Toronto Metropolitan University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Software Testing and QA
Course Number:	COE 891
Semester/Year (e.g. F2017)	W2025

Instructor	Reza Samavi

Assignment/Lab Number:	Lab 4
Assignment/Lab Title:	Control Flow Graph and Data Flow Coverage

Submission Date:	03/19/2025
Due Date:	03/19/2024

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Hamza	Malik	501112545	09	HM

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

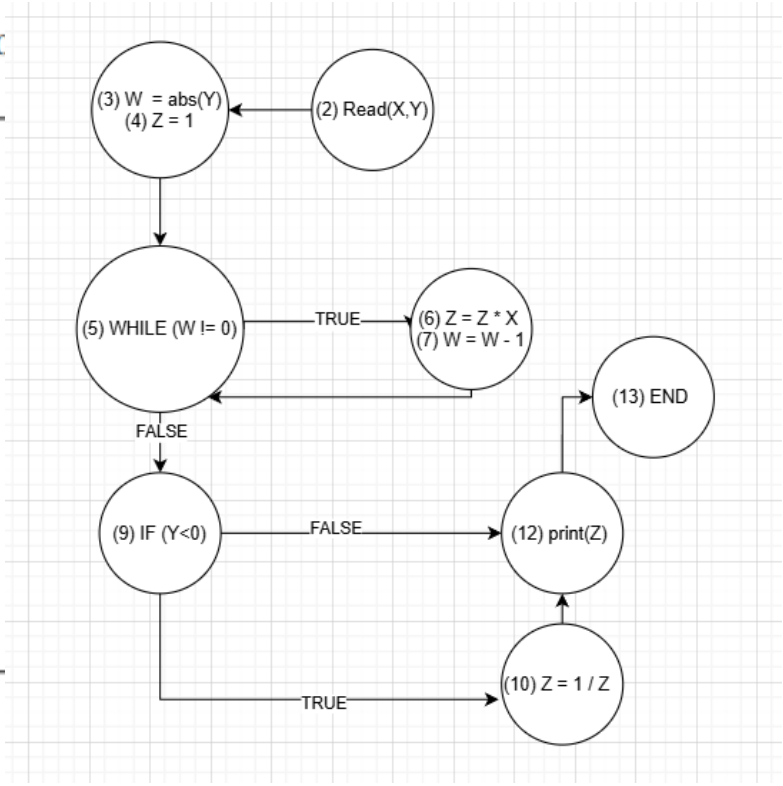
Question 1:

This program computes $Z = X^{Y}$ by first converting Y to a non negative integer. It then multiples the Z by X in a loop for W times. If the value of Y is less than zero, then the value of Z is inverted. Finally, it prints out the Z value.

Q1. Power function is a program to cc

```
1   BEGIN
2       read (X, Y);
3       W = abs(Y);
4       Z = 1;
5       WHILE (W != 0)
6           Z = Z * X;
7           W = W - 1;
8       END
9       IF (Y < 0)
10          Z = 1 / Z;
11      END
12      print (Z);
13  END
```

Identifv:



Control Flow:

- From reading inputs (line 2) and initializing (line 3–4), we reach the WHILE check (line 5).
- If W != 0, the loop body (lines 6–7) executes and flows back to the WHILE.
- When W == 0, we exit the loop and check the IF (Y < 0) (line 9).
- If true, invert Z (line 10). If false, go directly to print(Z).
- Finally, the program prints the result and ends.

a. **Infeasible paths:** There are no infeasible paths. If we choose Y = 0, the WHILE loop will become false. The WHILE loop can become true if we choose an absolute value that is greater than 0. We can make the IF statement true by choosing a value less then 0 or false by choosing a value greater or equal to 0.

b. **Enough test cases for node coverage:** Around four test cases can cover all the nodes.
   i. **Test Case 1:** (X = 2, Y = 2).

