Khushveen Kaur Umra

[redacted]

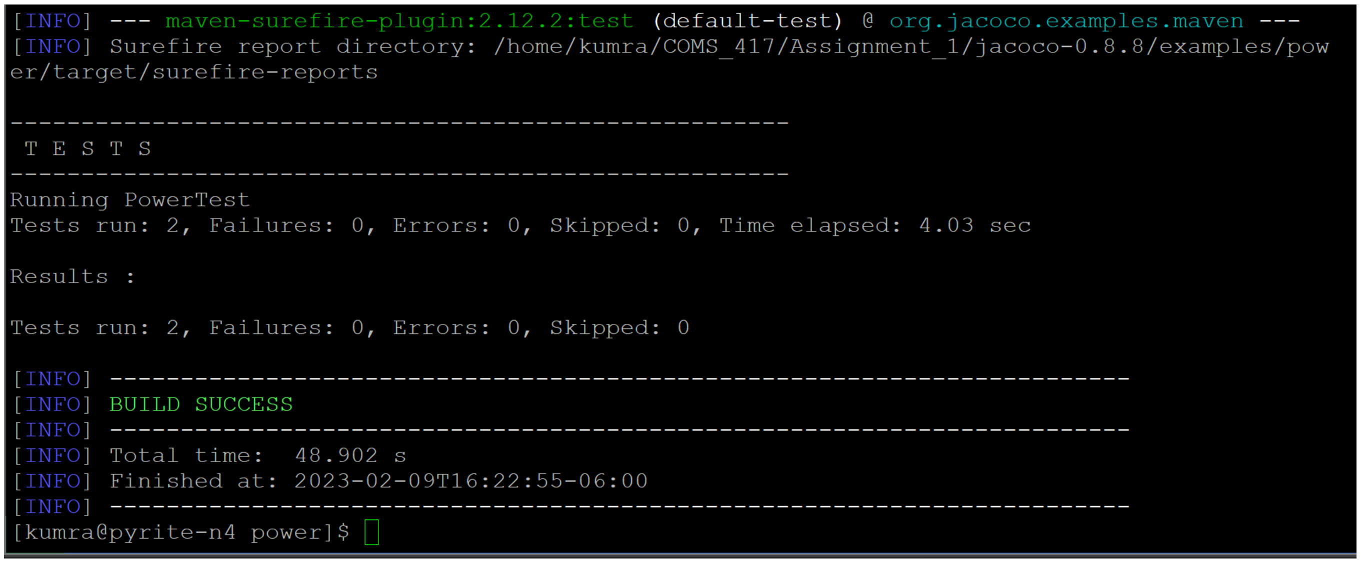
SE 417 Sec 1

9th February 2023

Assignment 1 Report – Power

1) **Set up the Jacoco program to run either on pyrite and/or your local machine. Run the initial test cases using the “mvn” commands as shown in class (and provided on the lecture notes). This is run inside of the power main directory.**

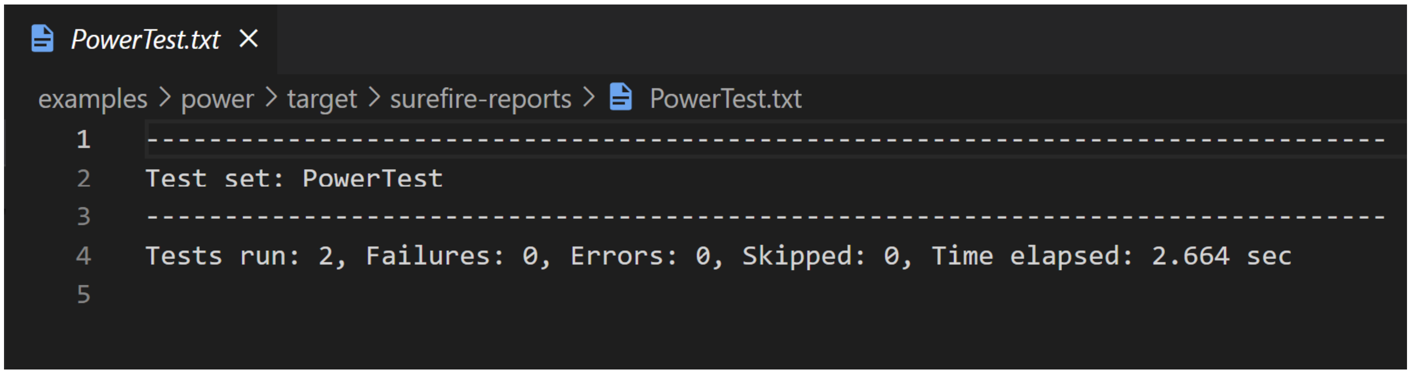
➔ After running the mvn clean, and mvn test commands, you can see that it shows both the tests have been complied successfully with no errors.



```
[INFO] --- maven-surefire-plugin:2.12.2:test (default-test) @ org.jacoco.examples.maven ---
[INFO] Surefire report directory: /home/kumra/COMS_417/Assignment_1/jacoco-0.8.8/examples/pow
er/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running PowerTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.03 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  48.902 s
[INFO] Finished at: 2023-02-09T16:22:55-06:00
[INFO] ------------------------------------------------------------------------
[kumra@pyrite-n4 power]$
```

2) **Look at the initial reports (in the target directory) for JUnit (under surefire) and for Code Coverage (under sites). You should keep these (to hand in). Write a short (sentence or two) summary of what these reports tell you and attach the reports to the assignment.**

➔ **a)** Surefire report: The report for JUnit tests gives us a general indication for how many junit tests there were, how many tests failed, and how many errors were there in the tests. For the initial run of tests, we can see that both the tests for power were successfully complied, without any failures and any errors.

