Binary Trees

1 Law and Order

Write the pre-order, in-order, post-order, DFS, and BFS traversal of the following binary search tree. Assume for DFS and BFS, process child nodes left to right.

```
        10
       /  \
      3    12
     / \     \
    1   7     13
                \
                 15
```

```
Pre-order: 10, 3, 1, 7, 12, 13, 15
In-order: 1, 3, 7, 10, 12, 13, 15
Post-order: 1, 7, 3, 15, 13, 12, 10
DFS: 10, 3, 1, 7, 12, 13, 15
BFS: 10, 3, 12, 1, 7, 13, 15
```

2 Is This a BST?

The following code should check if a given binary tree is a BST. However, for some binary trees, it is returning the wrong answer. Think about an example of a binary tree for which the method fails. Then, write `isBSTGood` so that it is correct. **Hint**: You will find `Integer.MIN_VALUE` and `Integer.MAX_VALUE` helpful.

```java
public static boolean isBSTBad(TreeNode T) {
    if (T == null) {
        return true;
    } else if (T.left != null && T.left.val > T.val) {
        return false;
    } else if (T.right != null && T.right.val < T.val) {
        return false;
    } else {
        return isBSTBad(T.left) && isBSTBad(T.right);
    }
}
```

An example of a binary tree for which the method fails:

```
            10
           /  \
          5    15
         / \
        3   12
```

The method fails for some binary trees that are not BSTs since it only checks that the value at a node is greater than its left child and less than its right child, not that its value is greater than every node in the left subtree and less than every node in the right subtree. Above is one example of a tree for which it fails.

By the way, the method does return true for every binary tree that actually is a BST.

Below is the correct code:

```java
public static boolean isBSTGood(TreeNode T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(TreeNode T, int min, int max) {
    if (T == null) {
        return true;
    } else if (T.val < min || T.val > max) {
        return false;
    } else {
        return isBST(T.left, min, T.val) && isBST(T.right, T.val, max);
    }
}
```

3 Sum Paths

Define a root-to-leaf path as a sequence of nodes from the root of a tree to one of its leaves. Write a method `printSumPaths(TreeNode T, int k)` that prints out all root-to-leaf paths whose values sum to k. For example, if RootNode is the binary tree rooted in 10 in the diagram below and k is 13, then the program will print out `10 2 1` on one line and `10 4 -1` on another.

```
      10
     /  \
    2    4
   / \    \
  5   1    -1
```

(a) Provide your solution by filling in the code below:

```java
public static void printSumPaths(TreeNode T, int k) {
    if (T != null) {
        sumPaths(T, k, "");
    }
}

public static void sumPaths(TreeNode T, int k, String path) {
    k -= T.val;
    if (T.left == null && T.right == null) {
        if (k == 0) {
            System.out.println(path + T.val);
        }
    } else {
        path += T.val + " ";
        if (T.left != null) {
            sumPaths(T.left, k, path);
        }
        if (T.right != null) {
            sumPaths(T.right, k, path);
        }
    }
}
```

(b) What is the worst case running time of the `printSumPaths` in terms of N, the number of nodes in the tree? What is the worst case running time in terms of h, the height of the tree? In the worst case the height of the tree is N and at each level performs a string concatenation. If we assume that all nodes in the tree have values bounded by some constant then at level l we perform a string concatenation of a string of length l (the length of the path from the root to that node) and a string whose length is bounded by some constant. Since string concatenation is linear, we get a running time of $1+2+\ldots+N = \Theta(N^2)$.

The worst case for the running time in terms of h is a complete binary tree. In this case, there are 2^h leaves, all at the bottom level of the tree. Each string concatenation on this level takes $\Theta(h)$ time (again assuming that the values in the tree are bounded by some constant). Thus the total running time is $\Theta(h2^h)$ (since there are at most 2^h non-leaf nodes and the string concatenation for these nodes takes $O(h)$ time).