International Institute of Information Technology, Bangalore

Software Testing Project (CS 731) Control Flow Graph Testing

Instructor: Prof. Meenakshi D’Souza

Ankita Dutta (MT2021017)
Richa Varma (MT2021107)

November 28, 2022

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, Aliquot Sequence, and Ramanujan Number. 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.

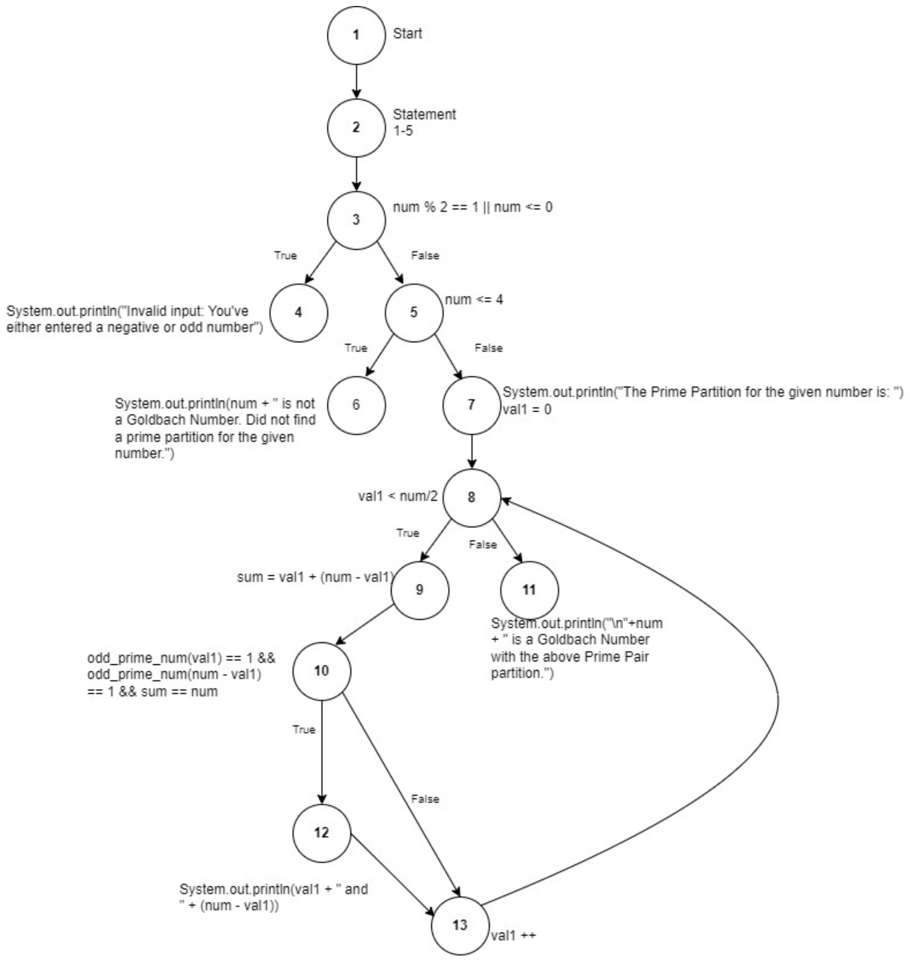
The expression of the two odd primes sum is called a Goldbach partition of that number.

Examples of Goldbach partitions:

6 = 3 + 3

10 = 3 + 7 = 5 + 5

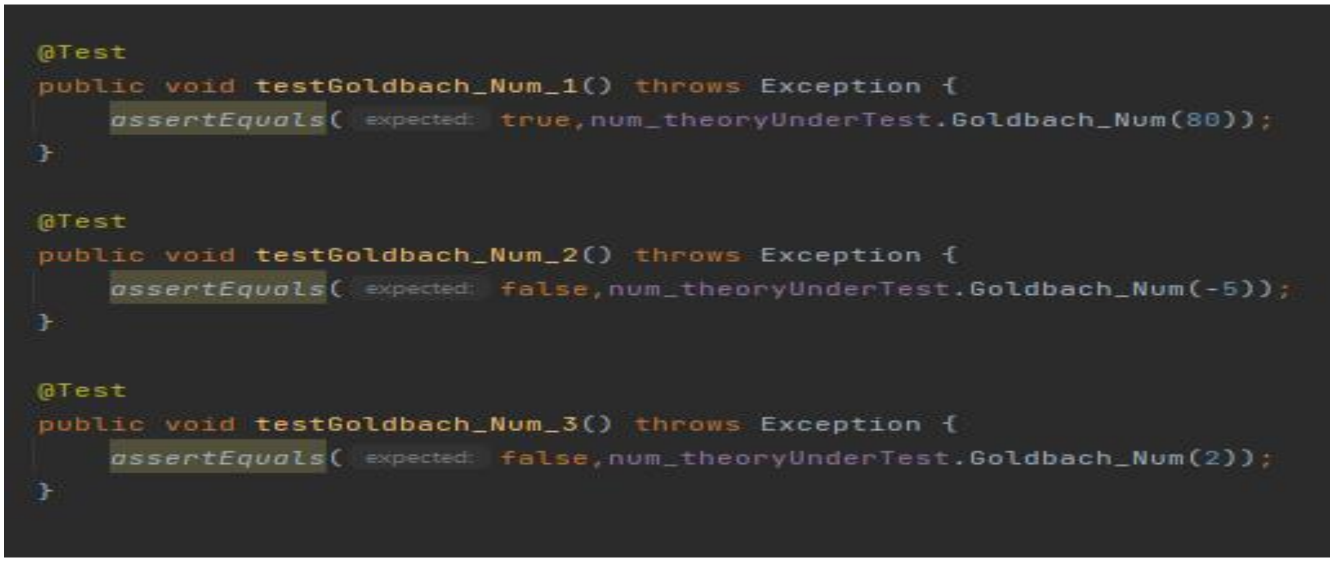
Control Flow Graph:


1
Start
2
Statement 1-5
3
num % 2 == 1 || num <= 0
True
False
4
System.out.println("Invalid input: You've either entered a negative or odd number")
5
num <= 4
True
False
6
System.out.println(num + " is not a Goldbach Number. Did not find a prime partition for the given number.")
7
System.out.println("The Prime Partition for the given number is: ")
val1 = 0
8
val1 < num/2
True
False
9
sum = val1 + (num - val1)
11
System.out.println("\n"+num + " is a Goldbach Number with the above Prime Pair partition.")
10
odd_prime_num(val1) == 1 && odd_prime_num(num - val1) == 1 && sum == num
True
False
12
System.out.println(val1 + " and " + (num - val1))
13
val1 ++

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));
}

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

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

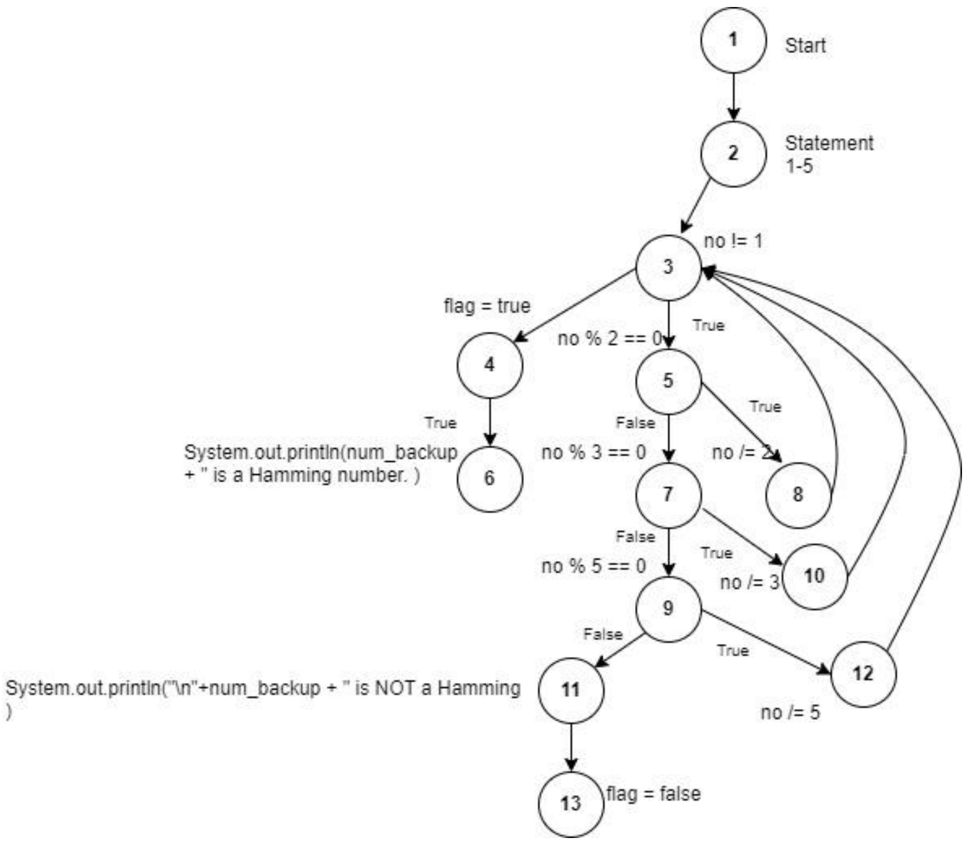
ii. Functionality Name: Hamming Number

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

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

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

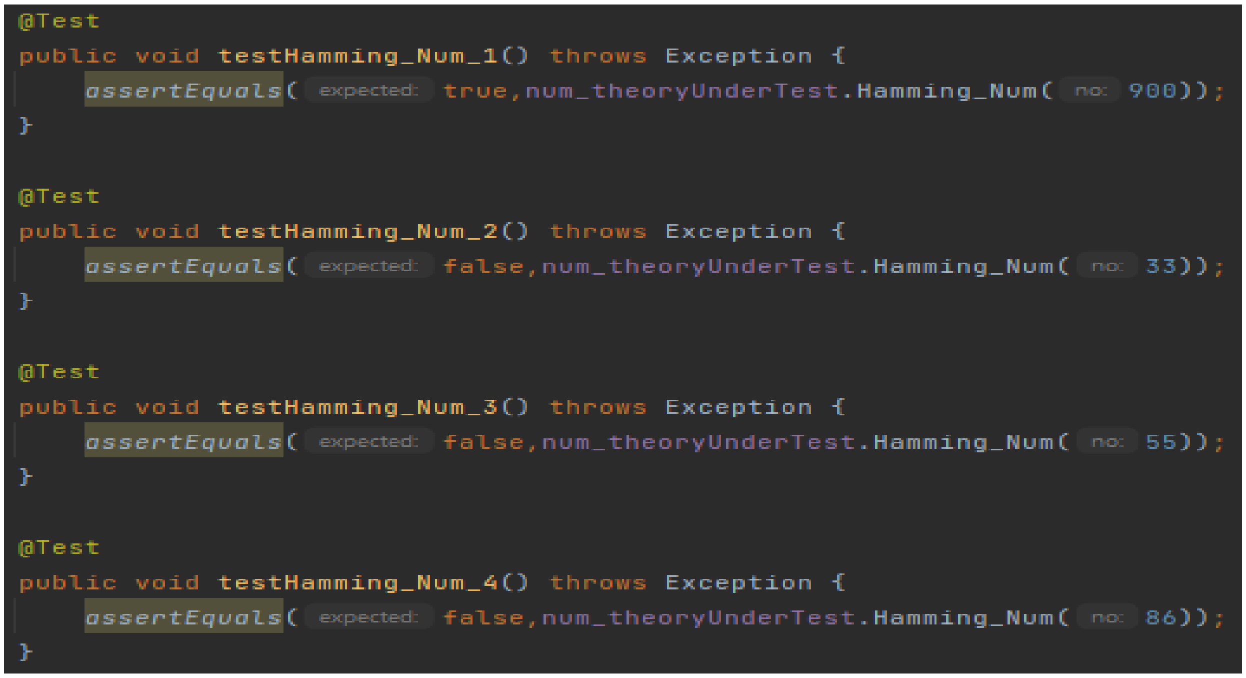
Control Flow Graph:



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:



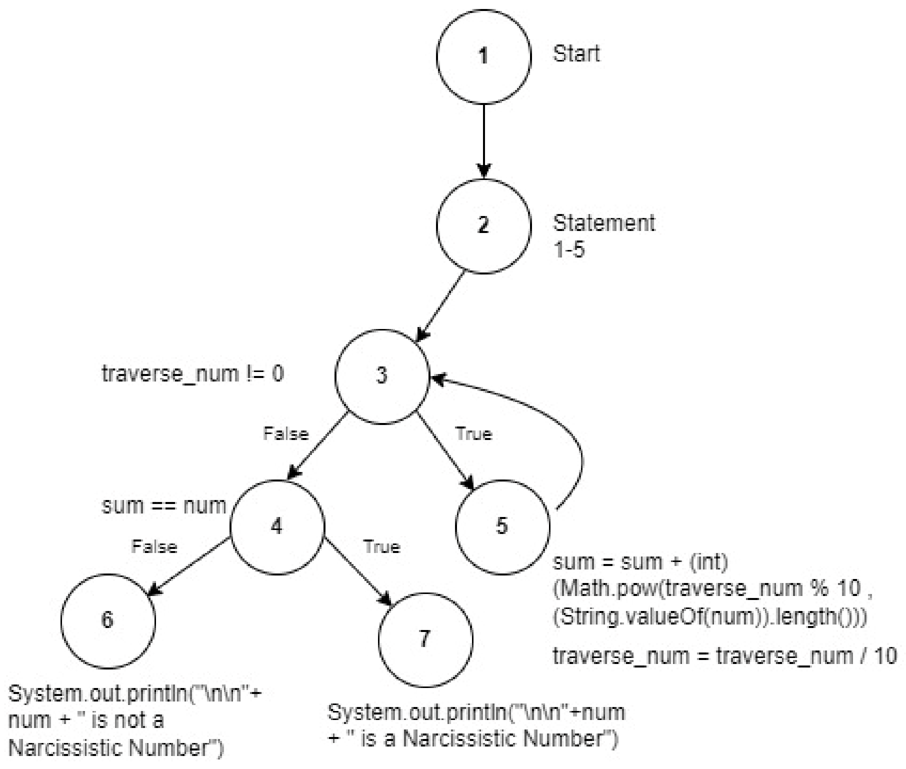
```
@Test
public void testHamming_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Hamming_Num( no: 900));
}

@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: Narcissistic Number

Description/Formula: Narcissistic number is a number that is the summation of its digits each raised to the power of the number of digits. It is also called Pluperfect Digital Invariant (PPDI).

Examples: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474

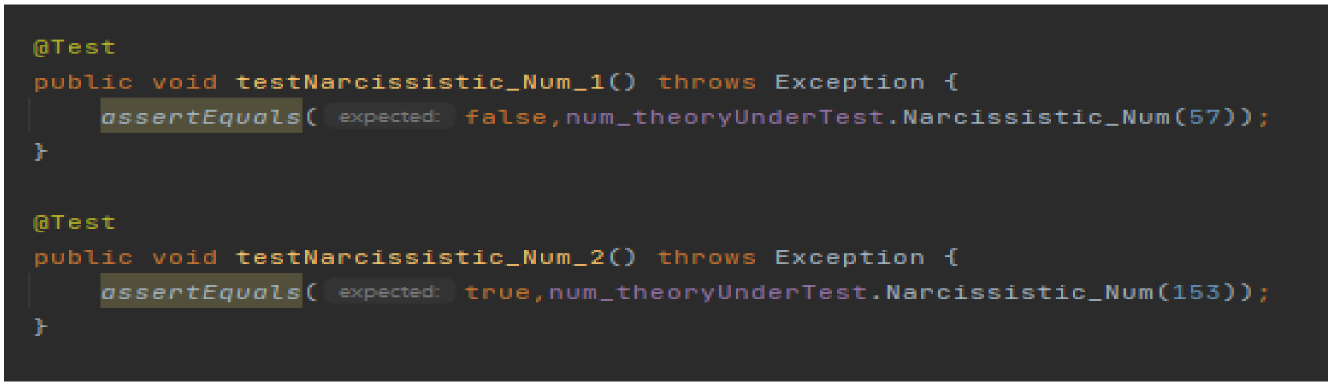
Control Flow Graph:


1
Start
2
Statement 1-5
3
traverse_num != 0
False
True
4
sum == num
False
True
5
sum = sum + (int) (Math.pow(traverse_num % 10 , (String.valueOf(num)).length()))
traverse_num = traverse_num / 10
6
System.out.println("\n\n"+ num + " is not a Narcissistic Number")
7
System.out.println("\n\n"+num + " is a Narcissistic Number")

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

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 testNarcissistic_Num_1() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Narcissistic_Num(57));
}

@Test
public void testNarcissistic_Num_2() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Narcissistic_Num(153));
}
```

iv. Functionality Name: Markov Number

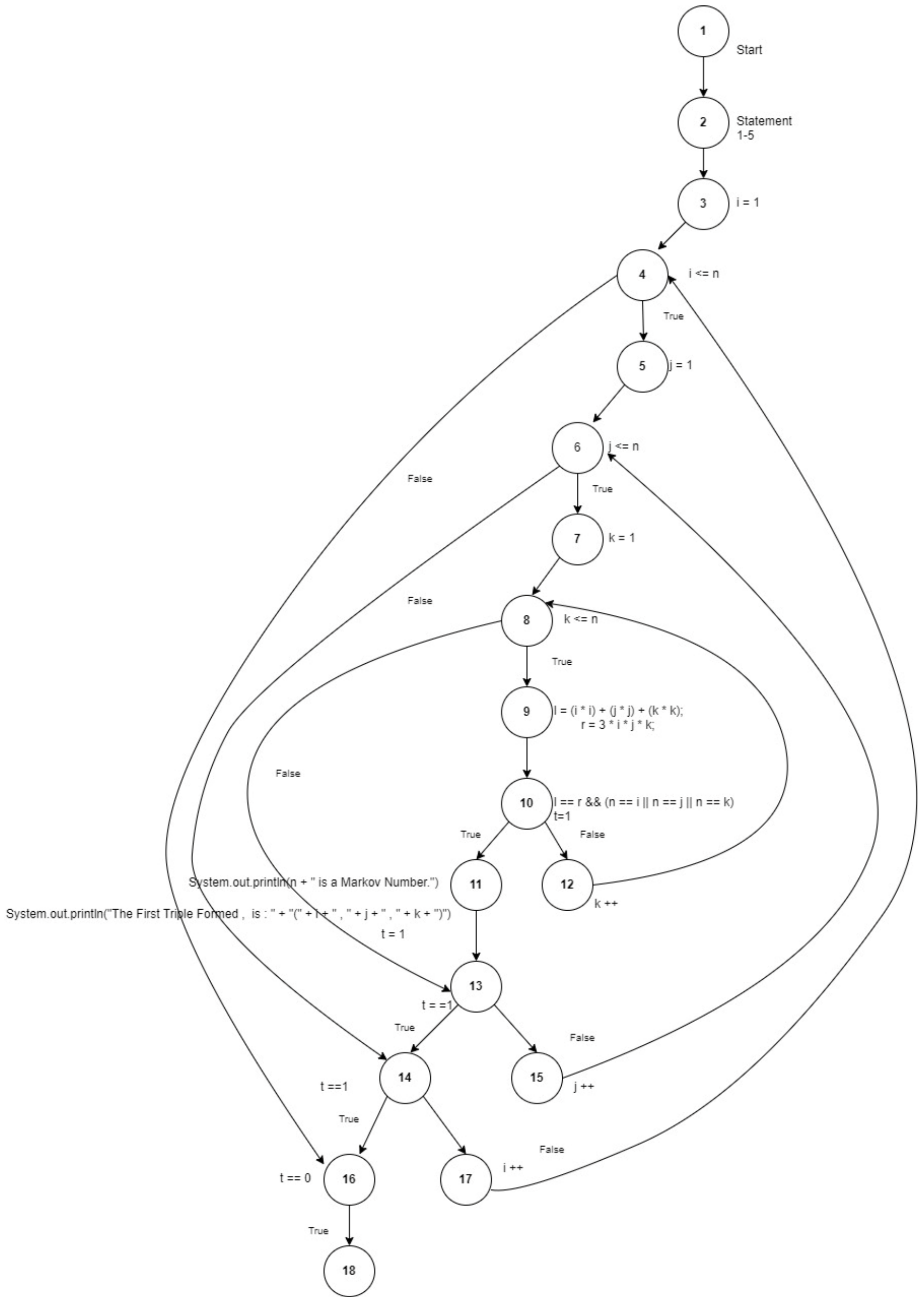
Description/Formula: Markov number are the positive integers x, y, z that appear in the solution of the Markov Diophantine equation:

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

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:

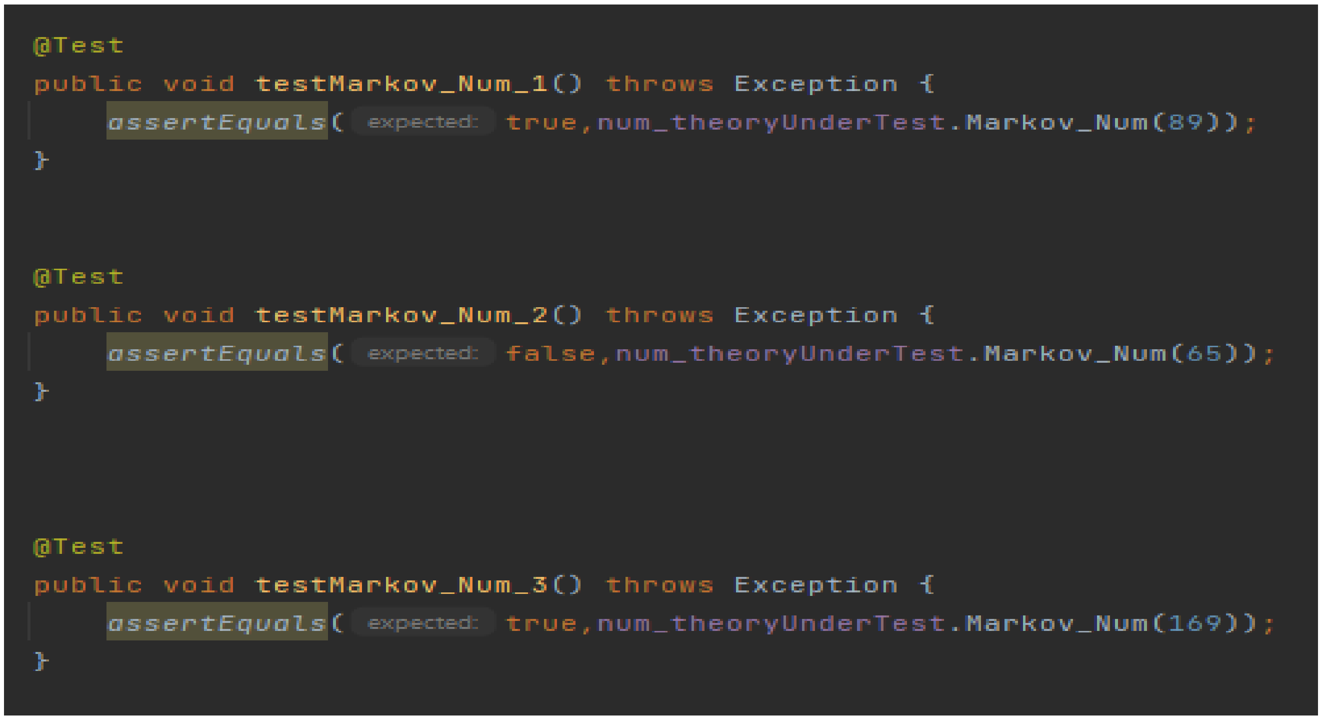


1
Start
2
Statement 1-5
3
i = 1
4
i <= n
True
5
j = 1
6
j <= n
False
True
7
k = 1
False
8
k <= n
True
9
l = (i * i) + (j * j) + (k * k);
r = 3 * i * j * k;
False
10
l == r && (n == i || n == j || n == k)
t=1
True
False
11
12
System.out.println(n + " is a Markov Number.")
System.out.println("The First Triple Formed , is : " + "(" + i + " , " + j + " , " + k + ")")
t = 1
k ++
13
t = =1
True
False
14
15
t ==1
j ++
True
False
16
17
t == 0
i ++
True
18

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:



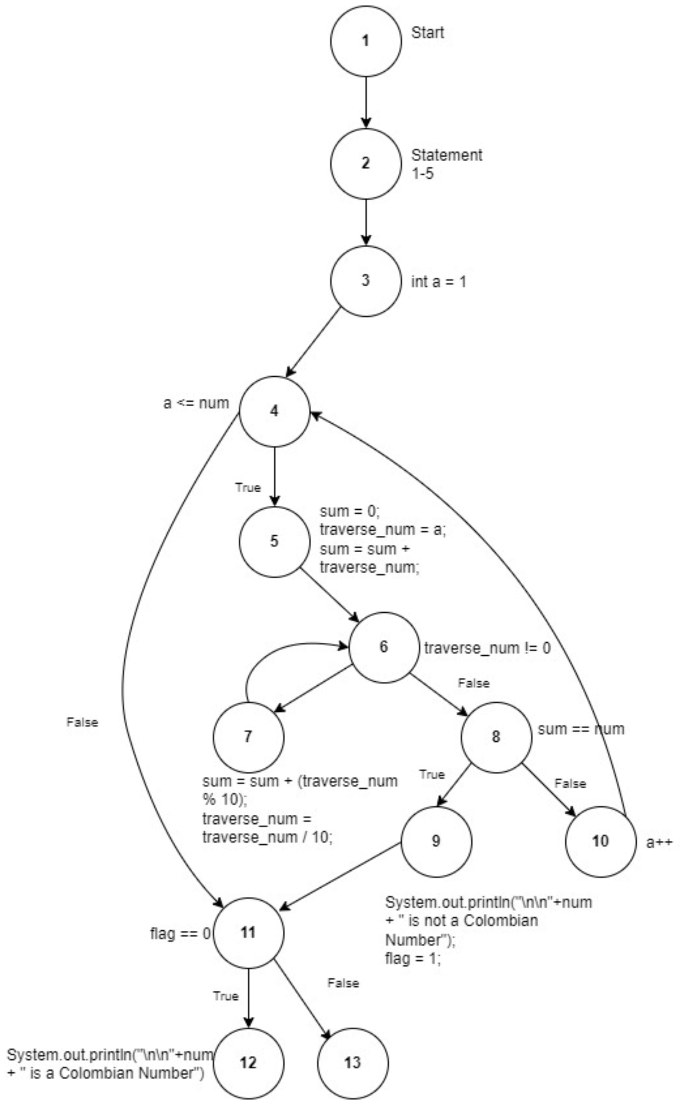
```
@Test
public void testMarkov_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Markov_Num(89));
}


@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));
}
```

v. Functionality Name: Colombian Number

Description/Formula: Colombian number is an integer that cannot be written as the summation of any other integer 'num' and the digits of 'num'.

