# Evaluating the Complexity of Mathematical Problems: Part 2

**Cristian S. Calude**

*University of Auckland, New Zealand*
*www.cs.auckland.ac.nz/~cristian*

**Elena Calude**

*Massey University at Albany, New Zealand*
*www.massey.ac.nz/~ecalude*

In this paper we present an implementation of the computational method in [1] that allows ranking mathematical statements by their complexity. We introduce the complexity classes $(\mathfrak{C}_{U,i})_{i\geq 1}$, and, accordingly, show that Legendre's conjecture, Fermat's last theorem, and Goldbach's conjecture are in $\mathfrak{C}_{U,1}$, Dyson's conjecture is in $\mathfrak{C}_{U,2}$, the Riemann hypothesis is in $\mathfrak{C}_{U,3}$, and the four color theorem is in $\mathfrak{C}_{U,4}$.

## 1. Introduction

Based on the possibility of expressing mathematical problems in terms of (very) simple programs reducible to the halting problem in [1, 2], we developed a uniform approach for evaluating the complexity of a large class of mathematical problems. In this paper we: (*a*) describe an implementation of the method, (*b*) introduce the complexity classes $(\mathfrak{C}_{U,i})_{i\geq 1}$, and (*c*) rank according to (*b*) these six mathematical statements: Goldbach's conjecture, Legendre's conjecture, Fermat's last theorem, Dyson's conjecture, the four color theorem, and the Riemann hypothesis. To this aim we describe a universal programming language that is a prefix-free Turing machine and a uniform method for evaluating the size (measured in bits) of the programs written in the language. For each of the six statements, we write the shortest possible program to systematically search for a counter-example. The programs never stop if and only if the statements are true. The ranking of a statement in a class $\mathfrak{C}_{U,i}$ is based on the size of its associated program.

The programs for Goldbach's conjecture and the Riemann hypothesis given here improve (in size) those in [2] and appear in [3]. The program for the four color theorem is in [4]. The other three programs appear here for the first time.

https://doi.org/10.25088/ComplexSystems.18.4.387

The paper is structured as follows. In Section 2 we introduce a universal programming language. In Section 3 we present the implementation of the method introduced in [1, 2] and the complexity classes $\left(\mathfrak{C}_{U,i}\right)_{i\geq 1}$. In Section 4 we present algorithms for some routines frequently used in the programs. In Sections 5 through 7 we discuss Legendre's conjecture, Fermat's last theorem, Dyson's conjecture, Goldbach's conjecture, the four color theorem, and the Riemann hypothesis. Section 8 presents some conclusions.

## 2. A Universal Programming Language

We briefly describe the syntax and semantics of a register machine language that implements a (natural) universal prefix-free binary Turing machine $U$. The language is a refinement of those described in [2, 5, 6].

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer. By default, all registers, named with a string of lower or uppercase letters, are initialized to 0. Instructions are labeled by default with 0, 1, 2, … .

Here is a list of the register machine instructions. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register. When referring to R we use, depending upon the context, either the name of register R or the non-negative integer stored in R.

- =R1, R2, R3: If the contents of R1 and R2 are equal, then the execution continues at the R3th instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence. There is an illegal branch error if the content of R3 is outside the scope of the program.
- &R1, R2: The content of register R1 is replaced by R2.
- +R1, R2: The content of register R1 is replaced by the sum of the contents of R1 and R2.
- !R1: One bit is read into the register R1, so the content of R1 becomes either 0 or 1. Any attempt to read past the last data bit results in a run-time error.
- %: This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: successful completion or an under-read error.

A *register machine program* consists of a finite list of labeled instructions from the complete 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 that does not read all of the data or attempts to read past the last data bit results in a runtime error. Some programs (such as the ones presented in this paper) have no input data, so they cannot halt with an under-read error.

The instruction =R, R, n is used for the unconditional jump to the $n^{\text{th}}$ instruction of the program. For Boolean data types we use integers $0 =$ false and $1 =$ true.

