Bounds of the Mertens Function

Darrell Cox

Abstract

A likely upper bound of the absolute value of the Mertens function ($\sqrt{\log(x!)} > |M(x)|$ when $x > 1$) is discussed.

1 Introduction

Let $M(x)$ denote the Mertens function ($M(x) = \sum_{i=1}^{x} \mu(i)$ where $\mu(i)$ is the Möbius function). Littlewood [1] proved that the Riemann hypothesis is equivalent to the statement that for every $\epsilon > 0$ the function $M(x)x^{-(1/2)-\epsilon}$ approaches zero as $x \to \infty$. Mertens conjectured that $|M(x)| < \sqrt{x}$. This was disproved by Odlyzko and te Riele [2]. The Stieltjes hypothesis states that $M(x) = O(x^{\frac{1}{2}})$.

2 A Likely Upper Bound of $|M(x)|$

Lehman [3] proved that $\sum_{i=1}^{x} M(\lfloor x/i \rfloor) = 1$. In general, $\sum_{i=1}^{x} M(\lfloor x/(in) \rfloor) = 1$, $n = 1, 2, 3, ..., x$ (since $\lfloor \lfloor x/n \rfloor / i \rfloor = \lfloor x/(in) \rfloor$). Let R' denote a square matrix where element (i, j) equals 1 if j divides i or 0 otherwise. (In a Redheffer matrix, element (i, j) equals 1 if i divides j or if $j = 1$. Redheffer [4] proved that the determinant of such a x by x matrix equals $M(x)$.) Let T denote the matrix obtained from R' by element-by-element multiplication of the columns by $M(\lfloor x/1 \rfloor)$, $M(\lfloor x/2 \rfloor)$, $M(\lfloor x/3 \rfloor)$, ..., $M(\lfloor x/x \rfloor)$. For example, the T matrix for $x = 12$ is

-2	0	0	0	0	0	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0	0	0	0	0
-1	0	-1	0	0	0	0	0	0	0	0	0
-1	-1	0	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0
1	1	0	1	0	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0	1	0	0	0
1	1	0	0	1	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	1	0	0	0	0	0	1

Let $A(x) = \sum_{i=1}^{x} \phi(i)$ where ϕ is Euler's totient function. Let U denote the matrix obtained from T by element-by-element multiplication of the columns by $\phi(j)$. The sum of the columns of U then equals $A(x)$. $i = \sum_{d|i} \phi(d)$, so $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)i$ (the sum of the rows of U) equals $A(x)$.

Theorem (1) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)i = A(x)$

By the Schwarz inequality, $A(x)/\sqrt{x(x+1)(2x+1)/6}$ is a lower bound of $\sqrt{\sum_{i=1}^{x} M(\lfloor x/i \rfloor)^2}$. Let $\Lambda(i)$ denote the Mangoldt function ($\Lambda(i)$ equals $\log(p)$ if $i = p^m$ for some prime p and some $m \geq 1$ or 0 otherwise). Mertens [5] proved that $\sum_{i=1}^{x} M(\lfloor x/i \rfloor) \log(i) = \psi(x)$ where $\psi(x)$ denotes the second Chebyshev function ($\psi(x) = \sum_{i \leq x} \Lambda(i)$). Let $\sigma_x(i)$ denote the sum of positive divisors function ($\sigma_x(i) = \sum_{d|i} d^x$). Replacing $\phi(j)$ with $\log(j)$ in the U matrix gives a similar result.

Theorem (2) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor) \log(i)\sigma_0(i)/2 = \log(x!)$

Let $\lambda(n)$ denote the Liouville function ($\lambda(1) = 1$ or if $n = p_1^{a_1} \cdots p_k^{a_k}$, $\lambda(n) = (-1)^{a_1+\ldots+a_k}$). $\sum_{d|n} \lambda(d)$ equals 1 if n is a perfect square or 0 otherwise. Let $L(x) = \sum_{n \leq x} \lambda(n)$. Let $H(x) = \sum_{n \leq x} \mu(n) \log(n)$. $H(x)/(x \log(x)) \to 0$ as $x \to \infty$ and $\lim_{x\to\infty}(M(x)/x - H(x)/(x \log(x))) = 0$. The statement $\lim_{x\to\infty} M(x)/x = 0$ is equivalent to the prime number theorem. Also, $\Lambda(n) = -\sum_{d|n} \mu(d) \log(d)$. (See pp. 91-92 of Apostol's [6] book.) Other relationships that can be derived using the T matrix are;

Theorem (3) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)\sigma_0(i) = x$

Theorem (4) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)\sigma_1(i) = x(x+1)/2$

Theorem (5) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)\sigma_2(i) = x(x+1)(2x+1)/6$

Theorem (6) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)$ where the summation is over i values that are perfect squares equals $L(x)$

Theorem (7) $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)\Lambda(i) = -H(x)$

The following conjecture is based on data collected for $x \leq 500,000$.

Conjecture (1) $\log(x!) > \sum_{i=1}^{x} M(\lfloor x/i \rfloor)^2 > \psi(x)$ when $x > 7$

By Stirling's formula, $\log(x!) = x\log(x) - x + O(\log(x))$. Since $\log(x)$ increases more slowly than any positive power of x, this is a better upper bound of $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)^2$ than $x^{1+\epsilon}$ for any $\epsilon > 0$. See Figure 1 for a plot of $\log(x!)$, $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)^2$, and $\psi(x)$ for x = 1, 2, 3, ..., 1000.

Let $j(x) = \sum_{i}^{x} M(x/i)^2$ where the summation is over i values where $i|x$. Let l_1, l_2, l_3, ... denote the x values where $j(x)$ is a local maximum (that is, greater than all preceding $j(x)$ values) and let m_1, m_2, m_3, ... denote the values of the local maxima. The local maxima occur at x values that equal products of powers of small primes (Lagarias [7] discusses colossally abundant numbers and their relationship to the Riemann hypothesis). See Figure 2 for a plot of $l_i/(\log(l_i)m_i)$, m_i/l_i, and $1/\log(l_i)$ for i = 1, 2, 3, ..., 772 (corresponding to the local maxima for $x \leq 15,000,000,000$). ($M(x)$ values for large x were computed using Deléglise and Rivat's [8] algorithm.) The first two curves cross frequently, so there are i values where m_i is approximately equal to $l_i/\sqrt{\log(l_i)}$. See Figure 3 for a plot of $j(x)$ and $\sum_{i=1}^{x} M(\lfloor x/i \rfloor)^2$ for x = 1, 2, 3, ..., 10,000. See Figure 4 for a plot of $\log(l_i)$, $\log(m_i)$, $\log(M(l_i)^2)$, and $\log(m_i/\sigma_0(l_i))$ for i = 1, 2, 3, ..., 772 (when $M(l_i) = 0$, $\log(M(l_i)^2)$ is set to -1). See Figure 5 for a plot of $|M(l_i)|/\sqrt{l_i}$ for i = 1, 2, 3, ..., 772. The largest known value of $M(x)/\sqrt{x}$ (computed by Kotnik and van de Lune [9] for $x \leq 10^{14}$) is 0.570591 (for $M(7,766,842,813) = 50,286$). The largest $|M(l_i)|/\sqrt{l_i}$ value for $x \leq 15,000,000,000$ is 0.568887 (for $l_i = 7,766,892,000$). The largest known value of $|M(x)|/\sqrt{x}$ (computed by Kuznetsov [10]) is 0.585767684 (for $M(11,609,864,264,058,592,345) = -1,995,900,927$).

Let l_i and m_i be similarly defined for the function $\sigma_0(x)$. (l_i, i = 1, 2, 3, ... are known as "highly composite" numbers. Ramanujan [11] initiated the study of such numbers. Robin [12] computed the first 5000 highly composite numbers.) Let m_i' denote $j(l_i)$. See Figure 6 for a plot of $l_i/(\log(l_i)m_i')$, m_i'/l_i, and $1/\log(l_i)$ for i = 2, 3, 4, ..., 160 (corresponding to the local maxima for $x \leq 2,244,031,211,966,544,000$). ($M(x)$ values for large x were computed using an algorithm similar to that used by Kuznetsov. The computations were done on an Intel i7-6700K CPU with 64 GB of RAM.) Although the first two curves cross frequently, m_i' does not appear to converge to $l_i/\sqrt{\log(l_i)}$. See Figure 7 for a plot of $\log(l_i) + \log(\log(l_i))$, $\log(l_i)$, $\log(m_i')$, and $\log(M(l_i)^2)$ for i = 2, 3, 4, ..., 160 (when $M(l_i) = 0$, $\log(M(l_i)^2)$ is set to -1). The vertical distance between the first and third curves appears to become roughly constant. See Figure 8 for a plot of $(\log(l_i) + \log(\log(l_i))) - \log(m_i')$ for i = 2, 3, 4, ..., 160. See Figure 9 for a plot of $\log(l_i) + \frac{1}{2}\log(\log(l_i))$, $\log(\sum_{n=1}^{l_i} M(\lfloor l_i/n \rfloor)^2)$, and $\log(l_i)$ for $i = 2, 3, 4, ..., 160$. $\log(l_i) + \frac{1}{2}\log(\log(l_i))$ is greater than $\log(\sum_{n=1}^{l_i} M(\lfloor l_i/n \rfloor)^2)$ and $\log(\sum_{n=1}^{l_i} M(\lfloor l_i/n \rfloor)^2)$ is greater than $\log(l_i)$ for $i > 4$. This is evidence in support of Conjecture 1. See Figure 10 for a plot of $\log(l_i) + \frac{1}{2}\log(\log(l_i)) - \log(\sum_{n=1}^{l_i} M(\lfloor l_i/n \rfloor)^2)$ for $i = 2, 3, 4, ..., 160$. See Table 1 for l_i, m'_i, and m_i values for $i = 2, 3, 4, ..., 160$. Let m''_i denote $\sum_{n=1}^{l_i} M(\lfloor l_i/n \rfloor)^2$. See Table 2 for l_i, m''_i, and m_i values for $i = 2, 3, 4, ..., 160$. C programs for computing m'_i and m''_i are attached.

References

[1] J. E. Littlewood, Quelques conséquences de l'hypothèse que la fonction $\zeta(s)$ n'a pas de zèros dans le demi-plan $\mathrm{Re}(s) > 1/2$. *C. R. Acad. Sci. Paris* **154**, 263-266 (1912)

[2] A. M. Odlyzko and H. J. J. te Riele, Disproof of the Mertens Conjecture, *Journal für die reine und angewandte Mathematik*, **357**:138-160 (1985)

[3] R. S. Lehman, On Liouville's Function, *Math. Comput.* **14**:311-320 (1960)

[4] R. M. Redheffer, Eine explizit lösbare Optimierungsaufgabe, *Internat. Schriftenreine Numer. Math.*, **36** (1977)

[5] F. Mertens, Über eine zahlentheoretische Funktion, *Akademie Wissenschaftlicher Wien Mathematik-Naturlich Kleine Sitzungsber*, **106** (1897) 761-830

[6] T. M. Apostol, *Introduction to Analytic Number Theory*, Springer, 1976

[7] J. C. Lagarias, An elementary problem equivalent to the Riemann hypothesis, *American Mathematical Monthly* **109**, 534-543 (2002)

[8] M. Deléglise and J. Rivat, Computing the Summation of the Möbius Function, *Experimental Mathematics* **5**, 291-295 (1996)

[9] T. Kotnik and J. van de Lune, On the order of the Mertens function, *Experimental Mathematics* **13**, pp. 473-481 (2004)

[10] E. Kuznetsov, Computing the Mertens function on a GPU, arXiv:1108.0135v1 [math.NT], (2011)

[11] S. Ramanujan, Highly composite numbers, *Proc. London Math. Soc.* **14** (1915), 347-407

[12] G. Robin, Méthodes d'optimalisation pour un problème de théories des nombres., *RAIRO Inform. Théor.* **17**, 239-247, 1983

