You may work in pairs for this assignment. If you choose to work with a partner, make sure only one of you submits a solution, and you paste a copy of the Partners Template that contains the names and PIDs of both students at the beginning of each of the files you submit.

You will submit your solution to this assignment to the Curator System (as `HW01`). Your submission will consist of two files, described below. Place the two files in a flat Linux tar file (not a zip archive, gzip'd tar file or other type), and submit that.

Part I BST Algorithms

For questions 1 and 2, refer to the BST generic interface provided for Project 2. You may assume you have a complete, working implementation of all the methods shown in that interface, and you may make use of any of them (or their implied private helper methods) that you like. However, you may not assume any modifications to the interfaces of those methods.

Download the posted zip file and unpack it. See the Testing section at the end of this specification for further instructions.

Strive for the most efficient solution you can devise. For example, if it's possible to report the correct answer without performing a full traversal of the tree, then that's what your solution should do. You may not use any Java collections, including ones you implement yourself, to store any additional information (like copying the data elements from the BST into an array).

You may, and should, write private (or package-access) helper functions. Just be sure that your solutions do not depend on any code other than that in your version of `BSTUtils.java`, Java library classes, and the BST generic class.

Also: in addition to showing a Java representation of your solution to each of questions 1 and 2, you must write a short paragraph that provides a clear, complete explanation of the logic used in your algorithm. Our evaluation of your explanation will count 40% of the score for the question.

1. **[30 points]** Design an algorithm to determine whether a given binary tree contains two or more occurrences of the same value. (For this question we will assume the implementation of the BST does allow the insertion of duplicate values.) Express your solution using Java code as if it were to be implemented as a public member function (with private helper) belonging to the BST generic specified in Project 2; your answer (which will include both the public and private functions) must conform to the public interface shown below:

```
// Pre:      tree is a valid BST<Integer> object
// Post:     the BST is unchanged
// Returns:  true iff the BST contains two or more occurrences of the
//             same value
//
// Logic:    DESCRIBE THE LOGIC OF YOUR ALGORITHM HERE
//
public static boolean hasEqualValues( BST<Integer> tree ) {

   // up to you. . .
}
```

2. **[30 points]** Design an algorithm to count the number of occurrences of values that are larger than a given value, according to the `compareTo()` method for the stored data type. Express your solution using Java code as if it were to be implemented as a public member function (with private helper) belonging to the BST generic specified in Project 2; your answer (which will include both the public and private functions) must conform to the public interface shown below:

```
// Pre:       none
// Post:      the BST is unchanged
// Returns:   the number of nodes in the tree that contain a value
//              that is larger than floor
//
// Logic:     DESCRIBE THE LOGIC OF YOUR ALGORITHM HERE
//
public int numExceeding( Integer floor, BST<Integer> tree ) {

   // up to you. . .
}
```

Note: in grading each question in Part I, we will allocate 12 points for the quality and correctness of the description of your logic, 12 points for the correctness of your solution in testing, and 6 points for the efficiency of your solution.

Part II BST Performance

Prepare your answers to the following questions, either in a plain text file or a typed MS Word document. Partial credit will only be given if you show relevant work.

3. **[20 points]** Suppose the function $C(N)$, is given by the recurrence relation:

$$C(N) = \begin{cases} N & \text{if } N = 0,1 \\ 1 + C\left(\frac{N-1}{2}\right) & \text{if } N > 1 \end{cases}$$

Use Induction to prove: for every $N \geq 1$, $C(N) = \log(N+1)$.

4. **[20 points]** A *perfect* binary tree is one in which all the leaves are at the same level and every nonleaf node has two children. It can be shown that in a perfect binary tree with *k* levels, the number of nodes equals $2^k - 1$. (You do not have to prove that to answer this question.)

If a binary search tree T is perfect, then the function *C*(*N*) given in question 4 is related to cost of performing a particular operation that is likely to be carried out on T. Identify the relevant operation, explain how *C*(*N*) is related to the cost of that operation, and what the fact proved in question 4 tells us about the cost of that operation. Be precise.

Testing

The posted zip file includes testing code for your solutions to questions 1 and 2 in Part I. You will find the following files:

`testDriver.java`	test manager
`Spring2017/HW1/DS/BSTUtils.java`	shell for implementing solutions
`Spring2017/HW1/DS/BST.class`	64-bit Java implementation of BST as in J2
`Spring2017/HW1/DS/BST$BinaryNode.class`	

Edit `BSTUtils.java`, and complete the implementations of the two Java functions described in questions 1 and 2. Note that these are not member functions, but rather functions belonging to a class in the same package as the BST implementation. Therefore, these functions can access any members of the BST class that were declared with package protection.

Compile the code by executing "`javac testDriver.java`" in the directory where that file resides. That will also compile your BSTUtils.java file in the directory `./Spring2017/HW1/DS/`.

Note: I'm describing this using Linux directory notation, but the same commands work in the Windows shell as long as the JDK is properly installed.

The command "`java testDriver`" will run the testing code and create three log files, one for the testing details for each of the specified functions and one summarizing the results. Once you've run the testing code once, running it with the switch `-repeat` will reuse the same test data as on the previous run.