For longer programs it is convenient to distinguish between the main program and some sets of instructions called "routines", which perform specific tasks within another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

## 3. Complexity

We present a method of evaluating the complexity of a $\Pi_1$-problem $\pi$, that is, a statement of the form $\pi = \forall\, \sigma P(\sigma)$ where $P$ is a computable predicate. To every $\Pi_1$-problem $\pi = \forall\, \sigma P(\sigma)$, we associate a program $\Pi_P = \inf\{n : P(n) = \text{ false}\}$ to search for a possible counter-example to $\pi$. The following equivalence holds true: $\pi$ is true if and only if $U(\Pi_P)$ never halts.

The complexity (with respect to $U$) of a $\Pi_1$-problem $\pi$ is defined by

$$C_U(\pi) = \inf\{|\Pi_P| : \pi = \forall\, nP(n)\}.$$

The choice of $U$ is not important because if $U$, $U'$ are universal, then there exists a constant $c = c_{U,U'}$ such that for every $\Pi_1$-problem $\pi$, $\left|C_U(\pi) - C_{U'}(\pi)\right| \leq c$. The "bad news" is that the complexity $C_U$ is not computable [7].

At first glance, the complexity $C_U$ may appear to separate the set of $\Pi_1$-problems into only two classes. However, this is false because $C_U$ is unbounded. Because of incomputability, we can work only with upper bounds of $C_U$. As the exact value of $C_U$ is not important, we classify $\Pi_1$-problems into the following classes:

$$\mathfrak{C}_{U,n} = \left\{\pi : \pi \text{ is a } \Pi_1 \text{ -problem}, \; C_U(\pi) \leq n \,\text{kbit}\right\}.$$

(A kilobit [kbit or kb] is equal to $2^{10}$ bits.) It is seen that for every $n \geq 1$ there is an $m > n$ such that $\mathfrak{C}_{U,n}$ is strictly included in $\mathfrak{C}_{U,m}$. We do not know whether $m$ can always be taken to be $n+1$, that is, if we have a strict hierarchy.

The goal is to compute an upper bound of the complexity $C_U(\pi)$ by choosing a representation $\pi = \forall\, nP(n)$ for which $|\Pi_P|$ is the smallest possible; hence $|\Pi_P|$ is the best possible upper bound for $C_U(\pi)$. The running time efficiency of the program $\Pi_P$ is irrelevant here; it is the size in bits that counts. (See more details and comments in [1].)

To compute an upper bound on $C_U(\pi)$ we need to compute the size in bits of the program $\Pi_P$, so we need to uniquely code in binary the programs for $U$. To this aim, we use the following prefix-free coding.

The binary coding of special characters (instructions and comma) is given in Table 1 ($\varepsilon$ is the empty string).

| Special Characters | Code | Instruction | Code |
|---|---|---|---|
| , | $\varepsilon$ | + | 111 |
| & | 01 | ! | 110 |
| = | 00 | % | 100 |

**Table 1.**

For registers, we use the prefix-free code $\text{code}_1 = \{0^{|x|}\,1\,x \mid x \in \{0, 1\}^*\}$. Table 2 gives the codes of the first 15 registers. The register names are chosen to optimize the length of the program, that is, the most frequent registers have the smallest $\text{code}_1$ length.

| Register | $\text{code}_1$ | Register | $\text{code}_1$ | Register | $\text{code}_1$ |
|---|---|---|---|---|---|
| $R_1$ | 010 | $R_6$ | 00111 | $R_{11}$ | 0001100 |
| $R_2$ | 011 | $R_7$ | 0001000 | $R_{12}$ | 0001101 |
| $R_3$ | 00100 | $R_8$ | 0001001 | $R_{13}$ | 0001110 |
| $R_4$ | 00101 | $R_9$ | 0001010 | $R_{14}$ | 0001111 |
| $R_5$ | 00110 | $R_{10}$ | 0001011 | $R_{15}$ | 000010000 |

