# SENG3320 Assignment 1: Test Case Design

|  |  |
| --- | --- |
| **Group 10** | |
| Kyle Beattie | C3303374 |
| Ni Zeng | C3238805 |
| Brandon Allen | C3279505 |
| Austin Baxter | C3356468 |
|  |  |

**Table of Contents**

[SENG3320 Assignment 1: Test Case Design 1](#_Toc102155812)

[Introduction 2](#_Toc102155813)

[Task 1: Blackbox Testing 3](#_Toc102155814)

[Method 1: public BigInteger(int signum,byte[] magnitude) 3](#_Toc102155815)

[Method 2: public BigInteger(String val, int radix) 4](#_Toc102155816)

[Method 3: public int compareTo(BigInteger val) 5](#_Toc102155817)

[Task 2: White-box Testing: Structural Testing 8](#_Toc102155818)

[Method 1: public BigInteger gcd(BigInteger y) 8](#_Toc102155819)

[JUnit Test Cases 10](#_Toc102155820)

[Method 2: private static int compareTo(BigInteger x, BigInteger y) 11](#_Toc102155821)

[JUnit Test Cases 13](#_Toc102155822)

[Task 3: White-box Testing: Data Flow Testing 14](#_Toc102155823)

[Method 1: public BigInteger gcd(BigInteger y) 14](#_Toc102155824)

[Method 2: private static int compareTo(BigInteger x, BigInteger y) 20](#_Toc102155825)

[Conclusion 25](#_Toc102155826)

[Group Member Contributions 26](#_Toc102155827)

## Introduction

This report details the testing of a provided copy of the BigInteger class. It includes the testing to four different methods (static and non-static compareTo methods combined) using three different testing methods.

The compareTo methods are examined using blackbox testing, structural testing, and data flow testing. Two constructors were analysed using blackbox testing. A method returning the greatest common divisor was analysed using structural testing and data flow testing.

The purpose of this report is to detail any faults found throughout the testing of the BigInteger class provided.

## Task 1: Blackbox Testing

### Method 1: public BigInteger(int signum,byte[] magnitude)

#### Equivalence Partitions:

* Signum may only be -1,0 or 1.
  + Each case has special properties.
  + -1 denotes a negative value.
  + 0 denotes value must be zero (requires an empty byte array or a byte array populated entirely by zeros).
  + 1 denotes a positive value.
  + -2 and 2 will be included as outside the upper and lower bounds. These values will result in a number format exception.
* Byte array read in big-endian manner.
  + Three simple ways formats of this array may be passed into the constructor.
  + Empty array results in:
    - Zero values for each signum in bounds.
    - Exception if signum is out of bounds.
  + Array populated entirely by zeros results in:
    - Zero values for each signum in bounds.
    - Exception if signum is out of bounds.
  + Array populated by at least one non-zero value produces:
    - Negative value if signum is -1.
    - Positive value if signum is 1.
    - Exception if signum is 0.
    - Exception if signum is out of bounds.

#### Test Values:

* signum values: {-2,-1,0,1,2}
* magnitude values: {empty array, array full of zeros, array with at least one non-zero value}

Every combination of the two sets are used to assess exceptions thrown and resulting values.

### Method 2: public BigInteger(String val, int radix)

#### Equivalence Partitions:

* Radix is in bounds must be greater than or equal to 2 and less than or equal to 36
  + Radix determines what base to read the value string by.
  + Radix may be 2 denoting base 2 all the way to 36 denoting base 36 (every number and alphabet character used)
  + Radix below 2 or above 36 will result in a number format exception.
* Beginning sign char (‘+’, ‘-‘ or neither)
  + Characters may be present at the beginning of the val string to reflect the sign of the resulting value.
  + The ‘+’ character and no character results in a positive value.
  + The ‘-‘ character results in a negative value.
* Illegal characters (characters that are not alphanumeric or ‘+’ or ‘-‘ at the start)
  + Their presence will cause a number format exception.
* Alphanumeric characters in the string cannot exceed the radix.
  + If radix equals 2, val may only contain ‘1’s and ‘0’s.
  + If the val string exceeds the radix a number format exception will be thrown.

