Fundamental Concepts of Computer Science (working title)

James Arvo

September 29, 2007

Preface

The aim of this book is to introduce the reader foundational ideas in computer science that allow us to *reason about computation.* By treating both computers and programs as abstract mathematical objects it is possible to prove facts about them that transcend any clever design or physical embodiment. For example, we may demonstrate that all computation can be accomplished using a trivial set of operations, and that all computers share intrinsic limitations as to what they can compute. Our focus shall therefore be on computation in the abstract, as opposed to analysis of specific algorithms or computer architectures.

This book is intended for readers who are already familiar with some aspects of computer science, such as programming in a high-level language, some elements of discrete mathematics, and preferably some exposure to fundamental algorithms and their time-complexity. However, it will be assumed that the typical reader has had little or no exposure to topics considered to be within the purview of "theoretical" computer science. For example, while the reader may well have programmed a variety of well-known algorithms and data structures in a language such as Java or C++, we shall assume that she has thus far never encountered a concrete example of a problem that is provably *unsolvable* via computation, and has never confronted and *intractable* problem, for which no practical algorithm exists as yet or is likely to be found.

While mathematical representations are central to ideas discussed here, the degree of formalism that we employ varies throughout the book. At times a high level of formalism is warranted and at others mere sketches of arguments suffice. Thus, some arguments are presented as reasonably complete proofs spelled out in detail using the language of set theory and logic, while others depend heavily upon the reader's ability to see past the details and perceive the larger picture. The singular rationale driving these choices has been clarity. To explicate important concepts it is necessary to provide sufficient detail to connect the underlying ideas, yet not become so mired in detail that the thrust of the argument is obscured. This creates a constant tension between the desiderata of completeness and conciseness; as the tradeoffs made were often a matter of taste, they will not always be ideal for a given reader.

Choosing an appropriate balance between completeness and conciseness is further complicated by the fact that a reader's perspective is a moving target; invariably what one considers to be the appropriate level of description changes with one's understanding of the subject. In an attempt to accommodate different backgrounds and changing preferences, many discussions throughout the book are framed in more than one way. It is hoped that seeing a single concept from multiple perspectives will be beneficial to most readers.

Most of this book can be covered in a semester-long course for upper-level undergraduates or first-year graduates. For a single-quarter course, the chapter on set theory can be given cursory treatment while the sections on recursive functions, Post's correspondence problem, and most of the challenging polynomial-time reductions can be skipped entirely.

Much of the material in this book was originally developed for a year-long sophomore-level course (CS20) that the author taught from 1996 to 2002 at the California Institute of Technology, and was later adapted to a single-quarter upper-division course (CompSci 162) at the University of California, Irvine. I am deeply indebted to the enthusiastic and insightful students I've had the privilege of interacting with at both Caltech and UCI, and I gratefully acknowledge that every page herein has been shaped by their influence.

Chapter 1

Introduction

Computer science, as a field, is concerned with the algorithmic solution of problems and the machines that can carry them out. By a *problem* we mean a question, often stated in mathematical terms, for which we seek an unambiguous answer; by an *algorithmic solution* we mean a clearly-specified sequence of operations simple enough to be carried out by a machine, at least in principle, by which the answer may be produced. For instance, we may seek an algorithmic solution to a problem such as determining whether a given number has any proper divisors (primality testing), or finding an optimal move in a two-player game (searching), or determining whether two geometric shapes overlap (collision detection). But there are other types of questions concerning algorithms, machines, and problems that are also central to computer science. For example, listed below are several theoretical questions that we will examine repeatedly, in various contexts, throughout the book:

- Is problem X solvable by some algorithm?
- Can problem X be solved in a reasonable amount of time?
- Can a machine M solve problem X?
- Are machines M_1 and M_2 equivalent?
- Are problems X and Y equivalent?

Of course, to address such questions we must be considerably more precise; for example, we must defining what is meant by a "reasonable" amount of time, and what it means for two machines or two problems to be "equivalent." Consequently, a crucial part of our endeavor will be to provide definitions that are sufficiently precise to permit clear and unambiguous answers to such questions.

