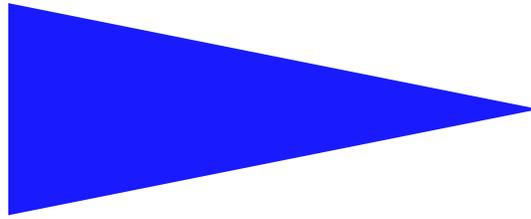


PUBLICATION
INTERNE
N° 1075



IMPROVED BOUNDS FOR GOLDBACH CONJECTURE
(FIRST VERSION)

YANNICK SAOUTER

Improved bounds for Goldbach conjecture (first version)

Yannick Saouter

Thème 1 — Réseaux et systèmes
Projet API

Publication interne n° 1075 — Janvier 1997 — 16 pages

Abstract: Goldbach's conjecture states that every even integer greater or equal to 6 is the sum of two prime numbers. This result is still unproved. This conjecture has been numerically checked up to $4 \cdot 10^{11}$ on an IBM 3083 mainframe. We describe here an implementation on a less powerful machine which raises the bound to 10^{12} .

Key-words: Prime numbers; Goldbach's problem

(Résumé : tsvp)

Améliorations de bornes au sujet de la conjecture de Goldbach (première version)

Résumé : La conjecture de Goldbach stipule que tout nombre pair supérieur ou égal à 6 est somme de deux nombres premiers. Ce résultat est à ce jour non démontré. Il a été vérifié numériquement jusqu'à $4 \cdot 10^{11}$ sur un IBM 3083. Nous décrivons ici une implémentation sur une machine moins puissante qui élève cette borne à 10^{12} .

Mots-clé : Nombres premiers; Problème de Goldbach

1 Introduction

In 1742, in a letter to Euler, Goldbach made the following conjecture: “Every even number is the sum of two prime numbers”. This conjecture, which is also a subpart of the eighth Hilbert’s problem, is widely believed to be true but has never been proved. Some close results have been obtained: (i) there exists a number S such that every integer is the sum of at most S numbers [1], (ii) every even number large enough is the sum of a prime number and the product of two prime numbers [2], (iii) every odd number large enough is the sum of three prime numbers [3] in the general case and with lower bound under the assumption of general Riemann hypothesis [4]. Moreover this conjecture has been verified numerically up to 4×10^{11} . In this paper we present an implementation which have enabled us to check this conjecture up to 10^{12} .

2 Definitions

We remind here classical notions.

Definition 2.1 (prime number) *Let p be an integer strictly greater than 1. Then p is said to be a prime number if and only if it is divisible by only 1 and p .*

Definition 2.2 (Goldbach partition) *Let n be an even integer greater or equal to 6. Then any couple (p, q) where p and q are odd prime numbers and $n = p + q$ will be called a Goldbach partition of n . If n has such partitions, the one which involves the least value for p will be called the least Goldbach partition of n .*

Definition 2.3 (pseudoprime) *Let a and p be integers such that $1 < a < p$. Then p is said to be a pseudoprime for basis a if the congruence $a^{p-1} \equiv 1 \pmod{p}$ holds.*

We have then:

Theorem 2.1 (Fermat’s theorem) *Let p be a prime number. Then p is pseudoprime for any basis $1 < a < p$.*

This congruence can be checked in $\mathcal{O}(\ln(p))$ operations on integers of the size of p by the classical binary method of exponentiation, leading to approximatively $\mathcal{O}(\ln^2(p) \cdot 2^{\sqrt{2 \cdot \ln(\ln(p))}})$ elementary operations if using the Schönage algorithm of multiplications for instance (see [5, p. 272]).

This latter theorem would give quite a good characterization of prime numbers if it was reciprocal. But we have for instance $2^{340} \equiv 1 \pmod{341}$, while $341 = 31 \times 11$. Moreover there exists numbers which are pseudoprimes for any basis. Such numbers are called Carmichael numbers [6] and it is known that their cardinal is infinite [7]. The least one is 561.