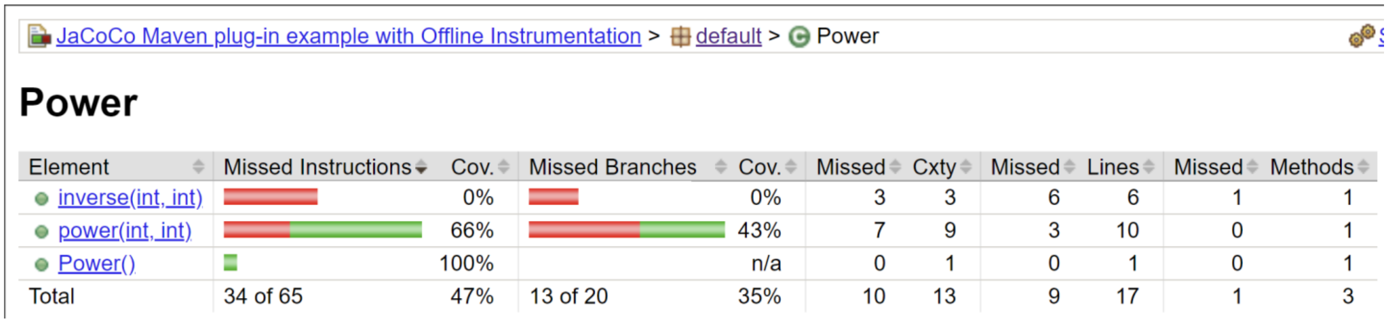


PowerTest.txt

examples > power > target > surefire-reports > PowerTest.txt

```
-------------------------------------------------------------------------------
Test set: PowerTest
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.664 sec

```

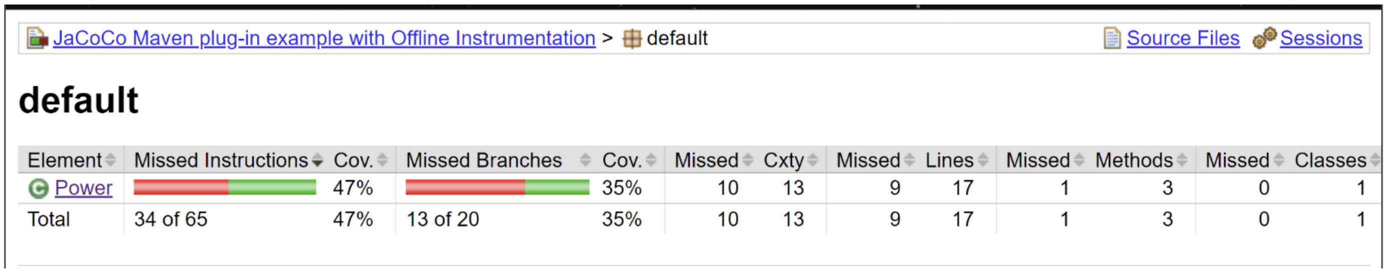
b) Code Coverage: For the code coverage report, we can see that the initial tests had 0% instructions and branch coverage for the inverse() function, and only 66% instruction coverage and 43% branch coverage for the power() function. This means, that the initial junit tests only partially covered some of the instructions for the power() function. Furthermore, after selecting the specific functions in the report, you can visually analyze the code coverage, as jacoco uses diamonds with colors for branches, which lets us know what specific instructions are being covered by our junit tests.



JaCoCo Maven plug-in example with Offline Instrumentation > default > Power

Power

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
inverse(int, int)	🟥	0%	🟥	0%	3	3	6	6	1	1
power(int, int)	🟥🟩	66%	🟥🟩	43%	7	9	3	10	0	1
Power()	🟩	100%		n/a	0	1	0	1	0	1
Total	34 of 65	47%	13 of 20	35%	10	13	9	17	1	3



JaCoCo Maven plug-in example with Offline Instrumentation > default   Source Files   Sessions

default

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Power	🟥🟩	47%	🟥🟩	35%	10	13	9	17	1	3	0	1
Total	34 of 65	47%	13 of 20	35%	10	13	9	17	1	3	0	1

3) **Now add additional tests to increase both the line and branch coverage as high as you can. - List the test cases you have added and for each state:**

(a) The input, the oracle, and the reason you added it (i.e. what condition/statement are you trying to cover – give line numbers from the Jacoco report)

→ Test # 3 – Coverage for line 28 (Left: 2 , Right: 0, Expected: 1)

Test # 4 – Coverage for line 22 (Left: 2 , Right: -3, Expected: -1)

Test # 5 – Coverage for out of bounds, 33 (Left: 11 , Right: 6, Expected: -1)

Test # 6 – Coverage for line 24 (Left: 5 , Right: 3, Expected: 125)