Figure 1

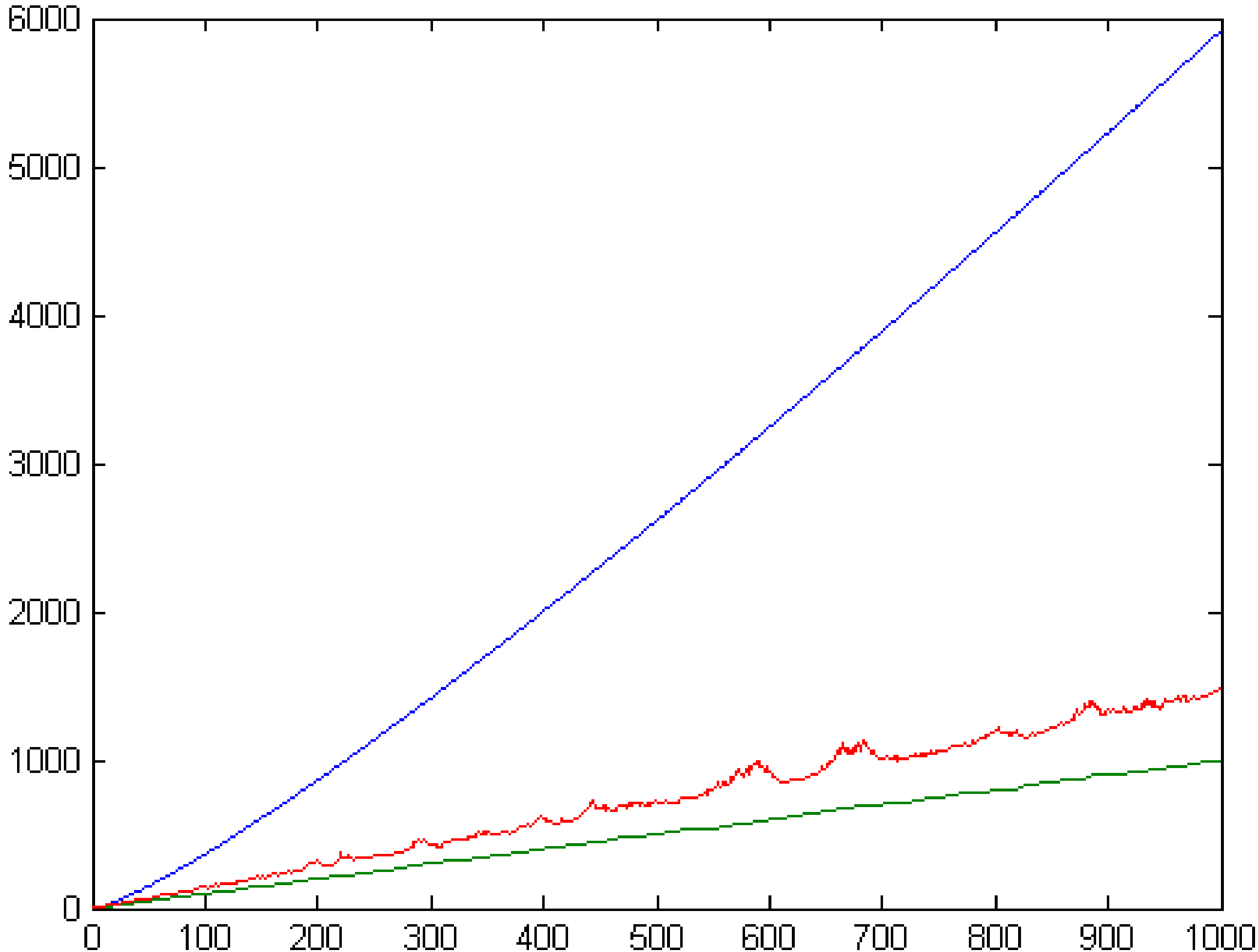

6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
700
800
900
1000

Figure 2

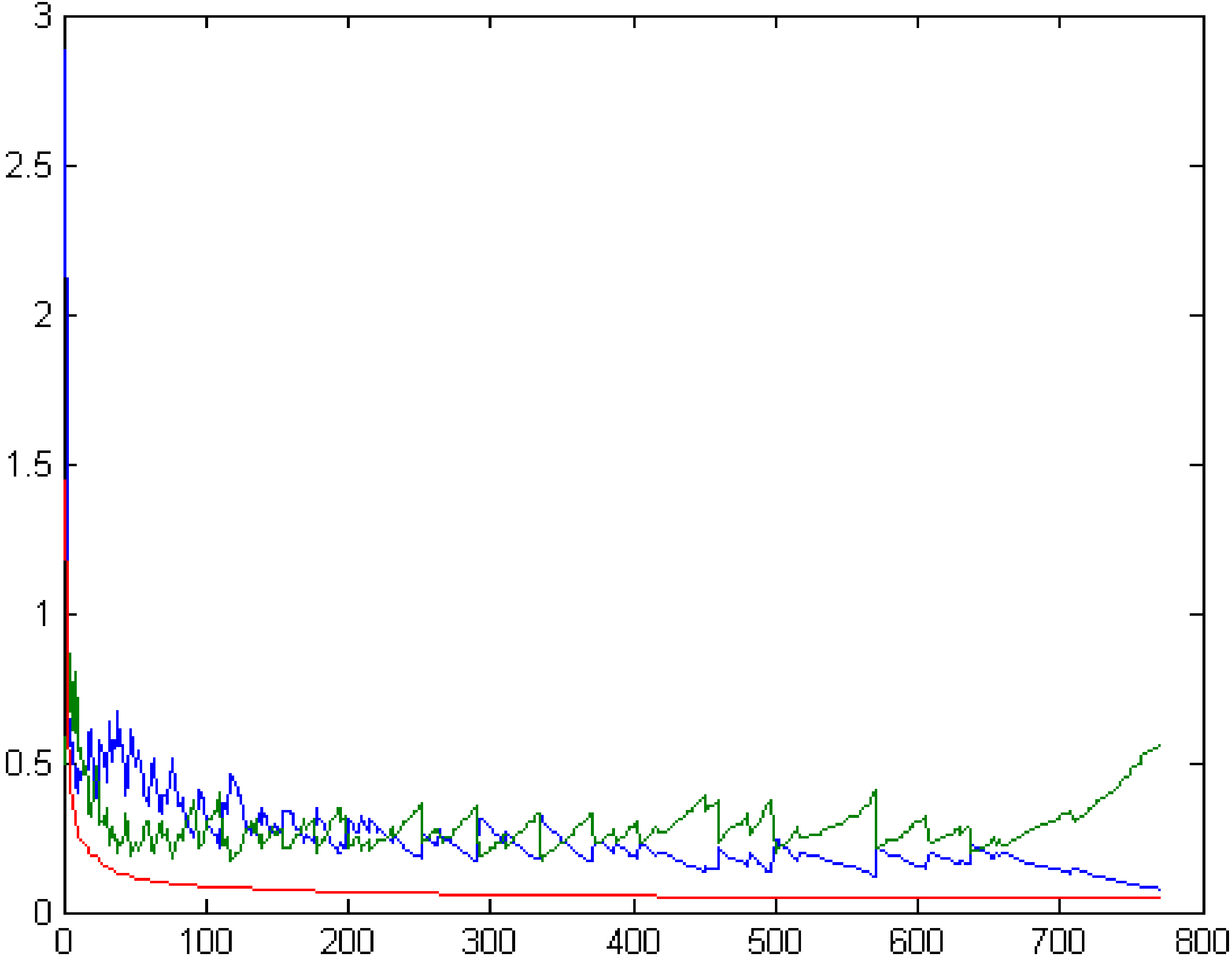


Figure 3

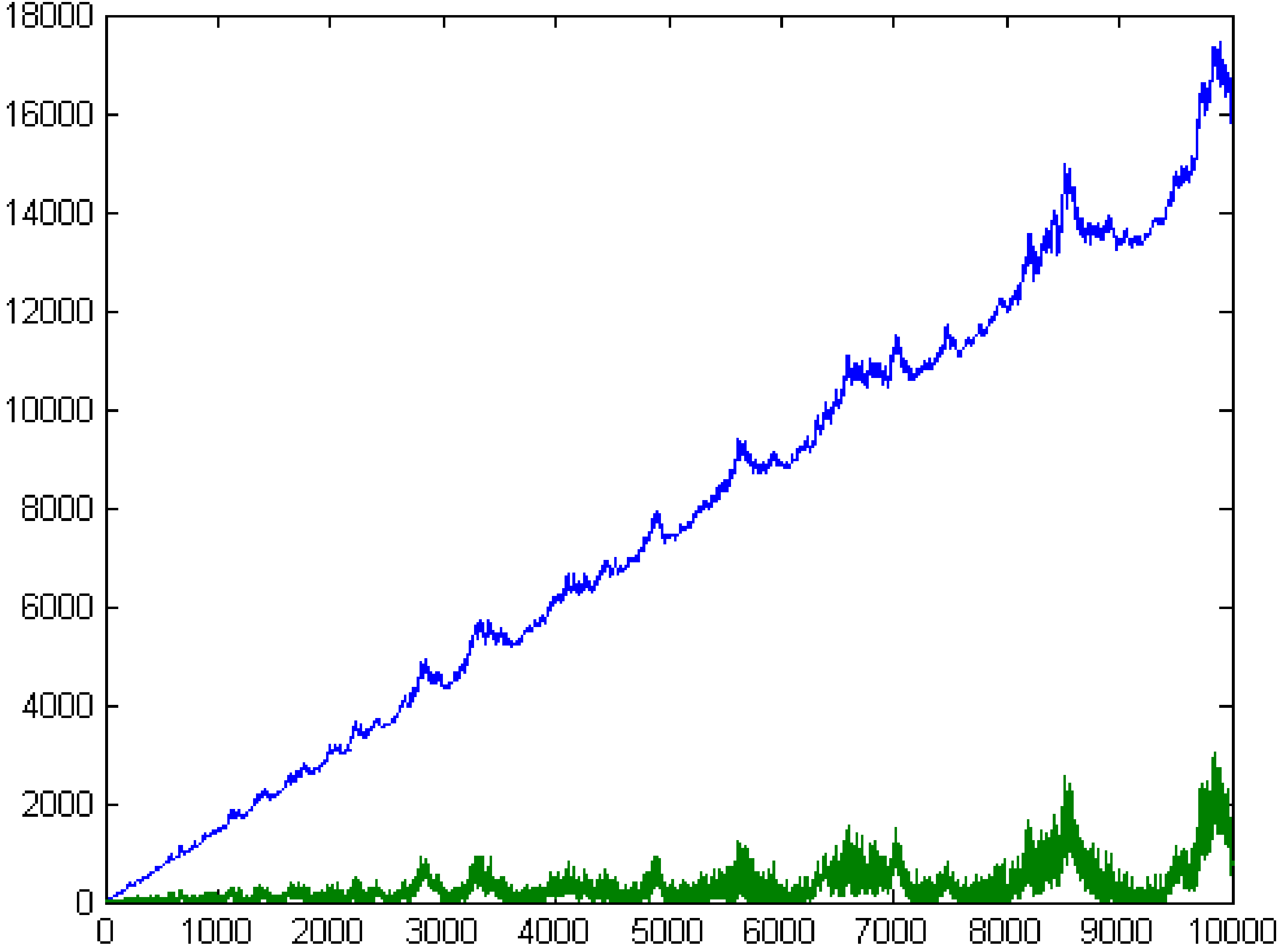

