MAT 3670: Lab 3
Bits, Data Types, and Operations

Background

In previous labs, we have used Turing machines to manipulate bit strings. In this lab, we will continue to focus on bit strings, placing more emphasis on the interpretation of these strings, as explained in Chapter 2 of our text.

Although Turing machines could be used for this week's lab exercises, you would probably agree that the design of these machines would be somewhat complicated, even though the algorithms themselves are fairly basic. To make life simpler, we will use a different way to implement these algorithms.

All of the algorithms for this lab are to be implemented in Java. Our goal is to experiment with and build various operations on bit strings. Thus, we will be building our bit string class from scratch.

Pre-Lab Exercises

During lab, you will be asked to implement many of the methods needed for a class that represents bit strings. To prepare yourself for this activity, do the following before this week's lab:

- Review all code from `BitString.java`.
- Read and understand what is required for the incomplete methods.
- Provide pseudo-code for each of the incomplete methods.
- As time allows, create implementations based on your pseudo-code. Do not use `Integer`, `BitSet`, or any other Java class to implement the missing methods. You may use the `Math` Class.

Laboratory Exercises

1. Descend to your 3670 directory; create a **lab3** directory to store the files you will create for this lab. Download `BitString.java` from the course website and place it in your **lab3** directory.
2. Open `BitString.java` with your favorite Java editor (such as NetBeans).
3. Provide implementations for the incomplete methods.
4. Add a `main` method for testing your methods.

Submissions

When you have completed the lab, submit your `lab3` folder by dragging it onto the EIU submission icon.

