SENG3320/6320 Software Verification and Validation
Semester 1, 2020

Assignment 1 – Test Case Design

100 marks (25% of the whole course assessment)
Due 11:59pm, Monday, 27 April 2020

1. Project Description

The BigInteger class is a Java math class. It is used for mathematical operations, which involve very big integer calculations that are outside the limit of all available primitive data types. For example, two big integers could be:
5454564684456454684646454545 and 42564564844646848648648648 64,
their sum is the big integer 971102116892113954951131940 9.

The API specification of this class is available at:
https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

The source code of this class is available at:
http://developer.classpath.org/doc/java/math/BigInteger-source.html

In this assignment, you will perform blackbox testing and whitebox testing for some functions of the BigInteger class.

Task 1: Blackbox Testing (21 marks)
Consider the following three methods in BigInteger class (their API specifications are shown in Appendix):

```
public BigInteger(int signum,byte[] magnitude)
public BigInteger(String val, int radix)
public int compareTo(BigInteger val)
```

For each method:
1) Design test cases using the Equivalence Partitioning and Boundary Value Analysis techniques. State clearly the equivalence classes. Clearly specify which partitions/classes is being tested, the corresponding test inputs, and the expected output. (15 marks)

2) Write and execute the test cases in JUnit. (6 marks)

Task 2: White-box Testing: Structural Testing (51 marks)
Consider the following three methods in BigInteger class (their source code is shown in Appendix):

```
public BigInteger gcd(BigInteger y)
public BigInteger modInverse(BigInteger y)
private static int compareTo(BigInteger x, BigInteger y) (Note that
this is a private method, you can test it through the public method public int
compareTo(BigInteger val), which only calls this method)
```

For each method:

1) Draw a control flow graph (3 marks)
2) Design test cases that can achieve 100% statement coverage (9 marks)
3) Design test cases that can achieve 100% branch decision coverage (when feasible) (9 marks)
4) Design test cases that can achieve 100% condition coverage (when feasible) (9 marks)
5) Design test cases that can achieve 100% condition/decision coverage (when feasible) (9 marks)
6) Design test cases that can achieve 100% multiple condition coverage (when feasible) (9 marks)
7) Write and execute the test cases in JUnit. (3 marks)

Task 3: White-box Testing: Data Flow Testing (18 marks)

Consider the following two methods in BigInteger class (their source code is shown in Appendix):

```
public BigInteger gcd(BigInteger y)
private static int compareTo(BigInteger x, BigInteger y) (Note that this is a private method, you can test it through the public method public int compareTo(BigInteger val), which only calls this method)
```

For each method:

1) Identify all the definition-use pairs (du-pairs) (6 marks)
2) Design test cases to achieve All-Defs coverage (4 marks)
3) Design test cases to achieve All-Uses coverage (6 marks)
4) Write and execute the test cases in JUnit. (2 marks)

Task 4: Test Report (10 marks)

A test report, in PDF format, should describe the test environment, test objective, test cases your designed for tasks 1-3, test execution results, and the test coverage you achieved. A ReadMe section should also be included in your test report, which describes the files and folders in your submission, and how to compile and run your tests. Please also document clearly the work performed by each group member.

The group project should be clear, informative, and well-structured.

2. Deliverables

This project will be a group project. Each group consists of 3-4 students.

This assignment should be submitted via Blackboard before 11:59pm, Monday, 27 April 2020. The submission should be a single zip file, which contains all source code, test cases, test results, a README text file, and the test report.

Copying, plagiarism and other malpractices are not allowed. Please remember to attach an Assignment Cover Sheet to each submission.

Start early and work efficiently and effectively!

- End –

Appendix:

public BigInteger(int signum,byte[] magnitude)

Translates the sign-magnitude representation of a BigInteger into a BigInteger. The sign is represented as an integer signum value: -1 for negative, 0 for zero, or 1 for positive. The magnitude is a byte array in *big-endian* byte-order: the most significant byte is in the zeroth element. A zero-length magnitude array is permissible, and will result in a BigInteger value of 0, whether signum is -1, 0 or 1.

Parameters:

`signum` - signum of the number (-1 for negative, 0 for zero, 1 for positive).

`magnitude` - big-endian binary representation of the magnitude of the number.

Throws:

`NumberFormatException` - `signum` is not one of the three legal values (-1, 0, and 1), or `signum` is 0 and `magnitude` contains one or more non-zero bytes.

public BigInteger(String val, int radix)

Translates the String representation of a BigInteger in the specified radix into a BigInteger. The String representation consists of an optional minus or plus sign followed by a sequence of one or more digits in the specified radix. The character-to-digit mapping is provided by `Character.digit`. The String may not contain any extraneous characters (whitespace, for example).

Parameters:

`val` - String representation of BigInteger.

`radix` - radix to be used in interpreting `val`.