#### Test Values:

* Radix values {1,2,10,36,37}
* Beginning character appended val values {‘+’, ‘-‘, ‘’}
* Special character inclusion {present, not present}
* Val string will characters values {zeros and ones, all base 10 digits, every alpha numeric character}

Every combination of these sets are used to assess exceptions thrown and resulting values.

#### Results:

The values with prepended ‘+’ symbols threw number format exceptions.

The bug was found. In cases of prepended ‘+’ there was no implementation to remove the symbol. A solution to this problem would be an if statement after line 1574 seen in figure 1. The if statement would check for a prepending ‘+’ symbol and remove it in the case it was there.

Text

Description automatically generated

Figure 1 BigInteger class code snippet showing the handling of sign symbols prepending the value string on a constructor

### Method 3: public int compareTo(BigInteger val)

#### Equivalence Partitions:

CompareTo had 3 equivalence classes the input could be separated into:

* The BigInteger the function is called through is smaller than BigInteger val
  + This is a valid partition and the function will output “-1”
* The BigInteger the function is called through is the same value as BigInteger val
  + This is also a valid partition and the function will output “0”
* The BigInteger the function is called through is larger than BigInteger val
  + This is also a valid partition and the function will output “1”

During our testing because compareTo requires 2 constructed BigIntegers that are valid, there isn’t any way to create an invalid partition that won’t be processed by compareTo, every valid BigInteger will either have a lower, higher or the same value as any other BigInteger, and any invalid BigInteger simply won’t be constructed and will output a NumberFormatException before the function is called.

#### Test Values:

* Partition 1: x < y
  + xValue = 100, yValue = 1000, Expected result = -1
  + xValue = -100, yValue = 1000, Expected Result = -1
  + xValue = -1000, yValue = -100, Expected Result = -1
  + xValue = 0, yValue = 1, Expected Result = -1
  + xValue = -1, yValue = 0, Expected Result = -1
  + xValue = 11328409283409823143513413247869678880980419280912412345243598747239467094586703945,

yValue = 12490832435987205730517057198325709132141241242144213241414141414141123414123421444, Expected Result = -1

* + xValue = -1132840928340982314351341324786967888098041928091246466423457678465313535578797876,

yValue = 12490832435987205730517057198325709132141241242144243563456345634564364363463456335, Expected Result = -1

* + xValue = -1332840928340982314351341324786967888098041928091242345523523452345234523452345234,
  + yValue = -1249083243598720573051705719832570913214124124214422345234523453252345324525235534, Expected Result = -1
* Partition 2: x = y
  + xValue = 100, yValue = 100, Expected Result = 0
  + xValue = -100, yValue = -100, Expected Result = 0
  + xValue = +100, yValue = +100, Expected Result = 0
  + xValue = 0, yValue = 0, Expected Result = 0
  + xValue = -0, yValue = -0, Expected Result = 0
  + xValue = 0, yValue = -0, Expected Result = 0
  + xValue = -0, yValue = 0, Expected Result = 0
  + xValue = 138946198734618239764978163429813267498126479821364981634812357234985723498674231487341981273409173587643826598243513752439850243759,
  + yValue = 138946198734618239764978163429813267498126479821364981634812357234985723498674231487341981273409173587643826598243513752439850243759, Expected Result = 0
* Partition 3: x > y
  + xValue = 1000, yValue = 100, Expected Result = 1
  + xValue = -100, yValue = -1000, Expected Result = 1
  + xValue = 1000, yValue = -100, Expected Result = 1
  + xValue = 133284092834098231435134132478696788809804192809124,

yValue = 124908324359872057305170571983257091321412412421442, Expected Result = 1

