Computation — Quantum and Otherwise

Robert Geroch

April 5, 2006

Contents

1 Introduction **2**

2 Characters and Strings **3**

3 Problems **5**

4 Computability **8**

5 Turing Machines **13**

6 Noncomputable Problems **21**

7 Noncomputable Numbers **26**

8 Formal Set Theory **28**

9 Probabilistic Computing **31**

10 Difficulty Functions **37**

11 Difficult Problems; Best Algorithms **41**

12 New Language **49**

13 Improving This Language **59**

14 Quantum Mechanics **62**

15 Grover Construction **64**

16 Grover Construction: Six Issues **68**
16.1 Initial State 68
16.2 Final Observation on H_{in} 69
16.3 Building the Operator W 70
16.4 Building the Operator V 72
16.5 Errors 79
16.6 What Is The Problem? 80

17 Quantum-Assisted Computing **84**

18 Quantum-Assisted Computability **90**

19 Quantum-Assisted Difficulty Functions **97**

20 Limitations on Quantum-Assisted Computing **106**

21 Quantum-Assisted Efficiency **110**

22 Non-Quantum Assistance **116**

1 Introduction

In Sects. 2 and 3 we introduce a few preliminary notions. Sects. 4-7 comprise a brief survey of the subject of computability. The key result is that there exist well-posed problems that are, nevertheless, not computable. Sect. 8 may be omitted. In Sect. 9, we introduce the idea of embedding probabilities in the computation process. Sects. 10-13 deal with computational difficulty. One key result (Blum) is to the effect that there exist essentially arbitrarily "hard" problems. We also address the issue of constructing a natural, general definition of the "difficulty" of any computation.

Sect. 15 is a self-contained, four-page summary of one example of a computation utilizing quantum mechanics. Readers wishing a quick look at this subject might read only this section. Sect. 16 is a discussion of a

number of issues raised by this example. In Sects. 17-19, we introduce the key concept — that of a quantum-assisted computation. Readers requiring neither background nor motivation may wish to start here. With a precise notion of a quantum-assisted program in hand, we are in a position to prove theorems about it. In Sect. 20 we show that the efficiency-increase from the use of quantum mechanics can never be more than exponential. In Sect. 21 we give a specific example of a problem along with a quantum-assisted program that computes it, such that it is hard to see how to write a regular program that can match its efficiency (although it is also hard to see how to prove that there is none).

2 Characters and Strings

Fix a finite set, $\mathcal{C}$, having at least two elements. This $\mathcal{C}$ will be called the *character set*; and its elements the *characters*. We shall normally introduce various symbols to denote the various elements of $\mathcal{C}$. For example, $\mathcal{C}$ might have just two elements, and these might be denoted $0, 1$. Or, $\mathcal{C}$ might consist of twenty-six elements, with these denoted $a, b, \cdots, z$; or 36 elements, denoted $a, b, \cdots, z, 0, 1, \cdots, 9$. Or, as a final example, $\mathcal{C}$ might consist of 256 elements, denoted by the various ASCII characters.

The underlying choice of character set makes no significant (i.e., no interesting) difference to anything that follows; although a poor choice can turn out to be inconvenient. Eventually (but not right now) we shall allow ourselves to be a little sloppy as to exactly what our character set is.

Fix a character set, $\mathcal{C}$. By a *string* over $\mathcal{C}$, we mean a finite, ordered list of characters. Examples of strings, for the character sets above, are: "001010", "unblowupable", "$3dafrq$", and "$\$ = log8+\} - -r\&C$", respectively. The empty list of characters, which we denote $\emptyset$, is also allowed as a string; and it is called the *empty string*. [Thus, in order to avoid confusion, we shall avoid denoting any character by "$\emptyset$".]

The set of strings, over a given character set $\mathcal{C}$, will be denoted $\mathcal{S}$ (or $\mathcal{S}_{\mathcal{C}}$, if there is a chance of confusion as to what the character set is). Note that $\mathcal{S}$ is an infinite set; in contrast to $\mathcal{C}$, which is a finite set. A critical idea in this subject is to pass to "the infinite" in a careful, controlled way. Here, that passage is taking place in the construction of $\mathcal{S}$ from $\mathcal{C}$. This is something that we carry out directly, as opposed to letting $\mathcal{C}$ already be infinite on its

own, in any old manner that it chooses.

Part of the reason why the "choice of character set makes no significant difference" is that it is possible to pass from one character set to another. For example, let $\mathcal{C} = \{0, 1\}$ and $\mathcal{C}'$ the ASCII character set. Then "writing a byte as eight bits" provides a mapping from $\mathcal{S}_{\mathcal{C}'}$ to $\mathcal{S}_{\mathcal{C}}$. For example, string "$ab$" $\in \mathcal{S}_{\mathcal{C}'}$ might be sent to the string "0000100100001010" $\in \mathcal{S}_{\mathcal{C}}$. Unfortunately, this mapping is not invertible: It is not true that every string over $\mathcal{C}$ arises in this way from *some* string over $\mathcal{C}'$. [Indeed, a string over $\mathcal{C}$ *does* arise in this way if and only if the number of characters of which it consists is divisible by eight.]

Next, fix a character set, $\mathcal{C}$; and fix also an ordering of the characters in that set (i.e., a choice of a "first" character, a "second" character, etc. through all the characters in the set). For example, $\mathcal{C}$ might be the lowercase Latin letters (the set of 26 characters, $\{a, b, \cdots, z\}$), and the ordering might be alphabetical. Having made these choices, we now construct an ordering also of the set $\mathcal{S}$ of strings over $\mathcal{C}$, in the following manner. First is the empty string, $\emptyset$; then all the one-character strings, in the ordering of the characters; then all two-character strings, in dictionary ordering; then all three-character strings, etc. So, for instance, in the example above the ordering of $\mathcal{S}$ is: $\emptyset$, "a", $\cdots$, "z", "aa", "ab", $\cdots$, "az", "ba", "bb", $\cdots$, "zz", "aaa", $\cdots$. Now assign, to these strings so ordered, successive integers, beginning with the integer 1. In this way, we set up a correspondence between the set $\mathcal{S}$ and the set $\mathcal{Z}^+$ of positive integers, In short, strings are really just positive integers, in thin disguise. Indeed, we shall allow ourselves to speak of "the n^{th} string", by which we shall mean the n^{th} string in this ordering, where some fixed ordering of the character set $\mathcal{C}$ is implicit (or explicit). When dealing with mathematical issues (such as the manipulation recursive functions), it is sometimes more convenient to stick entirely with the integers, ignoring character sets and strings altogether. But the strings seem better adapted to dealing with physical issues.

The reason that we required that the character set contain at least two elements is that, for $\mathcal{C}$ having but a single element, the length of the n^{th} string grows linearly with n (rather than logarithmically, when $\mathcal{C}$ contains two or more characters). This behavior would be inconvenient when we come to discuss difficulty.

The ordering above leads to a different way to pass from strings over one character set to those over another. First order both character sets (in any

way whatever); and then identify, for $n = 1, 2, \cdots$, the n^{th} string over the first character set with the n^{th} string over the second. Consider, for example, the character sets given by $\{0, 1\}$ and $\{a, b, \cdots, z\}$, in the indicated orderings. Then the string "1101" over the first character set would be identified, in this manner, with the string "ab" over the second character set. Note that (in contrast with the earlier method) this really is a correspondence between the two sets of strings, i.e., this mapping from $\mathcal{S}_{\mathcal{C}}$ to $\mathcal{S}_{\mathcal{C}'}$ is one-to-one and onto.

3 Problems

Fix a character set, $\mathcal{C}$. A *problem* is a mapping $\mathcal{S} \xrightarrow{\pi} \mathcal{S}$, i.e., a mapping from strings over $\mathcal{C}$ to strings over $\mathcal{C}$. Here is an example of a problem:

> **Example.** Let the character set have thirteen elements, $0, 1, \cdots, 9$, y, n, f, and let the mapping π be the following. If the string $S \in \mathcal{S}$ is an integer greater than or equal to two (i.e., if it is not $\emptyset$ or "1", does not contain the characters "y", "n", or "f", and does not have an initial character "0")[1] then let $\pi(S)$ be "y" ["yes"] if the integer S is prime, and "n" ["no"] if that integer is not prime. If the string S is not an integer greater than or equal to two, then set $\pi(S) =$ "f" ["forget it"]. This is indeed a mapping $\mathcal{S} \xrightarrow{\pi} \mathcal{S}$, and so is a problem.

Note that, in the example above, we are actually only interested in certain strings (namely, those that represent integers greater than or equal to two). But we cannot confine ourselves simply to these strings, for, by definition of a "problem", the mapping must apply to *all* strings. So we send the ones we aren't interested in to the trash ["f"]. It is convenient to set things up in this way. Suppose, for example, that we had defined a problem to be a mapping from a mere subset of $\mathcal{S}$ to $\mathcal{S}$. Then, e.g., it would be false that the composition of two problems is a problem. Even worse, we would have to confront eventually the issue of how we shall determine whether or not a

[1] In order to avoid having to say all this repeatedly, let us agree, here and hereafter, on the following definition: Over any character set that includes the digits, $0, 1, \cdots, 9$, a string will be called an *integer* if it contains only those ten characters, is not $\emptyset$, and (unless it is actually the string "0") does not have initial character "0".

given string is in the "certain subset". The present definition — requiring that the mapping π have domain all of $\mathcal{S}$ — puts the burden of sorting all this out back on the mapping π itself (where, as we shall see, it belongs).

In the example above, the problem is really the question "Is integer S prime, or is it not?" Note that we could ask essentially this same question by any number of other maps (i.e., by any number of other problems). For instance, we could eliminate "y", "n", and "f" from the character set, and then encode the answer as a digit (e.g., "0" for "prime"; "1" for "integer ≥ 2 but not prime"; and "2" for "not an integer ≥ 2"). We could also modify the input. for example, we could let the character set consist of the twenty-six lower-case Latin letters, impose alphabetical ordering, and let π ask whether, given string S (say, the n^{th} string in the induced ordering on $\mathcal{S}$) the integer n is prime. Thus, we see that a given question may give rise to many problems.

We emphasize that a problem is a *map*, and not the process by which we arrived at that map. Consider, for example, the following problem:

> **Example** Let the character set again be $0, 1, \cdots, 9, y, n, f$, and let $\pi(S)$ be "f" if S does not represent an integer greater than or equal to two, "y" if it does represent such an integer, and that integer is the largest prime, and "n" if it does represent such an integer, and that integer is not the largest prime.

This is the same *problem* (i.e., the same map) as that which sends string S to "n" if S represents an integer greater than or equal to two, and "f" otherwise. [This follows, since there is no largest prime.] But this is a very different characterization of the problem than our earlier one. Thus, we see that several apparently different questions may give rise to precisely the same problem.

Think of a problem π as a "broad question"; of a particular value for the argument S as a "specific instance" of that broad question; and of $\pi(S)$ as the answer to that question in that instance. Thus, a problem represents the answers to an infinite number of questions (since there is an infinite number of possible input strings). The prime problem above illustrates this point. But it is also possible to design a problem that answers a single question. For example, consider

> **Goldbach's Conjecture:** Every integer $n \geq 4$ is equal to a sum of two primes.

Let the character set be $0, 1, \cdots, 9, y, n, f$, as above, and let, for each string $S \in \mathcal{S}$, $\pi(S) =$ "y" if Goldbach's conjecture is true, and $\pi(S) =$ "n" if Goldbach's conjecture is false. [Note that π doesn't care what S is. No law says it must.] Since the conjecture above is, presumably, either true or false, this is indeed a problem. However, this problem is *either* the problem $\pi'(S) =$ "y" for all S; *or* the problem $\pi''(S) =$ "n" for all S. But these are both rather uninteresting problems. Thus, even though the original question ("Is Goldbach's conjecture true?") is interesting, translating it in this way into a problem yields what is guaranteed to be a pretty boring problem. Of course, to know whether π is π' or π'' would be interesting.

Here are some other candidates for problems.

1. Let the character set be $\{0, 1, \cdots, 9, f\}$, and let $\pi(S)$ be "f" if S is not a positive integer, and the S^{th} digit in the decimal expansion of the number π if S is a positive integer. This is a problem.

2. Let the character set be the same, but again let $\pi(S)$ "f" if S is not a positive integer. But if S is a positive integer, let $\pi(S)$ be a string of the form {digits of a positive integer i}f{digits of a positive integer i'}, where i/i' is within 10^{-S} of the number π. This is not a problem (since the actual mapping has not been specified). Rather, this is a description of a class of problems. There does indeed exist a problem in this class (e.g.: expand the number π decimally, as in the first example; stop as soon as you reach a rational within 10^{-S} of the number π; and finally reduce that fraction to lowest terms).

3. Let the character set be $\{0, 1, \cdots, 9, y, n, f\}$. Let $\pi(S)$ be "f" if S is not an integer ≥ 4. If S is an integer ≥ 4, let $\pi(S)$ be "y" if S is a counterexample to the Goldbach conjecture, and "n" if it is not. This is a problem. The question of whether Goldbach's conjecture is true is the question of whether or not the problem π is equal to a suitable simple problem (which always answers "f" or "n").

4. Let the character sets consist of the upper-case and lower-case Latin letters, together with an appropriate set of punctuation marks (period, comma, semicolon, question mark, exclamation mark, etc). Let $\pi(S)$ be "yes!" if S is the Gettysburg address, and "no" otherwise. This is a problem.

5. Let the character set be any ordered set that includes at least the ten digits (not necessarily in their natural order). Let π send string S over this character set to that integer n which is such that S is the n^{th} string. This is a problem.

It is interesting to note that there has taken place a progression to ever greater levels of the infinite. The character set is finite; the set of strings over that character set is countably infinite; and, finally, the set of problems on those strings is uncountably infinite. We pause to give a proof of this last assertion because it illustrates a method, called a diagonal argument, that we shall use several times later. Fix $\mathcal{C}$, and suppose, for contradiction, that we had a countable collection of problems, $\pi_1, \pi_2, \pi_3, \cdots$, that exhausted all problems on this character set. We now introduce a new problem, π, as follows. Say, one of the characters is "a". On the n^{th} string, S_n, set $\pi(S_n) = \emptyset$ if $\pi_n(S_n) =$ "a"; and $\pi(S_n) =$ "a" otherwise. Then this problem π, so constructed, is not equal to π_n for any n, since by construction $\pi(S_n) \neq \pi_n(S_n)$. Thus, the list $\pi_1, \pi_2, \cdots$ could not have been exhaustive — a contradiction.

Finally, we remark that this notion of a problem is rather robust. For virtually the entirety of the remaining discussion, we shall have before us some problem, as here defined, or other.

4 Computability

Fix a character set, $\mathcal{C}$. We next wish to introduce the notion of computability of a problem over this character set.

Roughly speaking, a problem $\mathcal{S} \xrightarrow{\pi} \mathcal{S}$ is computable provided there exists a computer that, when run with any given input string S, will ultimately halt, displaying at that point as output precisely the string $\pi(S)$. But what is a "computer"? We cannot take this to mean a physical computer, because no such computer is ever capable of solving any problem. My desktop, for

example, has a hard drive with capacity of only 10 GB. Thus, if I let the character set be, say, ASCII, and let the input string S consist, say, of 10^{11} characters, then surely this computer will be unable run with this input.

More promising would be to consider, rather than a physical computer, a computer language. Consider Fortran[2]. A given Fortran program has no space limitations whatever associated with it. You begin by purchasing some physical computer, and running the given program on it. Then if, during the course of the calculation, it turns out that there is insufficient space to complete that calculation, you will be politely invited to purchase a larger computer and rerun the program on it. So, let us call a problem $\mathcal{S} \stackrel{\pi}{\to} \mathcal{S}$ "Fortran-computable" if there exists a Fortran program with a single, initial, "Input" instruction (allowing the user to input some string, S), a single, final, "Print" instruction (allowing the program to display to the user some final string), having the following property: For any choice of the input S, the program ultimately halts (as opposed, e.g., to getting caught in an infinite loop), having printed precisely the string $\pi(S)$[3]. For instance, every example of a problem we have given so far is Fortran-computable in this sense. Indeed, one might even imagine at this point that *every* problem is Fortran-computable.