Among the questions listed above, the first one may seem the most unfamiliar. Moreover, it may be surprising the learn that the answer is not always "yes," even for concisely-stated mathematical problems. We will see a number of examples of problems that are provably insoluble by computation. To construct proofs of this nature, it is necessary to have a precise *formal* definition of what a computer (or algorithm) is. That is, in order to produce rigorous *negative* results, such as the

```
procedure Trivial()
    for $k \leftarrow 0...100$ do
        if $k \mod 72 = 3$ then
            print "Hello"
        endif
    endfor
endproc
```

Figure 1.1: *This simple procedure is trivial to analyze. Clearly, it will print "Hello" twice, once when the variable k reaches the values 3, and again when it reaches 75.*

```
procedure Fermat()
    for $k \leftarrow 3...\infty$ do
        for $n \leftarrow 3...k$ do
            for $x \leftarrow 0...k$ do
                for $y \leftarrow 0...k$ do
                    for $z \leftarrow 0...k$ do
                        if $x^n + y^n = z^n$ then
                            print "Hello"
                            return
                        endif
                    endfor
                endfor
            endfor
        endfor
    endfor
endproc
```

Figure 1.2: *This procedure will print "Hello" and return if and only if Fermat's last theorem is false. If the theorem is true, on the other hand, the procedure will loop forever without printing anything. Here we assume that the program can perform arithmetic on integers of arbitrary size.*

fact that *no* device is capable of solving all instances of problem X, it is necessary to completely characterize the class of all such devices; without such a characterization, we cannot rule out the possibility that some innovation could suddenly render the problem solvable. The task of providing such definitions was undertaken in the 1930's by a number of prominent mathematicians, long before the first electronic digital computer existed[1]. We will draw heavily from the theoretical developments in logic and set theory from this period, and see how these ideas have shaped the field of computer science.

1.1 Reasoning about Computation

What does it mean to reason *about* computation? Clearly this phrase connotes something more than writing a program to perform a certain task, or constructing a machine to follow a specific set of instructions. It connotes something that transcends any given programming language or machine architecture. For example, such reasoning may entail determining whether any conceivable algorithm or machine can perform some task, or determining the intrinsic difficulty of certain problems.

As a concrete example of reasoning about computation, let's consider the following question: Given a specific program written in a well-defined programming language, is it always possible to answer clearly-stated questions about what the program will do when it is run? For instance, can we always answer the question "Will this program eventually print 'Hello' when it is run?" Clearly, such a behavior would be trivial to ascertain for some programs, such as those containing no output instructions whatsoever (in which case, the answer is "no"), or those whose behavior is as transparent as the program *Trivial* shown in Figure 1.1. However, there are also programs for which this determination may be exceedingly difficult, such as the program *Fermat* shown in Figure 1.2. We shall refer to the task of determining whether a given program eventually prints the string "Hello" as the *Hello problem*.

Given any triple of positive integers, (x, y, z), and any integer $n > 2$, the program *Fermat* will eventually perform a test to determine whether $x^n + y^n = z^n$, and it will print "Hello" if the equality holds. While there is enormous redundancy in this testing, with the same triples being tested again and again, the central feature of the program is that every such combination is eventually checked *at least once*. While a more sophisticated program could manage to check each combination exactly once, this would be of no consequence regarding the "Hello" problem–that is, it would not change our answer as to whether the program would *eventually* print "Hello."

As the attentive reader may have noticed, the function *Fermat* will print "Hello" if and only if *Fermat's Last Theorem* is *false*; that is, if and only if there exists a *counter example* to Fermat's Last Theorem, which asserts that no such collection of integers exists. Up until Fermat's Last Theorem was finally proven by the mathematician Andrew Wiles in 1995 [**?**], it had been uncertain as to whether the theorem was actually true; while no counter-example had ever been found, the proof nonetheless evaded mathematicians for centuries. Hence, before 1995, nobody could say definitively whether the program *Fermat* would eventually print "Hello" or not. While it is true that today we can say with confidence that the program will run forever without ever printing "Hello," it is clear that proving this fact about the program *Fermat* is no less difficult than proving the theorem itself, which was a monumental task involving some rather esoteric mathematics. Evidently, then, being able to prove simple assertions about the behavior of programs must be exceedingly difficult in general. Put another way, the problem of determining whether an arbitrary program ever prints "Hello" is at least as difficult as resolving Fermat's Last Theorem, which we might express more succinctly as

$$\text{Fermat's Last Theorem} \;\leq\; \text{The "Hello" Problem.} \tag{1.1}$$

Here the use of an inequality symbol is apropos, for what we have established thus far is a *lower bound* on the difficulty of the "Hello" problem, where we are using the word "difficulty" in its colloquial

