

# **International Institute of Information Technology, Bangalore**

## **Software Testing Project (CS 731) Control Flow Graph Testing**

Instructor: Prof. Meenakshi D'Souza

Rohan Arora (MT2022091)  
Aakash Gadekar (MT2022002)  
November 26, 2023

# 1. Overview

The objective of this project is to analyze a Number Theory functionalities-based Java source code and perform Control Flow Graph testing on it. We have executed edge coverage and prime path coverage on various special numbers in the domain of number theory.

## 2. Code Description

We have written a program demonstrating the workings of various special numbers such as the Markov Number, Armstrong Number and Ramanujan Number, etc. The functionalities displayed in this project has application in numerous domains ranging from cryptography to graphics.

## 3. Testing Strategy

Control Flow Graph (CFG) testing is an approach that comes under white box testing. This technique determines the order of execution of statements in the program, i.e., how control flows in the graph. CFG offers a graphical representation of this control flow during the program execution. We have performed edge and prime path coverage on the CFGs for the application functionalities.

Our source code consists of decision statements, nested loops as well as a combination of both to thoroughly illustrate the usage of prime paths and explore all possible pathways over the nested loops and decision statements.

We have done tests on the designed test cases using JUnit. JUnit is a testing framework for Java and is used for unit testing.

We have also utilized a code coverage library named JaCoCo (Java Code Coverage) to check the percentage of line coverage achieved using our test cases.

**Edge Coverage:** Each edge in the graph should be traversed at least once to achieve edge coverage.

**Prime Path Coverage:** A prime path is a maximal simple path. Each prime path in the graph should be traversed at least once to achieve prime path coverage.

## 4. Testing Functionalities

### i. Functionality Name: Goldbach Number

**Description/Formula:** An even and positive number is called a Goldbach number if we can express the number as the sum of two odd prime numbers.

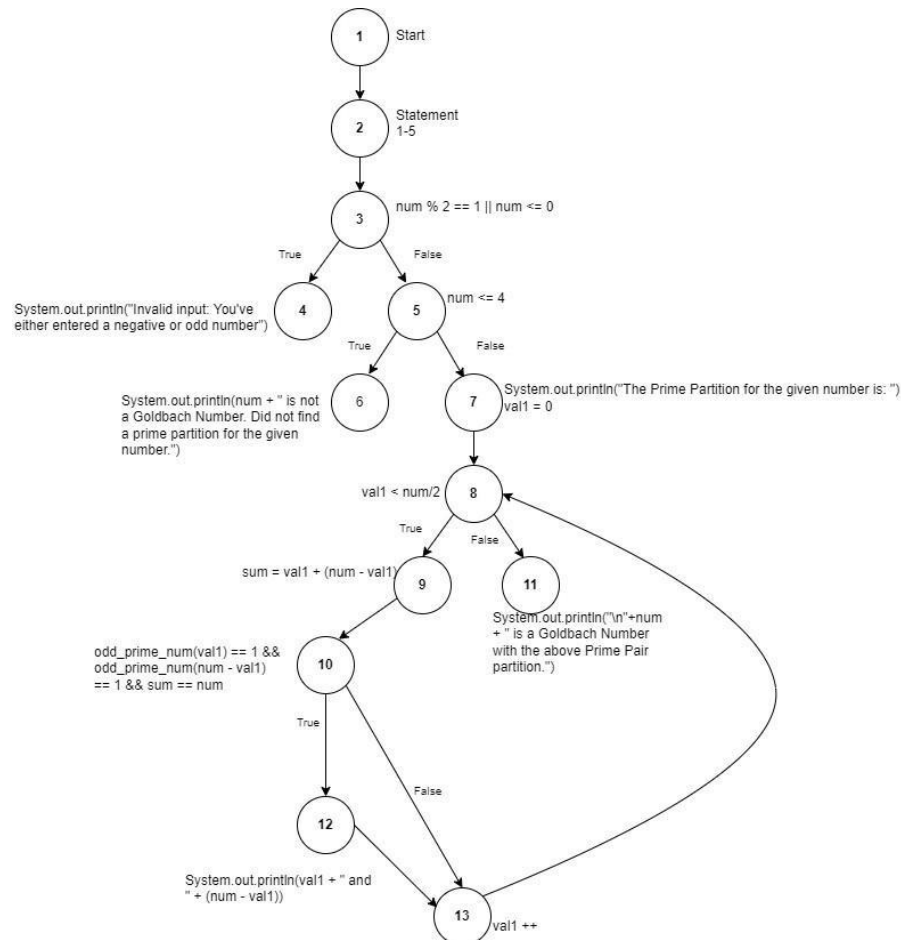
The expression of the two odd primes sum is called a Goldbach partition of that number.