**Table 2.**

For non-negative integers, we use the prefix-free code $\text{code}_2 = \{1^{|x|}\,0\,x \mid x \in \{0, 1\}^*\}$. Table 3 gives the codes of the first 16 non-negative integers.

| Integer | $\text{code}_2$ | Integer | $\text{code}_2$ | Integer | $\text{code}_2$ | Integer | $\text{code}_2$ |
|---|---|---|---|---|---|---|---|
| 0 | 100 | 4 | 11010 | 8 | 1110010 | 12 | 1110110 |
| 1 | 101 | 5 | 11011 | 9 | 1110011 | 13 | 1110111 |
| 2 | 11000 | 6 | 1110000 | 10 | 1110100 | 14 | 111100000 |
| 3 | 11001 | 7 | 1110001 | 11 | 1110101 | 15 | 111100001 |

**Table 3.**

The instructions are coded by self-delimiting binary strings as follows. (Because $x\varepsilon = \varepsilon x = x$, for every string $x \in \{0, 1\}^*$, we omit the $\varepsilon$.)

1. & R1, R2 is coded in two different ways depending on R2:

   01 $\text{code}_1$(R1) $\text{code}_i$(R2),

   where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

2. + R1, R2 is coded in two different ways depending on R2:

   111 $\text{code}_1$ (R1) $\text{code}_i$ (R2),

   where $i = 1$ if R2 is a register and $i = 2$ if R2 is a non-negative integer.

3. = R1, R2, R3 is coded in four different ways depending on the data types of R2 and R3:

   00 $\text{code}_1$ (R1) $\text{code}_i$ (R2) $\text{code}_j$ (R3),

   where $i = 1$ if R2 is a register and $i = 2$ if R2 is a non-negative integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is a non-negative integer.

4. ! R1 is coded by

   110 $\text{code}_1$ (R1).

5. % is coded by

   100.

All codings for instruction names, registers, and non-negative integers are self-delimiting. The prefix-free codes used for registers and non-negative integers are disjoint. The code of any instruction is the concatenation of the codes of the instruction name and the codes (in order) of its components, hence the set of codes of instructions is prefix-free. The code of a program is the concatenation of the codes of its instructions, so the set of codes of all programs is also prefix-free.

Table 4 gives some examples of instructions.

| Instruction | Code | Length |
|---|---|---|
| % | 100 | 3 |
| & $R_1$, 0 | 01 010 100 | 8 |
| & $R_1$, $R_2$ | 01 010 011 | 8 |
| + $R_1$, 1 | 111 010 101 | 9 |
| + $R_1$, $R_2$ | 111 010 011 | 9 |
| = $R_1$, 0, 1 | 00 010 100 101 | 11 |
| = $R_1$, $R_2$, 0 | 00 010 011 100 | 11 |

**Table 4.**

The shortest programs are

100 | 01010100100 | 01010011100.

The smallest program that halts is 100 and the smallest program that never halts is 00010010100100.

Table 5 gives a register machine routine that computes in *d* the product of two non-negative integers *a* and *b* (see the algorithm `MUL` in Section 4). We use: $R_1 = a$, $R_2 = b$, $R_3 = c$, $R_4 = d$, $R_5 = e$, $R_8 = h$.

| Instruction Number | Instruction | Code | Length |
|---|---|---|---|
| 0 | & h, e | 01 0001001 00110 | 14 |
| 1 | & d, 0 | 01 00101 100 | 10 |
| 2 | = b, 0, 8 | 00 011 100 1110010 | 15 |
| 3 | & e, 1 | 01 00110 101 | 10 |
| 4 | + d, a | 111 00101 010 | 11 |
| 5 | = b, e, 8 | 00 011 00110 1110010 | 17 |
| 6 | + e, 1 | 111 00110 101 | 11 |
| 7 | = a, a, 4 | 00 010 010 11010 | 13 |
| 8 | & e, h | 01 00110 0001001 | 14 |
| 9 | = a, a, c | 00 010 010 00100 | 13 |

**Table 5.**

The routine given in Table 5 can be uniquely encoded by concatenating the binary strings coding its instructions,

010001001001100100101100000111001110010010011010 1⋱
1110010101000011001101110010111001101010001001 0⋱
110100100110000100100010010 00100,

which is a string with a size of 128 bits.