The difficulty we now face is that there are many computer languages. That is, we also have, defined in a similar way, "C-computable", "Applesoft-computable", etc. But our goal here is to capture by a general definition an abstract notion of "computable" — i.e., to isolate the general structure of the computing process itself. The danger we face is that the various types of computability, as defined here, will say more about the individual languages that gave rise to them than they do about this general structure. But anyone who is familiar two or more languages will realize that this difficulty is more one of principle than one of practice. Consider two languages, e.g., Fortran and C. You can write a Fortran-emulator in C (and, indeed, this is probably what "Fortran" really is!). That is, you can write a C program that will

[2]We shall not be concerned here with details of any specific computer languages. In particular, we take "Fortran" as a generic term, which describes languages having such commands as "Set $x = ...$", "If $(\cdots)$, go to ...", "Do, for $I = 1, n\{\cdots\}$ Next", "Print ...", "Cat x, y", etc.

[3]To make this idea into a proper definition, we would have to specify the details of the language "Fortran". We shall not do this, since this entire discussion is intended merely as motivation for what follows.

accept as input lines of Fortran code: "Set x = 7", etc. The C program will then parse each such Fortran command, figure out what the Fortran language would have done to implement that command, and then itself do precisely that. From the mere existence of such a Fortran-emulator in C, it follows that every Fortran-computable problem is also C-computable. [To see this, consider any Fortran-computable problem π. Taking the Fortran program that computes this π and applying to it our emulator, we obtain a C program that computes π.] In a similar way, we can write a C-emulator in Fortran. We conclude, then, that the Fortran-computable problems are precisely the same as the C-computable problems. A similar argument shows that all the standard languages of the computer world generate precisely the same computable problems.

> **Exercise.** Consider three languages, A, B, and C. Suppose you were given an A-emulator in B, and a B-emulator in C. Could you use these two to construct an A-emulator in C? As a second question, consider two languages, A and B. Suppose that the A-computable problems are precisely the same as the B-computable problems. Can you exploit this fact to build an A-emulator in B?

How shall we turn this intuitive discussion into mathematics? Lest you imagine that this will be an easy exercise, we now introduce two new "languages".

The first, which we might call "MiniFortran", has just two commands: "Input", which allows the user to input any string S; and "Print $\emptyset$", which causes the computer to print the empty string. There is just one MiniFortran-computable problem, namely that with $\pi(S) = \emptyset$ for every input string S. Clearly, then, there are many fewer MiniFortran-computable problems than Fortran-computable problems. The problem with MiniFortran, of course, is that it is absurdly barren. A language must have a certain degree of richness [essentially, i) the ability to store plenty of intermediate data, ii) the ability to manipulate the data, and iii) the ability to branch, in response to those data] if it is to reach the mainstream (Fortran, C, etc) of computable problems.

Our second language, "HyperFortran", contains all the commands of Fortran, together with one additional command, with the following structure: "Do, for $I = 1, \infty\{\cdots\}$ Next". Here, "$\{\cdots\}$" consists of various Fortran commands, including those that may set certain variables. The rule is that

the computer will always exit from such a command (i.e., it will never hang here), and on doing so the variables will be set as follows. Consider one variable, "x". If, in the course of the execution of this Do-loop, the variable "x" was set to some value, and did not ever change that assigned value beyond some particular iteration (i.e., beyond some particular I-value), then on exit from this command "x" is to be assigned that value. If, on the other hand, "x" changed its value an infinite number of times during the course of the Do-loop, then on exit "x" is assigned value $\emptyset$. This is, arguably, a legitimate computer command, at least in the sense that it is completely determined what is to be done in response to such a line of code. I grant that HyperFortran seems a little strange at first sight, but, absent a careful definition of the term "language", a reasonable case could, perhaps, be made that it is one. Note, incidentally, that in HyperFortran we can solve Goldbach's conjecture. We would use a program of the following form:

Do, for $I = 1, \infty$
If ($x == \emptyset$ and I is a Goldbach-counterexample) set $x = I$
Next
Print x

If Goldbach's conjecture is true, then there will be returned the empty string. If it is false, then there will be returned a counterexample to that conjecture.

We all know in our hearts that HyperFortran is unacceptable, but it is not so easy to spell out exactly why. True, you cannot run it on a physical computer – but you can't run Fortran, either; and in any case it is usually a bad idea to try to base mathematics on physical implementability. What HyperFortran does is illustrate that some care is going to be necessary in order to formulate a suitable definition of "computable".

So, to summarize, all "reasonable" computer languages (in some sense we have yet to pin down) seem to produce the same computable problems. Our challenge is to turn this intuitive idea — which is called *Church's Thesis* — into a piece of mathematics. There are at least three different strategies by which one might imagine doing this.

The first strategy begins by producing a formal definition of "reasonable language". This definition would be along the following lines. A "reasonable language" must have some commands. [Presumably, there would be just a finite number of types of commands, but, since arbitrary strings can

normally appear in certain commands, there would be within these types an infinite number of actual commands. Just like Fortran.] With each command there would be associated something to "do" (such as manipulating a string, going somewhere, etc). We would require, as part of this definition, that these commands be rich enough to allow one to do "the necessary things for computing" (i.e., store arbitrary amounts of data, manipulate data, input/output, branch), but not so rich that they do "ridiculous things" (such as "Do, for $I = 1, \infty$"). Note that we are *not* specifying any specific language here, but rather are describing by the definition the characteristics that we will demand of a language in order that we deem it "reasonable". Then given such a language, $\mathcal{L}$, we would call a problem π *$\mathcal{L}$-computable* if there exists a program in $\mathcal{L}$ that, for every input string S, eventually halts, returning exactly $\pi(S)$. Finally, (the crowning result of this strategy) we would prove the following Theorem: For $\mathcal{L}$ and $\mathcal{L}'$ any two reasonable languages as defined above, the $\mathcal{L}$-computable problems are precisely the same as the $\mathcal{L}'$-computable problems. The key to this strategy, of course, is discovering the right definition: It has to look simple and not contrived, and at the same time be just right so that the Theorem really is a theorem. I feel that it might be enlightening to carry out this strategy, but it looks like a lot of work.

The second strategy begins by noting that, since strings can be replaced by integers, each problem thereby becomes an integer-valued function on integers. We would now introduce some axioms that are intended to characterize the "computable functions". There might be a few simple ones, such as "Every constant function is computable."; and "The composition of two computable functions is computable." But then there would be some more complicated ones, requiring that certain constructions involving computable functions result in computable functions. Then, a function would be deemed computable if it arises from these axioms. This strategy has in fact been carried out: It is the subject called recursive function theory. [Recursive functions are precisely the computable (to be defined shortly) problems.] This strikes me as an elegant and promising approach. Its downside is that recursive function theory doesn't seem especially well-matched to physics — and, in particular, not to quantum mechanics. Furthermore, the subject of computational difficulty doesn't, as far as I am aware, fit in naturally with this strategy.

The third strategy consists of inventing the "simplest possible language"

that is still (barely) rich enough that that it generates the same computable problems as the real-world languages. We then take computability to mean computability in this language. This is the strategy we shall pursue, in the following section.

5 Turing Machines

Fix a character set, $\mathcal{C}$. A Turing machine, operating with this character set, has two parts.

First, there is the machine itself. It is capable of being in any of a finite list of machine states, $q_1, q_2, \cdots, q_n, q_H$. Of these $(n+1)$ states, the last one, q_H, has a special role, as we shall see shortly. These machine states serve as the RAM: The machine will store data temporarily by the choice of the particular state in which it currently resides.

Second, there is a semi-infinite tape, divided into a succession of square boxes. Thus, at the beginning of the tape there appears the first box, followed, moving along the tape, by the second, then the third, etc. There is no "final end" of the tape, i.e., there is available as much tape and as many boxes, so arranged, as might be needed. Each of these boxes may have printed within it a single character from the set $\mathcal{C}$, or the box may be blank (having no character). We denote this blank box-state by $\emptyset$ (not to be confused with the empty string). This tape serves as the hard drive: The machine will store data here on a more permanent basis for later use in the computation.

The machine also has a read/write head, which at any one moment resides over one of the squares of the tape. Thus, the complete state of this system (machine, tape, and head), at any moment, is characterized by specifying i) the internal state of the machine, ii) the characters printed on the tape, and iii) over which square the head currently resides. For example, a typical system-state might be: "machine state q_7; tape configuration '3 \$ v Z x $\emptyset$ $\emptyset$ $\emptyset$ $\cdots$'; head over the fourth square". In this configuration, the head would be read the character "Z"; and would print to the fourth square. It will turn out that that only tape-configurations of interest will be those in which all squares of the tape beyond a certain one are blank.

This machine operates by going from one system-state to the next according to certain rules that are set down in a table (the "program"). A

typical row in this table is given below:

Curr State	Curr Char	$\rightarrow$	New State	New Char	Move
q_7	Z		q_3	p	L

This row demands that, if the computer finds itself in machine state q_7, with the head reading character "Z" on the tape, then the computer is to i) change its internal state to q_3, ii) erase the character "Z" from that square on the tape and print instead character "p", and iii) move the head one square to the left (i.e., toward the beginning of the tape). Thus, this particular row in the table would send the system from the system-state given in the previous paragraph to the following system-state: "machine state q_3; tape configuration '3 \$ v p x $\emptyset$ $\emptyset$ $\emptyset$ $\cdots$'; head over the third square". The full table for our Turing machine will contain of many such rows. In each row: The first entry must be one of the machine states $q_1, \cdots, q_n$ (but *not* the state q_H); the second entry must be a character, or the blank, $\emptyset$; the third entry must be a machine state (with q_H allowed here); the fourth entry must be a character or the blank; and the fifth entry must be either "R" ("right") or "L" ("left"). Finally, the full table must contain one and only one such row for every possible choice of the first two entries. Thus, if there were ten machine states (including q_H), and the character set had six elements, then the full table would have exactly 63 ($= 9 \times 7$) rows. The Turing machine now operates in the obvious way. At each stage, it looks up its current machine-state and character-under-the-head in this table. It then reads off from the table what should be its next machine state, what the head should print on that square of the tape, and what movement the head should make (just one square, either to the right or to the left). If ever the machine finds itself in state q_H, then the machine halts (stops computing). That is why we do not allow q_H-state as the first entry of any row.

The crucial features of this design are i) that the number of internal machines states is finite, while the number of tape-squares is infinite, and ii) what the machine will do next depends *only* on on the current machine-state and the character under the head, and not on what is printed elsewhere on the tape or whether the head resides over the fourth square or the nineteenth square.

To run a Turing machine, select the input string S and print it, one

character at a time, at the beginning of the tape, leaving all the other tape-squares blank. Begin with the head over the first square and the machine in initial state q_1. Now let the machine run, step by step, for each step looking up in the table what to do next. If the machine, during the course of its running, never achieves the halt-state, q_H, then it will continue running forever. Well, that's life. If, however, it does eventually achieve q_H and halt, then we read the output string from the tape, starting from the first square and continuing until we reach the first blank square. Note that, during the running of every Turing machine, the tape always contains but a finite number of non-blank characters (although, of course, the number of such characters may, as the S runs over all possible input strings, grow without bound). The crucial feature of this operation is that the table is to be fixed, once and for all, *before* we are given the input string S.

So, a Turing machine is a sort of mini-computer — a computer reduced to its essentials. First write the program. Then, input an initial string S. The computer computes away. Either it eventually halts, presenting an output string to you; or it runs forever, never presenting anything. A problem π over a given character set is said to be *Turing-computable* if there exists a Turing machine (i.e., a choice of the number of internal states and of the table) that computes it (in particular, never halting, no matter what the input string S), as just described. To check whether you understand how a Turing machine works, try to convince yourself that you could write a Fortran-emulator of Turing. Assuming you have convinced yourself, then we may conclude (from the mere existence of such an emulator) that the Turing-computable problems is a subset of the Fortran-computable problems.

We give just one example of a Turing-computable problem. A string S is called a *palindrome* if it reads the same backwards as forwards, e.g., "$K9s4\$q\$4s9K$". Let the character set contain "y" and "n" (to make answers easier to express); and let the problem $\mathcal{S} \xrightarrow{\pi} \mathcal{S}$ be the following: $\pi(S)$ is "y" if S is a palindrome, and "n" if it is not. This problem is Turing-computable. We will not write out the full table (which would have hundreds of rows!) that demonstrates this, but rather merely indicate how the machine would work.

The machine, in initial state q_1, reads the first entry in the string: Say it is a "K". The machine then goes into a state we call p_K, (whose description is "I've just read character 'K', and I'm now going to check to see if this is also the last character"), prints $\emptyset$, and moves one square to the right. If the

head now reads anything other than $\emptyset$, the head moves another square to the right without changing anything. [That is, the table entry for "current state p_K and current character {nonblank}" requires remaining in p_K, reprinting whatever is already in that square on the tape, and then moving one square to the right.] The head thus continues moving to the right, one step at a time, until it encounters a blank square. On encountering a blank square, the machine goes into a new state we call r_K (whose description is "I'm now ready to compare the last character of the string with K"), reprints $\emptyset$, and moves one square to the left. The table entry for "current state r_K, and current character anything but K" puts the machine into a new state q_n (whose description is "This string is not a palindrome. Tough luck. I'm going to go back to the beginning now, to report that fact." We'll return later to how this is done.) The table entry for "current state r_K, and current character K" puts the machine into a state q_2 (whose description is "So far so good. I'll go back to the beginning now and get the next character."), prints $\emptyset$, and moves one square to the left. As long as the head continues to encounter nonblank squares, it continues to move leftward back over the string. [That is, the table entry for "current state q_2, current character {nonblank}" retains the state q_2, reprints whatever is already under the head, and moves one square to the left.] However, as soon as the head meets a blank square, the machine goes back to state q_1, prints $\emptyset$, and moves one square to the right. The process now starts over (but now with a shorter string, for we have just removed from the original string S its first and last characters). That is, the machine reads the current square (yielding, say, character "9") goes into state p_9, moves to the right until it encounters a blank square, goes into state r_9, carries out a comparison of "9" with the current character, goes into either state q_n or q_s, etc. Continue in this way. If, eventually, the string is exhausted, then the machine goes into state q_y (whose description is "It is a palindrome! I can't wait to to back to the beginning and deliver the good news"). [That is, the table entry for "current state q_1, current character blank" places the machine in state q_y, prints $\emptyset$, and moves one square to the left.]

The process above eventually places the machine in either the state q_n or the state q_y. How does the reporting of the news ("y" or "n") work? We want the machine states q_n and q_y to move the head to the left, for that is where the reporting must take place. But how will our Turing machine know when it has reached the leftmost square? [There will just be blanks back there, for we have now erased the initial portions of our original string S.]

One way to accomplish this is to move, initially, the entire string S up the tape a little bit, to make room for a marker at square one. Here is how to move the entire string S one square to the right. Read the first character (say, "K", again). Go to state s_K (whose description is "I'm about to move a 'K' one square to the right"), and move the head one square to the right. If, say, the next square read by the head contains the character "9" then print the "K" in this square, go to state s_9, and move another square to the right. [That is, these are the instructions for "current machine state s_K, current character 9".] Continue until you reach a blank square (i.e., to the end of the string S). Then print the last character (as determined by what s-state you happen to be in at the time), and go into a state that causes movement to the left until you encounter a $\emptyset$. You have now moved the entire initial string S one square to the right. In a similar way, we may move the initial string to the right a second square, and print anything (say, "v") in the first square of the tape. All of this would be done *before* the program of the previous paragraph. Do this, and then run that program (on the original string S, as now displaced). We next describe is how the reporting works. The state q_y will require motion of the head to the left continue as long as that head encounters only blank squares. But as soon as it encounters character "v" on the tape (i.e., as soon as it reaches the first square of the tape), it will print "y" and go to the halt state, q_H. In this way there is returned that the original string S was a palindrome. The state q_n has to work a little differently. It will cause the head to continue moving to the left as long as there is encountered nonblank characters. But, according to q_n, as soon as the head encounters a blank, the machine goes into still another another state, q_{nn} (whose description is "OK. Now all I need is to is find that "v" off to my left, to whom I must my report"). So, the table entry for "current state q_{nn}, current character v" is "Go to state q_H, print character 'n', and move one square to the right." In this way there is reported that the original string S was not a palindrome.