1. The WHILE loop will run twice which covers line 6 and 7. The IF is less than 0, so it will be skipped and the program will print the results.

ii. **Test Case 2:** (X = 2, Y = -2).

1. The WHILE loop runs twice, covering lines 6 and 7. The IF is greater than 0, so it will run line 10 and the program will print the results.

c. **Enough test cases for edge coverage:** Edge coverage is when every node/branch has been covered at least once across the entire test case set.

i. **Test Case 1:** (X = 2, Y = 0).

1. The WHILE loop is **False** from the start (covers the False edge of line 5). The IF condition (Y<0) is **False** (covers the False edge of line 9). Executes lines 2, 3, 4, 5(False), 9(False), 12, 13.

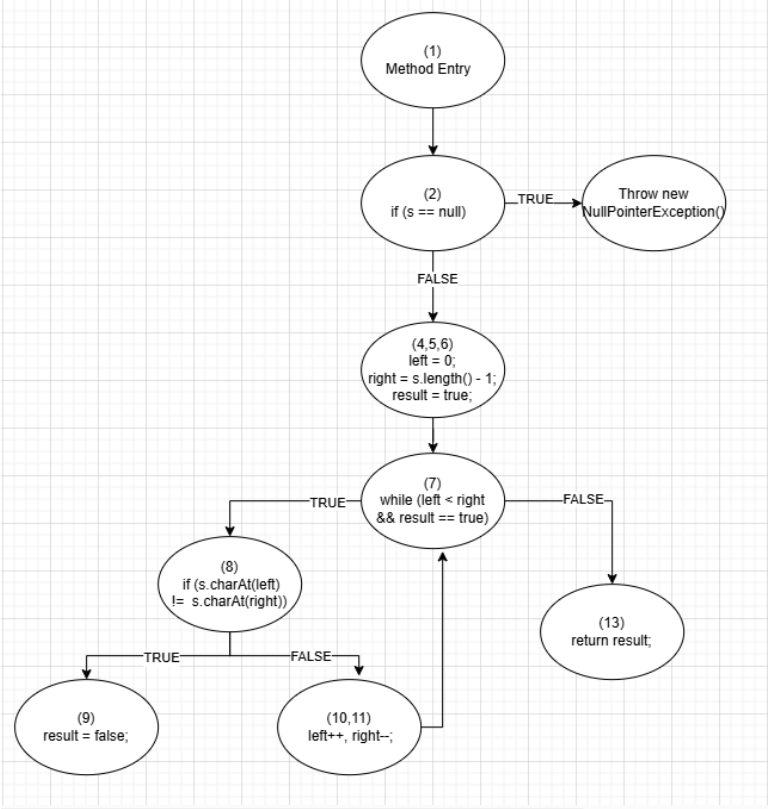
ii. **Test Case 2:** (X = 2, Y = 2).

1. The WHILE loop is **True** (covers the True edge) for two iterations, then eventually False to exit. The IF condition (Y<0) is **False**. Covers the loop's **True** branch, the IF's **False** branch. Executes lines 2, 3, 4, 5(True), 6, 7, 9(False), 12, 13.

iii. **Test Case 3:** (X = 2, Y = -2).

1. The WHILE loop is **True** (covers that edge) for two iterations, then False to exit. The IF condition (Y<0) is **True** (covers line 10). Covers the loop's True branch, the IF's True branch. Executes lines 2, 3, 4, 5(True), 6, 7, 9(True), 10, 12, 13.