[1]ENIAC, an acronym for *Electronic Numerical Integrator and Computer*, was arguably the first truly general-purpose electronic computer. It was switched on in 1946.

sense; that is, requiring considerable effort or expertise. In contrast, an equality symbol would be ill-advised, as there may well be even harder problems that could be solved by virtue of a solution to the "Hello" problem; Fermat's Last Theorem is, after all, a solved problem today. Inequality (1.1) is our first (informal) example of a *reduction*; that is, a statement that one type of problem can be expressed in the language of another. Here we have expressed a specific problem of number theory in terms of a specific question about the behavior of a program; if we could devise a way to answer the latter, this technique would immediately translate into a way to answer the former.

There are a great many theorems about numbers that have thus far resisted all attempts at proof or refutation. As another example, consider the following famous conjecture first posed in 1742 by the mathematician Christian Goldbach:

Theorem 1 *Every integer greater than five is the sum of three primes.*

The mathematician Leonhard Euler, a contemporary of Golbach, subsequently proposed the following refinement of the conjecture:

Theorem 2 *Every even integer greater than two is the sum of two primes.*

Neither Goldbach nor Euler was able to prove or disprove either conjecture; in fact, they remain open problems to this day, despite the efforts of countless mathematicians during the intervening centuries. The former version of the conjecture is today known as the *ternary Goldbach conjecture*, and the latter is known as the *strong Goldbach conjecture*. Both of these conjectures can be rather trivially converted into equivalent conjectures about programs, such as whether or not they print "Hello," which is left as an exercise for the reader. Thus, a hypothetical procedure that solves the "Hello" problem could immediately settle one of the most vexing open problems of number theory, which is to say

$$\text{Strong Goldbach Conjecture} \leq \text{The "Hello" Problem,} \tag{1.2}$$

using the same informal notation that we introduced earlier. This is further evidence that an algorithmic solution to the "Hello" problem must be extraordinarily difficult to express, if it is possible at all.

Let's consider one further number-theoretic problem. The mathematician Lothar Collatz posed an interesting problem in 1937 which has yet to be solved. The *Collatz problem*, also commonly known as the $3n + 1$ *conjecture*, involves a trivially computed sequence of natural numbers beginning with an arbitrary natural number, n_0. Given any element in the sequence, n_i, the following element, n_{i+1} is obtained by applying this rule: if n_i is even, then n_{i+1} is $n_i/2$, otherwise it is $3n_i + 1$. The resulting sequence, which behaves quite chaotically, is known as a *Collatz sequence*. The sequence is defined to terminates only when the value 1 is reached. The procedure *Collatz* shown in Figure 1.3 generates the Collatz sequence associated with any given initial value, n_0, assigning subsequent values to the variable n. For example, when $n_0 = 17$, the variable n will be assigned the following sequence of values: 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. The procedure *Collatz* terminates if and only if the given Collatz sequence is finite; that is, if and only if it eventually produces the value 1. The Collatz *conjecture* is that this procedure will always terminate, regardless of the starting value, n_0. While no counter-example has ever been discovered, the conjecture has never been proven.

```
procedure Collatz( natural $n_0$ )
    $n \leftarrow n_0$
    while $n \neq 1$ do
        if EVEN($n$)
            then $n \leftarrow n/2$
            else $n \leftarrow 3n + 1$
        endif
    endwhile
    print "Hello"
endproc
```

