.1 A 7
3.2 B 8
4 Question 4 8
4.1 A 8
4.2 B 11
5 Question 5 17
5.1 A 17
5.2 B 18
6 Question 6 22

1 Question 1

Explain the difference between ”sound” and ”complete” analysis in software analysis. Then, define what true positive, true negative, false positive, and false negative mean. How would these terms change if the goal of the analysis changes, particularly when ”positive” means finding a bug, and then when ”positive” means not finding a bug.

1.1 Soundness vs Completeness

Soundness

Soundness is the ability for the analysis to not report false positives. So having a high soundness means that you can be sure that the bugs that were found are actually bugs in the program.

Completeness

Completeness refers to the ability to find all the bugs in the software by an analysis. So if you can be sure that your analysis is complete, you can be sure that if all bugs are fixed there will be no more bugs. (unless fixing it adds more bugs.) Of course in pratice it's nearly impossible for any tool to be fully complete.

1.2 Positive = Find bug

True Positive

When the program reports a bug and the bug actually exists

True Negative

When a program doesn't report a bug and there is no bug to be found.

False Positive

When a bug is said to be found but none actually exists.

False Negative

When a bug isn't found but does actually exist. That is to say, the program fails to report the bug.

1.3 Positive = Not find bug

You literally invert the logic. It's pretty basic stuff here? Is this subpart just busy work? Take the logical not of each sentence and that's the answer. This results in confusing statements and shouldn't be used in practice.

True Positive

When the program does not report a bug, there is not a bug present.

True Negative

When a program reports a bug, there is a bug to be found.

False Positive

When a bug is not found but one actually exists.

False Negative

When a bug is found, but does doesn't actually exist.

2 Question 2

2.1 A

Using your preferred programming language, implement a random test case generator for a sorting algorithm program that sorts integers in ascending order. The test case generator should be designed to produce arrays of integers with random lengths, and values for each sorting method.

Source

Please find the source code in the ZIP attachment for the submission of this assignment. Compiling instructions are found within the source file. Any C++20 compiler with support for `.contains` and `std::find\_if()` will work.

Explanation

Since we are only testing random int32s of random sizes I took a very simplistic approach. For our convenience there are two global config variables you can change:

- `DEFAULT_TEST_COUNT`: Number of tests to run, defaults 10, and can be provided at runtime via the optional command line argument:
  `./main.run [TEST\_COUNT]`
- `MAX_RAM_USAGE`: The Max number of bytes a vector used in the test can consume, this is a rough estimate used to put an upper bound on the number of integers in the test. Each test invocation vector is independent and memory is cleared between cycles. This was not designed with buffer reuse or any amount of efficiency so keep that in mind when setting this value too high. The default of 1MiB is fine for this assignment (fails about 15% of the time).

To cover all possible test cases I generate values in the range `[INTEGER_MIN, INTEGER_MAX]` which populate a vector of size `[0, MAX_SIZE]`

```cpp
std::vector<std::int32_t> generateRandomData() {
  // setup random numbers
  static std::random_device dev;
  static std::mt19937_64 engine(dev());
  // we will generate numbers from integer min to integer max to cover the full possible test cases
  std::uniform_int_distribution dist(std::numeric_limits<std::int32_t>::min(), std::numeric_limits<std::int32_t>::max());
  // put a upper limit on the size of the array
  std::uniform_int_distribution dist_size(0ul, maxSize);
  // pick a random size
  size_t size = dist_size(engine);
  // populate the array
  std::vector<std::int32_t> ret;
  for (size_t i = 0; i < size; i++)
    ret.push_back(dist(engine));
  return ret;
}
```

Algorithm 1: Test case generator code

To make sure there are demonstrable bugs, the sorting code intentionally creates errors within the sorted array. To actually sort the values I used the C++ standard algorithm `std::sort` which won't fail. I did this because the standard sort is *fast* and creating a sorting algorithm which fails some of the time but not others is easier said than done.