Question 2:



Q2. For the program shown below:

```
1   public static boolean isPalindrome(String s) {
2       if (s == null)
3           throw new NullPointerException();
4       int left = 0;
5       int right = s.length() - 1;
6       boolean result = true;
7       while (left < right && result == true) {
8           if (s.charAt(left) != s.charAt(right)) {
9           result = false; }
10          left++;
11          right--; }
12      return result;
13      }
```

Control Flow:

- If s is null, immediately throw a NullPointerException.
- Initialize pointers left=0 and right=s.length()-1, and set result=true.
- While left<right **and** result==true, compare characters at s[left] and s[right].
  - If they differ, set result=false. Else, keep result = true.
  - After each comparison, do left++ and right--.
- Exit the loop if left >= right or if the result becomes false.
- Return result.

a) **Node Coverage (NC):** Every node in the CFG is executed at least once across the entire test. The following two tests ensure every statement/node is visited at least once.

i) **Test Case 1:** (s == Null)

1) The IF (s == null) → **True** → throw NullPointerException. This will covers the "throw" node but does not cover the rest of the method

ii) **Test Case 2:** (s == "abca")

1) IF (s == null) → **False** The while (left < right && result==true) → **True**, the loop sets result = false if it finds a mismatch. This will cover the loop

body, the IF inside the loop, the result=false assignment, and the final return.

b) **Edge Coverage (EC):** Every branch (True or False) in the CFG is executed at least once across the entire test. The following three tests ensure that all edges are covered.
   i) **Test Case 1:** (s = null)
      1) Covers IF (s == null) → **True** → throw → exit.
   ii) **Test Case 2:** (s = " ") or ("a"), (Empty string or a single character)
      1) Then left=0, right =- 1 (for "") or right=0 (for "a"). So left < right is **False** from the start → skip the loop → return **True** This covers the **False** edge of WHILE from the beginning .
   iii) **Test Case 3:** (s = "abca")
      1) **First iteration:** compare 'a' vs 'a' → mismatch? No → the inner IF is **False**,
      2) **Second iteration:** compare 'b' vs 'c' → mismatch? Yes → the inner IF is **True**, set result=false, loop ends.
      3) This covers the loop's **True** edge (twice) and the WHILE **False** edge upon result==false. We also see the inner if → **False** and then **True**

c) **Edge Pair Coverage (EPC) / Prime Path Coverage (PPC):** The following test cases will cover Multiple consecutive "false" outcomes of the inner if. A "true" outcome of the inner if. The while loop for multiple iterations (for "abba"). Immediate exit from the while loop (for "") and the throw path.
   i) **Test Case 1:** (s = null)
      1) Throws Exception
   ii) **Test Case 2:** (s = " ")
      1) The loop is **False** from start (covers no iterations).
   iii) **Test Case 3:** (s = "abba")
      1) Any palindrome that has the length > 2, will ensure that there are multiple loop iterations where the inner IF is **False** repeatedly.
   iv) **Test Case 4:** (s = "abca")
      1) This case is not a palindrome. It triggers the inner IF =False on the first iteration, then IF =True on the second iteration, forcing result=false and exiting the loop.

d) **NC but not EC:** {Null, "abc"}

e) **EC but not EPC:** {Null, " ", "abca"}

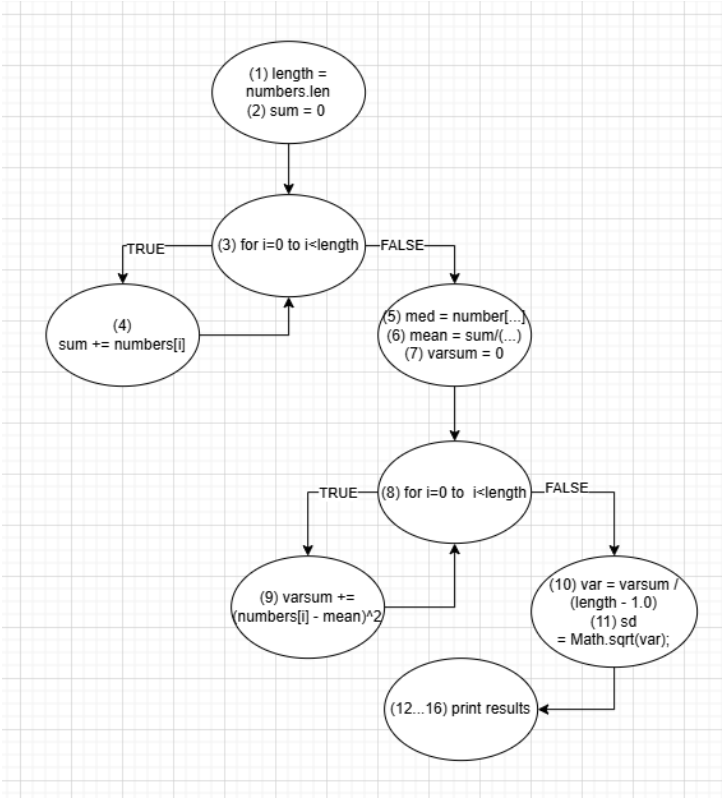
f) **EPC/PPC:** {Null, " ", "abba", "abca"}

