SENG3320 Assignment 1 Report

By Lachlan Higgins (c3374994), Matthew Clarke (c3380302), Sean Nagel (c3356603)

Test Environment: JUnit5

Table of Contents

Task 1. Black Box Testing

Completed by Lachlan Higgins (c3374994)

Public BigInteger(int signum, byte[] magnitude)

Equivalence Partitioning

In this constructor, there are two variables to account for, signum, and magnitude. The following equivalence classes can be made for signum:

Signum < -1	**Signum = [1, 1]**	**Signum > 1**
Invalid	Valid	Invalid

Beyond these equivalence classes, there are two more conditions that must be met.

Firstly, if signum = -1, or if signum = 1, the magnitude must contain 1 or more non-zero integers.

Otherwise, if signum = 0, magnitude must contain no non-zero integers.

Boundary Value Analysis

Based on equivalence partitioning for the function BigInteger(int signum, byte[] magnitude), the following test cases must be accounted for:

1. Signum: Lower-bounded outer limit
2. Signum: -1
3. Signum: +1
4. Signum: Upper-bounded outer limit
5. Signum: 0 (with non-zero magnitude)
6. Signum: 0 (with zero magnitude)
7. Signum: nonzero (with zero magnitude)

To meet this, the following test cases were designed:

1. BigInteger(-2, 12345): **Invalid**
2. BigInteger(-1, 12345): **Valid**
3. BigInteger(1, 12345): **Valid**
4. BigInteger(2, 12345): **Invalid**
5. BigInteger(0, 12345): **Invalid**
6. BigInteger(0, 00000): **Valid**
7. BigInteger(1, 00000): **Invalid**

Public BigInteger(String val, int radix)

Equivalence Partitioning

In this constructor, there are two variables to account for, val, and radix. The radix minimum and maximum is dependent on the String input. The radix must account for the amount of individual digits used.

As 32-bit integers are used in the BigInteger class, there is a hard limit of [2,36], however the actual MIN_RADIX and MAX_RADIX will vary depending on the String. For example, the String “010101” can be converted to binary, therefore a radix of 2 can be used. Conversely, if the String “123ABC” was input, the minimum radix would change to 13, as there are 13 possible characters available from: {0,1,2,3,4,5,6,7,8,9,A,B,C}

In the following equivalence partitions, Radix_MIN and Radix_MAX will be used, rather than 2 and 36, respectively.

Radix < MIN_RADIX	**MIN_RADIX <= Radix <= MAX_RADIX**	**Radix > MAX_RADIX**
Invalid	Valid	Invalid

Boundary Value Analysis

Based on the above equivalence partitions, the following test cases must be accounted for:

1. Radix < MIN_RADIX (Global)
2. Radix = MIN_RADIX (Global)
3. Radix < MIN_RADIX (Local)
4. Radix = MIN_RADIX (Local)
5. Radix > Radix_MIN && Radix < Radix_MAX
6. Radix = Radix_MAX
7. Radix > Radix_MAX

For these test cases, various val’s were use, so that global and local minimums can be tested.

1. BigInteger("010101", 1): Invalid
2. BigInteger("010101, 2): Valid
3. BigInteger(“123ABC”, 12): Invalid
4. BigInteger(“123ABC”, 13): Valid
5. BigInteger(“123ABC”, 24): Valid
6. BigInteger("123ABC, 36): Valid
7. BigInteger("123ABC, 37): Invalid

Public BigInteger compareTo(BigInteger val)

Equivalence Partitioning

In this method, the values of both the object that calls, and the object that is called must be accounted for. However, all BigInteger values will are valid. Because of this, there are two equivalence classes.