Figure 1.3: *Pseudo-code for generating the Collatz sequence. The famous "3n+1 conjecture" is that the above procedure will eventually print "Hello" for every value of $n_0 \geq 1$. As of the year 2007, this conjecture has neither been confirmed nor refuted.*

```
procedure TestCollatz()
    for $n_0 = 1 \ldots \infty$ do
        if Collatz($n_0$) prints "Hello"
            then do nothing
            else print "Hello"
        endif
    endfor
endproc
```

Figure 1.4: *Pseudo-code for testing the "$3n + 1$ conjecture. This procedure will eventually print "Hello" if and only if the conjecture is false. However, even if the conjecture is false, it requires that we solve the "Hello" problem for the Collatz procedure. This suggests that the Collatz conjecture may be even more difficult to resolve than the other number-theoretic problems we've considered.*

```
boolean function ResolveCollatzConjecture()
    if TestCollatz() prints "Hello"
        then return $\perp$
        else return $\top$
    endif
endproc
```

Figure 1.5: *Pseudo-code for resolving the Collatz conjecture, which depends upon solving the "Hello" problem for a problem that itself depends upon solving the "Hello" problem.*

Can the Collatz conjecture also be phrased in terms of the "Hello" problem? A difficulty arises immediately if we attempt to do so. If the conjecture is true, it implies that the Collatz sequence is finite *for all starting numbers*. If the conjecture is false, it implies that there exists an *infinite* Collatz sequence. Either case would seem to require that we verify an infinite collection of cases before we can reach a conclusion. Consequently, this problem seems to be subtly different from our previous examples in that no possible state of affairs would lead to a definitive answer in a finite number of steps. However, this problem can nevertheless be reduced to the "Hello" problem if we permit it to be "composed" with itself. That is, if we have an algorithmic solution to the "Hello" problem available as a subroutine, we can write a program whose behavior, when analyzed by such a

```
boolean function SubsetSum( list of naturals N, integer m )
1  if EMPTY(N) or m < 0 then return ⊥
2  if FIRST(N) = m then return ⊤
3  if SubsetSum( REST(N), m ) then return ⊤
4  if SubsetSum( REST(N), m − FIRST(N) ) then return ⊤
5  return ⊥
endfunc
```

Figure 1.6: *Pseudo-code for solving the subset sum problem. Given a list of natural numbers, N, this function returns ⊤ (true) if there is a subset of N whose sum is m, and returns ⊥ (false) otherwise. The double recursion (lines 3 and 4) causes this algorithm to consume exponential time in the worst case.*

procedure, provides a solution to the Collatz conjecture. The first step of this is shown in Figure 1.4, which assumes that we can solve the "Hello" problem for the Collatz function for each starting value, n_0.

Observe that there is nothing special about printing "Hello." We could have chosen practically any other identifiable behavior instead, such as reaching a specific line in the program, executing a particular type of instruction, computing a certain number, calling a given function, or failing to terminate at all owing to an infinite loop.

Now let's consider a problem of a very different nature, known as the *Subset Sum* problem. An instance of Subset Sum consists of an ordered pair (N, m), where N is a finite list of natural numbers and m is another natural number. The problem is to determine whether there is a subset of N whose sum is m. For example, the instance

$$(\ \{1, 3, 12, 12, 26, 31, 49\}, 42\ )$$

has such a solution, since $1 + 3 + 12 + 26 = 42$, while the instance

$$(\ \{7, 8, 9, 27\}, 30\ )$$

has no solution, as no subset of $\{7, 8, 9, 27\}$ sums to 30.

Unlike the previous examples that we've considered, a "yes/no" answer can be provided for every instance of this problem by means of a very straightforward procedure–at least in principle. We need only sum every combination of numbers in the list to see if any results in the given target value. The pseudo-code in Figure 1.6 does precisely this by means of a recursive search. The function is initially passed two variables; a list of zero or more natural numbers, N, and an integer m. The program will return "true," denoted by $\top$, if and only if there exists a subset of the integers in N whose sum is m, and "false," denoted by $\bot$, otherwise.