In order to strenghten the test Miller and Rabin [8] introduced strong pseudoprimes.

Definition 2.4 (strong pseudoprime) *Let p be an odd number and let $p = 1 + 2^K \cdot d$, where d is odd and $K > 0$. Then p will be said to be a strong pseudoprime for the basis a if we have either $a^d \equiv 1 \pmod{p}$ or $a^{2^k \cdot d} \equiv -1 \pmod{p}$ with $0 \leq k < K$.*

Again any prime number is a pseudoprime for any basis but composite strong pseudoprimes still exist: e.g. 2047 is the least one for basis 2. However there is a converse theorem: a composite number cannot be strong pseudoprimes for more than 1 basis out of 4. Moreover, under the assumption of generalized Riemann hypothesis, we have [9]:

Theorem 2.2 *If the GRH is true, then an odd number p is prime if and only if it is a strong pseudoprime for any basis a such that $1 < a < 2 \cdot \ln^2(p)$.*

With this hypothesis, the characterization of prime numbers would be of polynomial complexity.

Strong pseudoprimality to several basis gives a quite good insurance of primality. For instance (see [10, p. 96]):

Theorem 2.3 *Let p be an odd integer. If p is strong pseudoprime for basis 2 and 3, and $p < 161304001$, then p is a prime number unless $p = 25326001$.*

This theorem was used to compute the table of prime numbers up to 128000001 (see section 3).

We also used results of Jaeschke [11]:

Theorem 2.4 *If p is strong pseudoprime for basis 2, 13, 23, 1662803 and $p < 10^{12}$, then p is a prime number.*

Theorem 2.5 *If p is strong pseudoprime for basis 2, 3, 5, 7, 11, 13 and 17 and $p < 34155007178321 \simeq 3.4 \times 10^{14}$, then p is a prime number.*

3 Implementation choices

In his work [12], Sinisalo interested himself to the computation of the least Goldbach partition of even numbers and compared his computations to asymptotic conjectures. In our implementation, we only focus ourselves in the verification of the conjecture, by finding any possible partition, that is less time-consuming. Indeed, once you have found a partition $2n = p + q$, then you can deduce partitions for any even number $p' + q$ where p' is a prime number strictly superior to p .

On probabilistic grounds, the growth of the least prime numbers giving a Goldbach partition of $2n$ is slow: this number should be small in regard of $\ln^2(n)$. $\ln(\ln(n))$ [13]. Moreover the number of distinct partitions should be quite important and asymptotically proportionnal to $n/\ln^2(n)$ [14]. As a consequence, if one precompute a large database of prime numbers, one should be able to deduce many Goldbach partitions from a given one by translation. Thus we begin the development by the computation of the odd prime numbers up to 128000001 included. Using an implementation of theorem 2.3, it took five hours for a DECSTATION 3100 generate the database. Every odd number from 3 to 128000001 was represented by a single bit in the result file, this bit being equal to 1 if and only if the corresponding number is prime (see section B).

Our algorithm was then to check the conjecture by integer slices of 128000000 units wide. The table of integers was also represented by a single bit, this bit being equal to 1 if and only if a Goldbach partition was already found for the corresponding number.

Both these bit arrays were implemented as integer arrays, and the algorithm we implemented was roughly the following one:

Algorithm A

Verification of the Goldbach conjecture on the interval $[N_0, N_0 + 128000000]$.

(Integer ranges here from 1 to $2^l - 1$).

- **Step 1** *Load the integer array of the precomputed odd primes numbers in P . The number of the entries is then $128000000/(2^{l+1})$.*
- **Step 2** *Set the array G of the bit representation of Goldbach partitions to 0. The number of entries of this array is also $128000000/(2^{l+1})$ since only even numbers are considered.*
- **Step 3** *Find the least entry i in G and the least number j such that the j -th bit of $G[i]$ is not equal to 1. If no more such values exist, the pass is finished.*

- **Step 4** Find the least value k such that

- the j -th bit of $P[k]$ is equal to 1,
- the number $N_0 + 2^{l+1} * (i - k) - 3$ is prime.

