Zuqi Li
20512622
SE465 - Section 001
zq6li@uwaterloo.ca

A1

Q1. I found that C++ and Java implementations returned -1 while Python and Perl implementations returned 1. This discrepancy is due to the fact that C++ and Java treat "%" operator as remainders instead of modulus, which returns negative values for a negative input.

One strategy to cope with this is for the developer to write their own work-around implementation for their desired behaviour. For instance, in C++/Java, a modulus behaviour can be implemended by taking the absolute value of the "%" operator; while in Python/Perl, a remainder behaviour can be implemented by checking the sign beforehard.

Another way to cope with this is by updating the previous standards of languages to have the same behaviour for "%", or create a standardized library for all languages to have the same behaviour if the former is not feasible.

Refer to code samples in the q1 folder.

2. a) When the value of x is null, then a NullPointerException is thrown and the program terminates before the fault is executed.

Input: x = null;

Expected: NullPointerException

Actual: NullPointerException

b) When x is not empty and all of its values are positive, the fault is executed and there is no error.

Input: x = [2,3,4,5];

Expected: 2

Actual: 2

c) In this case, it is not possible to have an error but result in no failure.

d) First Error State:

x = [-10, -9, 0, 99, 100];

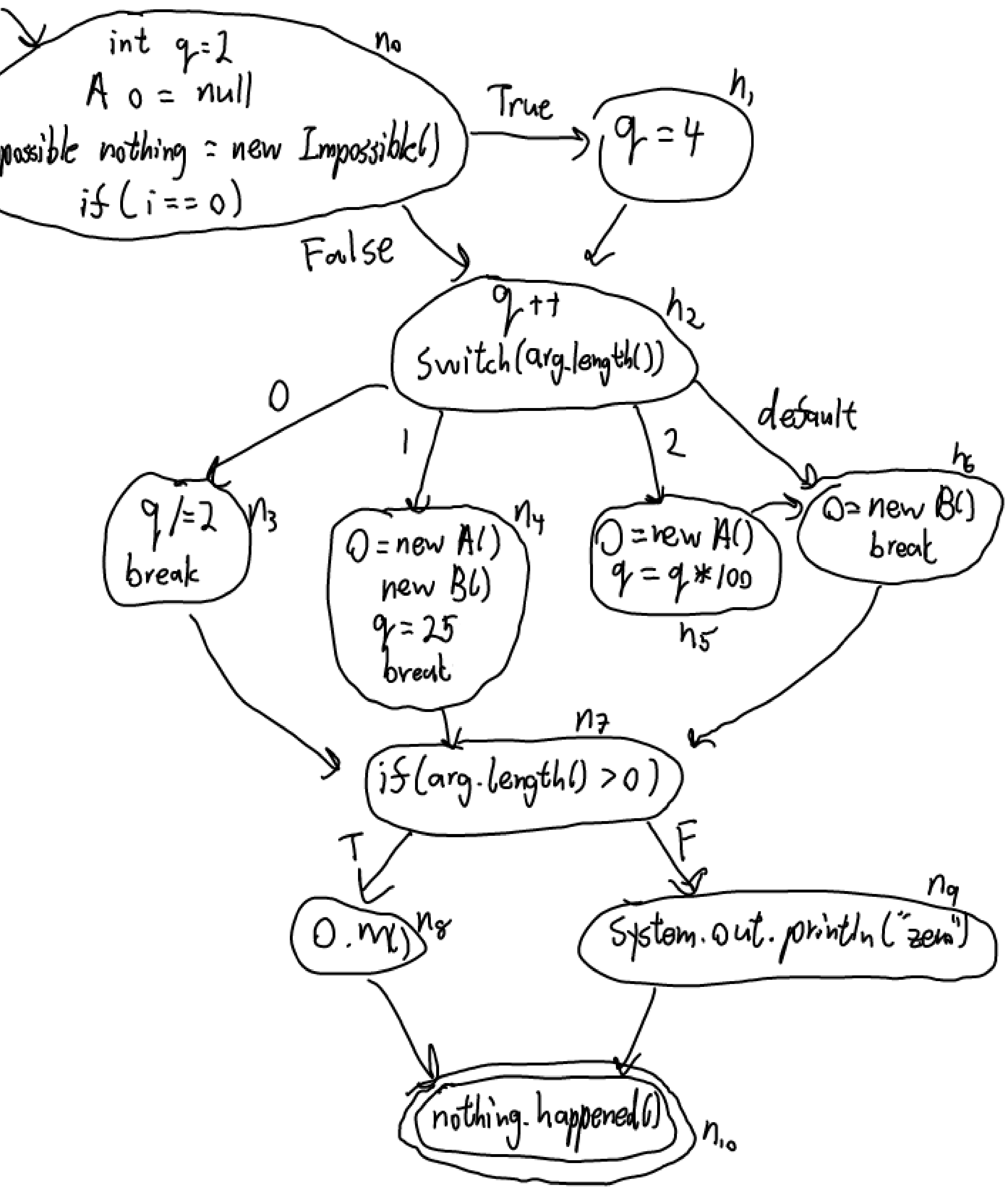
count = 0;

i = 1;