Either value is non-BigInteger	**Both values are BigInteger,** X>y, x=y, x<y
Invalid	Valid

Boundary Value Analysis

Based on equivalence partitioning for the function Public BigInteger compareTo(BigInteger val), the following test cases must be accounted for:

1. X<y: true
2. X<y: false
3. X=y: true
4. X=y: false
5. X>y: true
6. X>y: false

To meet this, the following test cases were designed:

1. 1<10: True: valid
2. 10<1: False: valid
3. 1=1: true: valid
4. 1=10: false: Valid
5. 10>1: false: valid
6. 1>10: false: valid

Task 2. Whitebox Testing: Structural Testing

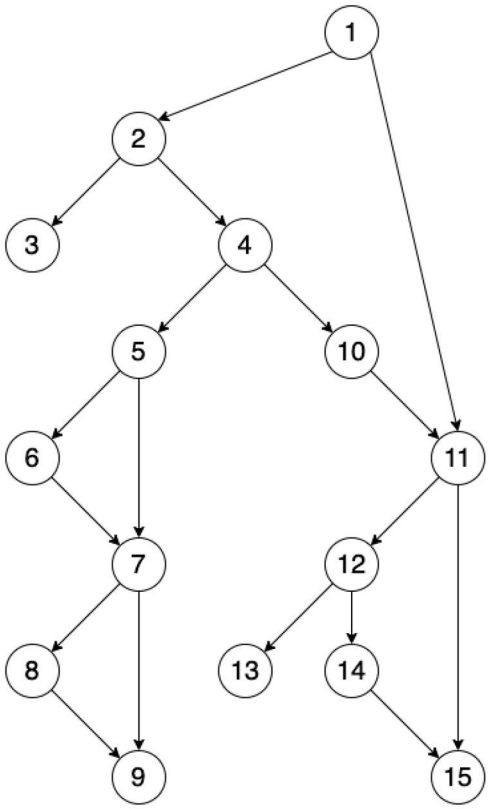
Testing objectives: Whitebox structural testing is being done to verify the flow of the structure, and that it works properly. It is also done to check the conditionality of loops and overall functionality.

public BigInteger gcd(BigInteger y)

Completed by Matthew Clarke (c3380302)

Control Flow Graph.

```
public BigInteger gcd(BigInteger y)
1229: public BigInteger gcd(BigInteger y)
1230: {
1231: int xval = ival;
1232: int yval = y.ival;
1233: if (words == null)
1234: {
1235: if (xval == 0)
1236: return abs(y);
1237: if (y.words == null
1238: && xval != Integer.MIN_VALUE && yval != Integer.MIN_VALUE)
1239: {
1240: if (xval < 0)
1241: xval = -xval;
1242: if (yval < 0)
1243: yval = -yval;
1244: return valueOf(gcd(xval, yval));
1245: }
1246: xval = 1;
1247: }
1248: if (y.words == null)
1249: {
1250: if (yval == 0)
1251: return abs(this);
1252: yval = 1;
1253: }
1254: int len = (xval > yval ? xval : yval) + 1;
1255: int[] xwords = new int[len];
1256: int[] ywords = new int[len];
1257: getAbsolute(xwords);
1258: y.getAbsolute(ywords);
1259: len = MPN.gcd(xwords, ywords, len);
1260: BigInteger result = new BigInteger(0);
1261: result.ival = len;
1262: result.words = xwords;
1263: return result.canonicalize();
1264: }
```



For all test cases, there are two BigInteger variables (‘x’ and ‘y’).

100% Statement Coverage

To achieve 100% statement coverage, at least four test cases are required:

1. To cover nodes 1, 2 and 3, the test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’.
2. To cover nodes 1, 2, 4, 5, 6, 7, 8 and 9, the test values ‘x=-6’ and ‘y=-8’ can be used, which should return a result of ‘2’.
3. To cover nodes 1, 2, 4, 10, 11, 12 and 13, the test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’.
4. To cover nodes 1, 11, 12, 14 and 15, the test values ‘x=12345678987654321’ and ‘y=5’ can be used, which should return a result of ‘1’.