Test # 7 – Coverage for line 24 (Left: 3 , Right: 7, Expected: 2187)

Test # 8 – Coverage for line 47 (Left: -2 , Right: 3, Expected: -8)

Test # 9 – Coverage for out of bounds, 33 (Left: 11 , Right: 11, Expected: -1)

Test # 10 – Coverage for line 26, 33 (Left: -1 , Right: 0, Expected: 1)

Test # 11 – Coverage for line 24 (Left: 5 , Right: 10, Expected: 9765625)

Test # 12 – Coverage for line 24 (Left: 10 , Right: 5, Expected: 100000)

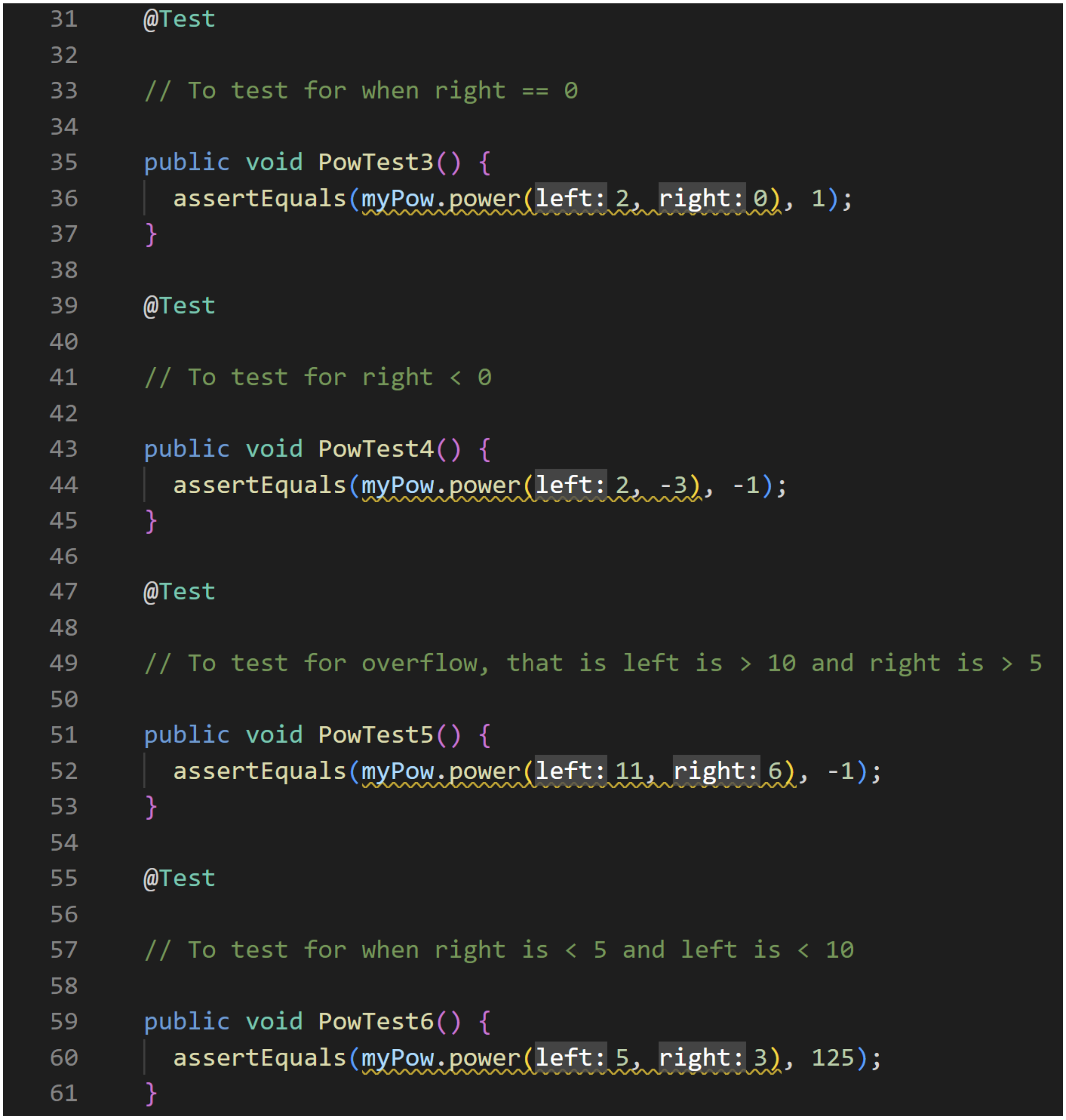
Test # 13 – Coverage for line 24 (Left: 0 , Right: 5, Expected: 0)

Test # 13 – Coverage for line 24 (Left: 0 , Right: 5, Expected: 0)