Well, that was exhausting. Suppose our initial character set contained m characters. Then we must introduce $3m$ machine states, for the p's, r's, and s; as well as additional nine machine states, for the various q's, including q_H. Thus, there will be a total of $(3m + 8)(m + 1)$ rows in the table. Even for just ten characters, for instance, this is a total of 418 rows! You might think it would have been easier to have the machine simply remember the string S as it passes over it the first time, and then just make a single check for

palindrome-ness when it reaches the far end of the string. But that won't work, because the machine is allowed only a finite number of internal states, and this number must be fixed fixed already in the original table — and there is no adjusting that number depending on the string S.

> **Exercise.** Let the character set be any one that includes the ten digits together with "n". Convince yourself that you could build a Turing machine that returns "n" if input string S is not an integer; and $S + 1$ if it is an integer. Convince yourself that you could build a Turing machine that, whenever the string S is two digits separated by a single "n", returns their sum; and otherwise returns "n". Convince yourself that you could build a Turing machine that no only solves the problem of the last paragraph, but cleans up the tape (i.e., removes everything but the answer-string) before reporting.

The next step in learning this subject is to play with Turing machines so as to get a feeling for what they can do. Convince yourself that you could build machines (but don't actually do it!) to solve successively harder problems. Start with easy problems, such as those of the exercise above. Then try harder ones: multiplication of integers, division of integers with remainder, deciding whether or not an integer is prime, deciding whether or not an integer is a counterexample to the Goldbach conjecture, etc. Through this process, you must eventually convince yourself of the following key fact: **There exists a Fortran-emulator in Turing.** [See, e.g., [11] for some details.] As a consequence of this fact, the Turing-computable problems are the same as the Fortran-computable problems; and, by similar arguments, with the C-computable problems, with the Applesoft-computable problems, etc. The idea, then, is that the Turing-language is the simplest one that still has sufficient richness that it generates the "right" computable problems.

Here is our main definition: A problem π, over a given character set $\mathcal{C}$, is said to be *computable* if it is Turing-computable — that is, if there exists a Turing computer T that, run on any string $S \in \mathcal{S}$, always eventually halts, returning $\pi(S)$. What you have done in the paragraph above should convince you that this is a reasonable definition. What most people do in this subject, I believe, is "talk in terms of Turing machines, but think in terms of their favorite language". We emphasize that, while the psychological situation

here is complex, the mathematical one is not: We define a Turing machine; and, using it, we define a computable problem.

Every problem we have discussed so far is computable (including the one that sends every string to "y" if Goldbach's conjecture is true; and every string to "n" if that conjecture is false). The composition of two computable problems is computable. [This fact is useful in showing that suitable changes in the input/output grammar do not affect computability.] For π and π' computable problems, the problem π'' with $\pi''(S) = \pi(S)\ \pi'(S)$ (concatenation of strings on the right) is computable.

Many constructions involving Turing machines rest on the following fact: Every Turing machine can be represented as a string. Here is one way to do this. Consider a Turing machine T over character set $\mathcal{C}$. Let, for example, the first few rows of the table for T be:

Curr State	Curr Char	$\rightarrow$	New State	New Char	Move
q_7	Z		q_H	p	L
q_{11}	$\emptyset$		q_7	\$	R
q_8	2		q_8	Z	R

The first step in rewriting this T as a string is to introduce the new character set, $\mathcal{C}'$, that results from adding one additional character, say "$*$", to $\mathcal{C}$. [We are assuming here that "$*$" does not denote any element of the original character set $\mathcal{C}$ itself. This element "$*$" will serve as a marker. More on this later.] The next step is to choose a string over $\mathcal{C}$ to represent each machine state for the Turing machine T. For example, we might represent states q_7, q_H, q_{11}, and q_8 by strings "$s6$", "\$$B4$", "$uU$", and "8", respectively. Then the rows of the table above would be represented by a string as follows:

$$*s6*Z*\$B4*p**uU**s6*\$***8*2*8*Z***\cdots \qquad (1)$$

We have simply written the entries in the table (replacing each machine state by its string), row by row, one after another, using the "$*$" to separate the entries. The reading or writing of a blank square is indicated by placing nothing between the two separators: "$**$". Movement of the head to the left is indicated by placing nothing between the separators ("$**$"); to the right by a "$*$" between them ("$***$").

Thus, each Turing machine over $\mathcal{C}$ gives rise to some string over $\mathcal{C}' = \mathcal{C} \cup \{*\}$. The machine for the palindrome problem with ten characters, for

example, results in a string of about 4,800 characters. Note that a given Turing machine can be represented by a string in many ways — e.g., by choosing different strings to represent the various machine states, or by changing the order in which the rows of the table are presented.

The time has come to simplify our language a little bit. In Sect. 3, we introduced a problem π, on character set $\{0, 1, \cdots, 9, f, y, n\}$; with $\pi(S)$ equal to "f", "y", or "n" according as S is not an integer greater than equal to two, is a prime integer, or is a nonprime integer. We shall now allow ourselves to describe this as "the problem of deciding whether or not an integer is prime". Thus, in this description, it is understood that i) the character set has sufficient characters to describe the input strings of interest (i.e., here, the digits), ii) any strings constructed from those or other characters, that are not the strings of interest (e.g., here, "007"), will be suitably branded by the problem (e.g., sent to "f"), and iii) the outcomes of interest (here, "prime" and "not prime") will be suitably encoded as strings over our character set. We can safely ignore how such details are arranged, and thereby avoid an unnecessary distraction. Next, recall, from the previous paragraph, that a Turing machine over character set $\mathcal{C}$ can be represented as a string over the character set $\mathcal{C}' = \mathcal{C} \cup \{*\}$, the extra character "$*$" having been introduced as a marker. Now choose an ordering for $\mathcal{C}'$, thus obtaining, as we noted earlier, an ordering for the strings over $\mathcal{C}'$, and thereby an assignment of an integer to each such string. Combining these two constructions, then, we assign, to each Turing machine over $\mathcal{C}$, an integer (although, of course, not every integer arises from some Turing machine). Here is a somewhat more useful assignment. Consider the first string over $\mathcal{C}'$ (in this ordering) that represents a Turing machine, and call that machine number one; then consider the second string that represents a Turing machine, and call that machine number two; then the third; etc. In this way, we assign to each Turing machine as an integer, such that now each integer also represents some Turing machine. Thus, we may speak of "the n^{th} Turing machine", implicitly invoking this numbering. Next, we may combine this construction with our correspondence between strings over (the now ordered) $\mathcal{C}$ and integers. There results an assignment, to each Turing machine over $\mathcal{C}$, of a string over this same character set; such that each string now represents some Turing machine. We denote by T_S the Turing machine associated with string S. Shortly, we will want to turn a pair, such as (T, S), where T is a Turing machine over $\mathcal{C}$ and S a string over $\mathcal{C}$, into a single string over $\mathcal{C}$. We may

do this, e.g., in the following manner. First take the string over $\mathcal{C}' = \mathcal{C} \cup \{*\}$ that represents T (as above), then append "$*****$" (a marker, to separate the representation of T from S), and finally appending the string S. In this way, we represent (T, S) as a string over $\mathcal{C}'$. But now we may convert this to an integer — or to a string over $\mathcal{C}$ — using the techniques above. If you find yourself uncomfortable with all these conventions, you might try to restore the missing material for a short while, until you get used to them.

> **Exercise.** Convince yourself that each of the following problems is computable: i) that of deciding whether or not a string over $\mathcal{C}'$ represents *some* Turing machine; ii) that of deciding whether or not two strings represent the same Turing machine (where by "same" we mean "differing only in rearrangement of the machine states (preserving q_1 and q_H), and in the order in which the rows are presented in the table"); iii) that which sends integer S to the string for the S^{th} Turing machine; iv) that which sends integer S to the string for the S^{th} Turing machine, eliminating repetitions (via "same"); iv) any problem π such that $\pi(S) = \emptyset$ for all but at most a finite number of strings v) the problem that assigns, to each string S, the positive integer that is the number of steps Turing machine T takes, on string S, before it halts; where T is some fixed Turing machine that does halt for every input string. Much more difficult, e.g., is the problem of deciding whether two Turing machines compute the same problem (or, indeed, whether a given Turing machine T computes any problem at all, i.e., whether that machine it always halts, no matter what the input string).

6 Noncomputable Problems

It is not hard to convince yourself that every problem is computable. A problem, after all, is merely a mapping $\mathcal{S} \xrightarrow{\pi} \mathcal{S}$. So, to specify a problem, you must specify what the mapping is; i.e., specify how to determine, for any string $S \in \mathcal{S}$, some string, $\pi(S)$; i.e., specify how to compute, given any S, some $\pi(S)$. But "compute", we've come to realize, means "Turing-compute".

But, while this intuitive argument may seem plausible, it is simply wrong: There do indeed exist non-computable problems. The easiest way to prove

this is by a cardinality argument. The set of all Turing machines that compute problems is countable (since it is a subset of the set of all Turing machines; which in turn can be represented as a subset of the (countable) set of strings over some character set). But the set of all problems, as we saw in Sect. 3, is uncountably infinite. Therefore, the mapping "send machine to the problem it computes" from the former to the latter cannot be onto.

While the above proof is simple, it doesn't give much insight into which problems are noncomputable and which are not. Fortunately, it turns out that there is an example that is both simple and illuminating.

> The **halting problem** is that mapping $\mathcal{S} \stackrel{\pi}{\rightarrow} \mathcal{S}$ that sends Turing machine T and string S to "halt" if the machine T, running on input string S, eventually halts, and to "not halt" if that machine on that string continues running indefinitely without halting.

Note that the halting problem is indeed a problem, for, given machine T and string S, then T on S either halts, or it does not. You might imagine that we could build a master Turing machine, **H**, that would compute the halting problem, in the following manner: Given (T, S), where T is some Turing machine and S some string, **H** would merely mimic the action of T on S, doing what T would do, step by step, and ultimately reporting the result: "halt" or "not halt". But, unfortunately, this doesn't work. There is no difficulty if T, applied to S, ultimately halts. Then **H** will discover this eventually, and duly report "halt". But what if T, applied to S, never halts? There will in this case never be a moment when **H** discovers this fact; and so no moment when **H** will report "not halt".

Now comes the central result of this subject:

Theorem. The halting problem is not computable.

Proof: For S any string, denote by T_S the Turing machine represented by that string, as described above. Suppose, for contradiction, that there existed a Turing machine, **H**, that computes the halting problem, reporting $\mathbf{H}(T, S) =$ "halt" or $\mathbf{H}(T, S) =$ "not halt", according as machine T, applied to string S, halts or not. We now construct a new Turing machine, $\tilde{T}$, as follows. Given string S, $\tilde{T}$ first runs machine **H** on (T_S, S), and then proceeds as follows; If $\mathbf{H}(T_S, S) =$ "halt", then $\tilde{T}$ continues running, without ever halting; while if

$\mathbf{H}(T_S, S)$ = "not halt", then $\tilde{T}$ immediately halts. [In other words, we build a Turing machine $\tilde{T}$ that, given string S, asks $\mathbf{H}$ about (T_S, S), and then *does* the opposite of what $\mathbf{H}$ *reports*.] Now, $\tilde{T}$ is a Turing machine, and so it is represented by some string: $\tilde{T} = T_{\tilde{S}}$, for some $\tilde{S}$. We now ask: What happens when machine $\tilde{T}$ is run on string $\tilde{S}$? Suppose, say, that it eventually halts. But this means, from the way we defined machine $\tilde{T}$, that $\mathbf{H}(T_{\tilde{S}}, \tilde{S})$ = "not halt". But this means, from the defining property of $\mathbf{H}$ that machine $T_{\tilde{S}}$ $(= \tilde{T})$, when run on string $\tilde{S}$, does not halt. This is a contradiction. Similarly, the supposition that machine $\tilde{T}$, run on string $\tilde{S}$, does not halt leads to a contradiction. We thus conclude, since the assumption that there exists a Turing machine $\mathbf{H}$ that computes the halting problem leads to a contradiction, that the halting problem is not computable. \

This proof — essentially, a diagonal argument — is at the same time very simple and very confusing. I urge you to return to it in the coming weeks, as often as necessary, until you have mastered it. The discussion below is intended to give you a feeling for what the theorem means.

Note that the theorem does *not* assert that there is a *specific* machine T and string S such that we will be unable to decide whether that T, run on that S, halts. Indeed, we expect that, given (T, S), we could, given enough time and ingenuity, determine whether halting occurs. What the theorem *does* assert is that there is no single algorithm that will correctly decide halting in *every* case, i.e., for *every* (T, S).

Here is a more poignant restatement of the paragraph above. Imagine having the following job: Occasionally, there is brought to you a Turing machine, T, and string, S, and you are to determine and report to your boss whether or not that machine, applied to that string, ever halts. In some cases — e.g., a machine for which q_H never appears in the third column of the table; or for which all states in the third column are q_H — your decision will take but a few minutes. In other cases — e.g., that in which there is a collection of machine states i) from which the machine cannot exit, ii) such that q_H does not appear in the third column for any of these states, and iii) into which the machine, by virtue of the given S, will enter — it may take take hours. More complicated cases it might take days . . . or even years. As you continue working in this job, you will build a repertoire of arguments for settling this question in specific cases. And you will note that you are continually adding new, ever more clever, arguments to your collection. At some point, you may ask yourself: "Will this job ever become routine? Will

I ever reach the point at which I have developed all the arguments that are needed to solve these puzzles — the point at which no further originality will be required for this job?" These questions are answered by the theorem above: The answers are all "No".

Suppose for a moment that we had felt inclined to include use of the additional command "Do, for $I = 1, \infty$ $\{\cdots\}$ Next" (i.e., the infinite Do-loop) in our notion of "computable". As a result, as we have noted, there would be more computable problems. However, we could still define the halting problem (now referring to Turing machines in which this additional command is allowed). But the theorem above would still hold in this case (for its proof would go through in the same way). In other words, we would conclude that, even in this stronger language, we cannot compute the halting problem for that language.

Next, suppose for a moment that we had a master Turing machine **H** that did compute the halting problem. Then, we claim, we could resolve the Goldbach conjecture. We do this as follows. Construct a Turing machine T that, applied to any string S, ignores S completely, and starts searching the even integers $(4, 6, \cdots)$ looking for a Goldbach-counterexample. If it finds a counterexample, it halts, announcing this result. As long as T hasn't yet found a counterexample, it just keeps looking. Now, all we have to do, having built this machine T, is run the master machine **H** on machine T and any string S. If the result is $\mathbf{H}(T, S) =$ "halt", then the Goldbach conjecture is false; if "not halt", true. Note that we settle this conjecture without doing any real work: We don't have to have deep thoughts about the structure of the primes, or about any other relevant mathematics. All we need in order to resolve the conjecture is, essentially, an understanding of what it is asking for. In a similar way, we could use **H** to resolve, again without doing any real work, many of the other open questions in mathematics. In short, a great deal of mathematics can be encoded into the question of whether certain Turing machines halt. Perhaps this observation makes the theorem seem less surprising.

One occasionally reads, in the Sunday supplement, an article suggesting that physics is dead: that we have now discovered the fundamental structure of Nature — the "theory of everything" — and that all that is remains is working out the details. Of course, this is a mere guess on the part of the writer: Nobody has (or can have?) any real insight into this question. But note that mathematics is very different from physics in this regard.

Mathematics isn't dead yet; and, we suggest, never will be. Indeed, we have a *theorem* to the effect that new and different insights will always be required in the development of mathematics!

> Show that the following problems are not computable: i) the problem that asks whether a given Turing machine solves *some* problem; ii) the problem that asks whether, given a Turing machine, there is *some* string S on which it halts; iii) the problem of deciding whether two Turing machines (both of which do solve some problem) solve the same problem. [Hint: Show that a Turing machine that computes these problems could be reconfigured to give a Turing machine that computes the halting problem.]