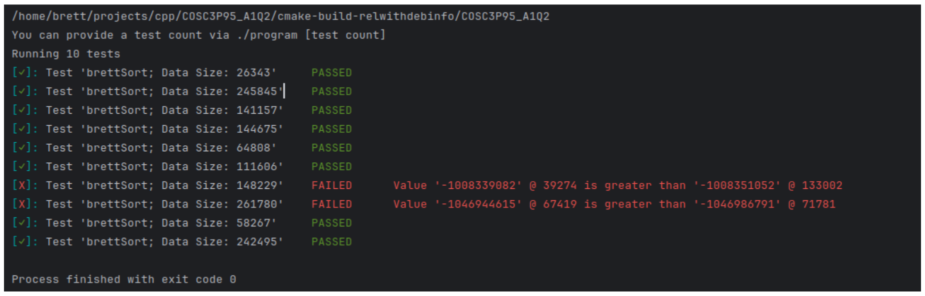
```cpp
[...]
std::uniform_int_distribution<int> dist(0, (int) v.size() - 1);
std::uniform_real_distribution realDist(0.0, 1.0);
int p1 = dist(engine);
int p2 = dist(engine);
if (!v.empty() && realDist(engine) > 0.85)
  v[p1] = v[p2];
[...]
```

Algorithm 2: Sort algorithm fail generator

This method of creating a failure also has the benefit of sometimes not failing, so it could execute, but the array may still be valid. It also shows that out of order pairs which are not neighbours will cause a failure condition that is detected. It should be noted that the validation will only report the first instance of an out of order number. This was done purely for performance. The report is generated as a tuple `{i, j}` where `i` is the failed number index and `j` is the first value index which causes the failure.

Sample Output

I took the liberty of automatically formatting the output because I like when things look pretty :3



```
/home/brett/projects/cpp/COSC3P95_A1Q2/cmake-build-relwithdebinfo/COSC3P95_A1Q2
You can provide a test count via ./program [test count]
Running 10 tests
[✓]: Test 'brettSort; Data Size: 26343'     PASSED
[✓]: Test 'brettSort; Data Size: 245845'    PASSED
[✓]: Test 'brettSort; Data Size: 141157'    PASSED
[✓]: Test 'brettSort; Data Size: 144675'    PASSED
[✓]: Test 'brettSort; Data Size: 64808'     PASSED
[✓]: Test 'brettSort; Data Size: 111606'    PASSED
[X]: Test 'brettSort; Data Size: 148229'    FAILED      Value '-1008339082' @ 39274 is greater than '-1008351052' @ 133002
[X]: Test 'brettSort; Data Size: 261780'    FAILED      Value '-1046944615' @ 67419 is greater than '-1046986791' @ 71781
[✓]: Test 'brettSort; Data Size: 58267'     PASSED
[✓]: Test 'brettSort; Data Size: 242495'    PASSED

Process finished with exit code 0
```

Figure 1: Sample output of the random test generator.

2.2 B

Provide a context-free grammar to generate all the possible test-cases.

---

(1) <Array> → ”{” <Array_Expression> ”}” | ”{}”

(2) <Array_Expression> → <Integral_t> | <Integral_t> ”,” <Array_Expression>

(3) <Integral_t> → <Value> | -<Value>

(4) <Value> → <Integer> | <Value><Integer>

(5) <Integer> → ”0” | ”1” | ”2” | ... | ”9”

(2.B) Explanation

I tried to be as explicit as possible based on my understanding of formal context free grammars (very limited, mostly with regard to simple compilers). I'm sure it could be simplified down, but the derivation of the grammar should generate a valid C++ initializer list.
My reasoning on each:

(1) Construct the actual array object itself. Can have any number of values in it or be an empty array.

(2) The contents of an array can either be a single integral type (termination / base case) or an integral type with more array content after it, which recursively expands to any number of integers.

(3) Needed this to make sure the negative sign is placed first and only once. Alternative would be: <Value> → `-`<Value> | <Integer> | <Value><Integer>, but that could (in theory) result in `-------1` which is why I included the integral_t step.

(4) Integers can be one of 10 characters so we recursively concat any number of <Integer> together to create our value. The examples in the slides don't do this, but it would not make sense without this step.

(5) The base characters for creating integer values.