18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Figure 4

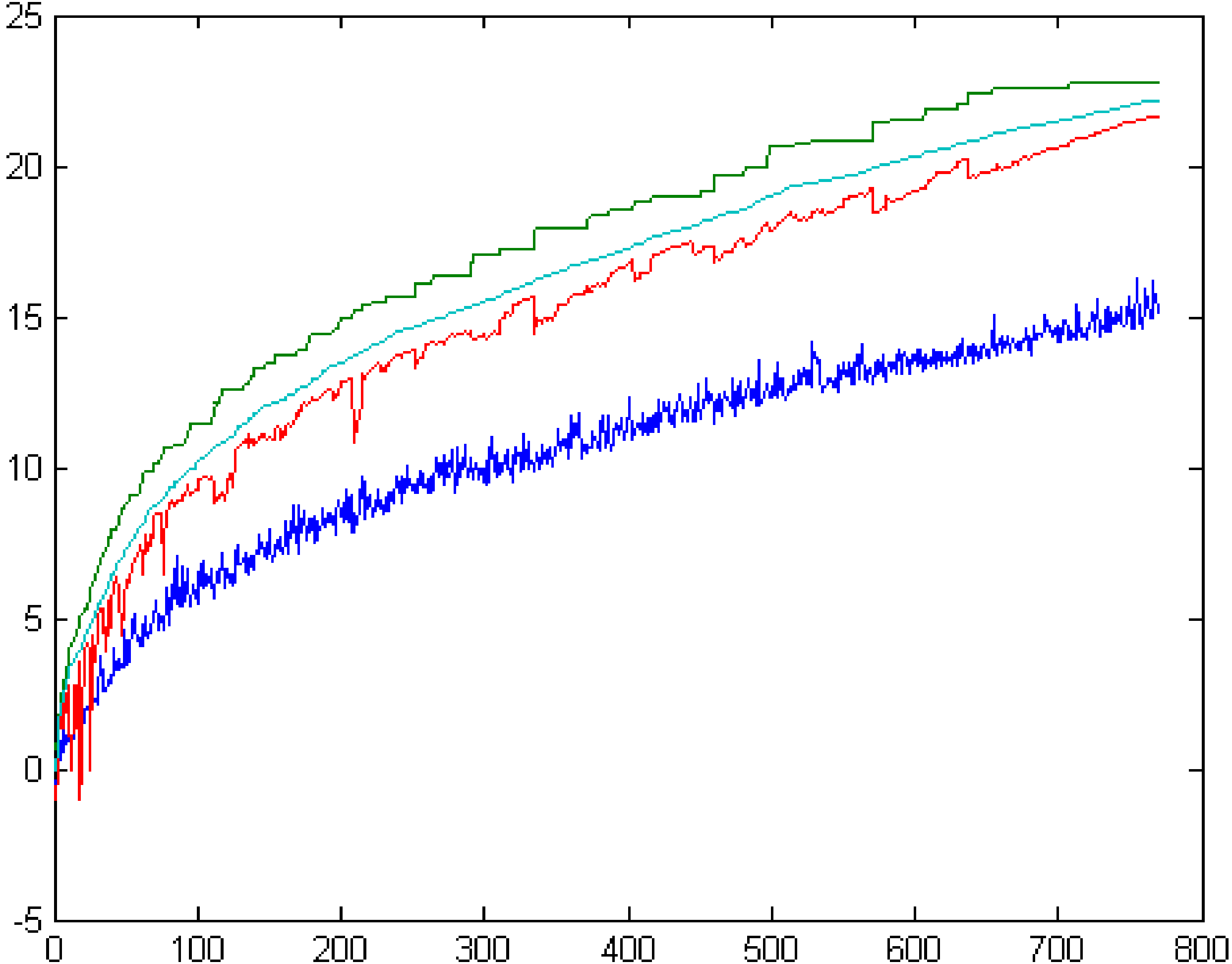


Figure 5

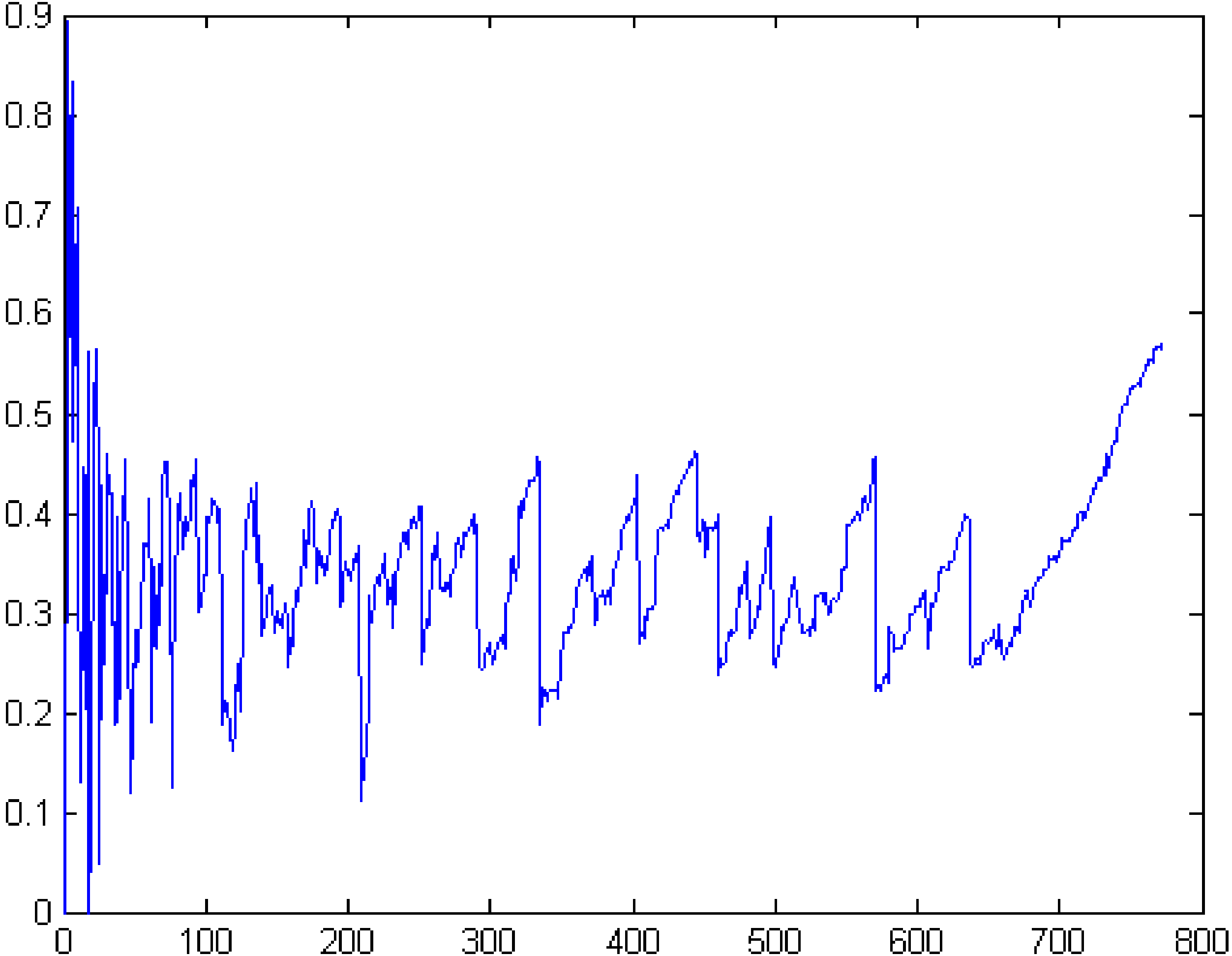


Figure 6

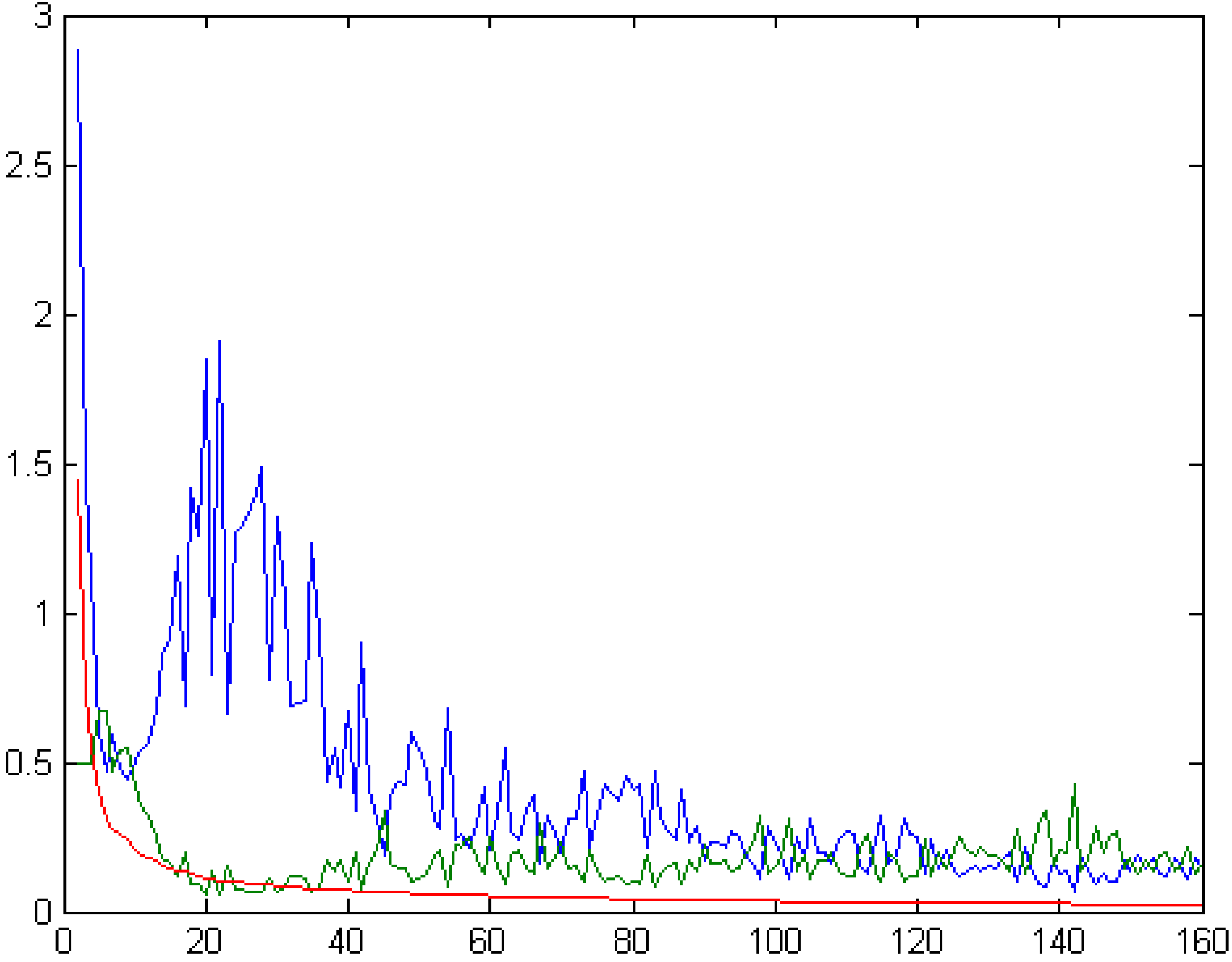
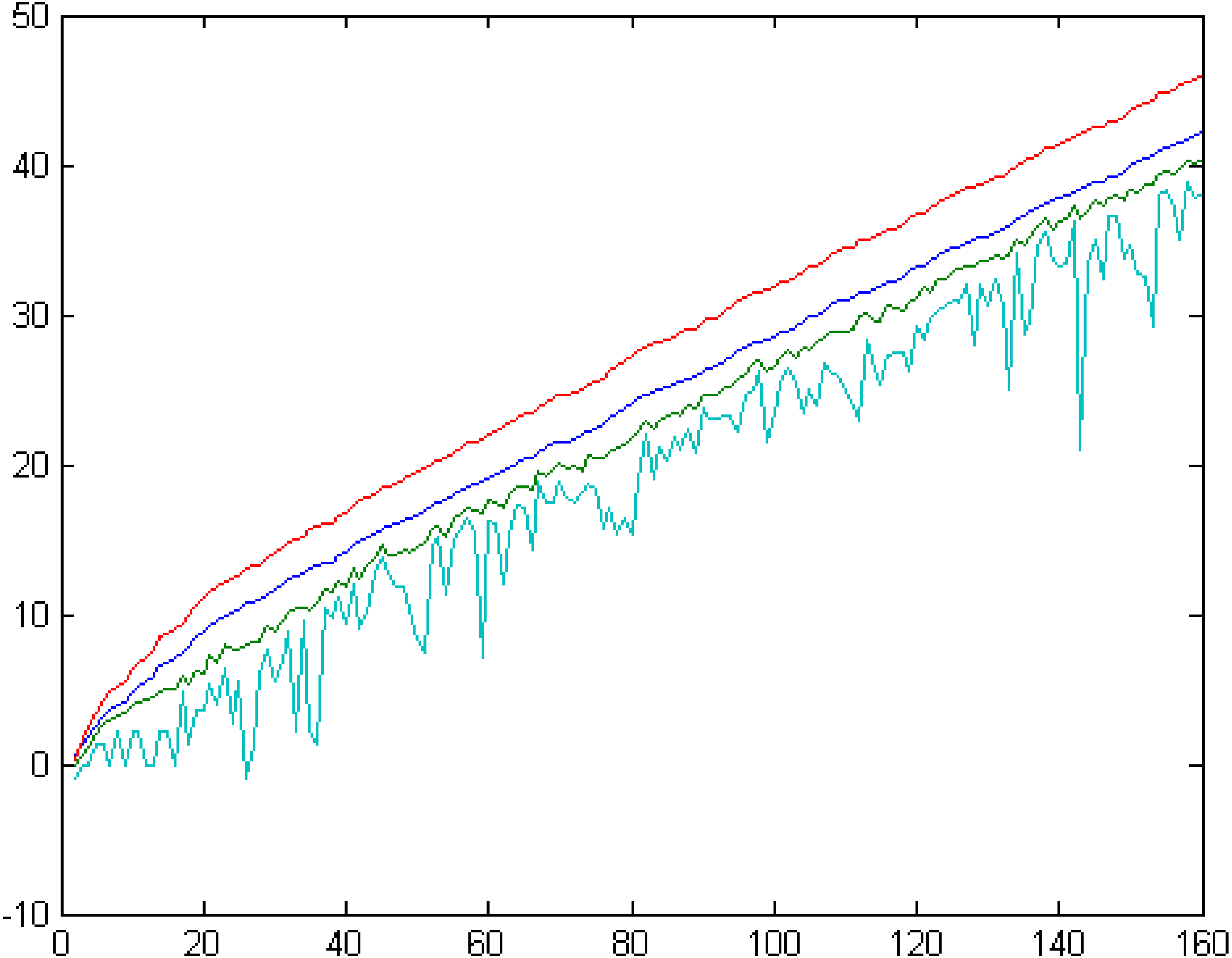


