ST 2019 Past Paper

1. The following function finds the sum of all positive integers, below an input number n, that are divisible by 3 or 5.

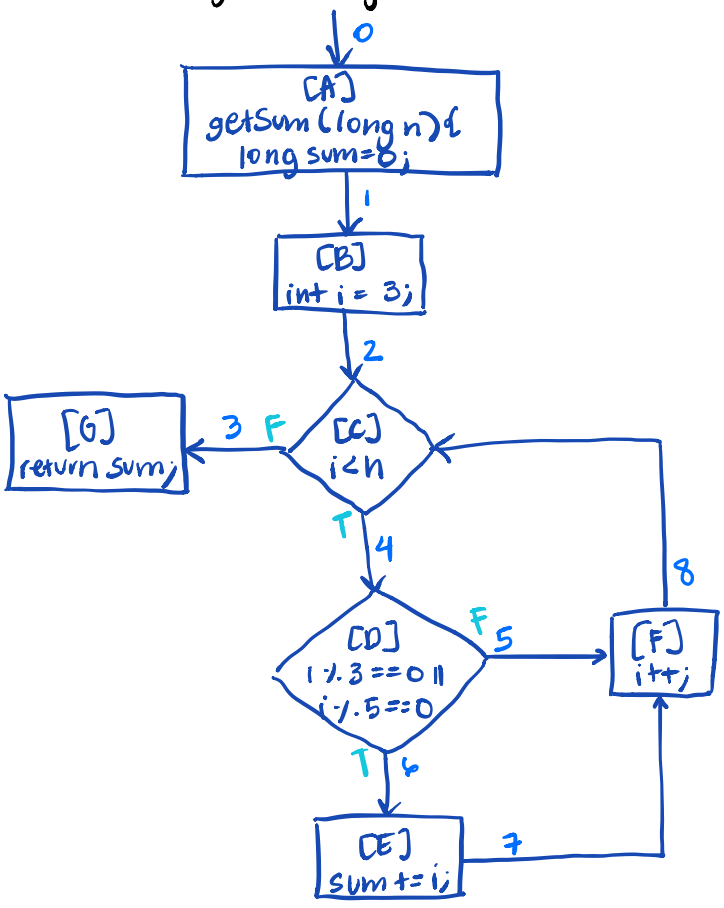
```
1   public static long getSum(long n) {
2     long sum = 0;
3     for (int i = 3;
4           i < n;
5           i++) {

6           if (i % 3 == 0 || i % 5 == 0)
7              sum += i;
8     }
9     return sum;
10  }
```

Test suites for getSum:

	Test	Input (n)	Expected output
#1	1	0	0
	2	4	3
#2	3	-1	0
	4	6	8

a. Draw the control flow graph for getSum.



b. Branch coverage achieved by Test Suites 1 and 2.

	Test	Input (n)	Expected output	Sequence of edges
#1	1	0	0	0, 1, 2, 3
	2	4	3	0, 1, 2, 4, 6, 7, 8, 3
#2	3	-1	0	0, 1, 2, 3
	4	6	8	0, 1, 2, 4, 6, 7, 8, 4, 5, 8, 4, 6, 7, 8, 3

Test suite #1 coverage = 8/9
Test suite #2 coverage = 9/9

c. Compute MC/DC coverage achieved by Test Suite #1 for each boolean expressions in the functions. If not 100%, write additional tests. Do the same for Test Suite #2.

There are 2 boolean expressions that need to be covered when considering MC/DC:

1. i < n
2. (i % 3 == 0) || (i % 5 == 0)

The first expression results in 2 MC/DC obligations - one that makes the expression true (e.g. i = 3 and n = 4) and the other that makes it false (e.g. i = 3 and n = 0). Tests achieve 100% MC/DC if they cover both these cases.

The second expression results in 3 MC/DC obligations. Any OR expr. of the form a||b has the MC/DC obligations:

1. a = T b = T
2. a = F b = T
3. a = T b = F

Test Suite #1 achieves 100% MC/DC for expression 1 as both obligations are covered.

Test Suite #1 achieves 33% MC/DC for expression 2 as only the third obligation among the three obligations is covered.

Test Suite #2 achieves 100% MC/DC for expression 1 and achieves 67% MC/DC for expression 2.

