A Few Novel, Quick Prime Number Counting Techniques

by Nathan McKenzie, December 5, 2012

Overview

I'm aiming this at smart programmers who are mathematically inclined, the sort who might find a tiny, extremely easy-to-implement $O(n)$ time $O(\epsilon)$ *space prime counting algorithm interesting. (Understanding why this all works, on the other hand, might really stretch your math muscles.) Consider this a possible competitor to the Sieve of Eratosthenes the next time you try to pick up Haskell or Go or CUDA or the programming language du jour. As an extra bonus, if you've ever attempted Project Euler #10 (about summing primes), I'll show a variant of this approach that solves it in roughly* $O(n)$ time $O(\epsilon)$ *space, too.*

Intro

Programming the Sieve of Eratosthenes, http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes, is a hoary chestnut of learning programming. The Sieve is fun (well, as fun as these things are likely to be) because it's not too complicated, so it's easy to implement, and it does something interesting – namely, it counts prime numbers, and much more quickly than trial division.

If you've ever programmed the Sieve of Eratosthenes, especially if you've implemented it a few times, you might have wondered idly if there are any other fast, interesting, easy-to-implement prime counting techniques.

If that sounds suspiciously like a leading statement... well, it is! Read on for another approach to counting primes, with a few variations, that is simple to implement and pretty quick. If you're impatient and want to see running code, here's the bright core of this document: a C implementation of a tiny $O(n)$ time, essentially no-memory prime counting algorithm.

```
#include "math.h"
typedef long long big;
big mu[] = { 0, 1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, 1, 1, 0,
-1, -1, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 1, 1, -1, 0, -1, 1, 0, 0, 1, -1 };
double D( big n, int k, big a ){
    if( k == 0 )return 1;
    if( k == 1 )return n - a + 1;
    double t = 0;
    for( int j = 1; j <= k; j++ ){
        double mul = 1;
        for( int i = 1; i <= j; i++ )mul *= ( (double)k - ( j - i ) ) / i;
        for( big m = a; m <= pow(n, 1.0 / k) + .00001; m++ )t += D( n / pow( (double)m, j ), k-j, m+1 )*big( mul + .001 );
    }
    return t;
}
big NumberOfPrimes(big n){
    double t = 0.0;
    for (int j = 1; j < log((double)n) / log(2.0); j++)
        for (int k = 1; k < log( pow( n, 1.0 / j ) ) / log(2.0); k++)
            t += pow( -1.0, (double)k + 1 ) * D(big( pow(n, 1.0 / j) + .00001 ), k, 2 ) / k / j * mu[j];
    return t + .001;
}
```

But what in the world does that all mean? Try it out; it's not too hard to confirm it counts primes, and with the time and space bounds claimed. But why? Well, the rest of this document will strive to explain why this works.

There's going to be a bit of math in the explanation, as a warning. The math isn't hard, exactly. But it might be a slog, because it's likely unfamiliar. Specifically, to understand this code will require gaining familiarity with Linnik's identity, Möbius inversion and the Möbius $\mu(n)$ function, and finally the math of permutations. We'll walk through each of these in turn.

So here's the plan of attack. First, I'm going to explain how we can count primes by instead counting the number of answers to certain simpler multiplication questions without ever seeing a single prime in our calculations or identifying a solitary prime number. (That's our function D, above) This will explain, loosely, the core idea in this approach.

Then, I'm going to show a few ways to count the answers to those multiplication questions and thus count primes. The first way will be slow but easiest to follow, the second will be the approach I've just shown (but it's improvable to around $O(n^{\frac{4}{5}})$ time through optimizations), and a final way will be mentioned that will be just a bit worse than $O(n^{\frac{2}{3}})$ time and $O(n^{\frac{1}{3}})$ memory use (but not at all simple, sadly), all with Mathematica and C implementations of these approaches.

A. The Function $D_{k,2}(n)$

So, I just claimed we can count primes by instead counting the number of answers to certain simpler multiplication questions. What do I mean by that? Well, let me talk through it, starting with a simple example and then generalizing a bit.

Let's start with a very simple question. How many solutions are there to $a \cdot b \leq 10$, if *a* and *b* must be whole numbers > 1? Well,

2x2 = 4, 2x3 = 6, 2x4 = 8, 2x5 = 10
3x2 = 6, 3x3 = 9
4x2 = 8
5x2 = 10

which tallied up is 8 answers total. Nothing too tricky so far.

Now let's suppose I want to be a bit more general – I want to know "Given some number *n*, how many solutions to $a \cdot b \leq n$ are there, if *a* and *b* have to be whole numbers ≥ 2?" Let's punt on how to count that for now and just name it so we can talk about it. Let's call it $D_{2,2}(n)$. The first *2* will mean it's counting the answers for pairs of numbers multiplied together. So now we can say $D_{2,2}(10)=8$. There are 8 pairs of numbers that satisfy our question.

What about the same question for triples? Given some number *n*, how many triples *(a,b,c)* satisfy $a \cdot b \cdot c \leq n$, with *a*, *b*, and *c* whole numbers ≥ 2? You can probably imagine what counting those answers by hand might look like, but let's just skip that for now and name it $D_{3,2}(n)$.

Now I want just a bit more generality. Rather than pairs or triples, I want to pick how many variables are multiplied together. Let's call this $D_{k,2}(n)$, with *k* the number of whole numbers multiplied together. So, for example, $D_{5,2}(80)$ is the count of answers to $a \cdot b \cdot c \cdot d \cdot e \leq 80$, with all variables whole numbers ≥ 2. There happen to be 26 such answers, so $D_{5,2}(80)=26$.