Examples: 1, 3, 5, 7, 9, 20

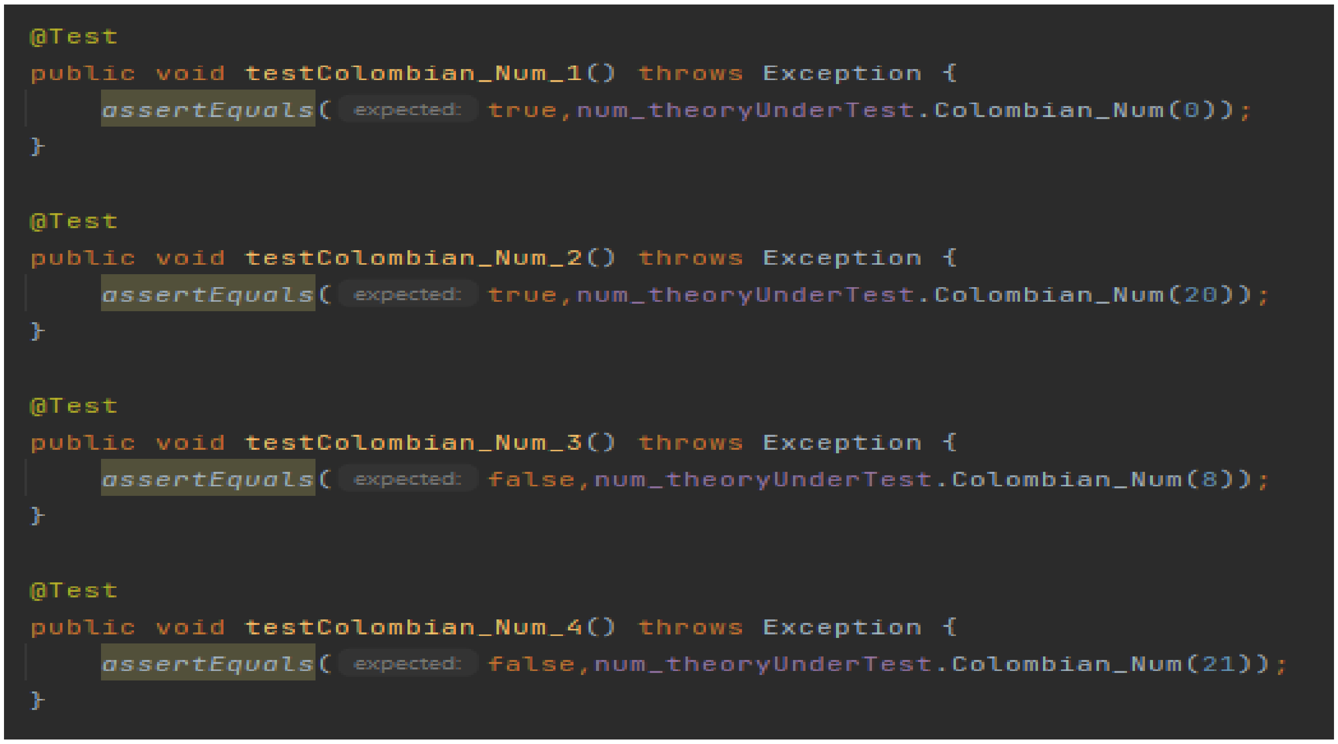
Control Flow Graph:


1
Start
2
Statement 1-5
3
int a = 1
a <= num
4
True
5
sum = 0; traverse_num = a; sum = sum + traverse_num;
6
traverse_num != 0
False
7
8
sum == num
False
sum = sum + (traverse_num % 10); traverse_num = traverse_num / 10;
True
False
9
10
a++
System.out.println("\n\n"+num + " is not a Colombian Number"); flag = 1;
flag == 0
11
True
False
System.out.println("\n\n"+num + " is a Colombian Number")
12
13

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

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

Test Case:



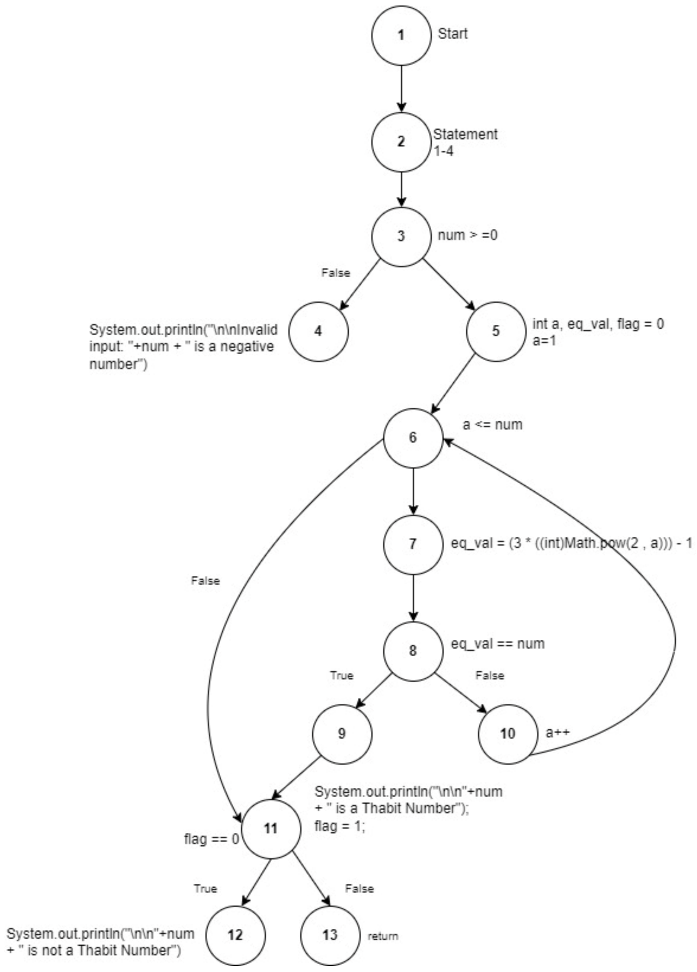
```
@Test
public void testColombian_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Colombian_Num(0));
}

@Test
public void testColombian_Num_2() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Colombian_Num(20));
}

@Test
public void testColombian_Num_3() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Colombian_Num(8));
}

@Test
public void testColombian_Num_4() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Colombian_Num(21));
}
```

vi. 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:

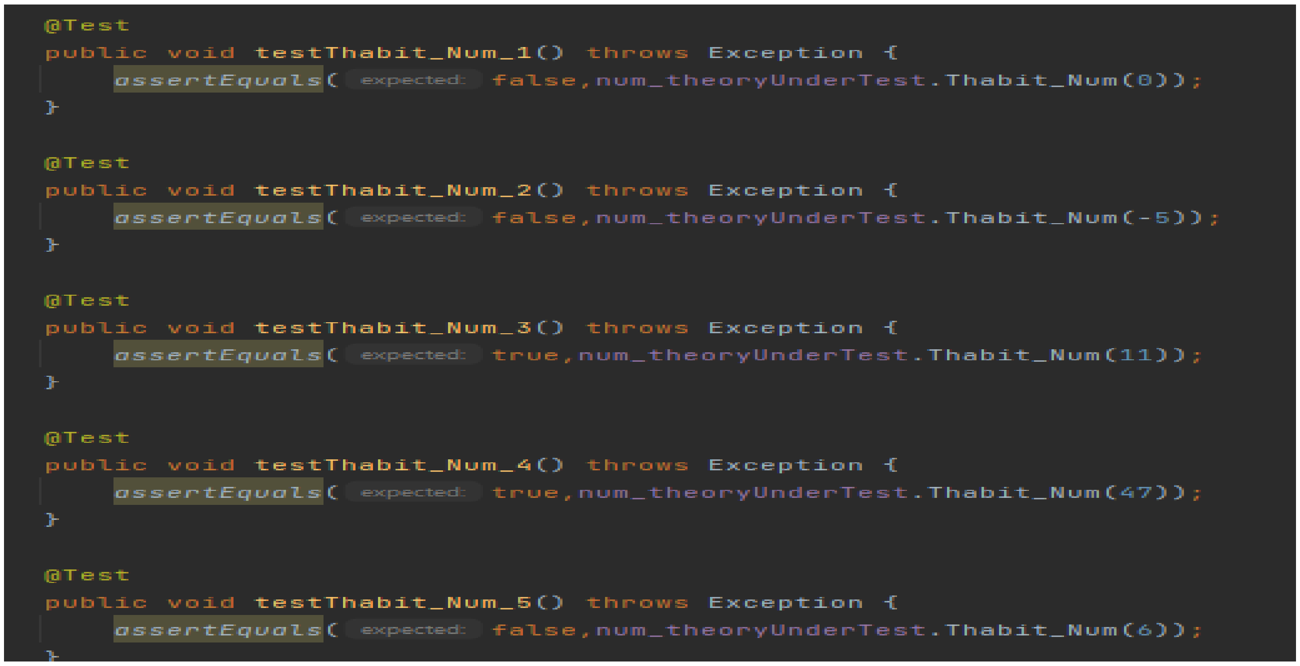


1
Start
2
Statement 1-4
3
num > =0
False
4
System.out.println("\n\nInvalid input: "+num + " is a negative number")
5
int a, eq_val, flag = 0
a=1
6
a <= num
False
7
eq_val = (3 * ((int)Math.pow(2 , a))) - 1
8
eq_val == num
True
False
9
10
a++
System.out.println("\n\n"+num + " is a Thabit Number");
flag = 1;
11
flag == 0
True
False
System.out.println("\n\n"+num + " is not a Thabit Number")
12
13
return

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:



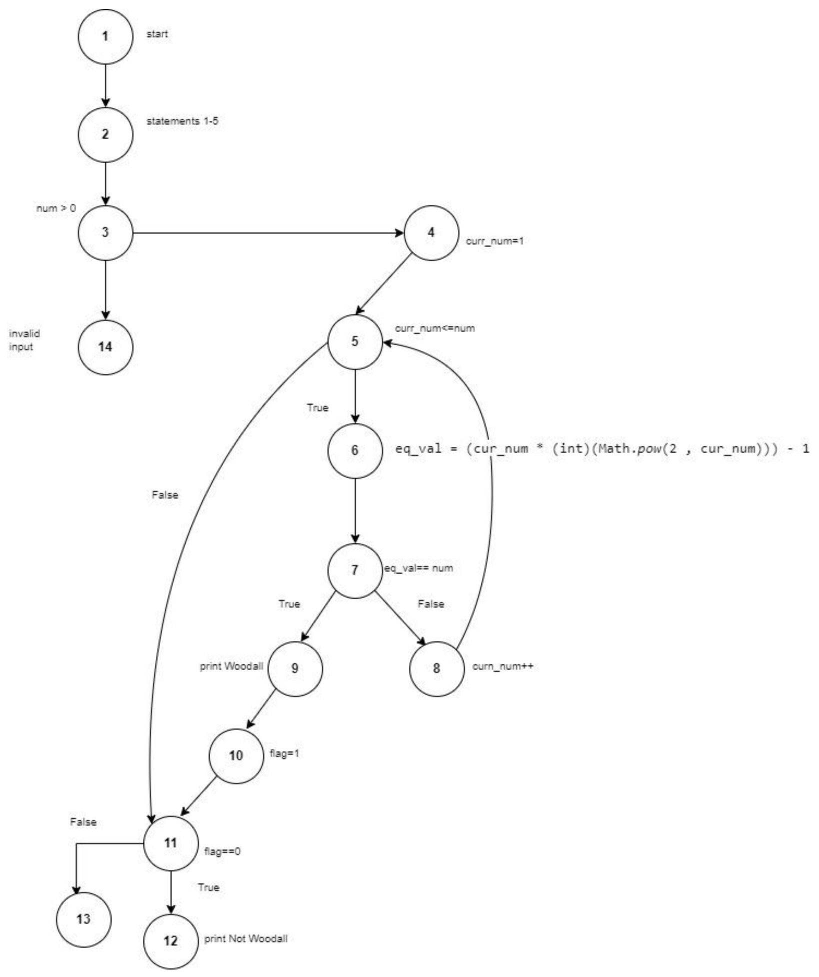
```
@Test
public void testThabit_Num_1() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Thabit_Num(0));
}

@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));
}
```

vii. Functionality Name: Woodall Number

Description/Formula: Woodall number is any natural number 'W_n' that satisfies form $W_n = n * (2^n) - 1$, n = natural number