100% Branch Decision Coverage

To achieve 100% branch decision coverage, at least six test cases are required (the same four from above can be used here):

1. To cover edges (1,2) and (2,3), the test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’.
2. To cover edges (1,2), (2,4), (4,5), (5,6), (6,7), (7,8) and (8,9), the test values ‘x=-6’ and ‘y=-8’ can be used, which should return a result of ‘2’.
3. To cover edges (1,2), (2,4), (4,10), (10,11), (11,12) and (12,13), the test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’.

4. To cover edges (1,11), (11,12), (12,14) and (14,15), the test values ‘x=12345678987654321’ and ‘y=5’ can be used, which should return a result of ‘1’.
5. To cover edges (1,2), (2,4), (4,5), (5,7) and (7,9), the test values ‘x=6’ and ‘y=8’ can be used, which should return a result of ‘2’.
6. To cover edges (1,11) and (11,15), the test values ‘x=12345678987654321’ and ‘y=98765432123456789’ can be used, which should return a result of ‘1’.

100% Condition Coverage

To achieve 100% condition coverage, at least seven test cases are required (the same six from above can be used here, except for test case 2). The only change is due to node 4, as this is the only decision which has more than one condition:

1. The test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’.
2. The test values ‘x=-2147483648’ and ‘y=-2147483648’ can be used, which should return a result of ‘2147483648’.
3. The test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’.
4. The test values ‘x=12345678987654321’ and ‘y=5’ can be used, which should return a result of ‘1’.
5. The test values ‘x=6’ and ‘y=8’ can be used, which should return a result of ‘2’.
6. The test values ‘x=12345678987654321’ and ‘y=98765432123456789’ can be used, which should return a result of ‘1’.
7. The test values ‘x=24’ and ‘y=12345678987654321’ can be used, which should return a result of ‘3’.

100% Condition/Decision Coverage

To achieve 100% condition/decision coverage, at least seven test cases are required. No change is required from the above seven, they are the same here:

1. The test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’.
2. The test values ‘x=-2147483648’ and ‘y=-2147483648’ can be used, which should return a result of ‘2147483648’.
3. The test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’.
4. The test values ‘x=12345678987654321’ and ‘y=5’ can be used, which should return a result of ‘1’.
5. The test values ‘x=6’ and ‘y=8’ can be used, which should return a result of ‘2’.
6. The test values ‘x=12345678987654321’ and ‘y=98765432123456789’ can be used, which should return a result of ‘1’.
7. The test values ‘x=24’ and ‘y=12345678987654321’ can be used, which should return a result of ‘3’.

Multiple Condition Coverage

It is impossible to achieve 100% multiple condition coverage. The best is 83.33% coverage, where at least nine test cases are required (the same seven from above can be used here). The extra two test cases come from node 4, which requires eight test cases (four of them are already covered, in test cases 2, (TFF), 3 (TFT), 5 (TTT), and 7 (FTT). The reason it is impossible is because there is no way for

‘y.words == null’ to be false and ‘yval != Integer.MIN_VALUE’ to be false at the same, meaning that the possibilities for the conditions to be ‘FTF’ or ‘FFF’ are impossible.

1. The test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’.
2. The test values ‘x=-2147483648’ and ‘y=-2147483648’ can be used, which should return a result of ‘2147483648’.
3. The test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’.
4. The test values ‘x=12345678987654321’ and ‘y=5’ can be used, which should return a result of ‘1’.
5. The test values ‘x=6’ and ‘y=8’ can be used, which should return a result of ‘2’.
6. The test values ‘x=12345678987654321’ and ‘y=98765432123456789’ can be used, which should return a result of ‘1’.
7. The test values ‘x=24’ and ‘y=12345678987654321’ can be used, which should return a result of ‘3’.
8. The test values ‘x=14’ and ‘y=-2147483648’ can be used, which should return a result of ‘2’.
9. The test values ‘2147483648’ and ‘y=1234567898765432’ can be used, which should return a result of ‘8’.