So here we are, finally. I've drawn your eye to this function I am right now calling $D_{k,2}(n)$. Here's a surprising fact: if, for some number *n,* you have values for all $D_{k,2}(n)$ functions, then you can also trivially and immediately know how many prime numbers there are less than *n*.

B. Introducing the prime power function $\Pi(n)$

So, how do we get from $D_{k,2}(n)$ to the number of primes? Well, to understand, we need an intermediate function on our quest to the number of primes.

You might remember from math class abandoning degrees for radians, or doing calculus and using $e=2.71818...$ as the natural base for powers. In both cases, the explanation was that math itself strongly pushes you towards those values. It's in the nature of the patterns.

We run into that here too. Mathematicians label the count of primes $\leq n$ as $\pi(n)$. (*For some more* *background, see http://mathworld.wolfram.com/PrimeCountingFunction.html)* It isn't the natural prime counting function, though. That's what math says. That honor goes to a related value we'll call $\Pi(n)$. It can be written as

$$\Pi(n)=\pi(n)+\frac{1}{2}\pi(n^{\frac{1}{2}})+\frac{1}{3}\pi(n^{\frac{1}{3}})+\frac{1}{4}\pi(n^{\frac{1}{4}})+...$$

(See here for more: http://mathworld.wolfram.com/RiemannPrimeCountingFunction.html)

Surprisingly, math strongly suggests this as the more natural value to work with. We can trivially invert this, which we will get to; if we have $\Pi(n)$, we can express $\pi(n)$ in terms of it.

C. Linnik's Identity: $\Pi(n)$ as a Function of $D_{k,2}(n)$

Okay. So I've claimed if we have this function $\Pi(n)$, we can get our count of prime numbers, $\pi(n)$, but I haven't said how. And I've talked up this function $D_{k,2}(n)$. What's the connection? Well, it turns out that we can express $\Pi(n)$ pretty cleanly in terms of $D_{k,2}(n)$:

$$\Pi(n)=D_{1,2}(n)-\frac{1}{2}D_{2,2}(n)+\frac{1}{3}D_{3,2}(n)-\frac{1}{4}D_{4,2}(n)+...$$

or, in the tidy language of summing notation,

$$\Pi(n)=\sum_{k=1}^{\lfloor\log_2 n\rfloor}\frac{(-1)^{k+1}}{k}D_{k,2}(n)$$

This is a summed version of what is called Linnik's identity. The upper bound is $\log_2 n$ because $D_{k,2}(n)$ is always 0 when n is less than 2^k - 2^k will always be the first answer counted by $D_{k,2}(n)$ for any value of k. I don't have the space to show why this identity works here, unfortunately, but I assure you that it does, and we'll use it to great effect in just a bit.

(For a refresher on summing notation, $\sum_{n=1}^{x} f(n)$ translates roughly in C to `for( n = 1; n <= x; n++ )t += f(n);`*. See http://mathworld.wolfram.com/Sum.html . For a reminder about the floor function, $\lfloor n \rfloor$, which works like casting a positive value to an int in C, see http://mathworld.wolfram.com/FloorFunction.html . This core identity expressing $\Pi(n)$ in terms of $D_{k,2}(n)$ is a summed version of Linnik's identity. See http://www.icecreambreakfast.com/primecount/logintegral.html for an interactive demonstration of it, or track down the phrase "Linnik's identity" in Opera De Cribro by John B. Friedlander and Henryk Iwaniec to see a derivation of it... but mostly I'm going to ask you to take a leap of faith and trust that it works.)*

D. The count of primes $\pi(n)$

We're getting close. Let's recap. I've described a function I've called $D_{k,2}(n)$, the number of solutions to $a_1 \cdot a_2 \cdot ... a_k \leq n$, where $a_k \geq 2$ and is a whole number. I've asserted that Linnik's identity let's us easily write a certain prime-counting function, $\Pi(n)$, in terms of $D_{k,2}(n)$. And I've shown how to express that same function $\Pi(n)$ in terms of our actual goal, the number of primes function $\pi(n)$. We need to do one last thing to have the number of primes in terms of $D_{k,2}(n)$. It's this. We start with our expression for $\Pi(n)$ in terms of $\pi(n)$. We need to flip this around, and get $\pi(n)$ in terms of $\Pi(n)$. Here's how we'll do that. Let's call our starting identity (A):

$$\Pi(n)=\pi(n)+\frac{1}{2}\pi(n^{\frac{1}{2}})+\frac{1}{3}\pi(n^{\frac{1}{3}})+\frac{1}{4}\pi(n^{\frac{1}{4}})+...$$

Now, take (A) and substitute $n^{\frac{1}{2}}$ for *n,* and multiply each side of the expression by one half...

$$\frac{1}{2}\Pi(n^{\frac{1}{2}})=\frac{1}{2}\pi(n^{\frac{1}{2}})+\frac{1}{4}\pi(n^{\frac{1}{4}})+\frac{1}{6}\pi(n^{\frac{1}{6}})+\frac{1}{8}\pi(n^{\frac{1}{8}})+...$$

and subtract that from (A). We've now removed $\pi(n^{\frac{1}{2}})$ from the right side of (A). Let's call what's left

(B):

$$\Pi(n)-\frac{1}{2}\Pi(n^{\frac{1}{2}})=\pi(n)+\frac{1}{3}\pi(n^{\frac{1}{3}})+\frac{1}{5}\pi(n^{\frac{1}{5}})+...$$

Now, substitute $n^{\frac{1}{3}}$ for n into (A) and multiply by one third....

