Nathan Krieger
SE 417
Assignment 1
02/05/2026

3. Based on the reports, it looks like the PrimeNumberFinder class is pretty well covered, but missing some key tests. The isPrime function is only at 70% coverage and the for-loop which contains the most complicated logic is not well tested.

PrimeNumberFinder

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isPrime(int)		70%		54%	9	13	4	12	0	1
computeSumOfPrimes(List)		80%		75%	1	3	1	7	0	1
findPrimes(int, int)		100%		100%	0	3	0	5	0	1
PrimeNumberFinder()		100%		n/a	0	1	0	1	0	1
Total	24 of 117	79%	12 of 32	62%	10	20	5	25	0	4

```java

    /* method to find primes between (including) two numbers */
    public static List<Integer> findPrimes(int lowerBound, int upperBound) {
        List<Integer> primeNumbers = new ArrayList<>();

        for (int number = lowerBound; number < upperBound; number++) {
            if (isPrime(number)) {
                primeNumbers.add(number);
            }

        }

        return primeNumbers;
    }

    /* method to compute the sum of primes given a list of prime numbers */

    public static int computeSumOfPrimes(List<Integer> primes) {
        int sum = 0;

    if(primes.size()>1){
            for (int prime : primes) {
               sum += prime;
        }
    }
    else
        sum=primes.get(0);
        return sum;
    }


    /* method to ask if a single number is prime */


    public static boolean isPrime(int num) {
    if (num < 1) {
        return false;
    }

    if (num == 2 || num == 3 || num == 7) {
        return true;
    }

    if (num % 2 == 0 || num % 3 == 0 || num % 6 == 0) {
        return false;
    }

    if (num > 5 && num % 5 == 0) {
        return false;
    }

    for (int i = 5; i * i <= num; i += 6) {
        if (num % i == 0 || num % (i + 2) == 0) {
            return false;
        }
    }

    return true;
    }

```

4.

a. Below are the tests I've added, each test has a comment showing what lines of the Jacoco report that I am testing, along with my reason for the test. The test code itself displays the input and output of the test (input shown with the method parameter, oracle shown with the assert statement, all tests passed).

```java
//test for a negative number to enter the first if condition
// line 67, 68 in jacoco report.
@Test
public void testIsPrime3() {
    assertFalse(PrimeNumberFinder.isPrime(-1));
}

// The next 3 tests are to check the 2nd if condition in isPrime expilicitly.
// Test 4 and 5 also test the logic for the first 2 conditions of the 3rd if statement.
// line 71, 75, 76 in jacoco report.
@Test
public void testIsPrime4() {
    assertTrue(PrimeNumberFinder.isPrime(2));
}

// line 71, 75, 76 in jacoco report.
@Test
public void testIsPrime5() {
    assertTrue(PrimeNumberFinder.isPrime(3));
}

// line 71, 72 in jacoco report.
@Test
public void testIsPrime6() {
    assertTrue(PrimeNumberFinder.isPrime(7));
}

// This test is to test the 3rd condition in the 3rd if statement.
// However, this will never reach because if a number is divisible by 6, it is also divisible by 2 and 3.
// line 75, 76 in jacoco report.
@Test
public void testIsPrime7() {
    assertFalse(PrimeNumberFinder.isPrime(6));
}
```

```java
// This test is to hit the 4th if condition
// line 79, 80 in jacoco report.
@Test
public void testIsPrime10() {
    assertFalse(PrimeNumberFinder.isPrime(25));
}

// Tests to check the for-loop logic in isPrime
// line 83-87 in jacoco report.
@Test
public void testIsPrime8() {
    assertFalse(PrimeNumberFinder.isPrime(26));
}
// line 83-87 in jacoco report.
@Test
public void testIsPrime9() {
    assertFalse(PrimeNumberFinder.isPrime(49));
}

// line 83-87 in jacoco report.
// Hits the "num % i == 0" branch
@Test
public void testIsPrime_LoopCondition1() {
    assertFalse(PrimeNumberFinder.isPrime(121));
}

// line 83-87 in jacoco report.
// Hits the "num % (i + 2) == 0" branch
@Test
public void testIsPrime_LoopCondition2() {
    assertFalse(PrimeNumberFinder.isPrime(91));
}

// line 89 in jacoco report.
// Hits the "return true" AFTER the loop finishes
@Test
public void testIsPrime_PassesLoop() {
    assertTrue(PrimeNumberFinder.isPrime(17));
}
```

```java
// This test is to hit the 4th if condition
// line 79 in jacoco report.
@Test
public void testIsPrime11() {
    assertTrue(PrimeNumberFinder.isPrime(5));
}
```