Private static int compareTo(BigInteger x, BigInteger y)

Completed by Sean Nagel (c3356603)

Control Flow Graph

383: private static int compareTo(BigInteger x, BigInteger y)
384: {

Node 1:
385: if (x.words == null && y.words == null)

Node 2:
386: return x.ival < y.ival ? -1 : x.ival > y.ival ? 1 : 0;

Node 3:
387: boolean x_negative = x.isNegative();
388: boolean y_negative = y.isNegative();

Node 4:
389: if (x_negative != y_negative)

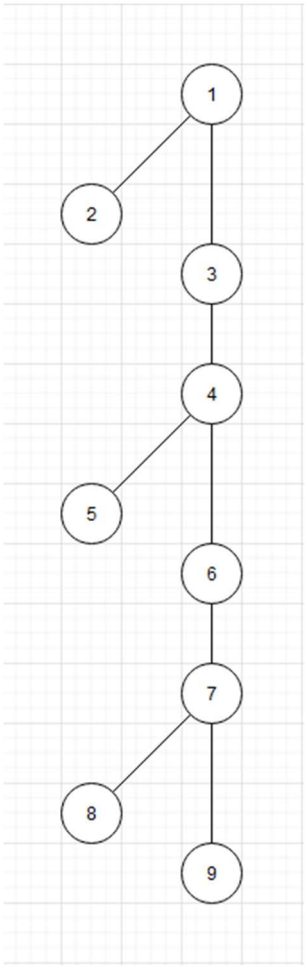
Node 5:
390: return x_negative ? -1 : 1;

Node 6:
391: int x_len = x.words == null ? 1 : x.ival;
392: int y_len = y.words == null ? 1 : y.ival;

Node 7:
393: if (x_len != y_len)

Node 8:
394: return (x_len > y_len) != x_negative ? 1 : -1;

Node 9:
395: return MPN.cmp(x.words, y.words, x_len);
396: }



100% Statement Coverage

To achieve 100% statement coverage for the compareTo() function, we need at least four test cases. These test cases will ensure that every statement in the function is executed at least once.

1. Test Case 1: x = 2, y = 4

This test case is checking the comparison between two BigIntegers where both x and y don't have their words array defined (Node 1). The compareTo() function should return -1 because 2 is less than 4.

2. Test Case 2: x = 5, y = 5

This test case is also hitting Node 1 as the words array is undefined for both x and y. The compareTo() function should return 0 because 5 equals 5.

3. Test Case 3: x = -98765432109876543210, y = 98765432109876543211

This test case covers Node 4. Both x and y are BigIntegers, and x is negative while y is positive. The function should return -1, indicating that x is less than y.

4. Test Case 4: x = 12345678901234567890123 4, y = 12345678901234567890

This test case covers Node 7 and Node 8. Both x and y are BigIntegers, and x_len is greater than y_len. The function should return 1, indicating that x is greater than y.

100% Branch Decision

To achieve 100% branch decision coverage for the compareTo() function, we need at least seven test cases. These test cases will ensure that each decision in the function has been executed in both true and false directions.

1. Test Case 1: x = 2, y = 4

This test case is checking the comparison between two BigIntegers where both x and y don't have their words array defined (Node 1). The compareTo() function should return -1 because 2 is less than 4.

2. Test Case 2: x = 4, y = 2

This is still targeting Node 1, but this time x is greater than y, so compareTo() should return 1.

3. Test Case 3: x = 5, y = 5

This test case also hits Node 1. The compareTo() function should return 0 because 5 equals 5.

4. Test Case 4: x = -98765432109876543210, y = 98765432109876543211

This test case covers Node 4. Both x and y are BigIntegers, and x is negative while y is positive. The function should return -1, indicating that x is less than y.

5. Test Case 5: x = 98765432109876543211, y = -98765432109876543210

This is another test case for Node 4. In this case, x is positive while y is negative, so compareTo() should return 1.

