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 the file you submit.

You will submit your solution to this assignment to the Curator System (as `HW01`). Your solution must be either a plain text file (e.g., NotePad++) or a typed MS Word document; submissions in other formats will not be graded.

Partial credit will only be given if you show relevant work.

For questions 1 through 4, refer to the BST generic interface provided for Project 2. 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).

1. **[20 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 ignore anything the Project 2 specification says about allowing 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:      none
// Post:     the BST is unchanged
// Returns:  true iff the BST contains two or more occurrences of the
//             same value
// Logic:    If there are two occurrences of the same value, we would
//             see them in consecutive order when we perform an inorder
//             traversal of the tree.
//
public boolean hasEqualValues( ) {

   // Then traverse and compare previous to next element
   return hasEqualValues( root, null );
}

private boolean hasEqualValues(BinaryNode sRoot, T previous) {

   // No value here, so no match is possible here:
   if ( sRoot == null ) return false;

   // Compare current value to previous
   int comparison = sRoot.element.compareTo(previous);

   // See if a match occurs here:
   if ( comparison == 0 ) return true;

   return hasEqualValues(sRoot.left, sRoot.element) ||
          hasEqualValues(sRoot.right, sRoot.element);
}
```

2. **[20 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
//
public int numExceeding( T floor ) {

   return numExceeding(floor, root);
}

public int numExceeding( T floor, BinaryNode sRoot ) {

   // Nothing here, nothing to count...
   if ( sRoot == null ) return 0;

   // Count number of bigger values found; no so far.
   int biggerFound = 0;

   // Compare current element to floor.
   int comparisonHere = sRoot.element.compareTo(floor);

   // If current element > floor, there may be other bigger values
   //   in the left subtree of this node (otherwise, not).
   if ( comparisonHere > 0 ) {
      biggerFound = 1 + numExceeding(floor, sRoot.left);
   }

   // There may always be bigger values to the right of our
   //    current location.
   biggerFound += numExceeding(floor, sRoot.right);

   return biggerFound;
}
```

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 $k \geq 0$ if $N = 2^k - 1$ then $C(N) = \log(N+1)$.

If $k = 0$, then $N = 0$, so from the definition of $C(N)$, $C(0) = 0 = \log(0+1)$.
Now, suppose that for some integer $K \geq 0$, for every k such that $0 \leq k \leq K$,
$C(N) = C(2^k - 1) = \log(2^k - 1 + 1) = \log(2^k) = \log(N+1)$. Then let $k = K+1$,
so that $N = 2^{K+1} - 1$.

$$\begin{aligned} C(N) = C\left(2^{K+1} - 1\right) &= 1 + C\left(\frac{2^{K+1} - 1 - 1}{2}\right) \\ &= 1 + C\left(\frac{2^{K+1} - 2}{2}\right) = 1 + C\left(2^K - 1\right) \\ &= 1 + \log\left(2^K - 1 + 1\right) = 1 + \log\left(2^K\right) \\ &= 1 + K \\ &= \log(2^{K+1}) = \log(2^{K+1} - 1 + 1) \\ &= \log(N+1) \end{aligned}$$

Therefore, by Induction, for every $N \geq 0$, $C(N) = \log(N+1)$.

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

From the definition of $C(N)$, $C(0) = 0 = \log(0+1)$.
Now, suppose that for some integer $N \geq 1$, for every N such that $1 \leq K \leq N$,
$C(K) = \log(K+1)$. Then

$$\begin{aligned} C(N+1) = 1 + C\left(\frac{N+1-1}{2}\right) &= 1 + C\left(\frac{N}{2}\right) \\ &= 1 + \log\left(\frac{N}{2} + 1\right) = 1 + \log\left(\frac{N+2}{2}\right) \\ &= 1 + \log(N+2) - \log(2) \\ &= 1 + \log(N+2) - 1 \\ &= \log(N+2) \end{aligned}$$

Therefore, by Induction, for every $N \geq 0$, $C(N) = \log(N+1)$.

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.

Since at any node in a perfect BST, the two subtrees contain exactly the same number of nodes, the recurrence describes the number of comparison operations that would be performed if we searched a perfect BST for a value that either did not occur in the tree, or that occurred in a leaf node.

So, the result from question 4 proves that, in a perfect BST with N nodes, the worst case for the number of comparisons during a search is O(log(N+1)).