3 Question 3

3.1 A

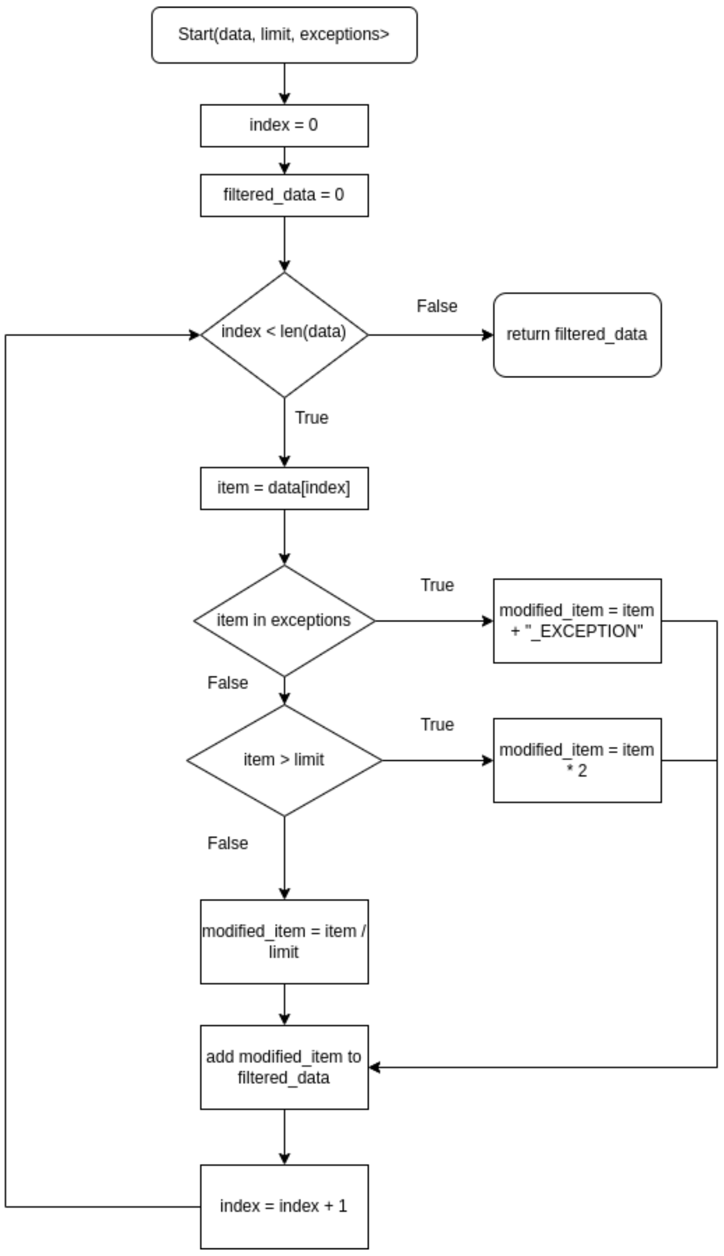


Figure 2: Direct translation of the code, logic errors and all.

3.2 B

Explain and provide detailed steps for “random testing” the above code. No need to run any code, just present the coding strategy or describe your testing method in detail.

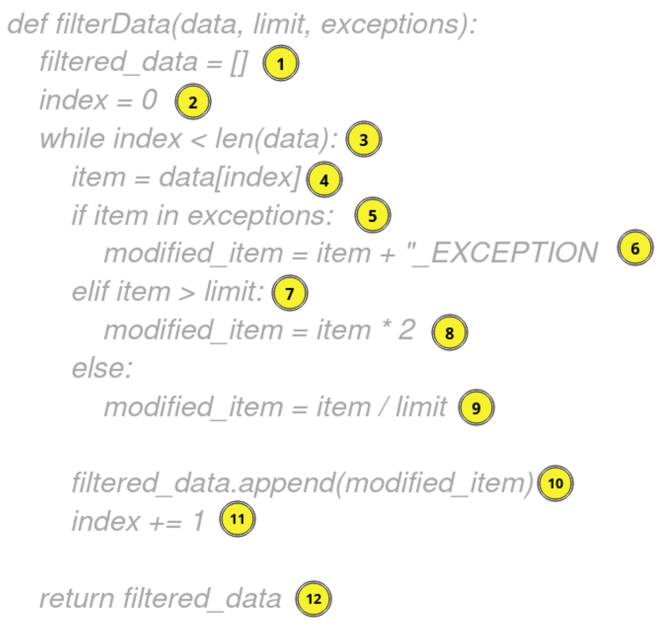
I don’t like the lack of static types in python. So, to remedy this I will assume items are numbers which can be trivially concatenated with strings. First start by generating a list of N random integers where N is also selected randomly from the whole numbers set. Second, pick a random subset of those integers, these will be your exception types. Randomly select an integer to act as the limit. Run the function and check the output to make sure it is what is expected. Repeat this until the number of tests you want have completed.