* + xValue = 113284092834098231435134132478696788809804192809124, yValue = -124908324359872057305170571983257091321412412421442, Expected Result = 1
  + xValue = -113284092834098231435134132478696788809804192809124, yValue = -124908324359872057305170571983257091321412412421442, Expected Result = 1
  + xValue = 1, yValue = 0, Expected Result = 1
  + xValue = 0, yValue = -1, Expected Result = 1

Design test cases using the Equivalence Partitioning technique. State clearly the equivalence classes. Clearly specify which partitions/classes are being tested, the corresponding test inputs, and the expected outputs

## Task 2: White-box Testing: Structural Testing

### Method 1: public BigInteger gcd(BigInteger y)

Diagram, schematic

Description automatically generated

Figure 2: Task 2 GCD CFG

#### Test Cases

##### Statement Coverage

The following test cases were able to achieve 100% statement coverage.

* GCDTest1: xValue = 0, yValue = 0
* GCDTest2: xValue = -1, yValue = -1
* GCDTest3: xValue = 1, yValue = -2147483648
* GCDTest4: xValue = -2147483648, yValue = 0
* GCDTest5: xValue = 948464564845641654444444, yValue = 56465165555

##### Branch Decision Coverage

The following test cases were able to achieve 100% branch decision coverage.

* GCDTest1: xValue = 0, yValue = 0
* GCDTest2: xValue = -1, yValue = -1
* GCDTest3: xValue = 1, yValue = -2147483648
* GCDTest4: xValue = -2147483648, yValue = 0
* GCDTest5: xValue = 948464564845641654444444, yValue = 56465165555
* GCDTest6: xValue = 1, yValue = 1

##### Condition Coverage

The following test cases were able to achieve 100% condition coverage.

* GCDTest1: xValue = 0, yValue = 0
* GCDTest2: xValue = -1, yValue = -1
* GCDTest3: xValue = 1, yValue = -2147483648
* GCDTest4: xValue = -2147483648, yValue = 0
* GCDTest5: xValue = 948464564845641654444444, yValue = 56465165555
* GCDTest6: xValue = 1, yValue = 1
* GCDTest7: xValue = 1, yValue = 948464564845641654444444

##### Condition/Decision Coverage

The following test cases were able to achieve 100% condition/decision coverage.

* GCDTest1: xValue = 0, yValue = 0
* GCDTest2: xValue = -1, yValue = -1
* GCDTest3: xValue = 1, yValue = -2147483648
* GCDTest4: xValue = -2147483648, yValue = 0
* GCDTest5: xValue = 948464564845641654444444, yValue = 56465165555
* GCDTest6: xValue = 1, yValue = 1
* GCDTest7: xValue = 1, yValue = 948464564845641654444444

##### Multiple Condition Coverage

The following test cases were able to achieve 100% of the feasible multiple condition coverage. When y.words == null is false than it is impossible for yval != Integer.MIN\_VALUE to also evaluate as false causing two test conditions to not be feasible.

* GCDTest1: xValue = 0, yValue = 0
* GCDTest2: xValue = -1, yValue = -1
* GCDTest3: xValue = 1, yValue = -2147483648
* GCDTest4: xValue = -2147483648, yValue = 0
* GCDTest5: xValue = 948464564845641654444444, yValue = 56465165555
* GCDTest6: xValue = 1, yValue = 1
* GCDTest7: xValue = 1, yValue = 948464564845641654444444
* GCDTest8: xValue = -2147483648, yValue = -2147483648
* GCDTest9: xValue = -2147483648, yValue = 9484645648456416544444445

### JUnit Test Cases

The test cases for the GCD method can be found in src/Tests/StructuralTestingGCD.java. Each test comprises of three Big Integer values xValue, yValue and expectedResult which are tested using the JUnit assertEquals testing method. These test cases result in 100% of the feasible coverage for all the required code coverage methods.

### Method 2: private static int compareTo(BigInteger x, BigInteger y)

Chart

Description automatically generated

#### Test Cases

##### Statement Coverage

The following test cases were able to achieve 100% statement coverage.

