Computing the Prime Counting Function with Linnik's Identity in $O\left(n^{\frac{2}{3}}\log n\right)$ Time and $O\left(n^{\frac{1}{3}}\log n\right)$ Space

Nathan McKenzie
11-23-2011

1. Overview

This paper will describe an algorithm for counting primes in roughly $n^{\frac{2}{3}}\log n$ time and $n^{\frac{1}{3}}\log n$ space based on Linnik's identity.

The identity of Linnik[1] which we are interested in states, for our purposes,

$$\sum_{k=1}^{\lfloor \log_2(n)\rfloor} \frac{-1^{k+1}}{k} d_k'(n) = \frac{1}{a} \text{ if } n = p^a, 0 \text{ otherwise} \tag{1.1}$$

where p is a prime number and $d_k'(n)$ is the strict count of divisors function such that

$$d_k'(n) = \sum_{j|n} d_1'(j) d_{k-1}'\left(\frac{n}{j}\right)$$

$$d_1'(n) = \begin{cases} 0 \text{ if } n=1 \\ 1 \text{ otherwise} \end{cases} \qquad d_0'(n) = \begin{cases} 1 \text{ if } n=1 \\ 0 \text{ otherwise} \end{cases}$$

Note that $d_k'(n) = 0$ when $n < 2^k$ The strict count of divisors function is connected to the standard count of divisors function by

$$d_k'(n) = \sum_{j=0}^{k} -1^{k-j}\binom{k}{j} d_j(n) \tag{1.2}$$

where d_k is the standard count of divisors function such that

$$d_k(n) = \sum_{j|n} d_1(j) d_{k-1}\left(\frac{n}{j}\right)$$

and

$$d_1(n) = 1 \qquad d_0(n) = \begin{cases} 1 \text{ if } n=1 \\ 0 \text{ otherwise} \end{cases}$$

The approach for prime counting here is closely related to the method for calculating Mertens function described by Deléglise and Rivat in [2].

2. Counting Primes with the Strict Count of Divisors Summatory Functions

We begin by defining the strict count of divisors summatory function for $d_k'(n)$:

$$D_k'(n) = \sum_{j=2}^{n} d_k'(j) \tag{2.1}$$

Summing Linnik's identity, (1.1), from *1* to *n* then gives the following identity

$$\sum_{k=1}^{\lfloor \log_2(n)\rfloor} \frac{-1^{k+1}}{k} D_k'(n) = \Pi(n) \tag{2.2}$$

where the right hand side is the prime power counting function.

If we rely on standard techniques, we can invert the prime power counting function like so

$$\pi(n) = \sum_{j=1} \frac{1}{j} \mu(j) \Pi\left(n^{\frac{1}{j}}\right) \tag{2.3}$$

where $\pi(n)$ is the prime counting function and $\mu(n)$ is the Möbius function, and we finally arrive at our goal, which is the number of primes in terms of the strict count of divisors summatory functions.

$$\pi(n) = \sum_{j=1} \sum_{k=1} \frac{-1^{k+1}}{jk} \mu(j) D_k'\left(n^{\frac{1}{j}}\right) \tag{2.4}$$

One basic combinatorial property of $d_k'(n)$ and $D_k'(n)$ we will need is

$$D_k'(n) = \sum_{j=2}^{|n|} D_{k-1}'\left(\frac{n}{j}\right)$$
$$D_1'(n) = \lfloor n \rfloor - 1 \tag{2.5}$$

3. The Core Strict Number of Divisors Summatory Identity

The identity we will use to compute the summed version of Linnik's identity is this

$$D_k'(n)= \sum_{j=a+1}^{n} D_{k-1}'(\frac{n}{j}) + \sum_{j=2}^{a} d_{k-1}'(j) D_1'(\frac{n}{j}) + \sum_{j=2}^{a} \sum_{s=\lfloor \frac{a}{j} \rfloor + 1}^{\lfloor \frac{n}{j} \rfloor} \sum_{m=1}^{k-2} d_m'(j) D_{k-m-1}'(\frac{n}{js}) \quad (3.1)$$

for $2 \le a \le n$. The derivation of this will follow shortly.