The algorithm in Figure 1.6 makes use of several elementary functions: EMPTY returns $\top$ if and only if the list has zero elements, FIRST returns the first element of a non-empty list, and REST returns a duplicate list, but with the first element removed. The function *SubsetSum* is easily implemented in virtually any programming language.[2]

[2] For example, the functions EMPTY, FIRST, and REST correspond exactly to (null N), (first N), and (rest N) in Common Lisp, and somewhat more loosely to N.empty(), N.front(), N.pop_front() in C++, if N is encoded as a list.

While the Subset Sum problem is clearly solvable by algorithmic means, the algorithm shown in Figure 1.6 is not practical, as its running time can grow exponentially with the size of the list of numbers passed to it. So, the interesting question now becomes whether or not this problem can be solved in a "reasonable" amount of time, where the word reasonable would clearly exclude any algorithm with a worst case exponential running time. As it happens, it is currently unknown whether this problem can be solved efficiently, as every algorithm for solving it has thus far had an exponential running time in the worst case; however, it has never been proven that exponential time is required. This fact alone would make the Subset Sum problem rather interesting, but there is another aspect that literally makes it one of the central problems of computer science; there are hundreds of similar problems that are also lacking efficient algorithmic solutions that are *equivalent* to the Subset Sum problem. That is, these problems *reduce* to Subset Sum in such a way that were we to find an efficient solution to it, we would immediately obtain efficient solution to all the others.

In the examples that we've explored thus far, we've seen that answering even very simple questions about the behavior of programs can be extraordinarily difficult; indeed, as we shall see, it is typically *impossible*. We have also seen an example of a problem that is solvable in principle, yet not solvable in practice due to exponential complexity. Both of these aspects will be explored in depth in the sequel. In the following section we examine our first concrete example of an uncomputable function. As this example will demonstrate, we need only use very general facts about programs, thus the proof is extremely general.

1.2 An Uncomputable Function

Fix in your mind a specific programming language, such as Java, C, or Lisp. One thing that we can say in advance about any such program is that it will consist of a finite sequence of symbols from a finite "alphabet", such as the ASCII[3] characters. We shall assume that the imagined language is rich enough to express standard arithmetic expressions and that it places no restriction on the size of integers that can be represented[4]. Let $\mathcal{P}$ denote the class of syntactically-correct programs in this language that require no input and, when run, eventually produce a single integer as output. We will not concern ourselves with the exact nature of input or output operations; we will simply think of the program as somehow "returning" an integer value when it is executed.

For any program $\mathbf{P} \in \mathcal{P}$, let $|\mathbf{P}|$ denote the number of symbols comprising it, which will always be finite in any conceivable language. We will speak of this as the "size" of the program. Using this measure, we can define the interesting mathematical function $f : \mathbb{N} \to \mathbb{N}$; that is, a function that maps the natural numbers to the natural numbers.

$$f(n) \overset{\text{def}}{=} \max \{\, \mathit{Value}(\mathbf{P}) \mid \mathbf{P} \in \mathcal{P} \ \text{ and } \ |\mathbf{P}| \leq n \,\}, \tag{1.3}$$

where $\mathit{Value}(\mathbf{P})$ denotes the value computed by the program $\mathbf{P}$, and the maximum is taken over all programs consisting of n or fewer symbols that eventually return a value. In words, *$f(n)$ is the largest value that can be returned by a program consisting of n or fewer symbols.* While this may

[3]ASCII stands for "American Standard Code for Information Interchange." It's a widely adopted seven-bit encoding of common symbols–essentially those found on a standard computer keyboard.

[4]Assuming that the language can handle integers of arbitrary size is not at all unrealistic. Some languages have this capability built in (such as Lisp), and for those that do not, it is always possible to simulate this feature through function calls.

```
**integer function** $MaxNum$( **natural** $n$ )
    **begin**
    ⋮
    Some algorithm that computes $f(n)$ and assigns the value to $k$.
    ⋮
    **return** $k$
**endfunc**

**integer function** $\mathbf{P}_1()$
    $k \leftarrow MaxNum(1)$
    **return** $k+1$
**endfunc**
⋮
**integer function** $\mathbf{P}_{1000}()$
    $k \leftarrow MaxNum(1000)$
    **return** $k+1$
**endfunc**
⋮
```