Figure 7



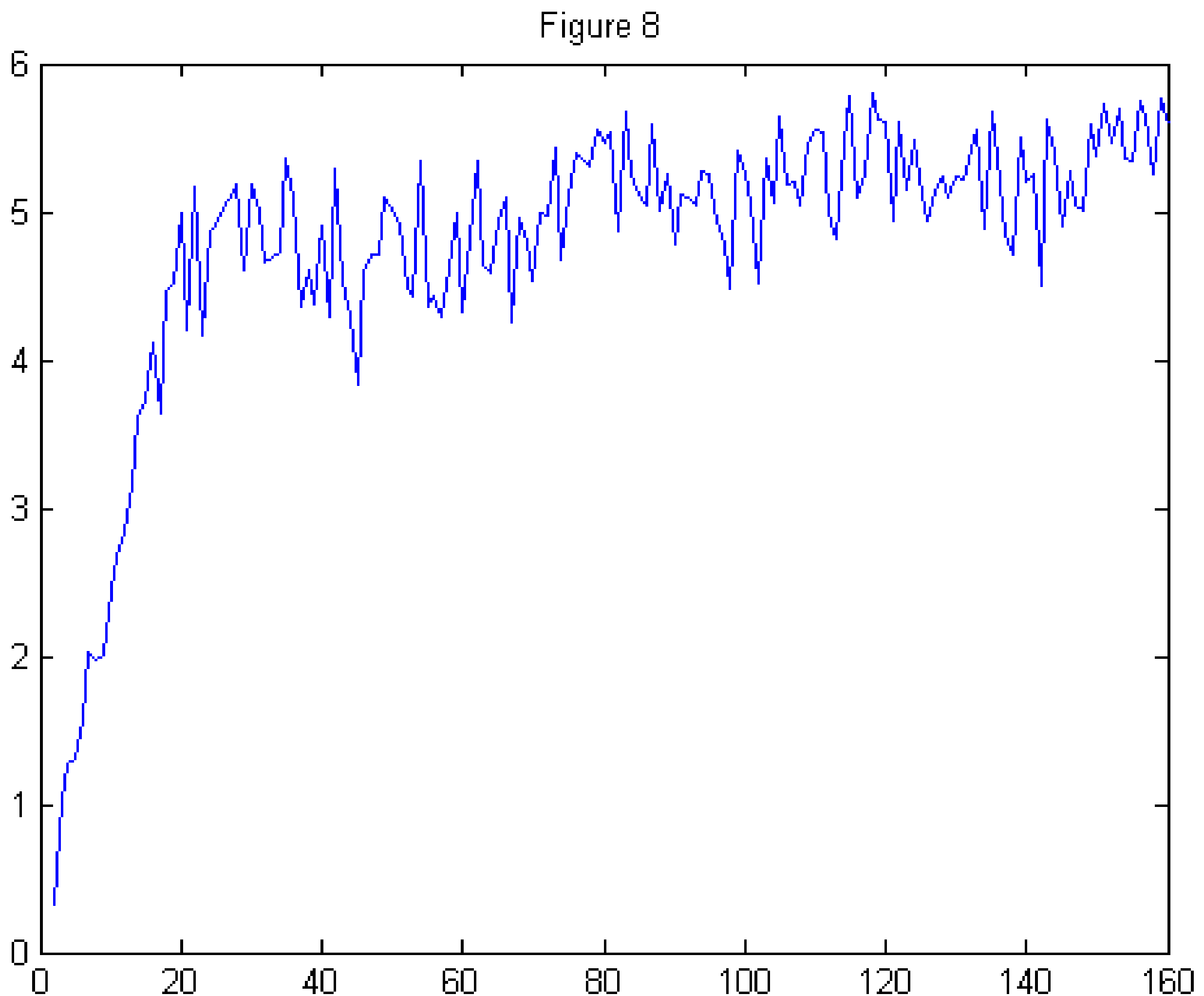


Figure 8
0
1
2
3
4
5
6
0
20
40
60
80
100
120
140
160

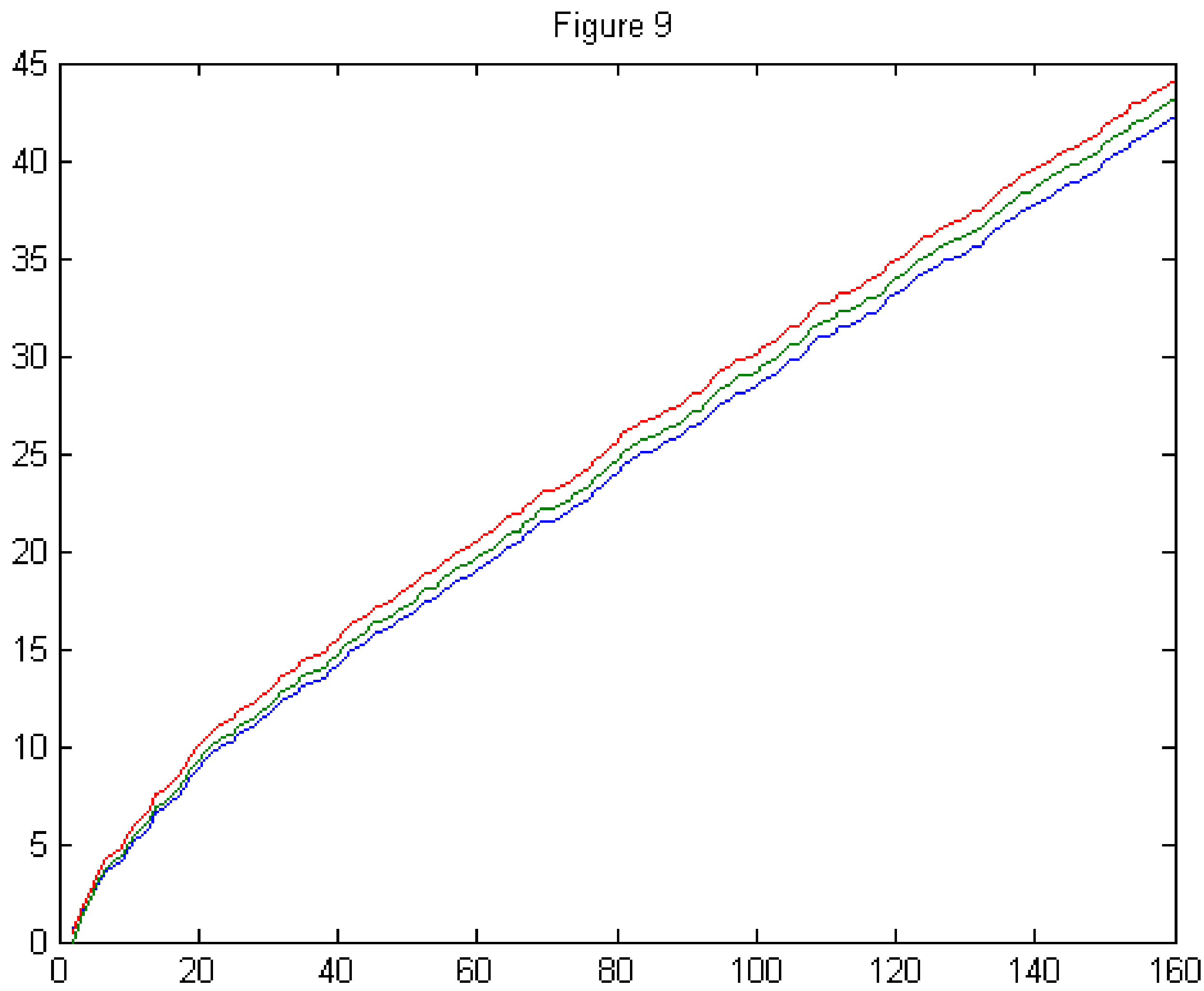


Figure 9
45
40
35
30
25
20
15
10
5
0
0
20
40
60
80
100
120
140
160

Figure 10

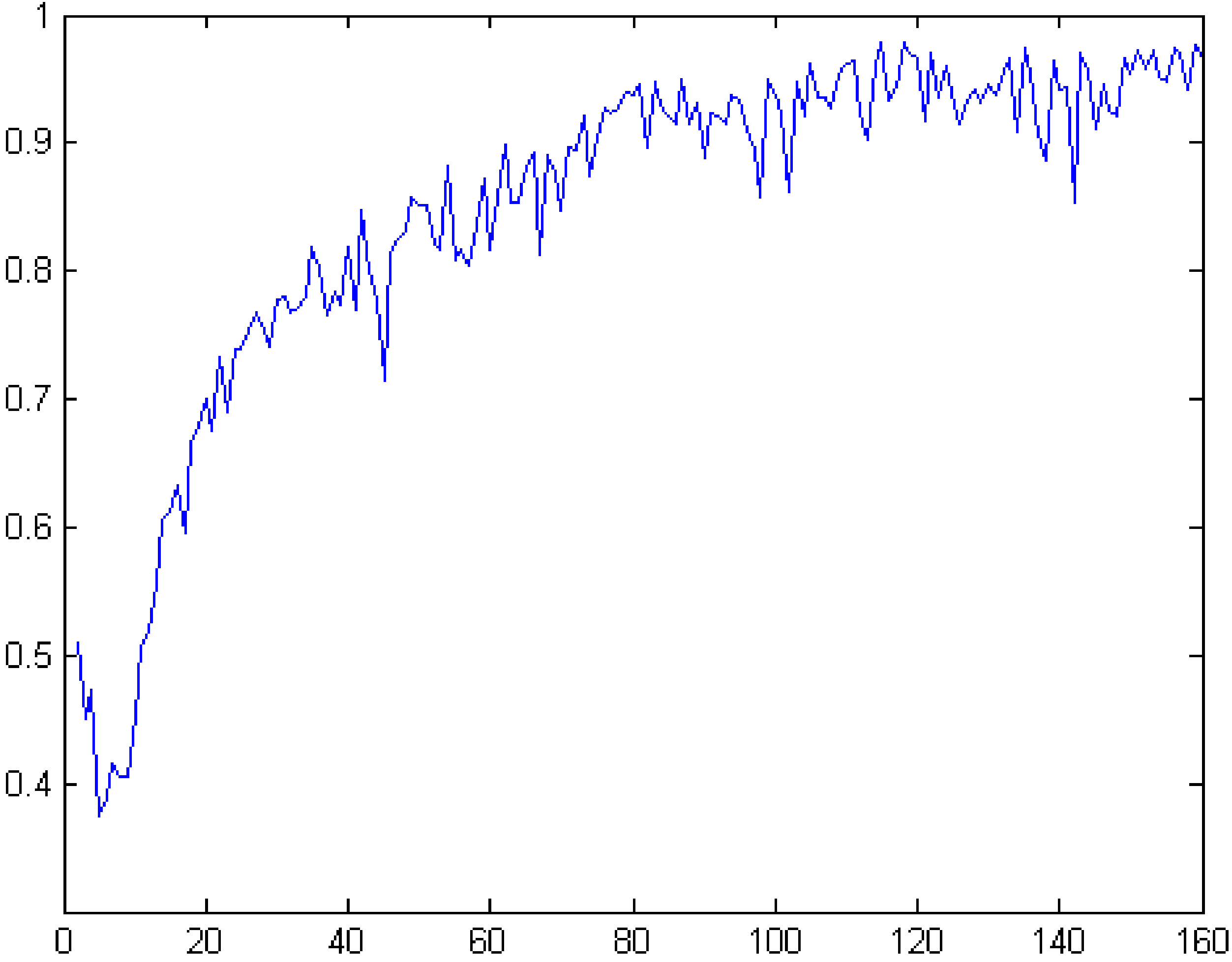


Table 1

l(i)	m'(i)	m(i)
0x0000000000000002	0x0000000000000001	2
0x0000000000000004	0x0000000000000002	3
0x0000000000000006	0x0000000000000003	4
0x000000000000000c	0x0000000000000008	6
0x0000000000000018	0x0000000000000010	8
0x0000000000000024	0x0000000000000011	9
0x0000000000000030	0x000000000000001a	10
0x000000000000003c	0x0000000000000021	12
0x0000000000000078	0x0000000000000032	16
0x00000000000000b4	0x0000000000000040	18
0x00000000000000f0	0x000000000000004d	20
0x0000000000000168	0x000000000000005b	24
0x00000000000002d0	0x0000000000000080	30
0x0000000000000348	0x0000000000000089	32
0x00000000000004ec	0x0000000000000094	36
0x0000000000000690	0x000000000000014a	40
0x00000000000009d8	0x00000000000000e3	48
0x00000000000013b0	0x00000000000001d6	60
0x0000000000001d88	0x00000000000001ca	64
0x0000000000002760	0x0000000000000568	72
0x0000000000003b10	0x0000000000000338	80
0x0000000000004ec0	0x0000000000000c06	84
0x0000000000006270	0x00000000000007a5	90
0x0000000000006c48	0x000000000000083f	96
0x000000000000b130	0x0000000000000c65	100
0x000000000000c4e0	0x0000000000000cfe	108
0x000000000000d890	0x0000000000000d5b	120
0x00000000000144d8	0x00000000000024d0	128
0x000000000001b120	0x0000000000001c36	144
0x00000000000289b0	0x00000000000033d1	160
0x0000000000036240	0x00000000000065ae	168
0x0000000000043ad0	0x0000000000007c6d	180
0x0000000000051360	0x000000000000918c	192
0x0000000000079d10	0x00000000000078c5	200
0x00000000000875a0	0x000000000000aeeb	216
0x00000000000a26c0	0x000000000001be2f	224
0x00000000000aff50	0x0000000000017b3a	240
0x0000000000107ef8	0x000000000002dfe9	256
0x000000000015fea0	0x0000000000024c55	288
0x000000000020fdf0	0x000000000006a7b8	320
0x00000000002bfd40	0x0000000000034aa7	336
0x000000000036fc90	0x000000000008f5a6	360
0x000000000041fbe0	0x00000000000dd881	384
0x000000000062f9d0	0x000000000021c2d1	400
0x00000000006df920	0x000000000011bcf6	432
0x000000000083f7c0	0x000000000012fe31	448
0x0000000000a4f5b0	0x0000000000180fc2	480
0x0000000000dbf240	0x0000000000164b17	504
0x000000000107ef80	0x00000000001cde45	512
0x000000000149eb60	0x000000000029a750	576
0x0000000001eee110	0x00000000005f7c2c	600
0x000000000230dcf0	0x00000000007387a9	640
0x000000000293d6c0	0x000000000037534f	672
0x0000000003a6c590	0x0000000000d4b03c	720
0x000000000461b9e0	0x0000000000f29023	768
0x00000000069296d0	0x0000000001aa702d	800
0x00000000074d8b20	0x000000000156cafb	864
0x0000000008c373c0	0x00000000011f3273	896
0x000000000af450b0	0x0000000002bd8438	960
0x000000000e9b1640	0x0000000002462105	1008
0x000000001186e780	0x0000000001a50279	1024
0x0000000015e8a160	0x00000000042ed839	1152
0x0000000020dcf210	0x0000000006bb92d8	1200
0x0000000029a065d0	0x000000000619dca2	1280
0x000000002bd142c0	0x00000000057e9e90	1344
0x0000000041b9e420	0x00000000137ae108	1440
0x000000005340cba0	0x000000000c5da7c1	1536
0x000000007ce13170	0x00000000163c868b	1600
0x000000008373c840	0x000000001e9296ed	1680
0x000000008ac15360	0x000000001483df6f	1728
0x00000000a6819740	0x0000000018d5f617	1792
0x00000000d021fd10	0x00000000142b74c5	1920
0x000000011582a6c0	0x00000000393b7388	2016
0x000000014d032e80	0x000000002cf0685e	2048
0x00000001a043fa20	0x000000002b072680	2304
0x000000027065f730	0x000000004371d22b	2400
0x000000034087f440	0x00000000609f7492	2688
0x00000004e0cbee60	0x0000000074038584	2880