* CompareTo\_Test2: xValue = 13, yValue = 12
* CompareTo\_Test4: xValue = -20428423987, yValue = 2000
* CompareTo\_Test6: xValue = 2923942394, yValue = 20

##### Branch Decision Coverage

The following test cases were able to achieve 100% branch decision coverage.

* CompareTo\_Test1: xValue = 12, yValue = 13
* CompareTo\_Test2: xValue = 13, yValue = 12
* CompareTo\_Test3: xValue = 50, yValue = 50
* CompareTo\_Test4: xValue = -20428423987, yValue = 2000
* CompareTo\_Test5: xValue = 1897987979798, yValue = -31290
* CompareTo\_Test6: xValue = 2923942394, yValue = 20
* CompareTo\_Test7: xValue = -2923942394, yValue = -20
* CompareTo\_Test8: xValue = 20, yValue = 1897987979798
* CompareTo\_Test9: xValue = 1897987979798, yValue = 1897987979799

##### Condition Coverage

The following test cases were able to achieve 100% condition coverage.

* CompareTo\_Test1: xValue = 12, yValue = 13
* CompareTo\_Test2: xValue = 13, yValue = 12
* CompareTo\_Test3: xValue = 50, yValue = 50
* CompareTo\_Test4: xValue = -20428423987, yValue = 2000
* CompareTo\_Test5: xValue = 1897987979798, yValue = -31290
* CompareTo\_Test6: xValue = 2923942394, yValue = 20
* CompareTo\_Test7: xValue = -2923942394, yValue = -20
* CompareTo\_Test8: xValue = 20, yValue = 1897987979798
* CompareTo\_Test9: xValue = 1897987979798, yValue = 1897987979799
* CompareTo\_Test10: xValue = 1897987979798, yValue = 1897987979798
* CompareTo\_Test11: xValue = 1897987979799, yValue = 1897987979798

##### Condition/Decision Coverage

The following test cases were able to achieve 100% condition/decision coverage.

* CompareTo\_Test1: xValue = 12, yValue = 13
* CompareTo\_Test2: xValue = 13, yValue = 12
* CompareTo\_Test3: xValue = 50, yValue = 50
* CompareTo\_Test4: xValue = -20428423987, yValue = 2000
* CompareTo\_Test5: xValue = 1897987979798, yValue = -31290
* CompareTo\_Test6: xValue = 2923942394, yValue = 20
* CompareTo\_Test7: xValue = -2923942394, yValue = -20
* CompareTo\_Test8: xValue = 20, yValue = 1897987979798
* CompareTo\_Test9: xValue = 1897987979798, yValue = 1897987979799
* CompareTo\_Test10: xValue = 1897987979798, yValue = 1897987979798
* CompareTo\_Test11: xValue = 1897987979799, yValue = 1897987979798

##### Multiple Condition Coverage

The following test cases were able to achieve 100% of the feasible multiple condition coverage.

* CompareTo\_Test1: xValue = 12, yValue = 13
* CompareTo\_Test2: xValue = 13, yValue = 12
* CompareTo\_Test3: xValue = 50, yValue = 50
* CompareTo\_Test4: xValue = -20428423987, yValue = 2000
* CompareTo\_Test5: xValue = 1897987979798, yValue = -31290
* CompareTo\_Test6: xValue = 2923942394, yValue = 20
* CompareTo\_Test7: xValue = -2923942394, yValue = -20
* CompareTo\_Test8: xValue = 20, yValue = 1897987979798
* CompareTo\_Test9: xValue = 1897987979798, yValue = 1897987979799
* CompareTo\_Test10: xValue = 1897987979798, yValue = 1897987979798
* CompareTo\_Test11: xValue = 1897987979799, yValue = 1897987979798

### JUnit Test Cases

The test cases for the CompareTo method can be found in src/Tests/StructuralTestingCompareTo.java. Each test comprises of three Big Integer values xValue, yValue and expectedResult which are tested using the JUnit assertEquals testing method. These test cases result in 100% of the feasible coverage for all the required code coverage methods.