4 Question 4

4.1 A

Develop 4 distinct test cases to test the above code, with code coverage ranging from 30to 100%. For each test-case calculate and mention its code coverage.

Using statement coverage since it was not specified which type of code coverage to use.



```
def filterData(data, limit, exceptions):
  filtered_data = []  1
  index = 0  2
  while index < len(data):  3
    item = data[index]  4
    if item in exceptions:  5
      modified_item = item + "_EXCEPTION  6
    elif item > limit:  7
      modified_item = item * 2  8
    else:
      modified_item = item / limit  9

    filtered_data.append(modified_item)  10
    index += 1  11

  return filtered_data  12
```

Figure 3: 12 Statements in the function

Test 1

To get 30% code coverage we can input with empty data. This will run statements, 1,2,3, and 12 for a total of 33.3% coverage. The values of limit and exceptions doesn't matter in this case.

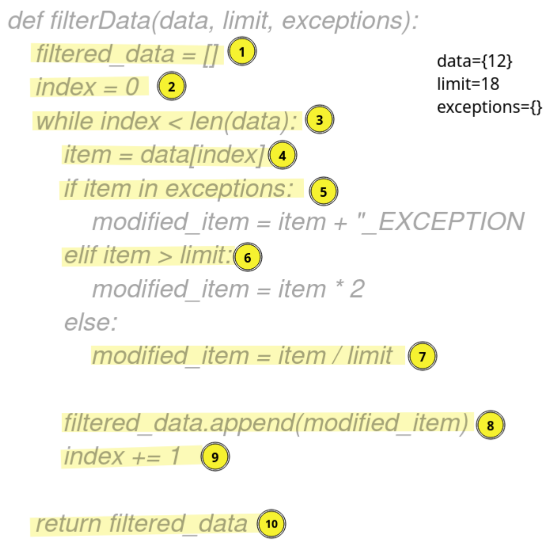
Test 2

```
data = {55, 12, 66}
limit = 18
exceptions = {55}
```

As statements 1,2,3 and 12 all run no matter what we add 4 to the base coverage. Since we have items in data statement 4 and 5 get executed. Since 55 is in exceptions statement 6 also gets executed. value 12 is less than limit so statement 7 and 9 gets executed. Since 66 is above the limit so does statement 8 also runs. 10 and 11 will run for all 3 values resulting in 12/12 coverage or 100%.

Test 3

```
data = {12}
limit = 18
exceptions = {}
```



```
def filterData(data, limit, exceptions):
  filtered_data = [] 1
  index = 0 2
  while index < len(data): 3
    item = data[index] 4
    if item in exceptions: 5
      modified_item = item + "_EXCEPTION
    elif item > limit: 6
      modified_item = item * 2
    else:
      modified_item = item / limit 7

    filtered_data.append(modified_item) 8
    index += 1 9

  return filtered_data 10
```

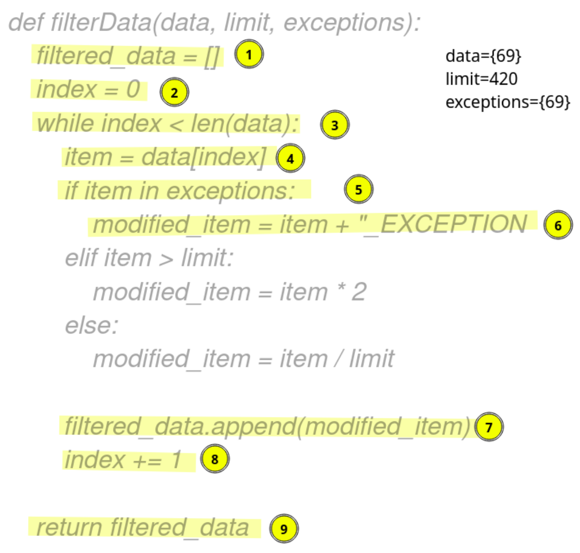
```
data={12}
limit=18
exceptions={}
```

Figure 4: Test 3 Graphical Results

Thus this test produces 10/12 or 83% code coverage.