This paragraph is mere whimsy, which you should feel free to ignore. I'd like to suggest that the expression "X never happens" (and its various siblings) has no real meaning whatever. Rather, it is a mere sociological convention that, when we hear that expression, we nod our heads knowingly (rather than, say, rolling our eyes). Wolves, for example, do not use this expression at all, and yet they get along, in the woods, quite well. Imagine a skeptic, who has been raised by wolves, and shares *their* sociology. You wish to explain to this person that "This Turing machine, on this string, never halts." The skeptic replies "I have no idea what you are talking about." You say "Well, it doesn't halt after 9 steps." "Right." "It doesn't halt after 137 steps." "Right." "And, in fact, it doesn't halt after any number of steps." "I have no idea what you are talking about." Or, you might try to argue using the structure of a particular Turing machine T. "The state q_H nowhere appears as the third entry in any row." "Right." "Therefore, the machine doesn't halt after 19 steps, because it couldn't be in the state q_H then." "Right." "Similarly, it doesn't halt after n steps, for any $n = 1, 2, \cdots$." "I have no idea what you are talking about." Your growing sense of frustration arises from your inability to express this idea in terms of anything else. Imagine that you were transported to another planet, the residents of which try to explain to you their term, "swerm". You are now on the other side of a similar conversation. They say "Horses are brown." "Right." "And three is an integer." "Right." "And swerm." "I have no idea what you are talking about." The residents of this planet study Turing machines, and they introduce the **swerm problem**: Given string S it returns "swerm" or "not

swerm" according to whether or not that string is swerm. [You, of course, recognize this as, not a problem at all, but just nonsense-talk.] Then, they prove a theorem: The swerm problem is not computable. You look through their so-called "proof", and discover that they use the concept of swerm in the proof itself! At worst, the residents of this planet are delusional. At best, they have managed to discover that Turing machines cannot account for their strange sociology. It is fun to reread the last several pages, mentally substituting, everywhere, "swerm" for "halt".

7 Noncomputable Numbers

As an example of an application of Turing machines, we now consider briefly the subject of noncomputable numbers.

A positive real number x is said to be *computable* if there exists a Turing machine that, when applied to any positive integer S as input, returns some rational number, a/b such that $|x - a/b| \leq 1/S$. In other words, the computable numbers are those to which we may compute approximations. Note that the two integers a and b in the fraction must be encoded into a single string in the output (e.g., by using a separator, and then translating back to the original character set). The reason that we approximate x by rationals is that it is easy to express a rational number in terms of a string. Note also that many Turing machines may compute the same number x (e.g., by providing different rational approximations to it). And finally, note that the function "$1/S$" on the right of the equation above could as well be replaced by any (computable) function of S that decreases monotonically to zero, e.g., $1/S^7$, or e^{-S}, resulting in the same notion of computable number: You can easily retrofit a Turing machine designed for one function on the right to one designed for another. The problem of whether a given Turing machine "computes" some real number x, in this sense, as well as that of deciding whether two compute the same number, is not computable.

> *Exercise.* Call a number x *hypercomputable* if there exists a Turing machine that, given integer S as input, returns a Turing machine that computes some real number y with $|x - y| \leq 1/S$. Clearly, every computable number is hypercomputable. Is every hypercomputable number computable?

The number e, for example, is computable. An appropriate Turing machine might use the formula $e = 1 + 1/1! + 1/2! + \cdots$, keeping enough terms and computing the terms with sufficient accuracy to determine a rational within $1/S$ of e. Similarly, the number $\log(\sin^{-1}(.714) + \sinh(e/4))/\pi^{2.7}$ is computable, as is every other other number you might think of offhand. Note that whether or not a number is computable depends only on the number itself, and not how that number is expressed. Thus, every rational number is computable, as is that number that is "1" if Goldbach's conjecture is true; and "0" if it is false. Indeed, it is tempting to imagine that every number might be computable. But, there do indeed exist noncomputable numbers, as follows immediately by a cardinality argument: The set of real numbers is uncountably infinite, while the set of Turing machines is only countable infinite.

Again, as with the case of noncomputable problems, we would like, not merely an existence argument, but a "concrete" example. Here is one. Set

$$c = \sum_{n=1}^{\infty} a_n/3^n, \tag{2}$$

where $a_n = 2$ if, for the n^{th} pair (T, S), the result of running the Turing machine T on the string S halts; and $a_n = 0$ if that machine on that string does not halt. Note that, since each Turing machine T on each string S either halts or does not halt, this c is a perfectly definite number. If you know this c to sufficient accuracy, then you know whether each of the first n Turing machines/strings halts. Indeed, either $c < 1/3$ (the case in which the first Turing machine/string does not halt), or $c > 2/3$ (the case in which it does halt); so knowing c within $(1/6)$ determines whether or not the first machine/string halts. Similarly, knowing c to within $1/(2 * 3^n)$ determines whether each of the first n machine/strings halt. It follows from these remarks that the number c is not computable. Indeed, suppose we were given Turing machine, $\tilde{T}$, that computes c, in the sense described above. Then, we could easily rebuild that machine into one that computes the halting problem, as follows: If you wish to know whether the n^{th} pair (T, S) halts, apply this $\tilde{T}$ to string $2 * 3^n$ (written out as its digits), and interpret the rational number that results. But we know that the halting problem is not computable, and so no such machine $\tilde{T}$ exists. Here is a curious corollary of these observations: The number c above is not rational. Note that this is not at all obvious from the formula (2).

Exercise. Show that there exists a Turing machine that accepts as input a positive integer S, returning a rational a/b, such that the resulting sequence of rational numbers increases monotonically and converges to c from below. Prove that if number x is such that there exists a Turing-generated monotonic sequence of rationals converging to it from below (in the sense of the previous sentence), and also one from above, then x is computable. Does there exist a non-computable number such that there exists neither such a sequence from above nor one from below?

It is interesting to speculate what might happen if ever a physical theory were to predict, for the outcome of some experiment, a noncomputable number, e.g., the c above. Then, since c really is a number, the theory would be making a perfectly definite prediction for the outcome of the experiment. However, to evaluate that predicted number, to higher and higher precision, would require new and ever more sophisticated insights. Thus, we might some day reach the situation in which the experimentalists, who have carried out the experiment to, say, one part in 10^7, are way ahead of the theoreticians, who have only been able to carry out the computation of what the theory predicts to one part in 10^5! And there would be no guarantee that any greater precision would be forthcoming from the theoreticians any time soon. This speculation is not entirely idle, for there are some (very weak) indications that noncomputable numbers may actually arise in some future quantum theory of gravity.

8 Formal Set Theory

The most famous application of computability is to a certain program for formalizing mathematics. We here merely touch on a few highlights of this subject: For more details, see, e.g., ([6]). Nothing in this section will be used later, so it may be skipped on first (in fact, on every) reading.

The key idea is to think of the expressions of set theory as mere strings of meaningless symbols, to be manipulated according to certain rules; and to ignore any preconceived idea that those strings have anything to do with "Truth". This is more easily said than done.

Fix a character set, $\mathcal{C}$. We next introduce a new character set $\tilde{\mathcal{C}}$, consisting of the characters in $\mathcal{C}$, together with the following ten additional characters:

$=$, $\in$, $\neg$, $\wedge$, $\forall$, $\}$, $\{$, $:$, $)$, and $($. Next, we introduce a certain collection of strings over $\tilde{\mathcal{C}}$, called the *formulae*. The rules are the following: i) For x and y any nonempty strings over $\mathcal{C}$, each of "$x = y$" and "$x \in y$" is a formula. ii) For $\mathcal{A}$ and $\mathcal{B}$ any formulae, each of "$\neg\mathcal{A}$" and "$(\mathcal{A} \wedge \mathcal{B})$" is a formula. iii) For $\mathcal{A}$ any formula, and x any nonempty $\mathcal{C}$-string, "$\forall x(\mathcal{A})$" is a formula. iv) The two expressions in item i) also result in formulae if either or both of x and y is instead replaced by a $\tilde{\mathcal{C}}$ string of the form "$\{z : \mathcal{A}\}$", where z is any nonempty $\mathcal{C}$-string and $\mathcal{A}$ is any formula. Using these rules, we may generate an enormous number of formulae, e.g., "$\forall x((y \in x \wedge \neg\forall s(z = y)) \wedge z \in \{w : x \in w\})$". A crucial fact about this construction is that the problem of whether or not a string is a formula is computable.

The nonempty strings over $\mathcal{C}$ are called *classes*. We also give these new symbols suggestive names: "$=$" is called "equals"; "$\in$" is called "is an element of"; "$\neg$" is called "not"; "$\wedge$" is called "and"; "$\forall$" is called "for all"; and "$\{z : \mathcal{A}\}$" is called "the collection of all sets z such that $\mathcal{A}$". The purpose of these names is merely to make the strings easier to remember and think about: The names are not to be construed as bestowing any "meaning".

A *definition* merely introduces a new symbol to stand for a certain $\tilde{\mathcal{C}}$-string. Here are a few examples of useful definitions (and their names): "$\mathcal{A} \vee \mathcal{B}$" stands for "$\neg(\neg\mathcal{A} \wedge \neg\mathcal{B})$" ("or"); "$\mathcal{A} \Rightarrow \mathcal{B}$" stands for "$\neg\mathcal{A} \vee \mathcal{B}$" ("implies"); "$\exists x(\mathcal{A})$" stands for "$\neg\forall x(\neg\mathcal{A})$" ("there exists an x such that"); "$x \cup y$" stands for "$\{z : z \in x \vee z \in y\}$" ("union"); "$x \subset y$" stands for "$z \in x \Rightarrow z \in y$" ("subset"); "$\emptyset$" stands for "$\{z : \neg z = z\}$" ("empty set"); "$\{x\}$" stands for "$\{z : z = x\}$" ("set whose only element is x"); The integers are now defined as follows: $0 = \emptyset$, $1 = 0 \cup \{0\}$, $2 = 1 \cup \{1\}$; etc. Thus, for example, 5 is the set with precisely these five elements: $0, 1, 2, 3$, and 4. There is also a definition (which we shall not give) of a set ω which deserves to be called the *integers*. Note that we cannot, e.g., merely write "$\omega = \{0, 1, 2, \cdots\}$", for neither "," nor "$\cdots$" are allowed symbols. We emphasize that these various definitions add nothing whatever to the logical structure: Their only role is to provide a convenient shorthand for writing long strings.

The next step is to isolate a certain collection of formula, called the *axioms*. We shall not attempt to write out any axiom system (of which there are several) in detail, but rather merely indicate what those systems look like. Typical axioms might include "$\neg\neg\mathcal{A} \Rightarrow \mathcal{A}$" and "$\mathcal{A} \wedge \mathcal{B} \Rightarrow \mathcal{A}$" (logical axioms); "$x = y \Rightarrow \forall z(z \in x \Rightarrow z \in y)$" and "$\forall z(z \in \{z : \mathcal{A}\} \Rightarrow \mathcal{A})$" (tying "$=$" and "$\{z : \cdots\}$ in with "$\in$"); "$\forall x((\neg x = \emptyset) \Rightarrow (\exists y(y \in x \wedge x \cap y = \emptyset)))$"

(which will, among other things guarantee that no class is an element of itself) and "$\exists x(\exists y(\emptyset \in x \wedge \forall z(z \in x \Rightarrow z \cup \{z\} \in x) \wedge x \in y))$" (which will, essentially, guarantee the existence of "infinite sets"). Other candidates for axioms might include strings that reflect the axiom of choice, the axiom of the excluded middle, the axiom that every subset of $[0, 1]$ is measurable, etc. the crucial thing about the axioms systems is that they are such that the problem of deciding whether or not a formula is an axiom is computable.

A **proof** is a finite ordered list of formulae, each of which is either i) an axiom, or ii) a formula $\mathcal{A}$, such that both "$\mathcal{B}$" and "$\mathcal{B} \Rightarrow \mathcal{A}$", for some formula $\mathcal{B}$, appear earlier in that list. A formula is a **theorem** if it is the last formula of some proof. Note that the **proofs** and **theorems** are both merely meaningless strings of symbols constructed in a certain way. They are not to be confused with the proofs the theorems of (informal) mathematics (which we think of as saying that "something is true"). Note that the problem of deciding whether or not a string is a **proof** is computable.

Let there be given some axiom system. Then there exists a Turing machine that, given any integer S as input, will write out a **theorem**; such that every **theorem** is included in this list. [The machine simply tries strings over $\tilde{\mathcal{C}}$ one at a time, checking for those that are **proofs**.] That is, we can "mechanically generate all **theorems**". On the other hand, it is by no means clear that the problem of whether or not a formula is a **theorem** is computable (since we cannot guarantee that a Turing machine that looks for **proofs** will ever halt). The Godel incompleteness theorem states that, for every such axiom system, one of two things is true. First, the system could be *inconsistent*. This means that there is some formula $\mathcal{A}$ such that both "$\mathcal{A}$" and "$\neg\mathcal{A}$" are **theorems**. Whenever this occurs, then every formula becomes a **theorem**. Second, the system could be *incomplete*. This means that there is some formula $\mathcal{A}$ that is closed (i.e., is such that every free variable is subject to a "$\forall$"), and is such that neither "$\mathcal{A}$" nor "$\neg\mathcal{A}$" is a **theorem**. When this occurs, we could, of course, always add one of these to get a new axiom system; but then the theorem guarantees inconsistency or incompleteness of that new system. The proof of the incompleteness theorem is like that that the halting problem is not computable. The crucial step is that the statements "there exists a **proof** of $\mathcal{A}$" and "there does not exist a **proof** of $\mathcal{A}$" can, using the set ω of integers, be reflected as formulae in the formal system (just as the crucial step in the halting problem is that Turing machines can query Turing machines).

9 Probabilistic Computing

As a prerequisite to our study of quantum-assisted computing, we consider here briefly the case of an ordinary computer that has access to a "random number generator". That is, we consider computing utilizing probabilities. These considerations will allow us to separate effects due to the full structure of quantum mechanics from those arising solely from its probabilistic aspects.

By a *probabilistic Turing machine*, we mean an ordinary Turing machine — a finite number of internal states, including q_H; a semi-infinite tape divided into squares into each of which a character may be printed; a head that, at any stage, is over one of the squares of the tape; and a table giving, for each current internal state and character (under the head), the new internal state, which new character to print on the tape, and whether to move one square to the right or left — with just one modification. Whereas in the original definition, we required that, for each current-state/current-character combination, there be exactly one row in the table, we now require instead that there be *at least* one row: There may now be more than one. Thus, for a probabilistic Turing machine, the entries of the table for current internal state "q_7" and character "Z" might read:

Curr State	Curr Char	→	New State	New Char	Move
q_7	Z		q_H	p	L
q_7	Z		q_7	$\emptyset$	R
q_7	Z		q_8	Z	R

The machine now operates as before. We begin with the initial string, say S^Ω, written on the tape, followed by blanks; with the machine in initial state q_1; and with the head over the first square on the tape. The machine now proceeds one step at a time, just as before: At each step, the machine looks up its current state and current character in the table — there finding the new state, the new character, and the required movement. But on reaching a *probabilistic step* — on facing several rows in the table corresponding to the current state and current character — the machine proceeds as follows. It chooses a particular one from among the possible actions, assigning to them equal probabilities. Thus for example, if the machine found itself in state q_7, with character Z under the head, then, consulting the table above, it would either go to state q_H, print p, and move one square to the left (1/3 of the

time); go to state q_7, print $\emptyset$, and move one square to the right (1/3 of the time); or go to state q_8, print Z, and move one square to the right (1/3 of the time). Ultimately, such a probabilistic Turing machine will either halt, at which point we may read the output string off the tape as before; or it will never halt. Thus, the final outcome of of running a probabilistic Turing machine is indistinguishable from that of running a regular Turing machine. The only difference is that, in the case of the former, which outcome will actually occur is not "determined".

We note that the formulation above is not absolutely "minimalist", in the spirit of Turing machines. For example we probably could, without loss of computing power, get by with i) all probabilistic choices based on a simple coin flip (i.e., on just two, equally likely, choices); and ii) requiring that the two corresponding rows be identical but for their third entries (i.e., that the probabilities apply only to the choice of new internal state, and not to which character is printed nor to which way to move the head). In a similar way, we could introduce probabilistic behavior into other languages. We do not explore these refinements because our purpose here is not to develop this subject fully, but rather merely to get a feeling for what happens when "computing" meets "probability".

