The Complexity of Goldbach's Conjecture and Riemann's Hypothesis

Elena Calude

Massey University at Albany, NZ

CDMTCS-370
August 2009

The Complexity of Goldbach's Conjecture and Riemann's Hypothesis

Elena Calude
Massey University at Albany, New Zealand
`http://www.massey.ac.nz/~ecalude`

August 16, 2009

Abstract

In this paper we obtain better upper bounds on the complexities of Goldbach's Conjecture and Riemann's Hypothesis in [3] and [8] by improving the register machine language used as well as the optimisation technique.

1 Introduction

Goldbach's conjecture, which is part of Hilbert's eighth problem [9], states that

all positive even integers greater than two can be expressed as the sum of two primes.

One can write a program Π_{Goldbach} which just enumerates all positive even integers n greater than two, and for each of them checks the required property, i.e. checks whether n can be expressed as the sum of two primes; the program Π_{Goldbach} stops if and only if it finds a counter-example for Goldbach's conjecture. Re-phrased: Π_{Goldbach} never stops if and only if Goldbach's conjecture is true.

The Riemann hypothesis is probably the most famous/important conjecture in mathematics. It appears in Hilbert's eighth problem [9]: the non-trivial complex zeros of Riemann's zeta function, which is defined for $Re(s) > 1$ by

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s},$$

lie exactly on the line $Re(s) = 1/2$.

According to Matiyasevich [10], p. 119–121, the negation of the Riemann hypothesis is equivalent to the existence of positive integers k, l, m, n satisfying the following six conditions (here $x \mid z$ means "x divides z"):

1. $n \geq 600$,
2. $\forall y < n\, [(y+1) \mid m]$,
3. $m > 0 \,\&\, \forall y < m\, [y = m \vee \exists x < n\, [\neg\, [(x+1) \mid y]]]$,
4. $explog(m-1, l)$,
5. $explog(n-1, k)$,
6. $(l-n)^2 > 4n^2k^4$,

and $explog(a, b)$ denotes the predicate

$$\exists x\, [x > b+1 \,\&\, (1+1/x)^{xb} \leq a+1 < 4(1+1/x)^{xb}].$$

An inspection of the above conditions shows that the Riemann hypothesis is of the form $\forall n, R(n)$, where R is a computable predicate. Hence, one can write a program Π_{Riemann} such that the Riemann hypothesis is false if and only if Π_{Riemann} halts.

In [3] we used a register machine language which implements a (natural) universal prefix-free Turing machine to write Π_{Goldbach} and Π_{Riemann}, and we found that the sizes in bits of those programs are as follows: Π_{Goldbach} has 135 instructions totaling length 871 4-bit characters, i.e. its size in bits is 3,484, and Π_{Riemann} consists of 290 instructions and 1953 4-bit characters (7,780 bits).

2 The method

Both conjectures above have the form $\forall n, R(n)$, where R is a computable predicate. Hence, one can write a program Π_{R} such that the hypothesis is false if and only if Π_{R} halts. Following [3, 4] we use the program Π_{R} as a measure of complexity of the conjecture R by counting the number of bits necessary to specify Π_{R} in some fixed "universal formalism" (a universal self-delimiting Turing machine [2]). Of course, there are many programs equivalent to Π_{R}, so a natural way to evaluate the complexity is to consider the *smallest* such program [3].

The choice of the universal formalism used to code programs is irrelevant up to an additive constant, so if a problem is significantly more complex in some fixed formalism than another one, then it will continue to be more complex in any other formalism. A drawback of the approach comes from the fact that the proposed measure is *uncomputable* (see [2]), so we have to work with an upper bound on the size of a program "describing" the conjecture/problem/theorem.

3 A universal programming language

We briefly describe the syntax and the semantics of the register machine language which implements a (natural) universal prefix-free Turing machine (it is a refinement of the languages described in [7, 3]) and is used to obtain the program $\Pi_{4\mathrm{CT}}$.

Any register machine has a finite number of registers, each of which may contain an arbitrarily large non-negative binary integer. The list of instructions is given below in two forms: our compact form and its corresponding Chaitin style version [7].

By default, all registers, labeled with a string of lower or upper case letters, are initialised to 0. Instructions are labeled by default with 0,1,2,... (in binary).