If such a value k exists, the couple $(N_0 + 2^{l+1} * (i - k) - 3, 2^{l+1} * k + 3)$ is a Goldbach partition of $N_0 + 2^{l+1} * i$. If no such value exists, abandon the entire pass of verification.

- **Step 5** Set m_0 to i and m_1 to k . While m_0 and m_1 are strictly inferior to $128000000/(2^{l+1})$, set the bit values of $G(m_0)$ to 1 if the corresponding bit values of $P(m_1)$ are equal to 1 and increment both m_0 and m_1 .
- **Step 6** Goto **Step 3**.

The primality test of $N_0 + 2^{l+1} * (i - k) - 3$ in **Step 4** was performed by implementing theorem 2.4. Moreover **Step 5** can be efficiently implemented by making **or** operations on integer (thus in only one machine cycle) which avoids making test values on the bits.

First it has to be noted that in every version of the program we wrote, the time spent in **Step 5** was at least of 96 % of the total time. That explained why we do not care to optimize the code for the other parts of the program: for instance the function `powermod` was implemented recursively and not iteratively (see B).

Second we made two different implementations of the program. In both, the array G was a regular array of integer but its access was not dealt the same way. In the first version, the array was classically addressed by an integer index incremented from one access to the following one. In the second version, we used a linked chain of indexes which was stored in another integer array. This array contain at the entry i the following address j of entry in G such that $G[j]$ is not equal to $2^l - 1$ (i.e. $G[j]$ contain still bits equal to 0). This implementation was motivated by practical remarks: for instance when dealing with a 32 bits integer implementation, and with an initial value $N_0 = 1000000$, when the 20 first entries of G are set to $2^{32} - 1$ by **Step 5**, in fact already 4490 entries of the array G are also equal to this value. As we will see, although the use of linked chains is, in general heavy, since it requires indirect addressing, the gain obtained by these versions were of about 40 % in regard of the corresponding regular implementation.

Third we have the problem of choosing the integer size in our implementations. In our final processing, we used a `DECSTATION 3100` to make our computations which enable up to 64 bits integers. Thus this give an inherent parallelism of at most 64

Integer size	Regular version	Linked chains version
32 bits	2:08.9 mn	1:23.3 mn
16 bits	4:16.5 mn	2:17.8 mn
8 bits	7:57.7 mn	4:11.7 mn

Table 1: Computation (user) times with respect to the integer size

Array size (in 32 bits integers)	Regular version	Linked chains version	Ratio
$2 * 10^6$	2:08.9 mn	1:23.3 mn	1.55
$1.5 * 10^6$	1:36.3 mn	1:00.6 mn	1.59
10^6	59.8 s	38.6 s	1.55
$8 * 10^5$	50.0 s	32.4 s	1.54
$5 * 10^5$	30.1 s	22.0 s	1.37
10^5	6.1 s	5.0 s	1.22

Table 2: Ratios of computation time with respect to the size of prime database

in the computations of array G . The drawback of taking the maximum length is that in **Step 3**, i will increase slower for the regular versions and the length of the linked chain of indexes will decrease slower in the alternative versions. Since as we have said, at most 96 % of the time is spent in **Step 5**, there is few chances that the drawback will be sufficient to erase the advantage of internal parallelism in long integers. However to establish this fact, we perform a set of experiments on a SUN 4M with a maximum number of 32 bits. We test both implementations for integers coded on 8, 16 and 32 integers, from an initial value $N_0 = 1000000$ for one pass. The table 1 summarizes timing results and illustrates well the need to implement the arrays with using the longest possible integers.

Another experiment we made was to compute the gain of the use of linked chains in regard of the length of the table of prime number precomputed. From an initial value $N_0 = 1000000$ for one pass and arrays stored in 32-bits integers we compute the timings of the regular version and of the linked chain version and compute the ratios. These results are collected in table 2.