6. Test Case 6: x = 123456789012345678901234, y = 12345678901234567890

This test case covers Node 7 and Node 8. Both x and y are BigIntegers, and x_len is greater than y_len. The function should return 1, indicating that x is greater than y.

7. Test Case 7: x = 12345678901234567890, y = 123456789012345678901234

This test case also targets Node 7 and Node 8. In this case, x_len is less than y_len, so compareTo() should return -1.

100% Condition Coverage

To achieve 100% condition coverage for the compareTo() function, we need at least five test cases. These test cases will ensure that each individual condition in a decision has been executed for both true and false outcomes.

1. Test Case 1: x = 2, y = 4

This test case is checking the comparison between two BigIntegers where both x and y don't have their words array defined (Node 1). The compareTo() function should return -1 because 2 is less than 4.

2. Test Case 2: x = 4, y = 2

This is still targeting Node 1, but this time x is greater than y, so compareTo() should return 1.

3. Test Case 3: x = -98765432109876543210, y = 98765432109876543211

This test case covers Node 4. Both x and y are BigIntegers, and x is negative while y is positive. The function should return -1, indicating that x is less than y.

4. Test Case 4: x = 98765432109876543211, y = -98765432109876543210

This is another test case for Node 4. In this case, x is positive while y is negative, so compareTo() should return 1.

5. Test Case 5: x = 12345678901234567890, y = 12345678901234567890123

This test case also targets Node 7 and Node 8. In this case, x_len is less than y_len, so compareTo() should return -1.

100% Condition/Decision Coverage

To achieve 100% condition/decision coverage for the compareTo() function, we need at least seven test cases. These test cases will ensure that each decision in the function has been executed in both true and false directions and that each condition has been evaluated to both true and false.

1. Test Case 1: x = 2, y = 4

This test case is checking the comparison between two BigIntegers where both x and y don't have their words array defined (Node 1). The compareTo() function should return -1 because 2 is less than 4.

2. Test Case 2: x = 4, y = 2

This is still targeting Node 1, but this time x is greater than y, so compareTo() should return 1.

3. Test Case 3: x = 5, y = 5

This test case also hits Node 1. The compareTo() function should return 0 because 5 equals 5.

4. Test Case 4: x = -98765432109876543210, y = 98765432109876543211

This test case covers Node 4. Both x and y are BigIntegers, and x is negative while y is positive. The function should return -1, indicating that x is less than y.

5. Test Case 5: x = 98765432109876543211, y = -98765432109876543210

This is another test case for Node 4. In this case, x is positive while y is negative, so compareTo() should return 1.

6. Test Case 6: x = 123456789012345678901234, y = 12345678901234567890

This test case covers Node 7 and Node 8. Both x and y are BigIntegers, and x_len is greater than y_len. The function should return 1, indicating that x is greater than y.

7. Test Case 7: x = 12345678901234567890, y = 123456789012345678901234

This test case also targets Node 7 and Node 8. In this case, x_len is less than y_len, so compareTo() should return -1.

100% Multiple Condition Coverage

To achieve 100% multiple condition coverage for the compareTo() function, we need at least seven test cases. These test cases will ensure that each possible combination of conditions in a decision has been covered.

1. Test Case 1: x = 2, y = 4

This test case is checking the comparison between two BigIntegers where both x and y don't have their words array defined (Node 1). The compareTo() function should return -1 because 2 is less than

2. Test Case 2: x = 4, y = 2

This is still targeting Node 1, but this time x is greater than y, so compareTo() should return 1.

3. Test Case 3: x = 5, y = 5

This test case also hits Node 1. The compareTo() function should return 0 because 5 equals 5.

4. Test Case 4: x = -98765432109876543210, y = 98765432109876543211

This test case covers Node 4. Both x and y are BigIntegers, and x is negative while y is positive. The function should return -1, indicating that x is less than y.

5. Test Case 5: x = 98765432109876543211, y = -98765432109876543210

This is another test case for Node 4. In this case, x is positive while y is negative, so compareTo() should return 1.

