# **Traditional vs. Tail Recursion**

## **Traditional Recursion Definition**

* In **traditional recursion**, the typical model is that you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result.
* In this manner, you don't get the result of your calculation until you have returned from every recursive call.
* With these types of problems, we "start from the end (base case) and works backwards."

**Example:**

A great example is finding the maximum depth of a binary tree.

We must first traverse in a DFS fashion down to the first leaf node.

Once we reach the base case (a node with null left and right children), we start making our way backwards up the tree.

public int **maxDepth**(TreeNode root) {

if(root == null) return 0;

return 1 + Math.max(**maxDepth**(root.left), **maxDepth**(root.right));

}

## **Tail Recursion Definition**

In **tail recursion**, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step.

This results in the last statement being in the form of

(return (recursive-function params)).

**Basically, the return value of any given recursive step is the same as the return value of the next recursive call**.

This is because we will be done with calculations when we reach the base case, so when we start returning, no matter how far we are into the stack, we return the same value from the base case all the way to the beginning.

**Main Difference**

The difference is that

* With *tail recursion*, the final answer is calculated by the LAST invocation of the method alone. This final answer is returned, meaning all the frames on the stack return the same value.
* With *regular recursion*, the final answer is calculated at the beginning, rather than the end. We need to keep the state of all recursive call return results for all frames to calculate the answer.

# **How To Divide Problem into Subproblems**

* Recursive solutions, by definition, are built off of **solutions to subproblems**.
* Many times, this will mean simply to compute **f(n)** by
  + adding something,
  + removing something,
  + changing the solution for f(n-1)
  + dividing the problem set in half then merging the results,
  + etc.
* There are many ways you might divide a problem into subproblems.
* Three of the most common approaches to develop an algorithm are bottom-up, top-down, and half-and-half.

1. **Bottom-Up Approach**

The bottom-up approach is often the most intuitive.

We start with knowing how to solve the problem for a simple base case.

Once we reach the base case, we work our way backwards, back up the tree.

We figure out how to solve the problem for one element,

Then two elements,

then three elements,

and so on…

The key here is to think about how you can build the solution for one case off of the previous case (or multiple previous cases).

1. **Top-Down Approach**

The top-down approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem.

In these problems, we think about how we can divide the problem for case N into subproblems.

Be careful of overlap between the cases.

1. **Half-and-Half Approach**

In addition to top-down and bottom-up approaches, it's often effective to divide the data set in half.

For example, binary search works with a "half-and-half" approach.

When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.

Another example of a "half-and-half" approach is merge sort.

We sort each half of the array and then merge together the sorted halves.