Examples of Goldbach partitions:

$$6 = 3 + 3$$

$$10 = 3 + 7 = 5 + 5$$

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 5], [5, 6], [5, 7], [7, 8], [8, 9], [8, 11], [9, 10], [10, 12], [10, 13], [12, 13], [13, 8]

**Prime Paths:** [1, 2, 3, 4], [1, 2, 3, 5, 6], [8, 9, 10, 13, 8], [9, 10, 13, 8, 9], [9, 10, 13, 8, 11], [10, 13, 8, 9, 10], [13, 8, 9, 10, 13], [8, 9, 10, 12, 13, 8], [9, 10, 12, 13, 8, 9], [9, 10, 12, 13, 8, 11], [10, 12, 13, 8, 9, 10], [12, 13, 8, 9, 10, 12], [13, 8, 9, 10, 12, 13], [1, 2, 3, 5, 7, 8, 11], [1, 2, 3, 5, 7, 8, 9, 10, 13], [1, 2, 3, 5, 7, 8, 9, 10, 12, 13]

**Test Case:**

```
@Test
public void testGoldbach_Num_1() throws Exception {
    assertEquals("expected: true,num_theoryUnderTest.Goldbach_Num(80));", true, num_theoryUnderTest.Goldbach_Num(80));
}

@Test
public void testGoldbach_Num_2() throws Exception {
    assertEquals("expected: false,num_theoryUnderTest.Goldbach_Num(-5));", false, num_theoryUnderTest.Goldbach_Num(-5));
}

@Test
public void testGoldbach_Num_3() throws Exception {
    assertEquals("expected: false,num_theoryUnderTest.Goldbach_Num(2));", false, num_theoryUnderTest.Goldbach_Num(2));
}
```

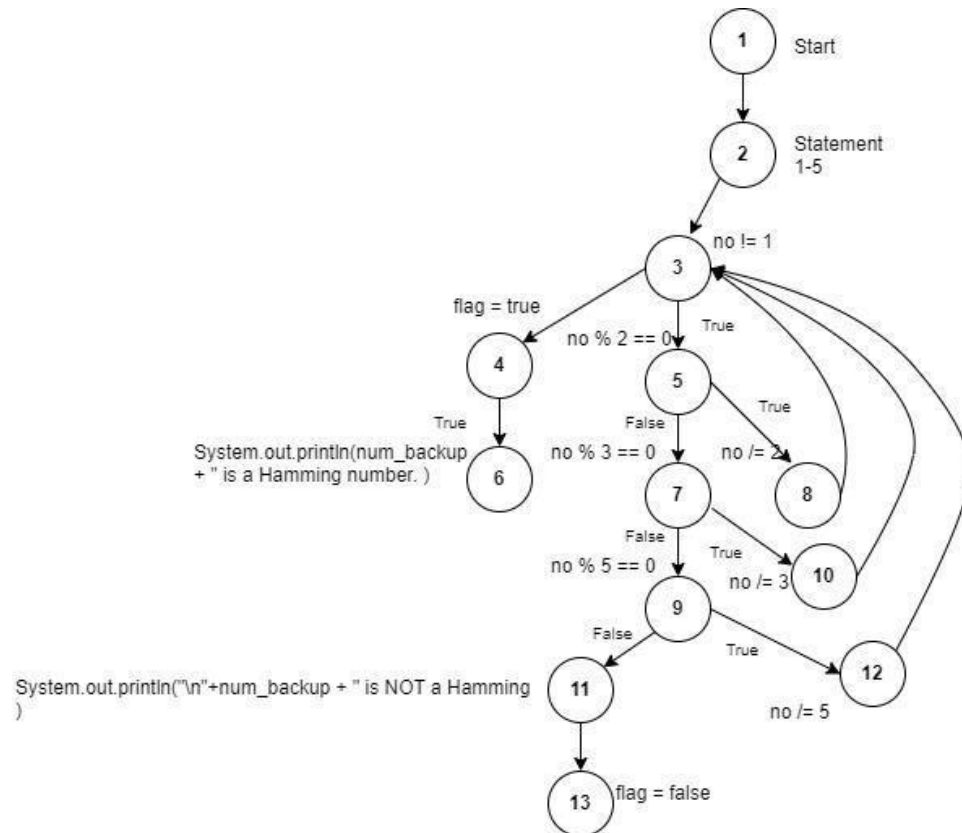
## ii. Functionality Name: Hamming Number

**Description/Formula:** Hamming numbers are numbers whose only prime factors are 2, 3 or 5.

$$H = 2^i \times 3^j \times 5^k \text{ where } i, j, k \geq 0$$

Examples: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15

## Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 5], [4, 6], [5, 7], [5, 8], [7, 9], [7, 10], ....

**Prime Paths:** [1, 2, 3, 4], [1, 2, 3, 5, 6], [8, 9, 10, 13, 8], [9, 10, 13, 8, 9], [9, 10, 13, 8, 11], [10, 13, 8, 9, 10], [13, 8, 9, 10, 13], [8, 9, 10, 12, 13, 8], [9, 10, 12, 13, 8, 9], [9, 10, 12, 13, 8, 11], [10, 12, 13, 8, 9, 10], [12, 13, 8, 9, 10, 12], [13, 8, 9, 10, 12, 13], [1, 2, 3, 5, 7, 8, 11], [1, 2, 3, 5, 7, 8, 9, 10, 13], [1, 2, 3, 5, 7, 8, 9, 10, 12, 13]

## Test Case:

```
@Test
public void testHamming_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Hamming_Num( no: 900));
}

@Test
public void testHamming_Num_2() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Hamming_Num( no: 33));
}

@Test
public void testHamming_Num_3() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Hamming_Num( no: 55));
}

@Test
public void testHamming_Num_4() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Hamming_Num( no: 86));
}
```

