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:

![Red-Black Trees Diagram](image)

3. (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.

```c
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 ith 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:

```java
def selectIth2(node x, int i) {
    // Output: ith smallest key, assume 1 ≤ i ≤ 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 ≤ n.
}
```

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.

```java
public void 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 = findCore2(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.