$$\frac{1}{3}\Pi(n^{\frac{1}{3}})=\frac{1}{3}\pi(n^{\frac{1}{3}})+\frac{1}{6}\pi(n^{\frac{1}{6}})+\frac{1}{9}\pi(n^{\frac{1}{9}})+\frac{1}{12}\pi(n^{\frac{1}{12}})+...$$

and subtract that from (B), and we've gotten rid of the $\pi(n^{\frac{1}{3}})$ on (B)'s right side. And so now we have:

$$\Pi(n)-\frac{1}{2}\Pi(n^{\frac{1}{2}})-\frac{1}{3}\Pi(n^{\frac{1}{3}})=\pi(n)+\frac{1}{5}\pi(n^{\frac{1}{5}})-\frac{1}{6}\pi(n^{\frac{1}{6}})+...$$

If we repeat this process long enough, we isolate $\pi(n)$ and have a definition for it exclusively in terms of $\Pi(n)$, that we call (C). It looks like this:

$$\Pi(n)-\frac{1}{2}\Pi(n^{\frac{1}{2}})-\frac{1}{3}\Pi(n^{\frac{1}{3}})-\frac{1}{5}\Pi(n^{\frac{1}{5}})+\frac{1}{6}\Pi(n^{\frac{1}{6}})-\frac{1}{7}\Pi(n^{\frac{1}{7}})+\frac{1}{10}...=\pi(n)$$

Now, you might be wondering what in the world kind of pattern is going on with those positive and negative signs in front of the $\Pi(n)$ terms. It turns out this isn't as unusual as you might think. What we've done is called a Möbius inversion. It reverses relationships between functions like these. It's so common there's a special function for those positive and negative signs, the Möbius $\mu(n)$ function. *(See* http://mathworld.wolfram.com/MoebiusFunction.html *for more.)*

So we can rewrite (C) more tidily as

$$\sum_{j=1}^{\lfloor\log_2 n\rfloor}\mu(j)\cdot\frac{1}{j}\Pi(n^{\frac{1}{j}})=\pi(n)$$

Notice that the upper bound is $\log_2 n$ because $\Pi(2)=1$ and $\Pi(1)=0$. We'll never need values of $\mu(j)$ for j more than around 64 for any values of n we're likely to be interested in, so we'll just store our Möbius $\mu(n)$ values in a table and look them up without regard for how to calculate them, but to give you a sense of the function, the first few values of $\mu(n)$, starting at $n=1$, are *1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1....*

E. Putting It All Together

And so, finally, we have our equations connecting the numbers of primes, $\pi(n)$, to $D_{k,2}(n)$, the real focus of this document. We are left with

$$\pi(n)=\sum_{j=1}^{\lfloor\log_2 n\rfloor}\frac{\mu(j)}{j}\Pi(n^{\frac{1}{j}}) \qquad \Pi(n)=\sum_{k=1}^{\lfloor\log_2 n\rfloor}\frac{(-1)^{k+1}}{k}D_{k,2}(n)$$

Whew! Take a deep breath, and now let's move on to our real job, finding fast or interesting ways to calculate $D_{k,2}(n)$ and so find the number of primes, $\pi(n)$.

1. Counting Primes: The Simplest Way

So let's take a first stab at calculating $D_{k,2}(n)$. It will be our simplest approach.

As a thought experiment, suppose you had to count, by hand, the number of answers to $a\cdot b\cdot c\cdot d\leq 1000$, all variables whole numbers ≥ 2. In our notation that would be $D_{4,2}(1000)$.

Well, a could be any whole number between and including 2 and $1000/(2\cdot 2\cdot 2)$, if you think about it, since each of b, c, and d must be at least 2. So we could have a loop where a takes on each of those values.

Then, b would be any whole number between and including 2 and $1000/(a\cdot 2\cdot 2)$ (because c and d must each be at least 2). So we would make a nested loop where b takes on each of those values.

Then c would any of the whole numbers between and including 2 and $1000/(a\cdot b\cdot 2)$. We would make that nested loop too.

And finally, in the innermost nested loop, d would take on values between and including 2 and $1000/(a\cdot b\cdot c)$. For every value d took on in our innermost loop, we'd add one to our count of possible solutions. So our code to calculate $D_{4,2}(1000)$ might look like this:

```
for( a = 2; a <= 1000/(2*2*2); a++ )
    for( b = 2; b <= 1000/(a*2*2); b++ )
        for( c = 2; c <= 1000/(a*b*2); c++ )
            for( d = 2; d <= 1000/(a*b*c); d++ )
                d_4_total += 1;
```

You could imagine how to generalize that for other values of k by nesting more or less variables. Well, it turns out, those loops are much more conveniently written recursively. In fact, we can say

$$D_{0,2}(n)=1 \qquad \Pi(n)=\sum_{k=1}^{\lfloor \log_2 n\rfloor}\frac{(-1)^{k+1}}{k}D_{k,2}(n)$$

$$D_{k,2}(n)=\sum_{j=2}^{\lfloor n\rfloor}D_{k-1,2}\left(\frac{n}{j}\right) \qquad \pi(n)=\sum_{j=1}^{\lfloor \log_2 n\rfloor}\frac{\mu(j)}{j}\Pi\left(n^{\frac{1}{j}}\right)$$

(We're sacrificing a bit of speed for simplicity here by not accounting for the divisors of 2 in the upper bounds – if you work through the calculations, you'll see it doesn't effect the result at all) This is the most straightforward way to calculate $D_{k,2}(n)$. It's also painfully slow. Still, it does count primes. It's somewhere to start. Try it out and see!