## 4. Algorithms

Some register machine programs may be difficult to follow because of their terse syntax. In order to facilitate understanding, we sometimes present parts of them as algorithms in pseudocode. The notation used in these algorithms is self-explanatory (e.g., the assignment instruction is denoted by `Set x to v`, `Next x` is the successor, and `GoTo Ln` specifies the unconditional jump).

We start with a simple routine `REM` that computes the integer remainder of *a* divided by *b*. A local register *e* is initialized to *b* and incremented by 1 until it reaches the value of *a* when the algorithm finishes. The value of *d*, the result of the algorithm, is initialized to 0 and incremented every time *e* is incremented. When *d* reaches the value of *b*, the value of *d* is reset to 0. The routine works for any non-negative integers *a* and *b*.

```
Algorithm REM
INPUT: a >= b >= 0
OUTPUT: d=rem(a,b) i.e., a=b*q+d, with 0 <= d < b, for some q
1. Set e to b
2. Set d to 0
3. if e = a
4.   then STOP
5.   else Next e
6.        Next d
7.        if d = b
8.             then GoTo 2 //reset the remainder to 0
9.             else GoTo 3
```

Here is the register machine program corresponding to REM.

```
//REM computes in d the integer remainder
// of a divided by b, assumes a>=b>=0.
//It uses the local register e to perform its task
0. &h,e //store locally the original value of e
1. &e,b //copy the value of b in e
2. &d,0 //set result to 0
3. =e,a,8 //e reached a, continue with instruction 8
4. +e,1 //as e < a, increase e
5. +d,1 //increase the result
6. =d,b,2 //result reached b, continue with instruction 2
7. =a,a,3 //continue with instruction 3
8. &e,h //restore original value in e
9. =a,a,c //computation completed, registers a, b, c, and
        //e have their original values and d contains
        //the integer remainder of a divided by b
```

We continue with the algorithms MUL and CMP for routines that are used repeatedly. MUL performs the multiplication of *a* and *b* and stores the product in *d*. The algorithm is based on the multiplication performed as a repeated addition. The local counter *e* keeps track of how many times *a* is added to itself.

```
Algorithm MUL
INPUT: a >= 0, b >= 0
OUTPUT: d= a*b
1. Set d to 0
2. if b = 0
3.   then STOP
4.   else Set e to 1
5.        Set d to d+a
6.        if e = b
7.             then STOP
```

```
8.          else Next e
9.          GoTo 5
```

The register machine program and its code for the multiplication algorithm appear in Table 5. CMP returns 0 if its two input values $a$ and $b$ are equal, returns 1 when $a < b$, and returns 2 when $b < a$.

```
Algorithm CMP
INPUT: a >= 0, b >= 0
OUTPUT: d is 1 if a < b, d is 0 for a = b, and d is 2 otherwise
1. Set e to a
2. Set f to b
3. Next e
4. Next f
5. Set d to 0
6. if e = f
7.   then STOP
8.   else Set d to 1
9.       if e = b
10.          then STOP
11.          else Set d to 2
12.              if f = a
13.                  then STOP
14.                  else GoTo 3
```

