CDMTCS
Research
Report
Series

Inductive Complexity of the P Versus NP Problem

Cristian S. Calude[1], Elena Calude[2],
Melissa S. Queen[3]

[1]University of Auckland, NZ
[2]Massey University, Albany, Auckland, NZ
[3]Dartmouth College, USA

CDMTCS-429
October 2012

Centre for Discrete Mathematics and
Theoretical Computer Science

INDUCTIVE COMPLEXITY OF THE P VERSUS NP PROBLEM*

CRISTIAN S. CALUDE†
Department of Computer Science, University of Auckland, Auckland, and New Zealand
Isaac Newton Institute for Mathematical Sciences, Cambridge, United Kingdom
`cristian@cs.auckland.ac.nz`

ELENA CALUDE
Institute of Natural and Mathematical Sciences, Massey University at Auckland, New Zealand
`e.calude@massey.ac.nz`

and

MELISSA S. QUEEN‡
Department of Computer Science, University of Auckland, Auckland, New Zealand
Dartmouth College, New Hampshire, USA
`melissa.s.queen.13@dartmouth.edu`

ABSTRACT

This paper does not propose a solution, not even a new possible attack, to the P versus NP problem. We are asking the simpler question: How "complex" is the P versus NP problem? Using the inductive complexity measure—a measure based on computations run by inductive register machines of various orders—developed in [2], we determine an upper bound on the inductive complexity of second order of the P versus NP problem. From this point of view, the P versus NP problem is significantly more complex than the Riemann hypothesis. To date, the P versus NP problem and the Goodstein theorem (which is unprovable in Peano Arithmetic) are the most complex mathematical statements (theorems, conjectures and problems) studied in this framework [8, 4, 5, 2, 19].

1. A class of complexity measures

Mathematics is built upon theorems, conjectures and problems both open and resolved. Some problems intuitively seem highly complex, and have perhaps eluded solution for centuries. Others appear to be less complicated. We would like to be able to quantitatively capture this complexity, and thus be able to compare conjectures from vastly different fields of mathematics. One possible scale we can use has been

---

*An extended abstract of this paper has appeared in J. Durand-Lose, N. Jonoska (eds.). *Proceedings UCNC 2012*, LNCS 7445, Springer, 2012, 2–9.

†Partially supported by a Visiting Fellowship, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 2012.

‡Partially supported by a University of Auckland International Summer Scholarship 2012.

developed in [8, 4, 5, 2, 7] and applied to different problems in [6, 9, 10, 12, 16]. This method considers the most intuitive way to solve a problem, a brute-force search for a counter-example to the claim. If the conjecture is false, a counter-example will eventually be found. But if a conjecture is true, the search will run on forever. If we could somehow determine ahead of time if the search will run forever, we would be able to prove the conjecture is true. Unfortunately, this equates to solving the halting problem, which is known to be undecidable. But not all is lost, since we are not actually trying to *solve* all mathematical conjectures, but rather to *compare* some of them: indeed, we wish to be able to compare conjectures regardless of their true/false or proven/unproven status.

For this aim we will use a more powerful model of computation than the Turing machine, the inductive computation. The search for a counter-example can be coded into a program, and the program can carefully be encoded into a string of ones and zeroes. Thus for any mathematical conjecture, we can create a string of bits (along with an explanation of how to unambiguously read off the program) and say 'if this program halts, the conjecture is false; if it does not halt, the conjecture is true'. It naturally follows that some conjectures can be 'encoded' into bits more simply than others; these conjectures will be considered of low complexity. More complicated conjectures may take a large program and a huge number of bits; these programs are considered to have high complexity. Time complexity plays no role in this analysis.

Although the results we obtain may shed new light on the statements we analyse, they are not intended to solve the problems expressed by those statements, nor to predict how easy/difficult could be to find their solutions.

2. The P versus NP problem

The processing power of computers has grown—and continues to grow—incredibly quickly, and computer-users have become accustomed to newer and faster computers continually being released on the market. In such an environment, it might seem like there is no bound to the size and type of problems that computers can solve—and even if a program runs slowly on today's computers, surely in a few years it will be zipping along on the faster computers of the future. Unfortunately, this is not the case. The problem lies in the asymptotic behaviour of certain algorithms, i.e. their behaviour when the problem instance size gets very large. It makes sense that the larger a problem input size, the longer it takes to solve it, but in some cases the needed time grows faster than we will ever be able to account for with faster computers. The usual solution is to simply find a faster, more efficient algorithm. But for a large class of problems, many of them of critical practical significance, *no* efficient algorithms have been found. This class is called NP (nondeterministic polynomial), while the class of problems that are known to have efficient algorithms is called P (deterministic polynomial, or, simply, polynomial).

Furthermore, there exists a set of NP problems, called NP–complete, which if one could figure out how to solve just one of them in polynomial-time then we could solve *all* of them in polynomial-time. In asking the question 'Does P=NP?' we are asking if it is possible to solve all NP problems in polynomial-time, or equivalently, if it is possible to solve an NP–complete problem in polynomial-time.

As a concrete example, we present the NP–complete problem used in our program: the subset-sum problem [15] (subsection 35.5: The subset-sum problem). This problem starts with a collection of numbers, and a target number—an instance of the problem—and asks the question: *Does some subset of our collection add up to equal the target?* In small instance sizes this is simple. For example, we can easily check that no subset of (1,2,5) adds up to 4, or that there is a subset of (1,2,5,8) that adds to 7 (namely, 2 and 5). But as the instance size gets larger, the number of possible subsets grows exponentially, and it takes exponential time to check every subset.

The brute-force algorithm for solving the subset-sum problem cycles through all subsets of N numbers and, for every one of them, checks if the subset sums to the right number. The running-time is of order $\mathrm{O}(N \cdot 2^N)$, since there are 2^N subsets and, to check each subset, we need to sum at most N elements. A faster algorithm proposed by Horowitz and Sahni [20] runs in time $\mathrm{O}(2^{N/2})$. If one could show that there is some algorithm that solves every possible instance of subset-sum in polynomial-time, then we would show that P=NP.

The P versus NP problem, formulated independently by Cook [13] and Levin [23], is considered to be one of the most challenging open problems in mathematics. The Clay Mathematics Institute will award a prize of $1.000.000 for its first correct solution, [27]. A constructive proof for $\mathrm{P} = \mathrm{NP}$ based on an efficient simulation would have significant practical consequences; a proof for $\mathrm{P} \neq \mathrm{NP}$ (which is widely believed to be the case) would lack practical computational benefits, but would have important theoretical value.

With decades of research dedicated to its resolution, substantial insight was obtained: see more in the official Clay Mathematics Institute presentation of the problem by Cook [14], the papers by Fortnow [17] and Mulmuley [25], Moore-Mertens book [24] (Chapter 6, The Deep Question: P vs NP), and Wöginger's webpage [29].

The use of parallel computers instead of sequential computers does not help because if a problem requires exponential time to be solved on a sequential computer, every parallel computer with finite number of processors solving it runs also in exponential time since the speed-up can be only by a constant factor (e.g. the number of processors). The above result is not "absolute", i.e. it depends on the geometry of the space where the computation is run. More precisely, the above result is true in case the real time-space is Euclidean, and the reason is that the volume of a sphere of radius r is a polynomial in r, $\mathrm{O}(r^3)$. If the real time-space is hyperbolic, the volume of the sphere grows exponentially with r, and this growth can be exploited to dramatically speed-up parallel computations, hence the possibility to solve NP–hard problems in polynomial time. For more details see [22].

Is the polynomial-time algorithm the "correct" mathematical model for feasible computation? An affirmative answer is provided by Cobham's thesis which states that "P" means *easy* while "the complement of P" means *hard*. Is Cobham's thesis as "credible" as the Church-Turing thesis, which deals with computability in principle, i.e. by disregarding resources? According to Davis [21] (p. 568–569) the answer is negative[a]. In fact, we believe that the concept of feasible computation like the concept of randomness are paradoxical (blind spots in the terminology of [3]): they cannot be grasped in finite words. More precisely, we conjecture that there is no good mathematical model for feasible computation. From this perspective, *the P versus NP problem is less a computer science problem than a mathematical one.*

3. Goal

By measuring the complexity of the P versus NP problem we hope to shine a little more light on the problem; certainly, this is not an attempt to solved it. To do this, we have developed an inductive register machine program that searches for a counter-example to the claim that "P $\neq$ NP". This counter-example would be a program that runs in polynomial-time for all instances of the subset-sum problem, our choice of NP–complete problem. The register machine program has a prefix-free binary encoding, and the length of this string determines an upper bound of the complexity class of the P versus NP problem.

4. Method

The register machine language we use is a refinement, constructed in [5], of the language in [8]; see also [16]. The register machine language is simple and minimal (each instruction is essential; no instruction can be reduced to a combination of the other instructions). It consists of the following instructions:

= R1,R2,R3 If the content of R1 and R2 are equal, then the execution continues at the R3^{rd} instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence.

& 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

---

[a] In the discussions following J. Hartmanis' invited lecture *Turing Machine Inspired Computer Science Results*, CiE2012, 22 June 2012, `http://www.mathcomp.leeds.ac.uk/turing2012/WScie12/Content/abstracts/juris.html`, M. Davis asked the question he posed the speaker about 30 years ago: "How would you feel if P=NP with a polynomial of degree 100?" Hartmanis' original answer was: "God cannot be so cruel!"