Fix a character set $\mathcal{C}$, and denote by $\tilde{\mathcal{S}}$ the set consisting of all strings over $\mathcal{C}$, together with one additional element "$*$", which we call "not halt". Then the final result, of running a probabilistic Turing machine T^Ω over $\mathcal{C}$ with some initial string S^Ω, is some element of this set $\tilde{\mathcal{S}}$. But different runs, of course, may produce different elements of $\tilde{\mathcal{S}}$. There is some probability distribution on the set $\tilde{\mathcal{S}}$, giving the probabilities for the various outcomes. That is, for each $x \in \tilde{\mathcal{S}}$, we have a nonnegative number $p(x)$, called the "probability of outcome x", and these satisfy $\sum_{x \in \tilde{\mathcal{S}}} p(x) = 1$. We say that an outcome $x \in \tilde{\mathcal{S}}$ is *possible* if there exists a sequence of machine-tape configurations, through which T^Ω, applied to initial string S^Ω, could possibly pass (by some choice of which action to take at each probability-step), such that this sequence produces outcome x (printing some specific string, or not halting, as the case may be). The following example will illustrate these ideas.

> **Example.** Let T^Ω be the probabilistic Turing machine that, applied to initial string $\emptyset$, first flips a coin. If the coin is "heads", the machine reports the total number of coin-flips it has carried

out, and halts. If the coin is "tails", the machine flips the coin again, and repeats the process. The possible outcomes in this example are the positive integers, together with "$*$". The probability distribution is: For n a positive integer, $p(n) = 2^{-n}$; and $p(*) = 0$.

This example shows that "possible" is different from "having nonzero probability", at least for the particular outcome "$*$". However, this phenomenon occurs only for this special outcome: We claim that, for $x \neq *$, $p(x) > 0$ if and only if x is possible. The "only if" is immediate. For "if", let $x \neq *$ be a possible outcome. This means that there exists a sequence of machine-tape configurations, for T^Ω acting on initial string S^Ω, that ends up with the machine halting with "x" printed on the tape. There must be only a finite number of configurations in this sequence (since the sequence ends up somewhere), and so a finite number of passages through probability-steps. Let r denote the (rational) number that results from multiplying the probabilities associated with each of these probability-steps. Then, clearly, $p(x) \geq r > 0$.

Let us now consider briefly the simplest case: A probabilistic Turing machine T^Ω and initial string S^Ω such that the outcome "$*$" is not possible. We claim that, in this special case, there is but a finite number of possible outcomes. To see this, call a machine-tape configuration *rich* if there is an infinite number of possible outcomes starting from that machine-tape configuration. Suppose, for contradiction, that the initial arrangement — T^Ω beginning in its initial state q_1 on initial string S^Ω — were rich. Now follow the running of the probabilistic Turing machine T^Ω. As long as we meet only non-probabilistic steps, we shall remain in a rich machine-tape configuration. Now consider the first probabilistic step. We are already in a rich machine-tape configuration at this point, and so at least one of the possible actions available from this point must itself be rich. Take any rich action. Continuing in this way (taking, at each probability-step an action resulting in a rich machine-tape configuration) we will always remain in a rich configuration; and therefore we can never halt (since the halt-state is hardly rich). We conclude that "$*$" must be a possible outcome — a contradiction. This shows that our initial supposition — that there was an infinite number of possible outcomes — must be false. A similar argument shows that, in this situation (probabilistic Turing machine T^Ω, running on initial string S^Ω, with outcome $*$ not possible), there must be an upper bound to the number

of machine steps that will ever be required to achieve the halt. The proof is the same as that above, merely redefining "rich" as "having no upper bound for the number of machine-steps required, from that point, to achieve the halt". These two proofs are virtually identical to that of the "tree theorem" in mathematics.

Thus, the case of probabilistic Turing machine T^Ω acting on S^Ω for which "$*$" is not a possibility, things are simple. There is but a finite number of possible outcomes, each has a rational probability (since T^Ω, while so running, can, by the paragraph above, pass through a probability-step at most a finite number of times), and the number of steps required to run that machine is bounded. Indeed, we could build a (regular) Turing machine T that, given such a T^Ω and S^Ω, will return the list of possible outcomes for T^Ω on S^Ω together with, for each such outcome, its probability. This new machine would simply simulate the running of T^Ω, keeping track, at each probability-step, of all the separate possibilities. When all these branches have finally ended in a "halt" (which we are guaranteed will happen in a finite number of steps, by the paragraph above), our new machine T would itself halt, compile the results for all the halt-branches it has followed, and report what it has found.

What happens when we drop the assumption that $*$ is not a possible outcome? The example above shows that there can then be an infinite number of possible outcomes. But do the probabilities remain rational? The following example shows that they need not even be computable!

> **Example.** Let the probabilistic Turing machine T^Ω, acting on the empty string $\emptyset$, proceed as follows. This T^Ω first simulates the running of the first (regular) Turing machine/string combination, TS_1, for one step. It then simulates the running of the second combination, TS_2 for one step, then TS_1 for another step. Then TS_3 for one step, TS_2 for another step, and TS_1 for another step. Then TS_4 for one step, TS_3 for another step, etc. If, during the course of these simulations, T^Ω ever finds a TS_k that halts, then T^Ω itself proceeds as follows: T either halts, returning "OK" (probability 3^{-k}); or continues running as described above (probability $1 - 3^{-k}$). Note that our T^Ω could indeed accomplish these probabilities, by going through a loop with a three-choice probability step. In this example, the possible outcomes, for T^Ω

running on string $\emptyset$, are "OK" and "$*$", and the probability distribution is: $p(OK) = c$ and $p(*) = 1 - c$, where c is the noncomputable number given in Sect. 7.

Thus, our statement of a few paragraphs ago, that "there exists a probability distribution . . . ", could not have been instead "we can compute a probability distribution . . . " Incidentally, there exist other noncomputable numbers that are so inaccessible that they cannot even arise as such probabilities.

Fix a probabilistic Turing machine T^Ω, and the initial string S^Ω on which we intend to run it, and let us now suppose that $*$ *is* a possible outcome. We may again introduce a (regular) Turing machine T, that simulates the behavior of T^Ω. As this simulation of T^Ω comes to each probability-step, T keeps track of all the possible actions, and their respective probabilities. When T discovers that one of these branches terminates by T^Ω's halting, T simply records the string that would result, and its (rational) probability. In this case, however — since now $*$ is a possible outcome for T^Ω — T will never discover that all branches of the T^Ω-simulation halt, and so T itself will never halt. In light of this, we proceed as follows. We modify T to accept as input a positive integer, n. This modified T now simulates the running of probabilistic Turing machine T^Ω, on initial string S^Ω, for precisely n steps. It then halts returning the following result: the list of all possible final strings S that T^Ω could have returned after just n steps, and for each string S in this list its (rational) probability, $p_n(S)$, as computed from T^Ω-halts observed so far. The sum of these $p_n(S)$, over all S in the list of final states, will be less than one, the difference being the probability that T^Ω, starting on S^Ω, has failed to halt at all after n steps. Note that, as the integer n increases, the resulting list of possible final states will grow (or, at least, not shrink) in length; and the then-probability for any given final state, $p_n(S)$, will be non-decreasing. Furthermore, the sum of these probabilities, over all final states in the list, will also be non-decreasing as n increases.

Let us now restrict further. We still have probabilistic Turing machine T^Ω acting on initial string S^Ω, but now suppose that, while "never halting" remains a possibility, its probability, $p(*)$, is zero. In this case, the simulating-machine T of the previous paragraph will produce a sum-of-probabilities that, as $n \to \infty$, approaches one. It follows from this, e.g., that, for every possible final string S, the number $p(S)$ is computable.

We say that a problem π is *probabilistically computable* if there exists a

probabilistic Turing machine, T^Ω, such that, for every choice of initial string S^Ω: $p(*) = 0$; and $p(\pi(S^\Omega)) > p(S)$ for every string $S \neq \pi(S^\Omega)$. In other words, for every input string S^Ω, T^Ω must have zero probability of failing to halt, and $\pi(S^\Omega)$ must be the most likely single final string. This appears to be a reasonable generalization of "computable" from the non-probabilistic case.

We now claim: A problem is probabilistically computable if and only if it is computable (in the regular way)[4]. The direction "if" is immediate, since every regular Turing machine is also a probabilistic Turing machine (namely, one with no probability-steps). For "only if", let T^Ω probabilistically-compute problem π. Our new (regular) Turing machine that computes π will do so as follows. Fix any string S^Ω. Our new machine simulates, as above, the behavior of T^Ω on S^Ω, for successively larger values of integer n, until such point as, for some choice S of final string, $p_n(S)$ exceeds all the other $p_n(S')$ by more than $1 - \sum_{S''} p_n(S'')$. When this happens, we are guaranteed that, under any further continuation of the simulation, $p_n(S)$ will always exceed all other $p_n(S')$ (since there is not enough probability unaccounted for for any $p_n(S')$ to catch up with $p_n(S)$). At this point, then, our new Turing machine halts, declaring this string S as the value of $\pi(S^\Omega)$.

Finally, we remark briefly on what happens when you are not permitted to "look inside" (i.e., to examine the table of) a probabilistic Turing machine. Let probabilistic Turing machine T^Ω probabilistically compute problem π. Fix initial string S^Ω. Fix some positive integer n, and run (not in simulation, but as seen from the outside) T^Ω, on initial string S^Ω, n times. Now, it is possible that, in one or another of these runs, T^Ω will fail to halt. But the probability of this is zero. Otherwise (with probability one), we shall be able to compile a table giving the frequency of various final strings S. Find that string S that is the most frequent outcome, and declare it the "answer". What is the probability that this S will be the wrong answer (i.e., not $\pi(S^\Omega)$)? We claim that this probability does not exceed α^{-n}, for some $\alpha > 1$ (which depends on what the probabilities are for the various possible outcomes, and so also on the input string S^Ω). Suppose, for example, that there are two outcomes, S (the correct answer) and S', with probabilities $p > p'$. Then,

[4]This result actually holds under weaker notions of "probabilistically computable". For example, it suffices that the $p(*)$ be, not necessarily zero, but merely computable in a suitable sense.

after a large number n of runs, the number of returns of S will be normally distributed, with mean np, and standard deviation $\sqrt{np(1-p)}$. Similarly, the difference between the number of returns of S and S' will have mean $n(p-p')$, and standard deviation less than $2\sqrt{np(1-p)}$. So, the probability of an (incorrect) report of S' does not exceed $\mathrm{erf}(n(p-p')/2\sqrt{np(1-p)})$, where "erf" is the error function. Using that $\mathrm{erf}(x) \leq \exp(-x^2/2)$, we obtain a probability distribution of the form given above.

Thus, in the case of a probabilistic Turing machine T^Ω probabilistically computing a problem π, it is apparently not possible to design a (regular) Turing machine to compute that same problem without opening T^Ω up for inspection. However, we can, merely by running T^Ω repeatedly, do something almost as good. Having chosen to carry out n runs of T^Ω, then: we may have nothing to report (probability zero); or we may report an incorrect string (probability falling off exponentially with n); or we may report the correct string (otherwise).

10 Difficulty Functions

So far, we have been interested largely in which problems can be computed and which cannot. We now turn to a somewhat different set of issues, involving what resources are required for the computation process. These "resources" can be of several types, e.g., of memory space, of program length, or of time. We shall be interested in the last of these, for the benefit of utilizing quantum mechanics during the computation process appears to lie in the time required for that computation. It is entirely possible that there might be other benefits.

Let us begin with a simple example. Consider a (regular) Turing machine T, which computes some problem, π. Then for any string S, T, when run with S as the initial string, will eventually halt. Denote by $f(S)$ the total number of steps the machine T will execute before halting — a measure of the "time" required for the computation. We call this f the *step-difficulty function* of T. This function f clearly depends on the problem π itself; but may also depend on the particular algorithm we implemented (via T) in the computation of π. Note that every step-difficulty function satisfies $f(S) \geq 1$ for every string S.

With this example in mind, we now introduce the following definition. Fix a character set, $\mathcal{C}$. By an *difficulty function* over $\mathcal{C}$, we mean a function, $\mathcal{S} \xrightarrow{f} R$, from the $\mathcal{C}$-strings to the reals, which is bounded away from zero, i.e., which, for some number $b > 0$, satisfies $f(S) \geq b$ for every string S. Think of the number b as the time required to boot the computer: We do not wish to address the possibility that, for a couple of very simple input strings, the computer might be able to provide an answer in "zero time", or in an arbitrarily small time. The step-difficulty function of a Turing machine that solves a problem is, of course, just one example of a difficulty function.

While the above is of course merely a definition within mathematics, it is our intention to apply it to certain computations — both Turing and otherwise. In light of this intended application, we realize that this definition has an unfortunate feature: The difficulty functions provide too much detail. For example, it might be argued that a Turing machine should be allotted less time for a step in which the character under the head remains unchanged than for a step in which the machine has to print a whole new character. Or, we might purchase for our Turing machine a new chip, which runs twice as quickly as the old. These changes in the computing set-up would, arguably, require a different choice of difficulty function. But, while such technological improvements can certainly be important, they are not the subject of interest here. We, rather, are concerned with issues such as comparing, with respect to their difficulty, several problems, or several algorithms for computing the same problem. These ideas motivate the following definition: Given two difficulty functions, f and f', we write $f \sim f'$ provided that, for some number $a > 0$, $f(S) \leq af'(S)$ and $f'(S) \leq af(S)$ for all strings S. We note that this is indeed an equivalence relation on difficulty functions. It is the equivalence classes that reflect the sense of difficulty that we are concerned with here; and we shall always be interested in difficulty functions *only* up to this equivalence.

> **Exercise.** i) Fix a Turing machine, with step-difficulty function f, that solves a problem. Let f' be the difficulty function that results if the charge is only half a unit for a Turing-step that leaves the character on the tape unchanged, but still a full unit for a Turing-step that prints a new character. Prove that $f \sim f'$. A similar result holds for new allocations of units depending on the machine internal state, on whether the head is to be moved to the

left or right, etc. ii) Prove that, for a any positive number, $f \sim af$ and $f \sim f+a$. iii) Prove that, for $a < \mathrm{glb}(f)$, $f \sim f-a$. iv) Prove that, if f and f' are equal for all but a finite number of strings S, then $f \sim f'$. v) Prove that every difficulty function f is equivalent to some integer-valued difficulty function. vi) Characterize the functions h with the following property: Whenever $f \sim f'$, then $h(f) \sim h(f')$. vi) Let Turing machines T and T' compute problems π and π', respectively. Then we have seen how to build from these two a new machine, T'', that computes $\pi'' = \pi \circ \pi'$. Show that the corresponding difficulties (up to equivalence) are related by $f''(S) = f'(S) + f(\pi'(S))$.

These examples show, among other things, that the equivalence classes have some very desirable properties: The difficulty equivalence class does not depend on how units are allocated for various types of Turing steps, on how much time is required for booting, on the purchase of a better chip, on the act of learning how to treat a few S's very quickly, or even on whether or not we demand that the difficulty function be integer-valued.

We next introduce two notions that compare difficulty functions.

Let f and f' be two difficulty functions. We write $f \leq f'$ provided that, for some number $a > 0$, we have $f(S) \leq af'(S)$ for every string S. We note that: i) replacing f and f' by equivalent difficulty functions does not change this relationship; ii) both $f \leq f'$ and $f' \leq f$ hold if and only if $f \sim f'$; iii) $f \leq f' \leq f''$ implies $f \leq f''$; and iv) for f bounded above, we have $f \leq f'$ for every f'. That is, "$\leq$" has the properties one would associate with "less than or equal to". But note that, given two difficulty functions f and f', it is not necessarily the case that either $f \leq f'$ or $f' \leq f$. For example, on the positive integers, let $f(n) = \sqrt{n}$ and $f'(n) = 1 + n \sin^2(n/20)$.

