Your New Favorite Recursive Function

Quick – if you were writing a programming tutorial about recursion, what would your example function be? Based on every previous tutorial *ever* written, here's my prediction. You'll settle on the factorial function. In math notation, then Javascript, maybe you'd write something in the ballpark of

$$f(n)=\begin{cases} n\cdot f(n-1) & \text{if } n>0 \\ 1 & \text{if } n=0 \end{cases}$$

```
function f( n ){ return (n == 0) ? 1 : n*f( n-1 );}
```

Or let's suppose you were testing recursion in a new language but had grown weary of that entirely cliched, totally played out factorial function. Maybe you'd go with another perennial favorite, good old Fibonacci.

$$b(n)=\begin{cases} b(n-1)+b(n-2) & \text{if } n>1 \\ 1 & n=0 \text{ or } n=1 \end{cases}$$

```
function b(n){ return (n == 0 || n == 1) ? 1 : b( n-1 ) + b( n-2 );}
```

Let me throw another hat in the ring for the next time you need a simple-to-implement recursive function. Try this:

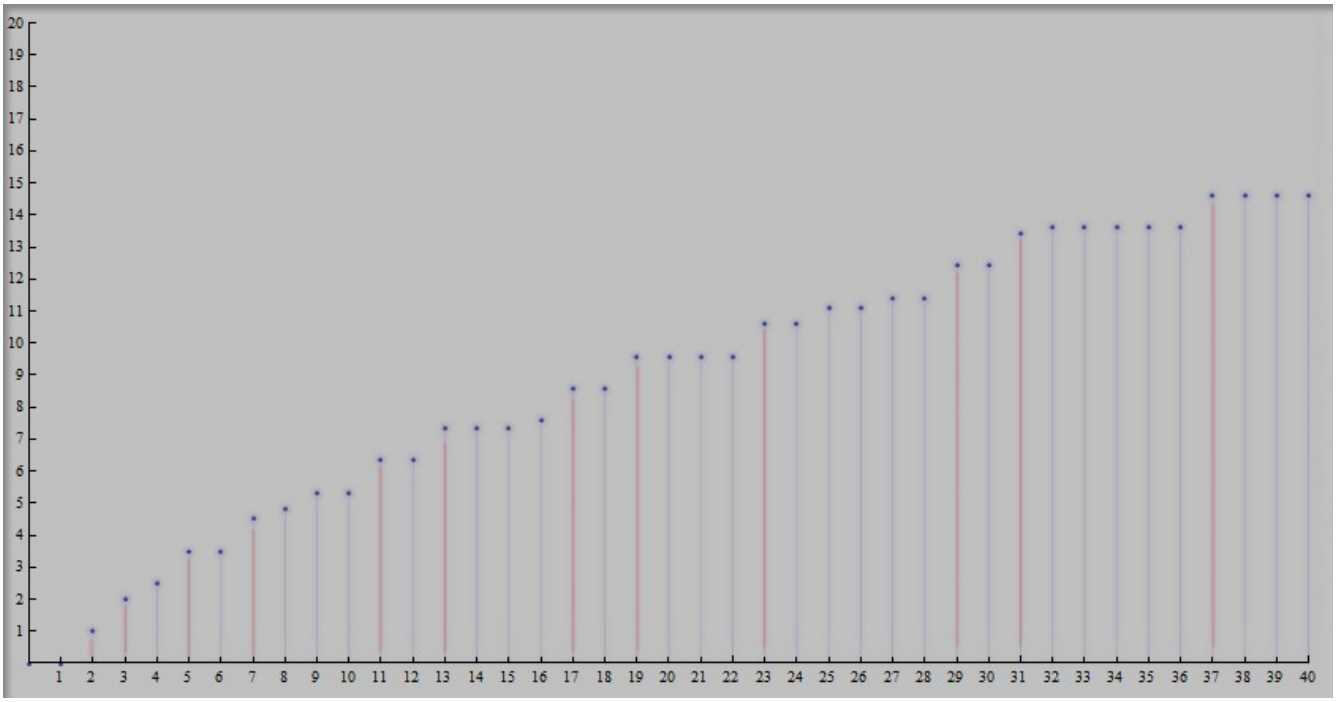
$$p(n,j,k)=\begin{cases} \frac{1}{k}-p(\frac{n}{j},2,k+1)+p(n,j+1,k) & \text{if } n\geq j \\ 0 & \text{if } n<j \end{cases}$$

```
function p( n, j, k ){ return (n < j) ? 0 : 1/k - p( n/j, 2, k+1 ) + p( n, j+1, k );}
```

To make this work its magic, we'll need some initial parameters. So here's a helper function:

$$P(n)=p(n,2,1)$$

Okay. So what's this do? Let's take a look. Here's $P(n)$ for values of n from 1 to 40.