b. Below is my new code coverage.

```
66.     public static boolean isPrime(int num) {
67.     if (num < 1) {
68.         return false;
69.     }
70.
71.     if (num == 2 || num == 3 || num == 7) {
72.         return true;
73.     }
74.
75.     if (num % 2 == 0 || num % 3 == 0 || num % 6 == 0) {
76.         return false;
77.     }
78.
79.     if (num > 5 && num % 5 == 0) {
80.         return false;
81.     }
82.
83.     for (int i = 5; i * i <= num; i += 6) {
84.         if (num % i == 0 || num % (i + 2) == 0) {
85.             return false;
86.         }
87.     }
88.
89.     return true;
90.     }
91.
```

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
computeSumOfPrimes(List)		80%		75%	1	3	1	7	0	1
isPrime(int)		100%		95%	1	13	0	12	0	1
findPrimes(int, int)		100%		100%	0	3	0	5	0	1
PrimeNumberFinder()		100%		n/a	0	1	0	1	0	1
Total	6 of 117	94%	2 of 32	93%	2	20	1	25	0	4

c. The maximum line coverage is 100%, and the maximum branch coverage is 95%. It was a challenge to cover all possible branches of each if statement. There is an if statement where 100% branch coverage is not possible. It is impossible to reach 100% on line 75 because the 3rd condition (num % 6 == 0) is not reachable due to the first 2 conditions. Take num = 6 as an example, 6 % 2 == 0, so it will return false before evaluating the 3rd condition.

5.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isPrime(int)		100%		95%	1	13	0	12	0	1
computeSumOfPrimes(List)		100%		100%	0	3	0	7	0	1
findPrimes(int, int)		100%		100%	0	3	0	5	0	1
PrimeNumberFinder()		100%		n/a	0	1	0	1	0	1
Total	0 of 117	100%	1 of 32	96%	1	20	0	25	0	4

```
49.     public static int computeSumOfPrimes(List<Integer> primes) {
50.         int sum = 0;
51.
52.     if(primes.size()>1){
53.             for (int prime : primes) {
54.                 sum += prime;
55.         }
56.     }
57.     else
58.         sum=primes.get(0);
59.         return sum;
60.     }
61.
```

I had to add 1 test to obtain 100% coverage

```
// tests for sumofP with an empty list to evaluate line 52 if statement to false and hit line 58
@Test
public void sumofP2() {
List<Integer> input = Arrays.asList(1);
assertEquals(1,PrimeNumberFinder.computeSumOfPrimes(input));
}
```

B. This is not good design, for each number in the list, there should be a check to ensure that the number is prime before computing the sum. Besides the name, all this method is right now is a list summer.

6.

a.

```java
// Faults and exceptions
@Test
public void testIsPrime_fault1() {
    assertFalse(PrimeNumberFinder.isPrime(1));
}

@Test
public void testFindPrimes_fault2() {
    assertArrayEquals(new Integer[]{2,3,5,7}, PrimeNumberFinder.findPrimes(1,7).toArray());
}

@Test
public void testComputeSum_Exception() {
    assertEquals(0, PrimeNumberFinder.computeSumOfPrimes(new ArrayList<Integer>()));
}
```

Fault 1: 1 is not a prime number, yet it returns true.
Fault 2: findPrimes does not check the last element of the array, causing it to miss the prime number if it is in the last index of the array.
Exception 1: computeSumOfPrimes does not have logic for checking if the array is empty, causing it to throw an exception when it references an index that doesn't exist.

b. I think both faults could be easily overlooked but especially fault 1. It requires knowledge that 1 is not a prime. The block of code that should have caught it is:

```
c.  if (num < 1) {
d.          return false;
e.      }
```

But there it is using < instead of <= which is especially subtle. To find this fault I had to specifically test the code with 1 as the parameter.