Test 4

```
data = {69}
limit = 420
exceptions = {69}
```



```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit

        filtered_data.append(modified_item)
        index += 1

    return filtered_data
```

```
data={69}
limit=420
exceptions={69}
```

Figure 5: Test 4 Graphical Results

This test produces 66% code coverage.

4.2 B

Generate 6 modified (mutated) versions of the above code.

Method call mutations not possible due to the lack of external function calls by the given function. (existing calls not possible to modify due to lack of viable alternatives)

```python
def filterData1(data, limit, exceptions):
  filtered_data = []
  index = 0
  while index < len(data):
    item = data[index]
    if item in exceptions:
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item + 2  # Arithmetic mutation
    else:
      modified_item = item / limit
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 3: Mutation 1

```
def filterData2(data, limit, exceptions):
  filtered_data = []
  index = 0
  while index < len(data):
    item = data[index]
    if item not in exceptions:      # Boolean mutation
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item * 2
    else:
      modified_item = item / limit
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 4: Mutation 2

```
def filterData3(data, limit, exceptions):
  filtered_data = []
  index = 1       # statement mutation
  while index < len(data):
    item = data[index]
    if item in exceptions:
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item * 2
    else:
      modified_item = item / limit
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 5: Mutation 3

```
def filterData4(data, limit, exceptions):
  filtered_data = []
  index = 0
  while index < len(data):
    item = data[index]
    if item in data:        # Variable mutation
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item * 2
    else:
      modified_item = item / limit
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 6: Mutation 4

```
def filterData5(data, limit, exceptions):
  filtered_data = []
  index = 0
  while index < len(data):
    item = data[index]
    if item in exceptions:
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item * 2
    else:
      modified_item = item / 2        # Statement mutation
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 7: Mutation 5

```
def filterData6(data, limit, exceptions):
  filtered_data = []
  index = 0
  while index < len(data):        # Boolean mutation
    item = data[index]
    if item in exceptions:
      modified_item = str(item) + "_EXCEPTION"
    elif item > limit:
      modified_item = item * 2
    else:
      modified_item = item / limit
    filtered_data.append(modified_item)
    index += 1
  return filtered_data
```

Algorithm 8: Mutation 6

C

Assess the effectiveness of the test cases from part A by using mutation analysis in conjunction with the mutated codes from part B. Rank the test-cases and explain your answer.

I made a script to all **24** cases for me :/ Ranked results are at the bottom.

```
Running with test cases:
  1: data = [], limit = 0, exceptions=[]
  2: data = [55, 12, 66], limit = 18, exceptions=[55]
  3: data = [12], limit = 18, exceptions=[]
  4: data = [69], limit = 420, exceptions=[69]



Running tests for mutation 1
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned ['55_EXCEPTION', 0.6666666666666666, 68] on test
     case 2
TEST CASE 2 FAILED
Unmodified function returned [0.6666666666666666] while the
    modified function returned [0.6666666666666666] on test
    case 3
TEST CASE 3 PASSED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned ['69_EXCEPTION'] on test case
    4
TEST CASE 4 PASSED


Running tests for mutation 2
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned [110, '12_EXCEPTION', '66_EXCEPTION'] on test
    case 2
TEST CASE 2 FAILED
Unmodified function returned [0.6666666666666666] while the
    modified function returned ['12_EXCEPTION'] on test case
    3
TEST CASE 3 FAILED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned [0.16428571428571428] on test
    case 4
TEST CASE 4 FAILED


Running tests for mutation 3
```

```
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned [0.6666666666666666, 132] on test case 2
TEST CASE 2 FAILED
Unmodified function returned [0.6666666666666666] while the
    modified function returned [] on test case 3
TEST CASE 3 FAILED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned [] on test case 4
TEST CASE 4 FAILED


Running tests for mutation 4
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned ['55_EXCEPTION', '12_EXCEPTION', '66_EXCEPTION']
     on test case 2
TEST CASE 2 FAILED
Unmodified function returned [0.6666666666666666] while the
    modified function returned ['12_EXCEPTION'] on test case
    3
TEST CASE 3 FAILED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned ['69_EXCEPTION'] on test case
    4
TEST CASE 4 PASSED