```
0x00000006810fe880 0x00000000a9504119 3072
0x00000009c197dcc0 0x00000000f0e44255 3360
0x0000000b61dbd6e0 0x000000022aacb0b0 3456
0x0000000ef5a496c0 0x000000014665805d 3584
0x0000001112c9c250 0x0000000237656099 3600
0x00000012b30dbc70 0x00000002cdd75b49 3840
0x00000016c3b7adc0 0x00000003b820f2ab 4032
0x0000001deb492d80 0x00000002e2e3e6de 4096
0x00000022259384a0 0x00000005d2057239 4320
0x00000025661b78e0 0x0000000506c10c78 4608
0x0000003819293550 0x0000000c6065aef5 4800
0x000000444b270940 0x0000000aee3e750a 5040
0x0000004acc36f1c0 0x0000000c2038265d 5376
0x0000007032526aa0 0x00000013784f4e5e 5760
0x00000095986de380 0x00000014bda35207 6144
0x000000e064a4d540 0x000000208a18aac1 6720
0x00000105cac04e20 0x00000033c213dbcd 6912
0x00000175fd12b8c0 0x00000058f254fba5 7168
0x00000188b0207530 0x0000007d437976b5 7200
0x000001c0c949aa80 0x00000388311d08f 7680
0x0000020b95809c40 0x0000004e41583ae8 8064
0x000002ebfa257180 0x0000008791667afd 8192
0x000003116040ea60 0x000000f5ce30ab97 8640
0x000004172b013880 0x00000090708ed00b 9216
0x00000622c081d4c0 0x00000127f4fdcb26 10080
0x0000082e56027100 0x000000deb4903dc5 10368
0x00000879223962c0 0x0000016a60e3c5ec 10752
0x00000c458103a980 0x0000020e454c6c7b 11520
0x000010f24472c580 0x0000035d6fec3938 12288
0x0000188b02075300 0x0000033dddc1125a 12960
0x0000196b66ac2840 0x0000030e18662bd2 13440
0x00001da7f7c8d9a0 0x000003a000e557a6 13824
0x00002a5dab1eedc0 0x000008fc82ebc97d 14336
0x00002c7bf3ad4670 0x00000b56c6d255d4 14400
0x000032d6cd585080 0x000006cdb5346eb6 15360
0x00003b4fef91b340 0x000005d01ee0e6f3 16128
0x000054bb563ddb80 0x000010a2514ad81b 16384
0x000058f7e75a8ce0 0x00000eddcbbd5c65 17280
0x0000769fdf236680 0x00000ba2107614f9 18432
0x0000b1efceb519c0 0x000015101346be5b 20160
0x0000ed3fbe46cd00 0x00001ce8bd200d69 20736
0x000106ab24f2f540 0x00003e8918539fd0 21504
0x000163df9d6a3380 0x00002baf372b6133 23040
0x00020d5649e5ea80 0x0000685c72b8a272 24576
0x0002c7bf3ad46700 0x00006597007c55ac 25920
0x000314016ed8dfc0 0x0000a42683fd3069 26880
0x0003975701525a60 0x0000e3b1f3fb3a94 27648
0x00052157b8beca40 0x00010f72f4bba1be 28672
0x0005630281fb8790 0x0000ff6ea62afa4d 28800
0x00062802ddb1bf80 0x00014fb5b26bb3f3 30720
0x00072eae02a4b4c0 0x000158f575c545d2 32256
0x000a42af717d9480 0x0001fa8a36ce6a33 32768
0x000ac60503f70f20 0x0001b772db5b82b1 34560
0x000e5d5c05496980 0x000201ab2aeff44a 36864
0x00158c0a07ee1e40 0x0005f3b789a5fb63 40320
0x001cbab80a92d300 0x0003a29cc3a3b1ed 41472
0x0023e9660d3787c0 0x00063859fee83910 43008
0x002b18140fdc3c80 0x000c752363ce7a55 46080
0x0040a41e17ca5ac0 0x0015a4fdf63261cf 48384
0x0047d2cc1a6f0f80 0x000aede1ff83866b 49152
0x005630281fb87900 0x0011d9528c61e6d6 51840
0x006bbc3227a69740 0x00157b4ebd76f135 53760
0x0081483c2f94b580 0x003709e4b6cf7994 55296
0x00ac60503f70f200 0x0017f64438f2f1c5 57600
0x00d778644f4d2e80 0x002590e26e392af6 61440
0x010290785f296b00 0x004a36788a3d41b4 62208
0x0109bf2661ce1fc0 0x003504039480ca48 64512
0x017ba35b67267680 0x005f66c19be960e7 65536
0x018e9eb992b52fa0 0x0068ecd576b505e0 69120
0x02137e4cc39c3f80 0x004e2f61f3a59290 73728
0x031d3d73256a5f40 0x00926c2c06cfb33e 80640
0x0426fc9987387f00 0x008a0bd4b1981217 82944
0x0530bbbfe9069ec0 0x00e201d7e7e2f4e8 86016
0x063a7ae64ad4be80 0x00db8a5e56ec7f6c 92160
0x0957b859703f1dc0 0x01cd0729e9c5bfef 96768
0x0a61777fd20d3d80 0x020dfb80947ca498 98304
0x0c74f5cc95a97d00 0x01a6c47eb7c2e7dc 103680
0x0f92333fbb13dc40 0x02507e2a269aaf92 107520
0x12af70b2e07e3b80 0x0417012bedfe67e2 110592
0x18e9eb992b52fa00 0x034479ab08d73577 115200
0x1f24667f7627b880 0x04f42961c146637b 122880
```

