1. Here is a solution to `https://github.com/jnylam/SJSU-cs146-s17/blob/master/02_Recursion2/src/cc/jennylam/cs146/TowersOfHanoi.java`:

```java
public void solve() {
    moveStack(n, 0, 2);
}

public void moveStack(int n, int fromPeg, int toPeg) {
    if (n == 0) return;
    int otherPeg = 3 - fromPeg - toPeg; // sum of the peg indices is 0
       + 1 + 2 = 3
    moveStack(n-1, fromPeg, otherPeg);
    moveTopDisk(fromPeg, toPeg);
    moveStack(n-1, otherPeg, toPeg);
}

// (This part was given.)
public void moveTopDisk(int fromPeg, int toPeg) {
    int disk = pegs.get(fromPeg).pop();
    Stack<Integer> peg = pegs.get(toPeg);
    assert (peg.isEmpty() || peg.peek() > disk); // valid move
    peg.push(disk);
}
```

which produces the following output when $n = 4$:

```
|    ==    |          |          |
|   ====   |          |          |
|  ======  |          |          |
| ======== |          |          |

|          |          |          |
|   ====   |          |          |
|  ======  |          |          |
| ======== |    ==    |          |

|          |          |          |
|          |          |          |
|  ======  |          |          |
| ======== |    ==    |   ====   |

|          |          |          |
|          |          |          |
|  ======  |          |    ==    |
| ======== |          |   ====   |

|          |          |          |
|          |          |          |
|          |          |    ==    |
| ======== |  ======  |   ====   |

|          |          |          |
|          |          |          |
|    ==    |          |          |
| ======== |  ======  |   ====   |
```

```
|          |          |          |
|          |          |          |
|    ==    |   ====   |          |
| ======== |  ======  |          |

|          |          |          |
|          |    ==    |          |
|          |   ====   |          |
| ======== |  ======  |          |

|          |          |          |
|          |    ==    |          |
|          |   ====   |          |
|          |  ======  | ======== |

|          |          |          |
|          |          |          |
|          |   ====   |    ==    |
|          |  ======  | ======== |

|          |          |          |
|          |          |          |
|          |          |    ==    |
|   ====   |  ======  | ======== |

|          |          |          |
|          |          |          |
|    ==    |          |          |
|   ====   |  ======  | ======== |

|          |          |          |
|          |          |          |
|    ==    |          |  ======  |
|   ====   |          | ======== |

|          |          |          |
|          |          |          |
|          |          |  ======  |
|   ====   |    ==    | ======== |

|          |          |          |
|          |          |   ====   |
|          |          |  ======  |
|          |    ==    | ======== |

|          |          |    ==    |
|          |          |   ====   |
|          |          |  ======  |
|          |          | ======== |
```

2. Here are the completed implementations of methods in `https://github.com/jnylam/SJSU-cs146-s17/blob/master/02_Recursion2/src/cc/jennylam/cs146/TreeNode.java`

```
/*
 * Return the max value among all nodes in the tree rooted at the
     current node.
 */
public E max() {
    E max = value;
    if (left != null) {
        E maxOfLeftSubtree = left.max();
        if (max.compareTo(maxOfLeftSubtree) < 0)
            max = maxOfLeftSubtree;
    }
    if (right != null) {
        E maxOfRightSubtree = right.max();
        if (max.compareTo(maxOfRightSubtree) < 0)
            max = maxOfRightSubtree;
    }
    return max;
}

/*
 * Return the number of nodes in the tree rooted at the current node.
     The current node is counted.
 */
public int size() {
    int size = 1;
    if (left != null)
        size += left.size();
    if (right != null)
        size += right.size();
    return size;
}

/*
 * Return a list of the nodes in the tree rooted at the current node,
     ordered by post-order traversal.
 */
public List<E> postorder() {
    List<E> list = new ArrayList<>();
    // left subtree
    if (left != null) {
        List<E> leftList = left.inorder();
        list.addAll(leftList);
    }
    // right subtree
    if (right != null) {
        List<E> rightList = right.inorder();
        list.addAll(rightList);
    }
    // do a little work
    list.add(value);
    return list;
}
```

3. **a)** The outer loop is called 567 times. The innermost loop is called j times, with j ranging between 0 and $n-1$ (inclusive). The number of calls is $567(n-1)n/2$ times, which is $O(n^2)$. The amount of space used is constant.

   b) Let n be the length of the input array. doSomething() is called $n/2$ times in the for-loop and $\log_2 n$ times in the while-loop for a total of $n/2 + \log_2 n$ or $O(n/2)$ calls. The space complexity is $O(1)$.

   c) There is one call (O(1)) to doSomething(), and $O(\log n)$ amount of space used.

   d) There are $\log_{10} n$ or $O(\log n)$ to doSomething(), and $O(\log n)$ amount of space used.

   e) The answer is the same as in (d).

   f) There are $n/3$ or $O(n)$ calls to doSomething() and $O(n)$ amount of space used.

   g) If $n < m$, there are $m - n$ calls to doSomething()and $O(m - n)$ amount of space used. Otherwise there is a singe call and a constant amount of space used.

4. **a)** There are $\log_{13} n$ calls to meow().

   b)
```
void makeSomeNoise(int n) {
    int m = n;
    for (int i = 0; i < m; i++) {
        while (n > 1) {
            meow();
            n /= 13;
        }
    }
}
```

   c)
```
void makeSomeNoise(int n) {
    if (n <= 1)
        return;
    meow();
    makeSomeNoise(n/13);
}
```

5. **a)**
```
int powerOf2(int n) {
    int p = 1;
    for (int i = 0; i < n; i++)
        p *= 2;
    return p;
}
```

   The time complexity of this function is $O(n)$.

   b) They compute the same value since, on input n, both return the square of foo($n/2$).

   c) Every call to foo() makes at most one recursive call, whereas bar() makes two. Specifically, foo makes $\log_2(n+1)$ calls, so runs in $O(\log n)$. On the other hand, bar(), on input $n = 2^k$, makes

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

   or $O(n)$ calls, each of which does a constant amount of work, for a total of $O(n)$ time.

d)
```
int powerOf2(int n) {
    if (n == 0)
        return 1;
    int p = powerOf2(n/2);
    return (n%2 == 0) ? p*p : 2*p*p;
}
```