There is, in addition to "$\leq$", a second type of inequality on difficulty functions. For f and f' two difficulty functions, we write $f \ll f'$ provided that, for every number $a > 0$, we have $f(S) \leq af'(S)$ for all but at most a finite number of strings S. We note that: i) replacing f and f' by equivalent difficulty functions does not change this relationship; ii) $f \ll f'$ and $f' \ll f$ cannot both hold; iii) $f \ll f' \ll f''$ implies $f \ll f''$; iv) $f \ll f'$ implies $f \leq f'$; and iv) either of $f \leq f' \ll f''$ or $f \ll f' \leq f''$ implies $f \ll f''$. Again, these are precisely the properties suggested by the notation. Since these special meanings of "$\leq$" and "$\ll$" relate only *functions*, there will be

no confusion with the usual meanings of these symbols, which relate only *numbers*.

We think of $f \leq f'$, with $f \not\sim f'$, as meaning that "on every string, f reflects no more difficulty than does f'; and there is an infinite number of strings on which f reflects strictly less difficulty". We think of $f \ll f'$ as meaning that "f reflects less difficulty than f' on every string". The following example will illustrate these ideas

> **Example.** Consider the palindrome problem of Sect. 5. Denote by f the step-difficulty function (counting steps) for the Turing machine T described in that Section. Set $L = \text{length}(S) + 1$, another difficulty function on $\mathcal{S}$. [The "+1" in this formula merely allows us to avoid treating the empty string separately.] Then we have $L \leq f \leq L^2$. The first follows because T must in any case traverse the entire string S (in order to examine the last character), and that traversal already requires L steps. The second follows because in the worst case, when S actually is a palindrome, T must go back and forth across the string (or a substantial portion thereof) a total of L times, together with a few extra steps at the ends. Note that these relations are not approximations: They hold *exactly*. Although $L \ll L^2$, we have neither $L \ll f$ nor $f \ll L^2$. Here is another Turing machine, $\tilde{T}$ for computing this problem. Machine $\tilde{T}$ works the same as T, except that, on the first pass, it makes an extra check to see if the string S is of the form "$aaa\cdots a$". If it finds that form, then $\tilde{T}$ immediately returns to the beginning and reports "yes". Denote by $\tilde{f}$ the step-difficulty function of $\tilde{T}$. Then, for every string S that is not all "a"'s, $\tilde{f}$ requires more steps than f (since T doesn't have to carry out those extra checks that $\tilde{T}$ does), but $\tilde{f}(S)$ and $f(S)$ differ at most by some numerical multiple of L (since this checking for "a"'s requires just a few extra steps for each character in S). However, for a string S that *is* all "a"'s, $f(S)$ is the order of L^2 (since T will have to go through the laborious process of checking for palindrome-ness), while $\tilde{f}(S)$ is the order of L (since $\tilde{T}$ will recognize this special form on the first pass). Note that there is an infinite number of such strings. It follows from all this that $\tilde{f} \leq f$, but neither $\tilde{f} \ll f$ nor $\tilde{f} \sim f$. We thus think of the computation

represented by machine $\tilde{T}$ as "definitely (but only slightly) more efficient" than that of T. Intuitively, it seems plausible that the following two assertions are true: i) There exists no Turing machine T' (step-difficulty f') that computes this problem and has $f' \ll L^2$; and ii) given any Turing machine T' (step-difficulty f') that computes this problem, there exists a Turing machine T'' (step-difficulty f'') that also computes this problem, such that $f'' \leq f'$ and $f'' \not\sim f'$. The first means that you will never be able to compute this problem *substantially* more efficiently that the L^2 difficulty function; the second that, no matter how efficient you feel your present Turing machine is, there always exists one that is a little more efficient. It would be interesting to find proofs of these assertions, particularly i).

This example illustrates the idea that this equivalence relation and these inequality-relations on difficulty functions are the "right" notions: They allow us to express, in a simple way, what we want to say; and they don't draw us into a discussion of what we don't want to say.

One could imagine inventing other, inequality-like, relations on difficulty functions. For example, one could compare averages of the values of the functions over certain strings; or consider the relative frequencies of the S's for which $f(S) \leq f'(S)$ or $f'(S) \leq f(S)$ occur. But these relations tend not to be very interesting, probably because they typically require some choice of an ordering for the strings, or they are are too sensitive to relatively benign relabelings of the strings.

11 Difficult Problems; Best Algorithms

In this section, we discuss two results of Blum [2]. Both of these results are insensitive to the particular difficulty-measure — or even language — employed; and both are proved by diagonal arguments. For ease of exposition, we shall discuss both results for the Turing case (i.e., the "machines" will be Turing machines, and the difficulty measures will be step-difficulty). But it should be noted that these restrictions are not necessary.

A "difficult" problem is, intuitively, one that requires many steps for its computation. It is easy to think of problems that appear, offhand, to be quite

difficult in this sense, e.g., that which sends any integer S to the integer that is the last digit of the $(10^{S!})!$-th prime. But it is hard to be certain that this problem really is as difficult as it appears: There might, for example, be some marvelous theorem that asserts that this particular problem π merely returns "7" when S is even; and "1" when S is odd. If this, or something like it, should turn out to be the case, then this problem π would turn out to be easy to compute.

Can we give an example of a problem π that is computable, and is such that we can *guarantee* that any Turing machine that computes it has step-difficulty, say, $\geq (10^{S!})!)$? The answer is yes, but for a silly reason. Let π be the problem that, applied to positive integer S, returns the string $a \cdots a$, where the total number of a's is $(10^{S!})!$. Then certainly the step-difficulty f of any Turing machine that computes this π satisfies $(10^{S!})! \leq f$, since it takes this many steps for the Turing machine merely to print out (never mind compute) the answer. This isn't exactly what we had in mind. So, to avoid this sort of foolishness, we introduce the following definition. A problem π will be said to be *bounded* if the lengths of the strings $\pi(S)$ as S ranges through all input strings, are bounded above.

So, are there very difficult — perhaps even "arbitrarily difficult" — bounded problems? We formulate this question precisely as follows:

Assertion. Let f_o be any difficulty function. Then there exists a bounded, computable problem π with the following property: Every Turing machine T (step-difficulty f) that computes π has $f \geq f_o$.

This assertion states, in other words, that you tell me how hard (f_o) you want the problem to be, and I'll find a problem (π) such that every method of computing it (T) is at least f_o-difficult ($f \geq f_o$). This assertion, as it turns out, is false. As a preliminary step in showing this, we prove a theorem that is of interest in its own right.

Theorem. Fix character set $\mathcal{C}$. Then there exists an integer-valued problem π_o on $\mathcal{S}$ with the following property: For π any computable integer-valued problem on $\mathcal{S}$, $\pi \ll \pi_o$.

Here, the "$\ll$" in the last sentence means in the sense of difficulty functions, i.e., it means that, for any number $a > 0$, $\pi(S) \leq a\pi_o(S)$ for all but at

most a finite number of S. We require in the theorem that π_o and the π be integer-valued in order to facilitate this comparison. The idea is to choose π_o so that $\pi_o(S)$ "grows quickly as the string S gets bigger — so quickly that no mere computable π can keep up with it". This growth is very fast indeed, for we can think of some pretty fast-growing computable π's, e.g., (for S an integer) $\pi(S) = 2$ to the power of 2 to the power of 2 . . . S times. Well, that particular π (being, as it is, computable) is child's play in the hands of the *really* fast-growing π_o of the theorem.

Proof of the Theorem: Let $\pi_1, \pi_2, \cdots$ be a list of of all integer-valued, computable problems; and let $S_1, S_2, \cdots$ be a list of all strings. Now let π_o be the following problem: For $n = 1, 2, \cdots$, set

$$\pi_o(S_n) = n \times max[\pi_1(S_n), \pi_2(S_n), \cdots, \pi_n(S_n)]. \tag{3}$$

[In other words, the value of π_o on string S_n is n times the largest of the values taken by the first n computable problems, acting on that S_n.] Now fix any positive integer m. Then, for any $n \geq m$, we have $\pi_o(S_n) \geq n\pi_m(S_n)$ (since, by $n \geq m$, π_m is included in the functions maxed-over in (3)). But this last inequality (for all $n \geq m$) implies $\pi_m \ll \pi_o$. We conclude that each of the π_m is $\ll \pi_o$; and therefore, since the π_m exhaust the computable, integer-valued problems, that every such problem, π, satisfies $\pi \ll \pi_o$. \

This proof, like so many diagonal arguments, is simple, yet confusing. The idea is that, since there is only a countable list of integer-valued, computable problems, we can arrange for π_o to keep an eye on these problems, and to decide what value to assume, on successive strings, by keeping ahead of the succession of computable problems. Of course, the problem π_o, whose existence is guaranteed by the theorem, is itself not computable. This remark may seem strange at first sight, for the proof above appears to be a simple "computation" of π_o. But closer inspection reveals that we do not actually compute π_o in the proof, for we cannot Turing-construct a list of Turing machines, $T_1, T_2, \cdots$ that solve the original list $\pi_1, \pi_2, \cdots$, of problems (without, that is, computing the halting problem). Yet, although we cannot Turing-construct this sequence, it certainly does exists, for the set of all Turing machines is already countable, and so therefore is the set of all computable problems, and so therefore is the set of all integer-valued computable problems.

Exercise. Is "fast growing" the only way an integer-valued problem can be noncomputable? That is, do there exist integer-valued problems π (computable) and π' (not computable) with $\pi' \ll \pi$ (in the same sense as in the Theorem)?

The demonstration that the assertion above is false is now easy. Indeed, to obtain a counterexample we merely choose, for the f_o of the assertion, the problem π_o of the Theorem. That this choice works follows from:

Lemma. Let T be a Turing machine (step-difficulty function f) that computes a problem. Then f, regarded as a problem, is computable.

Proof: Consider the Turing machine that merely simulates T, keeping track of T's steps, and then, when that machine discovers that T would halt, itself halts, returning the number of steps that T would have executed. This new machine computes f. \

So, to summarize, it is possible to invent absurd levels of difficulty (such as that described by the π_o of the theorem): There exist no computable problems that are *that* difficult. But what about lesser levels of difficulty? Suppose, for example, that we modified the assertion above to require in addition that the given degree of difficulty be computable? It turns out that, under this modification, the assertion is true:

Theorem. (Blum) Let $\tilde{f}$ be any computable difficulty function. Then there exists a bounded, computable problem π with the following property: Every Turing machine T (step-difficulty f) that computes π satisfies $f \geq \tilde{f}$.

Proof: Let $T_1, T_2, \cdots$ be a list of all Turing machines (over the given character set), and $S_1, S_2, \cdots$ a list of all strings. Now fix any integer $n \geq 1$, and consider the following prescription (ignoring for the moment words in braces):

> **Prescription**(n)**:** Attempt to run each of the first n machines, $T_1, \cdots, T_n$, in this order, on the initial string S_n, for a total of $\tilde{f}(S_n)$ steps each. If none of the {uncanceled} machines, so run, halts before reaching $\tilde{f}(S_n)$ steps, set $\pi(S_n) = \emptyset$. Otherwise, denote by T_i the first {uncanceled} machine in this list that *does* halt before reaching $\tilde{f}(S_n)$ steps. Then set $\pi(S_n)$ equal to a string

other than T_i's output: Set $\pi(S_n)$ = "a" if T_i, on S_n, halted with output string $\emptyset$; and $\pi(S_n) = \emptyset$ otherwise. {Finally, cancel that T_i.}

This prescription, carried out for all values of n, defines a problem π (since it prescribes what string $\pi(S)$ is to be, for every S). We note that the π, so defined, is bounded (since its only possible output strings are $\emptyset$ and "a"). Furthermore, this π is computable. This follows because we can build a Turing machine to i) produce the sequence $T_1, T_2, \cdots$ of Turing machines and the sequence $S_1, S_2, \cdots$ of strings; ii) simulate the running of the first n machines, as in the prescription; iii) find the first {uncanceled} machine that fails to run for at least $\tilde{f}(S_n)$ steps (here, using the fact that $\tilde{f}$ is computable!); and iv) set $\pi(S_n)$ accordingly.

We now reinstate the braces. We carry out the prescription above, in turn, for successive values of n: $1, 2, \cdots$. Each time this prescription (for some n-value) is carried out, that machine T_i (if any) used to set $\pi(S_n)$ is now "canceled", i.e., excluded from consideration in subsequent (i.e., larger-n) applications of the prescription. Thus, the new construction is identical to the old, except that, because of this cancellation, the list of Turing machines included at each stage may be smaller than it was before. But in any case the result is again some bounded, computable problem, π (different from the old π, with which we are no longer concerned).

This π is the problem whose existence is guaranteed by the theorem. To see that it has the required property, consider any Turing machine T (step-difficulty function f) that computes π. Then this T must appear somewhere in our list of machines: Say, $T = T_7$. Consider the machines $T_1, \cdots, T_6$. Let n_o be an integer such that every one of these six machines either was canceled already by the time n reached n_o; or never will be canceled for any n. [Such an n_o exists: Indeed, each of the machines $T_1, \cdots, T_6$ either i) is at some point (i.e., for some specific n-value) canceled, or ii) is never canceled. Let n_o be the largest of the specific n-values that occur in i).] Now fix $n > n_o$, and apply the prescription above to determine $\pi(S_n)$. Which, if any, machine is canceled during this application of the prescription? It could not be any of $T_1, \cdots, T_6$ (by definition of n_o). Therefore, T_7 is on the bubble: It will not be saved by cancellation of any of the first six machines, and so will be canceled if it halts before completing all $\tilde{f}(S_n)$ steps. But T_7 cannot be the one canceled either, for, by definition of $\pi(S_n)$, cancellation implies that T_7

on S_n differs from $\pi(S_n)$, while T_7 was assumed to compute π. We conclude from all this that T_7, on S_n, must run for at least $\tilde{f}(S_n)$ steps without halting. That is, we conclude that $f(S_n) \geq \tilde{f}(S_n)$. Since this holds for all $n > n_o$ (i.e., for all but at most a finite number of n), we conclude that $f \geq \tilde{f}$. \

This is quite a proof. For each n, we stage a contest between the first n Turing machines, applying each to S_n and seeing who can go at least $\tilde{f}(S_n)$ steps without halting. We find the first machine that fails, arrange for π to be different from what *that* machine computes, remove that machine from further competition, and then repeat the contest for the next n. Since $\tilde{f}$ generally increases, the successive contests will generally get harder and harder. In this way, π avoids the losers (the machines that halt early), and thus emerges as a problem that can only be computed by a consistent winner — a machine with step-difficulty satisfying the condition of the theorem. Note that the n_o in the proof is not computable. Note also that computability of $\tilde{f}$ is used at a critical place: To get computability of π.

> **Exercise.** Show that the theorem above continues to hold if the last formula in its statement is replaced by $f \gg \tilde{f}$. Does there exist a Turing machine that accepts as input the Turing machine that computes $\tilde{f}$, and returns a Turing machine that computes a problem π whose existence is guaranteed by the theorem?

So, there are some pretty hard problems out there. We now turn to a related issue. It would be of great interest to define, for any given problem, a difficulty intrinsic to a *problem itself* (rather than to whatever method is currently being used to compute that problem). A possible line for introducing such a notion would be to let the "intrinsic difficulty" of a problem mean the minimum step-difficulty function of Turing machines that compute that problem. But, in order to implement such an idea, we would need some result to the effect that this minimum is actually achieved. One result that would certainly do the trick is the following:

Conjecture. Let π be any computable problem. Then there exists a Turing machine T (step-difficulty f) that computes this problem, with the following property: Given any other Turing machine T' (step-difficulty f') that computes this problem, we have $f' \geq f$.

Then we would take the f of the conjecture to be our definition of the intrinsic difficulty of the problem π. Unfortunately, this conjecture is false. Indeed, even for the case of the palindrome problem we have observed that, for any Turing machine T (step-difficulty f) we can think of offhand to compute this problem, there exists another, T' (step-difficulty f') with $f' \not\geq f$. [Embarrassingly enough, we don't understand even this simple little problem well enough to generate from it an actual counterexample to this conjecture!] Here, however, is a possible alternative conjecture — weaker than the one above, but perhaps retaining enough strength to salvage some sort of notion of intrinsic problem-difficulty.