### iii. Functionality Name: Markov Number

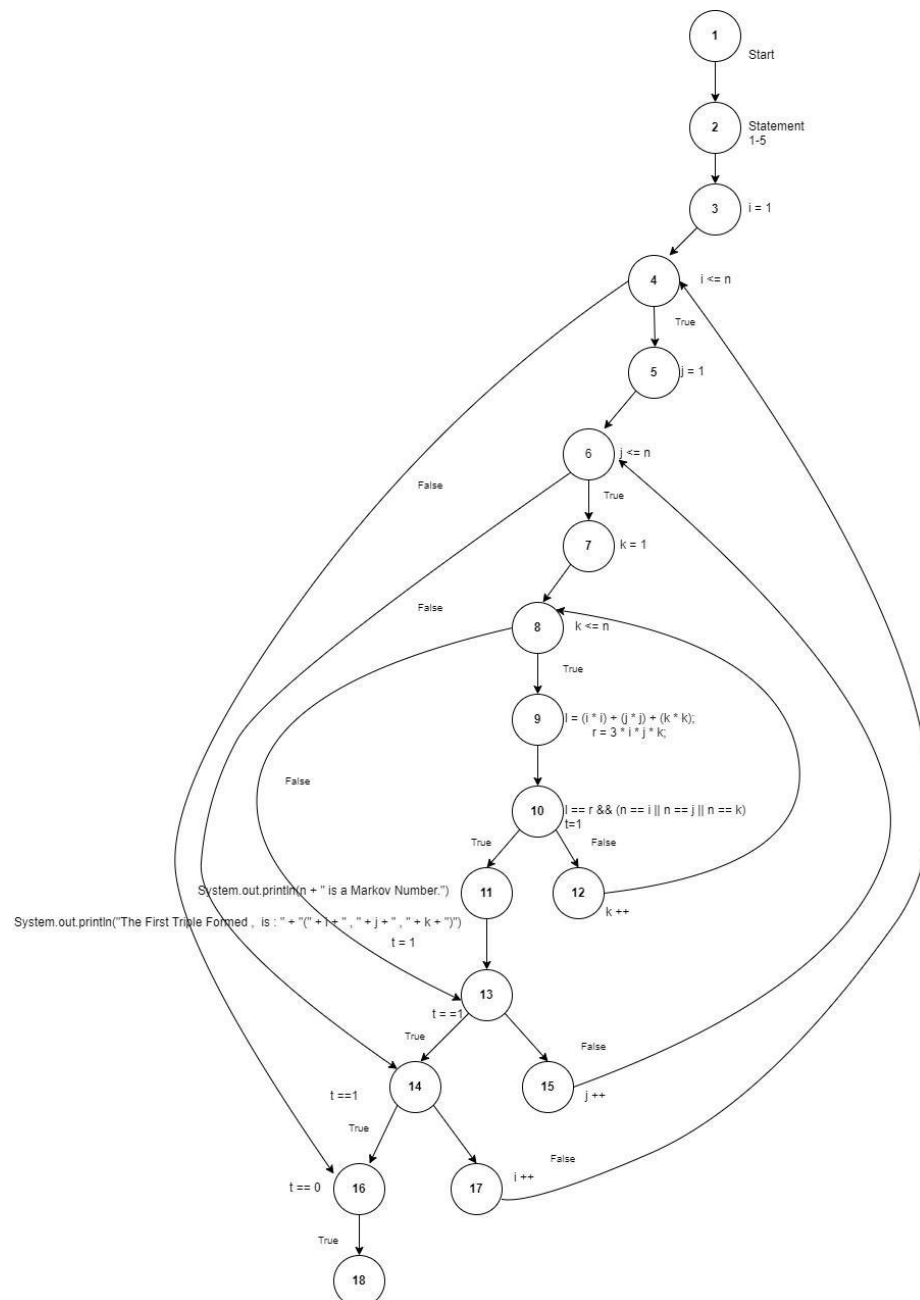
**Description/Formula:** Markov number are the positive integers x, y, z that appear in the solution of the Markov Diophantine equation:

$$x^2 + y^2 + z^2 = 3xyz$$

**Example:** 1, 2, 5, 13 appearing as solution of the Markov triplets:

(1, 1, 1), (1, 1, 2), (1, 2, 5), (1, 5, 13)

**Control Flow Graph:**



**Edges:** [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [14, 17], [14, 16], ...

**Prime Paths:** [8, 13, 15, 6, 7, 8], [13, 15, 6, 7, 8, 13], [14, 17, 4, 5, 6, 14], [15, 6, 7, 8, 13, 15], [9, 10, 12, 8, 13, 15, 6, 7], [15, 6, 7, 8, 13, 14, 16, 18], ...

**Test Case:**

```
@Test
public void testMarkov_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Markov_Num(89));
}

@Test
public void testMarkov_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Markov_Num(65));
}

@Test
public void testMarkov_Num_3() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Markov_Num(169));
}
```

