Linnik's Identity
and
Various Explicit Prime Counting Formulas

Nathan McKenzie
kenzi_x@yahoo.com
August 2009

[0] Contents

[1] Slight Preliminaries

So, I start with Linnik's identity, which Iwaniec gives as

$$\Lambda'(n) = -\sum_k \frac{(-1)^k}{k} \tau'_k(n).$$

where

$$\tau'_k(n) = |\{n_1, \ldots, n_k \geqslant 2; \;\; n_1 \ldots n_k = n\}|.$$

and

$$\Lambda'(n) = \frac{\Lambda(n)}{\log n} = \begin{cases} a^{-1} & \text{if } n = p^a, a > 0, \\ 0 & \text{otherwise} \end{cases}$$

or, to restate (where #{...} will mean the count of the set satisfying the equations contained within and where n is an integer),

$$\begin{aligned} & \#\{ n_1 = n \quad : n_k >= 2 \} \\ - \; & \tfrac{1}{2} \#\{ n_1 n_2 = n \quad : n_k >= 2 \} \\ + \; & \tfrac{1}{3} \#\{ n_1 n_2 n_3 = n \quad : n_k >= 2 \} \\ - \; & \ldots \\ = \; & a^{-1} \text{ if } n = p^a, \text{ 0 otherwise} \end{aligned} \tag{1.1}$$

If the leading fractions are dropped, we're also left with

$$
\begin{aligned}
& \#\{\, n_1 = n \qquad : n_k >= 2 \,\} \\
- \; & \#\{\, n_1 n_2 = n \qquad : n_k >= 2 \,\} \\
+ \; & \#\{\, n_1 n_2 n_3 = n \qquad : n_k >= 2 \,\} \\
- \; & \ldots \\
= \; & -\mu(n)
\end{aligned}
\tag{1.2}
$$

[2] Explicit formula for π*

If we sum Linnik's identity from 2 to n, we get

$\pi^*(n) =$

$$
\begin{aligned}
& \#\{\, n_1 <= n \qquad : n_k >= 2 \,\} \\
- \; & \tfrac{1}{2} \; \#\{\, n_1 n_2 <= n \qquad : n_k >= 2 \,\} \\
+ \; & \tfrac{1}{3} \; \#\{\, n_1 n_2 n_3 <= n \qquad : n_k >= 2 \,\} \\
- \; & \ldots
\end{aligned}
\tag{2.1}
$$

where

$$\pi^*(n) = \pi(n) + \tfrac{1}{2}\,\pi(n^{1/2}) + \tfrac{1}{3}\pi(n^{1/3}) + \ldots$$

So, this turns the task of calculating π^* into the task of calculating the number of lattice points in several hyperbolic lattices, and, in particular, it opens the door to some interesting ways of counting primes that don't involve using any prime numbers, at all, in the process of counting.

I have included, in section [6], the source code to a C algorithm that counts primes by counting the volumes of these various hyperbolic lattices; the algorithm seems to run in roughly $O(n^{.8})$ execution time and $O(\varepsilon)$ space, although the time bound has been arrived at empirically.

This is one of the areas of this paper where my status as an amateur is, I think, hindering my results. The algorithms I use to count the volume of these lattices are entirely my own; it's entirely possible that much faster lattice counting algorithms exist, in which case this might be a much faster way to count primes than I am presenting. I have been unable to track down any such algorithms, and would be grateful to know of better ones.

Similarly, if we sum equation (1.2) for 2 to N, we arrive at a similar identity, with similar implications about computational methods:

1 - Mertens(n)=

$$
\begin{aligned}
& \#\{\, n_1 <= n \qquad : n_k >= 2 \,\} \\
- \; & \#\{\, n_1 n_2 <= n \qquad : n_k >= 2 \,\} \\
+ \; & \#\{\, n_1 n_2 n_3 <= n \qquad : n_k >= 2 \,\} \\
- \; & \ldots
\end{aligned}
\tag{2.2}
$$

[3] Approximations connected to [2]