Examples: 7, 23, 63, 383

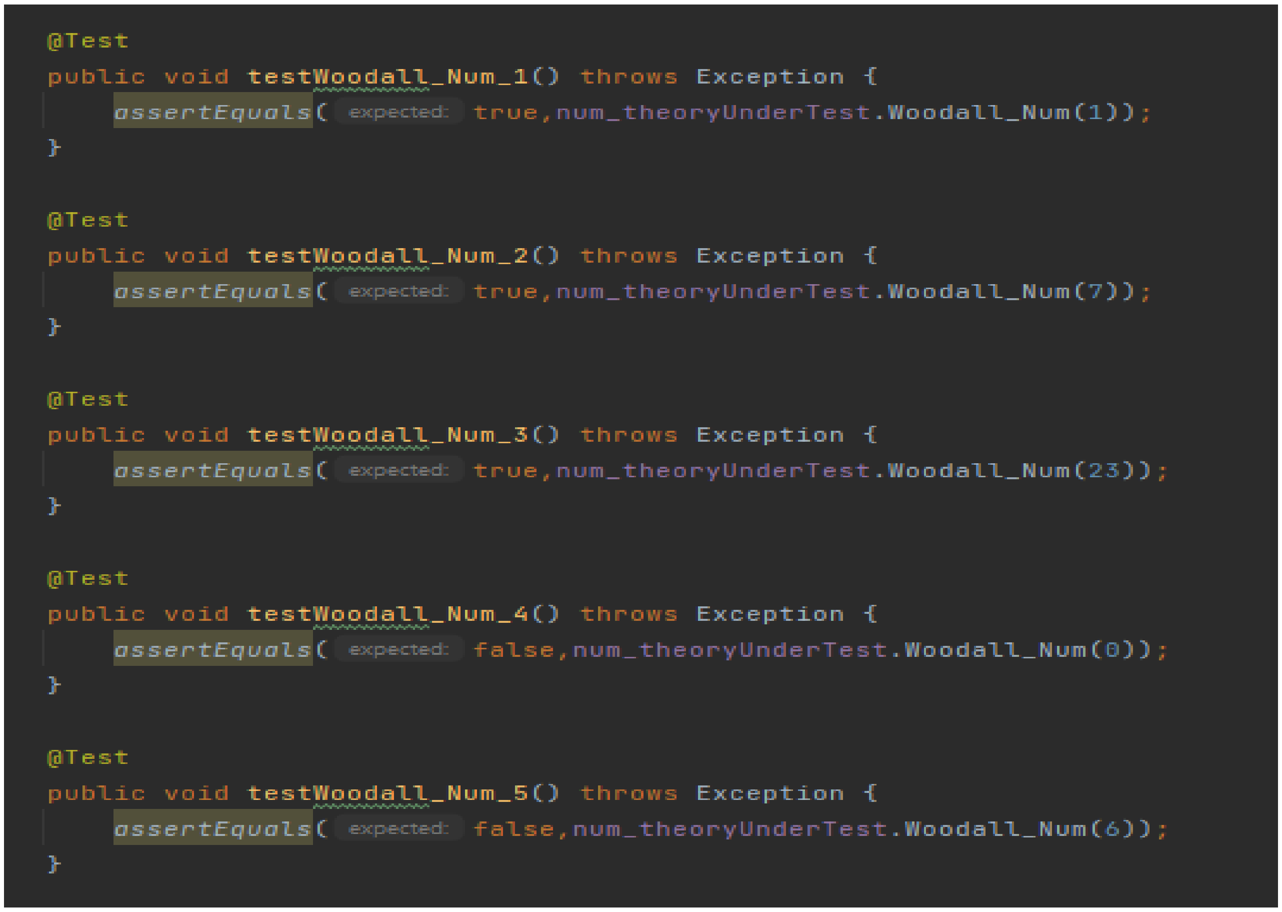
Control Flow Graph:



Edges: [1, 2], [2, 3], [3, 4], [3, 14], [5, 6], [6, 7], [7, 8], [7, 9], [10, 11], …

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

Test Case:



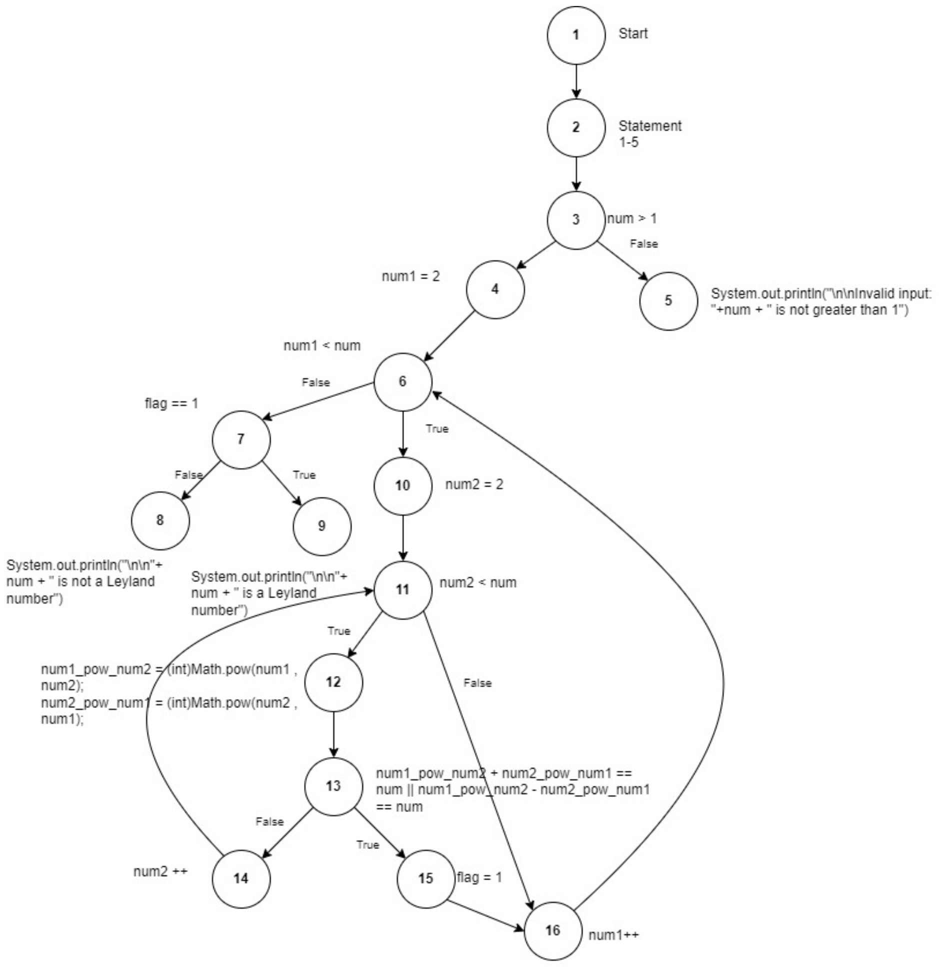
```
@Test
public void testWoodall_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Woodall_Num(1));
}

@Test
public void testWoodall_Num_2() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Woodall_Num(7));
}

@Test
public void testWoodall_Num_3() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Woodall_Num(23));
}

@Test
public void testWoodall_Num_4() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Woodall_Num(0));
}

@Test
public void testWoodall_Num_5() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Woodall_Num(6));
}
```

viii. 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:

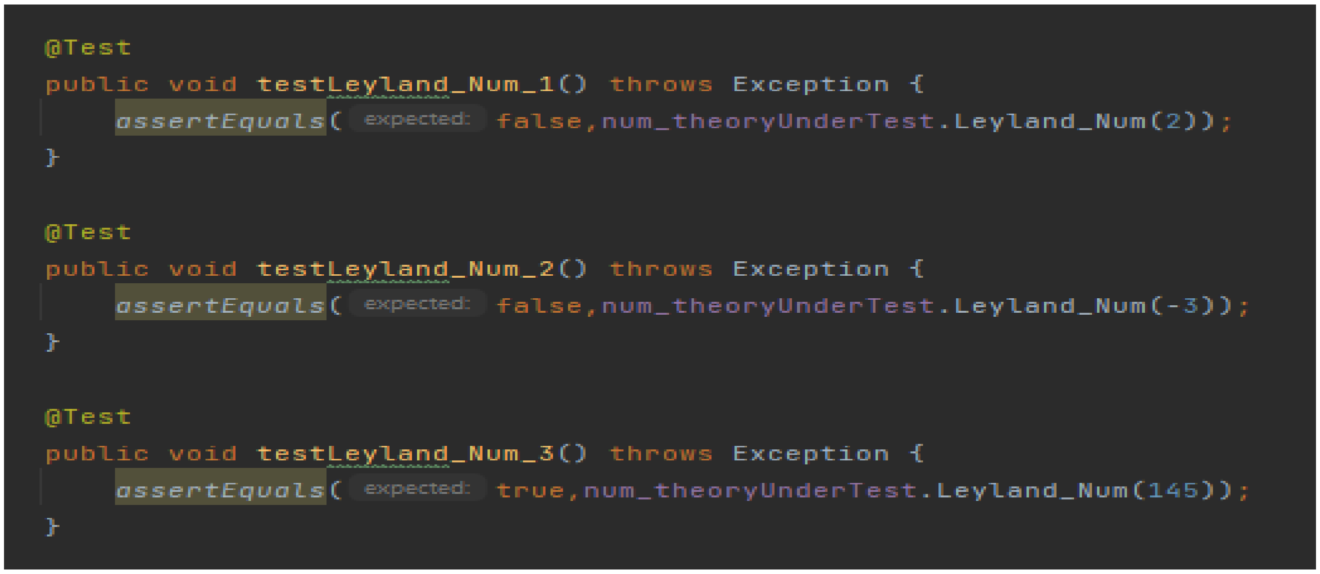


1 Start
2 Statement 1-5
3 num > 1
False
num1 = 2
4
5 System.out.println("\n\nInvalid input: "+num + " is not greater than 1")
num1 < num
6
False
True
flag == 1
7
False
True
8
9
10 num2 = 2
System.out.println("\n\n"+ num + " is not a Leyland number")
System.out.println("\n\n"+ num + " is a Leyland number")
11 num2 < num
True
False
num1_pow_num2 = (int)Math.pow(num1 , num2);
num2_pow_num1 = (int)Math.pow(num2 , num1);
12
13 num1_pow_num2 + num2_pow_num1 == num || num1_pow_num2 - num2_pow_num1 == num
False
True
num2 ++
14
15 flag = 1
16 num1++

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:



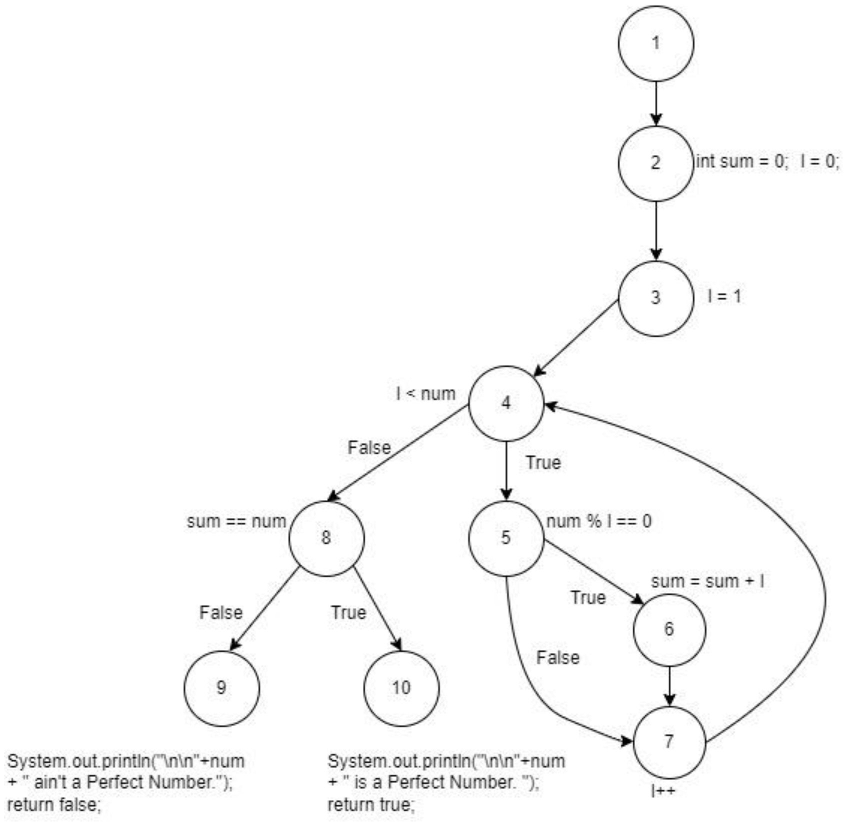
```
@Test
public void testLeyland_Num_1() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Leyland_Num(2));
}

@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));
}
```

ix. 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

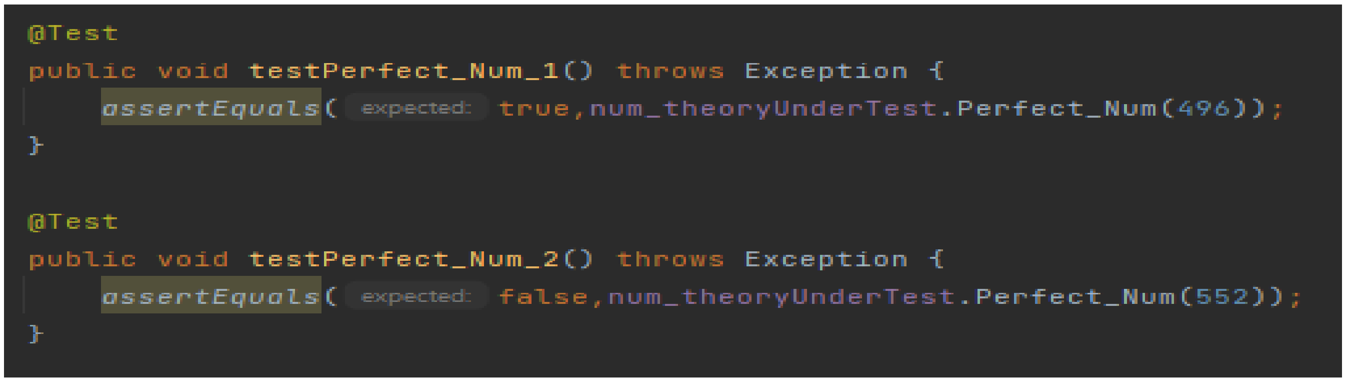
Control Flow Graph:


1
2
int sum = 0; I = 0;
3
I = 1
4
I < num
False
True
8
sum == num
5
num % I == 0
False
True
True
sum = sum + I
6
False
9
10
7
System.out.println("\n\n"+num + " ain't a Perfect Number."); return false;
System.out.println("\n\n"+num + " is a Perfect Number. "); return true;
I++

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:



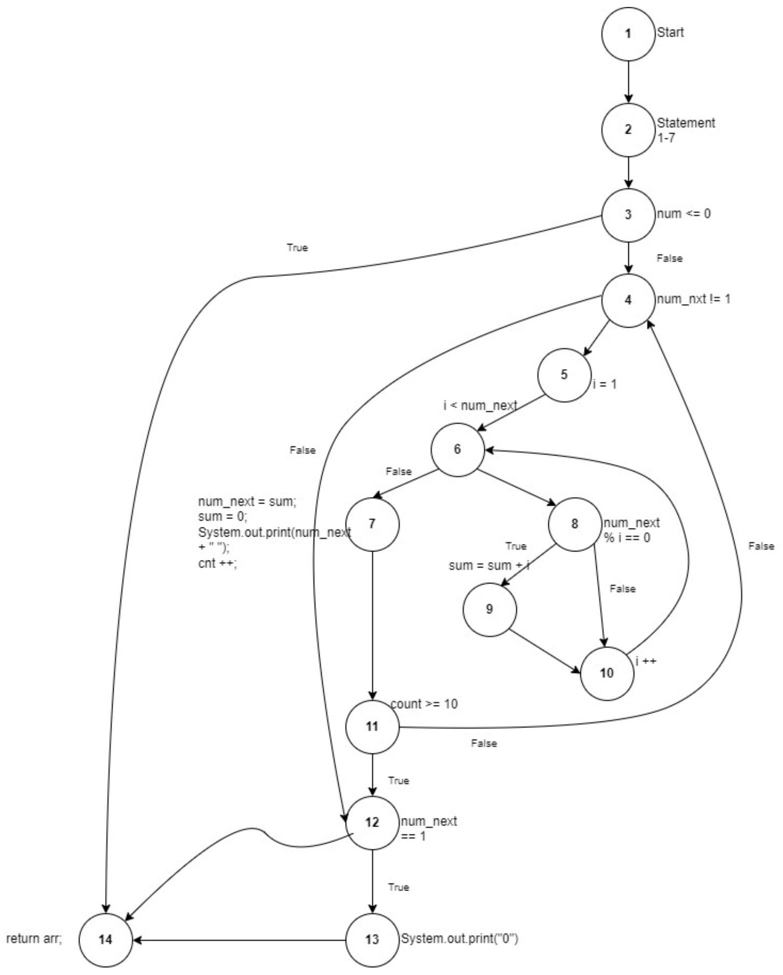
```
@Test
public void testPerfect_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Perfect_Num(496));
}

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

x. Functionality Name: Aliquot Sequence

Description/Formula: Aliquot sequence is a sequence of positive integers where every term is the summation of proper divisors of the previous term. If the sequence reaches number = 1, it stops as the sum for 1 is 0.

Examples: Aliquot Sequence of 10 = 10, 8, 7, 1, 0

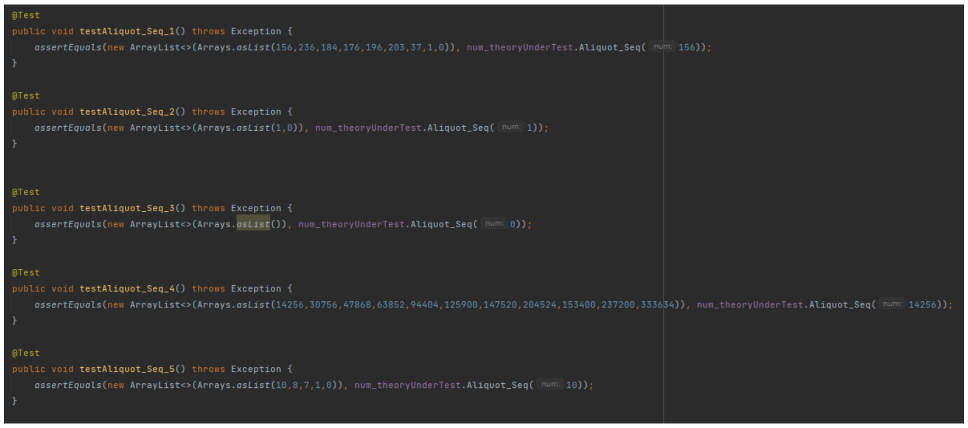
Control Flow Graph:



Edges: [1, 2], [2, 3], [3, 4], [3, 14], [4, 5], [6, 7], [7, 11], [8, 9], …

Prime Paths: [1, 2, 3, 14], [6, 8, 10, 6], [9, 10, 6, 8, 9], [1, 2, 3, 4, 12, 13, 14], [5, 6, 7, 11, 4, 12, 14], ...

Test Case:



```java
@Test
public void testAliquot_Seq_1() throws Exception {
    assertEquals(new ArrayList<>(Arrays.asList(156,236,184,176,196,203,37,1,0)), num_theoryUnderTest.Aliquot_Seq( num: 156));
}

@Test
public void testAliquot_Seq_2() throws Exception {
    assertEquals(new ArrayList<>(Arrays.asList(1,0)), num_theoryUnderTest.Aliquot_Seq( num: 1));
}


@Test
public void testAliquot_Seq_3() throws Exception {
    assertEquals(new ArrayList<>(Arrays.asList()), num_theoryUnderTest.Aliquot_Seq( num: 0));
}

@Test
public void testAliquot_Seq_4() throws Exception {
    assertEquals(new ArrayList<>(Arrays.asList(14256,30756,47868,63852,94404,125900,147520,204524,153400,237200,333634)), num_theoryUnderTest.Aliquot_Seq( num: 14256));
}

@Test
public void testAliquot_Seq_5() throws Exception {
    assertEquals(new ArrayList<>(Arrays.asList(10,8,7,1,0)), num_theoryUnderTest.Aliquot_Seq( num: 10));
}
```

xi. Functionality Name: Sociable Number

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

Examples: 1264460

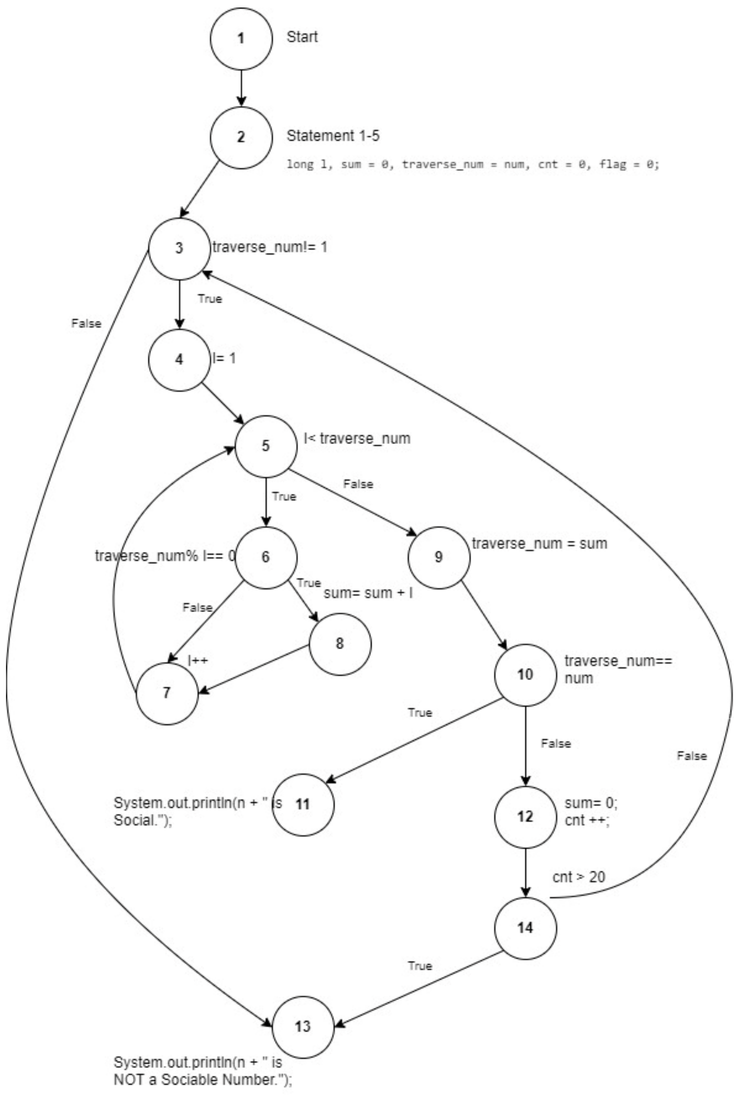
Proper divisors sum of 1264460 is = 1547860

Proper divisors sum of 1547860 is = 1727636

Proper divisors sum of 1727636 is = 1305184

Proper divisors sum of 1305184 is = 1264460

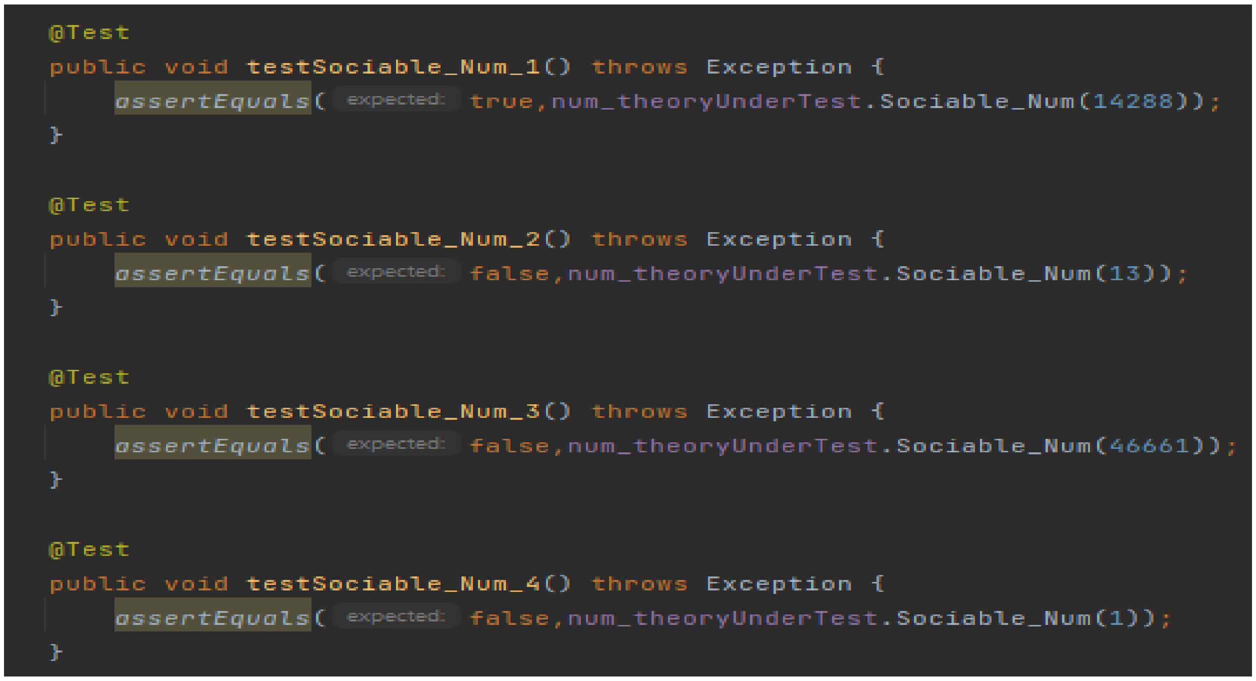
Control Flow Graph:



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:



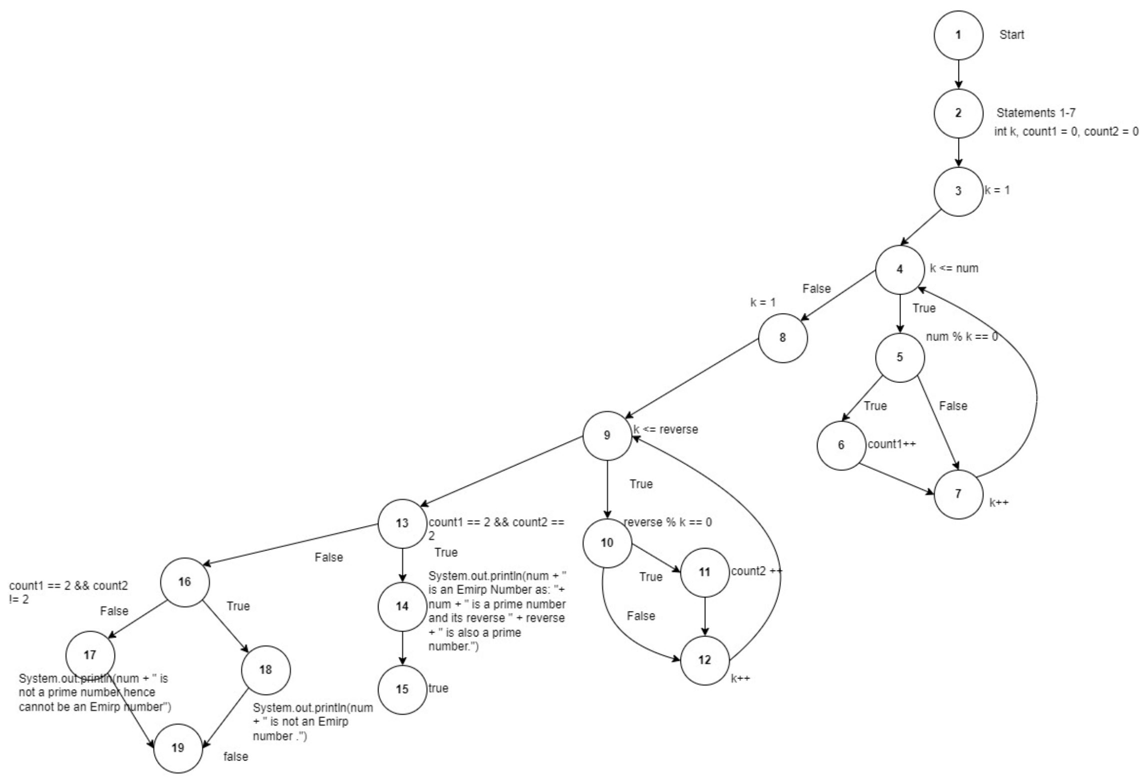
```
@Test
public void testSociable_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Sociable_Num(14288));
}

