15.2 Recursion

Here we go again...

Now that we've done a couple of nested for loops, let's take a look at our favorite problem: recursion.

Consider the recursive function `f3` below:

```
public static int f3(int n) {
   if (n <= 1)
      return 1;
   return f3(n-1) + f3(n-1);
}
```

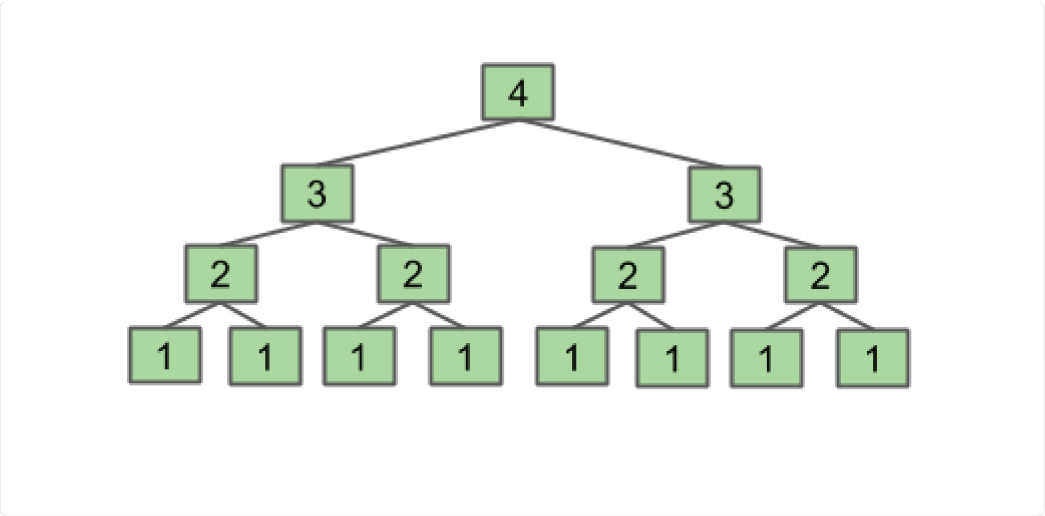
What does this function do?

Let's think of an example of calling `f3(4)`:

•

- The first call will return `f3(4-1) + f3(4-1)`
- Each `f3(3-1)` call will branch out to `f3(2-1) + f3(2-1)`
- Then for each `f3(2-1)` call, the condition `if (n <= 1)` will be true, which will return 1.
- What we observe at the end is that 1 will be returned 8 times, meaning we have `f3(2-1)` summed 8 times.
- Therefore, `f3(4)` will return 8.

We can visualize this as a tree, where each level represents a recursive call and each node value represents the argument to the function :

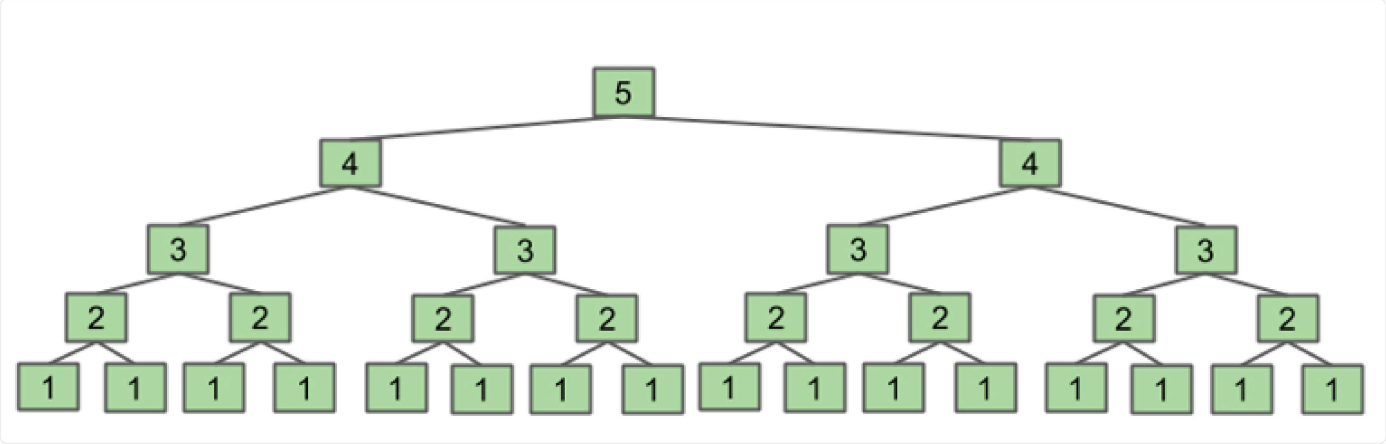


Visualization of f3's recursive calls

You can do a couple more examples, and see that this function returns $2^N - 1$. Visualizing the recursive calls is extremely useful for getting a sense of what the function is doing, and we will discuss a few methods of determining runtime in recursive functions.

Method 1: Intuition

Based on the visualization below, we can notice that every time we add one to `n` we double the amount of work that has to be done:



Then, adding one to `n` N times means doubling the amount of work N times, which results in the intuitive answer for runtime to be 2^N.

Method 2: Algebra

Another way to approach this problem is to count the number of calls to `f3` involved. Utilizing the same tree visualization above, we can see that the number of calls to f3 at each recursive level is equivalent to *the number of nodes at each recursive level* of the tree. For instance, the number of calls we made to `f3` at the top level is 1, at the second level is 2, at the third level is 4, etc.

The total number of calls to f3 then is the **sum** of the number of nodes at each recursive level, which can be expressed as the equation below:

$$C(N) = 1 + 2 + 4 + ... + 2^{N-1}$$

Applying the formula we saw earlier for the sum of the first powers of 2:

$$1 + 2 + 4 + 8 + ... + Q = 2Q - 1$$

Substituting Q with 2^{N-1}, we get:

$$C(N) = 2Q - 1 = 2(2^{N-1}) - 1 = 2^N - 1$$

The work during *each call* is constant , so the overall runtime for `f3` is $\theta(2^N)$.

Method 3: Recurrence Relation (Out of Scope)

This method is not required reading and is outside of the course scope, but worth mentioning for interest's sake.

We can use a "recurrence relation" to count the number of calls, instead of an algebraic approach. This looks like:

$$C(1) = 1 C(N) = 2C(N-1) + 1$$

Expanding this out with a method we will not go over but you can read about in the slides or online, we reach a similar sum to the one above. We can then again reduce it to $2^N - 1$, reaching the same result of $\theta(2^N)$.

Previous
15.1 For Loops

Next
15.3 Binary Search

Last updated 1 year ago