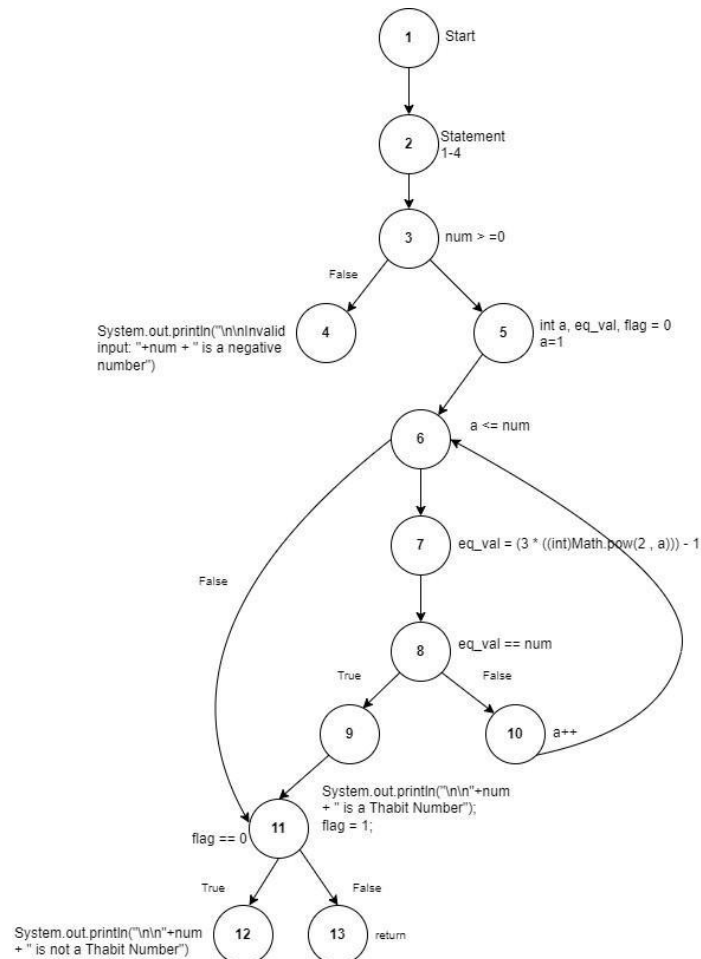


#### iv. Functionality Name: Thabit Number

**Description/Formula:** Thabit Number is an integer of form:  $3 * 2^n - 1$ ,  $n$  = non-negative integer.

Examples: 2, 5, 47, 95

#### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 5], [5, 6], [11, 13], [7, 8], ...

**Prime Paths:** [1, 2, 3, 4], [6, 7, 8, 10, 6], [8, 10, 6, 7, 8], [1, 2, 3, 5, 6, 7, 8, 10], [1, 2, 3, 5, 6, 7, 8, 9, 11, 13] ...

## Test Case:

```
@Test
public void testThabit_Num_1() throws Exception {
    assertEquals("expected: false", num_theoryUnderTest.Thabit_Num(8));
}

@Test
public void testThabit_Num_2() throws Exception {
    assertEquals("expected: false", num_theoryUnderTest.Thabit_Num(-5));
}

@Test
public void testThabit_Num_3() throws Exception {
    assertEquals("expected: true", num_theoryUnderTest.Thabit_Num(11));
}

@Test
public void testThabit_Num_4() throws Exception {
    assertEquals("expected: true", num_theoryUnderTest.Thabit_Num(47));
}

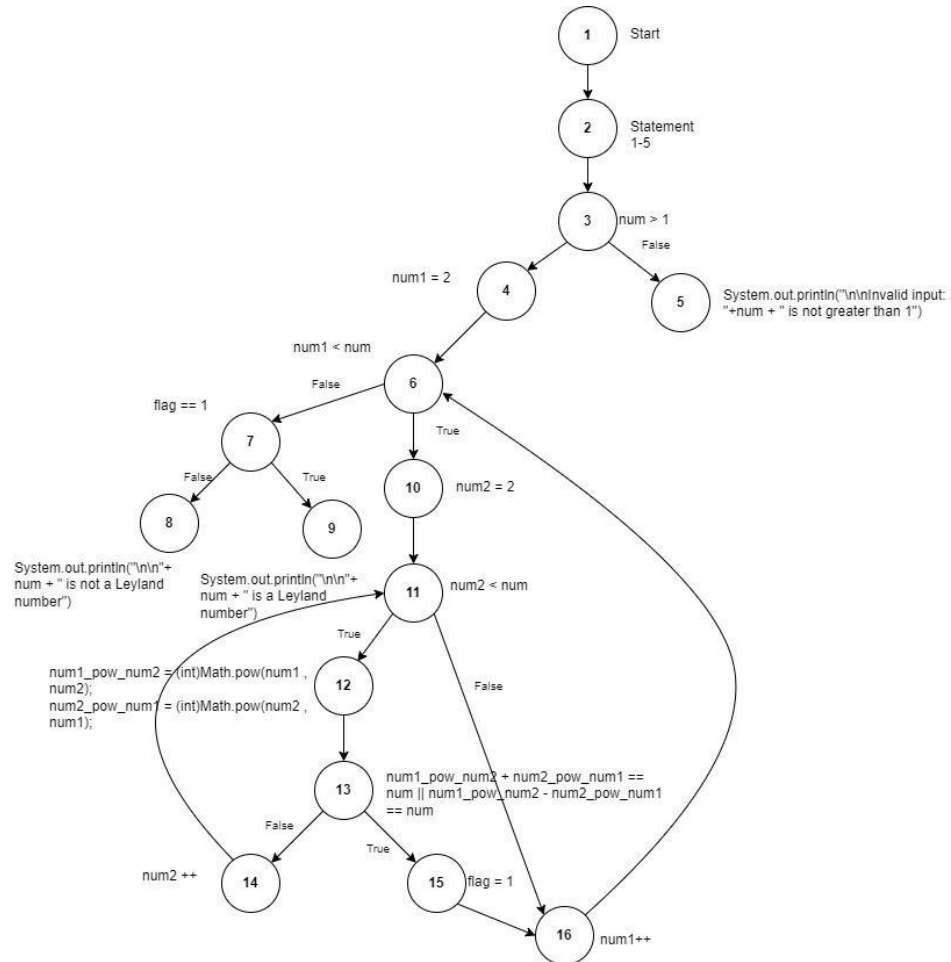
@Test
public void testThabit_Num_5() throws Exception {
    assertEquals("expected: false", num_theoryUnderTest.Thabit_Num(6));
}
```

## v. Functionality Name: Leyland Number

**Description/Formula:** Leyland number 'n' is a number where  $n = x^y + y^x$ ,  $x, y =$  integers greater than 1.

Examples: 8, 57, 100, 177

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 5], [4, 6], [6, 7], [7, 8], [7, 9], [10, 11], ...