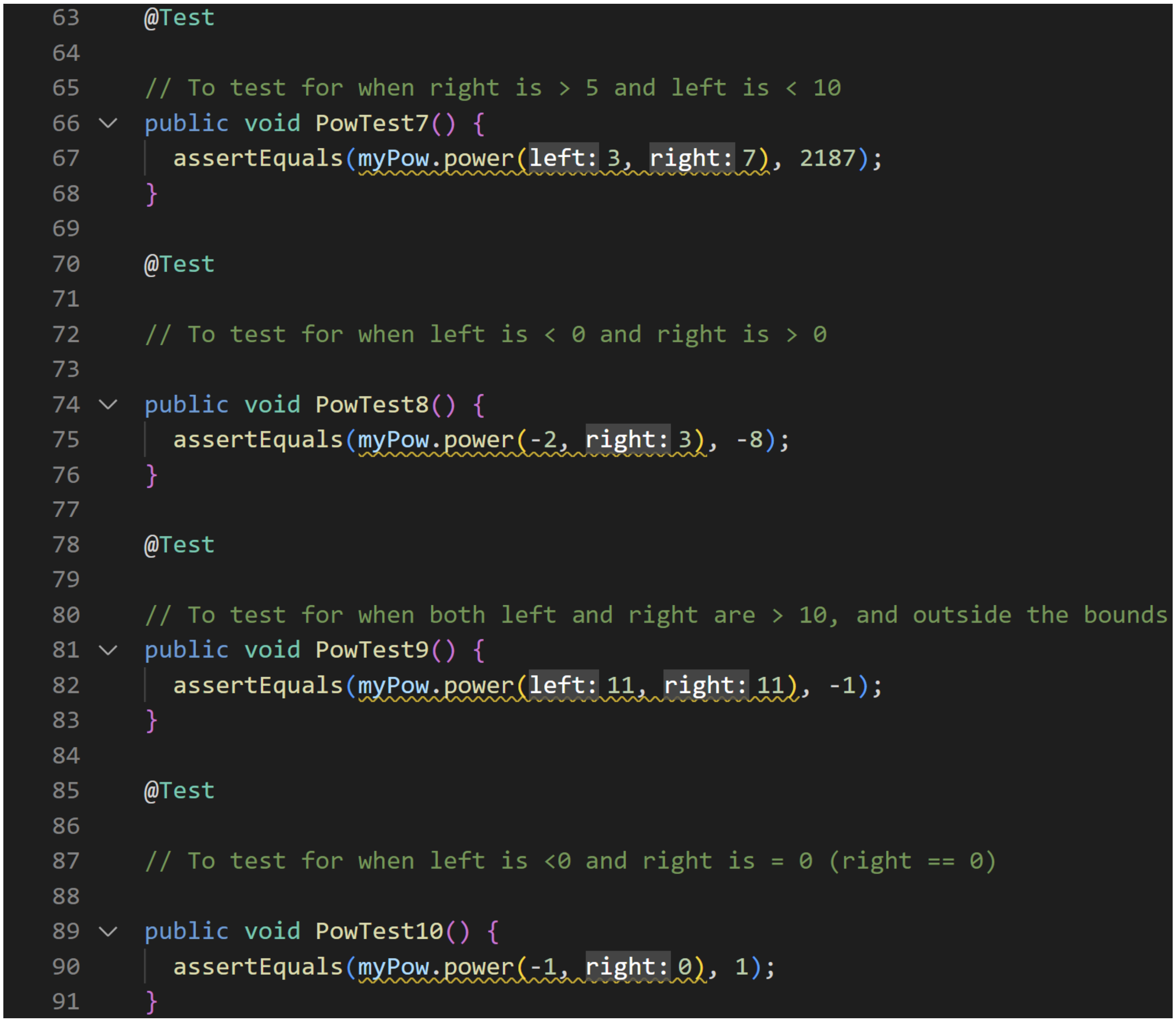
Test # 20 – Coverage for line 65 (Left: 3 , Right: 2, Expected: 8)

Test # 21 – Coverage for line 59 (Left: 2 , Right: 0, Expected: 1)