We are going to be working with 3 core computational rules for the sake of efficiency in our final algorithm. First, $D_1'(n)=n-1$ and so can be computed in constant time and memory. Second, $d_1'(n)=1$ and can also be computed in constant time and memory. Third, we will assume that our sieving process will let us look up $d_k'(j)$ where $j < a$ and $D_k'(\frac{n}{j})$ where $j \ge a$ in constant time. Thus, the key to this identity is transforming $D_k'(n)$ into a form that only uses values of $d_1'(n)$, $D_1'(n)$, $d_k'(j)$ where $j < a,$, and $D_k'(\frac{n}{j})$ where $j \ge a$. The right hand side of (3.1) usefully does exactly this.

To show why (3.1) works, we need to start with a simpler identity.

Establishing the simpler identity

First, we need to show the following

$$\sum_{j=2}^{a} d_m'(j) D_k'(\frac{n}{j}) = \sum_{j=2}^{a} d_{m+1}'(j) D_{k-1}'(\frac{n}{j}) + \sum_{j=2}^{a} \sum_{s=\frac{a}{j}+1}^{j} d_m'(j) D_{k-1}'(\frac{n}{js}) \quad (3.2)$$

where $2 \le a \le n$.

We begin with the left hand side of (3.2),

$$F(n)=\sum_{j=2}^{a} d_m'(j) D_k'(\frac{n}{j}) \quad (3.3)$$

One basic property of $D_k'(n)$ is

$$D_k'(n)=\sum_{j=2}^{\lfloor n \rfloor} D_{k-1}'(\frac{n}{j})$$

and so we can rewrite (3.3) as

$$F(n)=\sum_{j=2}^{a} \sum_{s=2}^{\lfloor \frac{n}{j} \rfloor} d_m'(j) D_{k-1}'(\frac{n}{js})$$

We can separate the inner sum into two pieces, giving us

$$F(n)=\sum_{j=2}^{a} \sum_{s=2}^{\lfloor \frac{a}{j} \rfloor} d_m'(j) D_{k-1}'(\frac{n}{js}) + \sum_{j=2}^{a} \sum_{s=\frac{a}{j}+1}^{\frac{n}{j}} d_m'(j) D_{k-1}'(\frac{n}{js}) \quad (3.4)$$

Now, another basic combinatorial property of $d_k'(n)$ is

$$\sum_{j=2}^{a} \sum_{s=2}^{\lfloor \frac{a}{j} \rfloor} d_m'(j) f(\frac{n}{js}) = \sum_{j=2}^{a} d_{m+1}'(j) f(\frac{n}{j}) \quad (3.5)$$

Looking at (3.4), we should be able to see that our first double sum exhibits the pattern in (3.5). So, we can replace it like so

$$F(n)=\sum_{j=2}^{a} d_{m+1}'(j) D_{k-1}'(\frac{n}{j}) + \sum_{j=2}^{a} \sum_{s=\lfloor \frac{a}{j} \rfloor + 1}^{\lfloor \frac{n}{j} \rfloor} d_m'(j) D_{k-1}'(\frac{n}{js})$$

thus establishing our identity from (3.2)

$$\sum_{j=2}^{a} d_m'(j) D_k'(\frac{n}{j}) = \sum_{j=2}^{a} d_{m+1}'(j) D_{k-1}'(\frac{n}{j}) + \sum_{j=2}^{a} \sum_{s=\lfloor \frac{a}{j} \rfloor + 1}^{j} d_m'(j) D_{k-1}'(\frac{n}{js})$$

Now we have the tool we need to show (3.1).

The Broader Identity

We begin with

$$A(n)=D_k(n)$$

Our identity from (2.5) lets us express this as

$$A(n)=\sum_{j=2}^{n} D_{k-1}'(\frac{n}{j})$$

We can obvious split this sum into the following two

pieces

$$A(n)=\sum_{j=2}^{a} D_{k-1}'(\frac{n}{j})+\sum_{j=a+1}^{n} D_{k-1}'(\frac{n}{j})$$

Based on our core three computational rules, we can look up each of the $D_k'(n)$ values in the second sum in constant time, and so we are left with

$$A_{r1}(n)=\sum_{j=2}^{a} D_{k-1}'(\frac{n}{j})$$

Because $d_1'(n)=1$, we can rewrite this as

$$A_{r1}(n)=\sum_{j=2}^{a} d_1'(j)\,D_{k-1}'(\frac{n}{j})$$

The identity is now in the form that (3.2) expects, and so this sum must equal

$$A_{r1}(n)=\sum_{j=2}^{a} d_2'(j)\,D_{k-2}'(\frac{n}{j})+\sum_{j=2}^{a}\sum_{s=\lfloor\frac{a}{j}\rfloor+1}^{j} d_1'(j)\,D_{k-2}'(\frac{n}{js})$$