It is possibly interesting to take the various lattices from (2.1) above and to examine the real-valued curves that bound them. Thus we can examine the integral volumes of the areas bounded by

$$
\begin{aligned}
& \{ x_1 <= n \quad : x_k >= 1 \} \\
- \tfrac{1}{2} & \{ x_1 x_2 <= n \quad : x_k >= 1 \} \\
+ \tfrac{1}{3} & \{ x_1 x_2 x_3 <= n \quad : x_k >= 1 \} \\
- \tfrac{1}{4} & \{ x_1 x_2 x_3 x_4 <= n \quad : x_k >= 1 \} \\
& + \ldots
\end{aligned}
\tag{3.1}
$$

Integrating, we get

$$
\begin{aligned}
& (n-1) \\
& - \tfrac{1}{2} (n \log n - n + 1) \\
& + \tfrac{1}{3} (\tfrac{1}{2} n (\log n)^2 - n \log n + n - 1) \\
& - \tfrac{1}{4} (1/6 \, n (\log n)^3 - \tfrac{1}{2} n (\log n)^2 + n \log n - n + 1) \\
& + \ldots
\end{aligned}
\tag{3.2}
$$

Empirically, (3.2) appears to be exactly equal to Li(n) + some constant.

Consequently, the difference between π*(n) and Li(n) appears to be representable as the volumes present in these real-valued hyperbolas that are not present in the corresponding integer lattices, or, which is to say,

$$
\begin{aligned}
& Li(n) + C - \pi^*(n) = \\
& \quad (n-1 - && \#\{ n_1 <= n : n_k >= 2 \}) \\
& - \tfrac{1}{2} (n \log n - n + 1 - && \#\{ n_1 n_2 <= n : n_k >= 2 \}) \\
& + \tfrac{1}{3} (\tfrac{1}{2} n (\log n)^2 - n \log n + n - 1 - && \#\{ n_1 n_2 n_3 <= n : n_k >= 2 \}) \\
& - \tfrac{1}{4} (1/6 \, n (\log n)^3 - \tfrac{1}{2} n (\log n)^2 + n \log n - n + 1 - && \#\{ n_1 n_2 n_3 n_4 <= n : n_k >= 2 \}) \\
& + \ldots
\end{aligned}
\tag{3.3}
$$

This seems to suggest an interesting approximation for π* relative to Li(n) by using analytical methods to approximate the error term for each of these lattice/hyperbola pairs and then adding or subtracting those terms from Li(n) to approximate π* or, which is to say,

$$
\begin{aligned}
& \pi^*(n) = Li(n) + C \\
& \quad + \tfrac{1}{2} \text{ (approximate error for 2 variable hyperbola)} \\
& \quad - \tfrac{1}{3} \text{ (approximate error for 3 variable hyperbola)} \\
& \quad + \tfrac{1}{4} \text{ (approximate error for 4 variable hyperbola)} \\
& \quad - \ldots \\
& \quad + O(?)
\end{aligned}
\tag{3.4}
$$

Unfortunately, although I would be very interested in exploring such an approximation, it is beyond my

rather paltry analytic number theory skills to go any further down this road.

It might be equally interesting, of course, to attempt a comparable approximation with Mertens function, starting with (2.2), as well.

[4] Recursive formula for π*

As should perhaps not be surprising, the various lattices from (2.1) above are by no means uncorrelated. In fact, when arranged appropriately, a certain kind of self-similarity appears. At present I can describe this self-similarity visually, but I'm not sure I can describe it mathematically. At any rate, by rearranging the various lattices correctly, the following recursive equation can be deduced:

$$\pi^*(n, k) = \frac{n-1}{k} - \sum_{j=2} \pi^*(\lfloor \frac{n}{j} \rfloor, k+1) \tag{4.1}$$

In this context, π*(n) as is commonly used means π*(n, 1).

By a similar process, (2.2) can be converted to produce

$$f(n) = n-1-\sum_{j=2} f(\lfloor \frac{n}{j} \rfloor) \tag{4.2}$$

Where Mertens(n) = 1-f(n).

Also potentially interesting in this context is the following

$$\lim_{k\to\infty} \pi^*(n, k) = \frac{f(n)}{k} \tag{4.3}$$

My experiments with these functions lead me to believe they are less useful as workhorses for computational methods than the lattices found in section [2].