## 5. Legendre's Conjecture

Legendre's conjecture [8] states that for any natural number $n$ there exists a prime number $p$ such that $n^2 \leq p \leq (n+1)^2$. The following algorithm checks whether for each natural number $n$ any of the numbers $n^2+1, \ldots, (n+1)^2-1$ is prime. If a prime is found, the algorithm generates the next $n$ and so on. If for some natural $n$, none of the numbers from the given set is prime, the algorithm stops and the conjecture is false; otherwise, the algorithm never stops. Here is the register machine program for Legendre's conjecture.

```
0. &n,2
1. &m,n
2. &p,1
3. =p,n,7 //m=n^2
4. +m,n
5. +p,1
6. =p,p,3
7. &M,m
8. +M,n
9. +M,n //M=n^2+2n
10. &x,m
11. =x,M,31 //no prime x was found
```

```
12. &p, 2 //is x divided by p?
13. &z,1 //z =1 if x is prime, z=0 if p is not prime
14. =x,p,26 //x is prime
15. &e,p
16. &q,0 //compute q=rem(x,p)
17. =e,x,22
18. +e,1
19. +q,1
20. =q,p,16
21. =p,p,17
22. =q,0,25 //x is not prime
23. +p,1
24. =p,p,13
25. &z,0
26. =z,0,29 //x is not prime
27. +n,1 //x is prime
28. =p,p,1
29. +x,1
30. =p,p,11
31. % //Legendre's conjecture is false
```

This register machine program for Legendre's conjecture has 32 instructions. Computing its size, we get $C_U$(Legendre's conjecture) $\leq 416$ when using $R_1 = p$, $R_2 = n$, $R_3 = x$, $R_4 = m$, $R_5 = q$, $R_6 = M$, $R_7 = z$, $R_8 = e$.

## 6. Fermat's Last Theorem

Fermat's last theorem is one of the most famous theorems in the history of mathematics. It states that there are no positive integers $x, y, z$ satisfying the equation $x^n + y^n = z^n$, for any integer value $n > 2$. The result was conjectured by Pierre de Fermat in 1637, and was finally proved in 1995 by A. Wiles [9] (see also [10]). Many illustrious mathematicians failed to prove it, but their efforts stimulated the development of algebraic number theory.

The following register machine program for Fermat's last theorem uses the integer $B \geq 5$ to enumerate all 4-tuples of integers $(x, y, z, n)$ with $z \leq B, x, y < z, n \leq B$ for which the equality $x^n + y^n = z^n$ is tested.

```
0. =a,a,20
1. &i,x //===POW(a,b)
2. &j,y
3. &k,z
4. &x,1
5. &d,a
6. =x,b,16 //d = a^b
```

```
7. &z,a //compute a*d
8. &y,1
9. =y,d,13 //z = a*d
10. +y,1 //y < d
11. +z,a
12. =a,a,9
13. &d,z
14. +x,1 //x < b
15. =a,a,6
16. &x,i
17. &y,j
18. &z,k
19. =a,a,c //d = a^b
20. &B,5 //===Main program
21. &n,4
22. &z,4
23. &x,3
24. &y,3
25. &c,29
26. &a,x
27. &b,n
28. =a,a,1 //d = x^n
29. &e,d
30. &c,33
31. &a,y
32. =a,a,1 //d = y^n
33. +e,d //e = x^n + y^n
34. &a,z
35. +c,4 //c = 37
36. =a,a,1 //d = z^n
37. =e,d,52 //x^n + y^n = z^n
38. +y,1 //x^n + y^n =/= z^n
39. =y,z,41
40. =a,a,25 //y < z
41. +x,1 //y = z
42. =x,z,44
43. =a,a,24 //x < z
44. +z,1 //x = z
45. =B,z,47
46. =a,a,23 //z < B
47. +n,1 //z = B
48. =n,B,50
49. =a,a,22 //n < B
50. +B,1 //n = B
51. =a,a,21
52. % //Fermat's last theorem is false
```

This register machine program for Fermat's last theorem has 53 instructions. Computing its size, we get $C_U$(Fermat's last theorem) $\leq 738$ when using $R_1 = a$, $R_2 = z$, $R_3 = x$, $R_4 = y$, $R_5 = d$, $R_6 = c$, $R_7 = B$, $R_8 = n$, $R_9 = e$, $R_{10} = b$, $R_{11} = i$, $R_{12} = j$, $R_{13} = k$.

