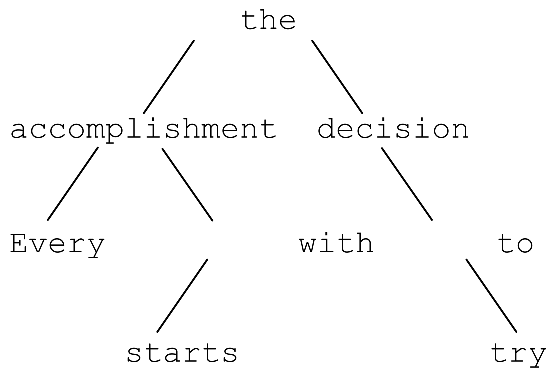
CSE 143 SUMMER 2010 FINAL EXAM SOLUTION (PART 2)

1. (8 points) Draw the tree of strings that would give the following traversals:

   pre-order: `the accomplishment Every with starts decision to try`
   in-order: `Every accomplishment starts with the decision to try`



   Write the post-order traversal of the tree:

   `Every starts with accomplishment try to decision the`

2. (13 points) Write a method for the `IntTree` class called `isBalanced` that returns `true` is the tree is balanced and `false` otherwise. A tree is balanced if the heights of its left and right subtrees differ by at most 1 AND the individual subtrees are themselves balanced. You may assume that there is a method called `height` that takes an `IntTreeNode` as a parameter and returns the height of the tree rooted at that node.
   [DESCRIPTION ABOUT `Math.abs` CUT]

```java
public boolean isBalanced() {
   return isBalanced(overallRoot);
}

private boolean isBalanced(IntTreeNode root) {
   if (root == null) {
     return true;
   } else {
      int heightLeft = height(root.left);
      int heightRight = height(root.right);
      if (Math.abs(heightLeft - heightRight) > 1) {
         return false;
      }

      return isBalanced(root.left) &&
             isBalanced(root.right);
   }
}
```

3. (13 points) Write a method for the `IntTree` class called `fill()` that replaces empty children subtrees of a node with new nodes provided that the new nodes do not increase the height of the tree. The new nodes will contain 0 as its value. Leaf nodes have two empty subtrees, so a call to `fill` would add a left and right node (again, only if the new nodes do not increase the height of the tree). You may assume that there is a method called `height` that takes an `IntTreeNode` as a parameter and returns the height of the tree rooted at that node. Below is an example of a tree and two successive calls to `fill`.

```java
public void fill() {
   int h = height(overallRoot);
   overallRoot = fill(overallRoot, h);
}

private IntTreeNode fill(IntTreeNode root, int max) {
   if (max == 0) {
      return null;
   } else if (root == null) {
      return new IntTreeNode(0);
   } else {
      root.right = fill(root.right, max - 1);
      root.left = fill(root.left, max - 1);
   }

   return root;
}
```

```java
public void fill() {
   overallRoot = fill(overallRoot, 0);
}

private IntTreeNode fill(IntTreeNode root, int depth) {
   if (root == null) {
      if (depth < height(overallRoot)) {
         return new IntTreeNode(0);
      }
   } else {
      root.right = fill(root.right, depth + 1);
      root.left = fill(root.left, depth + 1);
   }

   return root;
}
```

4. (13 points) A combination is three numbers in order (this is not the mathematical definition of combination). The `Combination` class has the following methods:

    `Combination()`
    Constructs a `Combination` with three random numbers as the combination

    `void setCombination(int first, int second, int third)`
    Sets the combination to be the given numbers (in that order)

    `boolean isCombination(int first, int second, int third)`
    Whether the given numbers are identical to those in this `Combination` object

    Design a new class called `ComboLock` that is a `Combination` that must be "open" before you can set the combination. The `ComboLock` has the following methods:

    `void setCombination(int first, int second, int third)`
    Sets the combination to be the given numbers (in that order) if the lock is open. If the lock is not open, then `"Open lock before setting combo."` is printed.

    `boolean isCombination(int first, int second, int third)`
    Whether the given numbers are identical to those in this `Combination` object

    `void open(int first, int second, int third)`
    If the lock is already open, `"Lock is already open!"` is printed. If the combination is correct, the lock opens and `"Lock is open."` is printed. If the combination is incorrect, then `"Wrong combination!"` is printed and the lock does not open.

    `void close()`
    Closes the lock and subsequently prints `"Lock is closed."`

    The `ComboLock` starts out being "open". Avoid redundancy where possible and use proper encapsulation.

```
public class ComboLock extends Combination {
   private boolean isOpen;

   public ComboLock() {
      isOpen = true;
   }

   public void close() {
      isOpen = false;
      System.out.println("Lock is closed.");
   }
```

```
    public void setCombination(int first, int second, int third) {
       if (isOpen) {
          super.setCombination(first, second, third);
       } else {
          System.out.println("Open lock before setting combo.");
       }
    }

    public void open(int first, int second, int third) {
       if (isOpen) {
          System.out.println("Lock is already open!");
       } else if (isCombination(first, second, third)) {
          isOpen = true;
          System.out.println("Lock is open.");
       } else {
          System.out.println("Wrong combination!");
       }
    }
}
```

5. (5 points) What is the running time of this method? Circle one: **O(n)**

```
public int mystery(int[] array) {
    int result = 0;
    for (int i = 0; i < 10; i++) {
        int sum = i;
        for (int j = 0; j < array.length; j++) {
            sum += array[j];
        }
        if (sum > result) {
            result = sum;
        }
    }

    return result;
}
```

Explain your answer in 100 words or less.

The outer for loop runs exactly 10 times. The body of that for loop has a few O(1) operations and an O(n) loop, so the total running time of the outer for loop is O(10•n). Since we don't care about the coefficient, the running time for that for loop is O(n). The remaining operations in the method are O(1) and thus insignificant compared to the for loop.