The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a label, a register or a binary constant of the form $1(0+1)^* + 0$, while R1 must be a register variable.

=R1,R2,R3 **(EQ R1 R2 R3)**

If the contents of R1 and R2 are equal, then the execution continues at the R3-th instruction, where R3 = 0 denotes the first instruction of the program. If they are not equal, then execution continues with the next instruction in sequence. If the content of R3 is outside the scope of the program, then we have an illegal branch error.

&R1,R2 **(SET R1 R2)**

The contents of register R1 is replaced by the contents of register R2.

+R1,R2 **(ADD R1 R2)**

The contents of register R1 is replaced by the sum of the contents of registers R1 and R2.

!R1 **(READ R1)**

One bit is read into the register R1, so the numerical value of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.

% **(HALT)**

This is the last instruction for each register machine program before the raw data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A *register machine program* consists of a finite list of labeled instructions from the above list, with the restriction that the `HALT` instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the `HALT` instruction. A program not reading the whole data or attempting to read past the last data-bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

Instructions can be translated directly in binary because the set of instructions is prefix-free (each instruction starts with one symbol from the set $\{=, \&, +, \#, \ \%\}$). The unique binary word coding a program is then the concatenation of the binary words corresponding to the sequence of its instructions. The *length of the program* is the length of the binary word describing the program. Comments do not count for the size of the program.

To improve development and presentation of the programs we use a consistent style for routines. The main program is a set of instructions that performs a specific task. A routine is a set of instructions that performs a specific task for another routine or main program, requiring, on completion of the task, direction back to the proper place in the calling routine or main program. Our routines are unary or binary only. We use the following conventions:

1. The register names and the values they store are used interchangeably.
2. The letter `L` followed by characters $(1, \ldots, 9)$ and terminated by ':' is used to mark line numbers in the main program. Routines labels include an extra letter, which is different for each routine, so that the labels are local within the routine. References to labels are replaced with the binary constant in the final program.
3. For all routines, registers a and b are used for arguments, c is used for the return line, and d for the result. The values stored in a, b and c are unchanged on return from routines. In case of a unary routine, b is not used. The value stored in c is set outside the routine. When the values of a, b and c are changed during the execution of a routine, they have to be restored on return.
4. The instruction `=a,a,Ln` is used for the unconditional jump to the instruction to line n; it will be abbreviated as `GoTo Ln`.
5. The read instruction is used here to guarantee universality of the language, but will not appear in our programs.
6. The notation #`RoutineName` is used to call `RoutineName` which produces the result in register d. Operationally, #RoutineName is an unconditional jump `GoTo L`#`RoutineName` to the first instruction of #`RoutineName`. The first line of each routine is labeled with `L` followed by the name of the routine.
7. For Boolean data types we use integers $0 =$ `false` and $1 =$ `true`.

The instructions used in the register machine programs are of four types:

1. instructions of more than six characters (e.g. `=h,0,LCMP` of size 5+ the size of `CMP`),
2. instructions of four characters (e.g. `+e,1` or `&g,d`),
3. instructions of more than two characters (e.g. `#IPR` of size 2+ the size of `IPR`),

4. one character instruction: %.

Finally, we note that the running time efficiency of a program is irrelevant for the problems we solve in this paper. The goal in designing an algorithm for testing a conjecture is to minimize the number of instructions of the program, as this number is used as an upper bound of the measure of the complexity of a conjecture represented by the program. See more details and comments in [4].