**Prime Paths:** [1, 2, 3, 5], [6, 10, 11, 16, 6], [10, 11, 16, 6, 10], [12, 13, 14, 11, 16, 6, 10], [1, 2, 3, 4, 6, 10, 11, 16], ...

## Test Case:

```
@Test
public void testLeyland_Num_1() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Leyland_Num(2));
}

@Test
public void testLeyland_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Leyland_Num(-3));
}

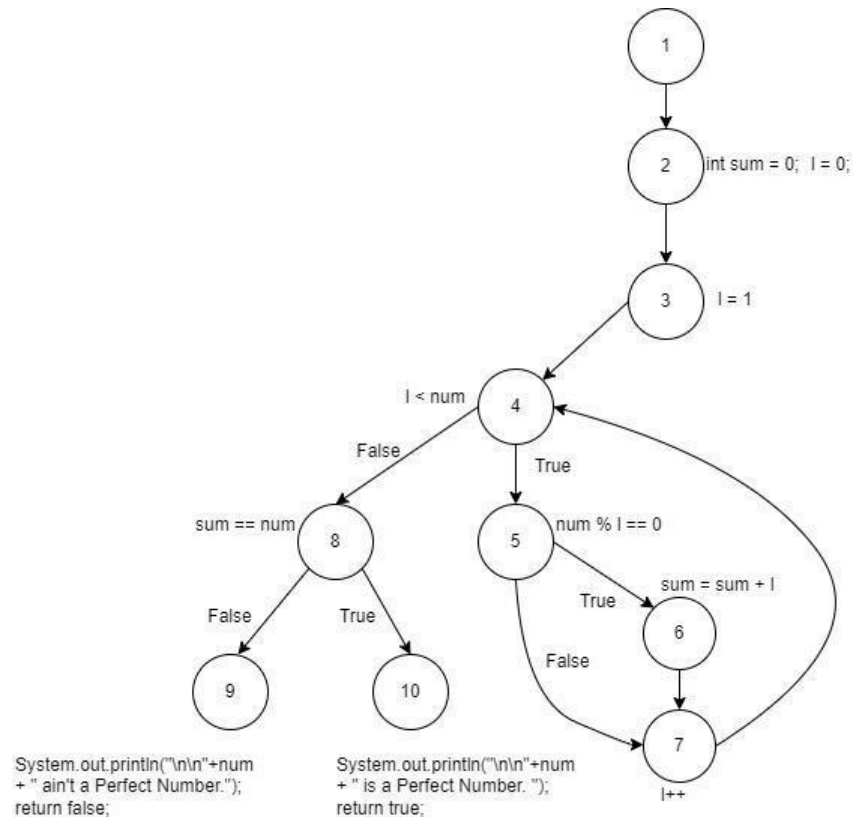
@Test
public void testLeyland_Num_3() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Leyland_Num(145));
}
```

## vi. Functionality Name: Perfect Number

**Description/Formula:** The summation of all factors of a given number except itself is equal to the number, then the number is called a Perfect Number.

Examples: 6, 28, 496

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [4, 5], [4, 8], [6, 7], [7, 4], [8, 9], ...

**Prime Paths:** [7, 4, 5, 7], [4, 5, 6, 7, 4], [5, 6, 7, 4, 5], [1, 2, 3, 4, 5, 6, 7], ...

## Test Case:

```
@Test
public void testPerfect_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Perfect_Num(496));
}

@Test
public void testPerfect_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Perfect_Num(552));
}
```