Table 2

l(i)	m''(i)	m(i)
0x0000000000000002	0x0000000000000001	2
0x0000000000000004	0x0000000000000003	3
0x0000000000000006	0x0000000000000005	4
0x000000000000000c	0x000000000000000d	6
0x0000000000000018	0x000000000000001d	8
0x0000000000000024	0x000000000000002d	9
0x0000000000000030	0x000000000000003f	10
0x000000000000003c	0x0000000000000051	12
0x0000000000000078	0x00000000000000a7	16
0x00000000000000b4	0x00000000000000f7	18
0x00000000000000f0	0x000000000000014f	20
0x0000000000000168	0x00000000000001f5	24
0x00000000000002d0	0x00000000000003ef	30
0x0000000000000348	0x000000000000049d	32
0x00000000000004ec	0x00000000000006fd	36
0x0000000000000690	0x00000000000009d9	40
0x00000000000009d8	0x0000000000000e29	48
0x00000000000013b0	0x0000000000001d25	60
0x0000000000001d88	0x0000000000002bd9	64
0x0000000000002760	0x0000000000003cd5	72
0x0000000000003b10	0x00000000000057f9	80
0x0000000000004ec0	0x0000000000007c77	84
0x0000000000006270	0x00000000000095a7	90
0x0000000000006c48	0x000000000000a561	96
0x000000000000b130	0x0000000000011135	100
0x000000000000c4e0	0x0000000000012c83	108
0x000000000000d890	0x0000000000014ed7	120
0x00000000000144d8	0x0000000000020943	128
0x000000000001b120	0x000000000002a637	144
0x00000000000289b0	0x00000000000407c9	160
0x0000000000036240	0x00000000000583bf	168
0x0000000000043ad0	0x000000000006eb65	180
0x0000000000051360	0x0000000000084aab	192
0x0000000000079d10	0x00000000000c2721	200
0x00000000000875a0	0x00000000000dcaed	216
0x00000000000a26c0	0x0000000000114d8f	224
0x00000000000aff50	0x0000000000126fdd	240
0x0000000000107ef8	0x00000000001c573f	256
0x000000000015fea0	0x0000000000247d95	288
0x000000000020fdf0	0x00000000003a4e53	320
0x00000000002bfd40	0x000000000048bca7	336
0x000000000036fc90	0x00000000005fe6eb	360
0x000000000041fbe0	0x00000000007662a3	384
0x000000000062f9d0	0x0000000000bfc99b	400
0x00000000006df920	0x0000000000c156ef	432
0x000000000083f7c0	0x0000000000e6ea1b	448
0x0000000000a4f5b0	0x0000000001216543	480
0x0000000000dbf240	0x00000000017a6f43	504
0x000000000107ef80	0x0000000001cbb349	512
0x000000000149eb60	0x00000000024304e7	576
0x0000000001eee110	0x000000000388d0bf	600
0x000000000230dcf0	0x00000000040a575b	640
0x000000000293d6c0	0x0000000004784b9d	672
0x0000000003a6c590	0x0000000006e49b81	720
0x000000000461b9e0	0x00000000083cba75	768
0x00000000069296d0	0x000000000ca5ad49	800
0x00000000074d8b20	0x000000000dae35a1	864
0x0000000008c373c0	0x000000000fe0124d	896
0x000000000af450b0	0x000000001520b089	960
0x000000000e9b1640	0x000000001b34684b	1008
0x000000001186e780	0x000000001f850369	1024
0x0000000015e8a160	0x0000000029723df3	1152
0x0000000020dcf210	0x000000003ed6fd41	1200
0x0000000029a065d0	0x000000004da824b5	1280
0x000000002bd142c0	0x0000000051209eab	1344
0x0000000041b9e420	0x00000000852fb0a5	1440
0x000000005340cba0	0x000000009cce4a9d	1536
0x000000007ce13170	0x00000000f0dceab7	1600
0x000000008373c840	0x0000000105590159	1680
0x000000008ac15360	0x0000000106b1420f	1728
0x00000000a6819740	0x000000013da54841	1792
0x00000000d021fd10	0x000000018404c49b	1920
0x0000000011582a6c0	0x0000000222d05965	2016
0x000000014d032e80	0x000000027e69bad5	2048
0x00000001a043fa20	0x000000030fece347	2304
0x000000027065f730	0x00000004a8295a05	2400
0x000000034087f440	0x00000006364fec8b	2688
0x00000004e0cbee60	0x00000009490c091d	2880

```
0x00000006810fe880 0x0000000c7e1a54ff 3072
0x00000009c197dcc0 0x00000012bf3a04f7 3360
0x0000000b61dbd6e0 0x00000170693cdcf 3456
0x0000000ef5a496c0 0x0000001ce5dba56d 3584
0x0000001112c9c250 0x00000021b3d41c97 3600
0x00000012b30dbc70 0x0000002546af574d 3840
0x00000016c3b7adc0 0x0000002dd657ad5d 4032
0x0000001deb492d80 0x0000003a90e7d29f 4096
0x00000022259384a0 0x000000454804e35f 4320
0x00000025661b78e0 0x0000004ae7c2e97b 4608
0x0000003819293550 0x0000007617674925 4800
0x000000444b270940 0x0000008b49247ff7 5040
0x0000004acc36f1c0 0x00000993229dc39 5376
0x0000007032526aa0 0x000000e920cf709f 5760
0x00000095986de380 0x0000013132d6aab1 6144
0x000000e064a4d540 0x000001cf5bf30ccf 6720
0x00000105cac04e20 0x0000022a755b2bd7 6912
0x00000175fd12b8c0 0x00000329e6cbd0b7 7168
0x00000188b0207530 0x0000037303ac4e61 7200
0x000001c0c949aa80 0x0000039b452906cf 7680
0x0000020b95809c40 0x0000044529f3159d 8064
0x000002ebfa257180 0x0000063998ea0599 8192
0x000003116040ea60 0x000006f5bbbf2121 8640
0x000004172b013880 0x00000890a3d27ccd 9216
0x00000622c081d4c0 0x00000d471944b75d 10080
0x0000082e56027100 0x0000110e4e387f41 10368
0x00000879223962c0 0x000012293ffe2c01 10752
0x00000c458103a980 0x00001a750444890d 11520
0x000010f24472c580 0x00002515d67509eb 12288
0x0000188b02075300 0x00003485fdb8da51 12960
0x0000196b66ac2840 0x00003605a6a98f8b 13440
0x00001da7f7c8d9a0 0x00003f13e1934985 13824
0x00002a5dab1eedc0 0x00005e60666f991b 14336
0x00002c7bf3ad4670 0x0000654c8f522b85 14400
0x000032d6cd585080 0x00006d814a499bab 15360
0x00003b4fef91b340 0x00007dd835fc1999 16128
0x000054bb563ddb80 0x0000bcc289a18ec5 16384
0x000058f7e75a8ce0 0x0000c454532fb169 17280
0x0000769fdf236680 0x0000fe44f8ef2539 18432
0x0000b1efceb519c0 0x000182ee60f8d5e3 20160
0x0000ed3fbe46cd00 0x000208030f22177f 20736
0x000106ab24f2f540 0x00025d6239a1326b 21504
0x000163df9d6a3380 0x00030e68903b10af 23040
0x00020d5649e5ea80 0x0004b11ac609b443 24576
0x0002c7bf3ad46700 0x00063bbcc68ef307 25920
0x000314016ed8dfc0 0x00071f59d4267fa1 26880
0x0003975701525a60 0x00087351f8d12a91 27648
0x00052157b8beca40 0x000bebdfd86ee4d5 28672
0x0005630281fb8790 0x000c68ef111ecec1 28800
0x00062802ddb1bf80 0x000e6059d009d305 30720
0x00072eae02a4b4c0 0x0010927d3704bf03 32256
0x000a42af717d9480 0x0017f41327cc8cf5 32768
0x000ac60503f70f20 0x0018bb9767251be3 34560
0x000e5d5c05496980 0x0020bce5ba6e0bbf 36864
0x00158c0a07ee1e40 0x00344f892b23cc49 40320
0x001cbab80a92d300 0x0041abf3f514fe1f 41472
0x0023e9660d3787c0 0x0053ffbd20923791 43008
0x002b18140fdc3c80 0x006a0064ec9ad7f5 46080
0x0040a41e17ca5ac0 0x00a2d8aa7c2719a3 48384
0x0047d2cc1a6f0f80 0x00a7d22e97b617d9 49152
0x005630281fb87900 0x00ce75ff73ee3c85 51840
0x006bbc3227a69740 0x01021d05af06d225 53760
0x0081483c2f94b580 0x0153c67185560fd3 55296
0x00ac60503f70f200 0x0194f6dfbcf00a79 57600
0x00d778644f4d2e80 0x0203030e7b111d1f 61440
0x010290785f296b00 0x028857ca33031801 62208
0x0109bf2661ce1fc0 0x0283ff24aee4f9d1 64512
0x017ba35b67267680 0x03aeca830693c91b 65536
0x018e9eb992b52fa0 0x03e21cb1448f7433 69120
0x02137e4cc39c3f80 0x04f9034d7de9850f 73728
0x031d3d73256a5f40 0x07943523d267ee93 80640
0x0426fc9987387f00 0x09f3fd784e5136f7 82944
0x0530bbbfe9069ec0 0x0ca7eec1e241d0d1 86016
0x063a7ae64ad4be80 0x0f03aa50986f0eab 92160
0x0957b859703f1dc0 0x171aec63d0439503 96768
0x0a61777fd20d3d80 0x19ccfe4dd401fa49 98304
0x0c74f5cc95a97d00 0x1e35acfa6ba81591 103680
0x0f92333fbb13dc40 0x260cfb33c2708c53 107520
0x12af70b2e07e3b80 0x2f1ea303b305efff 110592
0x18e9eb992b52fa00 0x3ce1160a1103481b 115200
0x1f24667f7627b880 0x4d1aec1286af8991 122880
```

```
/*CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  COMPUTE MERTENS FUNCTION (j(x) where x is a highly composite number)        C
C  02/06/16 (DKC)                                                              C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC*/
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "table.h"  // look-up table of the 1230 primes less than or equal to 10007
extern char *malloc();
// compute partial sums of Mertens function
long long newmert(unsigned long long s, unsigned long long x, int *M) {
unsigned long long t;
long long i;
long long sum;
t=(unsigned long long)sqrt((double)x);
t=t+2;
if (s>(x/t))
   return(0x7fffffffffffffff);
sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=(long long)s; i<=(long long)(x/t); i++)
   sum=sum+M[x/i-1];
#pragma omp parallel for reduction (+:sum)
for (i=1; i<(long long)t; i++) {
   sum=sum+(long long)M[i-1]*(x/(unsigned long long)i-x/(unsigned long long)(i+1));
   }
return(sum);
}
// compute Mobius function
int newmobl(unsigned long long a, unsigned long long b, long long *out, unsigned int *table,
            unsigned int tsize) {
unsigned int i,p,count;
unsigned long long beg,ps,rem;
long long t;
count=(unsigned int)(b-a);
for (i=0; i<count; i++)
   out[i]=1;
for (i=0; i<tsize; i++) {
   p=table[i];
   ps=(unsigned long long)p*(unsigned long long)p;
   if (ps>b)
      goto askip;
   rem=a-(a/ps)*ps;
   if (rem!=0)
      beg=ps-rem;
   else
      beg=0;
   while (beg<count) {
      out[beg]=0;
      beg=beg+ps;
      }
   rem=a-(a/p)*p;
   if (rem!=0)
      beg=p-rem;
   else
      beg=0;
   while (beg<count) {
      out[beg]=-out[beg]*p;
      beg=beg+p;
      }
   }
return(p);
askip:
for (i=0; i<count; i++) {
   if (out[i]==0)
      continue;
   t=out[i];
   if (t<0)
      t=-t;
   if ((unsigned long long)t<(i+a))
      out[i]=-out[i];
   }
for (i=0; i<count; i++) {
   if (out[i]==0)
      continue;
   if (out[i]>0)
      out[i]=1;
   else
```

```
      out[i]=-1;
   }
return(1);
}
// compute primes
unsigned int primed(unsigned int *out, unsigned int tsize,
                    unsigned int *table,unsigned int limit) {
unsigned int d;
unsigned int i,j,k,l,flag,count;
count=tsize;
for (i=0; i<tsize; i++)
   out[i]=table[i];
j=table[tsize-1]+1;
for (d=j; d<=limit; d++) {
   if(d==(d/2)*2) continue;
   if(d==(d/3)*3) continue;
   if(d==(d/5)*5) continue;
   if(d==(d/7)*7) continue;
   if(d==(d/11)*11) continue;
   l=(unsigned int)(10.0+sqrt((double)d));
   k=0;
   if (l>table[tsize-1])
      return(0);
   else {
      for (i=0; i<tsize; i++) {
         if (table[i]<=l)
            k=i;
         else
            break;
         }
      }
   flag=1;
   l=k;
   for (i=0; i<=l; i++) {
      k=table[i];
      if ((d/k)*k==d) {
         flag=0;
         break;
         }
      }
   if (flag==1)
      out[count]=d;
   count=count+flag;
   }
return(count);
}

unsigned long long Msize=15000000000;   // for 64 GB of RAM, 64-bit OS
unsigned int tsize=1230;           // prime look-up table size
unsigned int tmpsiz=10000;         // used to compute Mobius function
unsigned int t2size=200000;        // prime look-up table size
unsigned int Tsize=150000000;      // used to compute partial sums of Mobius function
unsigned int oldnews=170000;       // used to factor N
unsigned int prsize=1000;          // used to factor N
//
unsigned int incnt=159;
unsigned long long in[167*2]={  // highly composite numbers and their number of divisors
                   2,            2,
                   4,            3,
                   6,            4,
                  12,            6,
                  24,            8,
                  36,            9,
                  48,           10,
                  60,           12,
                 120,           16,
                 180,           18,
                 240,           20,
                 360,           24,
                 720,           30,
                 840,           32,
                1260,           36,
                1680,           40,
                2520,           48,
                5040,           60,
                7560,           64,
               10080,           72,
               15120,           80,
               20160,           84,
               25200,           90,
               27720,           96,
```

```
          45360,          100,
          50400,          108,
          55440,          120,
          83160,          128,
         110880,          144,
         166320,          160,
         221760,          168,
         277200,          180,
         332640,          192,
         498960,          200,
         554400,          216,
         665280,          224,
         720720,          240,
        1081080,          256,
        1441440,          288,
        2162160,          320,
        2882880,          336,
        3603600,          360,
        4324320,          384,
        6486480,          400,
        7207200,          432,
        8648640,          448,
       10810800,          480,
       14414400,          504,
       17297280,          512,
       21621600,          576,
       32432400,          600,
       36756720,          640,
       43243200,          672,
       61261200,          720,
       73513440,          768,
      110270160,          800,
      122522400,          864,
      147026880,          896,
      183783600,          960,
      245044800,         1008,
      294053760,         1024,
      367567200,         1152,
      551350800,         1200,
      698377680,         1280,
      735134400,         1344,
     1102701600,         1440,
     1396755360,         1536,
     2095133040,         1600,
     2205403200,         1680,
     2327925600,         1728,
     2793510720,         1792,
     3491888400,         1920,
     4655851200,         2016,
     5587021440,         2048,
     6983776800,         2304,
    10475665200,         2400,
    13967553600,         2688,
    20951330400,         2880,
    27935107200,         3072,
    41902660800,         3360,
    48886437600,         3456,
    64250746560,         3584,
    73329656400,         3600,
    80313433200,         3840,
    97772875200,         4032,
   128501493120,         4096,
   146659312800,         4320,
   160626866400,         4608,
   240940299600,         4800,
   293318625600,         5040,
   321253732800,         5376,