## 7. Dyson and Goldbach Conjectures, the Four Color Theorem, and Riemann's Hypothesis

Dyson's first conjecture [11] states that

the reverse of a power of two is never a power of five

and is motivated by the quest to find a simple true unprovable statement in Gödel's sense. In [11], p. 86, Dyson states:

> Thanks to Kurt Gödel, we know that there are true mathematical statements that cannot be proved. But I want a little more than this. I want a statement that is true, unprovable, and simple enough to be understood by people who are not mathematicians.

Dyson's second conjecture [11] states that

Dyson's first conjecture is unprovable.

(To be precise, we must specify the formal system in which Dyson's first conjecture is unprovable. A natural candidate is Peano arithmetic.) Here is the heuristic argument in support of Dyson's second conjecture [11]:

> The digits in a big power of two seem to occur in a random way without any regular pattern. If it ever happened that the reverse of a power of two was a power of five, this would be an unlikely accident, and the chance of it happening grows rapidly smaller as the numbers grow bigger. If we assume that the digits occur at random, then the chance of the accident happening for any power of two greater than a billion is less than one in a billion. It is easy to check that it does not happen for powers of two smaller than a billion.

In fact, this conjecture was verified in [12] up to all powers $2^k$ with $k \leq 10^5$ and in [13] up to all powers $2^k$ with $k \leq 10^8$.

Of course, if Dyson's first conjecture is false, that is, a counterexample is found, then Dyson's second conjecture is also false.

In [13] it was shown that the complexity of Dyson's first conjecture, shortly, Dyson's conjecture, has an upper bound of 3928 bits (150 register machine instructions). Here is a shorter program written for *U*.

```
0. =a,a,27
1. & E,e //===CMP(a,b)
2. &F,f
3. &e,a
4. &f,b
5. +e,1
6. +f,1
7. &d,0
8. =e,f,14 //a = b
9. &d,1
10. =e,b,14 //a < b
11. &d,2
12. =f,a,14 //a > b
13. =a,a,5
14. &f,F
15. &e, E
16. =a,a,c
17. & E,e //===MUL(a,b)
18. &d,0
19. =b,0,25 //ab = 0
20. &e,1
21. +d,a
22. =e,b,25 //d = ab
23. +e,1
24. =a,a,21
25. &e, E
26. =a,a,c
27. &k,1 //===MAIN PROGRAM
28. &n,1
29. +n,n
30. &c,34 // compute f = reverse of n
31. &a,n
32. &b,10
33. =a,a,1 //d = CMP(n,10)
34. =d,1,58 //n < 10
35. &f,0 //n >= 10
36. &e,b
37. &q,0
38. +q,1
39. &r,0
40. =e,n,45 //r = n mod 10, q = floor(n/10)
41. +e,1 //e < n
42. +r,1
43. =r,b,38
44. =a,a,40 //r < b
45. +f,r
46. &a,f
47. &c,49
```

```
48. =a,a,17 //d = (f+r)*10
49. +f,d
50. &a,q
51. +c,4 //c = 53
52. =a,a,1 //d = CMP(q,10)
53. =d,1,56 //q < 10
54. +f,q //q >= 10
55. =a,a,59
56. &n,q
57. =a,a,36
58. &f,n //reverse of n = n
59. &s,1
60. &j,0
61. +j,1
62. &c,66
63. &a,s
64. &b,5
65. =a,a,17 //d = MUL(5^(j-1),5)
66. &s,d
67. +c,5 //c = 71
68. &a,s //a = 5^j
69. &b,f //b = reverse(2^k)
70. =a,a,1 //d = CMP(s,f)
71. =d,1,61 //s < f
72. =d,0,75 //s = f
73. +k,1 //s > f
74. =a,a,29
75. % //Dyson's conjecture is false
```

This register machine program for Dyson's conjecture has 76 instructions. Computing its size, we get $\mathfrak{C}_U$(Dyson's conjecture) $\leq$ 1067 when using $R_1 = a$, $R_2 = e$, $R_3 = f$, $R_4 = d$, $R_5 = b$, $R_6 = c$, $R_7 = n$, $R_8 = q$, $R_9 = E$, $R_{10} = r$, $R_{11} = s$, $R_{12} = F$, $R_{13} = k$, $R_{14} = j$.

