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

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

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

2. (25 points) Display the Red-Black trees after inserting each of the following numbers in the given order: 1, 3, 6, 2, 4, 5 (using double circles for black nodes and single circle for red nodes).

Answer:

3. (25 points) Design an efficient sorting algorithm for an array of \( n \) integers where for each element \( x \) of the array, \( 0 < x < n^2 \). Please provide the complexity of your algorithm.

Answer: For each \( x \) in the array, we may write \( x = an + b \), where \( 0 \leq a, b < n \). We then apply the radix sort to sort \( <a, b> \): First sort the array according to \( b \), using the bucket sort (\( n \) buckets), then use the stable bucket sort (\( n \) buckets) to the array according \( a \). Note that \( a = x/n \) and \( b = x\%n \). Since both bucket sorts take \( O(n) \) time, so the total time is \( O(n) \). The space complexity is also \( O(n) \).
fastSort(A[]) {
    // A[] stores n integers, each satisfies 0 < x < n^2.
    // Bucket[i] is a linked list of integers, 0 <= i < n
    for (i = 0; i < n; i++) Bucket[i].clear();
    for (i = 0; i < n; i++)
        x = A[i];
        Bucket[x%n].add(x);  // add at the end of Bucket[x%n].
    S = {};
    for (i = 0; i < n; i++) S = append(S, Bucket[i]);
    for (i = 0; i < n; i++) Bucket[i].clear();
    for (x in S)
        Bucket[x/n].add(x);  // add at the end of Bucket[x/n].
    j = 0;
    for (i = 0; i < n; i++)
        for (x in Bucket[i])
            A[j++] = x;
    return A;
}

4. (25 points) Let T be a rooted binary tree with more than one node. The degree of any node x in T is the number of nodes connecting to x (as its children or parent). A node y of T is said to be a core node if there is no path of length two or less from y to a node x whose degree is one. Please design an efficient algorithm that identifies all core nodes of T.

Answer: Nodes of degree one are leaf nodes plus the root if the root has only one child. We can compute the distances from the children and the root if the root has only one child to decide if a node is a core node. A special case is that if a child is a leaf, then its sibling cannot be a core. All this can be done using a post-order traversal.

findCore()
    // Assume T is the root of the tree, and C is a list of nodes to store core nodes,
    //  Note that C is a global variable.
    C = {};  // empty list
    if (T.left != null && T.right != null)
        findCore2(T, 3);  // root of T is not degree 1
    else
        findCore2(T, 0);  // root of T is degree 1.
    return C;

int findCore2(node x, int d) {
    // Input: x is the current node; d is the distance from x to the root if the root is degree 1
    // Output: the minimal distance from x to a leaf node
if (x.left == null && x.right == null) // Base case
    return 0;

if (x.left != null)
    d1 = findCore(x.left, d+1);
else d1 = 3;

if (x.right != null) {
    if (d1 == 0) d2 = findCore2(x.right, 2); // special case
    else d2 = findCore2(x.right, d+1);
    if (d2 == 0 && x.left != null) C = delete(C, x.left); // special case
} else d2 = 3;

if (d > 2 && d1 > 1 && d2 > 1) C = insert(C, x); // x is a core node.
    return min(d1+1, d2+1);
}

The complexity of findCore2 is O(n), where n is the number of nodes in the tree rooted by x.