4 Performances

We design our code with 64-bits integers for a DECSTATION 3100. This machine is built around an Alpha 21064 processor scheduled at 100 MHz frequency. This machine has no pipeline integer arithmetic unit (pipelining is limited to floating point registers) thus it was no use of making data parallelism in the main loop of the `copy` function of the main program. Indeed in this loop you have a direct dependance from an iteration to the following one: the value of the pointer `nextindpt` is used at the beginning of the loop and set at the end. Thus if we have had pipelined arithmetical units, the values of the others integers involved in this cycle of dependance would have been to be computed inside a single loop and thus the crossing of pipeline queues would have been a factor of loss of speed. In this case to overcome this problem we would have had to juxtapose one or several copies instance in order to fill up the pipeline queue.

It took 97 hours to reach the limit of 10^{12} . This time has to be compared to the 130 hours necessary to Sinisalo on an IBM 3083 vectorial mainframe to compute the least Goldbach partition of even numbers up to 4.10^{11} . A total number of 1340626 prime numbers were used to reach this bound, the first one being 998717 (the first pass began at 10^6) and the last one being 999999700541.

5 Going further

It is possible, involving more computing resources to gain still several order of magnitudes by using the same technique. That will be the next point to be addressed: we are aiming to make an implementation on a cluster of 18 R8000 processors on hosts of the CRIN (Centre de Recherche en Informatique de Nancy, France). The prime certificate we will use is the one of theorem 2.5 and we are aiming to reach a limit of 2.10^{13} still in approximatively the same scheduling time.

In fact, if we desire to gain still several orders of magnitude, there is no real problem with the prime certificate. In this implementation, we used a certificate which establishes effectively the primality. Although methods using pseudo-primality tests are used here nearly at their limit, it is possible to establish primality of number of twenty or a bit more decimal digits in reasonable time with well-known methods such as [5, p. 347], and thus this is not a limitative constraint. However to increase the upper bound a quite huge amount of computational resources would be necessary.

Another solution would be to speed up arrays copy by dedicated hardware. Indeed, the copy procedure might well be done in parallel and pipeline fashion with regard to the generation of prime numbers with the host. The hardware architecture would consist then into the two arrays (one for the even numbers and one for the table of precomputed prime numbers). Given the initial value N_0 , the host would generate the least prime number p such that $N - p$ is also a prime number and would send to the hardware card the shift of index entries to apply to the prime array. The card would then iteratively make **or** operations and as soon as that it would detect a '0' in the bit stream it would send back to the host the corresponding offset. The host would then be able to search for the next necessary prime number before the entire copy was done. If we add a system of pipeline queue request to the card, it should be possible to dramatically decrease the time needed to make a pass. However, reconfigurable systems, such as FPGAs, are still not large enough to support such applications.

References

- [1] L. Schnirelmann. Über additive Eigenschaften von Zahlen. *Math. Ann.*, (107):649–660, 1933.
- [2] J.R. Chen. On the representation of a large even integer as the sum of a prime and the product of at most two primes. *Kexue Tongbao*, (17):385–386, 1966.
- [3] I.M. Vinogradov. Representation of an odd number as the sum of three primes. *Dokl. Akad. Nauk SSSR*, (15):169–172, 1937.
- [4] T.Z. Wang and J.R. Chen. On odd Goldbach problem under general Riemann hypothesis. *Sci. China Ser.*, (6):692–693, 1993.
- [5] D. Knuth. *The Art of Computer Programming: Seminumerical algorithms*. Addison-Wesley, 1969.
- [6] R.D. Carmichael. On composite numbers p which satisfy the Fermat congruence $a^{p-1} \equiv 1 \pmod{p}$. *Amer. Math. Monthly*, (19):22–27, 1912.
- [7] W.R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Ann. Math.*, (140):703–722, 1994.
- [8] G.L. Miller. Riemann's hypothesis and tests for primality. *J. Comp. Sys. Sci*, (13):300–317, 1976.