The double sum here satisfies our core computational rules, and so we can look up all of those values of $d_k'(n)$ and $D_k'(n)$ in constant time and never think of them again. Not so with the single sum. So, we are left with computing

$$A_{r2}(n)=\sum_{j=2}^{a} d_2'(j)\,D_{k-2}'(\frac{n}{j})$$

This sum is in a form that can work with (3.2), however. So, we can repeat the process again, yielding

$$A_{r2}(n)=\sum_{j=2}^{a} d_3'(j)\,D_{k-3}'(\frac{n}{j})+\sum_{j=2}^{a}\sum_{s=\lfloor\frac{a}{j}\rfloor+1}^{j} d_2'(j)\,D_{k-3}'(\frac{n}{js})$$

Once again, we can look up the values in the double sum, and we're left with a single sum we can apply (3.2) to.

If we repeat this process k-1 times, we are left with

$$A_{rk-1}(n)=\sum_{j=2}^{a} d_{k-1}'(j)\,D_1'(\frac{n}{j})$$

Our three rules of computation say both of the terms inside the sum can be evaluated in constant time. And so we are done. If we go back through this process and collect all terms that were actually calculated along the way, we will find that we have (3.1).

4. Using the Core Number of Divisors Summatory Identity to Count Primes

If we take (3.1), and we combine it with our summed version of Linnik's identity,

$$\sum_{k=1}^{\lfloor\log_2(n)\rfloor}\frac{-1^{k+1}}{k}D_k'(n)=\Pi(n)$$

and keep in mind that

$$D_1'(n)=\lfloor n\rfloor-1$$

then, as a first pass, we can compute the prime power counting function with

$$\begin{aligned}\Pi(n)=\;& n-1+\\ &\sum_{j=a+1}^{n}\sum_{k=2}^{\lfloor\log_2 n\rfloor}\frac{-1^{k+1}}{k}D_{k-1}'(\frac{n}{j})\\ &+\sum_{j=2}^{a}\sum_{k=2}^{\lfloor\log_2 n\rfloor}\frac{-1^{k+1}}{k}d_{k-1}'(j)D_1'(\frac{n}{j})\\ &+\sum_{j=2}^{a}\sum_{s=\lfloor\frac{a}{j}\rfloor+1}^{\lfloor\frac{n}{j}\rfloor}\sum_{k=2}^{\lfloor\log_2 n\rfloor}\frac{-1^{k+1}}{k}\sum_{m=1}^{k-2}d_m'(j)D_{k-m-1}'(\frac{n}{js})\end{aligned}\qquad(4.2)$$

There are two subsequent steps required for turning this equation into its final form for our purposes.

First, for any sum of the form

$$\sum_{j=2}^{n} f(\lfloor\frac{n}{j}\rfloor)$$

only has $2n^{\frac{1}{2}}$ terms we are concerned with and can be split into

$$\sum_{j=2}^{\lfloor n^{\frac{1}{2}}\rfloor} f(\lfloor\frac{n}{j}\rfloor)+\sum_{j=1}^{\lfloor\frac{n}{\lfloor n^{\frac{1}{2}}\rfloor}-1\rfloor}(\lfloor\frac{n}{j}\rfloor-\lfloor\frac{n}{j+1}\rfloor)f(j)\qquad(4.3)$$

We also have to choose a suitable value for *a*. For this paper, *a* will be $n^{\frac{1}{3}}$, so we will need to calculate $d'(n)$ up

to $n^{\frac{1}{3}}$ and $D'(n)$ up to $n^{\frac{2}{3}}$. That task will be covered in the next section. Our final identity for the prime power counting function is thus