If you want to try this out in Mathematica, try comparing the results of **NumberOfPrimesV1[n_]** here with Mathematica's own built in prime counting function, **PrimePi[n_]**.

```
DV1[0,n_]:=1
DV1[k_,n_]:= Sum[DV1[ k-1, n/j ],{j,2,n}]
PV1[n_] := Sum[ (-1)^(k+1) /k  DV1[k,n], {k,1, Log[2,n] } ]
NumberOfPrimesV1[n_]:=Sum[ MoebiusMu[j]/j PV1[n^(1/j)],{j,1,Log[2,n]}]
```

And the same idea expressed, much less tidily, in C.

```
#include "math.h"
typedef long long BigInt;
BigInt mu[] = { 0, 1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, 1, 1,
0, -1, -1, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 1, 1, -1, 0, -1, 1, 0, 0, 1, -1 };
BigInt DV1(BigInt n, int k){
   if (k == 0) return 1;
   if (k == 1) return n - 1;
   BigInt t = 0;
   for (BigInt i = 2; i <= n; i++) t += DV1( n / i, k - 1 );
   return t;
}
BigInt NumberOfPrimesV1( BigInt n){
   double t = 0.0;
   for (int j = 1; j < log((double)n) / log(2.0); j++)
```

```
    for (int k = 1; k < log( pow( n, 1.0 / j ) ) / log(2.0); k++)
        t += pow( -1.0, (double)k+1 ) * DV1(pow( n, 1.0 / j ), k) / k / j * mu[j];
    return (t+.001);
}
```

1b. The Simplest Way, Made More Recursive

The last section showed a way of counting primes that is pretty slow but got us off to a good start. Before moving on to something faster, I just have to show an interesting rearrangement of the idea from the last section.

$$\Pi_k(n) = \sum_{j=2}^{\lfloor n \rfloor} \frac{1}{k} - \Pi_{k+1}\left(\frac{n}{j}\right) \qquad \pi(n) = \sum_{j=1}^{\lfloor \log_2 n \rfloor} \frac{\mu(j)}{j} \Pi_1\left(n^{\frac{1}{j}}\right)$$

There was a recursive function lurking in there! This is just as slow, so it's not really practical. But I must admit I find it fascinating. If I weren't rushing here, it would be lots of fun to graph $\Pi_1(n)$, $\Pi_2(n)$, $\Pi_3(n)$, and so on. Their visual connections are fun to take in. Anyway, in Mathematica, this is the rather diminutive

```
PV2[k_,n_]:=Sum[1/k - PV2[ k+1, n/j ],{j,2,n}]
NumberOfPrimesV2[n_]:=Sum[MoebiusMu[j]/j PV2[1,n^(1/j)],{j,1,Log[2,n]}]
```

And in C, the likewise svelte

```
#include "math.h"
typedef long long BigInt;
BigInt mu[] = { 0, 1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, 1, 1,
0, -1, -1, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 1, 1, -1, 0, -1, 1, 0, 0, 1, -1 };
double PV2( BigInt n, int k ){
    double t= 0;
    for( BigInt j = 2; j <= n; j++ )t += 1.0 / k - PV2( n/j, k+1 );
    return t;
}
BigInt NumberOfPrimesV2( BigInt n ){
    double t = 0;
    for (int j = 1; j < log((double)n) / log(2.0); j++) t += PV2(pow( n, 1.0 / j ), 1) / j * mu[j];
    return t+.001;
}
```

2. Counting Primes: The Elegant Way

Now let's move on to a much faster way to calculate $D_{k,2}(n)$.

Here's the basic idea, worked through an example. Let's return to counting the answers to $a \cdot b \cdot c \cdot d \leq 1000$, all variables whole numbers ≥ 2. This was our $D_{4,2}(1000)$.

In the last section, when answering this, we counted $2 \cdot 3 \cdot 4 \cdot 5$ as one answer. We also counted $3 \cdot 2 \cdot 5 \cdot 4$ as one answer. We also counted $2 \cdot 5 \cdot 3 \cdot 4$ once, and $3 \cdot 5 \cdot 4 \cdot 2$, and $5 \cdot 3 \cdot 4 \cdot 2$, and … Actually, if we pay close attention, we notice that $2 \cdot 3 \cdot 4 \cdot 5$ shows up as 24 different answers, in different ordered configurations. So does $2 \cdot 3 \cdot 4 \cdot 6$, and $2 \cdot 4 \cdot 5 \cdot 6$, and many others besides. Some other answers show up repeatedly, but not 24 times. So $6 \cdot 6 \cdot 6 \cdot 4$ shows up in 4 configurations. $9 \cdot 9 \cdot 2 \cdot 2$ shows up in 6 different configurations.

So here's a thought. Can we avoid counting sets of numbers individually, and instead count, say, $2 \cdot 3 \cdot 4 \cdot 5$, once, and know that it shows up 24 times?

We can. The number of times an answer shows up is determined only by how many unique numbers it consists of. $2 \cdot 3 \cdot 4 \cdot 5$ shows up 24 times because all four of its numbers differ from each other. $6 \cdot 6 \cdot 6 \cdot 4$ only shows up 4 times because 6 is repeated three times.

To determine how many times a set shows up, we're going to need to use the math of permutations *(See* http://mathworld.wolfram.com/Permutation.html *for more on this topic).*

Here is the core of our new approach. Before, we counted sets explicitly as if unique order mattered. That's still our goal. But now we want to count as if order doesn't matter, and then use the math of permutations to get us back to our count of ordered sets.