## Task 3: White-box Testing: Data Flow Testing

### Method 1: public BigInteger gcd(BigInteger y)



Figure 3: Task 3 GCD CFG

#### Identify all the definition-use pairs (du-pairs)

Identifying du-pairs – variable **xval**:

all-defs: 1, 8, 9

all-uses:3, <3,5>, <3,6>, 6, <6,7>, <6,8>, 7, <7,9>, <7,10>, 9 , 12 , 16}

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1, 9) | <1,2,3,6,7,9> |
| (1,12) | <1,2,3,6,7,10,12> |
|  | <1,2,3,6,7,10,11,12> |
| (1,16) | <1,2,4,16> |
|  | <1,2,4,13,15,16> |
| (1,3) | <1,2,3> |
| (1, <3,5>) | <1,2,3,5> |
| (1, <3,6>) | <1,2,3,6> |
| (1,6) | <1,2,3,6> |
| (1, <6,7>) | <1,2,3,6,7> |
| (1, <6,8>) | <1,2,3,6,8> |
| (1,7) | <1,2,3,6,7> |
| (1, <7,9>) | <1,2,3,6,7,9> |
| (1, <7,10>) | <1,2,3,6,7,10> |
| (8, 16) | <8,4,16> |
|  | <8,4,13,15,16> |
| (9,12) | <9,10,12> |
|  | <9,10,11,12> |

Identifying du-pairs – variable **yval**:  
  
all-defs:1,11,15  
all-uses: 6, <6,7>, <6,8>, 10, <10,11>, <10,12>, 11,12, 13, <13,14>, <13,15> ,16}

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,11) | <1,2,3,6,7,10,11> |
|  | <1,2,3,6,7,9,10,11> |
| (1,12) | <1,2,3,6,7,10,12> |
|  | <1,2,3,6,7,9,10,12> |
| (1,16) | <1,2,4,16> |
|  | <1,2,3,6,8,4,16> |
| (1,6) | <1,2,3,6> |
| (1, <6,7>) | <1,2,3,6,7> |
| (1, <6,8>) | <1,2,3,6,8> |
| (1, 10) | <1,2,3,6,7,10> |
|  | <1,2,3,6,7,9,10> |
| (1, <10,11>) | <1,2,3,6,7,10,11> |
|  | <1,2,3,6,7,9,10,11> |
| (1, <10,12>) | <1,2,3,6,7,10,12> |
|  | <1,2,3,6,7,9,10,12> |
| (1,13) | <1,2,4,13> |
|  | <1,2,3,6,8,4,13> |
| (1, <13,14>) | <1,2,4,13,14> |
|  | <1,2,3,6,8,4,13,14> |
| (1, <13,15>) | <1,2,4,13,15> |
|  | <1,2,3,6,8,4,13,15> |
| (11,12) | <11,12> |
| (15,16) | <15,16> |

Identifying du-pairs – variable **words**:  
  
all-defs: 1  
all-uses:2, <2,3>, <2,4>

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,2) | <1,2> |
| (1, <2,3>) | <1,2,3> |
| (1, <2,4>) | <1,2,4> |

Identifying du-pairs – variable **y.words**:  
  
all-defs: 1  
all-uses:4, 6, <6,7>, <6,8>

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,4) | <1,2,4> |
| (1,6) | <1,2,3,6> |
| (1, <6,7>) | <1,2,3,6,7> |
| (1, <6,8>) | <1,2,3,6,8> |

Identifying du-pairs – variable **len**:  
  
all-defs: 16  
all-uses:16

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (16, 16) | <16> |

Identifying du-pairs – variable **result**:  
  
all-defs: 16  
all-uses:16,17

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (16,16) | <16> |
| (16, 17) | <16,17> |

Identifying du-pairs – variable **xwords**:  
  
all-defs: 16  
all-uses:16

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (16, 16) | <16> |

Identifying du-pairs – variable **ywords**:  
  
all-defs: 16  
all-uses:16

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (16, 16) | <16> |

#### Design test cases to achieve All-Defs coverage

t1 path: <1,2,3,6,7,9,10,11,12> (covered definition node: 1, 9, 11)  
 t2 path: <1,2,3,6,8,4,13,15,16,17>(covered definition node: 1, 8, 15, 16, 17)  
  
 test case t1 : (x = -8, y = -12) expect outcome: 4  
 test case t2: (x = 24, y =-2147483648) expect outcome: 8  
  
 test cases t1 and t2 will achieve all-defs coverage.