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: either successfully halts or it halts with an under-read error.

A register machine program is a finite list of these instructions. It is allowed access to an arbitrary number of registers, and each register can hold an arbitrarily large positive integer. The prefix free binary encoding of these instructions is discussed in detail in [4, 5], and briefly below.

Each instruction has its own binary op-code, registers names are encoded as the string $\mathrm{code}_1 = 0^{|x|}1x,\ x \in \{0,1\}^*$ and literals are encoded $\mathrm{code}_2 = 1^{|x|}0x,\ x \in \{0,1\}^*$. Some instructions can take registers or literals, but this encoding gives an unambiguous distinction between the two options. The encodings are summarised below:

(1) `& R1,R2` is coded in two different ways depending on R2: $01\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2})$, 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: $111\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2})$, where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(3) `= R1,R2,R3` is coded in four different ways depending on the data types of R2 and R3: $00\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2})\mathrm{code}_j(\mathrm{R3})$, where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.

(4) `!R1` is coded by $110\mathrm{code}_1(\mathrm{R1})$.

(5) `%` is coded by 100.

Programs often need to execute the same operations many different times, and it is convenient to create routines for these operations. Routines that our program uses include MUL (multiply), POW (power/exponentiation), CMP (compare), SUBT (subtraction) and DIV2 (halves a number).

As a concrete example, the subtraction routine is given in Table 1. The routine uses registers `a` through `e` (named SUBT), computes `a` − `b`, puts the answer in `d` then returns to the line number stored in `c`. It assumes that `a` ≥ `b`.

The register names, `a` = 010, `b` = 00100, `c` = 00101, `d` = 00111, and `e` = 011, were chosen to minimise the overall number of bits used. In total, this routine is represented by the 69 bit string:

010011110001011001111000110010010101101000101010011101110101001011010

5. Register machine language implementation of arrays

We use the coding for the array data structure library developed by Dinneen [16] which represents arrays (lists) in a single register variable. An integer element a_i

Label	Instruction	Comments	Binary representation
SUBT1	`& d, 0`		`01 00111 100`
LS1	`& e, d`		`01 011 00111`
	`+ e, b`		`100 011 00100`
	`= e, a, c`	`// d+b=a`	`101 011 010 00101`
	`& d, 1`		`01 00111 011`
	`= a, a, LS1`	`// loop`	`101 010 010 11010`

Table 1. SUBT

within an array A is represented as a sequence of a_i bits 0; the bit 1 is used as a (leading) separator or deliminator of the array elements. If there are no 1's (e.g. the register has value 0) then we have an array of size 0. For example, the array [1,4,0] is encoded 10100001, or, depending on chosen endianness, 11000010.

A number of array operations are used frequently, and these have been packaged into subroutines. In our particular program we make use of SIZE (returns the size of the array), APPEND (appends one element), ELM (returns the element at a particular index) and RPL (replaces the element at a particular index).

6. From standard computation to inductive computation

The main program for the P versus NP problem consists of two nested loops. The outer loop tests every program-polynomial tuple. For each program and polynomial, the inner loop checks to see if the program can solve all instances of the subset-sum problem in polynomial steps or less. In the usual model of computation these nested loops have a serious pitfall: The program may run forever for two different reasons. It may run forever because it never finds a program that works (there are infinitely many programs), in which case P does not equal NP. The second reason it may run forever is because it *has* found a program that works, and since there are an infinite number of instances of the subset-sum problem, it loops forever testing all of them.

To resolve this issue, we chose to use a slightly modified version of computation: the *inductive computation* [1]. Under this model, a program is allowed to run forever but still be considered to give an answer if, after a finite number of steps, the *output stabilises*. To make our program suitable for an inductive register machine program, we must modify each loop in the following way: If the loop is successfully running, write a 1 into the output register, otherwise when the loop halts write a 0 into the output register (and stop looping). We thus ensure that the output register will not oscillate, and under the inductive computation model it will always return a result. Namely, the output will be 1 if the loop runs forever, and 0 if at some point it will halt.

Finite standard Turing and inductive computations produce the same results; the inductive computation is more powerful than standard computation.

In what follows we will use the above register machine language as a *universal* *prefix-free inductive machine* U^{ind} (see more in [2]). This type of computation gives rise to an *inductive complexity measure.*

7. An inductive register program for P versus NP

It is easy to note that P $\neq$ NP if and only if every polynomial-time program (i.e. a pair consisting of a program and a polynomial controlling the time of the execution of the program) cannot solve at least one instance of the subset-sum problem. Hence the P versus NP problem can be represented by a Π_2–sentence, i.e. a sentence of the form $\forall n \exists i R(n,i)$, where R is a computable predicate.[b] Starting from a representation $\forall n \exists i R(n,i)$ (where R is a computable predicate) of the P versus NP problem, we construct the inductive register machine program of first order $T_R^{ind,1}$ defined by

$$T_R^{ind,1}(n) = \begin{cases} 1, \text{ if } \exists i R(n,i), \\ 0, \text{ otherwise}\,. \end{cases}$$

Next we construct the inductive register machine $M_R^{ind,2}$ defined by

$$M_R^{ind,2} = \begin{cases} 0, \text{ if } \forall n \exists i R(n,i), \\ 1, \text{ otherwise}\,. \end{cases}$$

Clearly,

$$M_R^{ind,2} = \begin{cases} 0, \text{ if } \forall n \, (T_R^{ind,1}(n) = 1), \\ 1, \text{ otherwise}\,, \end{cases}$$

hence we say that $M_R^{ind,2}$ is an *inductive register machine of second order.*

Note that the predicate $T_R^{ind,1}(n) = 1$ is well-defined because the inductive register machine of first order $T_R^{ind,1}$ always produces an output. However, the inductive register machine $M_R^{ind,2}$ is of the *second order* because it uses an inductive register machine of the first order $T_R^{ind,1}$. This shows that the inductive register machine of second order $M_R^{ind,2}$ solves the P versus NP problem.

MAIN, the main algorithm for $M_R^{ind,2}$ that solves the P versus NP problem, is presented in the algorithm below. As we have already mentioned, the program consists of two nested loops; the outer loop goes through all possible program and polynomial pairs, and the inner loop runs the program with every possible instance of the subset-sum problem, letting it execute at most polynomial steps for each instance.

Our algorithm requires the testing of the correctness of every possible polynomial-time program for each instance of the subset-sum problem. Is it possible to achieve this testing given that the correctness problem is undecidable? The answer is affirmative because we are dealing with time-controlled computations, each running in a finite amount of time; the correctness test may take in some cases exponential time to complete.

---

[b] By now the reader has understood that the word "problem" has different meanings in Cook's theory and in our complexity analysis.

It is important to note that the correctness of the polynomial-time program is established when it runs accurately on *all possible instances* of subset-sum problem. It is not enough for the program to run correctly in only some of the cases, and since we loop through all possibly instances, we will eventually come across the cases in which an invalid program fails. In particular, programs that randomly “guess”, or that always give the same answer eventually fail.

The program-polynomial tuples are generated by incrementing through the natural numbers, treating each number as an array and asking if that array has three elements. Non-complying numbers are ignored; otherwise, we consider the first and second elements to be `C` and `J` respectively, which define the polynomial[c] $\mathtt{C}*(x^{\mathtt{J}}+1)$, and the third element to be the program `P`. To enumerate all instances of subset-sum problem, we similarly go through the natural numbers and interpret them as arrays with at least 2 elements. For each array we ask the question: Does some subset of its first $(N-1)$ elements sum to the N^{th} element, where N is the size of the array?

MAIN: RESULT IS 1 IF P≠NP, 0 IF P=NP

```
// Z is the output register, while the loop is running it is set to 1
Z ← 1
for all tuples (C,J,P) do
  // Now we run the simulation (also on an inductive Turing machine)
  run SIM
  // check the result register (Y)
  if Y = 1 then
    // found a polynomial-time algorithm, P=NP
    Z ← 0
    HALT
  else
    // that program didn't work, try the next one
    continue
  end if
end for
```

When simulating the program `P` we give it access to an unlimited number of registers which are stored in an array `R`. The unique coding of a register name is used to represent the index of that register in the array `R`, and the array dynamically grows to include all possible simulated registers. After running the program `P` we assume that its answer to the subset sum instance is in the register encoded as 010, which corresponds to `R`[2][d]. One can easily check if the proposed answer is correct.

---

[c] Obviously, in this way we cover all possible run-time polynomials.

[d] There is no register named 00 or 01, so `R`[2] is the first possible register that the simulated program could use. Of course, it may not ever use this register, in which case `R`[2] would always be 0, the

As an example, consider the 672^{nd} loop of the outer (main) program. In binary, this corresponds to the number 101010, which, interpreted as an array is $[1, 1, 1]$, meaning $\texttt{C} = 1$, $\texttt{J} = 1$ and $\texttt{P} = 1$. In binary, the program is simply 1.