Figure 1.7: *A hypothetical function MaxNum that returns the largest possible number computable by a program (in some fixed programming language) consisting of n or fewer symbols. Also shown are two representatives of the sequence of programs $\mathbf{P}_1$, $\mathbf{P}_2$, ... that merely call MaxNum as a subroutine. The m'th program returns a value that is* greater *than that computable by any program consisting of m or fewer symbols. One of these programs leads to a contradiction.*

seem a rather peculiar definition, the function f is mathematically well-defined. For any integer n there are only finitely many programs of size n or smaller; each program either has the property that it eventually returns a value or it does not. Thus, the maximum is always over a well-defined finite set of natural numbers, albeit a set that we cannot expect to enumerate in practice. In fact, here is the surprising fact about the function f:

Theorem 3 *No program, in any language, can compute the mathematical function f.*

Proof: To prove the theorem we shall use a technique known as *proof by contradiction*. That is, we shall *assume*, for the sake of the proof, that a program capable of computing the mathematical function f actually does exist. We'll call this hypothetical program *MaxNum*, as shown in Figure 1.7. We will then demonstrate that an absurdity follows from this assumption. Specifically, we shall use the hypothetical program *MaxNum* to define a new program about which we can prove a nonsensical self-contradictory statement.

To find such a self-contradictory program, consider the sequence of programs $\mathbf{P}_1, \mathbf{P}_2, \ldots$ in the class $\mathcal{P}$ such that

$$Value(\mathbf{P}_m) \;=\; f(m)+1 \tag{1.4}$$

for $m = 1, 2, 3, \ldots$. Such functions are trivial to construct given the existence of the program *MaxNum*, which can be called as a subroutine as illustrated in pseudo-code at the bottom of Figure 1.7.

Because the value computed by $\mathbf{P}_m$ is greater than $f(m)$, and no program of m or fewer symbols can compute a number this large (by the definition of f), it follows that

$$|\mathbf{P}_m| > m \tag{1.5}$$

for all m. But we can now show that there exists an integer m for which this inequality is violated, and thereby reach a contradiction. To do this we need only examine the actual size of the program $\mathbf{P}_m$, which includes the program *MaxNum* that computes the function f.

Since the hypothetical program *MaxNum* is of fixed size, the programs $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_m$ are *almost* of fixed size as well; the only exception is the number that is passed to the function *MaxNum* in the first line of each program $\mathbf{P}_m$. Consequently, the size of the programs in this sequence must grow with m, as it requires more symbols to represent the number 1000 than it does to represent the number 1, for example[5]. However, the number of symbols needed to encode a number grows only logarithmically with the number itself, regardless of the number system used, provided its base is greater than one. Thus, we can bound the size of $\mathbf{P}_m$ as follows:

$$|\mathbf{P}_m| \leq a + b \lceil \log(m) \rceil, \tag{1.6}$$

where a and b are fixed constants, with a including the size of *MaxNum*. Regardless of what a and b are, it is always possible to find a large enough integer k so that $a + b \lceil \log(k) \rceil < k$. Therefore, for such a k, we have

$$|\mathbf{P}_k| < k, \tag{1.7}$$

in direct contradiction to Equation (1.5), which holds for all m. By virtue of this contradiction, we are forced to conclude that the hypothetical program *MaxNum* cannot actually exist, as its existence was the only assumption that we made in this chain of reasoning. □

A slightly different way to approach this proof is to first observe that the function f is necessarily monotonically increasing. That is,

$$n \leq m \implies f(n) \leq f(m). \tag{1.8}$$

With a larger budget of symbols, we can still express all the same programs as before, and possibly more. Therefore, allowing more symbols can only increase the size of the largest computable number. We then observe that

$$f(n) \; < \; \mathit{Value}(\mathbf{P}_n) \; \leq \; f(|\mathbf{P}_n|) \; = \; f(a + b\lceil \log(n) \rceil), \tag{1.9}$$

for some constants a and b, where the first inequality follows from the definition of $\mathbf{P}_n$, the second inequality follows from the definition of f, and the final equality follows from counting the symbols in $\mathbf{P}_n$, as we did above. Equation (1.9) results in a contradiction because it implies that f is non-monotonic for sufficiently large n.