#### Design test cases to achieve All-Uses coverage

t1: <1,2,3,6,7,9,10,11,12>

t2: <1,2,3,6,7,10,12>

t3: <1,2,3,6,7,9,10,12>

t4: <1,2,3,6,7,10,11,12>

t5: <1,2,3,6,8,4,13,15,16,17>

t6: <1,2,3,5>

t7: <1,2,3,6,8,4,13,14>

t8: <1,2,4,13,15,16,17>

t9: <1,2,4,16,17>

t10: <1,2,4,13,14>

t11: <1,2,3,6,8,4,16,17>

considerate the test cases executing paths that will achieve All-Uses coverage

t1: (x = -8, y = -12) expect outcome: 4

t2: (x = 8, y = 12) expect outcome: 4

t3: (x = -5, y = 25) expect outcome: 5

t4: (x = 3, y = -9) expect outcome: 3

t5: (x = 24, y =-2147483648) expect outcome: 8

t6: (x = 0, y = 100) expect outcome: 100

t7: (x = -2147483648, y =0) expect outcome: 2147483648

t8: (x= 68719476751, y = 23) expect outcome: 23

t9: (x= 695784701952, y = 36590037911583) expect outcome: 3

t10: (x = 1039382085632, y =0) expect outcome: 1039382085632

t11: (x = 11583, y =36590037911583) expect outcome: 39

#### Write and execute the test cases in JUnit.

See src/Tests DataFlowTesting\_gcd.java

### Method 2: private static int compareTo(BigInteger x, BigInteger y)



Figure 4: Task 3 compareTo CFG

#### Identify all the definition-use pairs (du-pairs)

Identifying du-pairs – variable **x**:  
  
all-defs: 1  
all-uses: 2, <2,3>, 3 , <2,4>, 4 , 7, 10

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,2) | <1,2> |
| (1, <2,3>) | <1,2,3> |
| (1,3) | <1,2,3> |
| (1, <2,4>) | <1,2,4> |
| (1,4) | <1,2,4> |
| (1,7) | <1,2,4,5,7> |
| (1,10) | <1,2,4,5,7,8,10> |

Identifying du-pairs – variable **y:**  
  
all-defs: 1  
all-uses: 2, <2,3>, 3, <2,4>, 4, 7, 10

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,2) | <1,2> |
| (1, <2,3>) | <1,2,3> |
| (1,3) | <1,2,3> |
| (1, <2,4>) | <1,2,4> |
| (1,4) | <1,2,4> |
| (1,7) | <1,2,4,5,7> |
| (1,10) | <1,2,4,5,7,8,10> |

Identifying du-pairs – variable **x.words:**  
  
all-defs: 1  
all-uses: 2, <2,3>, <2,4>, 7, 10

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,2) | <1,2> |
| (1, <2,3>) | <1,2,3> |
| (1, <2,4>) | <1,2,4> |
| (1,4) | <1,2,4> |
| (1,7) | <1,2,4,5,7> |
| (1,10) | <1,2,4,5,7,8,10> |

Identifying du-pairs – variable **y.words:**  
  
all-defs: 1  
all-uses: 2, <2,3>, <2,4>, 7, 10

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (1,2) | <1,2> |
| (1, <2,3>) | <1,2,3> |
| (1, <2,4>) | <1,2,4> |
| (1,4) | <1,2,4> |
| (1,7) | <1,2,4,5,7> |
| (1,10) | <1,2,4,5,7,8,10> |