6. Test Case 6: x = 123456789012345678901234, y = 12345678901234567890

This test case covers Node 7 and Node 8. Both x and y are BigIntegers, and x_len is greater than y_len. The function should return 1, indicating that x is greater than y.

7. Test Case 7: x = 12345678901234567890, y = 123456789012345678901234

This test case also targets Node 7 and Node 8. In this case, x_len is less than y_len, so compareTo() should return -1.

Task 3. White-box Testing: Data Flow Testing

Testing objective: Whitebox data flow testing is done to explore the sequences of events in relation to variables, and when they are defined and when they are used. For example, if a variable is defined but never used, it does not need to be there.

public BigInteger gcd(BigInteger y)

Completed by Matthew Clarke (c3380302)

Definition-Use Pairs (DU-Pairs)

Variable	Node Defined	Node Used
x		13 (c-use)
xval	1, 6, 10	2 (p-use), 4 (p-use), 5 (p-use), 6 (c-use), 9 (c-use), 15 (c-use)
ival		1 (c-use)
words		1 (p-use)
y	1	3 (c-use)
yval	1, 8, 14	4 (p-use), 7 (p-use), 8 (c-use), 9 (c-use), 12 (p-use), 15 (c-use)
y.ival		1 (c-use)
y.words		4 (c-use), 11 (c-use)
len	15	15 (c-use)
xwords	15	15 (c-use)
ywords	15	15 (c-use)
result	15	15 (c-use)
result.ival	15	
result.words	15	

DU-Pairs

x	None
xval	(1, 6), (1, 9), (1, 15), (1, <2, 3>), (1, <2, 4>), (1, <4, 5>), (1, <4, 10>), (1, <5, 6>), (1, <5, 7>), (6, 9), (10, 15)

ival	None
words	None
y	(1, 3)
yval	(1, 8), (1, 9), (1, 15), (1, <4, 5>), (1, <4, 10>), (1, <7, 8>), (1, <7, 9>), (1, <12, 13>), (1, <12, 14>), (8, 9), (14, 15)
y.ival	None
y.words	None
len	None
xwords	None
ywords	None
result	None
result.ival	None
result.words	None

For all test cases, there are two BigInteger variables (‘x’ and ‘y’).

All-Defs Coverage

For All-Defs coverage:

1. ‘xval’ will need to have at least one def-clear path from every definition (1, 6 and 10). This means three test cases are needed.
2. ‘y’ will need to have at least one def-clear path from every definition (1). This means one test case is needed.
3. ‘yval’ will need to have at least one def-clear path from every definition (1, 8 and 14). This means three test cases are needed.

To cover all the variable definitions, the test cases that can be used are:

1. The test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’. This covers the definitions in node 1 for variables ‘xval’ and ‘y’. The path is <1, 2, 3>.
2. The test values ‘x=-6’ and ‘y=-8’ can be used, which should return a result of ‘2’. This covers the definition in node 6 for variable ‘xval’ and definition in node 8 for variable ‘yval’. The path is <1, 2, 4, 5, 6, 7, 8, 9>.
3. The test values ‘x=30’ and ‘y=-2147483648’ can be used, which should return a result of ‘2’. This covers the definition in node 10 for variable ‘xval’ and definition in node 14 for variable ‘yval’. The path is <1, 2, 4, 10, 11, 12, 14, 15>.