4 The program Π_{Goldbach}

In this section we present the program Π_{Goldbach}:

```
      =a,a,L0  //jump to start Goldbach program
//SUB routine returns a-b, assumes a>=b>=1
//Total:  6 instrs, 4 of 4 chars= 16,
//2 of 6 chars=12, Total:  28 chars
LSUB: &x, b
      &d,0
LS1:  =x,a,c
      +x,1
      +d,1
      =a,a LS1
//REM routine returns integer remainder
//of a divided by b in d, a>=b>=2
//Total:  9 instrs, 4 of 6 chars =24,
//5 of 4 chars=20 chars, Total: 44 chars
LREM: &x, b
      &y, 0
LR4: =x,a,c
     +x,1
     +y,1
     =y, b, LR1
     =a,a,LR4
LR1: &y,0
     =a,a,LR4
//IS PRIME ROUTINE: IPR, routine returns 1 if a>=2 prime,
//otherwise returns 0, calls REM
//Total:18 instructs, 13 of 4 chars=52 chars
// 4 of 6 chars=24chars and 1 of 2 chars=2, Total: 78 chars.
LIPR:&aa,a
     &bb,b,
     &cc,c
     &p,2
     &d,1
LI1: =a,p, LI5
     &b, p    // call REM a,p
     &c, LI3
     #REM
     &q,d
LI3: =q,0, LI4
     +p,1
     =a,a,LI1
LI4: &d,0
LI5: &a,aa  //prepare exit
 &b,bb
 &c,cc
     =a,a, c
//---------------Goldbach starts here
L0: &e, 4         //enumeration of all even numbers >=4
L1: &f, 2        //enumeration of all primes >=2
L2: =f, e, L8
    &c, L5      //calling IS -PRIME(a), with a = f= NUM
    &a, f
    #IPR    //returned from IS-PRIME
    &g, d
L5: =g, 1, L4
L3: +f, 1     // NUM( or f) not prime, try next NUM
    =f, f, L2      // jump to L2
L4: &c, L6        // calling a - b (named SUB)
    &a, e
    &b, f
    #SUB    //returned from SUB
    &g, d          // h is now e-f (or EVEN-NUM)
L6: &c, L7      //calling IS_PRIME(a), with a = h= EVEN-NUM
    &a, g
    #IPR    //returned from IS-PRIME
    &g, d
L7: =g, 0, L3
    +e, 2  //test next EVEN
    =a, a, L1
L8: &d, 0
    %
```

The“binary” form of the program used to calculate its size is the following:

```
0000  =a,a,10000
0001  & e,10
0010  & d,1
0011  = a,e,c
0100  & d,0
0101  & f,e
0110  = f,a,1101
0111  + f,1
1000  + d,1
1001  = d,e,1011
1010  = a,a,0110
```

```
1011  & d,0
1100  = a,a,0110
1101  = d,0,c
1110  + e,1
1111  = a,a,0010
0001 0000 & g,100
0001 0001 & h,10
0001 0010 = g,h,100110
0001 0011 & c,10110
0001 0100 & a,h
0001 0101 # 0001
0001 0110 = d,0,100011
0001 0111 & i,0
0001 1000 & k,h
0001 1001 = k,q,11101
0001 1010 + i,1
0001 1011 + k,1
0001 1100 = a,a,1101
0001 1101 & c,100000
0001 1110 & a,i
0001 1111 # 0001
0010 0000 = d,0,100011
0010 0001 + g,10
0010 0010 = a,a,10001
0010 0011 + h,1
0010 0100 = a,a,10010
0010 0101 & d,0
0010 0110 %
```

The Goldbach's register machine program has 57 instructions and 247 characters (6 instructions and 28 chars from SUB, 9 instructions and 44 characters from REM, 18 instructions and 78 characters from IPR, 24 instructions and 97 characters from the main program, 20 characters from labels). So the total number of characters is 267 that gives a total of 1,068 bits. The sizes of the programs for the Goldbach's conjectures in [3] and [8] are 3,484 and 1,628 bits, respectively.