Running tests for mutation 5
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned ['55_EXCEPTION', 6.0, 132] on test case 2
TEST CASE 2 FAILED
Unmodified function returned [0.6666666666666666] while the
    modified function returned [6.0] on test case 3
TEST CASE 3 FAILED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned ['69_EXCEPTION'] on test case
    4
TEST CASE 4 PASSED


Running tests for mutation 6
----------------------------
Unmodified function returned [] while the modified function
    returned [] on test case 1
TEST CASE 1 PASSED
Unmodified function returned ['55_EXCEPTION',
    0.6666666666666666, 132] while the modified function
    returned ['55_EXCEPTION', 0.6666666666666666, 132] on
    test case 2
TEST CASE 2 PASSED
Unmodified function returned [0.6666666666666666] while the
    modified function returned [0.6666666666666666] on test
    case 3
TEST CASE 3 PASSED
Unmodified function returned ['69_EXCEPTION'] while the
    modified function returned ['69_EXCEPTION'] on test case
    4
TEST CASE 4 PASSED


Ranked Results:
---------------
  1. mutation 2 with 3/4 failed tests
  2. mutation 3 with 3/4 failed tests
  3. mutation 4 with 2/4 failed tests
  4. mutation 5 with 2/4 failed tests
  5. mutation 1 with 1/4 failed tests
  6. mutation 6 with 0/4 failed tests
```

results.txt

The first test case always passes because it's a trivial example. Most mutations will not cause it to fail. Other than that the results stand for themselves.

D

Discuss how you would use path, branch, and statement static analysis to evaluate/analyse the above code.

Branch Static Analysis Easy sweep and mark the code for branches, identifying the test cases in a type of tree (the tree would include code which modifies any variables used in the conditionals otherwise we don't care!) then preorder traverse the tree resulting in every test case being covered, in the correct order of execution. Print a warning to the user if any syntactic or common logical errors are found (since this is static we are not actually executing the code but looking at it for errors). Note: the left subtree would but the true case, the right subtree would be the false case, assuming `else` is present.

Path Static Analysis I put this second because you could do the exact same thing as branch analysis but instead of doing preorder, which covers all code paths, you pick a path you want to follow and execute through it. You would also want to record every statement (in order of function call from init/main) instead of just what is relevant to the conditions. As with the branch analysis we don't actually execute the path but rather look at the code for errors. (Clang tidy ftw)

Statement Static Analysis Parse through the file line by line looking at all the statements in the code printing errors as we find them. I'm not sure if you are looking for how we would actually detect that, other than syntax errors that question has a complicated answer. Even finding syntax errors can be hard, although you would only have to parse the AST and if you find an unexpected token it's pretty clear the programmer made an error.

5 Question 5

The code snippet below aims to switch uppercase characters to their lowercase counterparts and vice versa. Numeric characters are supposed to remain unchanged. The function contains at least one known bug that results in incorrect output for specific inputs.

```python
def processString(input_str):
  output_str = ""
  for char in input_str:
    if char.isupper():
      output_str += char.lower()
    elif char.isnumeric():
      output_str += char * 2
    else:
      output_str += char.upper()
  return output_str
```

5.1 A

Identify the bug(s) in the code. You can either manually review the code (a form of static analysis) or run it with diverse input values (a form of manual random testing). If you are unable to pinpoint the bug using these methods, you may utilize a random testing tool or implement random test case generator in code. Provide a detailed explanation of the bug, identify the line of code causing it, and describe your strategy for finding it.

- Bug on line 7: numbers should remain the same. This code not only is trying to double the value but it won't even do that! `char * 2` in python will add 2 characters of that number to the output string. Found via inspection and by running the program; at first I thought char * 2 would double the integer version of the char but I was wrong. It is a good thing I tested it in python as well.
- This is the only bug determined by visual analysis and python tests.

5.2 B

Implement Delta Debugging, in your preferred programming language to minimize the input string that reveals the bug. Test your Delta Debugging code for the following input values provided.