PC = i++;

3. a) refer to next page

3 a)



b) $TR_{NC} = \{ n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10} \}$

$TR_{EC} = \{ [n_0,n_1], [n_0,n_2], [n_1,n_2], [n_2,n_3], [n_2,n_4], [n_2,n_5], [n_2,n_6]$
$[n_5,n_6], [n_3,n_7], [n_4,n_7], [n_6,n_7]. [n_7,n_8], [n_7,n_9], [n_8,n_{10}],$
$[n_9,n_{10}] \}$

$TR_{EPC} = \{[n_0, n_1, n_2], [n_0, n_2, n_3], [n_0, n_2, n_4], [n_0, n_2, n_5], [n_0, n_2, n_6],$
$[n_1, n_2, n_3], [n_1, n_2, n_4], [n_1, n_2, n_5], [n_1, n_2, n_6], [n_2, n_3, n_7],$
$[n_2, n_4, n_7], [n_2, n_5, n_6], [n_2, n_6, n_7], [n_3, n_7, n_8], [n_3, n_7, n_9],$
$[n_4, n_7, n_8], [n_4, n_7, n_9], [n_5, n_6, n_7], [n_6, n_7, n_8], [n_6, n_7, n_9],$
$[n_7, n_8, n_{10}], [n_7, n_9, n_{10}]\}$

$TR_{PPC} = \{[n_0, n_1, n_2, n_3, n_7, n_8, n_{10}], [n_0, n_1, n_2, n_3, n_7, n_9, n_{10}],$
$[n_0, n_1, n_2, n_4, n_7, n_8, n_{10}], [n_0, n_1, n_2, n_4, n_7, n_9, n_{10}],$
$[n_0, n_1, n_2, n_5, n_6, n_7, n_8, n_{10}], [n_0, n_1, n_2, n_5, n_6, n_7, n_9, n_{10}]$
$[n_0, n_1, n_2, n_6, n_7, n_8, n_{10}], [n_0, n_1, n_2, n_6, n_7, n_9, n_{10}],$
$[n_0, n_2, n_3, n_7, n_8, n_{10}], [n_0, n_2, n_3, n_7, n_9, n_{10}],$
$[n_0, n_2, n_4, n_7, n_8, n_{10}], [n_0, n_2, n_4, n_7, n_9, n_{10}],$
$[n_0, n_2, n_5, n_6, n_7, n_8, n_{10}], [n_0, n_2, n_5, n_6, n_7, n_9, n_{10}],$
$[n_0, n_2, n_6, n_7, n_8, n_{10}], [n_0, n_2, n_6, n_7, n_9, n_{10}]\}$

Infeasible Tests $= \{[n_3, n_7, n_8], [n_4, n_7, n_9], [n_6, n_7, n_9],$
$[n_0, n_1, n_2, n_3, n_7, n_8, n_{10}], [n_0, n_1, n_2, n_4, n_7, n_9, n_{10}],$
$[n_0, n_1, n_2, n_5, n_6, n_7, n_9, n_{10}], [n_0, n_1, n_2, n_6, n_7, n_9, n_{10}],$
$[n_0, n_2, n_3, n_7, n_8, n_{10}], [n_0, n_2, n_4, n_7, n_9, n_{10}],$
$[n_0, n_2, n_5, n_6, n_7, n_9, n_{10}], [n_0, n_2, n_6, n_7, n_9, n_{10}]\}$

These tests are infeasible because if args.length() == 0 at n_2, then it cannot be greater than 0 at n_7. Similarly, if args.length == 1, 2, or more, then it cannot be 0 or less at n_7.

c) Refer to code

4. refer to CFG.Java.

f) refer to CFGTest.java for the added test cases.

<u>addNode</u>: The test cases satisfy NC & EC.
This method branches into 2 cases. The case when a node is not present is satisfied by addNode() and the case when a node already exists is satisfied by addNode_duplicate();

<u>addEdge</u>: The test cases do not satisfy NC & EC.
The test cases are not covering the case where the second node does not exist. Therefore I added "addEdge_oneNewNodeRight()" to test that case.

<u>deleteNode</u>: The test cases satisfy NC & EC.
The test cases cover both the case of when the node exists in the graph and the case of the node not existing. It also tests going through my for loop and out, therefore it covers all of the edges.

<u>deleteEdge</u>: The test does not satisfy EC.
The test case misses the edge of when my method's second node does not exist and exists the program. Therefore I added the case "deleteNode_missingDestNode()" for testing missing second node.

isReachable: The test cases satisfy NC & EC.

The tests check for missing source node as well as missing destination nodes. It also covers both cases of whether or not a neighbouring node has been visited or not. Finally, it checks for both the cases of when a node destination is reachable and the case of when it is not. Therefore, it covers all possible edges.