SIM: RESULT IS 1 IF PROGRAM P SUCCEEDS IN POLYNOMIAL-TIME, 0 IF NOT

```
// Y is the output register, while the loop is running it is set to 1
Y ← 1
for all instances S of subset-sum do
  Simulate program P with input S for at most $(\texttt{C} * (|\texttt{S}|^{\texttt{J}} + 1))$ steps.
  if P executed without error and calculated the correct answer then
    continue to next instance
  else
    // This program doesn't work, stop looping
    Y ← 0
    return
  end if
end for
```

The outer program then executes the simulation on another inductive Turing machine. The first instance of subset-sum that it will test is 11, the two element array $[0, 0]$. It asks: Is there a subset of elements in [0] that add up to 0? Clearly, the answer is positive. In preparation for simulation, the input for program `P` is made prefix free: 11 becomes 11011, so the program is expected to read five bits (which are read right to left) before halting. The simulation begins, but the parser quickly realises that 1 is not a valid program, and will branch to an error statement. In this statement, the inductive Turing machines writes 0 to its output register (named `Y` in the pseudocode) and halts. The main loop (the outer inductive Turing machine) queries the output register, reads the zero and continues on to the next loop.

In the simulation process every syntactical error is detected. The simulated program may fail because of compile or run time errors, e.g. invalid use of a register/literal encoding, the halt instruction appears in the middle of a program, branching out of bounds, not reading all input bits. The program is also disqualified if, after the prescribed polynomial number of steps, it has not naturally halted.

The flowcharts for the main program, simulation, run and commands for the instructions &, +, =, !, the commented program as well as the full program are presented in Appendices A, B and C.

default initial value.

8. An upper bound for the inductive complexity of P versus NP

To every mathematical sentence of the form $\rho = \forall n \exists i R(n,i)$, where $R(n,i)$ is a computable predicate, we associate the inductive register machine of second order $M_R^{ind,2}$ as above. Note that there are many programs for universal prefix-free inductive machine U^{ind} which implement $M_R^{ind,2}$. For each of them we have:

$$\forall n \exists i R(n,i) \text{ is true if and only if } U^{ind}(M_R^{ind,2}) = 0.$$

The inductive complexity measure of second order is defined by:

$$C_U^{ind,2}(\rho) = \min\{|M_R^{ind,2}| \,:\, \rho = \forall n \exists i R(n,i)\},$$

and, correspondingly, the *inductive complexity class of second order* is:

$$\mathfrak{C}_{U,n}^{ind,2} = \{\rho \,:\, \rho = \forall n \exists i R(n,i), C_U^{ind,2}(\rho) \leq 2^{10} \cdot n\}.$$

The complexity measure, as stated, is unfortunately *incomputable* (see [11]), so we resort to measuring upper bounds of the complexity. This is still a useful measurement and allows us to rank and compare conjectures [4].

The inductive register program based on the main algorithm described above consists of 362 instructions and was encoded with 6,495 bits (after optimising the coding according to the frequency of registers), putting the P versus NP problem into the *inductive complexity class of second order 7*. The syntactical correctness of the program was checked using a tool developed in [18]. Encodings of the first ten instructions of the program and the full program encoding are presented in Appendices D and E.

The Riemann hypothesis, another problem on the list of the Clay Mathematics Institute millennium open problems [28] and arguably the most important open problem in mathematics, is in the *inductive complexity class of first order 3*, a significantly lower complexity class. Goodstein theorem—which is unprovable in Peano Arithmetic—is also in *the inductive complexity class of second order 7*. The Collatz conjecture and the twin-prime conjecture fall into the first inductive complexity class of second order, cf.[2]; all other problems studied till now are Π_1–sentences and they all fall into inductive complexity classes of first order smaller than 4, cf. [6, 9, 10, 12, 16].

As with all complexity measures of this type, this is only an upper bound. There are probably further modifications that can be made to shorten the program, possibly by improving the simulation potential polynomial-time programs and/or by using a different NP–complete problem. Our analysis has used the representation of the P versus NP problem as a Π_2–sentence; it is an open question whether *the P versus NP problem is a* Π_1*–sentence.*

Acknowledgment

We thank M. Dinneen, J. Hertel and, particularly, S. Rudeanu, for many comments and suggestions which improved the paper. A. Gandhi and H. Naderi have developed the tool described in [18].

References

[1] M. Burgin. *Super-recursive Algorithms*, Springer, Heidelberg, 2005.

[2] M. Burgin, C.S. Calude, E. Calude. Inductive Complexity Measures for Mathematical Problems, *CDMTCS Research Report* 416, 2011, 11 pp.

[3] W. Byers. *The Blind Spot: Science and the Crisis of Uncertainty*, Princeton University Press, Princeton, 2011.

[4] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, 18-3 (2009), 267–285.

[5] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, 18-4, (2010), 387–401.

[6] C. S. Calude, E. Calude. The complexity of the Four Colour Theorem, *LMS J. Comput. Math.* 13 (2010), 414–425.

[7] C. S. Calude, E. Calude. The Complexity of Mathematical Problems: An Overview of Results and Open Problems, *CDMTCS Research Report*, 410, 2011, 12 pp.

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

[9] C. S. Calude, E. Calude, M. S. Queen. The complexity of Euler's integer partition theorem *Theoretical Computer Science*, 2012, doi: 10.1016./j.tcs.2012.03.02.

[10] C. S. Calude, E. Calude, K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.

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

[12] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, 18 (3-4) (2012), 257–265.

[13] S. Cook. The complexity of theorem proving procedures, in *STOC '71, Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ACM New York, NY, USA, 1971, 151–158.

[14] S. Cook. The P versus NP Problem, manuscript, 12 pages, `http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf`, visited on 16 June 2012.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* MIT Press and McGraw-Hill, (2nd ed.) (2001) [1990].

[16] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures, in M. J. Dinneen, B. Khoussainov, A. Nies (eds.). *Computation, Physics and Beyond*, LNCS 7160, Springer, Heidelberg, 2012, 81–93.

[17] L. Fortnow. The status of the P vs NP problem, *CACM* 52, 9 (2009), 78–86.

[18] A. Gandhi. *Register Machine Syntax Checker and Translator: Manual*, University of Auckland, Version 0.0.3.0, April, 2012.

[19] J. Hertel. Inductive complexity of Goodstein's theorem, in J. Durand-Lose, N. Jonoska (eds.). *Proceedings UCNC 2012*, LNCS 7445, Springer, 2012, 141–151.

[20] E. Horowitz, S. Sahni. Computing partitions with applications to the knapsack problem, *JACM* 21(1974), 277–292.

[21] A. Jackson. Interview with Martin Davis, *Notices AMS* 55, 5 (2008), 560–571.

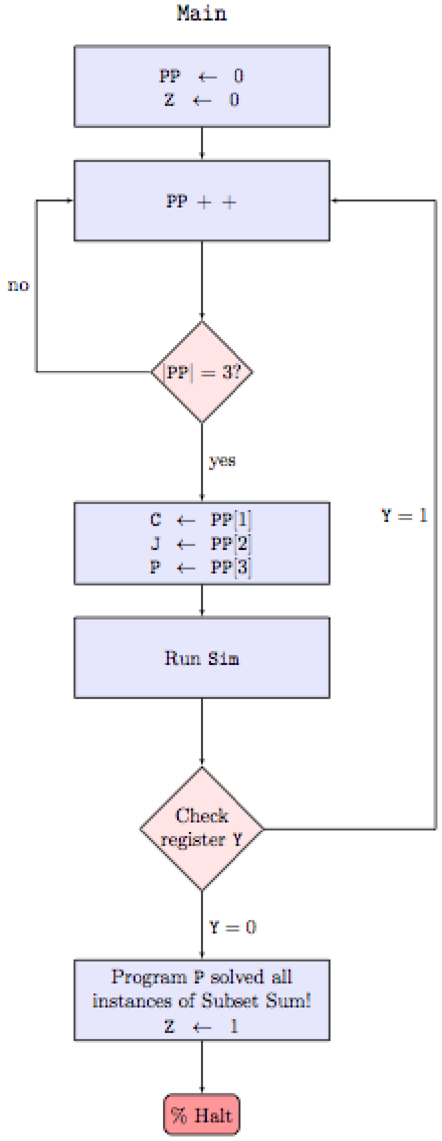
[22] V. Kreinovich, M. Margenstern. In some curved spaces, one can solve NP-hard problems in polynomial time, *Journal of Mathematical Sciences* 158, 5 (2009), 727–740. (Originally published in Russian in *Zapinski Nauchnykh Seminarov POMI*, 358 (2008), 224-250.)

[23] L. Levin. Universal search problems, *Problemy Peredachi Informatsii* 9 (1973), 265–266 (in Russian). English translation in [26].