So how would we count answers where order doesn't matter? How, when trying to calculate $D_{4,2}(1000)$, do we make sure to count either $2\cdot3\cdot4\cdot5$ or $3\cdot2\cdot5\cdot4$ (multiplied by 24 because of permutations), but not both?

This turns out to be pretty simple. Let's add a new rule. We want to count the answers to $a\cdot b\cdot c\cdot d\leq 1000$ as before, but let's add the restriction that $2\leq a\leq b\leq c\leq d$. Then, we'll use the math of permutations to see how many times that set of numbers shows up. This added restriction will make sure the permutations of each set of numbers is only counted once.

This is much, much faster. So, let's look again at how we would count $D_{4,2}(1000)$.

If we know that $2\leq a\leq b\leq c\leq d$, then the largest possible integer value for *a* is $1000^{\frac{1}{4}}$, which is 5. Make sure you follow this; if you multiply four whole numbers together, and the result is ≤ 1000, the smallest can't be > 5, because 6^4 is 1296. So, given that, we'll work through a loop where *a* is each of those values between and including 2 and 5.

Then, *b* is no smaller than *a,* and knowing that $b\leq c\leq d$, *b* can be no more than the largest integer less than $(\frac{1000}{a})^{\frac{1}{3}}$ for similar reasons. So we'll make a nested loop where *b* is each of those values.

Then, *c* is no smaller than *b*, and because $c\leq d$, *c* can't be larger than the largest integer less than $(\frac{1000}{a\cdot b})^{\frac{1}{2}}$. So we'll make that nested loop too.

And finally, in the innermost nested loop, *d* ranges between *c*, which it can't be smaller than, and $1000/(a\cdot b\cdot c)$. And if we tally up all the values that *d* takes on, counting permutations, we'll have our answer. So, to calculate $D_{4,2}(1000)$ this way might look something like:

```
for( a = 2; a <= pow(1000,1.0/4); a++ )
    for( b = a; b <= pow(1000/a, 1.0/3); b++ )
        for( c = b; c <= pow( 1000/(a*b), 1.0/2); c++ )
            for( d = c; d <= 1000/(a*b*c); d++ )
                total += permutations(a,b,c,d);
```

Hopefully you can see how we'd generalize this approach to more or fewer variables. But we have one outstanding question: how to do we count those permutations?

To do that, we'll need the idea of the binomial coefficient, which is used to express permutations. *(You can find it here: http://mathworld.wolfram.com/BinomialCoefficient.html)* Its notation looks like this: $\binom{k}{j}$ One simple way to evaluate it is $\binom{k}{j}=\frac{k!}{j!(k-j)!}$, where *k!* is the factorial function (as an example, $7!=7\cdot6\cdot5\cdot4\cdot3\cdot2\cdot1$). And for technical reasons, $0!=1$. It's beyond our scope to explain why the binomial coefficient works the way it does for permutations in general.

But here is how we'll use it. Suppose we have a set like $4\cdot6\cdot6\cdot6$, which is $a^1\cdot b^3$. The number of ways we can arrange that turns out to be $\binom{4}{1}\cdot\binom{3}{3}=4$. Another example: let's suppose we want to know how many arrangements of $2\cdot2\cdot9\cdot9$, which is $a^2\cdot b^2$, we can make. The number of ways is $\binom{4}{2}\cdot\binom{2}{2}=6$. How many arrangements of $2\cdot3\cdot4\cdot5$, which is $a^1\cdot b^1\cdot c^1\cdot d^1$, can we make? The number of ways is $\binom{4}{1}\cdot\binom{3}{1}\cdot\binom{2}{1}\cdot\binom{1}{1}=24$.

So what's the pattern here? Well, notice that the bottoms of the binomials match exactly the powers of our variables. So that's simple enough. Also notice that the top of the leftmost binomial is the sum of all the powers of our variables (4 in our examples). Finally, if we take any binomial and subtract the bottom from the top, the result is the top of the binomial directly to its right. So that's how, given a set of numbers of any size, we can calculate the number of permutations of that set we're going to add.

You might be able to imagine what it would look like to take our nested loops above and add in a permutation calculating function based on this description. Fortunately for us, as with some much else, I'm going to point to an extremely concise way of calculating it recursively for any number of variables. It looks like this:

$$D_{0,a}(n)=1 \qquad D_{1,a}(n)=\lfloor n\rfloor-a+1 \qquad \Pi(n)=\sum_{k=1}^{\lfloor\log_2 n\rfloor}\frac{(-1)^{k+1}}{k}D_{k,2}(n)$$

$$D_{k,a}(n)=\sum_{j=1}^{k}\binom{k}{j}\sum_{m=a}^{\lfloor n^{\frac{1}{k}}\rfloor}D_{k-j,m+1}\left(\frac{n}{m^j}\right) \qquad \pi(n)=\sum_{j=1}^{\lfloor\log_2 n\rfloor}\frac{\mu(j)}{j}\Pi(n^{\frac{1}{j}})$$

Written out like this, it's actually only a bit more complicated than our first approach, but it's much, much faster. In fact, at least from my tests, this way of counting primes runs a bit better than $O(n)$ time while using almost no memory! Keep in mind that the Sieve of Eratosthenes runs in the ballpark of $O(n)$ time *and* $O(n)$ memory (although that can be brought down to $O(n^{\frac{1}{2}})$ memory if a segmented sieve is used, at the expense of more implementation complexity).