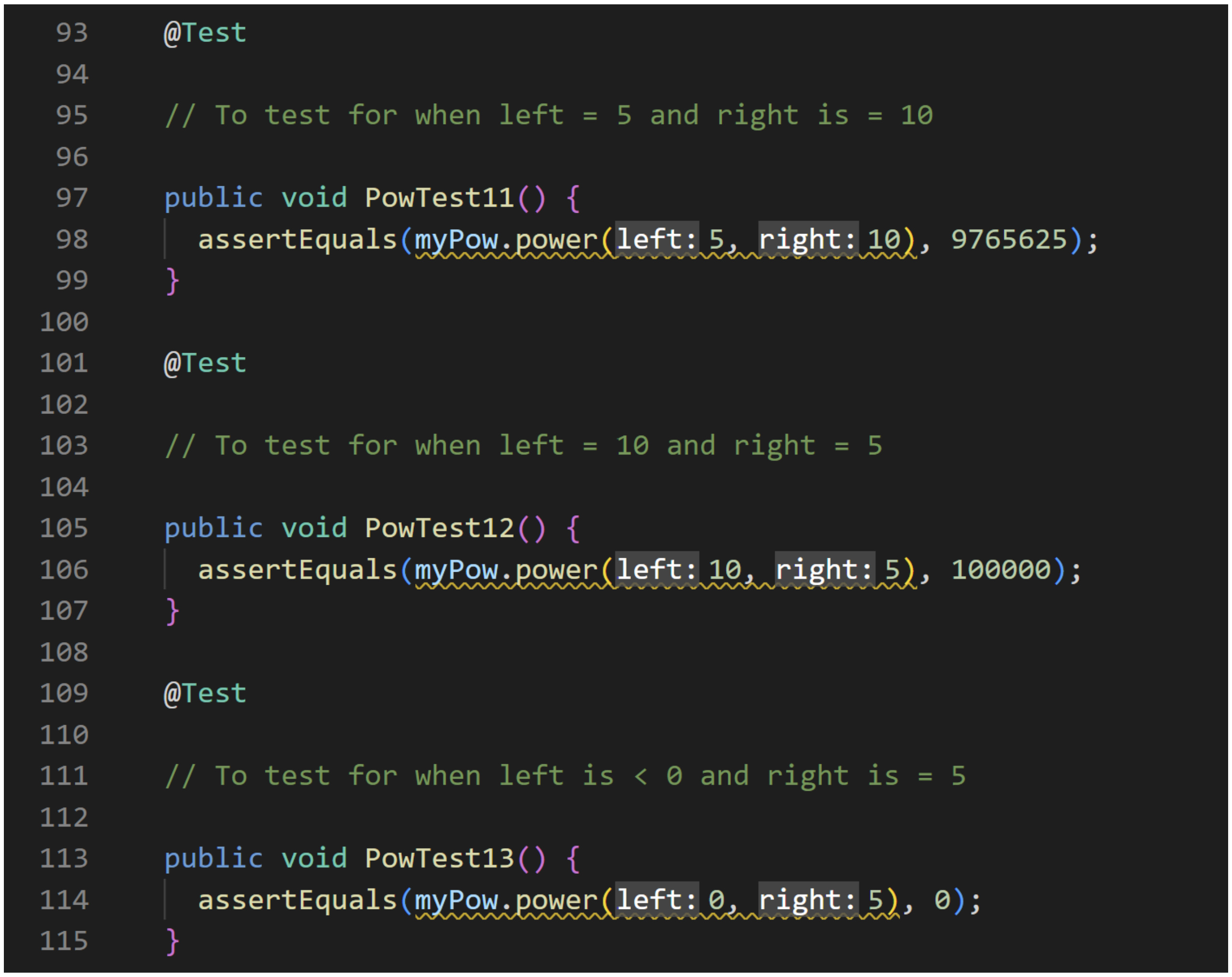
(Screenshots below)



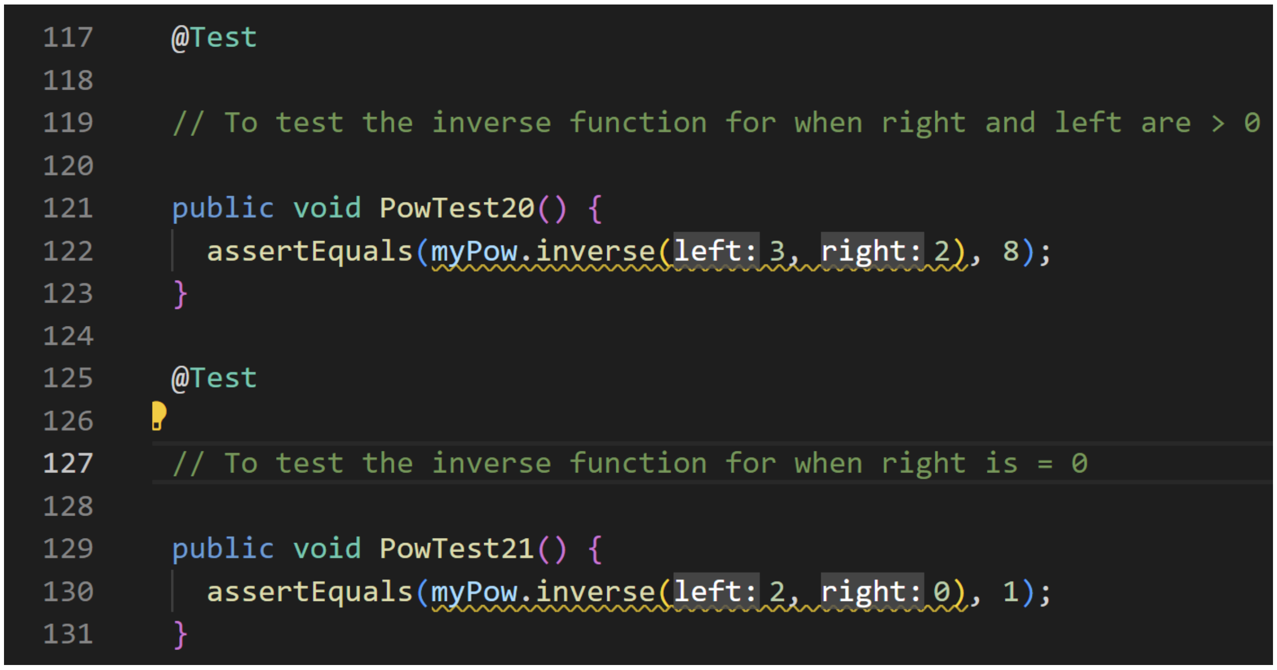
```java
    @Test

    // To test for when right == 0

    public void PowTest3() {
        assertEquals(myPow.power(left: 2, right: 0), 1);
    }

    @Test

    // To test for right < 0

    public void PowTest4() {
        assertEquals(myPow.power(left: 2, -3), -1);
    }

    @Test

    // To test for overflow, that is left is > 10 and right is > 5

    public void PowTest5() {
        assertEquals(myPow.power(left: 11, right: 6), -1);
    }

    @Test

    // To test for when right is < 5 and left is < 10

    public void PowTest6() {
        assertEquals(myPow.power(left: 5, right: 3), 125);
    }
```



```
    @Test

    // To test for when right is > 5 and left is < 10
    public void PowTest7() {
        assertEquals(myPow.power(left: 3, right: 7), 2187);
    }

    @Test

    // To test for when left is < 0 and right is > 0

    public void PowTest8() {
        assertEquals(myPow.power(-2, right: 3), -8);
    }

    @Test

    // To test for when both left and right are > 10, and outside the bounds
    public void PowTest9() {
        assertEquals(myPow.power(left: 11, right: 11), -1);
    }

    @Test

    // To test for when left is <0 and right is = 0 (right == 0)

    public void PowTest10() {
        assertEquals(myPow.power(-1, right: 0), 1);
    }
```



```java
@Test

// To test for when left = 5 and right is = 10

public void PowTest11() {
  assertEquals(myPow.power(left: 5, right: 10), 9765625);
}

@Test

// To test for when left = 10 and right = 5

public void PowTest12() {
  assertEquals(myPow.power(left: 10, right: 5), 100000);
}

@Test

// To test for when left is < 0 and right is = 5

public void PowTest13() {
  assertEquals(myPow.power(left: 0, right: 5), 0);
}
```