$$\begin{aligned}\Pi(n) = \; & n-1+ \\ & \sum_{j=\lfloor n^{\frac{1}{3}}\rfloor+1}^{\lfloor n^{\frac{1}{2}}\rfloor} \sum_{k=2}^{\lfloor \log_2 n\rfloor} \frac{-1^{k+1}}{k} D_{k-1}'(\frac{n}{j}) \\ & \sum_{j=1}^{\lfloor n^{\frac{1}{3}}\rfloor} (\lfloor \frac{n}{j}\rfloor - \lfloor \frac{n}{j+1}\rfloor) \sum_{k=2}^{\lfloor \log_2 n\rfloor} \frac{-1^{k+1}}{k} D_{k-1}'(j) \\ & + \sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor} \sum_{k=2}^{\lfloor \log_2 n\rfloor} \frac{-1^{k+1}}{k} d_{k-1}'(j) D_1'(\frac{n}{j}) \\ & + \sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor} \sum_{s=\lfloor \frac{\lfloor n^{\frac{1}{3}}\rfloor}{j}\rfloor+1}^{\lfloor \frac{n}{j}\rfloor} \sum_{k=2}^{\lfloor \log_2 n\rfloor} \frac{-1^{k+1}}{k} \sum_{m=1}^{k-2} d_m'(j) D_{k-m-1}'(\frac{n}{js}) \\ & + \sum_{j=2}^{\lfloor n^{\frac{1}{3}}\rfloor} \sum_{s=1}^{\lfloor \frac{n}{j}^{\frac{1}{2}}-1} (\lfloor \frac{n}{js}\rfloor - \lfloor \frac{n}{j(s+1)}\rfloor) \cdot \\ & \qquad \sum_{k=2}^{\lfloor \log_2 n\rfloor} \frac{-1^{k+1}}{k} \sum_{m=1}^{k-2} d_m'(j) D_{k-m-1}'(s)\end{aligned} \tag{4.4}$$

Obviously calculating $d'(n)$ up to $n^{\frac{1}{3}}$ can be done relatively quickly. Thus, if you can calculate $D'(n)$ up to $n^{\frac{2}{3}}$ in roughly $O(n^{\frac{2}{3}})$ time, the above equation can be computed in something like $O(n^{\frac{2}{3}}\log n)$ steps because of the final two lines in the equation (in actual practice, with some slight term rearrangement in the two small inner sums and sensible memoization, they can be flattened to a log n-sized sum).

5. Calculating $D_k'(n)$ Up to $n^{\frac{2}{3}}$

To compute $D_k'(n)$ up to $n^{\frac{2}{3}}$ in roughly $O(n^{\frac{2}{3}}\log n)$ time and $O(n^{\frac{1}{3}}\log n)$ space, we will turn to sieving. First, we compute primes up to $n^{\frac{1}{3}}$ - the largest primes needed to sieve numbers $\le n^{\frac{2}{3}}$. We then sieve in blocks of size $n^{\frac{1}{3}}$, establishing our memory boundary. This process is repeated $n^{\frac{1}{3}}$ times. We sieve in such a way that we have the full prime factorization of all entries in each block, with each entry in this form:

$$n = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$$

We will need the power signature of each entry.

The number of divisors function, given the above power signature, is

$$d_k(n) = \binom{a_1+k-1}{a_1} \cdot \binom{a_2+k-1}{a_2} \cdot \binom{a_3+k-1}{a_3} \cdot \ldots \tag{5.1}$$

This gives us the strict number of divisor functions by

$$d_k'(n) = \sum_{j=0}^{k} -1^{k-j} \binom{k}{j} d_j(n) \tag{5.2}$$

So, we use our sieve information to calculate the strict number of divisor function for each entry in the sieve. Since

$$D_k'(n) = D_k'(n-1) + d_k'(n)$$

we can then calculate all values of $D_k'(n)$ in each block. We will have to store off the final values of the $D_k'(n)$ functions at the end of each block to use as the starting values for the next block.

As mentioned, this process runs in something like $O(n^{\frac{2}{3}}\log n)$ time.

6. Conclusion for this Algorithm

The trick to implementing this algorithm is to interleave the sieving described in section 5 with a gradual computation of the sums from (4.4). Essentially, the sums from (4.4) need to be evaluated in order from smallest terms of $D_k'(n)$ to greatest, more or less as a queue. What this means in practice is that for the first two lines,

$$\begin{aligned} & \sum_{j=\lfloor n^{\frac{1}{3}}\rfloor+1}^{\lfloor n^{\frac{1}{2}}\rfloor} -\frac{1}{2} D_1'(\frac{n}{j}) + \frac{1}{3} D_2'(\frac{n}{j}) - \frac{1}{4} D_3'(\frac{n}{j}) + \ldots \\ & + \sum_{j=1}^{\lfloor n^{\frac{1}{2}}\rfloor} (\lfloor \frac{n}{j}\rfloor - \lfloor \frac{n}{(j+1)}\rfloor)(-\frac{1}{2} D_1'(j) + \frac{1}{3} D_2'(j) - \frac{1}{4} D_3'(j) + \ldots)\end{aligned}$$

the second sum will be evaluated first (again, interleaved with the sieving of blocks of $D_k'(n)$ $n^{\frac{1}{3}}$ in size), and, once finished, the first sum will be evaluated with j starting with the value of $n^{\frac{1}{2}}$ and then decreasing until it is $\lfloor n^{\frac{1}{3}}\rfloor + 1$, all interleaved with the sieving. In the C code, you can see this process manually worked through in the function calcS1().