g) Any path requiring Result = False in a multiple loop iteration would be infeasible because once the Result is set to False, the loop will terminate and end. For example, a path requiring IF inside loop → True (set result=false) in iteration 1 and again → True in iteration 2, it would be infeasible. After the first result=false, the loop ends.

h) **All possible test cases:**

1. (1 → 2 → Throw → Exit) for **S = Null**
2. (1 → 2 → 3 → 4 → 8 → Exit) for **S = “”**
3. (1 → 2 → 3 → 4 → 5 → 7) for **S = “aa”**
4. (1 → 2 → 3 → 4 → 5 → 6 → 7) for **S = “ab”**
5. (4 → 5 → 7 → 4) for **S = “aa”**
6. (4 → 5 → 6 → 7 → 4) for **S = “ab”**

Question 3:




```java
public static void computeStats(int[] numbers) {
    int length = numbers.length;          // (1)
    double med, var, sd, mean, sum, varsum;
    sum = 0;                              // (2)
    for (int i = 0; i < length; i++)      // (3)
        sum += numbers[i];                // (4)
    med = numbers[length / 2];            // (5)
    mean = sum / (double) length;         // (6)
    varsum = 0;                           // (7)
    for (int i = 0; i < length; i++)      // (8)
        varsum += (numbers[i] - mean) * (numbers[i] - mean);
    var = varsum / (length - 1.0);        // (10)
    sd = Math.sqrt(var);                  // (11)

    System.out.println("length: " + length);  // (12)
    System.out.println("median: " + med);     // (13)
    System.out.println("mean: " + mean);      // (14)
    System.out.println("variance: " + var);   // (15)
    System.out.println("standard deviation: " + sd);  // (16)
```

Control Flow:

- Line (1) and (2) initialize length and sum.
- Line (3) and (4) include the first **for** loop from i=0 to i<length, summing array elements.
- Line (5),(6), and (7) compute the median, mean, and set varsum=0.
- Line (8) and (9) include the second **for** loop from i=0 to i<length, and accumulate (numbers[i] - mean)^2.
- Line (10) and (11) compute the variance (var) and standard deviation (sd).
- Line (12) to (16) print the results and exit the program.