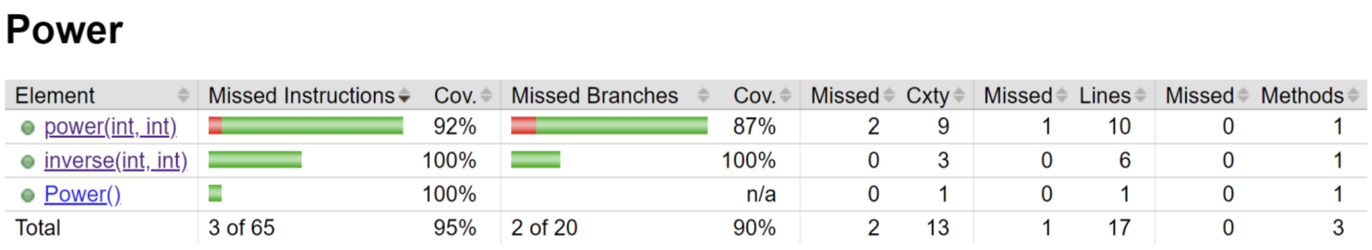


```java
@Test

// To test the inverse function for when right and left are > 0

public void PowTest20() {
  assertEquals(myPow.inverse(left: 3, right: 2), 8);
}

@Test

// To test the inverse function for when right is = 0

public void PowTest21() {
  assertEquals(myPow.inverse(left: 2, right: 0), 1);
}
```

4) **What is the maximum line and branch coverage for each method you have reached? If it is not 100 percent, explain why you were not able to obtain that goal.**

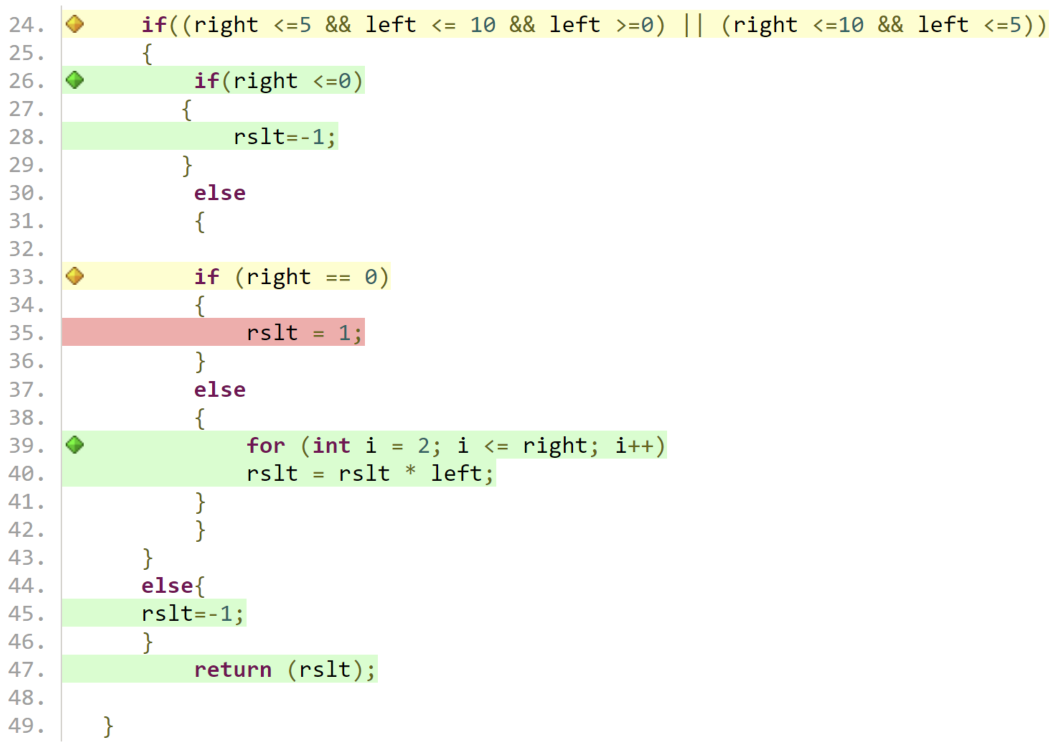
➔ I was able to achieve a 100% instructions and branch coverage for the inverse() function but creating test numbers 20 and 21.