[5] Approximations connected to [4]

The recursive function for π* (4.1) can also be approximated by removing the floor function. Thus we have

$$s(n, k) = \frac{n-1}{k} - \sum_{j=2} s(\frac{n}{j}, k+1) \tag{5.1}$$

s(n,1) also has, at least empirically, a very close relationship to Li(n)... This leads to another possibly interesting avenue for approximating π*, primarily by trying to approximate the difference between s(n,k) and π*(n,k). One way to do this is to try to approximate the following values:

$$v_2(n) = \sum_{j=2}^{n} \frac{n}{j} - \sum_{j=2}^{n} \lfloor \frac{n}{j} \rfloor \tag{5.2}$$

$$v_3(n) = \sum_{j=2}^{n} \sum_{k=2}^{\frac{n}{j}} \frac{n}{jk} - \sum_{j=2}^{n} \sum_{k=2}^{\frac{n}{j}} \lfloor \frac{\lfloor \frac{n}{j} \rfloor}{k} \rfloor \tag{5.3}$$

$$v_4(n) = \sum_{j=2}^{n} \sum_{k=2}^{\frac{n}{j}} \sum_{m=2}^{\frac{n}{jk}} \frac{n}{jkm} - \sum_{j=2}^{n} \sum_{k=2}^{\frac{n}{j}} \sum_{m=2}^{\frac{\lfloor \frac{n}{j} \rfloor}{k}} \frac{\lfloor \frac{\lfloor \frac{n}{j} \rfloor}{k} \rfloor}{m} \tag{5.4}$$

... and so on.

We would then have, similar to (3.4) above,

$$\pi^*(n,1) = s(n,1) + \tfrac{1}{2}\, v_2(n) - \tfrac{1}{3}\, v_3(n) + \tfrac{1}{4}\, v_4(n) - \ldots + O(?) \tag{5.5}$$

Once again, this is the point where my own relatively poor analytical abilities end my exploration. Empirically, $s(n,1)$, $v_2(n)$, $v_3(n)$, $v_4(n)$, and so on all look like relatively well-behaved curves with some slight but bounded scattering, with $s(n,1)$ resembling $Li(n)$, $v_2(n)$ close to $(1-\gamma)n$, $v_3(n)$close to $n \log n * C$, $v_4(n)$close to $n (\log n)^2 * C$, and so on. It's also clear that, in absolutely terms, the magnitude of these errors are smaller than the errors present in (3.3) – but that's not to say that, ultimately, their O(...) is any better.

[6] Algorithm and code for computing count of primes

The following C code calculates the number of primes less than some value n using the lattice identity from (2.1). It appears to run in roughly $O(n^{.8})$ time and $O(\varepsilon)$ space. Two main methods are used to speed computation up. First, internal symmetries in the lattices are used, meaning that, for example,

$$\#\{\, n_1 n_2 n_3 n_4 n_5 <= n \quad : n_k >= 2 \,\}$$

can be counted with loops that look, roughly, like this

$$\sum_{j_1=2}^{n^{\frac{1}{5}}} \sum_{j_2=j_1}^{(\frac{n}{j_1})^{\frac{1}{4}}} \sum_{j_3=j_2}^{(\frac{n}{j_1 * j_2})^{\frac{1}{3}}} (combinatorialValues + \sum_{j_4=j_3}^{(\frac{n}{j_1 * j_2 * j_3})^{\frac{1}{2}}} combinatorialValue * \lfloor \frac{n}{j_1} \rfloor)$$

Second, very large wheels can be used, which help reduce the size of, especially, the particularly slow outer loops.

This still isn't, unfortunately, enough to make the algorithm competitive as it is with the fastest prime counting algorithms...