Input n = 16 achieves 100%..

d. Coverage criteria for loops + compute coverage for test suites.

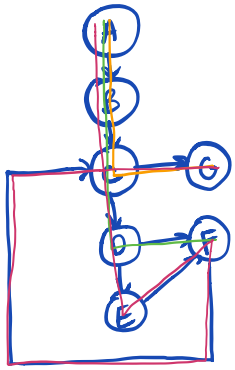
Loop boundary adequacy, boundary interior testing, LCSAJ, and cyclomatic adequacy criterion.

Loop boundary — loop executed 0 times = TS1.1 (n=0), TS2.1 (n=-1)
— loop executed exactly once = TS1.2 (n=4)
— loop executed more than once = TS2.2 (n=6)

TS#1 loop adequacy = 2/3 = 66.7 %. → Answer for Test-1 and Test-2 coverage!
TS#2 loop adequacy = 2/3 = 66.7 %.

ALTERNATIVE SOLUTION

Boundary Interior Testing



— 1st Path
— 2nd Path
— 3rd Path

Test-1 : n=0 (1st Path)
Test-2 : n=4 (2nd Path)

2/3 covered.

To cover 3rd path, n=6

e. Show tests with input and expected output that achieves "All DU pairs" coverage + write out <D,U> pairs.

Variable	Definitions	Uses	<D,U> pairs
n	1	4	<1,4>
sum	2, 7	7, 9	<2,7>, <2,8>, <7,7>, <7,8>
i	3, 5	4, 5, 6, 7	<3,4>, <3,5>, <3,6>, <3,7>, <5,4>, <5,5>, <5,6>, <5,7>

Tests from above achieve all DU pairs coverage! 100%.

f. Avail equations for Line 5 (need to provide Avail and AvailOut equations for lines 7 and 6).

The only way to get to Line 5 is from the inside of the loop body (Line 7 if statement on Line 6 was true and directly from Line 6 if it was false) because the incrementation of i in a for loop happens at the end of the loop body, before comparing i<n.

Avail(5) = AvailOut(6) ∧ AvailOut(7)

We continue:

AvailOut(6) = (Avail(6) \ kill(6)) ∪ gen(6) = Avail(6)

→ nothing is generated or killed at line 6

AvailOut(7) = (Avail(7) \ kill(7)) ∪ gen(7) = (Avail(7) \ {sum2, sum7y}) ∪ {sum7y}

→ sum may have been defined at line 2 or line 7, depending on which iteration of the loop we are on, so both are killed. Either way, a new sum is created on line 7.

We look at how we can get to line 6 (only from line 4) and line 7 (only from line 6):

Avail(6) = AvailOut(4)

Avail(7) = AvailOut(6)

We can now put everything we derived into the original equation:

Avail(5) = AvailOut(6) ∧ AvailOut(7)
= Avail(6) ∧ ((Avail(7) \ {sum2, sum7y}) ∪ {sum7y})
= AvailOut(4) ∧ ((AvailOut(6) \ {sum2, sum7y}) ∪ {sum7y})
= AvailOut(4) ∧ ((Avail(6) \ {sum2, sum7y}) ∪ {sum7y})
= AvailOut(4) ∧ ((AvailOut(4) \ {sum2, sum7y}) ∪ {sum7y})

g. Modification of line 6:

if ((i%2==0) && (i%3==0 || i%5==0)) → all even multiples of 3 or 5

Are tests 1-4 adequate? If not, add.

Tests 1-4 are not adequate as no one will execute sum += i.

MC/DC can be used to check for complex conditions.

Let a = i%2==0
b = i%3==0
c = i%5==0

a	b	c	a && (b || c)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4 tests needed.

* Test where a=0 RES=1 (RES=0 not possible!)
b=0 RES=1 and RES=0 c=0 RES=1 and RES=0
b=1 RES=1 and RES=0 c=1 RES=1 and RES=0

2. isPalindrome(char seq[250]) ← returns 1 if seq is a palindrome
returns 0 if seq is not a palindrome
returns -1 if input is invalid

a. If isPalindrome is an ITF, identify its characteristics.

seq's characters can be alpha-numeric (abc..123) or other (?!*..).
seq's length can be <=250 or >250.
seq can be a palindrome or not.

b. Define partitions/value classes for the characteristics.