Throws:

`NumberFormatException` - `val` is not a valid representation of a BigInteger in the specified radix, or `radix` is outside the range from `Character.MIN_RADIX` to `Character.MAX_RADIX`, inclusive.

public int compareTo(BigInteger val)

Compares this BigInteger with the specified BigInteger. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: `(x.compareTo(y)` <*op*> `0)`, where <*op*> is one of the six comparison operators.

Specified by:

`compareTo` in interface `Comparable<BigInteger>`

Parameters:

`val` - BigInteger to which this BigInteger is to be compared.

Returns:

-1, 0 or 1 as this BigInteger is numerically less than, equal to, or greater than `val`.

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:   }
```

public BigInteger modInverse(BigInteger y)

```
1098:   public BigInteger modInverse(BigInteger y)
1099:   {
1100:     if (y.isNegative() || y.isZero())
1101:       throw new ArithmeticException("non-positive modulo");
1102:
1103:     // Degenerate cases.
1104:     if (y.isOne())
1105:       return ZERO;
1106:     if (isOne())
1107:       return ONE;
1108:
1109:     // Use Euclid's algorithm as in gcd() but do this recursively
1110:     // rather than in a loop so we can use the intermediate results as we
1111:     // unwind from the recursion.
1112:     // Used http://www.math.nmsu.edu/~crypto/EuclideanAlgo.html as reference.
1113:     BigInteger result = new BigInteger();
1114:     boolean swapped = false;
1115:
1116:     if (y.words == null)
1117:       {
1118:     // The result is guaranteed to be less than the modulus, y (which is
1119:     // an int), so simplify this by working with the int result of this
1120:     // modulo y.  Also, if this is negative, make it positive via modulo
1121:     // math.  Note that BigInteger.mod() must be used even if this is
1122:     // already an int as the % operator would provide a negative result if
1123:     // this is negative, BigInteger.mod() never returns negative values.
1124:         int xval = (words != null || isNegative()) ? mod(y).ival : ival;
1125:         int yval = y.ival;
1126:
1127:     // Swap values so x > y.
```

```
1128:     if (yval > xval)
1129:       {
1130:         int tmp = xval; xval = yval; yval = tmp;
1131:         swapped = true;
1132:       }
1133:     // Normally, the result is in the 2nd element of the array, but
1134:     // if originally x < y, then x and y were swapped and the result
1135:     // is in the 1st element of the array.
1136:     result.ival =
1137:       euclidInv(yval, xval % yval, xval / yval)[swapped ? 0 : 1];
1138: 
1139:     // Result can't be negative, so make it positive by adding the
1140:     // original modulus, y.ival (not the possibly "swapped" yval).
1141:     if (result.ival < 0)
1142:       result.ival += y.ival;
1143:       }
1144:     else
1145:       {
1146:     // As above, force this to be a positive value via modulo math.
1147:     BigInteger x = isNegative() ? this.mod(y) : this;
1148: 
1149:     // Swap values so x > y.
1150:     if (x.compareTo(y) < 0)
1151:       {
1152:         result = x; x = y; y = result; // use 'result' as a work var
1153:         swapped = true;
1154:       }
1155:     // As above (for ints), result will be in the 2nd element unless
1156:     // the original x and y were swapped.
1157:     BigInteger rem = new BigInteger();
1158:     BigInteger quot = new BigInteger();
1159:     divide(x, y, quot, rem, FLOOR);
1160:         // quot and rem may not be in canonical form. ensure
1161:         rem.canonicalize();
1162:         quot.canonicalize();
1163:     BigInteger[] xy = new BigInteger[2];
1164:     euclidInv(y, rem, quot, xy);
1165:     result = swapped ? xy[0] : xy[1];
1166: 
1167:     // Result can't be negative, so make it positive by adding the
1168:     // original modulus, y (which is now x if they were swapped).
1169:     if (result.isNegative())
1170:       result = add(result, swapped ? x : y, 1);
1171:       }
1172: 
1173:     return result;
1174:   }
1175: 
```

private static int compareTo(BigInteger x,BigInteger y)

```
 383:   private static int compareTo(BigInteger x, BigInteger y)
 384:   {
 385:     if (x.words == null && y.words == null)
 386:       return x.ival < y.ival ? -1 : x.ival > y.ival ? 1 : 0;
 387:     boolean x_negative = x.isNegative();
 388:     boolean y_negative = y.isNegative();
 389:     if (x_negative != y_negative)
 390:       return x_negative ? -1 : 1;
 391:     int x_len = x.words == null ? 1 : x.ival;
 392:     int y_len = y.words == null ? 1 : y.ival;
 393:     if (x_len != y_len)
 394:       return (x_len > y_len) != x_negative ? 1 : -1;
 395:     return MPN.cmp(x.words, y.words, x_len);
 396:   }
```