All-Uses Coverage

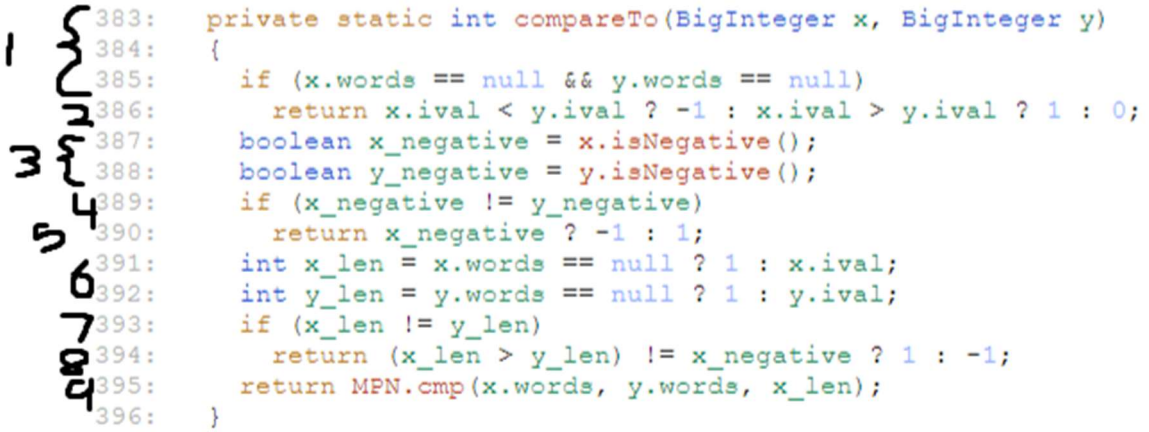
The test cases that can be used are:

1. The test values ‘x=0’ and ‘y=-5’ can be used, which should return a result of ‘5’. The path is <1, 2, 3>, meaning the du-pairs (1, <2, 3>) for ‘xval’ and (1, 3) for ‘y’ are covered.
2. The test values ‘x=-6’ and ‘y=-8’ can be used, which should return a result of ‘2’. The path is <1, 2, 4, 5, 6, 7, 8, 9>, meaning the du-pairs (1, 6), (1, <2, 4>), (1, <4, 5>), (1, <5, 6>) and (6, 9) for ‘xval’ and (1, 8), (1, <4, 5>), (1, <7, 8>) and (8, 9) for ‘yval’ are covered.
3. The test values ‘x=6’ and ‘y=8’ can be used, which should return a result of ‘2’. The path is <1, 2, 4, 5, 7, 9>, meaning the du-pairs (1, 9) and (1, <5, 7>) for ‘xval’ and (1, 9) and (1, <7, 9>) for ‘yval’ are covered.
4. The test values ‘x=-2147483648’ and ‘y=0’ can be used, which should return a result of ‘2147483648’. The path is <1, 2, 4, 10, 11, 12, 13>, meaning the du-pairs (1, <4, 10>) for ‘xval’ and (1, <4, 10>) and (1, <12, 13>) for ‘yval’ are covered.
5. The test values ‘x=24’ and ‘y=-2147483648’ can be used, which should return a result of ‘8’. The path is <1, 2, 4, 10, 11, 12, 14, 15>, meaning the du-pairs (10, 15) for ‘xval’ and (1, <12, 14>) and (14, 15) for ‘yval’ are covered.
6. The test values ‘x=12345678987654321’ and ‘y=98765432123456789’ can be used, which should return a result of ‘1’. The path is <1, 11, 15>, meaning the du-pairs (1, 15) for ‘xval’ and (1, 15) for ‘yval’ are covered.

Private static int compareTo(BigInteger x, BigInteger y)

By Lachlan Higgins (c3374994)

The following nodes will be used for this portion of Whitebox Testing



Definition-Use Pairs (DU-Pairs)

Variable	Node Defined	Node Used
X		1 (p-use), 2 (c-use), 6 (c-use), 9 (c-use)
Y		1 (p-use), 2 (c-use), 6 (c-use), 9 (c-use)
X_negative	3	4 (p-use), 5 (c-use), 8 (c-use)
Y_negative	3	4 (p-use)
X_len	6	7 (p-use), 8 (p-use), 9 (c-use)
Y_len	6	7 (p-use), 8 (p-use)

For the values of X.words, Y.words, X.ival and Y.ival, they are stored within X and Y, so they are defined with X and Y.