In Mathematica this is

```
DV3[0,a_,n_]:=1
DV3[1,a_,n_]:=Floor[n]-a+1
DV3[k_,a_,n_]:=Sum[Binomial[k,j] DV3[k-j,m+1,n/m^j],{m,a,n^(1/k)},{j,1,k}]
PV3[n_] := Sum[ (-1)^(k+1) /k  DV3[k,2,n], {k,1, Log[2,n] } ]
NumberOfPrimesV3[n_]:=Sum[ MoebiusMu[j]/j PV3[n^(1/j)],{j,1,Log[2,n]}]
```

And in C, which is a minefield of bad standard libraries and sharp edges about precision, we have, restating the approach from the beginning of the document, this:

```
#include "math.h"
typedef long long BigInt;
BigInt mu[] = { 0, 1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, 1, 1,
0, -1, -1, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 1, 1, -1, 0, -1, 1, 0, 0, 1, -1 };
BigInt inversepow( BigInt n, int k) {
   return BigInt( pow(n, 1.0 / k) + .00000001 );
}
BigInt binomial( double n, int k ){
   double t = 1;
   for( int i = 1; i <= k; i++ )t *= ( n - ( k - i ) ) / i;
   return BigInt( t + .001 );
}
double DV3( BigInt n, int k, BigInt a ){
   if( k == 0 )return 1;
   if( k == 1 )return n - a + 1;
   double t = 0;
   BigInt limit = inversepow(n,k);
   for( int j = 1; j <= k; j++ ){
      BigInt mul = binomial(k,j);
      for( BigInt m = a; m <= limit; m++ )t += DV3( n / pow( (double)m, (double)j ), k-j, m+1 )*mul;
   }
   return t;
}
BigInt NumberOfPrimesV3(BigInt n){
   double t = 0.0;
   for (int j = 1; j < log((double)n) / log(2.0); j++)
      for (int k = 1; k < log( pow( n, 1.0 / j ) ) / log(2.0); k++)
         t += pow( -1.0, (double)k + 1 ) * DV3(inversepow( n, j ), k, 2 ) / k / j * mu[j];
   return t + .001;
}
```

2b. The Elegant Way with a Wheel

The second section contains the following line as its core:

$$D_{k,a}(n)=\sum_{j=1}^{k}\binom{k}{j}\sum_{m=a}^{\lfloor n^{\frac{1}{k}}\rfloor} D_{k-j,m+1}\left(\frac{n}{m^j}\right)$$

We can speed that up immensely if we add an extra constraint. Here's the constraint – if m is divisible by certain small primes (I use 2,3,5,7,11,13,17, and 19), we just skip it and move on to the next entry. With that, we'll find the equations above run much faster. We have to add back in the count of primes we exclude to our final answer, however. This is essentially applying a wheel to our code.

Implementing a wheel is more elaborate than I want to describe or show here. I do use this technique, though, yielding empirically a $O(n^{\frac{4}{5}})$ run time function with no memory use, in the function countprimes4 in this reference: http://www.icecreambreakfast.com/primecount/primescode.html .

3. Counting Primes: The Fast-but-Painful Way

What follows is more of a sketch, because it's too complicated for this document. But the method here is the fastest I know based on the approach of the $D_{k,2}(n) \rightarrow \pi(n)$ connection we're exploring here. This section is more confusing than the rest of the document; don't feel like you've given up if you chose to jump ahead to the conclusion. The goal here to count primes in roughly $O(n^{\frac{2}{3}})$ time and $O(n^{\frac{1}{3}})$ space.

First, here's a quick helper function, $d_{k,2}(n)=D_{k,2}(n)-D_{k,2}(n-1)$. With that, here are our new identities:

$$\Pi(n)=n-1+\sum_{j=\lfloor n^{\frac{1}{3}}\rfloor+1}^{\lfloor n^{\frac{1}{2}}\rfloor}\sum_{k=2}^{\lfloor \log_2 n\rfloor}\frac{-1^{k+1}}{k}D_{k-1,2}\left(\frac{n}{j}\right)+\sum_{j=1}^{\lfloor n^{\frac{1}{2}}\rfloor}\left(\left\lfloor\frac{n}{j}\right\rfloor-\left\lfloor\frac{n}{j+1}\right\rfloor\right)\sum_{k=2}^{\lfloor \log_2 n\rfloor}\frac{-1^{k+1}}{k}D_{k-1,2}(j)$$
$$+\sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor}\sum_{k=2}^{\lfloor \log_2 n\rfloor}\frac{-1^{k+1}}{k}d_{k-1,2}(j)\,D_{1,2}\left(\frac{n}{j}\right)+\sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor}\sum_{s=\lfloor\frac{\lfloor n^{\frac{1}{3}}\rfloor}{j}\rfloor+1}^{\lfloor\frac{n}{j}\rfloor}\sum_{k=2}^{\lfloor \log_2 n\rfloor}\frac{-1^{k+1}}{k}\sum_{m=1}^{k-2}d_{m,2}(j)\,D_{k-m-1,2}\left(\frac{n}{js}\right)$$
$$+\sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor}\sum_{s=1}^{\lfloor\frac{n}{j}\rfloor^{\frac{1}{2}}-1}\left(\left\lfloor\frac{n}{js}\right\rfloor-\left\lfloor\frac{n}{j(s+1)}\right\rfloor\right)\cdot\sum_{k=2}^{\lfloor \log_2 n\rfloor}\frac{-1^{k+1}}{k}\sum_{m=1}^{k-2}d_{m,2}(j)\,D_{k-m-1,2}(s)$$

$$\pi(n)=\sum_{j=1}^{\lfloor \log_2 n\rfloor}\frac{\mu(j)}{j}\Pi(n^{\frac{1}{j}})$$

What a mess, right? Welcome to the magic of optimization – it's like a gutted fish.