Variable	Def	Use	Du Pairs
length	(1) length = numbers.length	- Loop 1 condition (3): i < length - Index (5): numbers[length / 2] - Division (6): sum / (double) length - Loop 2 condition (8): i < length - Division (10): varsum / (length - 1.0) - Print (12): printing length	D(1) → U(3), U(5), U(6), U(8), U(10), U(12)
sum	(2) sum = 0 (4) sum += numbers[i] (re- def after each iteration)	- (6) mean = sum / (double) length This is mainly for printing the mean, but that's after we store it in mean, so the direct use is line (6) only.	D(2) → U(4) → re- def at (4) for each iteration → final use at (6)
med	(5) med = numbers[length/2]	- (13) printing "median: " + med	D(5) → U(13)
mean	(6) mean = sum / (double) length	- (9) varsum += (numbers[i] - mean) * (numbers[i] - mean) inside loop 2 - (14) printing "mean: " + mean	D(6) → U(9), U(14)
varsum	(7) varsum = 0 (9) varsum += … re- def each iteration in loop 2	- (10) var = varsum / (length - 1.0)	D(7) → U(9) → re- def at (9) → final use at (10)
var	(10) var = varsum / (length - 1.0)	- (11) sd = Math.sqrt(var) - (15) printing "variance: " + var	D(10) → U(11), U(15)
sd	(11) sd = Math.sqrt(var)	- (16) printing "standard deviation: " + sd	D(11) → U(16)
i	**First loop:** (3) for (int i=0; i<length; i++) **Second loop:** (8) for (int i=0; i<length; i++)	**First loop:** - i is defined at loop initialization (i=0) and updated each iteration - i is used in the loop condition (i<length) and in (4) numbers[i] **Second loop:** - Similarly, local definition for i=0, used in i<length and in (9) numbers[i]	**First loop:** D(3.init) → U(3.cond), U(4), re- def by i++ → U(3.cond)... **Second loop:** D(8.init) → U(8.cond), U(9), re- def by i++ → U(8.cond)...

Test cases to cover Du Paths:

The following 4 test cases will show a crash on an empty array (which covers out-of-bounds), a 1-element scenario (which covers the minimum valid length), and multiple elements, which shows that the loops run more than once, covering the definitions and uses of all variables in multi-iteration contexts.

a) **Test Case 1:** Length of 0. (numbers = { })
   i) The first FOR loop does 0 iterations. Then line (5) attempts numbers[0], which throws ArrayIndexOutOfBoundsException. This partially covers the definition of length (line 1) and is used in the first loop condition, but it never reaches the rest. In general, the code crashes at line (5), so it does not fully demonstrate any subsequent uses.

b) **Test Case 2:** Length of 1. (numbers = {5})
   i) The first loop runs once (sum = 5). Med = numbers[0] = 5. Mean = 5/ (double)1 = 5.0.
   ii) The second loop runs once, computing varsum += (5 - 5.0)^2 = 0. var = 0/(1 - 1.0) = 0/0.0 → NaN so the standard deviation is also NaN. This covers the definitions and uses of sum, med, mean, varsum, var, sd in a 1-element scenario.