## 8. Final Comments

We have calculated the upper bounds on the $\mathfrak{C}_U$ complexity of these six mathematical statements: Goldbach's conjecture 756, Legendre's conjecture 416, Fermat's last theorem 738, Dyson's conjecture 1067, the Riemann hypothesis 2741, and the four color theorem 3289. Accordingly, Legendre's conjecture, Fermat's last theorem and Goldbach's conjecture are in $\mathfrak{C}_{U,1}$, Dyson's conjecture is in $\mathfrak{C}_{U,2}$, the Riemann hypothesis is in $\mathfrak{C}_{U,3}$, and the four color theorem is in $\mathfrak{C}_{U,4}$.

It is still possible to improve the size of the programs for these statements or to use a different implementation of the method. We conjecture that, with the possible exception of the four color theorem, our ranking of the six mathematical statements cannot be improved. It is open whether for every $i \geq 1$, $\mathsf{C}_{U,i} \subset \mathsf{C}_{U,i+1}$.

Finally, the halting problem can be expressed in Peano arithmetic (PA), so reducing a problem to an instance of the halting problem shows the possibility of expressing that problem in PA. In some cases this was evident without any reducibility; in others, like the Riemann hypothesis, this was not so clear. In all cases it is interesting to look for solutions of the problem in PA (see [14] for a discussion of Fermat's last theorem).

## Acknowledgment

We thank M. Dinneen, J. Hertel, and N. Kasto for comments, suggestions, and extensive discussions which improved this paper. We thank also the anonymous referee for critical comments and useful suggestions.

This work was supported in part by The Andrea von Braun Foundation, Munich, under the grant for "Artistic Forms and Complexity".

## References

[1] C. S. Calude and E. Calude, "Evaluating the Complexity of Mathematical Problems: Part 1," *Complex Systems*, **18**(3), 2009 pp. 267-285. See also *CDMTCS Research Report* **353**, 2009, 19 pp.

[2] C. S. Calude, E. Calude, and M. J. Dinneen, "A New Measure of the Difficulty of Problems," *Journal for Multiple-Valued Logic and Soft Computing*, **12**(3-4), 2006 pp. 285-307.

[3] E. Calude, "The Complexity of the Goldbach's Conjecture and Riemann's Hypothesis," *CDMTCS Research Report* **369**, 2009, pp. 14.

[4] C. S. Calude and E. Calude, "The Complexity of the Four Colour Theorem," *CDMTCS Research Report* **368**, 2009, pp. 14.

[5] G. J. Chaitin, *Algorithmic Information Theory*, Cambridge: Cambridge University Press, 1987.

[6] C. S. Calude, M. J. Dinneen, and C.-K. Shu, "Computing a Glimpse of Randomness," *Experimental Mathematics*, **11**(2), 2002 pp. 369-378.

[7] C. S. Calude, *Information and Randomness: An Algorithmic Perspective*, 2nd ed., Berlin: Springer, 2002.

[8] E. W. Weisstein. "Legendre's Conjecture" from Wolfram *MathWorld*—A Wolfram Web Resource.
mathworld.wolfram.com/LegendresConjecture.html.

[9] A. Wiles, "Modular Elliptic Curves and Fermat's Last Theorem," *Annals of Mathematics*, **141**(3), 1995 pp. 443-551.

[10] A. Aczel, *Fermat's Last Theorem: Unlocking the Secret of an Ancient Mathematical Problem*, New York: Four Walls Eight Windows, 1996.

[11] F. Dyson, *What We Believe But Cannot Prove: Today's Leading Thinkers on Science in the Age of Certainty*, (J. Brockman, ed.), London: Pocket Books, 2006 pp. 85-86. See also www.edge.org/q2005/q05_9.html#dysonf.

[12] C. S. Calude. "Dyson Statements Are Likely to be True But Unprovable." (12 Jun 2008) www.cs.auckland.ac.nz/~cristian/fdyson.pdf.

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

[14] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.