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

const double EPSILON = .00000000001;
typedef long long BigInt;
/* A wheel including 19 has 9.6 million entries.   Mem usage == wheel cycle period * 12 bytes at peak, it looks like... right?
Right now it looks like it uses about 45 megs of ram.*/
const int g_wheelLargestPrime = 19;
/* Used for the construction of the wheel - include more primes as needed, but this is already enough primes to consume
over 75 gigs of RAM */
const int g_primes[] ={ 2,   3,   5,   7,  11,  13,  17,  19,  23,  29 };
int g_wheelCycleEntries;
int g_wheelCyclePeriod;
int g_wheelFirstPrime;
int g_wheelBasePrimes;
int* g_wheelTranslation = 0;
int* g_wheelOffsets = 0;
BigInt g_latticepoints;
BigInt g_minVarValue;
BigInt g_boundary;
BigInt g_scale;
BigInt g_divisor;
BigInt g_lastMax;
int g_variablesLeft;
BigInt g_lastScaleDivisor;
BigInt g_scaleVal;

int g_moo[] = {
  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,   1,  -1,   0,   1,   1,   1,   0,  -1,
  0,   1,   0,   1,   1,   1,   0,  -1,   0,   0,
  0,  -1,  -1,  -1,   0,  -1,   1,  -1,   0,  -1,
 -1,   1,   0,  -1,  -1,   1,   0,   0,   1,   1,
  0,   0,   1,   1,   0,   0,   0,  -1,   0,   1,
 -1,  -1,   0,   1,   1,   0,   0,  -1,  -1,  -1,
  0,   1,   1,   1,   0,   1,   1,   0,   0,  -1,
  0,  -1,   0,   0,  -1,   1,   0,  -1,   1,   1,
  0,   1,   0,  -1,   0,  -1,   1,  -1,   0,   0,
 -1,   0,   0,  -1,  -1,   0,   0,   1,   1,  -1,
  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,   1,   1,   0,   0,   1,
  1,  -1,   0,   1,   1,   1,   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,  -1,   0,   0,   0,   0,  -1,   1,   0,   1,
```

```
 0, -1,  0,  1,  1, -1,  0
};
BigInt g_factorial[] = { /* Note that with 64 bit ints, we can't go above factorial( 20 ) anyway. */
        0, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600,
        6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000,
        6402373705728000, 121645100408832000, 2432902008176640000
};

inline BigInt InversePower( BigInt x, BigInt y ){
        return ( (BigInt)( pow( (double)x + EPSILON, ( 1.0 / (double)y ) ) + EPSILON ) );
}

//_____________________________________________________
//                                      Wheel Functions
//_____________________________________________________

inline int GetwheelCyclePeriod( int cap ){
        int val = 1;
        int i   = 0;
        while( g_primes[ i ] <= cap ){
                val *= g_primes[ i ];
                i++;
        }
        return val;
}
inline int GetFirstIncludedPrime( int cap ){
        int i = 0;
        while( g_primes[ i ] <= cap )i++;
        return g_primes[ i ];
}
inline int GetBasePrimes( int cap ){
        int i = 0;
        while( g_primes[ i ] <= cap )i++;
        return i;
}
inline void IncrementWheel( int &offset ){
        offset++;
        if( offset >= g_wheelCycleEntries )offset = 0;
}
void MakeWheel( int cap ){
        g_wheelBasePrimes = GetBasePrimes( cap );
        g_wheelCyclePeriod = GetwheelCyclePeriod( cap );
        g_wheelCycleEntries = 0;
        int cur = 0;
        int offset          = -1;
        int* wheelBase = 0;
        wheelBase = ( int* )malloc( g_wheelCyclePeriod * sizeof( int ) );
        g_wheelTranslation = ( int* )malloc( ( g_wheelCyclePeriod + 1 ) * sizeof( int ) );
        g_wheelOffsets = ( int * )malloc( g_wheelCyclePeriod * sizeof( int ) );
        g_wheelFirstPrime = GetFirstIncludedPrime( cap );
        for( int i = 0; i < g_wheelCyclePeriod; i++ ){
                wheelBase[ i ] = 1;
                for( int j = 2; j <= cap; j++ ){
                        if( !( ( i + 1 ) % j ) ){
                                wheelBase[ i ] = 0;
                                break;
                        }
                }
```