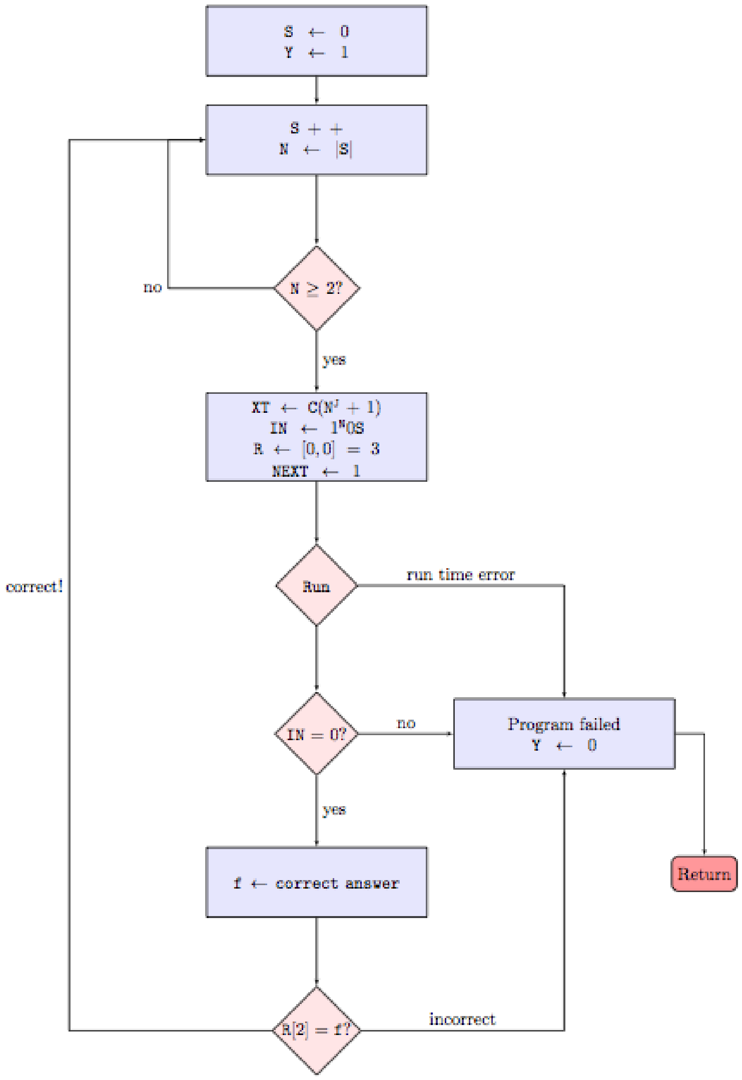
[24] C. Moore, S. Mertens. *The Nature of Computation*, Oxford University Press, Oxford, 2011.

[25] K. D. Mulmuley. The GCT program toward the P vs NP problem, *CACM* 55, 6 (2012), 98–107.
[26] B. A. Trakhtenbrot. A survey of Russian approaches to Perebor (brute-force search) algorithms, *Annals of the History of Computing* 6 (1984), 384–400.
[27] `http://www.claymath.org/millennium/P_vs_NP/`, visited on 16 June 2012.
[28] `http://www.claymath.org/millennium/Riemann_Hypothesis/`, visited on 16 June 2012.
[29] G. J. Wöginger. The P-versus-NP webpage, `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm`, visited on 16 June 2012.

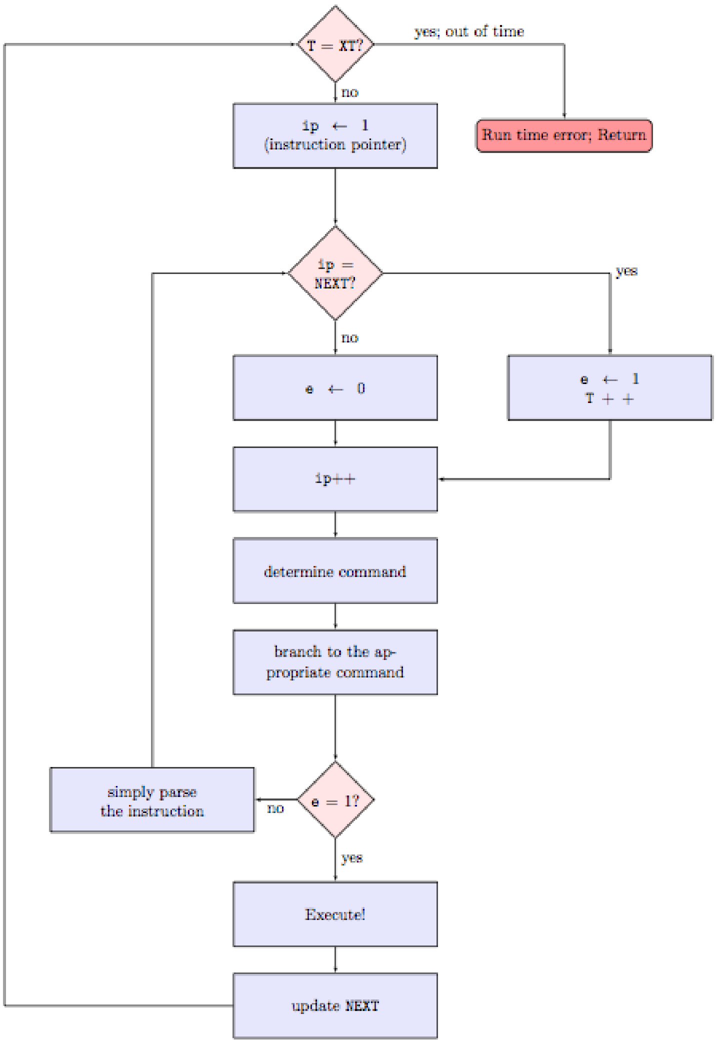
Appendix A: Program flowcharts



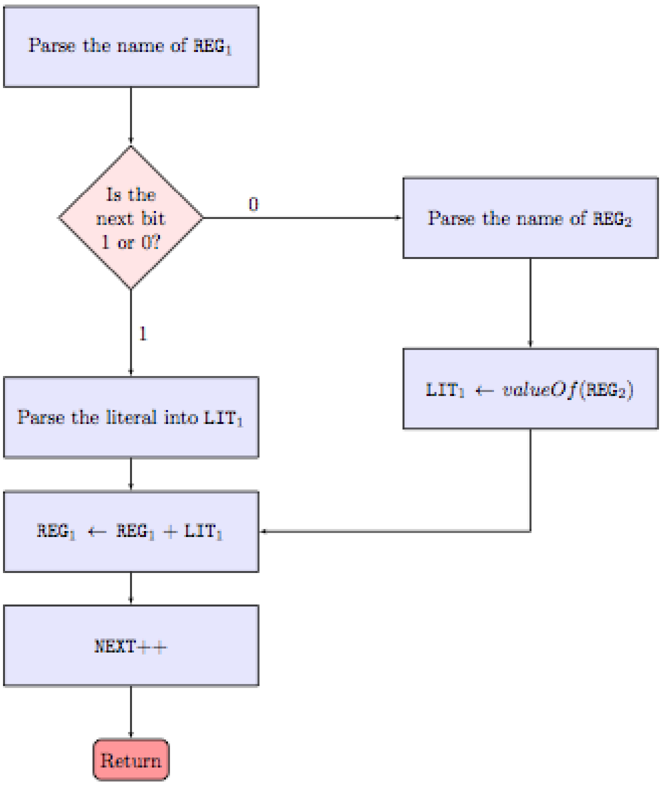
`Sim`


S ← 0
Y ← 1
S + +
N ← |S|
no
N ≥ 2?
yes
XT ← C(N^J + 1)
IN ← 1^N0S
R ← [0, 0] = 3
NEXT ← 1
correct!
Run
run time error
IN = 0?
no
Program failed
Y ← 0
Return
yes
f ← correct answer
R[2] = f?
incorrect

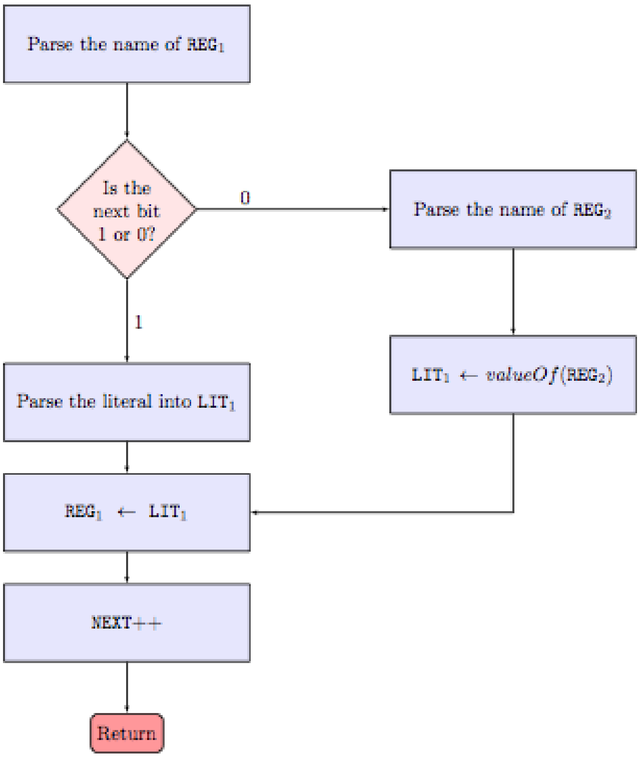
Run


T = XT?
yes; out of time
Run time error; Return
no
ip ← 1
(instruction pointer)
ip =
NEXT?
yes
no
e ← 0
e ← 1
T + +
ip++
determine command
branch to the appropriate command
simply parse
the instruction
no
e = 1?
yes
Execute!
update NEXT