@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));
}
```

xii. Functionality Name: Emirp Number

Description/Formula: Emirp Number is a prime number that results in another prime number when its digits are reversed.

Examples: 13, 17, 31, 113, 149, 157

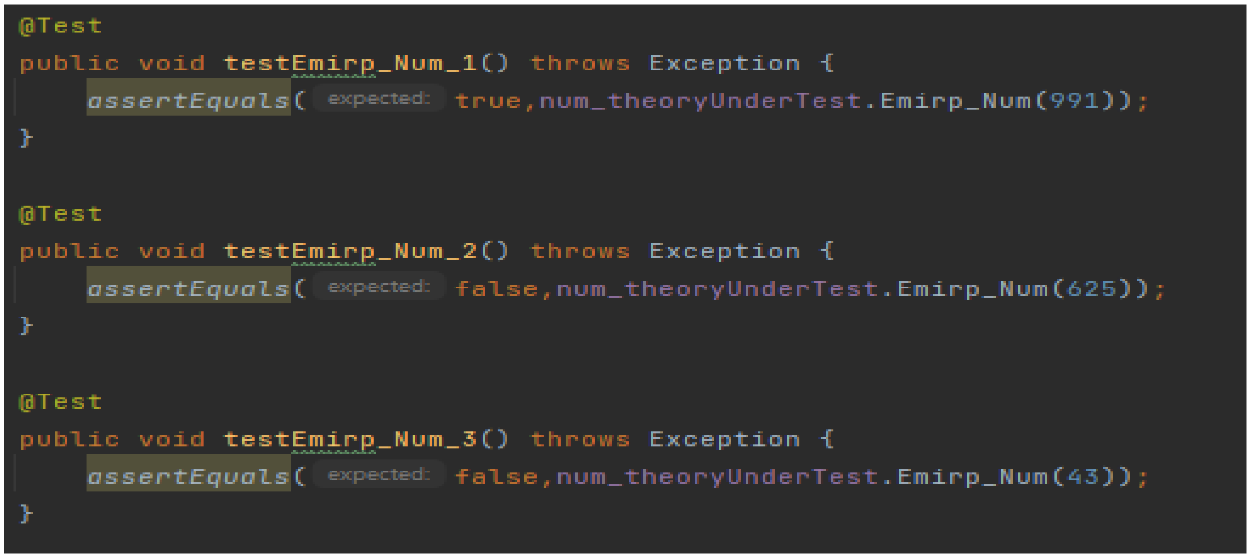
Control Flow Graph:


1
Start
2
Statements 1-7
int k, count1 = 0, count2 = 0
3
k = 1
4
k <= num
False
True
k = 1
8
5
num % k == 0
True
False
6
count1++
7
k++
9
k <= reverse
True
13
count1 == 2 && count2 == 2
True
10
reverse % k == 0
True
11
count2 ++
False
12
k++
False
16
count1 == 2 && count2 != 2
False
True
14
System.out.println(num + " is an Emirp Number as: "+ num + " is a prime number and its reverse " + reverse + " is also a prime number.")
15
true
17
System.out.println(num + " is not a prime number hence cannot be an Emirp number")
18
System.out.println(num + " is not an Emirp number .")
19
false

Edges: [1, 2], [2, 3], [3, 4], [9, 13], [4, 5], [6, 7], [7, 4], [8, 9], …

Prime Paths: [4, 5, 7, 4], [5, 7, 4, 5], [4, 5, 6, 7, 4], [5, 6, 7, 4, 5], ...

Test Case:



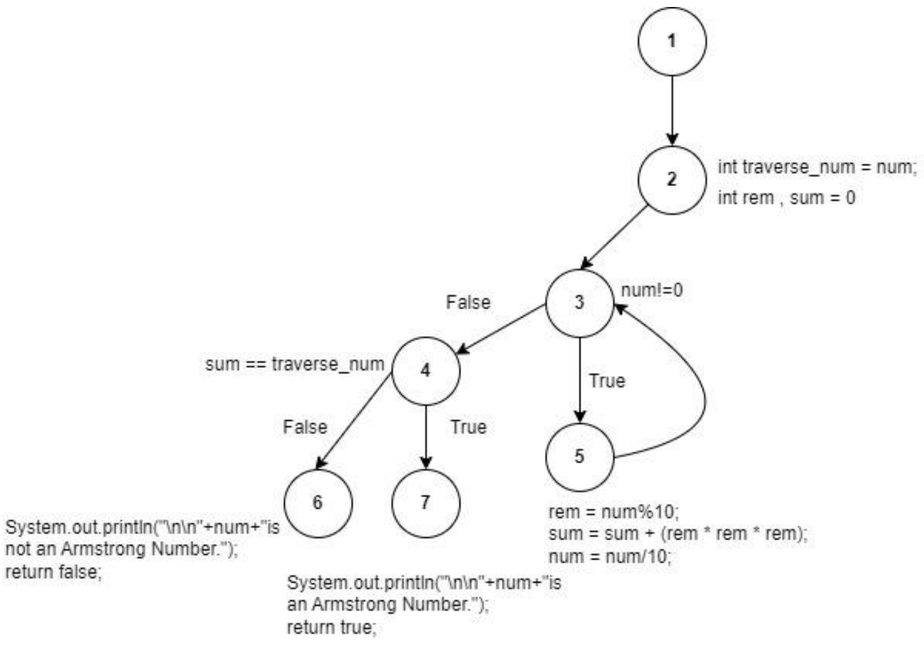
```
@Test
public void testEmirp_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Emirp_Num(991));
}

@Test
public void testEmirp_Num_2() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Emirp_Num(625));
}

@Test
public void testEmirp_Num_3() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Emirp_Num(43));
}
```

xiii. 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

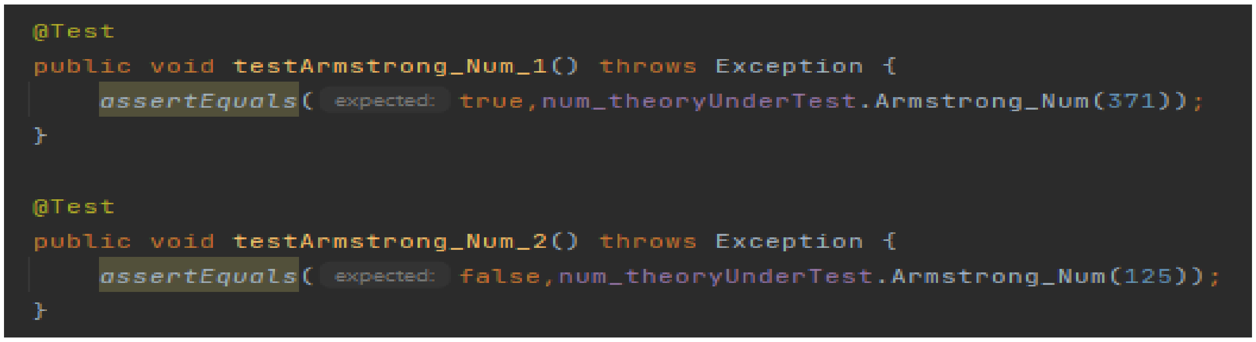
Control Flow Graph:


1
2
int traverse_num = num;
int rem , sum = 0
3
num!=0
False
True
4
sum == traverse_num
False
True
5
6
7
System.out.println("\n\n"+num+"is not an Armstrong Number.");
return false;
System.out.println("\n\n"+num+"is an Armstrong Number.");
return true;
rem = num%10;
sum = sum + (rem * rem * rem);
num = num/10;

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:

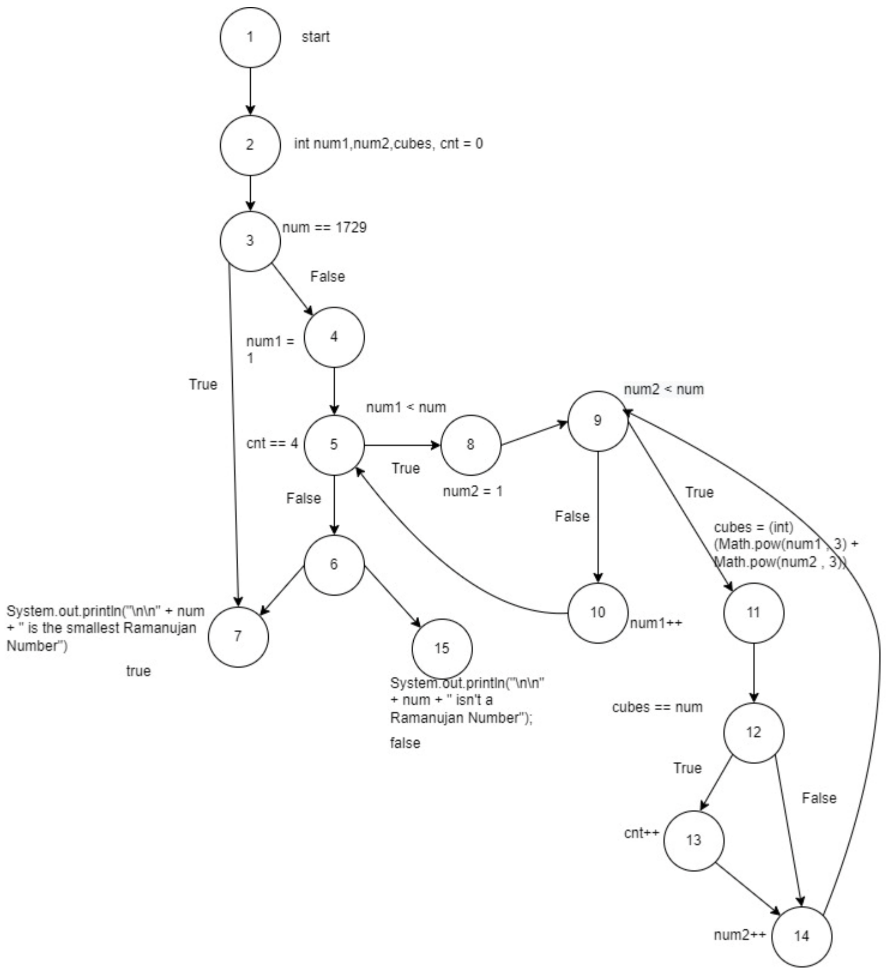

```
@Test
public void testArmstrong_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Armstrong_Num(371));
}

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

xiv. 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:

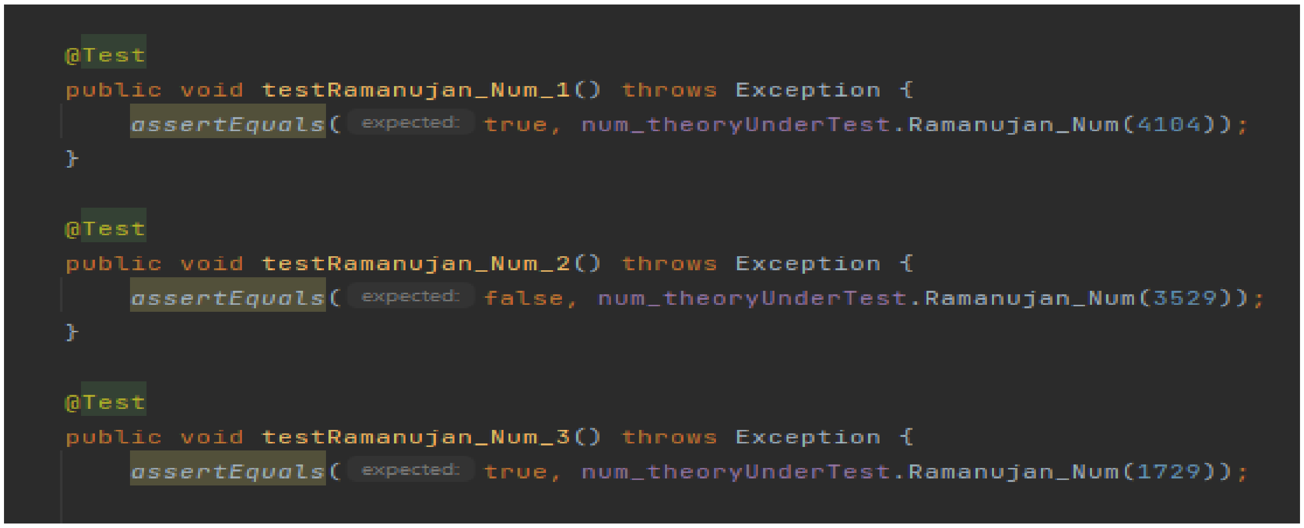


start
int num1,num2,cubes, cnt = 0
num == 1729
False
True
num1 = 1
num1 < num
cnt == 4
True
False
num2 = 1
num2 < num
False
True
cubes = (int) (Math.pow(num1 , 3) + Math.pow(num2 , 3))
System.out.println("\n\n" + num + " is the smallest Ramanujan Number")
true
num1++
System.out.println("\n\n" + num + " isn't a Ramanujan Number");
false
cubes == num
True
False
cnt++
num2++