A similar process is necessary for the double sums one the

last two lines of (4.4). In the C code, you can see this process worked through in the function calcS3().

This algorithm can be sped up by using a wheel, which decreases the amount of operations involved in sieving, the double sums calculated in (4.4), and potentially a constant factor in the memory usage as well. C source code for the algorithm is present in an Appendix, but it doesn't implement a wheel and doesn't run as fast as it could.

If anything is too unclear in this description, hopefully browsing the source code in the Appendix will help. Alternatively, the paper in [2] covers many of the same ideas and might be a useful reference for another description of an extremely similar process.

8. References

[1] J. B. Friedlander and H. Iwaniec, *Opera de Cribro*, 346-347
[2] Deleglise, Marc and Rivat, Joel, Computing the summation of the Mobius function. Experiment. Math. 5 (1996), no. 4, 291-295.

Appendix

This is a C implementation of the algorithm described in this paper. Owing to precision issues , it actually stops returning valid values at relatively low values, say around 10^{11} , an eminently fixable problem. This code can be sped up quite a bit, at least in constant terms, by implementing a wheel. There are almost certainly other bits and pieces of this code (particularly in the functions d1 and d2) that can be sped up quite a bit as well, in constant terms.

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "conio.h"
#include "time.h"


typedef long long BigInt;

static 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, -1, 0, 1, -1,
    -1, 0, -1, 1, 0, 0, 1, -1, -1, 0, 0 };

static BigInt* binomials;               /* This is
used as a doubly subscripted array, 128x128.  Indexing
is done manually.*/
static BigInt nToTheThird;
static BigInt logn;

static BigInt numPrimes;
static BigInt* primes;

static BigInt* factorsMultiplied;
static BigInt* totalFactors;
static BigInt* factors;                 /* This is
used as a doubly subscripted array, n^1/3 x ln n.
Indexing is done manually.*/
static BigInt* numPrimeBases;

static BigInt* DPrime;                  /* This is
used as a doubly subscripted array, n^1/3 x ln n.
Indexing is done manually.*/

static BigInt curBlockBase;

static double t;

static BigInt nToTheHalf;
static BigInt numDPowers;
static double* dPrime;

static BigInt S1Val;
static BigInt S1Mode;
static BigInt* S3Vals;
static BigInt* S3Modes;

static bool ended;
static BigInt maxSieveValue;

static BigInt ceilval;

static BigInt n;

BigInt binomial( double n, int k ){
    double t = 1;
    for( int i = 1; i <= k; i++ ){
        t *= ( n - ( k - i ) ) / i;
    }
    return BigInt( t + .1 );
}

static BigInt invpow(double n, double k) {
    return (BigInt)(pow(n, 1.0 / k) + .00000001);
}

/* See
http://www.icecreambreakfast.com/primecount/primecount
ing.html#ch5 for a description of
calculating d_k'(n) from a complete factorization of a
number n.*/
static BigInt d1(BigInt* a, BigInt o, BigInt k, BigInt
l){
    BigInt t = 1;
    for (BigInt j = 0; j < l; j++) t *=
binomials[(a[o*logn+ j] - 1 + k)*128 + a[o*logn+ j]];
    return t;
}

/* See
http://www.icecreambreakfast.com/primecount/primecount
ing.html#ch5 for a description of
calculating d_k'(n) from a complete factorization of a
number n.*/
```