Here's why this equation exists: imagine I gave you a table with all the values for $D_{k,2}(n)$ precomputed up to $n^{\frac{2}{3}}$ and $d_{k,2}(n)$ precomputed up to $n^{\frac{1}{3}}$. Looking up those values happens instantly. Now, it is the case that $D_{1,2}(n)=n-1$ and $d_{1,2}(n)=1$ and thus are instantly computable for any n too.

Given that, this identity for $\Pi(n)$ has two useful properties.

First, it only uses values of $D_{k,2}(n)$ and $d_{k,2}(n)$ that we can look up instantly.

And with that fact, second, the sprawling sums of $\Pi(n)$ can be evaluated in slightly worse than $O(n^{\frac{2}{3}})$ time.

It turns out, we can't actually look up those values instantly, but we can compute them in our desired algorithm time and space bounds. Here is roughly how. We use a segmented Sieve of Eratosthenes and sieve all the numbers from 1 to $n^{\frac{2}{3}}$ in blocks of roughly size $n^{\frac{1}{3}}$, but we adjust the sieve so that we get each number in its prime factorized form (specifically, its power signature). It turns out we can calculate $d_{k,2}(n)$ quickly if we have a number's power signature, and $D_{k,2}(n)$ is just $d_{k,2}(n)+D_{k,2}(n-1)$, so we keep a running total of those values. We can do all this in slightly more than $O(n^{\frac{2}{3}})$ time and $O(n^{\frac{1}{3}})$ space. So, to make this algorithm run, we have to interleave the calculation of $\Pi(n)$ with a segmented sieving process, which means in practice, we have to order the evaluation of $\Pi(n)$ from smallest $D_{k,2}(n)$ to largest. It is not a pretty thing, but it can be done.

Much as with our previous approach, a wheel can also speed up computation, though not with such extravagant speed gains.

I don't have the space or inclination to detail this approach any better here, but if it sounds interesting, I have written it up in better detail here: http://www.icecreambreakfast.com/primecount/PrimeCounting_NathanMcKenzie.pdf .

A C implementation can be found as the function countprimes5 here: http://www.icecreambreakfast.com/primecount/primescode.html .

If you want to see, essentially, this technique work, here is some Mathematica code. It's a bit different in form from the equations above, but it's essentially the same computation. Remember, for the actual implementation of this algorithm, the functions labeled CachedD and Cachedd would be computed in an interleaved fashion through sieving.

```
CachedD[k_,n_,s_]:=Sum[Binomial[k,j] CachedD[j,n/(m^(k-j)),m+1],{m,s,n^(1/k)},{j,0,k-1}]
CachedD[1,n_,s_]:=Floor[n]-s+1
CachedD[0,n_,s_]:=1
Cachedd[k_,n_]:=CachedD[k,n,2]-CachedD[k,n-1,2]

DV5[k_,n_]:=
 Sum[CachedD[k-1,n/j,2],{j,Floor[n^(1/3)]+1,n^(1/2)}]+
  Sum[(Floor[n/j] - Floor[n/(j+1)])CachedD[k-1,j,2],{j,1,n/Floor[n^(1/2)]-1}]+
  Sum[Cachedd[k-1,j] (Floor[n/j]-1),{j,2,n^(1/3)}]+
  Sum[Cachedd[m,j] CachedD[k-m-1,n/(j s),2],{j,2,n^(1/3)},{s,Floor[Floor[n^(1/3)]/j]
+1,Floor[n/j]^(1/2)},{m,1,k-2}]+
  Sum[(Floor[n/(j s)] - Floor[n/(j(s+1))])(Sum[Cachedd[m,j]CachedD[k-m-1,s,2],{m,1,k-2}]),
{j,2,n^(1/3)},{s,1,Floor[n/j]/Floor[Floor[n/j]^(1/2)]-1}]
DV5[1,n_]:=Floor[n]-1
PV5[n_]:=Sum[(-1)^(k+1)/k DV5[k,n],{k,1,Log[2,n]}]
NumberOfPrimesV5[n_]:=Sum[MoebiusMu[j]/j PV5[n^(1/j)],{j,1,Log[2,n]}]
```

Conclusion

So there you go; a handful of curious ways to count primes, some simple, some fast... and some a bit of both! Hopefully you found this stuff at least mildly interesting.

I have an ulterior motive in writing this, I will now admit. This is where the Amway pitch comes out.

I've turned these identities around in my head hundreds of ways. I've tried speeding up or fiddling with them from a multitude of angles. I've explored scores of dead ends. This document represents the end point of my explorations. But I'm still fascinated by the general approach here, finding fast ways to calculate $D_{k,2}(n)$ so as to count primes. This is as far as I know how to take these ideas. I would be thrilled to see the ideas taken further, or in some other direction, by someone else if they had some flash of insight that I've overlooked.

Thanks for reading!

Nathan McKenzie, December 5, 2012

References

The core identity at the heart of all this ($\Pi(n)-\Pi(n-1)=\sum_{k=1}\frac{(-1)^{k+1}}{k}(D_{k,2}(n)-D_{k,2}(n-1))$ in the terminology of this paper) was originally published by Yuri Linnik. I describe it more explicitly, with interactive web applications, here: http://www.icecreambreakfast.com/primecount/logintegral.html . Better yet, track down the phrase "Linnik's identity" in Opera De Cribro by John B. Friedlander and Henryk Iwaniec, which will give it mathematical rigor and context.