Can you see what $P(n)$ does? Here's another view, the differences between $P(n)$ and $P(n-1)$ for the first 40 values of *n*.

n	$P(n)-\ P(n-1)$
1	0
2	1
3	1
4	1/2
5	1
6	0
7	1
8	1/3
9	1/2
10	0
11	1
12	0
13	1
14	0
15	0
16	1/4
17	1
18	0
19	1
20	0
21	0
22	0
23	1
24	0
25	1/2
26	0
27	1/3
28	0
29	1
30	0
31	1
32	1/5
33	0
34	0
35	0
36	0
37	1
38	0
39	0
40	0

So that makes things pretty clear. It *almost* looks like $P(n)$ counts all prime numbers $\leq n$ ... except it also counts (as fractions, for some reason) prime numbers raised to whole number powers (like 4, 8, or 25) $\leq n$, too.

So what is with those extra fractions? Well, here's the thing. You might know that math's mechanics *strongly* suggest *e* as the natural base of logarithms and 2π as as the natural unit for trig functions. It turns out, this is the natural prime counting function. It's called the Riemann Prime Counting function (http://mathworld.wolfram.com/RiemannPrimeCountingFunction.html). If you can compute it, you can easily use it to count the actual number of primes $\leq n$ - see equation (5) at that mathworld link for how.

So that's reason one $P(n)$ is a really cool recursive function. It's only ever-so-slightly more complicated than the Fibonacci function, and yet computes the natural prime counting function!

Here's reason two it's cool. Look more closely at *p*. Normally, to count primes, you'd need to identify them first, right. To do that, you'd be testing for divisibility. But $P(n)$ doesn't do that. No divisibility tests. Play around with that terse function definition, $p(n, j, k)$. You simply won't find any divisibility tests or prime number identifying in there, try as you might. It must be relying on some important mathematical property of the prime numbers to work.

Pretty cool, huh!

But how does this works, you might very well be asking. What exactly do you think $p(n, j, k)$ does that counts the primes?

Well, what's happening is a bit tricky, actually – tricky enough that it would be quite another task entirely to explain here.

But! If you really are curious why this works, it couldn't hurt to look at http://icecreambreakfast.com/primecount/LinnikVariations.pdf . Specifically look at sections 2, 3, and then finally 4-1, which work through the general ideas at the heart of this. In particular, 4-1a is the main identity; our $p(n, j, k)$ is just one of a few compact ways to write it recursively. A warning: I'm neither a mathematician nor educator... but these identities are cool, and no one else writes about them.

MORE IDEAS

Here are a few more intriguing possibilities that spring immediately from the deep idea at the heart of $p(n, 2,1)$. These links get a lot more math-heavy.

Don't Blow the Stack

$p(n, 2,1)$ computes the Riemann Prime Counting Function as advertised, but in most programming languages, it blows the stack at pretty low values of *n*. A more stack-respectful (and still *extremely* simple) recursive function is given as 4-1b in http://icecreambreakfast.com/primecount/LinnikVariations.pdf .

Summing Primes

Here's a quick extra bonus: with just an ever-so-slight adjustment, the recursive formula from the beginning sums primes instead of just counting them! Try this out:

```
function s( n, j, k ){ return (n < j) ? 0 : j*(1/k - s( n/j, 2, k+1 )) + s( n, j+1, k );}
function S(n){ return s(n,2,1);}
```

You should find that $S(n)-S(n-1)=n\cdot(P(n)-P(n-1))$. If you find the challenge of summing primes compelling, you might find http://icecreambreakfast.com/euler10.html worth a peruse. And can you guess what happens if "j*(" is replaced with "j*j*(" in that definition, or j to any power?