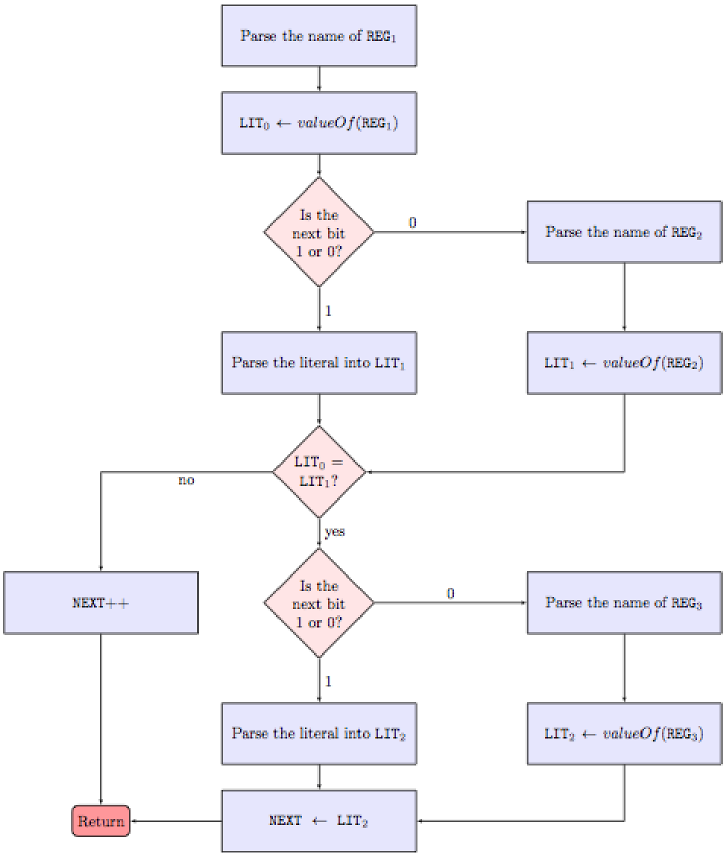
COMMAND “+”


Parse the name of $\texttt{REG}_1$
Is the next bit 1 or 0?
0
Parse the name of $\texttt{REG}_2$
1
Parse the literal into $\texttt{LIT}_1$
$\texttt{LIT}_1 \leftarrow valueOf(\texttt{REG}_2)$
$\texttt{REG}_1 \leftarrow \texttt{REG}_1 + \texttt{LIT}_1$
NEXT++
Return

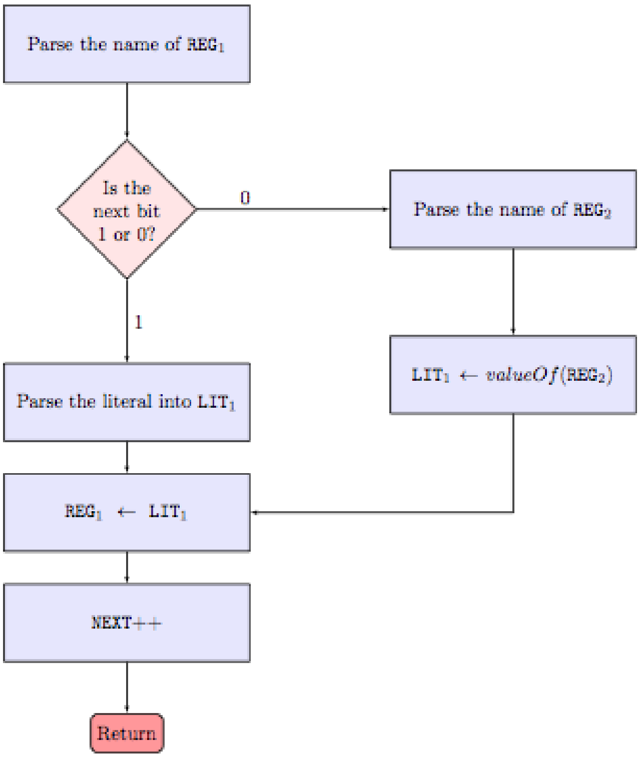
COMMAND “&”


Parse the name of $\texttt{REG}_1$
Is the next bit 1 or 0?
0
Parse the name of $\texttt{REG}_2$
1
$\texttt{LIT}_1 \leftarrow valueOf(\texttt{REG}_2)$
Parse the literal into $\texttt{LIT}_1$
$\texttt{REG}_1 \leftarrow \texttt{LIT}_1$
NEXT++
Return

COMMAND "="


Parse the name of REG$_1$
LIT$_0$ ← $valueOf$(REG$_1$)
Is the next bit 1 or 0?
0
Parse the name of REG$_2$
1
Parse the literal into LIT$_1$
LIT$_1$ ← $valueOf$(REG$_2$)
LIT$_0$ = LIT$_1$?
no
yes
NEXT++
Is the next bit 1 or 0?
0
Parse the name of REG$_3$
1
Parse the literal into LIT$_2$
LIT$_2$ ← $valueOf$(REG$_3$)
Return
NEXT ← LIT$_2$

COMMAND "&"


Parse the name of $\texttt{REG}_1$
Is the next bit 1 or 0?
0
Parse the name of $\texttt{REG}_2$
1
$\texttt{LIT}_1 \leftarrow valueOf(\texttt{REG}_2)$
Parse the literal into $\texttt{LIT}_1$
$\texttt{REG}_1 \leftarrow \texttt{LIT}_1$
NEXT++
Return