[5]This assertion is false, of course, if the base of the number system happens to be greater than 1000. In this case, we may simply pick a number larger than the base to make the same point.

The above theorem actually shows that there is no mathematical expression of finite size that can produce an *upper bound* on $f(n)$ for all n. That is, no function that can be written down grows faster than f, not even Ackermann's function[6]!

It is instructive to consider why the following strategy for implementing *MaxNum* will not work. Suppose we let *MaxNum* iterate over all possible strings of length n or shorter, and attempt to "run" all of the strings as programs. Those programs that are malformed can be skipped; among those that return a value, the largest value is kept and returned as the value of *MaxNum*. Such an enumeration of strings is easy to express algorithmically (albeit requiring exponential time), and the determination of syntactic correctness as well as the running of programs can be accomplished by an interpreter, which can be written for any programming language. At first this would appear to work, even if it would not be a practical approach for even moderately large n. However, there is a serious flaw in this approach that cannot be overcome. The interpreter cannot tell whether the program it is running will eventually terminate or not. Thus, there is a fundamental connection between the function f and the classical *halting problem*, which we shall study in some depth in the sequel.

1.3 summary

We began this chapter by asking whether we might be able to determine, in advance, what an arbitrary program will do. For instance, might we be able to determine whether any given program will print the word "Hello" at some point during its execution? We observed that if indeed there were a general method by which this could be determined, then such a procedure could be immediately pressed into a different and vastly more interesting service: it could be used to solve a huge variety of deep mathematical problems. That fact alone tells us that the task of making such a determination in advance must be enormously difficult, if it is possible at all.

We then considered a different type of problem; one for which there is obviously an algorithmic solution, but for which there is no apparent *good* solution. Specifically, we saw that the "subset sum" problem can be solved by a very short and simple program; however, this program will require vast amounts of time to run even for problem instances of relatively modest size. This is the first of many such problems that we will encounter.

We also examined the problem of computing a specific well-defined mathematical function from the natural numbers to the natural numbers; the function that returns the largest number computable by a program consisting of n or fewer symbols. We discovered that such a function cannot be computed by any program, as the existence of such a program leads to a contradiction. Specifically, we showed that such a program, if it existed, could be "tricked" into computing a larger number than it should be able to compute. This is our first example of a well-defined mathematical problem that is demonstrably unsolvable by means of computation. In subsequent chapters we shall encounter many more problems that have this property, including the "Hello" problem discussed above.

[6]We will study Ackermann's function in section 2.2.3 and demonstrate that it grows at an inconceivably fast rate.

1.4 Exercises

1. Write a pseudo-code description of an algorithm that prints "Hello" if and only if the strong Goldbach conjecture is false.

2. Can you write a similar program that prints "Hello" if and only if the strong Goldbach conjecture is true?

3. Write a pseudo-code description of an algorithm that prints "Hello" if and only if there exists an odd *perfect number*. A perfect number is one that is equal to the sum of its proper divisors, such as 28, which is $1+2+4+7+14$. Whether or not an odd perfect number exists is currently an open problem.

4. Imagine that your favorite programming language had a built-in function called HALTS such that HALTS(FUN, NUM) evaluates to $\top$ if FUN would eventually terminate and return a value when run on input NUM, and returns $\perp$ otherwise; that is, if the function was malformed, or if its evaluation would run forever, or if it would result in some type of error. You can assume that the argument FUN is a string encoding a program, or any other representation that would be convenient in the language you have chosen. Write a function called SOLVE-COLLATZ that settles the "$3n + 1$ conjecture" using the hypothetical HALTS function. Assume that the programming language you have chosen accommodates arbitrarily large integers, so you needn't worry about overflow.

5. It would be a useful feature of a compiler if it could determine *at compile time*, whether the program it is compiling would eventually perform some illegal operation, such as dividing by zero, or attempt to access a non-existent part of memory (e.g. dereferencing a null pointer). Discuss the theoretical barrier to performing such an analysis in light of what we have discussed in this chapter.