5 The program Π_{Riemann}

In this section we present the program Π_{Riemann}.

```
&a,Riemann
=b,c,RIEM     // b=c=0 so jumps to start of Riemann algorithm
// CMP(a,b) returns 1 if a<b, 0 if a=b, or 2 if a>b
//It does not invoke any other routine
//Elena version:7 instr x 4 chars= 28 chars,4 instr x 6 chars=24 chars;
//11 instr, Total:  52 chars
LCMP:&e,a
```

```
&f,b
LC1: +e,1
+f,1
&d,0
=e,f,c
&d,1
=e,b,c
&d,2
=f,a,c
=a,a,LC1
// Pow(a,b) returns a^b where we assume a>0 and b>0
//It uses  MUL (a,b) =a*b--a  will not change so it is not copied on the local stack
//Elena: 9 instr of 4 char =36 chars,2 instr x 6=12 chars,1 instr of 2 chars=2 chars
//Total:12 instrs, =50 chars
LPOW:&e,b     // keep our own runtime stack for b,c
&f,c
&x,1
&d,a
LP1:=x,b,c
&b,d  // get ready to compute a*d
&c,LP3
#MUL    // replace with address of routine MUL
LP3: +x,1
&b,e       // unpop stack to restore b and c
&c,f
=a,a,LP1
//SUB routine returns a-b, assumes a>=b>=1
//it does not call any other routine
//4 instr x 4 chars=16 chars, 3 instructs x 6 chars= 18,
//Total:  7 instrs,  34 chars
LSUB: &x, b
      &d,0
LS1:  =x,a, LS2
      +x,1
      +d,1
      =a,a LS1
LS2:  =a,a,c
//REM routine returns integer remainder of a divided by b, assumes a>=b>2
//it does not call any other routine
//5 instrs x4 chars=20, 5 instr x6 chars=30 chars,
//Total: 10 instructions, 50 chars

LREM: &x, b
      &d, 0
LR4: =x,a, LR3
     +x,1
     +d,1
     =d, b, LR1
     =a,a,LR4
```

```
LR1: &d,0
     =a,a,LR4
LR3: =a,a,c

//MUL returns in d the product of a and b-to add here
//7 instructs, 4 of 4 chars= 16 chars;
//3 of 6 chars=18 chars, total 34 chars
LMUL: &d,0
    =b,0,c      //stop and return d
    &x,1
LM1:+d, a
   = x,b,c      //stop and return d
    +x,1
    =a,a LM1
//-------------------------------------------
// Explog(a+1,b); uses global variable ah as upper bound
// to test if there exists ah >= x > b+1 such that
// (x+1)^{xb} <= (a+1)x^{xb} < 4(x+1)^{xb}
// It uses: MUL, POW, CMP.
//41 instrs 4 chars=164, 6  instrs 6 chars= 36,
// 7 instrs of 2 chars=14 chars
//Total:  54 instructs,  214chars
&d,0
=ah,0,c    // return if no upper bound set
&ba,a      // store parameters locally
&bb,b
&bc,c
&bd,b      // start checking x at b+2
+bd,2
L1: &a,bd  // compute x+1 and x*b and store into be and bf
&be,bd
+be,1
&a,bd
&b,bb
&c,L2
#MUL
L2: &bf,d
&a,be      // compute bg = (x+1)^{bf}
&b,bf
&c,L3
#POW
L3: &bg,d
&a,bd      // compute x^{bf} and this times a+1, yielding bh
&c,L4
#POW
L4: &a,d
&b,ba
&c,L5
#MUL
```

```
L5: &bh,d
&a,bg      // test first <=
&b,bh
&c,L6
#CMP
L6: &a,L9
=d,2,a     // if false try next x
&a,4       // else compute 4*bg
&b,bg
&c,L7
#MUL
L7: &a,bh  // test second <
&b,d
&c,L8
#CMP
L8: &a,L10
=d,1,a     // test if found x
L9: +bd,1  // next x
&d,0
&a,L10
=bd,ah,a   // exit if reach upper bound
&d,L1
=a,a,d
L10: &a,ba // return, so restore initial values
&b,bb
&c,bc
=a,a,c
//=========================================

// Riemann uses: SUB, MOD, EXPLOG,MUL,CMP
//IT HAS 98 lines of instructions
//It has 78 instrs of 6 chars= 468 chars,
// 10 instr of 4 chars=40 chars
// 9 instrs of 2 chars =18 chars
//and 1 instr of 1 char,
// Total: 527 chars

&ca,599    // start of Riemann conjecture testing program
L1: +ca,1
&cb,600
L2: &ah,0
L3: &cc,0
L4: &cd,0
L5: &b,cb  // compute ce=ca-cb-ah-cc-cd (=ca-b)
+b,ah
+b,cc
+b,cd
&a,ca
&c,L6
```

```
#SUB    //-----------------use of SUB
L6: &ce,d
&cf,0      // check Cond 1
L9: +cf,1  // next y
=cf,cb,L7   // did we check all y<n (if yes, check Cond 2)
&a,cf
&b,cd
&c,L8
#MOD //-----------------use of MOD
L8:=d,0,L9    // divides when a mod b is zero
=a,a,L10
L7: &cf,1  // check Cond 2
LA2: +cf,1      // next y
=cf,cd,LA1   // did we check all y<m (if yes, check Cond 3)
&be,0
=cd,0,L10    // check m>0 (kludge!)
LA4: =be,n,L10
+be,1      // did we try all x < n?  (next m, if so)
&a,be
&b,cf
&c,LA3
#MOD
LA3: &c,LA4
=d,0,c     // divides so try next x
=a,a,LA2     // do next y
LA1: &a,cd // check Cond 3
&b,cc
&c,LA5
#EXPLOG
LA5: &c,L10
=d,0,c     // if fail do next m
&a,cb      // check Cond 4
&b,ce
&c,LA6
#EXPLOG
LA6: &c,L10
=d,0,c     // if fail do next m
&a,cb      // check Cond 5
&b,ce
&c,LA9
#MUL
LA9: &bg,d
+bg,bg     // bg=2nk
&a,cc
&b,cb
&c,LA7
#CMP
LA7: &c,LA8 // if l > n
=d,2,c
```

```
&a,cb       // else l <= n
&b,bg
+b,cc
&c,LB0
#CMP
LB0: &c,LX
=d,2,c     // last condition passes
&c,L10     // else try next m
LA8: &a,cc
&b,bg
+b,cb
&c,LB1
#CMP
LB1: &c,LX
=d,2,c      // 2nd way last condition passes
L10: &bg,cd // next m
+bg,cb
+bg,ah
+bg,cc
=bg,ca,L11   // check at end of m loop
+cd,1      // else increment
&d,L5
L11: &bg,cc // next l
+bg,cb
+bg,ah
=bg,ca,L12  // check at end of l loop
+cc,1
=a,a,L4
L12: &bg,ah // next b
+bg,cb
=bg,ca,L3  // check at end of b loop
+ah,1
=a,a,L3
L13: +cb,1
=cb,ca,L1 // next n and next s
=a,a,L2
LX: %
```