```
static BigInt d2(BigInt* a, BigInt o, BigInt k, BigInt
l, BigInt numfacts ){
    if (numfacts < k) return 0;
    BigInt t = 0;
    for (BigInt j = 1; j <= k; j++) t += ( ( k - j ) %
2 == 1 ? -1:1 ) * binomials[k * 128 + j] * d1(a, o, j,
l);
    if( t < 0 ){
        int asdf  = 9;
    }
    return (BigInt)t;
}

static void allocPools( BigInt n ){
    nToTheThird = (BigInt)pow(n, 1.0 / 3);

    logn = (BigInt)(log(pow(n, 2.00001 / 3)) /
log(2.0)) + 1;
    factorsMultiplied = new BigInt[nToTheThird];
    totalFactors = new BigInt[nToTheThird];
    factors = new BigInt[nToTheThird * logn];
    numPrimeBases = new BigInt[nToTheThird];
    DPrime = new BigInt[(nToTheThird + 1) * logn];
    binomials = new BigInt[128*128+ 128];
    for (BigInt j = 0; j < 128; j++) for (BigInt k =
0; k <= j; k++)binomials[j * 128 + k] = binomial(j,
k);
    for (BigInt j = 0; j < logn; j++) DPrime[j] = 0;
    curBlockBase = 0;

    t = n - 1;

    nToTheHalf = (BigInt)pow(n, 1.0 / 2);
    numDPowers = (BigInt)(log(pow(n, 2.00001 / 3)) /
log(2.0)) + 1;
    dPrime = new double[(nToTheThird + 1) *
(numDPowers + 1)];

    S1Val = 1;
    S1Mode = 0;
    S3Vals = new BigInt[nToTheThird + 1];
    S3Modes = new BigInt[nToTheThird + 1];

    ended = false;
    maxSieveValue = (BigInt)(pow(n, 2.00001 / 3));

    for (BigInt j = 2; j < nToTheThird + 1; j++){
        S3Modes[j] = 0;
        S3Vals[j] = 1;
    }
}

static void deallocPools(){
    delete factorsMultiplied;
    delete totalFactors;
    delete factors;
    delete numPrimeBases;
    delete DPrime;
    delete binomials;
    delete dPrime;
    delete S3Vals;
    delete S3Modes;
    delete primes;
}

/* This finds all the primes less than n^1/3, which
will be used for sieving and generating complete
factorizations of numbers up to n^2/3*/
static void fillPrimes(){
    BigInt* primesieve = new BigInt[nToTheThird + 1];
    primes = new BigInt[nToTheThird + 1];
    numPrimes = 0;
    for (BigInt j = 0; j <= nToTheThird; j++)
primesieve[j] = 1;
    for (BigInt k = 2; k <= nToTheThird; k++){
        BigInt cur = k;
        if (primesieve[k] == 1){
            primes[numPrimes] = k;
            numPrimes++;
            while (cur <= nToTheThird){
                primesieve[cur] = 0;
                cur += k;
            }
        }
    }
    delete primesieve;
}

/* This resets some state used for the sieving and
factoring process.*/
static void clearPools(){
    for (BigInt j = 0; j < nToTheThird; j++){
        numPrimeBases[j] = -1;
        factorsMultiplied[j] = 1;
        totalFactors[j] = 0;
    }
}

/* We can use sieving on our current n^1/3 sized block
of numbers to
get their complete prime factorization signatures,
with which we can then
quickly compute d_k' values.*/
static void factorRange(){
    for (BigInt j = 0; j < numPrimes; j++){
        // mark everything divided by each prime,
adding a new entry.
        BigInt curPrime = primes[j];
        if (curPrime * curPrime > curBlockBase +
nToTheThird) break;
        BigInt curEntry = ( curBlockBase % curPrime ==
0 ) ? 0:curPrime - (curBlockBase % curPrime);
        while (curEntry < nToTheThird){
            if( curEntry+curBlockBase != 0 ){
                factorsMultiplied[curEntry] *=
curPrime;
                totalFactors[curEntry]++;
                numPrimeBases[curEntry]++;
                factors[curEntry*logn+
numPrimeBases[curEntry]] = 1;
            }
            curEntry += curPrime;
        }
        // mark everything divided by each prime power
        BigInt cap = (BigInt)( log((double)
(nToTheThird+curBlockBase)) / log((double)curPrime) +
1 );
        BigInt curbase = curPrime;
        for (BigInt k = 2; k < cap; k++){
            curPrime *= curbase;
            curEntry = (curBlockBase % curPrime ==
0) ? 0 : curPrime - (curBlockBase % curPrime);
            while (curEntry < nToTheThird){
                factorsMultiplied[curEntry] *=
curbase;
                totalFactors[curEntry]++;
                if (curEntry + curBlockBase !=
0)factors[curEntry*logn+ numPrimeBases[curEntry]] = k;
```