## vii. Functionality Name: Sociable Number

**Description/Formula:** Sociable numbers are numbers whose Aliquot sums form a periodic sequence.

Examples: 1264460

Proper divisors sum of 1264460 is =

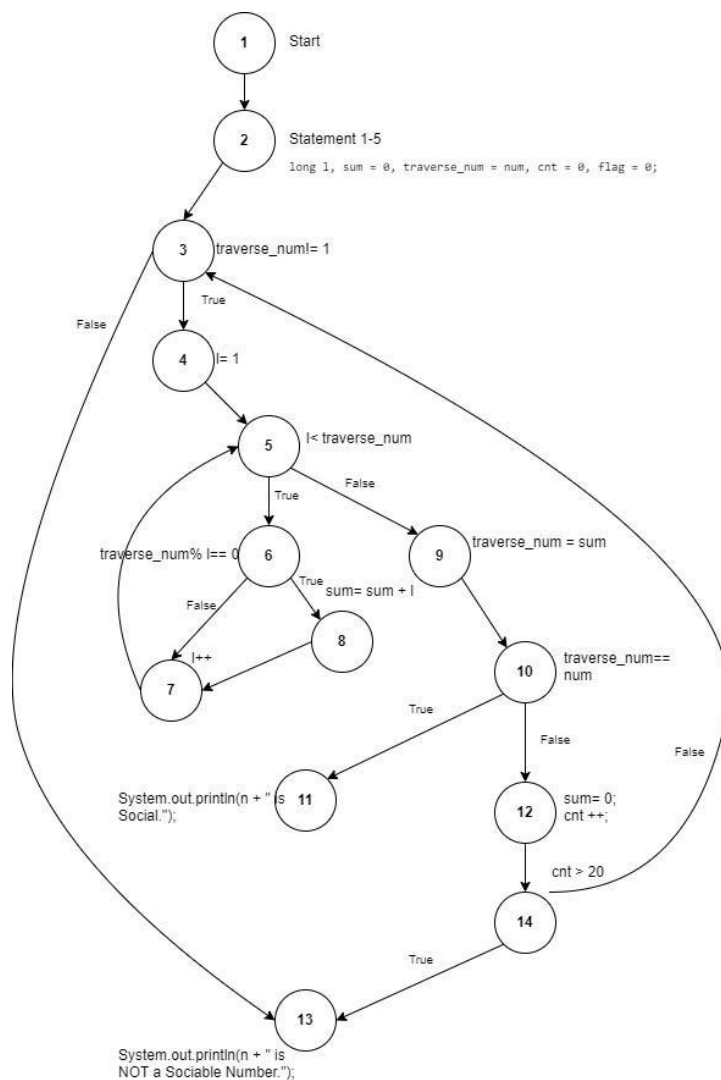
1547860 Proper divisors sum of 1547860 is

= 1727636 Proper divisors sum of 1727636

is = 1305184 Proper divisors sum of

1305184 is = 1264460

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 13], [4, 5], [6, 7], [7, 5], [8, 7], ...

**Prime Paths:** [6, 8, 7, 5, 6], [7, 5, 6, 8, 7], [8, 7, 5, 6, 8], [6, 7, 5, 9, 10, 11], [4, 5, 9, 10, 12, 14, 3, 13], ...

## Test Case

```
@Test
public void testSociable_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Sociable_Num(14288));
}

@Test
public void testSociable_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Sociable_Num(13));
}

@Test
public void testSociable_Num_3() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Sociable_Num(46661));
}

@Test
public void testSociable_Num_4() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Sociable_Num(1));
}
```

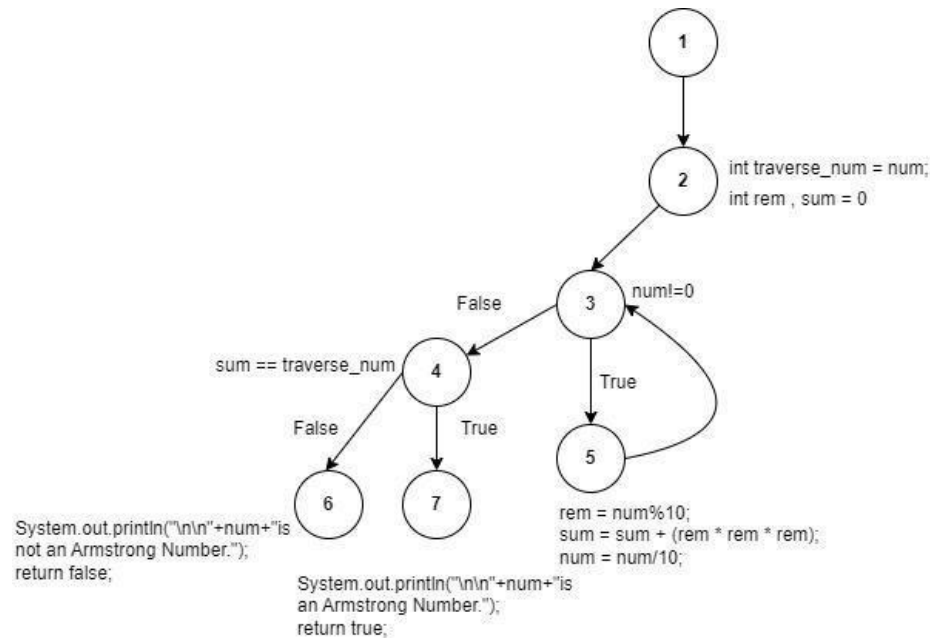


### viii. Functionality Name: Armstrong Number

**Description/Formula:** Armstrong number is a number that when expressed as the sum of cubes of the digits of a given number is equal to the number itself.

Examples: 371, 125

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 5], [4, 6], [4, 7], [5, 3]