Appendix B: Commented register machine program

```
 // Program for P=?NP

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~
// UTILITIES
// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~

        = a, a, MAIN

// d = a^b, returns to c
POW     & d, 1
        & e, 0
LPW1    = e, b, c
        + e, 1
        & f, 1            // d = d*a = d+d+ ... +d
        & dp, d
LPW2    = f, a, LPW1
        + d, dp
        + f, 1
        = a, a, LPW2

// d = 0, 1, 2 for a==b, a<b, a>b
CMP     & d, 0
        = a, b, c         // a==b
        & e, 0
LCP1    & d, 1
        = a, e, c         // a<b
        & d, 2
        = b, e, c         // a>b
        + e, 1

// d = a-b, returns to c
// Assumes a >= b
SUBT    & d, 0
LS1     & e, d
        + e, b
        = e, a, c        // d+b=a
        + d, 1
        = a, a, LS1

// Divides a in half ( a=a/2 ) and puts the remainder ( a%2 ) into c
// returns to b
```

```
DIV2    & ad, a
        & a, 0
LD1     & c, 0            // a%2 = 0 when a is even
        & e, a
        + e, e            // calculate 2a
        = e, ad, b        // if 2a==ad, then a is our answer, so exit
        & c, 1            // a%2 = 1 when a is even
        + e, 1
        = e, ad, b        // if 2a+1==ad, then a is our answer, so exit
        + a, 1
        = a, a, LD1       // otherwise, a++ and continue looping

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~
// ARRAY LIBRARY
// Array elements represent their value by a string of x number of zeroes,
// x being the value of the element. Elements are deliminated by ones.
// For example: A = 100010100 = [3,1,2]
// We start indexing at 1, so from above A[1]=3 A[2]=1 A[3]=2
// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~

// c = |a|
// returns to b
SIZE    & as, a
        & bs, b
        & cs, 0           // counter
LS1     & b, LS2
        = a, a, DIV2      // look at the last bit
LS2     + cs, c           // add to the counter
        = a, 0, LS3       // if there are no more 1's, then exit
        = a, a, LS1       // otherwise continue looping
LS3     & a, as           // restore values and exit
        & b, bs
        & c, cs
        = a, a, b

// Appends value 'b' to the end of array a, returns to c. Alters a directly.
APPEND  & ap, 0           // loop counter
        + a, a
        + a, 1            // add the `1' element separator
LP1     = ap, b, LP2      // loop until we have added b zeros
        + a, a
        + ap, 1
        = a, a, LP1
```

```
LP2     = a, a, c         // exit

// Gets element I from array a, puts into d ( d = a[I] )
// returns to c
// Since indexing starts from the left but we read off bits from the right,
// it is easiest to get the (N-I)th element from the right
ELM     & ae, a
        & be, b
        & ce, c
        & b, LE1
        = a, a, SIZE
LE1     & a, c            // a = N = size of array
        & b, I
        & c, LE2
        = a, a, SUBT      // d = N-I
LE2     & ie, d
        & d, 0            // counter
LE3     = ie, d, LE6      // if we have counted enough 1's, goto LE6
        & b, LE4
        = a, a, DIV2      // otherwise, halve a
LE4     + d, c            // add last bit to our counter
        = a, a, LE3
LE6     & d, 0            // reset the counter; now we're counting 0's
LE7     & b, LE8
        = a, a, DIV2
LE8     = c, 1, LE10      // if we got to a 1, then exit
        + d, 1            // otherwise increment the counter
        = a, a, LE7
LE10    & a, ae
        & b, be
        = a, a, ce        // restore and return

// replaces a[I] with b, returns to c
// assumes that I is within bounds
RPL     & ar, a
        & br, b
        & cr, c
        & ir, I
        & b, LR0
        = a, a, SIZE
LR0     & nr, c           // nr = N = size of array
        & I, 0            // counter
        & fr, 0           // new array
```

```
LR1     = I, nr, LR7      // have reached end of the array
        = I, ir, LR5      // have reached the element to replace
        & a, ar
        & c, LR2
        = a, a, ELM
LR2     & b, d            // b = A[i]
        & a, fr
        & c, LR3
        = a, a, APPEND    // adds element onto our new array
LR3     & fr, a
        + I, 1
        = a, a, LR1       // loop
LR5     & b, br           // insert the new element
        & a, fr
        & c, LR3
        = a, a, APPEND    // loop back
LR7     & a, fr
        & b, br
        & c, cr
        & I, ir
        = a, a, c

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~
// SUBROUTINES
// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~

// parse a reg or a lit
// Determines if it should branch to READREG or READLIT
// Reads from stream a, return value in b
REGLIT  & bg, b
        & ISLIT, 0           // indicator register, used by the
                             // executing routine
        & b, LP27
        = a, a, DIV2
LP27    & b, bg              // restore the return value for READREG or READLIT
        + a, a               // restore the stream
        = c, 0, READREG      // parse the reg
        & ISLIT, 1
        + a, 1               // restore the stream and parse a lit

// Read a literal encoding off of the stream a
// And puts the actual value into LIT (for use by executing routines)
// returns to b
```

```
READLIT & brl, b
        & LIT, 0
        = a, 0, ERROR        // can't parse an empty string
        & irl, 0             // counter, counts leading ones and the deliminator
LRL0    & b, LRL1
        = a, a, DIV2
LRL1    + irl, 1             // counter++
        = c, 0, LRL2
        = a, a, LRL0         // loop until we find a zero
LRL2    & jrl, 0             // counter, counts the final i bits
LRL3    + jrl, 1
        + LIT, LIT           // record the value of the literal
        = c, 0, LRL4         // push on a 0
        + LIT, 1             // push on a 1
LRL4    = irl, jrl, brl      // done, return to b
        & b, LRL3            // otherwise read another bit and loop
        = a, a, DIV2

// Read a register encoding off of stream a
// And puts the actual register name into REG (for use by executing routines)
// Does check to make sure the stream represents a register
// Also checks to see if the register array is large enough to contain
// this register, and expands it if not.
// returns to b
READREG       & brr, b
        & REG, 0
        = a, 0, ERROR        // can't parse an empty string
        & irr, 0             // counter, counts leading zeroes
                             // and the deliminator
        & b, LRR0
        = a, a, DIV2
LRR0    = c, 1, ERROR        // register must begin with a string of zeroes
LRR1    + irr, 1             // counter++
        = c, 1, LRR2
        & b, LRR1
        = a, a, DIV2
LRR2    & jrr, 0             // counter, counts the final i bits
LRR3    + jrr, 1
        + REG, REG           // record the name of the register as well
        = c, 0, LRR4         // push on a 0
        + REG, 1             // push on a 1
LRR4    = irr, jrr, LRR5     // parsed name, now see if register is
                             // in our array
```

```
        & b, LRR3
        = a, a, DIV2         // otherwise read another bit and loop
LRR5    & arr, a             // save stream
LRR8    & a, R
        & b, LRR6
LRR6    = a, a, SIZE         // get the size of the register array
        & a, c
        & b, REG
        & c, LRR7
        = a, a, CMP
LRR7    = d, 1, LRR9         // REG > |R| , array not large enough
        & a, arr             // array is large enough, restore stream
                             // and return
        = a, a, brr
LRR9    + R, R               // add another element (with value zero) to the
                             // array and recheck
        + R, 1
        = a, a, LRR8

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~
// PARSING AND EXECUTING
// Program is in register P
// Parses through NEXT-1 instructions, and then executes the NEXT instruction.
// If there are only NEXT-1 instructions, then we go to the HALT command.
// If an error, compile or run-time, is encountered it returns to ERROR.
// Otherwise, it returns to b.
RUN     & bp, b
        = T, XT, ERROR       // ran out of time
        & a, P               // make a copy of the program
        & ip, 1              // counts the number of instructions we've parsed
LP0     = a, 0, LP26         // we are done reading when the stream is 0
        = ip, NEXT, EXECUTE  // when we've parsed NEXT-1 instructions,
                             // it's time to execute
        & e, 0               // not executing, set e to false (0)
        = a, a, LP50
EXECUTE & e, 1               // executing, set e to true (1)
        + T, 1
LP50    + ip, 1
        & b, LP1
        = a, a, DIV2         // start reading off bits to determine the command
LP1     = c, 0, LP2
        & b, LP5
        = a, a, DIV2
```

```
LP5     = c, 0, ERROR        // 10 no such instruction (we don't allow % except
                             // as last instruction)
        & b, LP6
        = a, a, DIV2
LP6     = c, 0, LP7

//~~~*~~~*~~~*~~~ 111 [ + ] ~~~*~~~*~~~*~~~
LP21    = e, 1, LX1          // execute this instruction

        & b, LP19            // simply parse, don't execute
        = a, a, READREG      // parse a reg
LP19    & b, LP0
        = a, a, REGLIT       // parse a reg or lit, and loop to next instruction

LX1     & rn, LX15           // set the local return value
        = a, a, LX14         // LX14 reads a reg, then a reg or lit, and
                             // increments NEXT
LX15    + LIT2, LIT1         // LIT2 = LIT2+LIT1
LX16    & I, REG1
        & a, R
        & b, LIT2
        & c, RUN
        = a, a, RPL          // R[REG]=LIT2, returns successfully

//~~~*~~~*~~~*~~~ 110 [ ! ] ~~~*~~~*~~~*~~~
LP7     = e, 1, LX2          // execute this instruction

        & b, LP0             // simply parse, don't execute
        = a, a, READREG      // parse a reg, and loop to next instruction

LX2     + NEXT, 1            // NEXT++
        & b, LX8
        = a, a, READREG      // reads the name of the register into REG
LX8     = IN, 0, ERROR       // can't read from an empty input stream
        & a, IN
        & b, LX19
        = a, a, DIV2
LX19    & b, c               // b = next bit from the input
        & a, R
        & I, REG
        & c, RUN
        = a, a, RPL          // replaces REG with the input bit, and returns
                             // successfully
```

```
//~~~*~~~*~~~*~~~ parsing the command
LP2     & b, LP3
        = a, a, DIV2
LP3     = c, 0, LP4

//~~~*~~~*~~~*~~~ 01 [ & ] ~~~*~~~*~~~*~~~
        = e, 1, LX3          // execute this instruction

        = a, a, LP21         //(same arguments as [+]) parse, don't execute

LX3     & rn, LX16
        = a, a, LX14         // executes same code as [ + ] except for line
                             // LX15 [+ LIT2, LIT1]

//~~~*~~~*~~~*~~~ 00 [ = ] ~~~*~~~*~~~*~~~
LP4     = e, 1, LX4          // execute this instruction

        & b, LP22            // simply parse, don't execute
        = a, a, READREG      // parse a reg
LP22    & b, LP24
        = a, a, REGLIT       // parse a reg or lit
LP24    = a, a, LP19         // parse another reg or lit, and loop to
                             // next instruction

LX4     & rn, LX17
        = a, a, LX14         // get LIT1 and LIT2
LX17    = LIT1, LIT2, LX18   // see if we'll branch
        = a, a, RUN          // return if we don't branch
LX18    & rn, LX20
        = a, a, LX21         // get the line we are branching to
LX20    & NEXT, LIT2
        = a, a, RUN          // update NEXT and return successfully

//~~~*~~~*~~~*~~~
LP26    = ip, NEXT, bp       // if the stream is empty and we need to execute
                             // the next command, we halt by returning to the
                             // main program
        = a, a, ERROR        // otherwise, instruction number is invalid

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~
// EXECUTING
// Registers are stored in array R
```

```
// Input stream is in register IN
// If there is a run-time error, go to ERROR
// Otherwise, after executing return to RUN and continue simulating the
// program

// Parses a REG, puts into REG1 and value into LIT1, then a REG or LIT
// and puts value into LIT2
// returns to rn
// It’s a routine that almost all the commands use in some form
LX14    + NEXT, 1            // NEXT++
        & b, LX9
        = a, a, READREG      // read a register
LX9     & REG1, REG
        & I, REG1
        & ax, a              // save the instruction stream
        & a, R
        & c, LX10
        = a, a, ELM
LX10    & LIT1, d            // LIT1 = valueOf(REG1) = R[REG]
        & a, ax              // restore the instruction stream
LX21    & b, LX11
        = a, a, REGLIT
LX11    & ax, a              // save the instruction stream
        = ISLIT, 1, LX12
        & I, REG             // if it’s a register, get its value
        & a, R
        & c, LX13
        = a, a, ELM
LX13    & LIT, d
LX12    & LIT2, LIT          // LIT2 = valueOf(REG2) = R[REG2]
        & a, ax              // restore the instruction stream
        = a, a, rn           // return

// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~ //
//   |\   /|    /\    -----  |\   |    //
//   | \ / |   /  \     |    | \  |    //
//   |  ’  |  /----\    |    |  \ |    //
//   |     | /      \ -----  |   \|    //
// ~~~~~*~~~~~*~~~~~*~~~~~*~~~~~*~~~~~ //
// This main program is considered to run on an Inductive Turing machine.
// Furthermore, the inner loop that tests all possible instances of Subset Sum
// is also considered to run on an Inductive Turing machine, making the total
// machine of second order.
```

```
// The result of the inner machine is written into register Y and read by the
// outer machine.
// The outer machine (the one that loops and tests all possible programs) writes
// its result into register Z. If Z=1, it means that some program has been found,
// but if Z=0 it means none of the programs have worked.

// Generate a program and a polynomial
MAIN    & PP, 0
        & Z, 0              // have not yet found a working program
L0      + PP, 1             // generate a new array (any errors that disqualify
                            // a program go here to generate a new program)
        & a, PP             // count the elements
        & b, L1
        = a, a, SIZE
L1      = c, 3, L2
        = a, a, L0          // only use arrays with exactly 3 elements
L2      & I, 1
        & c, L3
        = a, a, ELM
L3      & C, d              // C = 1st element
        & I, 2
        & c, L4
        = a, a, ELM
L4      & J, d              // J = 2nd element
        & I, 3
        & c, L5
        = a, a, ELM
L5      & P, d              // P = 3rd element
        = a, a, SIM         // run simulation: this means running the second
                            // inductive Turing machine
L6      = Y, 0, L0          // see what the result was; if it failed (0) go try
                            // another program
        & Z, 1              // otherwise, we have found a program that solves all
                            // instances in poly time, and P=NP
        = a, a, HALT        // we can halt

// Now that we have a program and a polynomial, we will simulate the program on
// every instance of Subset Sum and see if it can correctly generate the answer
// in polynomial time for every case. This part of the program is considered to
// also run on an inductive turing machine; its output register is Y and a 1
// indicates that the program is successfully executing all instances, and a 0
// indicates it failed on some instance (when it fails we also halt)
SIM     & S, 0              // S is our instance of Subset Sum
```

```
        & Y, 1               // we have been successful thus far
L7      + S, 1               // Generate a new instance
        & a, S
        & b, L8
        = a, a, SIZE
L8      & N, c               // N = size of the array
        = N, 1, L7           // Must have at least 2 elements
        & a, N               // calculate our max time, based on polynomial
                             // values C and J
        & b, J
        & c, L9
        = a, a, POW
L9      + d, 1               // d = N^J + 1
        & e, 0
        & XT, 0
L10     = e, C, L11
        + XT, d
        + e, 1
        = a, a, L10          // XT = max time = C(N^J + 1)
L11      & IN, S              // the input to the program is this instance
        + IN, IN             // to keep the input prefix-free, we include the
                             // size of the array
        & i, 0               // counter
L12     = i, N, L13
        + i, 1
        + IN, IN
        + IN, 1
        = a, a, L12          // add N ones to denote that there are N elements

L13     & T, 0               // reset the clock
        & R, 3               // empty the register array (11 = [a1,a2] = [0,0])
        & NEXT, 1            // begin with the first instruction
        & b, L14
        = a, a, RUN          // Run the simulation!

ERROR   & Y, 0               // encountered an error... this program fails
        = a, a, L6           // additionally, we can halt this loop (go
                             // try another program)

// We returned from running the simulation, meaning we got to % without error.
// To execute the HALT command, we check two things:
// -> Did it read all of the input? If not, this is an under-read error
// -> Did it compute the correct answer? We assume that R1 is the answer register.
```

```
// If the program got the answer correct, we loop to test another instance of
// the problem.
L14     = IN, 0, L15
        = a, a, ERROR        // underread error

// Calculating the correct answer
// Answer is 1 if there exists a subset that sums up to the last element, and
// 0 if not

L15     & a, 2
        & b, N
        & c, L16
        = a, a, POW
L16     & da, d              // da = 2^N
        & e, 0               // counter
        & f, 0               // answer
L17     + e, 1               // test another subset
        = e, da, L24         // tested all subsets, exit
        & a, e
        & I, 0
        & s, 0               // s = sum
L18     = a, 0, L17          // finished adding up that subset
        + I, 1
        & b, L19
        = a, a, DIV2
L19     = c, 0, L18
        & aa, a              // save the number
        & a, S
        & c, L20
        = a, a, ELM
L20     + s, d               // add the Ith element to our sum
        & a, aa              // restore the number
        = a, a, L18          // and test next digit
L21     & a, S
        & I, N
        & c, L22
        = a, a, ELM
L22     = s, d, L23          // does sum = target?
        = a, a, L17          // if not, test another subset
L23     & f, 1               // did find a satisfying subset

        // The first register is encoded in binary as 010, which corresponds
        // to the second index in our register array.
```

```
L24     & I, 2               // check the element
        & a, R
        & c, L7
L25     = a, a, ELM          // d = the answer the simulated program
                             // ‘calculated’
        = d, f, L7           // got the answer correct, test next instance
        = a, a, ERROR        // got the answer wrong...

HALT    %
```