DU-Pairs

X	(1, 2), (1, 6), (1,9)
Y	(1, 2), (1,6), (1,9)
X_negative	(3,4), (3,5), (3,8)
Y_negative	(3,4)
X_len	(6,7), (6,8), (6,9)
Y_len	(6,7), (6,8)

DU-Pair and Paths for X

DU-PAIR	PATHS
(1,2)	<1,2>
(1,6)	<1,3,4,6>
(1,9)	<1,3,4,6,7,9>

DU-Pair and Paths for Y

DU-PAIR	PATHS
(1,2)	<1,2>
(1,6)	<1,3,4,6>
(1,9)	<1,3,4,6,7,9>

DU-Pair and Paths for X_negative

DU-PAIR	PATHS
(3,4)	<3,4>

(3,5)	<3,4,5>
(3,8)	<3,4,6,7,8>

DU-Pair and Paths for Y_negative

DU-PAIR	PATHS
(3,4)	<3,4>

DU-Pair and Paths for X_len

DU-PAIR	PATHS
(6,7)	<6,7>
(6,8)	<6,7,8>
(6,9)	<6,7,8,9>

DU-Pair and Paths for Y_len

DU-PAIR	PATHS
(6,7)	<6,7>
(6,8)	<6,7,8>

All-Defs Coverage

All variable definitions occur within nodes 1, 3 and 6. To achieve All-Defs coverage, one variable use per definition must be reached.

In the path <1,3,4,6,7,9>, X.words and Y.words are used in node 1, x_negative and y_negative are used in node 4, x.ival and y.ival are used in node 6, and x_len and y_len are used in node 7. Additionally, the path <1,3,4,6,7,8> would also work to reach all variable uses. To follow the latter path (<1,3,4,6,7,8>), the following conditions must be met with test variables:

- X.words != null || y.words != null

BigInteger must be storing number larger than a 32-bit integer, otherwise it is stored in BigInteger.ival

- X_negative == y.negative

Both X and y must be the same sign.

- X_len != y_len

X and y cannot be the same length.

Therefore, the following test cases are produced to achieve All-Defs coverage:

1. compareTo(23980057693048759662345 3462, 40857920715979775646363346 66): Expected Result: -1, Path: <1,3,4,6,7,8>

2. compareTo(40857920715979775646363346666, 23980057693048759662345346 2): Expected Result: 1, Path: <1,3,4,6,7,8>
3. compareTo(40857920715979775646363346666, 40857920715979775646363346666): Expected Result: 0, Path: <1,3,4,6,7,9>

All-Uses Coverage

To have All-Use coverage, every use must be covered, with at least one def. For this, all nodes must be reached, as every node include a variable use. This can be done efficiently by targeting all return statements.

To reach node 2, x.words and y.words must both be null. To achieve this, the BigInteger being stored must fit within a standard 32-bit integer. The following test cases travel along path <1,2>

1. compareTo(1, 1): Equal
2. compareTo(2, 1): Less than
3. compareTo(1, 2): Greater than

To reach node 5, both X and Y must be differently signed, and they must be larger than 32-bit integers. As they must be differently signed, there can be no test case for equality here. These will travel along path <1,3,4,5>.

4. compareTo(23980057693048759662345346 2, -40857920715979775646363346666): Expected result: 1
5. compareTo(-40857920715979775646363346666, 23980057693048759662345346 2): Expected result: -1

Next, node 8 must be reached. This is already achieved with test cases 1 and 2 of All-Defs coverage.

Finally, node 9 must be reached. To do this, all criteria for node 8 must be met, except that both BigIntegers must be the same length, rather than different. An equality test is already made in test case 3 of All-Defs coverage. These will take the path: <1,3,4,6,7,9>

6. compareTo(40857920715979775646363346665, 40857920715979775646363346666): Expected Result: -1
7. compareTo(40857920715979775646363346667, 40857920715979775646363346666): Expected Result: 1