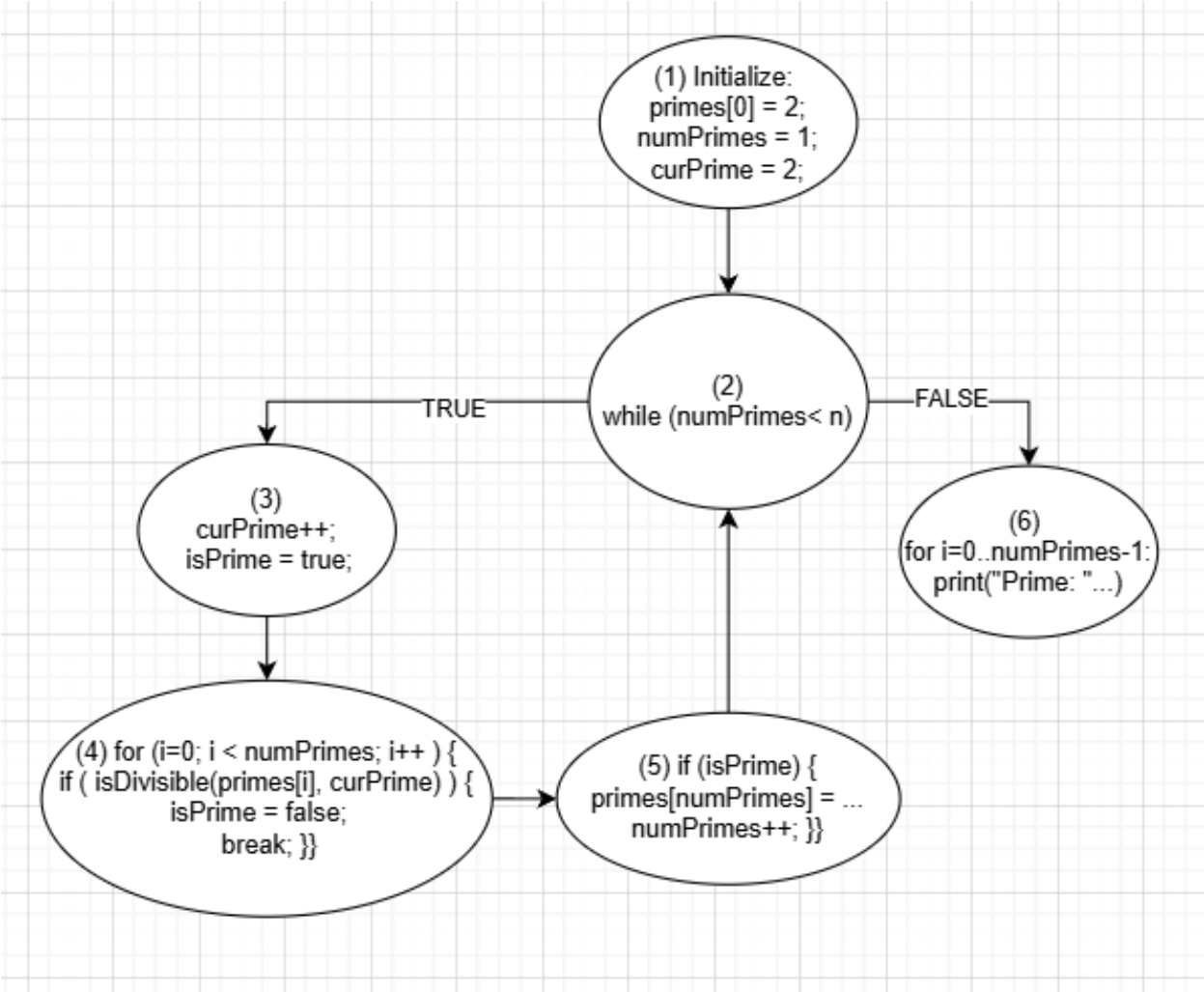
c) **Test Case 3:** Length of 2. (numbers = {2, 4})
   i) The first loop runs twice. sum=2+4=6. med = numbers[2/2] = numbers[1] = 4. mean=6/2=3.0.
   ii) The second loop runs twice: each iteration updates varsum += (value - 3)^2. So (2-3)^2 + (4-3)^2 = 1+1=2. var = 2 / (2-1.0)= 2.0. sd=√2.0=1.4142. This covers multi-iteration usage and redefinition of sum and varsum.

d) **Test Case 4:** Length 3. (numbers = {1, 3, 5})
   i) Summation loop: sum=1+3+5=9. med = numbers[3/2]= numbers[1]=3. mean=9/(double)3=3.0.
   ii) Second loop: (1-3)^2+(3-3)^2+(5-3)^2 = 4+0+4=8. var=8/(3-1.0)=8/2=4.0, sd=2.0. This again exercises multi-iteration usage but with an odd length.

1. What happens if number.length == 0?

a. At line (5), we do med = numbers[length/2] = numbers[0]. This is an invalid index (out of bounds), which will cause an ArrayIndexOutOfBoundsException.
b. In addition, line (6) does sum / (double) length = sum / 0.0, which would produce NaN or Infinity, but we never reach that line because the program already crashed at line (5).