If you poke around number theory texts, the function I call $D_{k,2}(n)$ lacks a standard name; usually $D_k(n)$ refers to a related function counting from 1, not 2. (It is equal to $D_{k,1}(n)$ from my "Elegant Way" section) See http://en.wikipedia.org/wiki/Divisor_summatory_function for that.

There are other fast ways to count primew, but they can't generally be called simple. Here are the two that seem to matter most:

The paper "Computing π(x): The Meissel, Lehmer, Lagarias, Miller, Odlyzko Method" from M. Deleglise and J. Rivat in 1996 in Mathematics of Computation contains a description of another, slightly better than $O(n^{\frac{2}{3}})$ time and $O(n^{\frac{1}{3}})$ memory use prime counting approach based on combinatorial ideas. You can find that here: http://www.dtc.umn.edu/~odlyzko/doc/arch/meissel.lehmer.pdf

Another core approach is the analytical approach of Lagarais and Odlyzko from "Computing π(x): An Analytic Method". This approach relies on some complicated analytical number theory ideas, and is difficult to implement well, but its run time bounds are either in the ballpark of $O(n^{\frac{1}{2}})$ time and $O(n^{\frac{1}{4}})$ memory use or $O(n^{\frac{3}{5}})$ time nearly no memory use depending on your preferences. That paper can be found here:http://www.dtc.umn.edu/~odlyzko/doc/arch/analytic.pi.of.x.pdf

Both approaches are described well in Crandall & Pomerance's "Prime Numbers: A Computational Perspective", in the last section of the third chapter, which I highly recommend.

Bonus! Solving Euler Project #10.

Project Euler #10, if you're familiar with it, asks you to add up the primes less than 2,000,000 with reasonable haste. It turns out all the techniques in this paper can be modified to add up primes, rather than just counting them. Here's a revision to our method #2 doing just that:

$$D_{S,0,a}(n)=1 \qquad D_{S,1,a}(n)=\frac{\lfloor n(n+1)\rfloor}{2}-\frac{a(a-1)}{2} \qquad P_S(n)=\sum_{k=1}^{\lfloor\log_2 n\rfloor}\frac{-1^{k+1}}{k}D_{S,k,2}(n)$$

$$D_{S,k,a}(n)=\sum_{j=1}^{k}\binom{k}{j}\sum_{m=a}^{\lfloor n^{\frac{1}{k}}\rfloor}m^{S\cdot j}D_{S,k-j,m+1}\left(\frac{n}{m^j}\right) \qquad \text{Sum of Primes(n)}=\sum_{j=1}^{\lfloor\log_2 n\rfloor}\frac{\mu(j)}{j}P_j(n^{\frac{1}{j}})$$

In Mathematica (with a few specializations to speed up computation) this is

```
DV6[S_,1,a_,n_]:=Sum[j^S,{j,a,n}]
DV6[1,1,a_,n_]:=Floor[n](Floor[n]+1)/2-a (a-1)/2

DV6[S_,0,a_,n_]:=1
DV6[S_,k_,a_,n_]:=Sum[Binomial[k,j]m^(S j) DV6[S,k-j,m+1,n/m^j],{m,a,n^(1/k)},{j,1,k}]
PV6[S_,n_]:=Sum[(-1)^(k+1)/k DV6[S,k,2,n],{k,1,Log[2,n]}]
SumPrimes[n_]:=Sum[MoebiusMu[j]/j PV6[j,n^(1/j)],{j,1,Log[2,n]}]
```

and in C it is

```
#include "math.h"
typedef long long BigInt;
BigInt mu[] = { 0, 1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, 1, 1,
0, -1, -1, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 1, 1, -1, 0, -1, 1, 0, 0, 1, -1 };
BigInt inversepow( BigInt n, int k) {
   return BigInt( pow(n, 1.0 / k) + .00000001 );
}
BigInt binomial( double n, int k ){
   double t = 1;
   for( int i = 1; i <= k; i++ )t *= ( n - ( k - i ) ) / i;
   return BigInt( t + .001 );
}
double DV6( int S, int k, BigInt a, BigInt n ){
   if( k == 0 )return 1;
   if( S == 1 && k == 1 )return n*(n+1)/2 - a* (a-1)/2;
   double t = 0;
   BigInt limit = inversepow(n,k);
   for( int j = 1; j <= k; j++ ){
      BigInt mul = binomial(k,j);
      for( BigInt m = a; m <= limit; m++ )t += pow( (double)m, S*j)*DV6( S, k-j, m+1, n / pow( (double)m, (double)j ) )*mul;
   }
   return t;
}
BigInt primeSum(BigInt n){
   double t = 0.0;
   for (int j = 1; j < log((double)n) / log(2.0); j++)
      for (int k = 1; k < log( pow( n, 1.0 / j ) ) / log(2.0); k++)
         t += pow( -1.0, (double)k + 1 ) * DV6(j, k, 2, inversepow( n, j ) ) / k / j * mu[j];
   return t + .001;
}
```

So, this handily sums primes in $O(n)$ time with essentially no memory use. If you're feeling adventurous, for a variant of #3 from earlier, see this working C code that sums primes in roughly $O(n^{\frac{2}{3}})$ time and $O(n^{\frac{1}{3}})$ space here http://www.icecreambreakfast.com/primecount/primesumcount.cpp. I've written up some more notes on this topic here: http://www.icecreambreakfast.com/primecount/PrimeSumming_NathanMcKenzie.pdf . Finally, there's some good discussion on this general topic here: http://mathoverflow.net/questions/81443/fastest-algorithm-to-compute-the-sum-of-primes