Conjecture. Let π be any computable problem. Then there exists a Turing machine T (step-difficulty f) that computes this problem, with the following property: There is no Turing machine T' (step-difficulty f') that computes this problem, such that $f' \ll f$.

This conjecture, at least, does not appear to fail for π the palindrome problem (although, alas, we also don't even understand this problem well enough to prove *this* conjecture, for that π!)

But in fact, much as we might wish it to be otherwise, this conjecture is also false. Indeed, we have

Theorem. (Blum) There exists a computable problem π with the following property: For any Turing machine T (step-difficulty f) that computes π, there is another Turing machine T' (step-difficulty f') that also computes π, such that $f' \ll f$.

Thus, according to this theorem, for this particular problem π, no matter how much effort you put into finding an efficient machine of computing π, there always exists a much more efficient machine waiting in the wings. You can, if you wish, submit that new, more efficient machine to the theorem, and it will then go ahead and guarantee the existence of a still more efficient machine, and so on. Thus, for the problem π of the theorem, there is an infinite succession ever more efficient Turing machines that compute it. There would seem to be no hope of defining an "intrinsic difficulty" for this problem, at least.

We shall merely sketch the the proof of the theorem. First, let h be

the integer-valued function on nonnegative integers defined by: $h(0) = 1$, and, for $n > 0$, $h(n) = 2^{(h(0)+\cdots+h(n-1))}$. This function is rather rapidly-growing: $h(1) = 2$; $h(2) = 8$; $h(3) = 2048$; and $h(4)$ would take about ten lines to write out, and $h(5)$ could not be written on all the paper ever manufactured. We next construct the problem π of the theorem, as follows. This construction is identical with the construction of the problem π in the proof of the previous theorem, with just one small change. In carrying out the prescription, for some n-value, instead of running each of the machines $T_1, T_2, \cdots, T_n$ for the same number, $\tilde{f}(S_n)$, of steps, we now run the i-th machine in this list for $h(n-i)$ steps. Thus (since h is rapidly growing), the early machines in the list are run for vastly more steps (to see if they halt) than are the later machines in the list. In any case, the result, after this one change, is a certain computable problem π (different, of course, from the π of the previous theorem). This π, believe it or not, has the property required in the theorem.

To see this, let Turing machine T (step-difficulty f) compute π. Then this T must be one of the T_i in our list, say $T = T_7$. By construction, using the same argument as in the previous proof, it follows that $f(S_n) \geq h(n-7)$ for every n. We now introduce a new problem, π'. We set $\pi'(S_1) = \emptyset, \cdots, \pi'(S_7) = \emptyset$. For $n > 7$, we define $\pi'(S_n)$ by exactly the same prescription that defined π above, except that we use for our list of machines, not $T_1, \cdots, T_n$ as was done above, but rather just $T_8, T_9, \cdots, T_n$. This π' is of course also computable.

We next note that π' and π are actually equal on all but at most a finite number of strings. This follows because for sufficiently large n, say, $n \geq n_o$, each of $T_1, \cdots, T_6$ that ever will be canceled in the computation of π has already been canceled (while T_7, of course, will never be canceled). Once no more cancellation of these seven machines is possible, then π' and π are left to examine precisely the same machines at each step, namely, $T_8, \cdots T_n$, and so these two will end up with the same values.

We next introduce a Turing machine T' that computes π in the following manner. For $n \leq n_o$, T' simply simulates T, in this way finding out what $\pi(S_n)$ is, and returns that string. On the other hand, for $n > n_o$, T' computes π' in the manner described above (i.e., using, in the prescription at each stage, only machines $T_8, \cdots, T_n$). Denote by f' the step-difficulty of this T'.

Finally, we claim that $f' \ll f$. It suffices to compare these two difficulty functions on S_n with $n > n_o$ (since these S_n include all but a finite number of

strings). Fix $n > n_o$. Then, in order to compute $\pi'(S_n)$ $(= \pi(S_n))$, machine T' must simulate Turing machine T_8 (on S_n) for $h(n-8)$ steps, machine T_9 for $h(n-9)$ steps, and so on up to machine T_n for $h(0)$ steps. Thus, T' must run a total of not more than $h(0) + h(1) + \cdots + h(n-8) = \log_2(h(n-7))$ steps, where the last equality follows from the construction of h. Thus, we conclude that for $n > n_o$, $2^{f'(S_n)} \leq h(n-7) \leq f(S_n)$, where the last step is the bound on $f(S_n$ found earlier. The result follows.

The first thing to notice about this argument is that it contains a flaw: Right at the end, we are comparing the number of steps that T *actually executes* with the number that T' must *simulate*. Simulating looks like a lot more work that merely executing. But for reasonable difficulty-measures in reasonable languages (although not for step-difficulty in Turing) a machine can be simulated in the same number of steps (up to equivalence) as it can be run. In these cases, which include all those of serious interest, the argument is complete. But for the Turing case, a further, somewhat complicated, workaround is necessary, which we shall not discuss.

Actually, we prove more than is stated in the theorem, namely that $2^{f'} \leq f$. In fact, one can obtain a similar result for other, specific, choices of an inequality relating f and f', by simply changing the choice of the function h. It is interesting to note that, although the theorem guarantees the *existence* of T', it does not tell us how to compute it. The crucial non-computable step is that in which n_o is found. In fact, there exists no Turing machine that, with input a Turing machine T that computes the π of the theorem, returns a Turing machine T' the existence of which is guaranteed by the theorem.

So, to summarize, the prospects for assigning to each problem an "intrinsic difficulty", in some reasonable way, look pretty dismal. It may be possible to do better by some appropriate restriction on the class of problems considered. Or, there may be some way to take the greatest lower bound of the difficulty functions for machines that compute the problem, even though that lower bound is itself not realized by any machine.

12 New Language

Clearly, Turing machines are highly inefficient. The central problem is that storing scratch work on a single long tape requires that the machine plod, again and again, over the same portion of tape, looking for one little piece

of data after another. In the case of the palindrome problem, for example, we suspect (but, alas, cannot prove) that no Turing machine can compute this problem in step-difficulty $\ll L(S)^2$; and yet we might expect a "normal" computer to require only $L(S)$ steps. Thus, Turing step-difficulty functions tell us too much about Turing language and too little about the subject of real interest: the "intrinsic difficulty" of the problem or algorithm. It is time to upgrade.

We might do so, e.g., to Fortran. We would assign, in some "reasonable" way, a number of steps to each Fortran command; and thereby arrive at a Fortran-difficulty function for each Fortran-computed problem. We could, of course, do the same for C language, etc. While these new difficulty functions would certainly be more realistic than Turing step-difficulty, there remains the danger that they, too, would manifest excessive language-dependence. But it seems, intuitively, that such dependence may be small, or — if things are set up carefully — even absent. One might imagine, for example, that we could write a C-emulator in Fortran that is difficulty-function preserving.

This situation with respect to difficulty, then, is very like that we faced earlier with respect to computability: There appears to be a universal notion lurking in the background, but that notion finds expression through many languages. We want to distill out the notion itself. The answer, in the case of computability, was Turing machines. We find the simplest language that is still rich enough to encompass our idea of computability, and then *define* computability in terms of that language. We would now like to do the same thing for difficulty. That is, we would like to invent a language, with an associated difficulty function, that is as simple as possible, but not so simple that it generates unnecessary inefficiencies. In short, we want to find a language that is to difficulty as Turing language is to computability. It will turn out, unfortunately, that our innate sense of what is the "correct" difficulty function is somewhat less firm that that of what is "computable". But, in any case, we propose, below, a language that seems to capture a more or less reasonable notion of "difficulty". There may very well be better proposals.

Fix a character set $\mathcal{C}$. For S any string over $\mathcal{C}$, we write $L(S)$ for the number of characters in the string S plus one [The "plus one" is so we don't have to treat $S = \emptyset$ as an exception.]

Let there be created an infinite number of storage locations, each labeled by some string over $\mathcal{C}$; and each capable of holding an arbitrary string over $\mathcal{C}$.

Thus, we impose no upper bound on the number of storage locations being utilized, nor on the lengths of the strings in the various locations (although each location, at any one moment, contains merely a string, i.e., a finite sequence of characters; and it will turn out that only a finite number of storage locations are in play at any one moment). We write $C(S)$ for the string in the location labeled S. The idea here is that in this way we create an ample amount of highly accessible storage space.

In the present language there will be commands, each of which directs that a certain action (mostly involving what is stored in certain locations) be taken. There is a total of five classes of commands in this language: two for input/output; two for manipulating strings; and one for branching. Listed below are these five classes of commands (with, for each, a brief explanation of what is to be done; and, in braces, a number representing the "difficulty" of the command, which we shall discuss shortly).

> A command results if, in any of the five items below, "S" is replaced by any explicit string, "x" by any explicit character, and "n" by any (positive or negative) explicit integer:
>
> 1. INPUT TO $C(S)$: allows the user to enter any string, which is then placed in location S. {L(whatever string is entered)}]
>
> 2. OUTPUT FROM $C(S)$: allows the user to retrieve the string stored in location S. {L(C(S))}
>
> 3. APPEND x TO $C(C(S))$: replace whatever string is stored in location $C(S)$ with that same string, but with character x appended on the right. {L(C(S))}]
>
> 4. DELETE LAST OF $C(C(S))$: replace whatever string is stored in location $C(S)$ with the string that results from deleting its rightmost character (if any). If $C(C(S)) = \emptyset$, do nothing. {L(C(S))}
>
> 5. IF (LAST $C(C(S)) == x$) SKIP n LINES: if the last character (if any) of the string in location $C(S)$ is "x", then skip forward n program lines (if n is positive), backward $|n|$ lines (if negative). If $C(S) = \emptyset$ or if the last character of $C(S)$ is other than x, or if there are insufficient lines in the program to carry out the indicated skip, do nothing. {L(C(S))}

A *program* is a finite ordered list of commands, with the following property: The program contains exactly one INPUT command, and it is the first command of the list; and exactly one OUTPUT command, and it is the last command of the list. Here is an example of a program

INPUT $C(abc)$
APPEND d TO $C(C(yzr574))$
IF (LAST $C(C(m)) == a$) SKIP -1 LINES
OUTPUT $C(yes)$

To run a program, place $\emptyset$ in every storage location, begin at the first program line (INPUT), and enter any string. The machine then carries out the instruction of each command in turn, then moving on to the next command in the list (except for the case of command 5 (IF), for which the next command to be executed is the one indicated above). If and when the machine reaches the last command (OUTPUT) of the list, the machine halts, allowing the user to read the output string.

Any program, run on any input string, either halts or does not halt. If it halts for every input string, then that program *computes* some problem π, where $\pi(S)$ is the output string when string S is entered at INPUT. The program above, for example, indeed computes a problem, namely that with $\pi(S) = \emptyset$ for every string S. Note that we have so structured the commands that the program cannot "hang" within a single command: As long as the command follows the grammatical rules above, then — no matter how pointless that command might be — the machine will always do *something* (or maybe nothing) and move on. Failure to halt can only occur by continuing to execute command after command, indefinitely.

We could have modified the way the IF command works, in the following manner. We would, first, require that each command in the program be labeled by a unique string. Then, we would rewrite IF to direct, not that some number of program lines be skipped, but rather that there be executed next that command with some explicit string-label.

The numbers in braces, accompanying each of the five commands above, give the number of "steps" we deem the computer to require to execute that command. We call this number the *difficulty* of the command; and, for a program that, acting on a certain string, halts, we call the total number of steps executed the *difficulty* of that run of the program; and, for a program

that computes a problem, we call the total number of steps executed before halting (a function now of the input string) the *difficulty function* of the program. As always, we are interested in difficulty functions only up to equivalence. There follows a discussion of the difficulties assigned, above, to the five classes of commands.

If, in response, to an INPUT command, there is entered a string of, say, 13 characters, then the execution of that command requires, as dictated above, 14 steps. This surcharge for entering long strings turns out to be very convenient (e.g., already in the following paragraph).

The number of steps assigned to the OUTPUT command is L(the string returned). This formula was chosen merely for aesthetics: Even changing it, e.g., to "1" would result in equivalent difficulty functions. To see this, first note that, for any program on any string that runs up to the OUTPUT command, the total difficulty up to that point will be greater than or equal to the length of the longest string stored. [This follows since each command adds at least as much to the cumulative difficulty as it adds to the length of the longest string.] Thus, changing the difficulty for the OUTPUT command to "1" would, at most, reduce the total difficulty function by a factor of two. But such a reduction results in an equivalent difficulty function.

For the APPEND command, we append a character to the string in the location given by the string in the location S. We have to look up location S, to find $C(S)$, and then look up location $C(S)$ to find the string to be appended. Think of the difficulty, $L(C(S))$, of this command as a "lookup charge". Why is not the formula instead $L(C(S)) + L(S)$, i.e., why don't we also have a charge for "looking up" S? The reason is that this change results in an equivalent difficulty function. Indeed, in any given program there will be a finite number of APPEND commands, and so a finite number of explicit strings S in those commands. So, there will be a longest such string, say seven characters. Thus, a change in the difficulty of APPEND to $L(C(S)) + L(S)$ will add at most eight steps for this command, i.e., will increase the difficulty for this command by a factor of at most nine. As a result, the final difficulty function for this program will increase by at most a factor of nine. But such an increase results in an equivalent difficulty function. Note that the same argument does not apply to the term $L(C(S))$ in the difficulty of APPEND: This number (one more than the number of characters stored in location S) depends on what happens to be stored in S at the time, and so cannot be bounded a priori. Thus, this term may make a nontrivial contribution to the

final difficulty function. Why not include a term $L(C(C(S)))$ in the difficulty function of the APPEND command? After all, we have to travel to the end of the string $C(C(S))$ to append the x, and there should be some travel allowance. This is a reasonable position, which might be worth pursuing. But we have to make some decision here, and we have elected the viewpoint that there has been constructed some sort of pointer that allows us to find the end of the string easily. Similar remarks apply to the DELETE and IF commands.

In the IF command, why not include also a term $|n|$, the number of command-lines skipped? After all, skipping lines is hard work, and there should be some compensation. But, again, a given program has but a finite number of IF commands, each with an explicit n; and so a maximum value of $|n|$ for all such commands in the program. Therefore, such a change in the difficulty for the IF command would always result in an equivalent difficulty function.

We now have to deal with two matters. First, we need to show that the computable problems in this new language are precisely the computable problems (defined earlier, using Turing machines). And, second, we would like to argue that the difficulty functions generated by this language are "reasonable", i.e., that they correctly capture our intuitive sense of what the difficulty "should" be. We shall attempt to resolve both of these matters in one sweep, by generating a list of illustrative subroutines — i.e., of short program-fragments. On the one hand, these subroutines will show the richness of what can be computed in this language. On the other hand, the difficulties of these subroutines, computed from the command-difficulties above, will illustrate the typical difficulty functions this language generates. In these subroutines, $S, S', \cdots$ stand for any explicit strings, x for any explicit character, and n for any explicit integer.

L. APPEND x TO $C(S)$. $\{1\}$
2. DELETE LAST OF $C(S)$. $\{1\}$
3. IF (LAST $C(S) == x$) SKIP n LINES. $\{1\}$

For subroutine 1, let, say, $S =$ "yzr". First, choose any string, say "$h8$", not used elsewhere in the program, so $C(h8) = \emptyset$. Then: APPEND y TO $C(C(h8))$; APPEND z TO $C(C(h8))$; APPEND r TO $C(C(h8))$; APPEND x TO $C(C(\emptyset))$; DELETE LAST OF $C(C(h8))$; DELETE LAST OF $C(C(h8))$;

DELETE LAST OF $C(C(h8))$. The first three lines achieve $C(\emptyset) =$ "yzr"; the last three restore $C(\emptyset)$ to $\emptyset$. The total difficulty, for any one instance of this subroutine, is some fixed integer (in the example above, 10), not depending on what is in the various memory locations at the time. But this subroutine can appear at most a finite number of times in any program, and so the actual difficulty contributed by this subroutine, each time it is run, is bounded above. So, we may assign this subroutine a difficulty 1, up to equivalence of difficulty functions. Similar remarks apply to subroutines 2 and 3.