```
	}
	while( cur < g_wheelCyclePeriod ){
		if( wheelBase[ cur ] && cur != 0 ){
			g_wheelOffsets[ g_wheelCycleEntries ] = offset + 1;
			offset = 0;
			g_wheelCycleEntries++;
		}
		else offset++;
		cur++;
	}
	g_wheelOffsets[ g_wheelCycleEntries ] = 2;
	g_wheelCycleEntries++;
	int total = 0;
	g_wheelTranslation[ 0 ] = 0;
	for( int i = 0; i < g_wheelCyclePeriod; i++ ){
		if( i && wheelBase[ i - 1 ] )total++;
		g_wheelTranslation [ i + 1 ] = total;
	}
	free( wheelBase );
}
/* This function calculates how many entries the wheel leaves in the range from (rangeStart, rangeStop].*/
inline BigInt CountWheelEntries( BigInt rangeStart, BigInt rangeEnd ){
	rangeEnd++;
	int a = rangeStart % g_wheelCyclePeriod;
	int b = rangeEnd % g_wheelCyclePeriod;
	return ( rangeEnd - b - rangeStart + a ) / g_wheelCyclePeriod * g_wheelCycleEntries + g_wheelTranslation[ b ] -
g_wheelTranslation[ a ] ;
}

//_____________________________________________________________
//                                    Lattice Counting Functions
//_____________________________________________________________

void CountHyperbolaLattice_2_Variables( void ){
	BigInt finalBoundary = g_boundary / g_minVarValue;
	BigInt boundaryRoot = (BigInt)( sqrt( (double)finalBoundary ) + EPSILON );
	/* For if the final two digits happen to be the same. */
	g_latticepoints += g_scale / ( g_divisor * ( g_divisor + 1 ) );
	/* Leading digit is same, final digit is not. */
	g_latticepoints +=  ( CountWheelEntries( g_minVarValue, finalBoundary / g_minVarValue ) - 1 ) * ( g_scale /
g_divisor );
	/* For if the final two digits happen to be the same,but both differ from the previous. */
	g_latticepoints += ( CountWheelEntries( g_minVarValue, boundaryRoot ) - 1 ) * ( g_scale / 2 );
	/* Both digits differ from all other digits - This is the hellish evil loop of portentious doom. */
	int curWheelOffset = g_wheelTranslation[ g_minVarValue % g_wheelCyclePeriod ];
	BigInt curLeadingVar = g_minVarValue + g_wheelOffsets[ curWheelOffset ];
	BigInt subTotal = 0;
	IncrementWheel( curWheelOffset );
	while( curLeadingVar <= boundaryRoot ){
		subTotal += CountWheelEntries( curLeadingVar, finalBoundary / curLeadingVar ) - 1;
		curLeadingVar += g_wheelOffsets[ curWheelOffset ];
		IncrementWheel( curWheelOffset );
	}
	g_latticepoints += subTotal * g_scale;
}
void CountHyperbolaLattice_3_Variables( BigInt hyperbolaBoundary, BigInt minVarValue ){
	BigInt maxVarValue = InversePower( hyperbolaBoundary, 3 );
	int curWheelOffset = g_wheelTranslation[ minVarValue % g_wheelCyclePeriod ];
```

