Q1

Prove the following by induction:

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

(a) Show that true when n = 1.

$$\sum_{i=1}^{n} i^3 = 1^3 = \left(\frac{1(1+1)}{2}\right)^2 = 1$$

(b) Assume true when n= k.

$$\sum_{i=1}^{k} i^3 = \left(\frac{k(k+1)}{2}\right)^2$$

(c) Show that true when n = k+1.

First, evaluate the left hand side (LHS).

$$\begin{aligned}
\sum_{i=1}^{k+1} i^3 &= \sum_{i=1}^{k} i^3 + (k+1)^3 \\
&= \left(\frac{k(k+1)}{2}\right)^2 + (k+1)^3 \\
&= \frac{k^4+2k^3+k^2}{4} + (k+1)^3 \\
&= \frac{k^4+2k^3+k^2}{4} + (k+1)(k+1)(k+1) \\
&= \frac{k^4+2k^3+k^2}{4} + \frac{4k^3+12k^2+12k+4}{4} \\
&= \frac{k^4+6k^3+13k^2+12k+4}{4}
\end{aligned}$$

Next, evaluate the right hand side (RHS).

$$\begin{aligned}
\left(\frac{(k+1)(k+2)}{2}\right)^2 &= \left(\frac{k^2+3k+2}{2}\right)^2 \\
&= \frac{(k^2+3k+2)(k^2+3k+2)}{4} \\
&= \frac{k^4+6k^3+13k^2+12k+4}{4}
\end{aligned}$$

From this, we see that LHS = RHS.
QED.

Q2

(a) Worst case for insertion sort (according to textbook page 27) is

$$an^2 + bn + c \quad \text{(quadratic)}$$

Thus, if there are n/k lists of length k, the time will be:

$$\frac{n}{k}(ak^2 + bk + c) = ank + bn + \frac{cn}{k}$$

The highest order term is 'ank', thus the worst case time complexity is O(nk).

(b) From page 37 of the textbook, the recursion tree for merge sort has $lg(n) + 1$ levels, each of which costs cn. In our case, there are only n/k sublists (as opposed to n sublists). Thus, there are only $lg(n/k) + 1$ levels, each of which costs (n/k)*k = n. The final result then is...

$$\frac{n}{k}k\left(lg\frac{n}{k} + 1\right) = nlg\frac{n}{k} + n$$

The highest term is $nlg\frac{n}{k}$, thus the complexity is O($nlg\frac{n}{k}$).

(c) Given O($nk + nlg\frac{n}{k}$). Standard merge sort is nlgn. So that what point is the nk term greater than the nlg$\frac{n}{k}$ term?

At k = 1, O($n + nlgn$). nlgn is greater than n.

At k = n, O($n^2 + nlg1$). n^2 is greater than nlg1.

The term must be between 1 and n. Let's try k = lgn.

$$O(nlgn + nlg\frac{n}{lgn})$$

The $nlg\frac{n}{lgn}$ can be simplified as follows:

$$nlg\frac{n}{lgn} = n(lgn - lg(lgn)) \approx nlgn$$

When k = lgn, the term are approximately equal. Thus, this is the maximum value.

(d) In practice, k should be chosen so that the cost of insertion sort for a list of size k is less than the cost of merge sort on a list of size k.

Q3

The following is Java/pseudocode for an algorithm to print the in-degree and out-degree of every vertex in an m-edge, n-vertex directed graph represented using adjacency lists. Some assumptions are made. These are (1) that the index increments from zero, and (2) the number of vertices is fixed. The complexity is O(m+n).

```
printDegree(Graph graph) {
   int[] in_count = new int[graph.V];
   int[] out_count = new int[graph.V];
   List[] adj = graph.getAdj();

   // this block iterates through the m edges and determines the
   // in-degree and out-degree of each vertex
   for ( int i = 0; i < graph.V; i++ ) {
      out_count[i] = adj[i].size();
      for ( int j = 0; j < adj[i].size(); j++ ) {
         in_count[adj[i].get(j)]++;
      }
   }

   // this blocks iterates through each vertex and prints
   // the in-degree and out-degree of each
   for ( int i = 0; i < graph.V; i++ ) {
      System.out.println("Vertex " + i + " in: " + in_count[i];
      System.out.println("Vertex " + i + " out: " + out_count[i];
   }
}
```

Q4

The textbook (page 288) was utilized greatly in answering this question. If starting at root, the traversal visits all nodes (n total nodes). For an empty tree, there is still some constant time to process, i.e. T(0) = c. A tree can be divided to left and right subtrees with k and n-k-1 nodes, respectively. Thus:

$$T(n) \leq T(k) + T(n-k-1) + d$$

where d is some constant.

$$T(k) = c_e k + c$$
$$T(n-k-1) = c_e n - c_e k - c_e + c$$

Thus,

$$T(n) \leq T(k) + T(n-k-1) + d = c_e n + c$$

From this, we see that the highest term is $c_e n$ and so the complexity is O(n). Simply put, each node is visited only once and there is a constant time cost for each, ultimately yielding O(n).

Q5

The search for an AVL tree is idenital to that of a standard binary search tree. However, because the AVL tree is balanced, the complexity of an AVL tree search is O(lgn) as opposed to O(n) for binary search tree. In this particular problem, we are seeking to identify not whether a key exists, but whether the value of a key falls between a range specified in the function arguments. This requires slight modification of the algorithm.

The algorithm is implemented as a recursive algorithm, with arguments for a, b, and node. First, we check if the node is null, returning false if so. This indicates either that the tree is empty, or that the no node exists between the range a and b. Next, we check if the node key is between a and b, returning true if so. If the node is not null, but the key is not between a and b, we call the method recursively on the left or right subtree, depending on whether or not the key was smaller than a. The Java/pseudocode for this method is below.

```
avlSearch(int a, int b, Node node) {

   if (node == null ) {
      return false;
   }

   else if ( node.key <= b && node.key >= a ) {
      return true;
   }

   else if ( node.key < a ) {
      avlSearch(a, b, node.right);
   }

   else {
      avlSearch(a, b, node.left);
   }

}
```

As mentioned, the complexity of this method is O(lgn), because the method evaluates at most lgn nodes (the maximum height of the tree).