- [9] E. Bach. *Analytic methods in the Analysis and design of number-theoretic algorithms*. MIT Press, Cambridge, 1976.
- [10] P. Ribenboim. *The Book of Prime Number Records*. Springer-Verlag, 1988.
- [11] G. Jaeschke. On strong pseudoprimes to several bases. *Math. Comp.*, 61(204):915–926, 1993.
- [12] M.K. Sinisalo. Checking the Goldbach conjecture up to $4 \cdot 10^{11}$. *Math. Comp.*, 61(204):931–934, 1993.
- [13] A. Granville, J. van de Lune, and H.J.J. te Riele. *Number Theory and Applications*, chapter Checking the Goldbach conjecture on a vector computer. Kluwer, 1989.
- [14] V. Brun. Über das Goldbasche Gesetz und die Anzahl der Primzahlpaare. *Archiv für Math. og Naturv.*, XXXIV(8), 1915.
- [15] S.L. Graham, P.B. Kessler, and M.K. McKusick. `gprof`: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, 1982.

A Statistics

In this section, we give the run-time statistics given by the profiler `gprof` [15] for an implementation on a `DECSTATION 3100` of the program with linked chains. This implementation used arrays of 1000000 integers of 64 bits wide and compute the G array from the initial value $N_0 = 1000000$ up to 129000000. This illustrates well that most of the computation time is spent in the `copy` routine.

granularity: each sample hit covers 8 byte(s) for 0.00% of 34.03 seconds

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
97.3	33.13	33.13	169	196.01	196.01	copy [3]
2.3	33.90	0.78	80624	0.01	0.01	mulmod [6]
0.3	34.01	0.11	1	105.47	34030.27	main [1]
0.1	34.03	0.02	2493	0.01	0.30	powermod [7]
0.0	34.03	0.00	2493	0.00	0.32	miller [5]
0.0	34.03	0.00	1986	0.00	0.40	isprime [4]
0.0	34.03	0.00	1	0.00	0.00	myprintfu [8]

Index by function name

[3] copy	[5] miller	[7] powermod
[4] isprime	[6] mulmod	
[1] main	[8] myprintfu	

B Main program

```

/* Code for checking of Goldbach conjecture */
/* 64 bit integers. All integers are supposed to */
/* be less than 2^63 */

#include <stdio.h>

#define addmod(a,b,M) ((a+b)%M)
#define min(x,y) ((x>y)?y:x)

/* Linked list implemented */

int BEGIN;
int *NEXTINDEX;

unsigned long *array, *table;

myprintfu(unsigned long u)
{
    char *s;
    unsigned long tmp;
    int i,j;

    s=(char *)malloc(20*sizeof(char));
    if (!(u)) { s[0]='\0';s[1]='\0'; }
    else {
        for (i=0,tmp=u; tmp!=0;tmp/=10,i++) s[i]=(tmp % 10)+'0';
        s[i]='\0'; }
    for (j=strlen(s);j>=0;j--) { putchar(s[j]); fputc(s[j],stderr); }
}

unsigned long mulmod(unsigned long a, unsigned long b, unsigned long M)
{
    unsigned long ret;
    int i;

```

```

unsigned long a1,a2,b1,b2,tmp;

a1=((a>>16)&65535)<<16)+(a & 65535);
b1=((b>>16)&65535)<<16)+(b & 65535);
a2=(a >> 32);
b2=(b >> 32);
ret=(a2*b2)%M;
for (i=0;i<32;i++)
    ret=((ret<<1)%M);
ret+=(a1*b2);
ret%=M;
ret+=(a2*b1);
ret%=M;
for (i=0;i<32;i++)
    ret=((ret<<1)%M);
tmp=(a1*b1)%M;
ret=(ret+tmp)%M;
return ret;
}

unsigned long powermod(unsigned a, unsigned long b, unsigned long M)
{
    unsigned long tmp,ret;

    if (b==0) ret=1;
    else if (b==1) ret=a%M;
    else {
        if (b & 1)
            { tmp=powermod(a,(b>>1),M);
              tmp=mulmod(tmp,tmp,M);
              ret=mulmod(a,tmp,M) ; }
        else {
            tmp=powermod(a,(b>>1),M);
            ret=mulmod(tmp,tmp,M); }}
    return ret;
}

miller(unsigned long base, unsigned long p)
{
    unsigned long d,tmp;
    int h,ret,k,cont;