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:



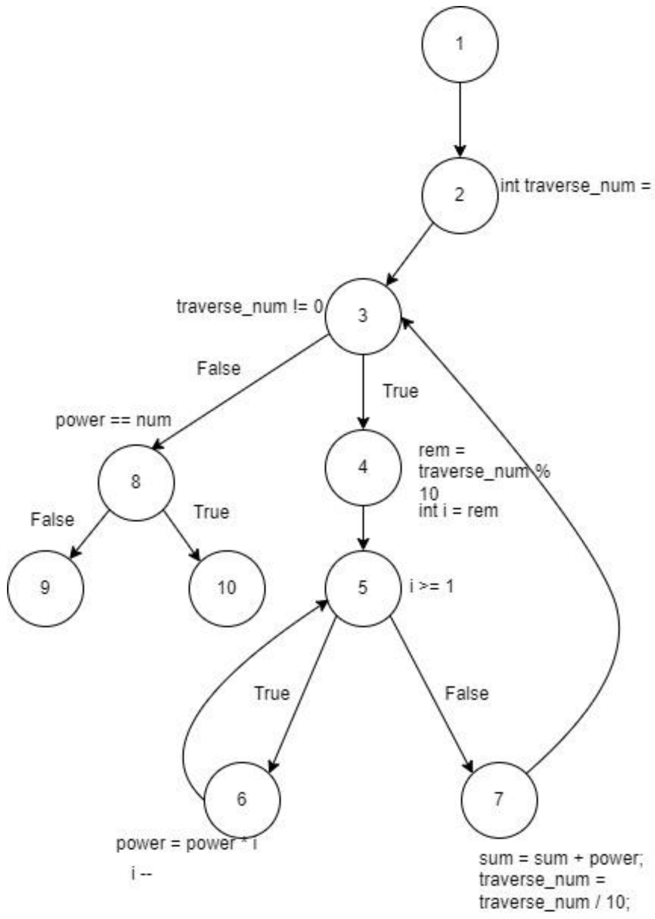
```
@Test
public void testRamanujan_Num_1() throws Exception {
    assertEquals( expected: true, num_theoryUnderTest.Ramanujan_Num(4104));
}

@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));
```

xv. 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

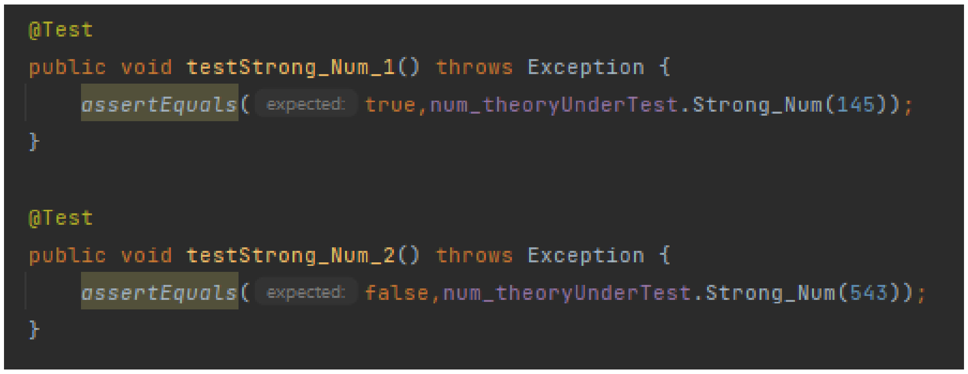
Control Flow Graph:



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:



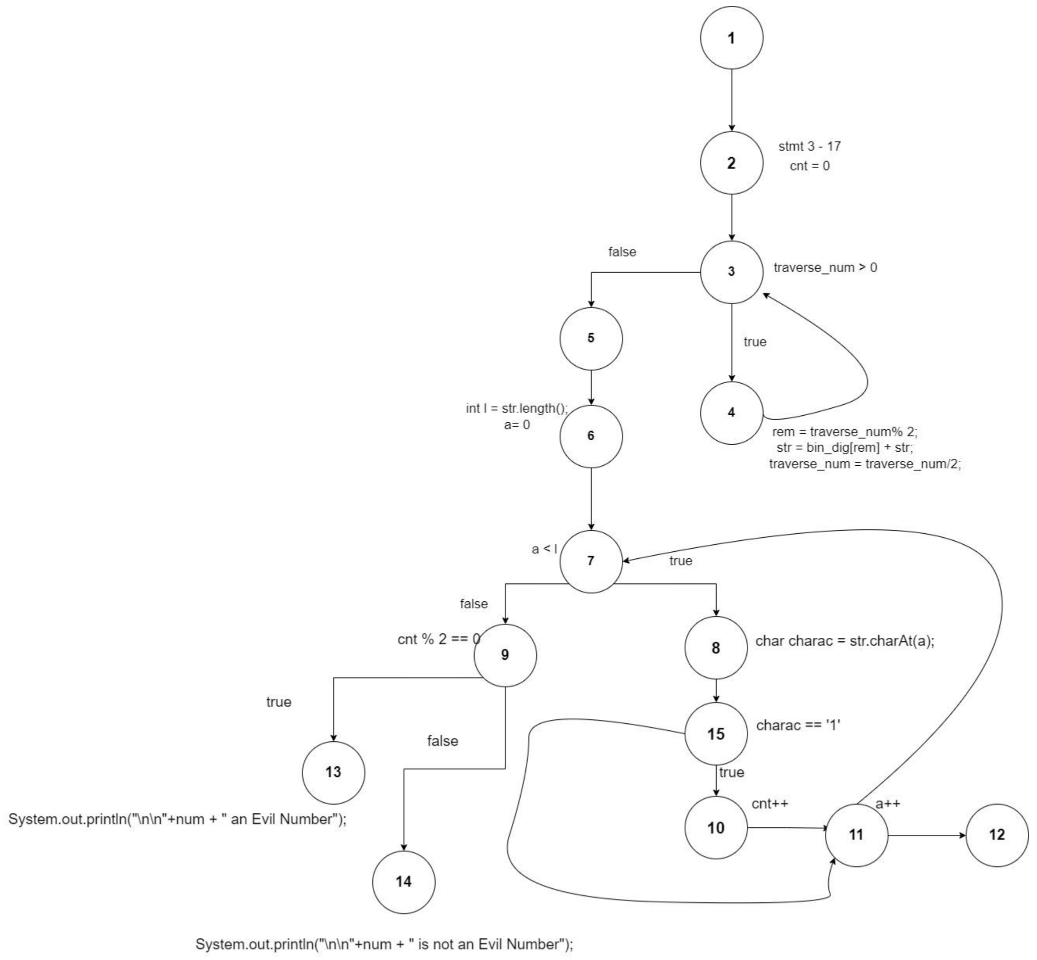
```
@Test
public void testStrong_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Strong_Num(145));
}

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

xvi. Functionality Name: Evil Number

Description/Formula: A positive and whole number which has even number of 1's in its binary form is called an evil number.

Examples: 24, 4324

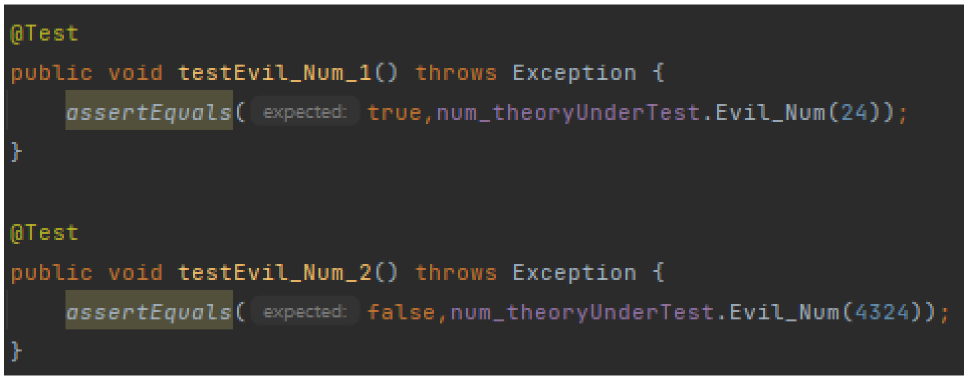
Control Flow Graph:


1
2
stmt 3 - 17
cnt = 0
3
traverse_num > 0
false
true
4
rem = traverse_num% 2;
str = bin_dig[rem] + str;
traverse_num = traverse_num/2;
5
6
int l = str.length();
a= 0
7
a < l
true
false
8
char charac = str.charAt(a);
15
charac == '1'
true
10
cnt++
11
a++
12
9
cnt % 2 == 0
true
false
13
System.out.println("\n\n"+num + " an Evil Number");
14
System.out.println("\n\n"+num + " is not an Evil Number");

Edges: [1, 2], [2, 3], [3, 4], [3, 5], [5, 6], [6, 7], [7, 9], [8, 15], [11, 7], ...

Prime Paths: [15, 10, 11, 7, 8, 15], [4, 3, 5, 6, 7, 9, 13], [8, 15, 10, 11, 7, 9, 14], [1, 2, 3, 5, 6, 7, 9, 13], ...

Test Case:

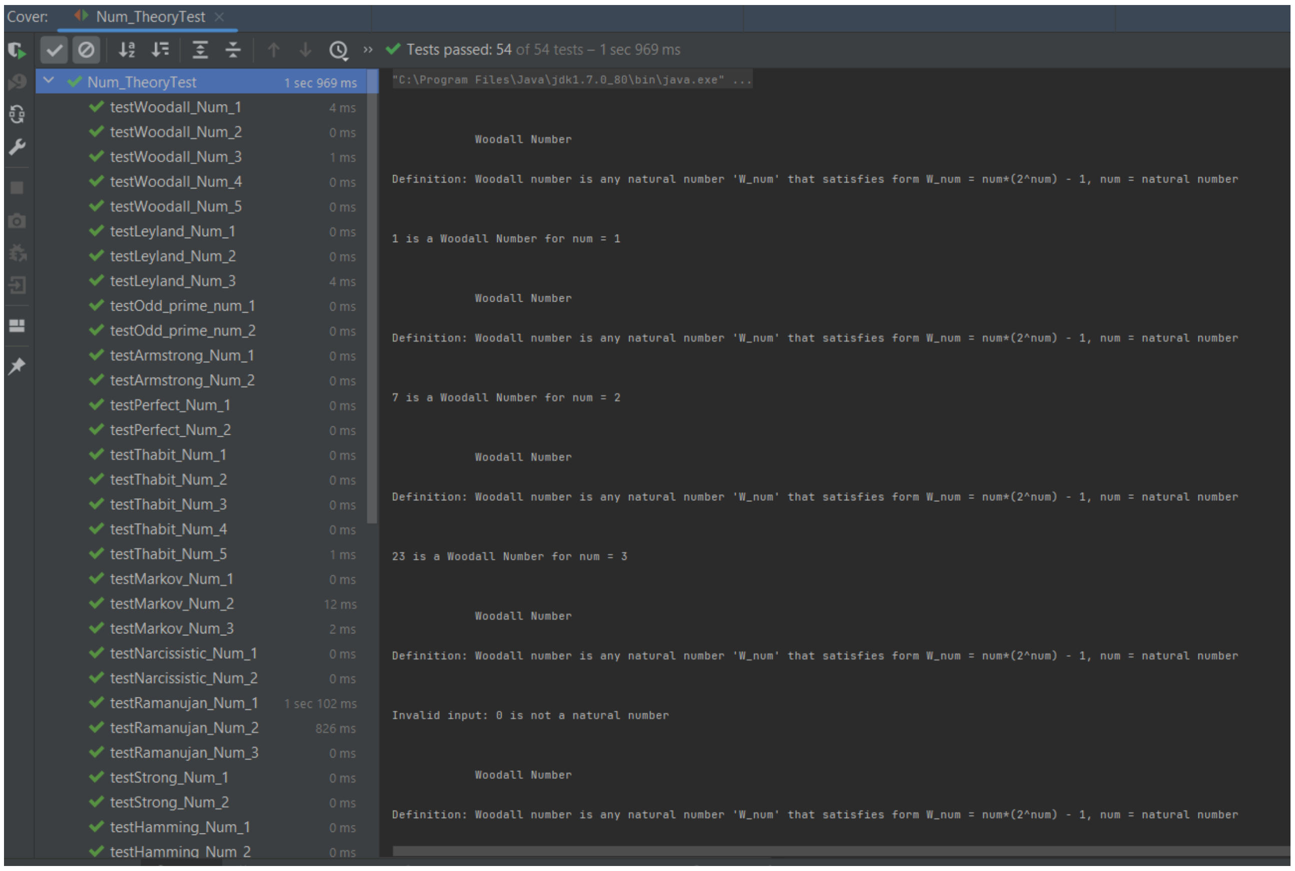