```
                curEntry += curPrime;
            }
        }
    }
    // account for prime factors > n^1/3
    for (BigInt j = 0; j < nToTheThird; j++){
        if (factorsMultiplied[j] < j+curBlockBase){
            numPrimeBases[j]++;
            totalFactors[j]++;
            factors[j*logn+ numPrimeBases[j]] = 1;
        }
    }
}

/* By this point, we have already factored, through
sieving, all the numbers in the current n^1/3 sized
block we are looking at.
With a complete factorization, we can calculate
d_k'(n) for a number.
Then, D_k'(n) = d_k'(n) + D_k'(n-1).*/
static void buildDivisorSums(){
    for (BigInt j = 1; j < nToTheThird+1; j++){
        if (j + curBlockBase == 1 || j + curBlockBase
== 2) continue;
        for (BigInt k = 0; k < logn; k++){
            DPrime[j * logn + k] = DPrime[(j - 1) *
logn + k] + d2(factors, j - 1, k, numPrimeBases[j - 1]
+ 1, totalFactors[j - 1]);
        }
    }
    for (BigInt j = 0; j < logn; j++) DPrime[j] =
DPrime[nToTheThird*logn+ j];
}

/* This general algorithm relies on values of D_k' <=
n^2/3 and d_k' <= n^1/3.  This function calculates
those values of d_k'.*/
static void find_dVals(){
    curBlockBase = 1;
    clearPools();
    factorRange();
    buildDivisorSums();

    for (BigInt j = 2; j <= nToTheThird; j++){
        for (BigInt m = 1; m < numDPowers; m++){
            double s = 0;
            for (BigInt r = 1; r < numDPowers; r++) s
+= pow(-1.0, (double)( r + m )) * (1.0 / (r + m + 1))
* (DPrime[j * logn + r] - DPrime[(j - 1) * logn + r]);
            dPrime[j*(numDPowers + 1)+ m] = s;
        }
    }
}

static void resetDPrimeVals(){
    curBlockBase = 0;
    for (BigInt k = 0; k < nToTheThird + 1; k++)
        for (BigInt j = 0; j < logn; j++)
            DPrime[k * logn + j] = 0;
}

/* This function is calculating the first two sums of
http://www.icecreambreakfast.com/primecount/primecount
ing.html#4_4
It is written to rely on values of D_k' from smallest
to greatest, to use the segmented sieve.*/
static void calcS1(){
    if (S1Mode == 0){
        while (S1Val <= ceilval){
            BigInt cnt = (n / S1Val - n / (S1Val +
1));
            for (BigInt m = 1; m < numDPowers; m++) t
+= cnt * (m % 2 == 1 ? -1 : 1) * (1.0 / (m + 1)) *
DPrime[(S1Val - curBlockBase + 1) * logn + m];
            S1Val++;
            if (S1Val >= n / nToTheHalf){
                S1Mode = 1;
                S1Val = nToTheHalf;
                break;
            }
        }
    }
    if (S1Mode == 1){
        while (n / S1Val <= ceilval){
            for (BigInt m = 1; m < numDPowers; m++) t
+= (m % 2 == 1 ? -1 : 1) * (1.0 / (m + 1)) * DPrime[(n
/ S1Val - curBlockBase + 1) * logn + m];
            S1Val--;
            if (S1Val < nToTheThird + 1){
                S1Mode = 2;
                break;
            }
        }
    }
}

/* This loop is calculating the 3rd term that runs
from 2 to n^1/3 in
http://www.icecreambreakfast.com/primecount/primecount
ing.html#4_4*/
static void calcS2(){
    for (BigInt j = 2; j <= nToTheThird; j++)
        for (BigInt k = 1; k < numDPowers; k++)
            t += (n / j - 1) * pow(-1.0, (double)k) *
(1.0 / (k + 1)) * (DPrime[j * logn + k] - DPrime[(j -
1) * logn + k]);
}