Question 4:



Control Flow:

- Program begins from **(1)** (initialization) to **(2)** (the while condition).
- The while (numPrimes < n) has a **True** edge going into (3) and a **False** edge skipping straight to (6).
- Inside the loop, from (3) to (4) (the for loop).
- The for loop may take the **break** if isDivisible(...) is true, or may complete without a break.
- After the for loop, we reach the **if (isPrime)** block (5).
- Then control flows back to the While condition (2).
- Eventually, when the while is false, we go to (6), the final printing loop.

a) Test case such that the corresponding test path visits the edge that connects the beginning of the while statement to the for statement that appears after the while loop, without going through the body of the while loop:

i) **Test Case 1:** (n = 1)

1) Given a value of (n< 1), the While(1 < 1) is false, so we jump directly to the final For (i=0; i<numPrimes; i++), which prints only the first prime (2) and terminates. This execution visits the "False" edge from the while test without entering the loop body.

b) **Test path that achieves Edge Coverage but not Prime Path Coverage on the graph:**

For this question, using n=1 and n=3, we do not use up all possible simple cycles or prime paths, but we do indeed cover all the edges. The following test cases are examples of achieving Edge Coverage but not Prime Path Coverage:

i) **Test Case 1:** (n = 1)

1) The while is false immediately (covers the "False" edge of the while). Skips the loop, goes to final for and prints 2.

ii) **Test Case 2:** (n = 3)

1) **Start:** numPrimes=1, curPrime=2, while(1<3)? → True.
2) **Loop iteration #1:** curPrime=3, isPrime=true. for(i=0..0): check isDivisible(2,3)? → false → no break → isPrime remains true. If (isPrime) → true → store 3, numPrimes=2.
3) **Loop iteration #2:** while(2<3)? → True → curPrime=4, isPrime=true. for(i=0..1): check divisibility by 2 then 3. On i=0: isDivisible(2,4)? → true → break out, set isPrime=false. if (isPrime)? → false → skip storing.
4) **Loop iteration #3:** while(2<3)? → still True → curPrime=5, isPrime=true. for(i=0..1): check divisibility by 2 → false, by 3 → false → no break → remains true. if (isPrime)? → true → store 5, numPrimes=3.
5) **Next check:** while(3<3)? → false → exit loop.
6) **Final:** for(...): prints 2, 3, 5.

iii) **Overall Coverage:** The Test case covers the while loop = True, multiple times, then eventually turning False. The For loop in the body is taken multiple times. The "break" inside the for loop (for curPrime=4). The "no break" path (for curPrime=3 and curPrime=5). The If (isPrime) True branch (for 3 and 5) and False branch (for 4).

Appendix:

```java
package question_2;

import static org.junit.Assert.*;



public class Palindrome_Test {
    Palindrome p = new Palindrome();

    @Test (expected = NullPointerException.class)
    public void testPalindromeNull() {
        String str = null;
        Palindrome.isPalindrome(str);
    }
    @Test
    public void testPalindrome1() {
        String str = "ba";
        assertFalse(Palindrome.isPalindrome(str));
    }
    @Test
    public void testPalindrome2() {
        String str = "baaa";
        assertFalse(Palindrome.isPalindrome(str));
    }
}

```

Figure 1: Palindrome Test Class

```java
package question_3;
import org.junit.Test;


public class ComputeStats_Test {

        @Test
        public void testComputeStats() {
            ComputeStats.computeStats(new int[]{5});
        }

    }


```

Figure 2: ComputeStats Test Class

```java
package question_4;
import static org.junit.Assert.*;

public class Prime_test {
    @Test
    public void testPrintPrimesWithZero() {
        try {
            isPrime.printPrimes(0);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }

    @Test
    public void testPrintPrimesWithOne() {
        try {
            isPrime.printPrimes(1);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }

    @Test
    public void testPrintPrimesWithTwo() {
        try {
            isPrime.printPrimes(2);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }

    @Test
    public void testPrintPrimesWithThree() {
        try {
            isPrime.printPrimes(3);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }

}
```

Figure 3: Prime Number Test Class