**Prime Paths:** [3, 5, 3], [5, 3, 5], [1, 2, 3, 5], [5, 3, 4, 6], [5, 3, 4, 7], [1, 2, 3, 4, 6], [1, 2, 3, 4, 7]

### Test Case:

```
@Test
public void testArmstrong_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Armstrong_Num(371));
}

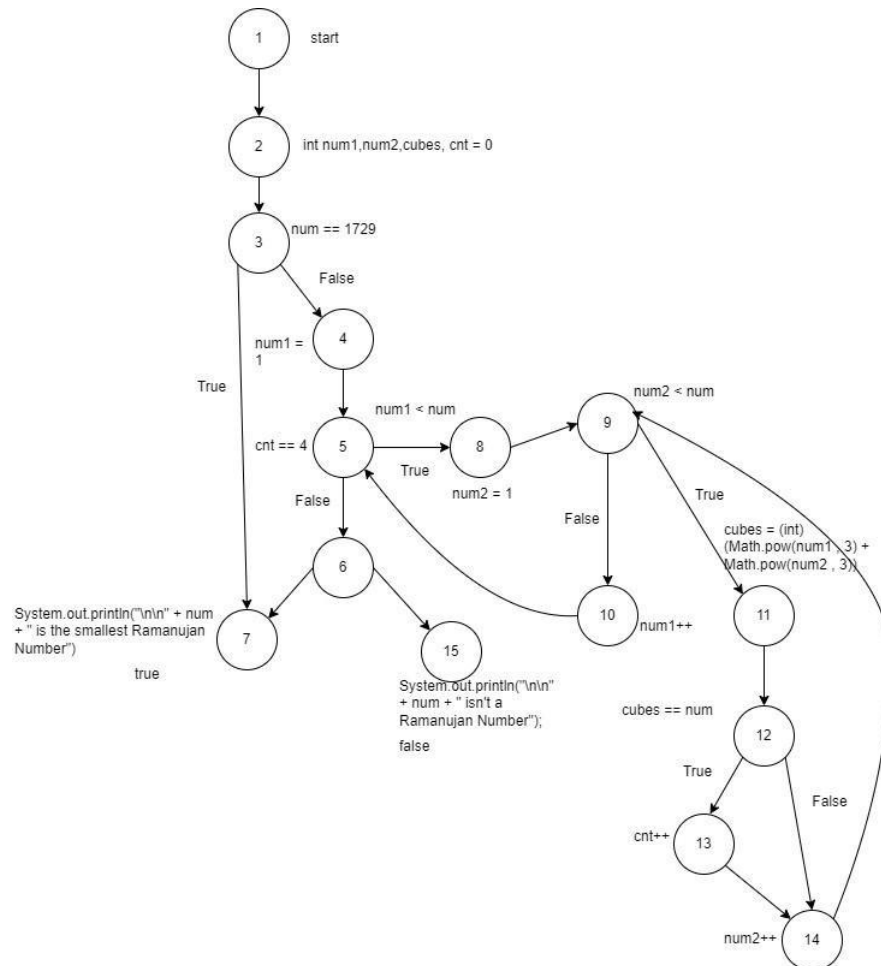
@Test
public void testArmstrong_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Armstrong_Num(125));
}
```

## ix. Functionality Name: Ramanujan Number

**Description/Formula:** A number that can be expressed as the summation of two positive cubes in at least two ways is called a Ramanujan Number.

Examples: 4104, 1729, 3529

### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 7], [5, 6], [5, 8], [8, 9], [9, 11], [9, 10], [10, 5], [12, 13], ...

**Prime Paths:** [9, 11, 12, 14, 9], [10, 5, 8, 9, 10], [12, 13, 14, 9, 11, 12], [13, 14, 9, 11, 12, 13], ...

## Test Case:

```
@Test
public void testRamanujan_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Ramanujan_Num(4104));
}

@Test
public void testRamanujan_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Ramanujan_Num(3529));
}

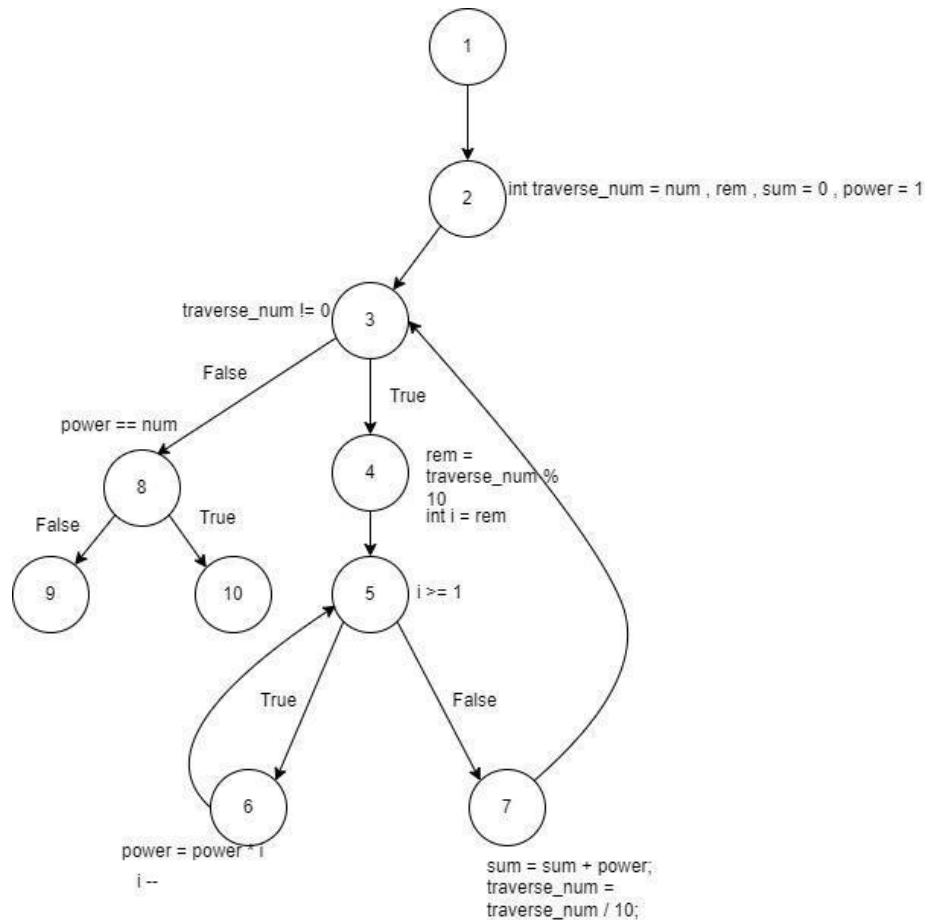
@Test
public void testRamanujan_Num_3() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Ramanujan_Num(1729));
}
```

#### x. Functionality Name: Strong Number

**Description/Formula:** A number whose sum of factorials is equal to the number itself, is called a strong number.

Examples: 145, 543

#### Control Flow Graph:



**Edges:** [1, 2], [2, 3], [3, 4], [3, 8], [5, 6], [5, 7], [8, 9], [6, 5], [7, 3], [12, 13], [4, 5], ...

**Prime Paths:** [7, 3, 4, 5, 7], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 7], [4, 5, 7, 3, 8, 9], [4, 5, 7, 3, 8, 10], ...