Contents of BitString.java

```
1   import java.util.Arrays;
2
3   /**
4    * This class implements an inefficient fixed size mutable binary string. The
5    * bits of a BitString are indexed by non-negative integers. Individual bits
6    * can be examined, set (made true), cleared (made false), or flipped (set to
7    * the opposite value). The logical AND, logical inclusive OR, and and other
8    * operations can be used to combine two BitStrings of the same size and to
9    * manipulate BitStrings. By default, all bits in the string initially are
10   * false. Unlike text strings, bit strings indexes increase from right to left
11   * i.e. the most significant bits have larger indexes. Do not use Integer,
12   * BitSet, or any other Java class to implement the missing methods. You may use
13   * the Math Class.
14   */
15  public class BitString implements Cloneable {
16    // An array to hold the bits that make up the bit string.
17    private boolean bits[];
18    /**
19     *  A constant that defines the size of the default bit string.
20     */
21    public static final int DEFAULT_SIZE = 8;
22
23    /**
24     * Creates a new, all false, bit string of the given size.
25     */
26    public BitString(int size) {
27      if(size < 1) throw new IllegalArgumentException("Size must be positive");
28      bits = new boolean[size];
29    }
30
31    /**
32     * Creates a new all false bit string of size DEFAULT_SIZE.
33     */
34    public BitString() {
35      this(DEFAULT_SIZE);
36    }
37
38    /**
39     * Creates a new bit string from the give text string of ones and zeros.
40     */
41     public BitString(String s) {
42       this(s.length());
43       //Read string from right to left
44       int index = s.length() - 1;
45       for (char digit : s.toCharArray())
46       {
47         switch(digit)
48         {
49           case '0':
50             bits[index] = false;
51             break;
52           case '1':
53             bits[index] = true;
54             break;
55           default:
```

```
56              throw new IllegalArgumentException("String " + s +
57                                                 "may only contain 0's and 1's.");
58          }
59          --index;
60        }
61      }
62
63      /**
64       * Creates a copy of the given bit string.
65       */
66      @Override
67      public Object clone() {
68        BitString copy = new BitString(bits.length);
69        copy.bits = Arrays.copyOf(bits, bits.length);
70        return copy;
71      }
72
73      /**
74       * Returns a representation of this BitString as a string of 1's and 0's.
75       */
76      @Override
77      public String toString() {
78        StringBuilder out = new StringBuilder(bits.length);
79
80        // For each bit append either a 1 or a 0 to out. Note that we are going
81        // from large indexes to small as the most significant bits have larger
82        // indexes.
83        for (int index = bits.length - 1; index >= 0; --index) {
84          out.append(bits[index] ? 1 : 0);
85        }
86
87        return out.toString();
88      }
89
90      /**
91       * Return the value of a bit string at the given index.
92       */
93      public boolean get(int index) {
94        return bits[index];
95      }
96
97      /**
98       * Set the value of a bit string at the given index to true.
99       */
100     public void set(int index) {
101       bits[index] = true;
102     }
103
104     /**
105      * Set the value of a bit string at the given index to false.
106      */
107     public void clear(int index) {
108       bits[index] = false;
109     }
110
111     /**
112      * Set the value of a bit string at the given index to the opposite value.
```

```
113      */
114     public void flip(int index) {
115       bits[index] = !bits[index];
116     }
117
118     /**
119      * Returns the number of bits in this bit string.
120      */
121     public int size() {
122       return bits.length;
123     }
124
125     /**
126      * Returns the number of true bits.
127      */
128     public int populationCount() {
129       int count = 0;
130
131       // For each true bit increment count.
132       for (boolean bit : bits) {
133         if (bit) {
134           ++count;
135         }
136       }
137
138       return count;
139     }
140
141     @Override
142     public int hashCode() {
143       int hash = 5;
144       hash = 29 * hash + Arrays.hashCode(this.bits);
145       return hash;
146     }
147
148     /**
149      * Two bit strings are equal if they have the same size and same bits.
150      */
151     @Override
152     public boolean equals(Object obj) {
153       if (!(obj instanceof BitString)) {
154         throw new IllegalArgumentException("obj must be a BitString");
155       }
156
157       return Arrays.equals(bits, ((BitString) obj).bits);
158     }
159
160     /**
161      * An object factory method that creates a bit string corresponding to the
162      * non-negative decimal value n, using size bits. If size is too small to
163      * hold n an InsuffisantNumberOfBitsException is thrown.
164      */
165     public static BitString decimalToUnsigned(int n, int size) {
166       throw new UnsupportedOperationException("This function needs to be completed!");
167     }
168
169     /**
```

```
170      * Turns the bit string into its binary successor. For example, the successor
171      * of 101011 is 101100. Returns reference to self for object chains.
172      */
173     public BitString successor() {
174       throw new UnsupportedOperationException("This function needs to be completed!");
175     }
176
177     /**
178      * Turns bit string into its binary complement. For example, the complement of
179      * 101011 is 010100. Returns reference to self for object chains.
180      */
181     public BitString complement() {
182       throw new UnsupportedOperationException("This function needs to be completed!");
183     }
184
185     /**
186      * Turns bit string into its two's complement. For example, the two's
187      * complement of 101011 is 010101. Returns reference to self for object
188      * chains.
189      */
190     public BitString twosComplement() {
191       throw new UnsupportedOperationException("This function needs to be completed!");
192     }
193
194     /**
195      * Returns the value of this bit string when interpreted as an unsigned
196      * decimal. For example, the string 111 has unsigned value 7.
197      */
198     public int unsignedValue() {
199       throw new UnsupportedOperationException("This function needs to be completed!");
200     }
201
202     /**
203      * Returns the value of this bit string when interpreted as a signed
204      * decimal with the MSB being the sign bit. For example, the string 111 has
205      * unsigned value -3.
206      */
207     public int signedValue() {
208       throw new UnsupportedOperationException("This function needs to be completed!");
209     }
210
211     /**
212      * Returns the value of this bit string when interpreted as a decimal in
213      * one's complement form. For example, in one's complement form the string
214      * 111 has the value 0.
215      */
216     public int onesComplementValue() {
217       throw new UnsupportedOperationException("This function needs to be completed!");
218     }
219
220     /**
221      * Returns the value of this bit string when interpreted as a decimal in
222      * two's complement form. For example, in two's complement form the string
223      * 111 has the value -1.
224      */
225     public int twosComplementValue() {
226       throw new UnsupportedOperationException("This function needs to be completed!");
```

```
227     }
228
229     /**
230      * Performs binary addition on bit strings of the same size. If they are
231      * not of the same size then a BitSizeMismatchException is thrown. Does not
232      * change the receiver.
233      */
234     public BitString add(BitString rightSummand) {
235       throw new UnsupportedOperationException("This function needs to be completed!");
236     }
237
238     /**
239      * Performs binary subtraction on bit strings of the same size. If they are
240      * not of the same size then a BitSizeMismatchException is thrown. Does not
241      * change the receiver.
242      */
243     public BitString subtract(BitString subtrahend) {
244       throw new UnsupportedOperationException("This function needs to be completed!");
245     }
246
247     /**
248      * Performs "logical and" on bit strings of the same size. If they are
249      * not of the same size then a BitSizeMismatchException is thrown. Does not
250      * change the receiver.
251      */
252     public BitString logicalAnd(BitString operand) {
253       throw new UnsupportedOperationException("This function needs to be completed!");
254     }
255
256     /**
257      * Performs "logical inclusive or" on bit strings of the same size. If they
258      * are not of the same size then a BitSizeMismatchException is thrown. Does not
259      * change the receiver.
260      */
261     public BitString logicalOr(BitString operand) {
262       throw new UnsupportedOperationException("This function needs to be completed!");
263     }
264
265     /**
266      * Performs a left shift by the given non negative amount. For example, the
267      * left shift of 01101 by 2 is 10100. Returns reference to self for object
268      * chains.
269      */
270     public BitString leftShift(int amount) {
271       throw new UnsupportedOperationException("This function needs to be completed!");
272     }
273
274     /**
275      * Performs a sign extension by the given non negative amount. For example,
276      * the extension of 10101 by 2 is 1110101. Returns reference to self for object
277      * chains.
278      */
279     public BitString signExtension(int amount) {
280       throw new UnsupportedOperationException("This function needs to be completed!");
281     }
282
283     /**
```

```
284      * A run time exception that should be thrown whenever a method does not have
285      * enough bits to successfully execute.
286      */
287     public static class InsuffisantNumberOfBitsException extends RuntimeException {}
288
289     /**
290      * A run time exception that should be thrown whenever two bit strings should
291      * be of the same size but are not.
292      */
293     public static class BitSizeMismatchException extends RuntimeException {}
294   }
```