Identifying du-pairs – variable **x\_negative:**  
  
all-defs: 4  
all-uses: 4,5, <5,6>, 6, <5,7>,9

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (4,4) | <4> |
| (4,5) | <4,5> |
| (4, <5,6>) | <4,5,6> |
| (4, 6) | <4,5,6> |
| (4, <5,7>) | <4,5,7> |
| (4,9) | <4,5,7,8,9> |

Identifying du-pairs – variable **y\_negative:**  
  
all-defs: 4  
all-uses: 4,5, <5,6>, <5,7>,9

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (4,4) | <4> |
| (4,5) | <4,5> |
| (4, <5,6>) | <4,5,6> |
| (4, <5,7>) | <4,5,7> |
| (4,9) | <4,5,7,8,9> |

Identifying du-pairs – variable **x\_len:**   
  
all-defs: 7  
all-uses: 8, <8,9>, 9, <8,10>, 10

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (7,8) | <7,8> |
| (7, <8,9>) | <7,8,9> |
| (7,9) | <7,8,9> |
| (7, <8,10> | <7,8,10> |
| (7,10) | <7,8,10> |

Identifying du-pairs – variable **y\_len:**   
  
all-defs: 7  
all-uses: 9,8, <8,9>, 9, <8,10>

|  |  |
| --- | --- |
| Du-pair | Path(s) |
| (7,8) | <7,8> |
| (7, <8,9>) | <7,8,9> |
| (7,9) | <7,8,9> |
| (7, <8,10> | <7,8,10> |

#### Design test cases to achieve All-Defs coverage

t1 path: <1,2,4,5,7,8,10> (coverage of all definition nodes: 1,4,7)

test case 1 : x= BigInteger(2147483649) , y = BigInteger(2147483648) expect result: 1

#### Design test cases to achieve All-Uses coverage

t1 path:<1,2,3>

t2 path:<1,2,4,5,6>

t3 path:<1,2,4,5,7,8,9>

t4 path:<1,2,4,5,7,8,10>

considerate the test cases executing paths listed above that will achieve All-Uses coverage

test case t1: x= 88 , y = 99 expected result: -1

test case t2: x = BigInteger(“-2147483649”) , BigInteger(“2147483648”) expect result: -1

test case t3: x = BigInteger(“8888”), BigInteger (“2147483648”) expect result: -1

test case t4: x= BigInteger (“2147483648”) y=BigInteger(“2147483648”) expect result: 0

#### Write and execute the test cases in JUnit.

See src/Tests DataFlowTesting\_compareTo.java

## Conclusion

The report contains all the related test data. It contains the outputs and test specific data such as equivalence partitions and du-pairs. And it contains any results and findings revealed through these tests.

## Group Member Contributions

|  |  |  |
| --- | --- | --- |
| **Tasks** | **Date** | **Member** |
| Task2: public BigInteger gcd(BigInteger y) | 29/04/2022 | Kyle Beattie |
| Task3: White-box Testing: DataFlow Testing private static int compareTo(BigInteger x, BigInteger y) | 28/04/2022 |
| Final report formatting | 29/04/2022 |
|  | | |
| Initial Required classes of the entire Java project | 01/04/2022 | Ni Zeng |
| Initial test report template | 01/04/2022 |
| Task3: White-box Testing: DataFlow Testing public BigInteger gcd(BigInteger y) | 28/04/2022 |
| Task3: White-box Testing: DataFlow Testing private static int compareTo(BigInteger x, BigInteger y) | 28/04/2022 |
| Initial Required classes of the entire Java project | 01/04/2022 |
|  | | |
| Task1 part1: complete code | 17/04/22 | Brandon Allen |
| Task1 part2: complete code | 26/04/22 |
| Task1 part1 and 2: report dot point outline | 27/04/22 |
|  | | |
| Task 1 Blackbox Testing: public int compareTo(BigInteger val) code and report writeup | 23/04/22 | Austin Baxter |
| Task 2 White-box Testing Structural Testing public static int compareTo(BigInteger x, BigInteger y) code and report writeup | 25/04/22 |
|  |  |
|  |  |
|  | | |