## Test Case:

```
@Test
public void testStrong_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Strong_Num(145));
}

@Test
public void testStrong_Num_2() throws Exception {
    assertEquals( expected: false, num_theoryUnderTest.Strong_Num(543));
}
```

## 5. Results

We have tested functionalities using JUnit and JaCoCo. JaCoCo (Java Code Coverage) has been used to check the percentage of line coverage achieved using our test cases. We have written a total of 49 test cases for the functionalities and the results are as shown below:

TestingProj

### TestingProj

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		84%		83%	28	112	78	331	1	19	0	1
Total	226 of 1,477	84%	28 of 171	83%	28	112	78	331	1	19	0	1

Let's break down the provided metrics:

**Total Lines:** There are a total of 1,477 lines of code in your project.

**Coverage for Lines:** The tests cover 84% of the lines of code, which means that 84% of your code has associated test cases.

**Missed Lines:** There are 226 lines of code that are not covered by tests.





































**Total Branches:** Your code has 171 branches (decision points where the control flow can take different paths).

**Coverage for Branches:** The tests cover 83% of the branches, meaning that 83% of your code's decision points are exercised by your tests.

This screenshot shows the percentage of instruction coverage for each of the 15 functionalities.

TestingProj > default > Num\_Theory

## Num\_Theory

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0%		0%	17	17	73	73	1	1
Sociable_Num(int)		96%		83%	2	7	2	20	0	1
Markov_Num(int)		98%		85%	3	11	1	21	0	1
Woodall_Num(int)		97%		87%	1	5	1	16	0	1
Hamming_Num(int)		97%		90%	1	6	1	18	0	1
Emirp_Num(int)		100%		93%	1	9	0	19	0	1
Goldbach_Num(int)		100%		85%	2	8	0	16	0	1
Evil_Num(int)		100%		100%	0	5	0	21	0	1
Leyland_Num(int)		100%		91%	1	7	0	18	0	1
Ramanujan_Num(int)		100%		100%	0	6	0	16	0	1
Thabit_Num(int)		100%		100%	0	5	0	16	0	1
Colombian_Num(int)		100%		100%	0	5	0	18	0	1
Strong_Num(int)		100%		100%	0	4	0	15	0	1
Narcissistic_Num(int)		100%		100%	0	3	0	11	0	1
Armstrong_Num(int)		100%		100%	0	3	0	13	0	1
Perfect_Num(int)		100%		100%	0	4	0	11	0	1
odd_prime_num(int)		100%		100%	0	5	0	7	0	1
static{...}		100%		n/a	0	1	0	1	0	1
Num_Theory()		100%		n/a	0	1	0	1	0	1
Total	226 of 1,477	84%	28 of 171	83%	28	112	78	331	1	19

## 6. Conclusion

Num\_TheoryTest: 46 total, 46 passed

1.97 s

[Collapse](#) | [Expand](#)

Num_TheoryTest.testisWoodallNumber_1	passed	4 ms
Num_TheoryTest.testisWoodallNumber_2	passed	0 ms
Num_TheoryTest.testisWoodallNumber_3	passed	1 ms
Num_TheoryTest.testisWoodallNumber_4	passed	0 ms
Num_TheoryTest.testisWoodallNumber_5	passed	0 ms
Num_TheoryTest.testisLeylandNumber_1	passed	0 ms
Num_TheoryTest.testisLeylandNumber_2	passed	0 ms
Num_TheoryTest.testisLeylandNumber_3	passed	4 ms
Num_TheoryTest.testoddPrimeNumber_1	passed	0 ms
Num_TheoryTest.testoddPrimeNumber_2	passed	0 ms
Num_TheoryTest.testisArmstrongNumber_1	passed	0 ms
Num_TheoryTest.testisArmstrongNumber_2	passed	0 ms
Num_TheoryTest.testisPerfectNumber_1	passed	0 ms
Num_TheoryTest.testisPerfectNumber_2	passed	0 ms
Num_TheoryTest.testisThabitNumber_1	passed	0 ms
Num_TheoryTest.testisThabitNumber_2	passed	0 ms
Num_TheoryTest.testisThabitNumber_3	passed	0 ms
Num_TheoryTest.testisThabitNumber_4	passed	0 ms
Num_TheoryTest.testisThabitNumber_5	passed	1 ms
Num_TheoryTest.testisThabitNumber_6	passed	0 ms

In this software testing project, we have performed control flow graph testing on a number theory validator and sequence generator application. We have tested 15 functionalities using JUnit and JaCoCo. The project helped us analyze the program's underlying control flow structure and write test requirements for performing edge and prime path coverages. We devised the test paths to cover all test requirements and designed test cases corresponding to them. We were able to write 49 test cases to test the application's control flow.