```

```

d=p-1;
h=0; ret=0;
while (!(d & 1)) { d>>=1; h++; }
tmp=powermod(base,d,p);
if (tmp==1) ret=1;
else {
    cont=1;for (k=0;(k<h)&&(cont);k++) {
        if (tmp==(p-1)) { ret=1; cont=0; }
        tmp=mulmod(tmp,tmp,p); } }
return ret;
}

isprime(unsigned long p)
{
    /* Test for primality valid up to 10^12 */
    int ret;

    switch(p) {
    case 2:
    case 13:
    case 23:
    case 1662803:
        ret=1;
        break;
    default:
        ret=miller(2,p);
        if (ret) ret=miller(13,p);
        if (ret) ret=miller(23,p);
        if (ret) ret=miller(1662803,p);
        break; }
    return ret;
}

copy(int k)
{
    int k1,pt,t,nt,diff;
    unsigned long *dbk,*a,u;
    int *nextindt, *nextindpt;

    dbk=table+(k1=k);
    pt=t=BEGIN;
    a=array+BEGIN;

```

```

nextindpt=nextindt=NEXTINDEX+t;
while ((t>=0) && (k1<1000000)) {
    u=((*a) | = (*dbk));
    nt=*nextindt;
    if (u==18446744073709551615)
        (*nextindpt)=nt;
    else
        { pt=t;
          nextindpt=nextindt; }
    k1+=(diff=(nt-t));
    dbk+=diff;
    a+=diff;
    t=nt; nextindt+=diff; }
if (array[BEGIN]==18446744073709551615)
    BEGIN=NEXTINDEX[BEGIN];
}

main()
{
    unsigned long *array2,tmp,mj,seed;
    unsigned long *a,*t;
    int i,j,k,l,m;
    int cont;
    FILE *infile;
    int nbprimesused;

    table=(unsigned long *)malloc(1000000*sizeof(unsigned long));
    infile=fopen("databaseprime.altair","r");
    fread(table,sizeof(unsigned long),1000000,infile);
    /* Table is an array of bits. (table[i]&(1<<j)) is equal to */
    /* 1 iff 128*i+2*j+3 is prime. */
    fclose(infile);
    seed=1000000;
    array=(unsigned long *)malloc(1000000*sizeof(unsigned long));
    /* Array is a table of bits. (array[i]&(1<<j)) will be equal */
    /* to 1 iff 128*i + 2*j + seed is effectively decomposable as */
    /* the sum of two prime numbers */
    NEXTINDEX=(int *)malloc(1000000*sizeof(int));
    /* NEXTINDEX will contain the linked chain of indexes */
    for(;seed<1000001;seed+=128000000) {

```

```
nbprimesused=0;
for (i=0;i<1000000;i++) NEXTINDEX[i]=i+1;
NEXTINDEX[999999]=-1; BEGIN=0;
memset(array,0,1000000*sizeof(unsigned long));
while (BEGIN>=0) {
    tmp=(array+BEGIN); j=0; mj=1;
    while (tmp & 1) { tmp>>=1; j++; mj<<=1;}
    cont=1;
    for (k=0;(cont);k++) {
if (k==1000000) { myprintfu(mj); printf(" Overflow\n"); exit(0); }
if (table[k]&mj) {
    tmp=seed-3+128*BEGIN-128*k;
    if (isprime(tmp)) {
        nbprimesused++;
        cont=0;
        copy(k); }
}
    }
}
myprintfu(seed);
printf(" -> %u prime numbers used\n",nbprimesused);
fprintf(stderr," -> %u prime numbers used\n",nbprimesused); }
}
```