The register machine program for the Riemann hypothesis consists of 96 instructions for routines, 93 instructions in the main program giving a total of 1,087 characters (including 196 characters from labels), hence a total length of 4,348 bits.

6 Final comments

The upper bounds of complexities obtained in this paper for the Goldbach conjecture (1,068) and the Riemann hypothesis (4,348) improve the values 3,484 and 7,780 obtained in [3]; it also improves the result in [8] of 1,628 for the Goldbach conjecture.

The complexity of the Riemann hypothesis is close to the complexity of the four colour problem (4,336) [6]and is almost four times higher than the complexity of Fermat's last theorem (1,388) [5]. T It is still possible to improve the size of the programs for these statements. However, because the technique was used uniformly, it is unlikely that the complexity of the four colour theorem can be decreased to a value comparable with the complexity of the Fermat's last theorem. However, the relation between the complexities of the four colour theorem and the Riemann hypothesis is more delicate.

References

[1] F. E. Browder (ed.). *Mathematical Developments Arising from Hilbert Problems*, Amer. Math. Soc., Providence, RI, 1976.

[2] C. S. Calude. *Information and Randomness: An Algorithmic Perspective*, 2nd Edition, Revised and Extended, Springer-Verlag, Berlin, 2002.

[3] C. S. Calude, Elena Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.

[4] C. S. Calude, Elena Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, accepted. See also *CDMTCS Research Report* 353, 2008, 19 pp.

[5] C. S. Calude, Elena Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, accepted. See also *CDMTCS Research Report* 369, 2009, 14 pp.

[6] C. S. Calude, Elena Calude. The Complexity of the Four Colour Theorem, *CDMTCS Research Report* 368, 2009, 14 pp.

[7] G. J. Chaitin. *Algorithmic Information Theory*, Cambridge University Press, Cambridge, 1987. (third printing 1990)

[8] J. Hertel. On the Difficulty of Goldbach and Dyson Conjectures, *CDMTCS Research Report* 367, 2009, 15pp.

[9] D. Hilbert. Mathematical problems, *Bull. Amer. Math. Soc.* 8, 437–479, 1901–1902.

[10] Yu. V. Matiyasevich. *Hilbert's Tenth Problem*, MIT Press, Cambridge, MA, 1993.