```
		g_boundary = hyperbolaBoundary;
		g_minVarValue = minVarValue;
		g_scale = g_scaleVal / g_lastScaleDivisor;
		g_divisor = g_lastScaleDivisor + 1;
		CountHyperbolaLattice_2_Variables();
		g_minVarValue += g_wheelOffsets[ curWheelOffset ];
		IncrementWheel( curWheelOffset );
		g_scale = g_scaleVal;
		g_divisor = 2;
		while( g_minVarValue <= maxVarValue ){
			CountHyperbolaLattice_2_Variables();
			g_minVarValue += g_wheelOffsets[ curWheelOffset ];
			IncrementWheel( curWheelOffset );
		}
}
void CountHyperbolaLattice_X_Variables( BigInt hyperbolaBoundary, BigInt minVarValue ){
		BigInt maxVarValue = InversePower( hyperbolaBoundary, g_variablesLeft );
		BigInt scaleVal = g_scaleVal;/* Save global variables that will be restored at end of function */
		BigInt lastScaleDivisor = g_lastScaleDivisor;
		int curWheelOffset = g_wheelTranslation[ minVarValue % g_wheelCyclePeriod ];
		g_variablesLeft--;
		g_lastScaleDivisor = lastScaleDivisor + 1;
		g_scaleVal = scaleVal / lastScaleDivisor;
		if( g_variablesLeft == 3 ){ CountHyperbolaLattice_3_Variables( hyperbolaBoundary / minVarValue,
minVarValue );}
		else{CountHyperbolaLattice_X_Variables( hyperbolaBoundary / minVarValue, minVarValue );}
		g_lastScaleDivisor = 2;
		g_scaleVal = scaleVal;
		minVarValue += g_wheelOffsets[ curWheelOffset ];
		IncrementWheel( curWheelOffset );
		while( minVarValue <= maxVarValue ){
			if( g_variablesLeft == 3 ){CountHyperbolaLattice_3_Variables( hyperbolaBoundary / minVarValue,
minVarValue );}
			else{ CountHyperbolaLattice_X_Variables( hyperbolaBoundary / minVarValue, minVarValue );}
			minVarValue += g_wheelOffsets[ curWheelOffset ];
			IncrementWheel( curWheelOffset );
		}
		g_lastScaleDivisor = lastScaleDivisor;/* Restore global variables */
		g_variablesLeft++;
}
BigInt CountHyperbolaLattice( BigInt hyperbolaBoundary, int hyperbolaVariables ){
		g_latticepoints		= 0;
		g_variablesLeft		= hyperbolaVariables;
		g_lastScaleDivisor	= 1;
		g_scaleVal			= g_factorial[ hyperbolaVariables ];
		if( hyperbolaBoundary < (BigInt)pow( (double)g_wheelFirstPrime, (double)hyperbolaVariables ) ){return 0;}
		switch( hyperbolaVariables ){
		case 1: g_latticepoints = CountWheelEntries( g_wheelFirstPrime, hyperbolaBoundary ); break;
		case 2: /* CountHyperbolaLattice_2_Variables expects a number of global variables to be initialized when it is
called, which generally happens in CountHyperbolaLattice_3_Variables.  We have to do it manually here. */
			g_minVarValue	= g_wheelFirstPrime;
			g_boundary		= g_wheelFirstPrime * hyperbolaBoundary;
			g_scale			= 2;
			g_divisor		= 1;
			CountHyperbolaLattice_2_Variables();
			break;
		case 3: CountHyperbolaLattice_3_Variables( hyperbolaBoundary, g_wheelFirstPrime ); break;
		default: CountHyperbolaLattice_X_Variables( hyperbolaBoundary, g_wheelFirstPrime ); break;
```

```
	}
	return g_latticepoints;
}

//_____________________________________________________________
//                                          Main Functions
//_____________________________________________________________

BigInt CountPrimes( BigInt numLimit ){
	int maxPower = ( int )( log( ( double )numLimit + EPSILON ) / log ( ( double )g_wheelFirstPrime + EPSILON ) +
EPSILON ) + 1;
	double total = 0.0;
	int oldClock = (int)clock();
	int totalTime = 0;
	for( int curPower = 1; curPower < maxPower; curPower++ ){
		if( !g_moo[ curPower ] ){continue;}
		BigInt curMax = InversePower( numLimit, curPower );
		double subTotal = 0.0;
		BigInt hyperbolaEntries = 1;
		double sign = 1;
		while( 1 ){
			double temp        = sign / hyperbolaEntries;
			sign       *= -1;
			// This will start failing when curmax is greater than 23^21, because the factorial table only goes
to 20.....but that's a REALLY big number.
			temp *= (double)CountHyperbolaLattice( curMax, hyperbolaEntries );
			if( temp == 0.0 )break;
			subTotal += temp;
			hyperbolaEntries++;
			int newClock       = (int)clock();
			totalTime += newClock - oldClock;
			oldClock             = newClock;
		}
		subTotal /= curPower * g_moo[ curPower ];
		total += subTotal;
	}
	total += g_wheelBasePrimes;
	return (BigInt)( total + 0.5 );
}/* the .5 is to prevent truncation errors - but it's clearly sloppy */

int main(int argc, char** argv){
	MakeWheel( g_wheelLargestPrime );
	int oldClock = (int)clock();
	int lastDif = 0;
	for( BigInt i = 10; i <= 100000000000000; i *= 10 ){
		printf( "%20I64d(%4.1f): ", i, log( (double)i ) );
		BigInt total = CountPrimes( i );
		int newClock = (int)clock();
		printf( " %20I64d %8d : %4d: %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;}
```