1. (25 points) List the following 8 functions according to their growth rate:

\[ n^{1000}, (n \log n)^{999}, n!, n^{\log(n)}, (n/2)^n, 3^n, n^{n/2}, (\log n)^n \]

Answer: \((n \log n)^{999}, n^{1000}, n^{\log(n)}, 3^n, n^{n/2}, n!, (n/2)^n\)

2. (25 points) Display the AVL trees after inserting each of the following numbers in the given order: 1, 3, 6, 2, 4, 5.

Answer:

![AVL Trees Diagram]

3. (25 points) The University of Iowa uses 8-digit number as student IDs. Please design a linear time sorting algorithm which sorts student records by their IDs.

Answer: We use radix-sort to sort the student records using eight rounds of bucket-sort, each on one digit, starting from the least significant digit. The bucket sort uses 10 buckets, each for one digit, and is stable. Suppose each student’s ID is \(x = d_7d_6 \ldots d_1d_0\). The pseudocode is as follows:

```
sortByID(R) {
    // R stores all student records, and Bucket[j] is a list of studentRecords, 0 \leq j < 10.
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 10; j++) Bucket[j].clear();
```
for each $x$ in $R$, such that $ID(x) = d_7d_6 \ldots d_1d_0$.
    // $d_i$ can be computed as $d_i = (ID(x)/10^i)\%10$
    Bucket[d_i].add(x);  // add at the end of Bucket[d_i].
S = {};
for (j = 0; j < 10; j++) S = append(S, Bucket[j]);
R = S;
}
return R;
}

The outer loop takes 8 rounds, the inner loop takes $O(n + 10)$, where $n$ is $|R|$. The total time is $O(8(n+10)) = O(n)$.

4. (25 points) Given a balanced binary search tree $T$ of $n$ nodes and a number $i$, $1 \leq i \leq n$, please provide an algorithm (in pseudocode) to return the node in $T$ which contains the $i$th smallest key. If each node contains the size of the subtree rooted by the node, how to use this information in your algorithm? Please provide the complexity of both algorithms.

Answer: If the size of each subtree is not available, we have to travel the tree in in-order traversal to find the $i$th smallest key.

```
int selectIth(node x, int i) {
    // We use a global variable, A, to store the $i$th smallest key. Initially, A = null;
    // Output: the size of the subtree rooted by x or -1 if $i$th smallest key has been found.
    If (x.left != null) { j = selectIth(x.left, i); if (j <= i) return -1;
    If (i == j+1) { A = x.key; return -1; }
    k = selectIth(x.right, i-j-1);
    if (k == -1) return -1; else return (j+k+1);
}
```

The first call is “$A = null; selectIth(root, i)$”. The worst case complexity is $O(n)$, where $n$ is the number of nodes in the tree.

If the size of each subtree is available, then the search is faster:

```
int selectIth2(node x, int i) {
    // Output: $i$th smallest key, assume $1 \leq i \leq n$.
    s = 0;
    if (x.left != null) s = x.left.size;
    if (i <= s) return selectIth2(x.left, i);
    else if (i == s+1) return x.key;
    else return selectIth2(x.right, i-s-1);  // x.right cannot be null, because $i \leq n$.
}
```
The first call is “A = selectIth2(root, i)”. The worst case complexity is O(h), where h is the height of the search tree.