```
@Test
public void testEvil_Num_1() throws Exception {
    assertEquals( expected: true,num_theoryUnderTest.Evil_Num(24));
}

@Test
public void testEvil_Num_2() throws Exception {
    assertEquals( expected: false,num_theoryUnderTest.Evil_Num(4324));
}
```

5. Results

We have tested sixteen 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 54 test cases for the functionalities and the results are as shown below:

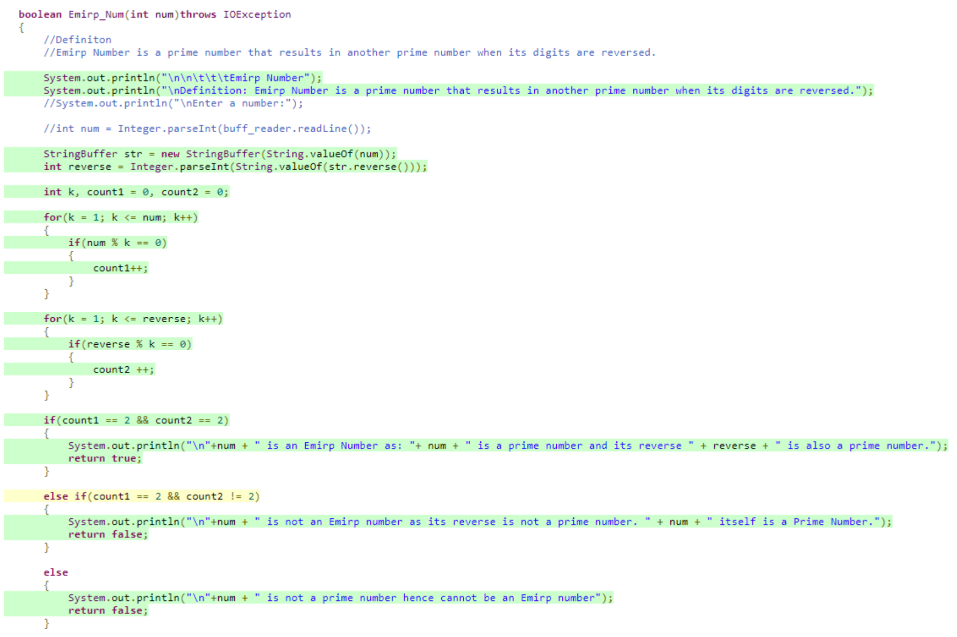
i. This screenshot shows that the test cases (54 of 54 tests) have passed successfully.



```
Tests passed: 54 of 54 tests – 1 sec 969 ms

Num_TheoryTest            1 sec 969 ms
  testWoodall_Num_1       4 ms
  testWoodall_Num_2       0 ms
  testWoodall_Num_3       1 ms
  testWoodall_Num_4       0 ms
  testWoodall_Num_5       0 ms
  testLeyland_Num_1       0 ms
  testLeyland_Num_2       0 ms
  testLeyland_Num_3       4 ms
  testOdd_prime_num_1     0 ms
  testOdd_prime_num_2     0 ms
  testArmstrong_Num_1     0 ms
  testArmstrong_Num_2     0 ms
  testPerfect_Num_1       0 ms
  testPerfect_Num_2       0 ms
  testThabit_Num_1        0 ms
  testThabit_Num_2        0 ms
  testThabit_Num_3        0 ms
  testThabit_Num_4        0 ms
  testThabit_Num_5        1 ms
  testMarkov_Num_1        0 ms
  testMarkov_Num_2        12 ms
  testMarkov_Num_3        2 ms
  testNarcissistic_Num_1  0 ms
  testNarcissistic_Num_2  0 ms
  testRamanujan_Num_1     1 sec 102 ms
  testRamanujan_Num_2     826 ms
  testRamanujan_Num_3     0 ms
  testStrong_Num_1        0 ms
  testStrong_Num_2        0 ms
  testHamming_Num_1       0 ms
  testHamming_Num_2       0 ms
```

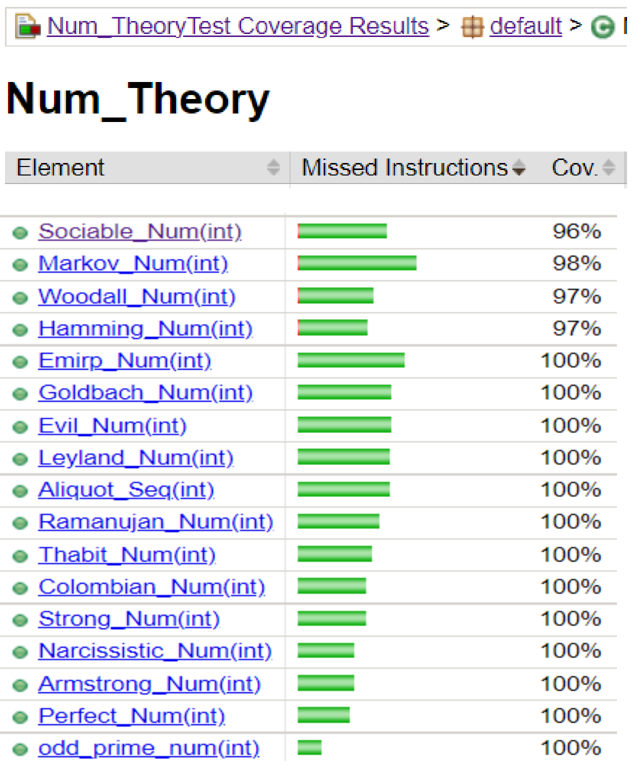
```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...


            Woodall Number

Definition: Woodall number is any natural number 'W_num' that satisfies form W_num = num*(2^num) - 1, num = natural number


1 is a Woodall Number for num = 1


            Woodall Number

Definition: Woodall number is any natural number 'W_num' that satisfies form W_num = num*(2^num) - 1, num = natural number


7 is a Woodall Number for num = 2


            Woodall Number

Definition: Woodall number is any natural number 'W_num' that satisfies form W_num = num*(2^num) - 1, num = natural number


23 is a Woodall Number for num = 3


            Woodall Number

Definition: Woodall number is any natural number 'W_num' that satisfies form W_num = num*(2^num) - 1, num = natural number


Invalid input: 0 is not a natural number


            Woodall Number

Definition: Woodall number is any natural number 'W_num' that satisfies form W_num = num*(2^num) - 1, num = natural number
```

ii. This screenshot shows the line coverage achieved by our test cases.



```java
boolean Emirp_Num(int num)throws IOException
{
    //Definiton
    //Emirp Number is a prime number that results in another prime number when its digits are reversed.

    System.out.println("\n\n\t\t\tEmirp Number");
    System.out.println("\nDefinition: Emirp Number is a prime number that results in another prime number when its digits are reversed.");
    //System.out.println("\nEnter a number:");

    //int num = Integer.parseInt(buff_reader.readLine());

    StringBuffer str = new StringBuffer(String.valueOf(num));
    int reverse = Integer.parseInt(String.valueOf(str.reverse()));

    int k, count1 = 0, count2 = 0;

    for(k = 1; k <= num; k++)
    {
        if(num % k == 0)
        {
            count1++;
        }
    }

    for(k = 1; k <= reverse; k++)
    {
        if(reverse % k == 0)
        {
            count2 ++;
        }
    }

    if(count1 == 2 && count2 == 2)
    {
        System.out.println("\n"+num + " is an Emirp Number as: "+ num + " is a prime number and its reverse " + reverse + " is also a prime number.");
        return true;
    }

    else if(count1 == 2 && count2 != 2)
    {
        System.out.println("\n"+num + " is not an Emirp number as its reverse is not a prime number. " + num + " itself is a Prime Number.");
        return false;
    }

    else
    {
        System.out.println("\n"+num + " is not a prime number hence cannot be an Emirp number");
        return false;
    }
```

iii. This screenshot shows the percentage of instruction coverage for each of the 16 functionalities.

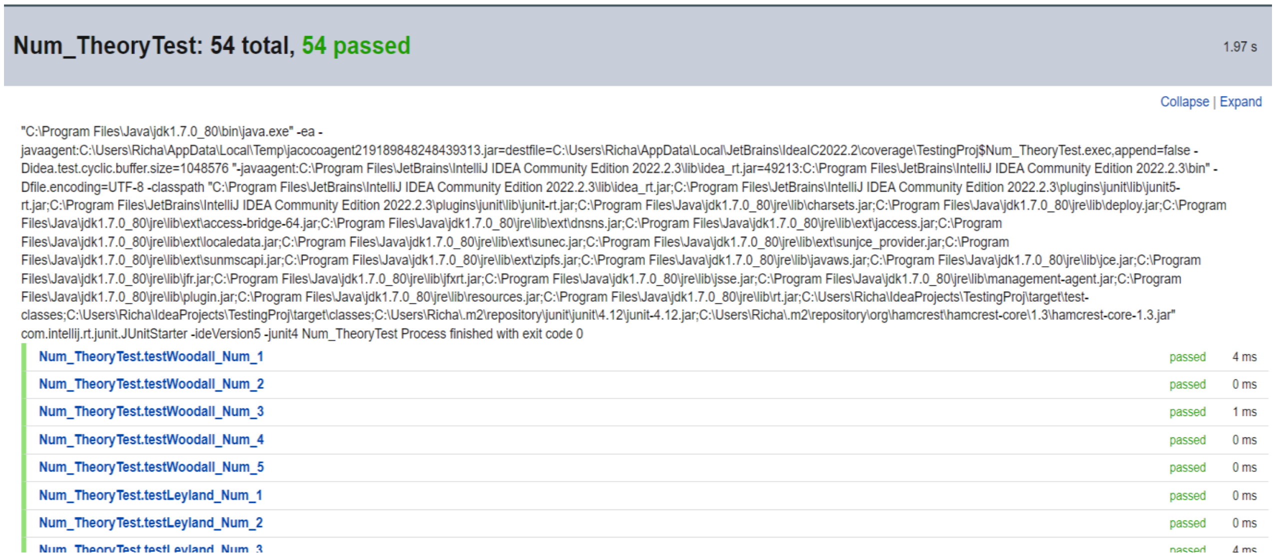


Num_TheoryTest Coverage Results > default >

Num_Theory

Element	Missed Instructions	Cov.
Sociable_Num(int)	🟩	96%
Markov_Num(int)	🟩	98%
Woodall_Num(int)	🟩	97%
Hamming_Num(int)	🟩	97%
Emirp_Num(int)	🟩	100%
Goldbach_Num(int)	🟩	100%
Evil_Num(int)	🟩	100%
Leyland_Num(int)	🟩	100%
Aliquot_Seq(int)	🟩	100%
Ramanujan_Num(int)	🟩	100%
Thabit_Num(int)	🟩	100%
Colombian_Num(int)	🟩	100%
Strong_Num(int)	🟩	100%
Narcissistic_Num(int)	🟩	100%
Armstrong_Num(int)	🟩	100%
Perfect_Num(int)	🟩	100%
odd_prime_num(int)	🟩	100%

6. Conclusion



Num_TheoryTest: 54 total, 54 passed 1.97 s

Collapse | Expand

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" -ea -
javaagent:C:\Users\Richa\AppData\Local\Temp\jacocoagent219189848248439313.jar=destfile=C:\Users\Richa\AppData\Local\JetBrains\IdeaIC2022.2\coverage\TestingProj$Num_TheoryTest.exec,append=false -
Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\lib\idea_rt.jar=49213:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\bin" -
Dfile.encoding=UTF-8 -classpath "C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\lib\idea_rt.jar;C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\plugins\junit\lib\junit5-
rt.jar;C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\plugins\junit\lib\junit-rt.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\deploy.jar;C:\Program
Files\Java\jdk1.7.0_80\jre\lib\ext\access-bridge-64.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\jaccess.jar;C:\Program
Files\Java\jdk1.7.0_80\jre\lib\ext\localedata.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\sunec.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\sunjce_provider.jar;C:\Program
Files\Java\jdk1.7.0_80\jre\lib\ext\sunmscapi.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\zipfs.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\javaws.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\jce.jar;C:\Program
Files\Java\jdk1.7.0_80\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\jfxrt.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\management-agent.jar;C:\Program
Files\Java\jdk1.7.0_80\jre\lib\plugin.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\resources.jar;C:\Program Files\Java\jdk1.7.0_80\jre\lib\rt.jar;C:\Users\Richa\IdeaProjects\TestingProj\target\test-
classes;C:\Users\Richa\IdeaProjects\TestingProj\target\classes;C:\Users\Richa\.m2\repository\junit\junit\4.12\junit-4.12.jar;C:\Users\Richa\.m2\repository\org\hamcrest\hamcrest-core\1.3\hamcrest-core-1.3.jar"
com.intellij.rt.junit.JUnitStarter -ideVersion5 -junit4 Num_TheoryTest Process finished with exit code 0
```

Num_TheoryTest.testWoodall_Num_1	passed	4 ms
Num_TheoryTest.testWoodall_Num_2	passed	0 ms
Num_TheoryTest.testWoodall_Num_3	passed	1 ms
Num_TheoryTest.testWoodall_Num_4	passed	0 ms
Num_TheoryTest.testWoodall_Num_5	passed	0 ms
Num_TheoryTest.testLeyland_Num_1	passed	0 ms
Num_TheoryTest.testLeyland_Num_2	passed	0 ms
Num_TheoryTest.testLeyland_Num_3	passed	4 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 sixteen 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 54 test cases to test the application's control flow.