Although, I was not able to achieve a 100% coverage for the power() function. I was only able to cover 92% instructions coverage and 87% branch coverage. This is because, in the power() function, there are 2 "if" statements, that have the same inquires but different results, such as one if statements checks that when right <=0, it should print -1 and also if right == 0, it should print 1. Both of these statements collide with each other, and because of that, even though right is =0 or < 0, it always expects the answer to be 0. This is why, I could not get a coverage of 100%.



Power

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
power(int, int)	🟥🟩	92%	🟥🟩	87%	2	9	1	10	0	1
inverse(int, int)	🟩	100%	🟩	100%	0	3	0	6	0	1
Power()	🟩	100%		n/a	0	1	0	1	0	1
Total	3 of 65	95%	2 of 20	90%	2	13	1	17	0	3



```
        if((right <=5 && left <= 10 && left >=0) || (right <=10 && left <=5))
        {
            if(right <=0)
           {
               rslt=-1;
           }
            else
            {

            if (right == 0)
            {
                rslt = 1;
            }
            else
            {
                for (int i = 2; i <= right; i++)
                rslt = rslt * left;
            }
            }
        }
        else{
        rslt=-1;
        }
            return (rslt);

    }
```

5) **Identify at least two faults not due to numeric overflow in the program. For each, state if they lead to failures (and if so provide a test case). If not, explain why not using the RIPR model.**

➔ The first fault in the program is that it doesn't handle cases when right is negative, but the precondition states that right should be greater than or equal to 0. When right is negative, the program will return -1, but it should throw an exception or return an error message indicating that the input is invalid. This leads to a failure, as the following test case will fail since right is negative.

```
@Test
public void powtest22() {
assertEquals(myPow.power( 2, -1 ),  -1);
}
```

The second fault is that the "else" statement in the first if statement will be executed even when the left and right values are within the specified bounds. This means that the program will return -1 even when the input is valid, which is not the expected answer. This leads to a failure, as the following test case should fail, as the program will return -1, even though the answer should be 8.

```
@Test
public void powtest23() {
assertEquals(myPow.power(2, 3), 8);
}
```

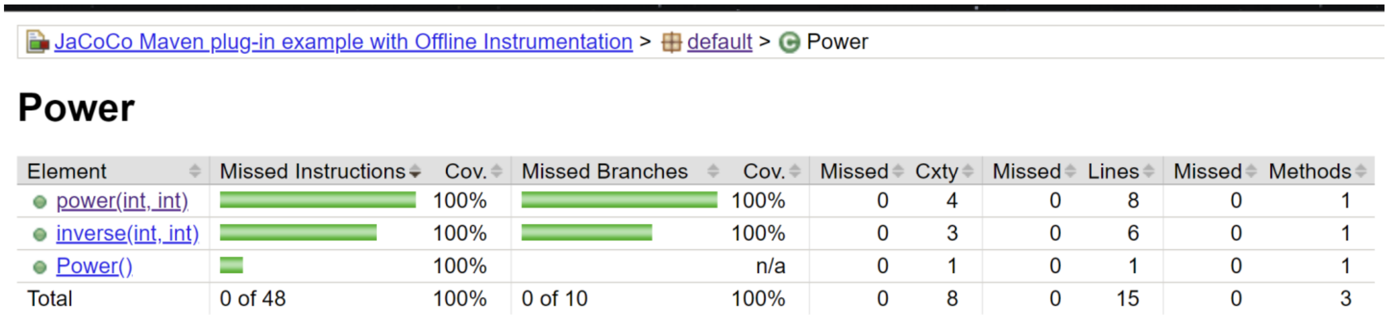
The third fault is the colliding if statements. According to the preconditions, if right is = 0, then the expected answer should be 1 regardless of what left is equal to. Instead, any test case, where right is = 0, the program gives the resulted answer as -1. The following test case will fail, as the expected answer is 1, but result is -1. This causes the test cases to fail. The following if statements:

```
If (right <=0) { rslt = -1; }
If (right == 0) { rslt = 1; }
```

```
@Test
public void powtest24() {
assertEquals(myPow.power(2, 0), 1);
}
```

6) **Try fixing the faults you have found and see if you can increase the code coverage – explain what you discover and state how you fix the faults (provide line numbers and fixes – or screenshots)**