7.
a.
Fault 1 fix: add the missing = sign.

```java
public static boolean isPrime(int num) {
if (num <= 1) { // FIX: add = sign
    return false;
}

if (num == 2 || num == 3 || num == 7) {
    return true;
}

if (num % 2 == 0 || num % 3 == 0 || num % 6 == 0) {
    return false;
}

if (num > 5 && num % 5 == 0) {
    return false;
}

for (int i = 5; i * i <= num; i += 6) {
    if (num % i == 0 || num % (i + 2) == 0) {
        return false;
    }
}

return true;
}
```

Fault 2 fix: add missing = sign in order to check the last element.

```java
/* method to find primes between (including) two numbers */
public static List<Integer> findPrimes(int lowerBound, int upperBound) {
    List<Integer> primeNumbers = new ArrayList<>();

    for (int number = lowerBound; number <= upperBound; number++) { // FIX: changed < to <= to include upperBound
        if (isPrime(number)) {
            primeNumbers.add(number);
        }

    }

    return primeNumbers;
}
```

Exception 1 fix: check if the list is empty before accessing elements.

```java
/* method to compute the sum of primes given a list of prime numbers */

public static int computeSumOfPrimes(List<Integer> primes) {
    int sum = 0;

if (primes.isEmpty()) { // Fix: check if the list is empty before accessing elements
    return 0;
}

if(primes.size()>1){
        for (int prime : primes) {
            sum += prime;
    }
}
else
    sum=primes.get(0);
    return sum;
}
```

My code coverage and test results after applying the fixes above:

PrimeNumberFinder

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isPrime(int)		100%		95%	1	13	0	12	0	1
computeSumOfPrimes(List)		100%		100%	0	4	0	9	0	1
findPrimes(int, int)		100%		100%	0	3	0	5	0	1
PrimeNumberFinder()		100%		n/a	0	1	0	1	0	1
Total	0 of 122	100%	1 of 34	97%	1	21	0	27	0	4

```
[INFO] Running PrimeNumberFinderTest
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.039 s - in PrimeNumberFinderTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.478 s
[INFO] Finished at: 2026-02-04T14:04:36-06:00
[INFO] ------------------------------------------------------------------------
```

B. I am still not able to reach 100% code coverage due to the redundant logic of the third if-statement of isPrime(). In order to fix this I can simply remove the num % 6 == 0 check. As explained above, this never gets hit because a number that is divisible by 6 is also divisible by 2 and 3. This makes the check unneeded, it can be safely removed.

```java
public static boolean isPrime(int num) {
if (num <= 1) { // FIX: add = sign
    return false;
}

if (num == 2 || num == 3 || num == 7) {
    return true;
}

if (num % 2 == 0 || num % 3 == 0) { // removed num % 6 == 0 check
    return false;
}

if (num > 5 && num % 5 == 0) {
    return false;
}

for (int i = 5; i * i <= num; i += 6) {
    if (num % i == 0 || num % (i + 2) == 0) {
        return false;
    }
}

return true;
}
```

Resulting code coverage:

```
67.     /* method to ask if a single number is prime */
68.
69.
70.     public static boolean isPrime(int num) {
71.     if (num <= 1) { // FIX: add = sign
72.         return false;
73.     }
74.
75.     if (num == 2 || num == 3 || num == 7) {
76.         return true;
77.     }
78.
79.     if (num % 2 == 0 || num % 3 == 0) {
80.         return false;
81.     }
82.
83.     if (num > 5 && num % 5 == 0) {
84.         return false;
85.     }
86.
87.     for (int i = 5; i * i <= num; i += 6) {
88.         if (num % i == 0 || num % (i + 2) == 0) {
89.             return false;
90.         }
91.     }
92.
93.     return true;
94.     }
95.
96.
```