Characteristic	Partitions	Value classes
seq's characters	P1 P2	alpha-numeric non-alpha-numeric (other)
seq's length	P3 P4	<=250 >250
seq == palindrome	P5 P6	True False

c. Generate exhaustive test case specs for the characteristics.

Test	seq. characters	seq. length	seq==palindrome	Expected output
a	alpha-numeric	<=250	TRUE	1
b	other	<=250	TRUE	-1
c	alpha-numeric	<=250	FALSE	0
d	other	<=250	FALSE	-1
e	alpha-numeric	>250	TRUE	-1
f	other	>250	TRUE	-1
g	alpha-numeric	>250	FALSE	-1
h	other	>250	FALSE	-1

d. Name and describe at least 2 common concurrency bugs.

Data Race - occurs when two conflicting accesses to one shared var. are executed without proper synchronisation.
e.g. not protected by a lock

Deadlock - occurs when two or more operations circularly wait for each other to release the acquired resource.
e.g. locks

Atomicity violation bugs - bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.

Order violation bugs - bugs that don't follow the programmer's intended order.

e. Assume the following function is called using multiple threads. Arguments amount, account_from, account_to are shared between the threads. What concurrency bug is this function prone to + prevention method + code for corrected function.

```
boolean bankTransfer (int amount, Acct account_from, Acct account_to)
{
  if (account_from.balance < amount) return false;
  account_to.balance += amount;
  account_from.balance -= amount;
  return true;
}
```

Order violation bugs - bugs that do not follow the programmer's intended order.

CODE FIX

(synchronized boolean bankTransfer (int amount, Acct account_from, Acct account_to)

```
boolean bankTransfer (int amount, Acct account_from, Acct account_to)
{
  if (account_from.balance < amount) return false;
  account_to.balance += amount;
  account_from.balance -= amount;
  return true;
}
```

f. What is concolic testing?

Testing method that combines concrete and symbolic execution.

Concolic execution workflow:

1. Execute the program for real on some input and record path taken.
2. Encode path as query to SMT solver and negate one branch condition.
3. Ask the solver to find new satisfying input that will give a different path.

g. Explain how concolic testing would work for input = (4, 5, 8, 9, 4, 5, 0, 3)

```
public int BinarySearch(int[] sortedArray, int key, int low, int high)
{
    int index = Integer.MAX_VALUE;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (sortedArray[mid] < key) {
            low = mid + 1;
        } else if (sortedArray[mid] > key) {
            high = mid - 1;
        } else if (sortedArray[mid] == key) {
            index = mid;
            break;
        }
    }
    return index;
}
```

recorded path constraint w/ the input

Recorded path constraint with the input is (low <= high) && !(sortedArray[mid] < key) && !(sortedArray[mid] > key) && (sortedArray[mid] == key).

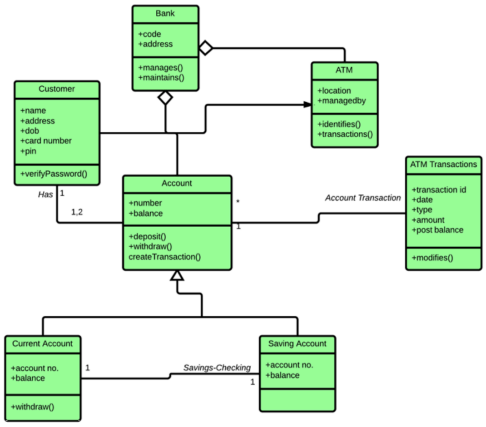
New path constraints can be generated by negating any of the existing conditions in the recorded path constraint and keep going.

3. a. Why is interclass testing in OO programs carried out? Explain how.

test interactions between classes.

HOW - Bottom-up integration according to 'depends' relation.
→ A depends on B: build test B, then A
- start from use/include hierarchy
→ Implementation-level parallel to logical 'depends' relation
- class A makes method calls on class B
- class A objects include references to class B methods
- But only if reference means 'is part of'

b. Draw use-include hierarchy for the following diagram. Which interactions would you test?



