Big O notation

Big O notation is used to compare algorithmic running times. It is NOT an exact measure of how fast an algorithm is. Instead, it will give you an idea of how fast an algorithm will compute compared to other algorithms.

How to apply Big O notation to an algorithm:

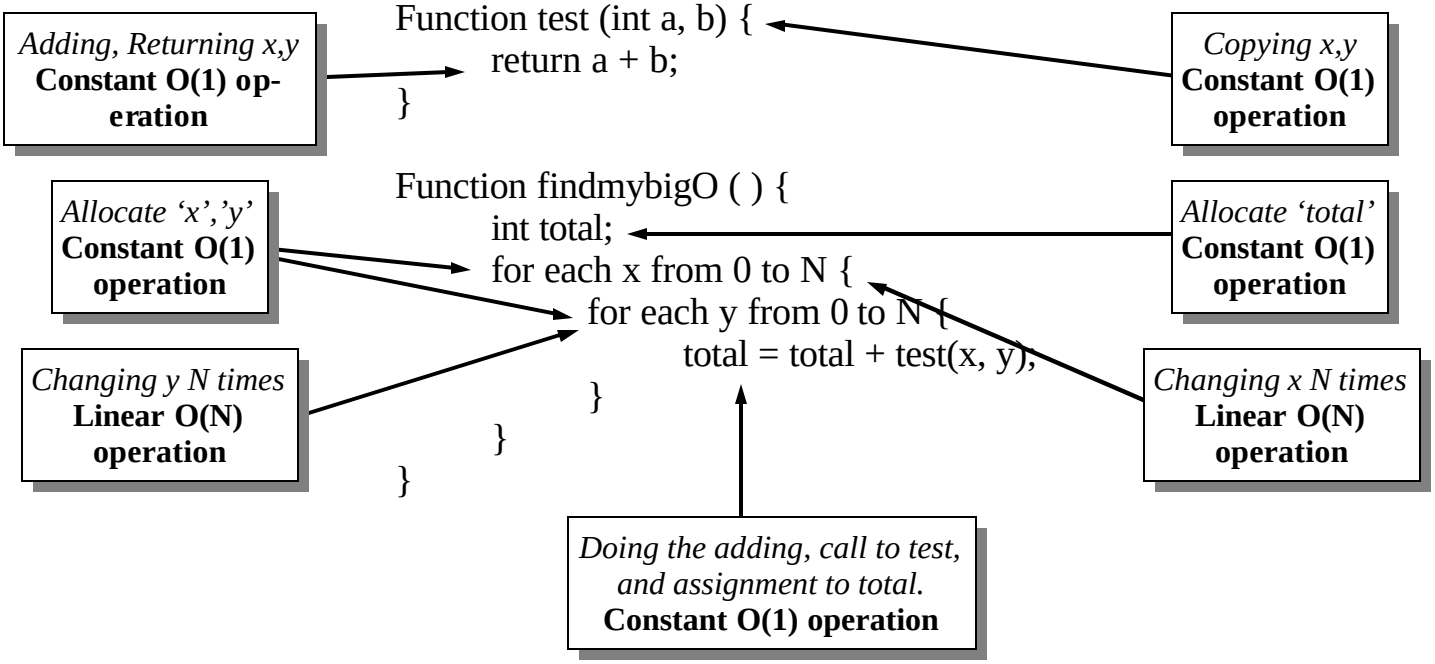
Check out the following example psuedo-code. . .

```
Function test (int a, b) {
        return a + b;
}

Function findmybigO ( ) {
        int total;
        for each x from 0 to N {
                for each y from 0 to N {
                        total = total + test(x, y);
                }
        }
}
```

What is the big-O running time of function findmybigO( )?

Here's what you should look at when asked a question like this:



Big O Notation, part 2

So, starting at the top of findmybigO( ), we have:

$$O(1) + O(1) + N * (N * (O(1) + O(1) + O(1)))$$

$$= (2 * O(1)) + N * (3 * O(1) * N)$$

$$= (2 * O(1)) + (3 * O(1) * N^2)$$

Now, remember that we are looking for an **UPPER BOUND**. This means that we take the part of this function that has the **FASTEST GROWTH** as our answer. A good way to see if one function grows faster than the other is to graph them and see! (Or, you could check out derivatives if you're really not sure. . .)

We know that N^2 grows way faster than 2 * O(1), so we toss that term. So our answer to the question is:

$O(3N^2)$, or $O(N^2)$

Example 2: How good ARE those data structures?

Remember this chart (I've changed it a bit)? Lets go through it again with a little more accuracy. . .

Worst Case Comparison of Lists and Arrays					
Type of ADT	Creation	Adding	Deleting	Find Item	Merge
Arrays	*Easy*	*Hard*	*Hard*	*Easy*	*Fair*
Single Lists	*Easy*	*Easy*	*Hard*	*Hard*	*Good*
Double Lists	*Easy*	*Medium*	*Medium*	*Hard*	*Good*
Trees (soon!)	*Varies*	*Varies*	*Varies*	*Varies*	*Varies*

What do you think each cell should be in Big O notation?