/* This loop is calculating the two double sums in
http://www.icecreambreakfast.com/primecount/primecount
ing.html#4_4
It is written to rely on values of D_k' from smallest
to greatest, to use the segmented sieve.*/
static void calcS3(){
    for (BigInt j = 2; j <= nToTheThird; j++){
        if (S3Modes[j] == 0){
            BigInt endsq = (BigInt)(pow(n / j, .5));
            BigInt endVal = (n / j) / endsq;
            while (S3Vals[j] <= ceilval){
                BigInt cnt = (n / (j * S3Vals[j]) -
n / (j * (S3Vals[j] + 1)));
                for (BigInt m = 1; m < numDPowers; m+
+) t += cnt * DPrime[(S3Vals[j] - curBlockBase + 1) *
logn + m] * dPrime[j*(numDPowers + 1)+ m];
                S3Vals[j]++;
                if (S3Vals[j] >= endVal){
                    S3Modes[j] = 1;
                    S3Vals[j] = endsq;
                    break;
                }
            }
        }
        if (S3Modes[j] == 1){
            while (n / (j * S3Vals[j]) <= ceilval){
                for (BigInt m = 1; m < numDPowers; m+
+) t += DPrime[(n / (j * S3Vals[j]) - curBlockBase +
1) * logn + m] * dPrime[j * (numDPowers + 1) + m];
                S3Vals[j]--;
```

```
                if (S3Vals[j] < nToTheThird / j + 1){
                    S3Modes[j] = 2;
                    break;
                }
            }
        }
    }
}

/*      This is the most important function here. How
it works:
*        first we allocate our n^1/3 ln n sized pools
and other variables.
*        Then we go ahead and sieve to get our primes
up to n^1/3
*        We also calculate, through one pass of
sieving, values of d_k'(n) up to n^1/3
*        Then we go ahead and calculate the loop S2
(check the description of the algorithm), which only
requires
*        values of d_k'(n) up to n^1/3, which we
already have.
*        Now we're ready for the main loop.
*        We do the following roughly n^1/3 times.
*        First we clear our sieving variables.
*        Then we factor, entirely, all of the numbers
in the current block sized n^1/3 that we're looking
at.
*        Using our factorization information, we
calculate the values for d_k'(n) for the entire range
we're looking,
*        and then sum those together to have a rolling
set of D_k'(n) values
*        Now we have values for D_k'(n) for this block
sized n^1/3
*        First we see if any of the values of S1 that
we need to compute are in this block. We can do this
by
*        (see the paper) walking through the two S1
loops backwards, which will use the D_k'(n)
*        values in order from smallest to greatest
*        We then do the same thing will all of the S3
values
*        Once we have completed this loop, we will
have calculated the prime power function for n.
*
*       This loop is essentially calculating
*

http://www.icecreambreakfast.com/primecount/primecount
ing.html#4_4
*/

static double calcPrimePowerCount(BigInt nVal){
    n = nVal;
    allocPools(n);
    fillPrimes();
    find_dVals();
    calcS2();
    resetDPrimeVals();

    for (curBlockBase = 0; curBlockBase <=
maxSieveValue; curBlockBase += nToTheThird ){
        clearPools();
        factorRange();
        buildDivisorSums();

        ceilval = curBlockBase + nToTheThird - 1;
        if (ceilval > maxSieveValue) {
            ceilval = maxSieveValue;
            ended = true;
        }

        calcS1();
        calcS3();
        if (ended) break;
    }

    deallocPools();

    return t;
}

static BigInt countprimes(BigInt num) {
    double total = 0.0;
    for (BigInt i = 1; i < log((double)num) /
log(2.0); i++) {
        double val = calcPrimePowerCount( invpow(num,
i)) / (double)i * mu[i];
        total += val;
    }
    return total+.1;
}

int scaleNum = 10;
int main(int argc, char* argv[]){
    int oldClock = (int)clock();
    int lastDif = 0;

    printf( "
Time\n");
    printf( "
Increase\n");
    printf( "
for x%d\n", scaleNum);
    printf( "          __ Input Number __    __ Output
Number __ _ MSec _ _ Sec _  Input\n");
    printf( "
\n");
    for( BigInt i = scaleNum; i <=
1000000000000000000; i *= scaleNum ){
        printf( "%17I64d(10^%4.1f): ", i,
log( (double)i )/log(10.0) );
        BigInt total = (BigInt)(countprimes( i )
+.00001);
        int newClock = (int)clock();
        printf( " %20I64d %8d : %4d: x%f\n",
            total, newClock - oldClock, ( newClock -
oldClock ) / CLK_TCK,
            ( lastDif ) ? (double)( newClock -
oldClock ) / (double)lastDif : 0.0 );
        lastDif = newClock - oldClock;
        oldClock = newClock;
    }

    getch();

    return 0;
}
```