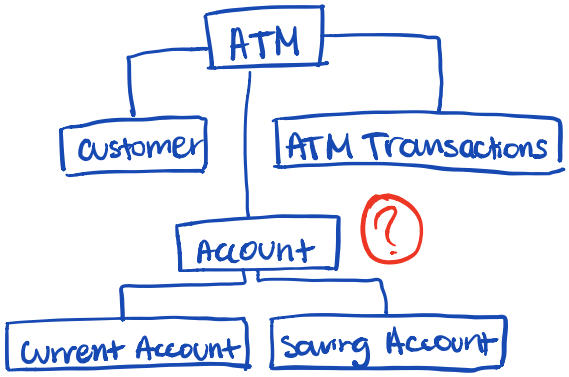


Figure 1: Question 3b: Class Diagram of ATM Trasactions

c. TDD for the following.

```
Take a list of integers and size of the list as inputs,
Spec 1: If size of list is 0 or greater than 50, return -100.
Spec 2: If any number in the list is negative, return -1.
Spec 3: If there are no negative numbers and
        if any number in the list is zero, return 0.
Spec 4: If all numbers in the list are positive,
          return maximum value.
```

Test 1: Return 0 for all. This test must initially fail.

Code Stage 1: def function (list, n):
return 0

Test 2: Return -100 if the size of the list is 0 or greater than 50
Input: [] Expected output: -100

Code stage 2: def function(list, n):
```
    if n==0 or n>50:
        return -100
```

Test 3: Return -1 if there is a negative number in the list and n>0 and n<=50.
Input: [0,1,2,-1] Expected output: -1

Code stage 3: def function(list, n):
```
    if n==0 or n>50:
        return -100
    elif (sum([i<0 for i in list])==1):
        return -1
```

Test 4: Return 0 if there is a 0 in the list AND no negative numbers
Input: [0,1,2,3] Expected output = 0

Code stage 4: def function(list, n):
```
    if n==0 or n>50:
        return -100
    elif (sum([i<0 for i in list])==1):
        return -1
    elif (sum([i==0 for i in list])==1):
        return 0
```

Test 5: Return max(list) if there are no zeros or negative numbers.
Input: [1,2,3] Expected output: 3

Code stage 5: def function(list, n):
```
    if n==0 or n>50:
        return -100
    elif (sum([i<0 for i in list])==1):
        return -1
    elif (sum([i==0 for i in list])==1):
        return 0
    return max(list)
```

d. Pros/cons of mutation-based and generation-based fuzzing.

MUTATION-BASED FUZZING
- ⊕ easy to get started
- ⊕ no (or little) knowledge of specific input format needed
- ⊖ typically yields low code coverage - inputs tend to deviate too much from expected format and are rejected by early sanity checks.
- ⊖ hard to reach 'deeper' parts of programs by random-guessing (e.g. nested control flows)

GENERATION-BASED FUZZING

⊕ input is much closer to expected, much better coverage
⊕ can include models of protocol state machines to send messages in the sequence expected by SUT
⊖ requires input format to be known
⊖ may take considerable time to write the input format grammar/specification
⊖ not widely applicable

e. For the following function that computes factors of an integer input a, create one mutation that "replaces an arithmetic operator" and another mutation that "replaces a constant by a scalar variable". Provide both mutations by only showing the changed line in each. Present tests to reveal each of these two mutations, showing input, expected output and actual output.

```
1    public ArrayList<Integer> allFactors(int a) {

2        int upperlimit = (int)(Math.sqrt(a));
3        ArrayList<Integer> factors = new ArrayList<Integer>();
4        for(int i=1;i <= upperlimit; i+= 1){
5            if(a%i == 0){
6                factors.add(i);
7                if(i != a/i){
8                    factors.add(a/i);
9                }
10           }
11       }
12       Collections.sort(factors);
13       return factors;
14   }
```

MUTATIONS

Mutation	Line no.	Mutation type	change
M1	2	constant → scalar	int upperlimit = 0 - (int) (Math.sqrt(a));
M2	4	arithmetic op.	i < upperlimit

TESTS (to kill mutants)

Test	Input	Expected output	Actual Output
a	9	1,3	nothing → reveals M1
b	9	1,3	1 → reveals M2