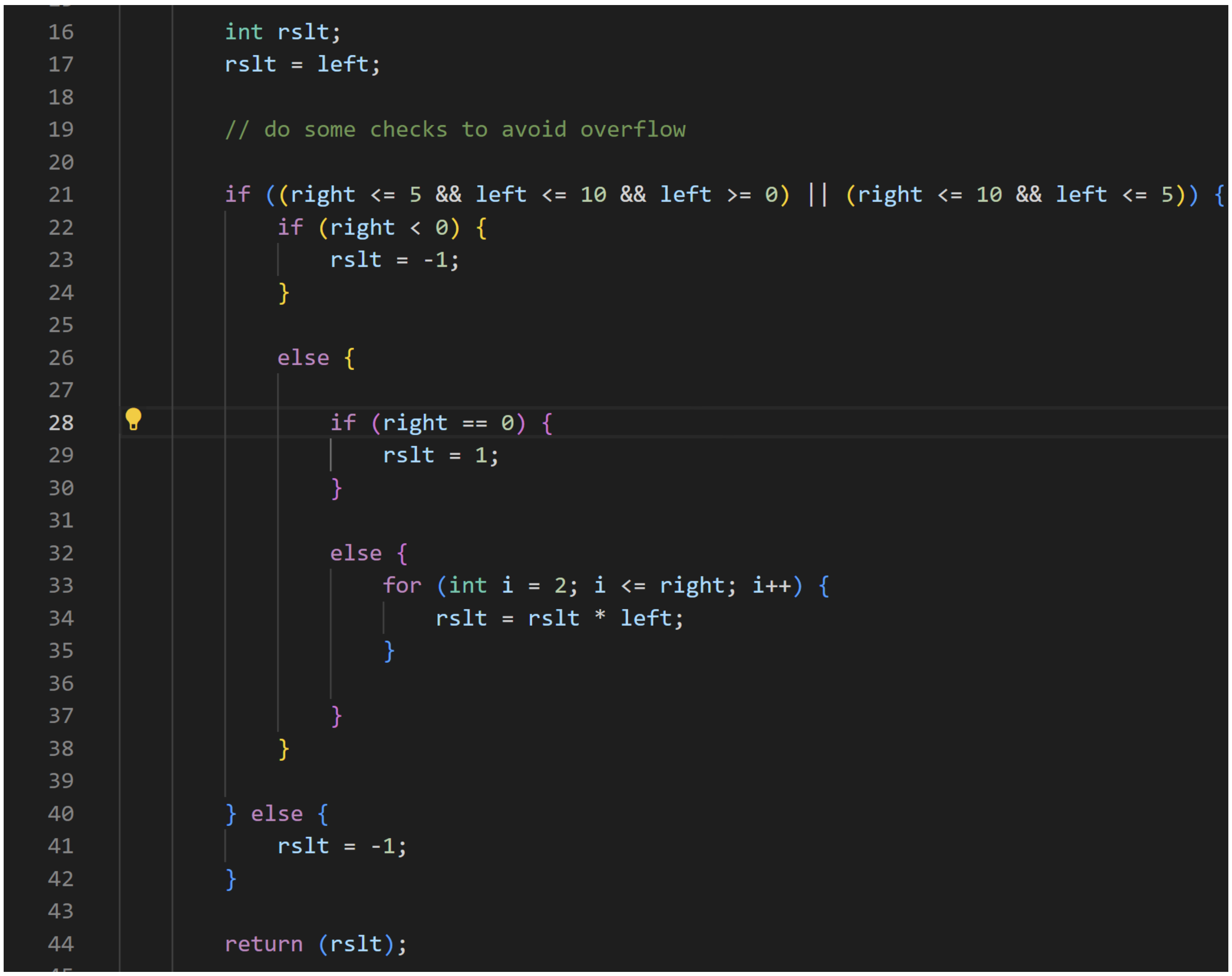
I made changes to the if statements that checked the conditions of the right value. I changed right <= 0 to right < 0, to make sure that the program was checking for right values that were less than 0. Making just these changes, gave me a 100% code coverage for the power() function.



JaCoCo Maven plug-in example with Offline Instrumentation > default > Power

Power

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
power(int, int)		100%		100%	0	4	0	8	0	1
inverse(int, int)		100%		100%	0	3	0	6	0	1
Power()		100%		n/a	0	1	0	1	0	1
Total	0 of 48	100%	0 of 10	100%	0	8	0	15	0	3



```
        int rslt;
        rslt = left;

        // do some checks to avoid overflow

        if ((right <= 5 && left <= 10 && left >= 0) || (right <= 10 && left <= 5)) {
            if (right < 0) {
                rslt = -1;
            }

            else {

                if (right == 0) {
                    rslt = 1;
                }

                else {
                    for (int i = 2; i <= right; i++) {
                        rslt = rslt * left;
                    }

                }
            }

        } else {
            rslt = -1;
        }

        return (rslt);
```

7) **The first method (power) has some checks to prevent numeric overflow, however it misses one case. Find a test case that will cause an overflow (and fail). Does adding this test change the code coverage at all? Discuss why or why not and the implications for testing.**

➔ In the power method, if left = Integer.MAX_VALUE and right = 2, the result of the calculation rslt * left, will exceed the maximum value of an “int” and cause an overflow. This does not cause any changes to code coverage %, as using this test case, the code is still 100% test covered. This is because, I already test cases in place, where left and right were greater than 10, which resulted in -1.If I didn’t have that test case, initializing left to Integer.MAX_VALUE and right to 2, would have given a 100% code coverage.