   481880599200,         5760,
   642507465600,         6144,
   963761198400,         6720,
  1124388064800,         6912,
  1606268664000,         7168,
  1686582097200,         7200,
  1927522396800,         7680,
  2248776129600,         8064,
  3212537328000,         8192,
  3373164194400,         8640,
  4497552259200,         9216,
  6746328388800,        10080,
  8995104518400,        10368,
  9316358251200,        10752,
```

```
      13492656777600,        11520,
      18632716502400,        12288,
      26985313555200,        12960,
      27949074753600,        13440,
      32607253879200,        13824,
      46581791256000,        14336,
      48910880818800,        14400,
      55898149507200,        15360,
      65214507758400,        16128,
      93163582512000,        16384,
      97821761637600,        17280,
     130429015516800,        18432,
     195643523275200,        20160,
     260858031033600,        20736,
     288807105787200,        21504,
     391287046550400,        23040,
     577614211574400,        24576,
     782574093100800,        25920,
     866421317361600,        26880,
    1010824870255200,        27648,
    1444035528936000,        28672,
    1516237305382800,        28800,
    1732842634723200,        30720,
    2021649740510400,        32256,
    2888071057872000,        32768,
    3032474610765600,        34560,
    4043299481020800,        36864,
    6064949221531200,        40320,
    8086598962041600,        41472,
   10108248702552000,        43008,
   12129898443062400,        46080,
   18194847664593600,        48384,
   20216497405104000,        49152,
   24259796886124800,        51840,
   30324746107656000,        53760,
   36389695329187200,        55296,
   48519593772249600,        57600,
   60649492215312000,        61440,
   72779390658374400,        62208,
   74801040398884800,        64512,
  106858629141264000,        65536,
  112201560598327200,        69120,
  149602080797769600,        73728,
  224403121196654400,        80640,
  299204161595539200,        82944,
  374005201994424000,        86016,
  448806242393308800,        92160,
  673209363589963200,        96768,
  748010403988848000,        98304,
  897612484786617600,       103680,
 1122015605983272000,       107520,
 1346418727179926400,       110592,
 1795224969573235200,       115200,
 2244031211966544000,       122880,
 2692837454359852800,       124416,
 3066842656354276800,       129024,
 4381203794791824000,       131072,
 4488062423933088000,       138240,
 6133685312708553600,       147456,
 8976124847866176000,       153600,
 9200527969062830400,       161280,
 12267370625417107200,      165888};
//
void main() {
unsigned long long *oldtmp,*newtmp;
long long *temp,sum,sump,*T,ltemp;
int *M;
unsigned int *ntable;
unsigned int *pritab;
unsigned int p,tindex,prind,delta,joff,newind,count,total,h,k,ntsize;
unsigned int g,f,L;
unsigned long long index,ta,tb,j,N,ut,pz,tz,mcount,start;
long long i;
int savet,t,ID,d,e;
FILE *Outfp;
Outfp = fopen("out1arq.dat","w");
omp_set_dynamic(0);
omp_set_num_threads(8);
#pragma omp parallel
{
   ID=omp_get_thread_num();
```

```
   printf(" ID=%d \n",ID);
}
ntable=(unsigned int*) malloc((t2size+1)*4);
if (ntable==NULL) {
   printf("not enough memory \n");
   goto zskip;
   }
pritab=(unsigned int*) malloc((prsize+1)*4);
if (pritab==NULL) {
   printf("not enough memory \n");
   return;
   }
temp=(long long*) malloc((tmpsiz+1)*8);
if (temp==NULL) {
   printf("not enough memory \n");
   return;
   }
oldtmp=(long long*) malloc((oldnews+1)*8);
if (oldtmp==NULL) {
   printf("not enough memory \n");
   return;
   }
newtmp=(long long *) malloc((oldnews+1)*8);
if (newtmp==NULL) {
   printf("not enough memory \n");
   return;
   }
M=(int*) malloc((Msize+1)*4);
if (M==NULL) {
   printf("not enough memory \n");
   return;
   }
T=(long long*)malloc((Tsize+1)*8);
if (T==NULL) {
   printf("not enough memory \n");
   return;
   }
ntsize=primed(ntable,tsize,table,2000000);
printf("prime look-up table size=%d, largest prime=%d \n",ntsize,ntable[ntsize-1]);
printf("computing Mobius function \n");
index=0;
ta=1;
tb=(unsigned long long)(tmpsiz+1);
mcount=Msize/(unsigned long long)tmpsiz;
for (i=0; i<(long long)mcount; i++) {
   t=newmobl(ta,tb,temp,ntable,ntsize);
   if (t!=1) {
      printf("error \n");
      goto zskip;
      }
   ta=tb;
   tb=tb+(unsigned long long)tmpsiz;
   for (j=0; j<tmpsiz; j++)
      M[j+index]=(int)temp[j];
   index=index+(unsigned long long)tmpsiz;
   }
//
// compute Mertens function
//
printf("computing Mertens function \n");
for (i=1; i<=(long long)Msize; i++) {
    M[i]=M[i-1]+M[i];
   }
//
printf("computing j(x) \n");
for (g=0; g<incnt; g++) {
//
// factor N
//
   N=in[2*g];
   ut=N;
   prind=0;
   tindex=0;
   total=1;
   h=(unsigned int)(sqrt((double)ut)+0.01);  // for p>2, the difference between
   for (f=0; f<ntsize; f++) {                // successive primes is greater
      p=table[f];                            // than 1, so adding 0.01 is okay
      if (p>h)
         goto fskip;
      count=0;
      while (ut==(ut/p)*p) {
```

```
         if (count==0)
            oldtmp[tindex]=p;
         else
            oldtmp[tindex]=oldtmp[tindex-1]*p;
         tindex=tindex+1;
         if (tindex>oldnews) {
            printf("divisor table not big enough (1): N=%d \n",N);
            goto zskip;
            }
         ut=ut/p;
         count=count+1;
         }
      if (count!=0) {
         total=total*(count+1);
         pritab[prind]=count;
         prind=prind+1;
         if (prind>prsize) {
            printf("prime table not big enough: N=%d \n",N);
            goto zskip;
            }
         }
      if (ut==1)
         goto askip;
      }
   printf("error: prime look-up table not big enough \n");
   goto zskip;
//
//  compute combinations of factors
//
fskip:
   oldtmp[tindex]=ut;
   tindex=tindex+1;
   if (tindex>oldnews) {
      printf("divisor table not big enough (2): N=%d \n",N);
      goto zskip;
      }
   count=1;
   total=total*(count+1);
   pritab[prind]=count;
   prind=prind+1;
   if (prind>prsize) {
       printf("prime table not big enough: N=%d \n",N);
       goto zskip;
       }
askip:
   if (total!=(unsigned int)in[2*g+1]) {
      printf("error: total=%d %d \n",total,(unsigned int)in[2*g+1]);
      goto zskip;
      }
   if (prind==1) {
      newind=tindex;
      goto cskip;
      }
   delta=0;
   pritab[prind]=0;
   pritab[prind+1]=0;
bskip:
   joff=0;
   delta=0;
   newind=0;
   for (f=0; f<(prind+1)/2; f++) {
      count=pritab[2*f];
      for (j=0; j<count; j++) {
         newtmp[newind]=oldtmp[j+joff];
         newind=newind+1;
         }
      for (j=0; j<pritab[2*f+1]; j++) {
         newtmp[newind]=oldtmp[j+joff+count];
         newind=newind+1;
         }
      for (j=0; j<count; j++) {
         tz=oldtmp[j+joff];
         for (k=0; k<pritab[2*f+1]; k++) {
            pz=tz*oldtmp[k+count+joff];
            newtmp[newind]=pz;
            newind=newind+1;
            if (newind>oldnews) {
               printf("divisor table not big enough (3): N=%d \n",N);
               goto zskip;
               }
            }
```

```
         }
      joff=joff+pritab[2*f]+pritab[2*f+1];
      pritab[delta]=pritab[2*f]*pritab[2*f+1]+pritab[2*f]+pritab[2*f+1];
      delta=delta+1;
      }
   for (f=0; f<newind; f++)
      oldtmp[f]=newtmp[f];
   pritab[delta]=0;
   pritab[delta+1]=0;
   prind=delta;
   if (delta>1)
      goto bskip;
//
// compute j(x)
//
cskip:
   if ((newind+1)!=(unsigned int)in[2*g+1]) {
      printf("error: newind=%d %d \n",newind,(unsigned int)in[2*g+1]);
      goto zskip;
      }
   L=0;
   sum=0;
   if (N>Msize) {
      L=(unsigned int)(N/(unsigned long long)Msize);
      if (L>(unsigned long long)Tsize) {
         printf("error: not enough memory \n");
         goto zskip;
         }
      for (e=1; e<=(int)L; e++) {
         start=(N/(unsigned long long)e)/(unsigned long long)Msize+1;
         ltemp=newmert(start,N/e,M);
         if (ltemp==0x7fffffffffffffff) {
            printf("error: s>(x/t) \n");
            goto zskip;
            }
         T[e-1]=1-ltemp;
         }
      for (e=(int)(L/2); e>=1; e--) {
         sum=0;
#pragma omp parallel for reduction (+:sum)
         for (d=1; d<=(int)(L/(unsigned int)e-1); d++)
            sum=sum+T[(d+1)*e-1];
         T[e-1]=T[e-1]-sum;
         }
      sum=0;
      savet=(int)T[0];
#pragma omp parallel for reduction (+:sum)
      for (e=1; e<=(int)L; e++)
         if (N==(N/(unsigned long long)e)*(unsigned long long)e)
            sum=sum+T[e-1]*T[e-1];
      }
   sump=1;
   for (f=0; f<newind; f++) {
      tz=oldtmp[f];
      if (tz<=Msize) {
         t=M[tz-1];
         sump=sump+(long long)t*(long long)t;
         if (tz==N)
            savet=t;
         }
      }
   sum=sum+sump;
   printf(" %I64x %I64x %d %d %d \n",N,sum,newind+1,savet,L);
   fprintf(Outfp," %I64x, %I64x, %d, %d, \n",N,sum,newind+1,savet);
   }
zskip:
fclose(Outfp);
return;
}
```

```
/*CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  COMPUTE MERTENS FUNCTION (sum of M(x/i)^2 where x is highly composite)      C
C  02/06/16 (DKC)                                                              C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC*/
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "table.h"  // look-up table of the 1230 primes less than or equal to 10007
extern char *malloc();
// compute partial sums of Mertens function
long long newmert(unsigned long long s, unsigned long long x, int *M) {
unsigned long long t;
long long i;
long long sum;
t=(unsigned long long)sqrt((double)x);
t=t+2;
if (s>(x/t))
   return(0x7fffffffffffffff);
sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=(long long)s; i<=(long long)(x/t); i++)
   sum=sum+M[x/i-1];
#pragma omp parallel for reduction (+:sum)
for (i=1; i<(long long)t; i++) {
   sum=sum+(long long)M[i-1]*(x/(unsigned long long)i-x/(unsigned long long)(i+1));
   }
return(sum);
}
// compute Mobius function
int newmobl(unsigned long long a, unsigned long long b, long long *out, unsigned int *table,
            unsigned int tsize) {
unsigned int i,p,count;
unsigned long long beg,ps,rem;
long long t;
count=(unsigned int)(b-a);
for (i=0; i<count; i++)
   out[i]=1;
for (i=0; i<tsize; i++) {
   p=table[i];
   ps=(unsigned long long)p*(unsigned long long)p;
   if (ps>b)
      goto askip;
   rem=a-(a/ps)*ps;
   if (rem!=0)
      beg=ps-rem;
   else
      beg=0;
   while (beg<count) {
      out[beg]=0;
      beg=beg+ps;
      }
   rem=a-(a/p)*p;
   if (rem!=0)
      beg=p-rem;
   else
      beg=0;
   while (beg<count) {
      out[beg]=-out[beg]*p;
      beg=beg+p;
      }
   }
return(p);
askip:
for (i=0; i<count; i++) {
   if (out[i]==0)
      continue;
   t=out[i];
   if (t<0)
      t=-t;
   if ((unsigned long long)t<(i+a))
      out[i]=-out[i];
   }
for (i=0; i<count; i++) {
   if (out[i]==0)
      continue;
   if (out[i]>0)
      out[i]=1;
   else
```

```
      out[i]=-1;
   }
return(1);
}
// compute primes
unsigned int primed(unsigned int *out, unsigned int tsize,
                    unsigned int *table,unsigned int limit) {
unsigned int d;
unsigned int i,j,k,l,flag,count;
count=tsize;
for (i=0; i<tsize; i++)
   out[i]=table[i];
j=table[tsize-1]+1;
for (d=j; d<=limit; d++) {
   if(d==(d/2)*2) continue;
   if(d==(d/3)*3) continue;
   if(d==(d/5)*5) continue;
   if(d==(d/7)*7) continue;
   if(d==(d/11)*11) continue;
   l=(unsigned int)(10.0+sqrt((double)d));
   k=0;
   if (l>table[tsize-1])
      return(0);
   else {
      for (i=0; i<tsize; i++) {
         if (table[i]<=l)
            k=i;
         else
            break;
         }
      }
   flag=1;
   l=k;
   for (i=0; i<=l; i++) {
      k=table[i];
      if ((d/k)*k==d) {
         flag=0;
         break;
         }
      }
   if (flag==1)
      out[count]=d;
   count=count+flag;
   }
return(count);
}

unsigned long long Msize=15000000000;   // for 64 GB of RAM, 64-bit OS
unsigned int tsize=1230;          // prime look-up table size
unsigned int tmpsiz=10000;        // used to compute Mobius function
unsigned int t2size=200000;       // prime look-up table size
unsigned int Tsize=150000000;     // used to compute partial sums of Mobius function
//
unsigned int incnt=159;
unsigned long long in[167*2]={  // highly composite numbers and their number of divisors
                   2,           2,
                   4,           3,
                   6,           4,
                  12,           6,
                  24,           8,
                  36,           9,
                  48,          10,
                  60,          12,
                 120,          16,
                 180,          18,
                 240,          20,
                 360,          24,
                 720,          30,
                 840,          32,
                1260,          36,
                1680,          40,
                2520,          48,
                5040,          60,
                7560,          64,
               10080,          72,
               15120,          80,
               20160,          84,
               25200,          90,
               27720,          96,
               45360,         100,
               50400,         108,
```