The Riemann Hypothesis

Does the name "the Riemann Hypothesis" (http://en.wikipedia.org/wiki/Riemann_hypothesis) sound familiar at all? If not, it's possibly the most famous open question in all of mathematics. And almost magically, if we extend $p(n, 2,1)$ just the tiniest bit, we can express one version of the Riemann Hypothesis. Here's how. Generalize *p* by adding an extra parameter, *d* - our new definition for p is:

```
function p( n, j, k, d ){ return (n < j) ? 0 : d*(1/k - p( n/j, 1+d, k+1, d )) + p( n, j+d, k, d );}
```

If d=1, we have our original $p(n, 2,1)$ from the very beginning, right? Make sure you follow that. Okay. Now suppose we have some infinitesimally small value *a*, in the ballpark of $\frac{1}{\infty}$ (let's ignore floating point precision issues in what follows). Then the Riemann Hypothesis is true if, for any n > 58 and assuming infinite precision, the difference between p(n,2,1,1) and p(n,1+a,1,a) is always less than $\frac{1}{8\pi}\sqrt{n}\log n$ … and vice versa. That's it.

Pretty mysterious sounding, I know. Want to understand it better? There are more notes about this idea at http://icecreambreakfast.com/notorioustruncation.html .

Counting Primes Faster

Even so, both these approaches suffer from being pretty slow. But the core identity they're both based on can actually be used to count primes pretty quickly. If you're up for some heavier mathematical lifting, there are two general approaches I know of.

The first runs in something like O(n) time and O(log n) space, but with a certain technique called a wheel, you can speed it up to around O(n^4/5) time. It's described in section 4-2 of http://icecreambreakfast.com/primecount/LinnikVariations.pdf, section 2 of http://icecreambreakfast.com/primecount/PrimeCountingSurvey.pdf, and section 5 of http://icecreambreakfast.com/primecount/GeneralizedDivisorResults.pdf. I've written it up a few times because I've never been all that satisfied with how I've covered the idea. Hopefully one of those write ups might make sense to an interested reader. If you just want to dive in and read some C code implementing this technique, a very compact implementation of this algorithm can be found at http://icecreambreakfast.com/primes/PrimeCount.cpp . A wheel optimized version, which is much harder to follow but much, much faster, can be found at http://icecreambreakfast.com/primes/NMPrimeCounter.cpp .

Counting Primes Even Faster

Another approach based on the same core identity is far more complicated still, but it's quite fast, running in something like O(n^2/3 log n) time and O(n^1/3 log n) space. The entire paper http://icecreambreakfast.com/primecount/PrimeCounting_NathanMcKenzie.pdf is dedicated to explaining how it works; there are also attempts at describing the combinatorial ideas in section 4-3 of http://icecreambreakfast.com/primecount/LinnikVariations.pdf , section 3 of http://icecreambreakfast.com/primecount/PrimeCountingSurvey.pdf , section 6 of http://icecreambreakfast.com/primecount/GeneralizedDivisorResults.pdf, and the first part of http://icecreambreakfast.com/primecount/primecounting.php . The first and last of those links have C implementations of the algorithm, but if you'd like to dive right in to some C code, you can take a look at http://icecreambreakfast.com/primecount/primes.cpp : the function countprimes5 implements this algorithm.

Other Identities for Counting Primes

Another way to count primes http://mathworld.wolfram.com/LegendresFormula.html Crandall and Pomerace

And Power Series and Primes

If you remember intro calculus, you might recognize that the Taylor series for the logarithm - $\log(1+n)=\frac{n}{1}-\frac{n^2}{2}+\frac{n^3}{3}-\frac{n^4}{4}+...$ - can be written recursively as $L(n,k)=n(\frac{1}{k}-L(n,k+1))$ with $\log(1+n)=L(n,1)$ when -1 < n <= 1. Do you think that looks at least a little similar $\frac{1}{k}-p(\frac{n}{j},2,k+1)+p(n,j+1,k)$? If you do, that's no accident. Actually, there are scores of equations where the Riemann Prime Counting function mirrors log(x). I find them utterly fascinating. I've collected a bunch of them in section 3 of http://icecreambreakfast.com/primecount/GeneralizedDivisorResults.pdf , although a skim of sections 1 and 2 in that same paper will help section 3 make sense.

CONCLUSION