```python
# Brett Terpstra
# bt19ex@brocku.ca 6920201


# functional version of the code to test against
def workingProcessString(input_str):
    output_str = ""
    for char in input_str:
        if char.isupper():
            output_str += char.lower()
        elif char.isnumeric():
            output_str += char
        else:
            output_str += char.upper()
    return output_str

# the failing version of the code
def processString(input_str):
    output_str = ""
    for char in input_str:
        if char.isupper():
            output_str += char.lower()
        elif char.isnumeric():
            output_str += char * 2
        else:
            output_str += char.upper()
    return output_str

# function to test an input string
def processTest(input):
    # fancy output.
```

```python
print(f"-------------------[ {input} ]-------------------")
parts_to_test = []
failing_strings = []
passing_strings = []
# simple recursion emulation with a while loop
parts_to_test.append(input)
while parts_to_test:
    # process the front of the queue
    s = parts_to_test[0]
    o1 = workingProcessString(s)
    o2 = processString(s)
    # only make substrings that make sense, recursion base case. Append these to the queue
    if len(s) > 1:
        h1, h2 = s[:len(s)//2], s[len(s)//2:]
        parts_to_test.append(h1)
        parts_to_test.append(h2)
    # remove the front element from the queue
    parts_to_test.pop(0)
    # append to the tracking lists for later printing and sorting
    if o1 != o2:
        failing_strings.append(s)
    else:
        passing_strings.append(s)
# sort the failures in order of largest to smallest
sorted_failings = sorted(failing_strings, key=lambda x: -len(x))
# I don't know what kind of output you were looking for so I printed every bit of information I have
if len(passing_strings) > 0:
    print(f"For input {input} we found {len(passing_strings)} passing strings:")
    for p in passing_strings:
        print(f"\t{p}")
    print()

if len(sorted_failings) > 0:
    print(f"For input {input} we found {len(sorted_failings)} failing strings:")
    for f in sorted_failings:
        print(f"\t{f}")
    print()

    # printing the smallest is probably the most important
    # since it'll tell us the place of the bug
    smallest = len(sorted_failings[len(sorted_failings)-1])
```

```python
        print(f"For input {input} the smallest erroring strings are:")
        for f in sorted_failings:
            if len(f) == smallest:
                print(f"\t{f}")
        print()
    else:
            print("No failing strings found!")

processTest("abcdefG1")
print()
processTest("CCDDEExy")
print()
processTest("1234567b")
print()
processTest("8665")
```

delta_debugger.py

(This file is included in the zip for this assignment. I'm not completely sure why I pasted this here but not the C++. Something Something tired Something Something Python being cleaner than C++)
And some test results running on my computer:

```
-------------------[ abcdefG1 ]-------------------
For input abcdefG1 we found 11 passing strings:
  abcd
  ab
  cd
  ef
  a
  b
  c
  d
  e
  f
  G

For input abcdefG1 we found 4 failing strings:
  abcdefG1
  efG1
  G1
  1

For input abcdefG1 the smallest erroring strings are:
  1


-------------------[ CCDDEExy ]-------------------
For input CCDDEExy we found 15 passing strings:
  CCDDEExy
  CCDD
  EExy
  CC
  DD
  EE
  xy
  C
  C
  D
  D
  E
  E
  x
  y

No failing strings found!

-------------------[ 1234567b ]-------------------
For input 1234567b we found 1 passing strings:
  b

For input 1234567b we found 14 failing strings:
  1234567b
  1234
  567b
  12
  34
  56
  7b
  1
  2
  3
  4
  5
  6
  7

For input 1234567b the smallest erroring strings are:
  1
  2
  3
  4
  5
  6
  7


-------------------[ 8665 ]-------------------
For input 8665 we found 7 failing strings:
  8665
  86
  65
  8
  6
  6
  5

For input 8665 the smallest erroring strings are:
  8
  6
  6
  5
```

delta_debugging_results.txt

I would've preferred C++ but trying to run the python function from C++ in a cross-platform way isn't something I think you or I want to deal with. This python code is a simple but clean implementation of delta debugging which finds a set of minimal strings which cause errors within the program. From the results of the delta debugging program it appears my static analysis was correct in there being only one bug. This delta debugging implementation uses a binary search method where the input string is recursively split in half until either a minimal error string is found or the string can no longer be split. A string is known to error if the result of the incorrect function does not match a known correct output which is generated at runtime. Since there is no hard way to pre-generate valid and invalid strings we must assume the output of my 'correct' function is in fact valid. The actual implementation of this concept uses a while loop with a queue to do the recursion, a queue was used because using a stack is unnecessary. If I had implemented this with C++ I would have used a stack like structure as `std::vector` has a `pop_back` function.

6 Question 6