SENG 5811 Homework #4

Date: 3/09/2023
Due: 3/18/2023
Topics: White-box techniques
Problems: 2 (The first one has six parts and the second one has four parts)
Points: 50

This assignment is to be completed individually. Please submit a PDF of your completed work.

I. **(20 Points)**
In Homework #3 you created a test suite - a collection of test specifications of the form (Relational-operator, Boolean, Relational-operator) – for testing the compound condition:

$$(((a + b) < c) \wedge \neg p) \vee (r > s)$$

We will now examine the fault-finding capability of your test-suite using mutation analysis. Assume that the class of possible faults we are interested in detecting is operator faults, i.e., incorrect relational and logical operators. (E.g., > instead of <, $\wedge$ instead of $\vee$ or vice versa, missing/extra $\neg$).

Questions:

(a) (2 points) Create 4 "mutants" by introducing 4 different faults from the above class of faults into the given predicate. Each mutant should have exactly one introduced fault.

(b) (8 points) Identify which mutants you created are *killed* by each of your test cases in your test suite. A mutant is killed by a test case if the input data for that test case will cause the mutant and the given predicate to evaluate to different truth values.

(c) (3 points) What would you have to do, in theory, to prove that all the test cases in a test suite that you constructed are *necessary* to kill *all possible mutants* formed by introducing faults from the above class of faults into the given predicate? Briefly state the approach you would take, not an actual proof.

(d) (2 points) Are all the tests in your test suite *necessary* for killing such mutants? Briefly justify your answer.

(e) (3 points) What would you have to do to prove that your test suite is *sufficient* for killing *all possible mutants* formed by introducing faults form the above class into the given predicate? Briefly state the approach you would take, not an actual proof.

(f) (2 points) Is your test suite *sufficient* for killing such mutants? Briefly justify your answer.

II. **(30 Points)**

In Homework #3 you created a "requirements-based" test-suite for the following Java program.

- Number of permutations of 3 elements

```
➢ java Choices 3
Output: 3! = 6
```

- Number of 2-element permutations possible from a set of 10 elements

```
➢ java Choices 10 2
Output: 10P2 = 90
```

- Number of 5-element combinations possible from a set of 8 elements

```
➢ java Choices 8 5 -
Output: 8C5 = 56
```

- Unsuitable or no arguments will cause the program to print a USAGE message.

```
➢ java Choices
Output:
Usage: Choices NUMBER1 [ NUMBER2 [-] ]
   Number of possible choices for selecting a tuple (or set) of size
NUMBER2 from a set of size NUMBER1.
   A third argument (must be a -) if present, indicates unordered
selection (set instead of tuple).
   NOTE: [ ] indicate optional arguments; NUMBER2, if not given, is
NUMBER1; both must be non-negative integers.
```

The source code for an implementation of that program in Java is included here at the end of the file.

Use this implementation to answer the following questions in the next page.

Plus, A BONUS OFFER: Extra credits will be awarded for any bug found if reasonably justified as a bug! You may assume that the USAGE string is the *specification* for the program!

Questions:

(a) (8 points) Draw a control-flow graph representation for each method separately (**`main`** and **`choose`**).

(The lecture slides have an example of this for the triangle classification example function. Your goal is to derive such a graph, with decision-to-decision paths condensed into labeled nodes as in slide 35).

A few things to keep in mind with respect to Java semantics for creating control-flow graphs:

- Short-circuiting evaluation semantics needs to be represented in the graph for logical operators. Thus, a code fragment such as "`(a && b)`" should result in two nodes, one for `a` and another for `b` because there are two outgoing edges from `a`, going to different nodes (to the node for `b` when `a` is true and to the node following that code fragment when `a` is false).

- The ternary operator `(c ? x : y)` is really `(if c then x else y)` used as an expression.

- At a method call, the arguments are first evaluated in left-to-right order before the call is made.

(b) (8 points) Find the minimum number of test cases required to exercise:

i. every node in the two control-flow graphs at least once

ii. every edge in the two control-flow graphs at least once

Provide a brief explanation of how you determined the number of test cases needed. Note that a test case is executed on the program by calling the `main` method with the test inputs as arguments. The `choose` method cannot be called directly. A node (or an edge) in the control-flow graph is exercised by a test case if the code fragment represented by the node (or edge) executes when the test case is run on the program.

(c) (8 points) From the test suite that you created for this program in Homework #3.

i. Identify a minimal subset of test cases that achieves maximal *statement* coverage. Briefly justify your answer.

ii. Identify a minimal subset of test cases that achieves maximal *condition* coverage. Briefly justify your answer.

Notes:

A subset of tests cases from a test suite is *minimal* for a given coverage criterion, if removing a test case from that subset will reduce coverage with respect to that criterion.

A subset of test cases from a test suite achieves *maximal* coverage for a given criterion, if adding more test cases from that test suite will not increase coverage with respect to that criterion.

(d) (6 points) Provide a set of test inputs that achieves *modified condition/decision coverage* (MC/DC) for the decision in the *if*-statement at line 20 in the `main` method.

```java
import java.math.BigInteger;

public abstract interface Choices {
    String USAGE = "USAGE: Choices NUMBER1 [ NUMBER2 [-] ]\n" +
          "\tNumber of possible choices for selecting a tuple (or a subset) of size NUMBER2 from a set of size NUMBER1.\n" +
          "\tA third argument if given, must be '-' and it indicates unordered selection (i.e., a subset instead of a tuple).\n" +
          "\t[ ] surround optional arguments; " + "NUMBER2, if not given, is NUMBER1; both must be non-negative integers.\n";

    private static BigInteger choose(final int n, int k, final boolean ordered) {
        if (k > n) return BigInteger.valueOf(0);
        if (ordered || k <= n/2) k = n - k;
        BigInteger num = BigInteger.valueOf(1);
        int i = 0;
        while (k < n) {
            num = num.multiply(BigInteger.valueOf(++k));
            if (!ordered) num = num.divide(BigInteger.valueOf(++i));
        }
        return num;
    }

    public static void main(final String ...argv) {
     A  final int argc = argv.length;
        B  C  D  E  F
        if (argc >= 1 && argc <= 3 && (argc != 3 || (argv[2].length() >= 1 && argv[2].charAt(0) == '-'))) {
         G  final int  arg1 = Integer.parseUnsignedInt(argv[0]);
            final int  arg2 = argc > 1 ? Integer.parseUnsignedInt(argv[1]) : arg1;
            H  I  J  K  L
            final char sym = argc == 3 ? 'C' : argc == 2 ? 'P' : '!';
            System.out.printf("%1$s%2$c%3$s = %4$s%n", argv[0], sym, argc > 1 ? argv[1] : "", choose(arg1, arg2, argc < 3));
        }  S  M  M  N  O  P  Q  R  R  R
        else System.err.println(USAGE);
    }  U  T
}
```