Appendix C: The register machine program

```
1       = a a 265
2       & d 1
3       & e 0
4       = e b c
5       + e 1
6       & f 1
7       & dp d
8       = f a 4
9       + d dp
10      + f 1
11      = a a 8
12      & d 0
13      = a b c
14      & e 0
15      & d 1
16      = a e c
17      & d 2
18      = b e c
19      + e 1
20      & d 0
21      & e d
22      + e b
23      = e a c
24      + d 1
25      = a a 21
26      & ad a
27      & a 0
28      & c 0
29      & e a
30      + e e
31      = e ad b
32      & c 1
33      + e 1
34      = e ad b
35      + a 1
36      = a a 28
37      & as a
38      & bs b
39      & cs 0
40      & b 42
41      = a a 26
42      + cs c
43      = a 0 45
44      = a a 21
45      & a as
46      & b bs
47      & c cs
48      = a a b
49      & ap 0
50      + a a
51      + a 1
52      = ap b 56
53      + a a
54      + ap 1
55      = a a 52
56      = a a c
57      & ae a
58      & be b
59      & ce c
60      & b 62
61      = a a 37
62      & a c
63      & b I
64      & c 66
65      = a a 20
66      & ie d
67      & d 0
68      = ie d 73
69      & b 71
70      = a a 26
71      + d c
72      = a a 68
73      & d 0
74      & b 76
75      = a a 26
76      = c 1 79
77      + d 1
78      = a a 74
79      & a ae
80      & b be
81      = a a ce
82      & ar a
83      & br b
84      & cr c
85      & ir I
86      & b 88
87      = a a 37
88      & nr c
89      & I 0
90      & fr 0
91      = I nr 107
92      = I ir 103
93      & a ar
94      & c 96
95      = a a 57
96      & b d
97      & a fr
98      & c 100
99      = a a 49
100     & fr a
101     + I 1
102     = a a 91
103     & b br
104     & a fr
105     & c 100
106     = a a 49
107     & a fr
108     & b br
109     & c cr
110     & I ir
11      = a a c
112     & bg b
113     & ISLIT 0
114     & b 116
115     = a a 26
116     & b bg
117     + a a
118     = c 0 138
119     & ISLIT 1
120     + a 1
```

```
121    & brl b
122    & LIT 0
123    = a 0 321
124    & irl 0
125    & b 127
126    = a a 26
127    + irl 1
128    = c 0 130
129    = a a 125
130    & jrl 0
131    + jrl 1
132    + LIT LIT
133    = c 0 135
134    + LIT 1
135    = irl jrl brl
136    & b 131
137    = a a 26
138    & brr b
139    & REG 0
140    = a 0 321
141    & irr 0
142    & b 144
143    = a a 26
144    = c 1 321
145    + irr 1
146    = c 1 149
147    & b 145
148    = a a 26
149    & jrr 0
150    + jrr 1
151    + REG REG
152    = c 0 154
153    + REG 1
154    = irr jrr 157
155    & b 150
156    = a a 26
157    & arr a
158    & a R
159    & b 160
160    = a a 37
161    & a c
162    & b REG
163    & c 165
164    = a a 12
165    = d 1 168
166    & a arr
167    = a a brr
168    + R R
169    + R 1
170    = a a 158
171    & bp b
172    = T XT 321
173    & a P
174    & ip 1
175    = a 0 240
176    = ip NEXT 179
177    & e 0
178    = a a 181
179    & e 1
180    + T 1
181    + ip 1
182    & b 52
183    = a a 26
184    = c 0 56
185    & b 187
186    = a a 26
187    = c 0 321
188    & b 190
189    = a a 26
190    = c 0 204
191    = e 1 196
192    & b 194
193    = a a 138
194    & b 175
195    = a a 112
196    & rn 198
197    = a a 242
198    + LIT2 LIT1
199    & I REG1
200    & a R
201    & b LIT2
202    & c 171
203    = a a 82
204    = e 1 207
205    & b 175
206    = a a 138
207    + NEXT 1
208    & b 210
209    = a a 138
210    = IN 0 321
211    & a IN
212    & b 214
213    = a a 26
214    & b c
215    & a R
216    & I REG
217    & c 171
218    = a a 82
219    & b 221
220    = a a 26
221    = c 0 226
222    = e 1 224
223    = a a 191
224    & rn 199
225    = a a 242
226    = e 1 232
227    & b 229
228    = a a 138
229    & b 231
230    = a a 112
231    = a a 194
232    & rn 234
233    = a a 242
234    = LIT1 LIT2 236
235    = a a 171
236    & rn 238
237    = a a 253
238    & NEXT LIT2
239    = a a 171
240    = ip NEXT bp
```

```
241    = a a 321
242    + NEXT 1
243    & b 245
244    = a a 138
245    & REG1 REG
246    & I REG1
247    & ax a
248    & a R
249    & c 251
250    = a a 57
251    & LIT1 d
252    & a ax
253    & b 255
254    = a a 112
255    & ax a
256    = ISLIT 1 262
257    & I REG
258    & a R
259    & c 261
260    = a a 57
261    & LIT d
262    & LIT2 LIT
263    & a ax
264    = a a rn
265    & PP 0
266    & Z 0
267    + PP 1
268    & a PP
269    & b 271
270    = a a 37
271    = c 3 273
272    = a a 267
273     & I 1
274     & c 276
275     = a a 57
276     & C d
277     & I 2
278     & c 280
279     = a a 57
280     & J d
281     & I 3
282    & c 284
283    = a a 57
284    & P d
285    = a a 289
286    = Y 0 267
287    & Z 1
288    = a a 362
289    & S 0
290    & Y 1
291    + S 1
292    & a S
293    & b 295
294    = a a 37
295    & N c
296    = N 1 291
297    & a N
298    & b J
299    & c 301
300    = a a 2
301    + d 1
302    & e 0
303    & XT 0
304    = e C 308
305    + XT d
306    + e 1
307    = a a 304
308    & IN S
309    + IN IN
310    & i 0
311    = i N 316
312    + i 1
313    + IN IN
314    + IN 1
315    = a a 311
316    & T 0
317    & R 3
318    & NEXT 1
319    & b 323
320    = a a 171
321    & Y 0
322    = a a 286
323    = IN 0 325
324    = a a 321
325    & a 2
326    & b N
327    & c 329
328    = a a 2
329    & da d
330    & e 0
331    & f 0
332    + e 1
333    = e da 356
334    & a e
335    & I 0
336    & s 0
337    = a 0 332
338    + I 1
339    & b 341
340    = a a 26
341    = c 0 337
342    & aa a
343    & a S
344    & c 346
345    = a a 57
346    + s d
347    & a aa
348    = a a 337
349    & a S
350    & I N
351    & c 353
352    = a a 57
353    = s d 355
354    = a a 332
355    & f 1
356    & I 2
357    & a R
358    & c 291
359    = a a 57
360    = d f 291
361    = a a 321
362    %
```