55440,	120,
83160,	128,
110880,	144,
166320,	160,
221760,	168,
277200,	180,
332640,	192,
498960,	200,
554400,	216,
665280,	224,
720720,	240,
1081080,	256,
1441440,	288,
2162160,	320,
2882880,	336,
3603600,	360,
4324320,	384,
6486480,	400,
7207200,	432,
8648640,	448,
10810800,	480,
14414400,	504,
17297280,	512,
21621600,	576,
32432400,	600,
36756720,	640,
43243200,	672,
61261200,	720,
73513440,	768,
110270160,	800,
122522400,	864,
147026880,	896,
183783600,	960,
245044800,	1008,
294053760,	1024,
367567200,	1152,
551350800,	1200,
698377680,	1280,
735134400,	1344,
1102701600,	1440,
1396755360,	1536,
2095133040,	1600,
2205403200,	1680,
2327925600,	1728,
2793510720,	1792,
3491888400,	1920,
4655851200,	2016,
5587021440,	2048,
6983776800,	2304,
10475665200,	2400,
13967553600,	2688,
20951330400,	2880,
27935107200,	3072,
41902660800,	3360,
48886437600,	3456,
64250746560,	3584,
73329656400,	3600,
80313433200,	3840,
97772875200,	4032,
128501493120,	4096,
146659312800,	4320,
160626866400,	4608,
240940299600,	4800,
293318625600,	5040,
321253732800,	5376,
481880599200,	5760,
642507465600,	6144,
963761198400,	6720,
1124388064800,	6912,
1606268664000,	7168,
1686582097200,	7200,
1927522396800,	7680,
2248776129600,	8064,
3212537328000,	8192,
3373164194400,	8640,
4497552259200,	9216,
6746328388800,	10080,
8995104518400,	10368,
9316358251200,	10752,
13492656777600,	11520,
18632716502400,	12288,

```
        26985313555200,         12960,
        27949074753600,         13440,
        32607253879200,         13824,
        46581791256000,         14336,
        48910880818800,         14400,
        55898149507200,         15360,
        65214507758400,         16128,
        93163582512000,         16384,
        97821761637600,         17280,
       130429015516800,         18432,
       195643523275200,         20160,
       260858031033600,         20736,
       288807105787200,         21504,
       391287046550400,         23040,
       577614211574400,         24576,
       782574093100800,         25920,
       866421317361600,         26880,
      1010824870255200,         27648,
      1444035528936000,         28672,
      1516237305382800,         28800,
      1732842634723200,         30720,
      2021649740510400,         32256,
      2888071057872000,         32768,
      3032474610765600,         34560,
      4043299481020800,         36864,
      6064949221531200,         40320,
      8086598962041600,         41472,
     10108248702552000,         43008,
     12129898443062400,         46080,
     18194847664593600,         48384,
     20216497405104000,         49152,
     24259796886124800,         51840,
     30324746107656000,         53760,
     36389695329187200,         55296,
     48519593772249600,         57600,
     60649492215312000,         61440,
     72779390658374400,         62208,
     74801040398884800,         64512,
    106858629141264000,         65536,
    112201560598327200,         69120,
    149602080797769600,         73728,
    224403121196654400,         80640,
    299204161595539200,         82944,
    374005201994424000,         86016,
    448806242393308800,         92160,
    673209363589963200,         96768,
    748010403988848000,         98304,
    897612484786617600,        103680,
   1122015605983272000,        107520,
   1346418727179926400,        110592,
   1795224969573235200,        115200,
   2244031211966544000,        122880,
   2692837454359852800,        124416,
   3066842656354276800,        129024,
   4381203794791824000,        131072,
   4488062423933088000,        138240,
   6133685312708553600,        147456,
   8976124847866176000,        153600,
   9200527969062830400,        161280,
  12267370625417107200,        165888};
//
void main() {
long long *temp,sum,*T,ltemp,c;
int *M;
unsigned int *ntable;
unsigned int ntsize;
unsigned int g,L;
unsigned long long index,ta,tb,i,j,k,N,mcount,start;
int savet,t,ID,d,e;
FILE *Outfp;
Outfp = fopen("out1asr.dat","w");
omp_set_dynamic(0);
omp_set_num_threads(8);
#pragma omp parallel
{
   ID=omp_get_thread_num();
   printf(" ID=%d \n",ID);
}
ntable=(unsigned int*) malloc((t2size+1)*4);
if (ntable==NULL) {
   printf("not enough memory \n");
```

```
   goto zskip;
   }
temp=(long long*) malloc((tmpsiz+1)*8);
if (temp==NULL) {
   printf("not enough memory \n");
   return;
   }
M=(int*) malloc((Msize+1)*4);
if (M==NULL) {
   printf("not enough memory \n");
   return;
   }
T=(long long*)malloc((Tsize+1)*8);
if (T==NULL) {
   printf("not enough memory \n");
   return;
   }
ntsize=primed(ntable,tsize,table,2000000);
printf("prime look-up table size=%d, largest prime=%d \n",ntsize,ntable[ntsize-1]);
printf("computing Mobius function \n");
index=0;
ta=1;
tb=(unsigned long long)(tmpsiz+1);
mcount=Msize/(unsigned long long)tmpsiz;
for (i=0; i<mcount; i++) {
   t=newmobl(ta,tb,temp,ntable,ntsize);
   if (t!=1) {
      printf("error \n");
      goto zskip;
      }
   ta=tb;
   tb=tb+(unsigned long long)tmpsiz;
   for (j=0; j<tmpsiz; j++)
      M[j+index]=(int)temp[j];
   index=index+(unsigned long long)tmpsiz;
   }
//
// compute Mertens function
//
printf("computing Mertens function \n");
for (i=1; i<=Msize; i++) {
    M[i]=M[i-1]+M[i];
   }
//
printf("computing sum \n");
for (g=0; g<incnt; g++) {
   N=in[2*g];
   L=0;
   if (N>Msize) {
      L=(unsigned int)(N/(unsigned long long)Msize);
      if (L>(unsigned long long)Tsize) {
         printf("error: not enough memory \n");
         goto zskip;
         }
      for (e=1; e<=(int)L; e++) {
         start=(N/(unsigned long long)e)/(unsigned long long)Msize+1;
         ltemp=newmert(start,N/e,M);
         if (ltemp==0x7fffffffffffffff) {
            printf("error: s>(x/t) \n");
            goto zskip;
            }
         T[e-1]=1-ltemp;
         }
      for (e=(int)(L/2); e>=1; e--) {
         sum=0;
#pragma omp parallel for reduction (+:sum)
         for (d=1; d<=(int)(L/(unsigned int)e-1); d++)
            sum=sum+T[(d+1)*e-1];
         T[e-1]=T[e-1]-sum;
         }
      sum=0;
      savet=(int)T[0];
#pragma omp parallel for reduction (+:sum)
      for (e=1; e<=(int)L; e++)
         sum=sum+T[e-1]*T[e-1];
      k=(unsigned long long)sqrt((double)N);
      k=k+2;
      if ((L+1)>(N/(long long)k)) {
         printf("error: s>(x/t) \n");
         goto zskip;
         }
```

```
#pragma omp parallel for reduction (+:sum)
      for (c=(long long)(L+1); c<=(long long)(N/(long long)k); c++)
         sum=sum+(long long)M[N/c-1]*(long long)M[N/c-1];
#pragma omp parallel for reduction (+:sum)
      for (c=1; c<(long long)k; c++)
         sum=sum+(long long)M[c-1]*(long long)M[c-1]*(N/(unsigned long long)c-N/(unsigned long
long)(c+1));
      }
   else {
      sum=0;
      savet=M[N-1];
      k=(unsigned long long)sqrt((double)N);
      k=k+2;
#pragma omp parallel for reduction (+:sum)
      for (c=1; c<=(long long)(N/(long long)k); c++)
         sum=sum+(long long)M[N/c-1]*(long long)M[N/c-1];
#pragma omp parallel for reduction (+:sum)
      for (c=1; c<(long long)k; c++)
         sum=sum+(long long)M[c-1]*(long long)M[c-1]*(N/(unsigned long long)c-N/(unsigned long
long)(c+1));
      }
   printf(" %I64x %I64x %d %d %d \n",N,sum,in[2*g+1],savet,L);
   fprintf(Outfp," %I64x, %I64x, %d, %d, \n",N,sum,in[2*g+1],savet);
   }
zskip:
fclose(Outfp);
return;
}
```