Appendix D: Examples of encodings[e]

The first ten instructions in the program in Appendix C are presented in machine code followed by the length of the corresponding binary encoding.

```
Instruction number=1
= 00
a 010
a 010
265 11111111000001011
The number of bits 25

Instruction number=2
& 01
d 00110
1 101
The number of bits 10

Instruction number=3
& 01
e 00101
0 100
The number of bits 10

Instruction number=4
= 00
e 00101
b 011
c 00100
The number of bits 15

Instruction number=5
+ 111
e 00101
1 101
The number of bits 11
```

```
Instruction number=6
& 01
f 0001011
1 101
The number of bits 12

Instruction number=7
& 01
dp 00000101100
d 00110
The number of bits 18

Instruction number=8
= 00
f 0001011
a 010
4 11010
The number of bits 17

Instruction number=9
+ 111
d 00110
dp 00000101100
The number of bits 19

Instruction number=10
+ 111
f 0001011
1 101
The number of bits 13
```

---

[e] Produced by the tool [18].

Appendix E: Binary encoding of the program[f]

```
0001001011111111100000101101001101010100101100000010101100100111001011
0101000101110101000001011000011000000101101011010111001100000001011001
1100010111010001001011100100100110100000100110010001001011000100110101
1000100010100100010011011000000110010100100111001011010100110100010001
0100110111001010110000101010001001110011010100010010111100111010000101
1100100101010001001001000100101010111001010010100001010000101100110101
0100101111001011010000101000010110011111010101000100101111011100100000
0101010010010000011101001101000001001001000101111111001100000010010111
1011001110000001001000010000010100111110011110001001011110011101010000
0010101001011000001110100100100000000100100000100100110100001011110011
1010010111010101000000101110111111110110101110100101110000101111010001
0010111110101100001001000100010000010011101001000001101100110100000011
0001001000101111111110000000000100101111100011101010001000101100111010
0100111111000010000010010111100110010000010110100110010011010000000000
1011010011011111110001011010111111111000100100010010111101100111001100
1000001001011111110000110010011010001011111111100011100001001011110110
0000100101111111001000111100110101000100101111111000110001010000001001
1101011000001101100001001000000110001010000001010010100100000010011001
0100000110010001000100001101000111010111111111001101000010010111110001
1101000001110110010001001111000100001001010000000111000001110111111111
1011010000111000011010111111010100101010000000101001010010011111101000
1000010010111110110110101100110010100000100100100100111111101001100001
0010111110100110100001001001011100111101000100101111110011101010110000
0010011001010000010010010010011111101001100001001011111010011010100000
0100100101100000100110010010000000110010010011100001101000010010001000
0100000110111011010000010010110001011111111011011000010010111101100011
0110000011011111101001000001001001111111000011000100000100101101111011
0101010000011100001101000110110000010100111111111001000011010000111101
0001011111111110000000100010010111101100111000011110101000010010011111
1100000100000100101111110111111010000010001010011100000100010101111001
0110100011010000100100111111100001001111000110110100000001111000000100
0100000011100001011111111110000010100010010111101100010000011001101101
0001010100000101001111111100100001101000000100000100010111111111000100
1000010010111101100000010010111111111001000011111000000100000010100001
0101111111100010111010111111111100010011000100101111011000100001111111
0111000011111101111000101000010100000100100111111110001110011100010101
0100000001000000000111111111111000111110101111111111000110000000100101
1101100010000011010001001010000100001011111111110010001000010010111111
```

---

[f] Produced by the tool [18].

```
0011101010001000101100010100100100111111100100111000100101110110000001
1010111111110010101001010000001101000001001000000110011111000100000001
0001110001000101000100101111111001000000100000111001011000000011101000
0010000111111111001000011010100000010111001000010011101000101001111111
1011100100000000100110001111111111100110101010010110000010010111111100
1101110100101101111000011101101111000010011101010111111101011000010 01
0111101100000010010011111011010010111111111001111010001001011111011000
0001001001111111100100001101011111111101000000000100101111011000000010
0100111111101001110000010110111111110100011001011111111110100010000010
0101111111000011000101111111110011000100010010111111011001001000010 00
1111111101001000000010010111111101110100111000010000000011011010011100
0010101010100001000010110000100000100100111111100101101000100101111111
0010100000010110111111110101000101011111111100110001000100101111111100
0011001110001111101010111111111010101000001001011111110000110000000 10
0110011111111001000011010100001001010111111111010110000000100101111011
0001011001000101000010000100111000101001001001111111001011010001001 01
1111100101000101111111110101111100010010111101100000010010011111 11011
0010000000101101111111101100010000100101111111101000000101000010001111 11
1101001001000100101111111011101000000101101111111101101010010111111111
1011001110001001011111110000110001011111111101101001000100101111111011
0010000100101111111010001000100001000111111110110110000010010111111 110
1110100000000110110000100001111111011011100001001011111110010110101 00
0010001111111101110000000100101111111011111110100011110000100000001 00
1011111111001011010000000100110001111000000111001000100101111111100100 00
1111100011111010101111111110111011100010010111111110000110001000010101
0001010010011100001010101000010100010010100001000010010011111110111111
0100010010111110110110100001101100110010100000010100010111111111100000
0001000100101111110110010010000101000100000000100101101111111111000001
0000100111000101001010000100001001001111111100000011100010010111111011
0110100011010011001000010000000110101010000001010000001001000000100 01010
0000100011100010000010101110011100000100011101010100000001000110101111
1111110000100010001001011111000111000010011001111111110000100110001 00
1011111111000001101010011110101001001111111100001011000010010111111 011
0110100000110000001100100111110000100100111111111000011010000100101111
1011011010000010111100110010011111001010010011111111000011110000100 10
1111101101101000000101110001100001001011111111100010001100000001110 01001
1111111000001101010000001010111010001001011111111100110110001000110 0100
0100001110010111100011001010101000011000101111111111100010100100010010
1111100011101000111000100000000111010111111111100010010101010000111 0010
1100000101111010010011111111100010111100010010110001110011010101001011
0001000000100001100000010100000110000111111110001101101110000010000100
1101110010110100010010111111111000110010010001001000110011100010010001
0010100001100110000000001100100011101111111110001111101110000110011 0111
```

```
100010010001001111000100110100010010111111110001110010100001110110001
000100011001010001111101010111111111100100010100010010111111100101101
010000111001000001001011111111000100000000001001100111111110010001110
001001011111111001000011010101100001011000111001001001111111100100101
1000100101100001000000110101001100100101100010001011100111001011010000
10100000011010111111111100110011001010001010100111100010000110001000001
010011111111001001110111001111010101111111111001010111000100101111011
000000100100111111110010100110100000101000010010100001100010010011111
111001011100000010010111110110111110000110000011001010000000101000000010
010111111110010100110101000011000100111000111001001001111111100110001
1000100101111110110110000000110000001101111111110011001010001001011111111
00010010111110110110000011000010111111111100010010100010010111111111001
001001110010001011101010011111000010100001000010010011111111000100101
000011100
```