4. SKIP n LINES. $\{1\}$
5. IF $(C(S) == \emptyset)$ SKIP n LINES. $\{1\}$
6. IF $(C(S) == C(S'))$ SKIP n LINES. $\{L(C(S)) + L(C(S'))\}$
7. SET $C(S) = \emptyset$. $\{\mathrm{L(C(S))}\}$

For subroutine 4, use APPEND a TO $C(\emptyset)$; IF (LAST $C(\emptyset) == a$) SKIP n LINES; and then place DELETE LAST OF $C(\emptyset)$ as the first command executed after the skip. For subroutine 5, use the commands IF (LAST $C(S) == x$) SKIP .. as x runs over all possible characters; arranging the skips so that we skip n lines if all the IF's fail, but merely proceed to the next line if any succeeds. The difficulty of this subroutine, 1, results from the fact that the total number of characters is fixed. For subroutine 7, use, repeatedly, DELETE LAST FROM $C(S)$), in conjunction with subroutine 5 (to test whether $C(S)$ is empty yet). Note that subroutine 7 has a variable difficulty: Its value depends on how many characters will have to be removed from $C(S)$.

For the subroutines below, we suppose that we begin with $C(S) = \emptyset$. [If this location were not empty, then it would be necessary to use subroutine 7 first, to achieve $C(S) = \emptyset$; and to adjust the difficulty appropriately.]

8. SET $C(S) = S'$. $\{1\}$
9. SET $C(S) = C(S')$. $\{L(C(S'))\}$

For subroutine 9, we first use IF (LAST $C(S') == x$) SKIP ..., skipping to the command APPEND x TO $C(h8)$ (where "$h8$" is some location with $C(h8) = \emptyset$). Continue to test in this way each possible candidate, x, for the last character of $C(S')$. Then DELETE LAST OF $C(S')$, test whether $C(S') == \emptyset$ (subroutine 5); and, if not, repeat. In this way, we place $C(S')$, with its characters in reverse order, into $C(h8)$. Now do this all again, placing

$C(h8)$, in reverse order, into $C(S)$. It should be clear at this point that we can carry out complicated string-manipulations, e.g.: Place in $C(S)$ every other character of $C(S')$, up to the first occurrence of "a", and with each "c" replaced by "$8k$", provided that $C(S'')$ contains at least 6 characters not including the combination "yzr"; otherwise ...

For the next three subroutines, we assume that the digits, $0, 1, \cdots, 9$, are included in the character set; that strings subject to arithmetic operations are already integers; and that, again, we begin with $C(S) = \emptyset$.

10. SET $C(S) = C(S') + C(S'')$. {L(C(S'))+L(C(S''))}
11. SET $C(S) = C(S') * C(S'')$. {L(C(S'))*L(C(S''))}
12. SET $C(S) = L(C(S'))$. {L(C(S'))}

For subroutine 10, for example, we first use IF (LAST $C(S') == x$) SKIP ... and IF (LAST $C(S'') == y$) SKIP ..., for the one hundred possible combinations of digits substituted for x and y; placing, for each combination, the appropriate digit in $C(h8)$, say, as well as a marker in $C(h9)$, which tells whether or not we are carrying the 1. Then DELETE LAST OF $C(S')$; DELETE LAST OF $C(S'')$, test whether either $C(S') = \emptyset$ or $C(S'') = \emptyset$, and repeat. We will end up with the sum, with digits in reverse order, in $C(h8)$. Now transcribe $C(h8)$ into $C(S)$, reversing the order of digits. For subroutine 11, use the usual pencil-and-paper multiplication (in the course of which each digit of $C(S')$ must be multiplied by each digit of $C(S'')$). Using similar techniques, we can write write subroutines for loops, e.g., WHILE and DO; and also for complicated branchings, such as IF ((... AND NOT ...) OR ...) CARRY OUT ...; ELSE CARRY OUT ... .

It should be clear by this point that a problem is computable in this language if and only if it is (Turing) computable. After all, we have in this language the ability to enter and recover strings (INPUT, OUTPUT), the ability to manipulate strings (APPEND, DELETE) freely, and the ability to branch (IF). It should also be noted that all of the subroutines 4-12 were constructed solely from commands 1 and 2 and subroutines 1-3 (i.e., with these three subroutines replacing commands 3-5).

The role of the "$C(C(S))$" in commands 3-5 is to allow indexed arrays. Here are three subroutines that use this ability in an essential way:

13. SET $C(S'1)$ = FIRST CHARACTER OF $C(S)$, $C(S'2)$ = SECOND

CHARACTER OF $C(S)$, ETC. $\{L(C(S))\ \log(L(C(S)))\}$
14. SET $C(S) = C(C(S'))$. $\{L(C(S')) * L(C(C(S')))\}$
15. SET $C(S) = C(C(C(S')))$. $\{L(C(C(S')))*(L(C(S'))+L(C(C(C(S')))))\}$

In subroutine 13, we are assuming that the character set contains the digits; and "$S'2$" means the string resulting from appending the character "2" to the string S', etc. Thus, this subroutine allows us to place the individual characters of the string $C(S)$ in separate locations. This makes those characters directly accessible (without having to go through all of $C(S)$ each time a character is needed). The factor "$\log(L(C(S))$" in the difficulty reflects the fact that the length of the locations ($S'n$, for $n = 1, 2, \cdots$) increases logarithmically as the length of $C(S)$. Note that the base for this logarithm is irrelevant, up to equivalence. Subroutine 15 shows that we can index arrays with indexed arrays. This subroutine is given by SET $C(e8k) = C(C(S'))$; SET $C(S) = C(C(e8k))$; and this construction yields the indicated difficulty.

> **Exercise.** Explain how to write a subroutine SKIP $C(S)$ LINES (which, say, does nothing if $C(S)$ is not an integer). What is its difficulty?

This completes our summary of the present language. We conclude this discussion with a few remarks. First note that, for any program that computes a problem, the difficulty function is $\geq L(S)$. This follows, since INPUT already imposes a difficulty equal to the length of the string entered plus 1. Next, consider two programs, which compute problems π and π'. Then it is easy to write a program that computes problem $\pi \circ \pi'$: Simply juxtapose the two programs, and remove the two lines where the OUTPUT of one abuts the INPUT of the other (and, possibly, change a few explicit strings). The difficulty of the new program is the sum of the difficulties of the two components. It is easy to write short programs that change the grammar of inputs and outputs: Encoding "yes" and "no" in different ways, changing the number base, changing character set, using character-orderings in various ways, rejecting uninteresting inputs, inserting and removing separators, etc. These always have difficulty $L(S)$, where S is the string entered. It follows from all these remarks, taken together, that the difficulty function (up to equivalence) is independent of the input-output grammar.

For f and f' difficulty functions, denote by $\text{glb}(f, f')$ the function whose value, for each string S, is the smaller of the values of $f(S)$ and $f'(S)$.

Then $\text{glb}(f, f')$ is also a difficulty function, and, up to equivalence, depends only on the equivalence classes of f and f'. We have: $\text{glb}(f, f') \leq f$ and $\text{glb}(f, f') \leq f'$; and $\text{glb}(f, f') \sim f$ if and only if $f \leq f'$. Now let π be a problem, and let P (difficulty function f) and P' (difficulty function f') be programs that compute π. Then there exists a program, P'', that computes π, with difficulty function $\text{glb}(f, f')$. This P'' is constructed as follows. Program P'' first makes a copy of the initial string S, then simulates the running of P on S for ten steps; then the running of P' on the copy of S for ten steps; then continues the simulation of P for ten more steps; then P' for ten more steps; etc. Eventually, during these interlaced simulations, P'' will detect a halt, and when it does so P'' itself halts, returning the appropriate output string.

Here is a program that computes the palindrome problem. First INPUT $C(zz)$ (difficulty $L(S)$, where S is the string entered). Then dump $C(zz)$ into $C(zzz)$, with the order of the characters reversed (difficulty $L(S)$). Then use IF $(C(zz) == C(zzz))$ SKIP ... $\{L(S)\}$ to check for palindrome-ness. This program has difficulty function $L(S)$. So, by the discussion above, this program is at least as efficient as every program computing this problem. Note also that this program has difficulty function $\ll$ the step-difficulty for the Turing computation.

Here is a naive program that computes whether or not a string is prime. To compute whether integer m divides integer n (by the usual long-division method) requires $L(m)(L(n) - L(m) + 1)$ steps (for we have to multiply m by a digit ($L(m)$ steps) a total number of times given by $(L(n) - L(m) + 1)$. So, we merely check whether the integer $n \geq 2$ entered is divisible, in turn, by each of the integers $2, 3, \cdots, \sqrt{n}$. The difficulty function of this program (at most $\sqrt{n}$ runs, each of difficulty not exceeding $(\log n)^2$) is $\leq \sqrt{n}\,(\log n)^2$ (but is not equivalent to this function, for, e.g., the even integers will be disposed of very quickly by this program). It is easy to write programs that are more efficient than this naive one, e.g., by checking first to see if n is a perfect square, and only if this fails looking for factors of n, as above. In fact, there exist [1] [7] programs (based on very different methods) that are *much* more efficient than that above.

> **Exercise.** Find a program that computes whether or not a positive integer is a perfect square; and find its difficulty function.

Conjecture. Given any program (difficulty function f) that computes the

prime problem, there exists another program that computes that problem, whose difficulty function, f', satisfies $f' \leq f$ and $f' \not\sim f$.

We remark that we could have introduced this language from the beginning, instead of Turing language, using it right off as the definition of "computable". Had we done so, then the determination of what can be computed in the language would have been simpler, if less charming.

13 Improving This Language

Recall that our goal is to obtain the simplest possible language that still captures what we hope is a universal notion of "difficulty". The language constructed in the previous section is intended, as we noted, as merely a suggestion. Here, we comment on a few possible alternatives.

What about dispensing with indexed arrays altogether, i.e., replacing the APPEND, DELETE, and IF commands with subroutines 1-3? This would simplify everything, including the difficulty functions. But, we claim, doing so will likely result in a genuine loss of efficiency. Here is an example. Let the input, S, be a sequence of digits, and set $m = L(S)$. [This will be easier to follow if you think of m as being about 1,000,000, so S is written down, say, in book of some 200 pages.] Now set, for $1 \leq x \leq m$, $f_m(x) = x^{\text{digit}(x)} + 1 \bmod(m)$, where digit$(x)$ means the x^{th} digit of S. Thus, $f_m(x)$ is also an integer between 1 and m. The problem is now the following. Let there be given some input string, S. Start with $x = 7$: Then find $f_m(7)$, then $f_m(f_m(7))$, etc, up to a total of m iterations. Report the result. Let us first compute this problem without benefit of indexed arrays. To determine $f_m(x)$, we must i) find digit(x) (m, steps, since we must search through S); and then ii) raise x to a small power ($\leq (\log m)^2$ steps, since x contains at most $(\log m)$ digits). So, the difficulty to compute $f_m(x)$ is $\leq m$, and so the total difficulty to compute the problem (which entails computing $f_m(x)$ m times) is $\leq m^2$. But with indexed arrays, we may first dump the characters of S into individual locations (via subroutine 13), for a one-time difficulty of $m \log m$. But having done this, computing $f_m(x)$ requires only $(\log m)^2$ steps (one log for locating digit(x), one log for taking the power). This yields a final difficulty function of $m(\log m)^2$. Thus, using indexed arrays is much more efficient than not. The idea of this example is that computing this problem requires that we repeatedly find characters in S, and things

are so arranged that character is to be found is almost random, making it, apparently, impossible to do all the "finding" on a single pass or two through S. It thus becomes more efficient to dump the characters of S into an array, once and for all at the beginning: The resulting easy access to the characters of S ultimately pays off. Of course, we have not *proved* that there exists no way to compute this problem, without indexed arrays, that is much more efficient than the way above, although this looks unlikely. So, the critical issue here is whether our intuitive sense is that the difficulty of this problem should be m^2, or $m(\log m)^2$. If it is the latter, then we must retain indexed arrays.

Even if we begin with commands 1-2 and subroutines 1-3, we could still recover indexed arrays in a simpler way: Introduce two additional basic commands, SET $C(S) = C(C(S'))$ and SET $C(C(S')) = C(S)$. These would allow us to transfer strings currently in indexed arrays to regular locations for further processing, and then to transfer the results back again to the indexed array. What difficult shall we assign to these commands? We might use $L(C(S')) * L(C(C(S'))$, the difficulty of current subroutine 14. If we do this, then the new language will, apparently, be less efficient than the old. If, for example, we merely want to deal with the last character of a string in an array, $C(C(S'))$, then the original language permits this in just $L(C(S'))$ steps (lookup charge only), while the new language requires that the entire string be copied into a regular location before its last character is accessed. We could avoid this by making the difficulty, for the two new commands above, just $L(C(S'))$. But then the new language would be *more* efficient than the old, for we could copy an entire string from one regular location to another in just 1 step — by copying to an indexed location, and then back. Again, the issue here is what we would like our difficulty function to be.

These complications are caused by lookup charges. Then why not eliminate them entirely, i.e., imagine a world in which looking something up is free, but charges are still made for printing and erasing? This could be achieved, e.g., by retaining the present five classes of commands, but changing the difficulties for each of the last three classes to one. Consider, in this version, subroutine 15. Its difficulty will now be $L(C(C(S'))) * L(C(C(C(S'))))$. Thus, a lookup charge has crept back in: It is reflected in the factor $L(C(C(S')))$, which arises from the necessity to store the string $C(C(S'))$ in order to implement this subroutine. It seems unnatural to have a lookup charge in this case but not in others. We could eliminate that charge here with a new basic

command: SET C(S) = C(C(C(S'))). $\{L(C(C(C(S'))))\}$. But then how will we deal with SET $C(S) = C(C(C(C(S'))))$? Again, there will arise a lookup charge if this is made a subroutine, rather than an additional basic command. Are there examples in which such exotic indexed arrays actually impact the final difficulty functions?

Here is a more systematic method by which we might find a natural language with a natural efficiency function. We introduce *machine language(2)*, as follows. Storage locations are labeled by strings of exactly two characters, and each such location always contains exactly one character. Thus, "$C(h8)$" denotes the character in location $h8$; while "$C(C(h8)C(21))$" denotes the character in the location described by the two-character string whose first character is $C(h8)$ and whose second character is $C(21)$. In this machine language(2) there are (in addition to INPUT, OUTPUT, with which we are not concerned right now) four commands:

1. SET $C(xy) = z$.
2. SET $C(C(xy)C(zw)) = C(pq)$
3. SET $C(pq) = C(C(xy)C(zw))$
4. IF $(C(xy) == z)$ SKIP N LINES

where x, y, z, w, p, and q are to be replaced by arbitrary explicit characters, and n by an arbitrary (positive or negative) explicit integer. You can convince yourself that this is enough to carry out simple computations: manipulate strings (whose characters are now stored in individual locations), utilize indexed arrays, branch, count, etc. Indeed, machine language(2) is the *actual* machine language of my old Apple II+. There are 256 characters; and, thus, the total RAM of the computer is just over 65 KB! The good news about machine language(2) is that there is an obvious choice of what difficulty to assign to each command: One step. The bad news is that machine language(2) cannot compute any problem at all (as we have defined those terms), for it utilizes a finite total memory. You can make available more memory by passing to machine language(3) — the same as that above, except that now *three* characters are needed to describe a location, with the obvious modifications of the basic commands above — or, if still more space is needed, to machine language(4), etc.

The idea, now, is the following. We would introduce a certain basic language, much like that of the previous section; together with a compiler, which

would compile programs written in that language into machine language(n) for some n. [Indeed, this is what the Apple II+ does: Here, $n = 2$, and the basic language is Basic.] Given an input string S, the program that is actually run would be the compiled one, written in machine language(n). In this way, we obtain an unambiguous count of steps. If, in the course of that run, it emerged that more memory was needed, then the compiler would kick in again, to recompile the basic-language program in machine language(n') for some $n' > n$. Computation in machine language would then continue. Best if these recompilations could take place seamlessly, e.g., if the machine-language commands could be adjusted so as to be n-universal. Thus, we are free to introduce any sorts of exotic commands we wish in our basic language — the only burden being that these be compiled into machine language. And, we needn't make hard choices as to what the difficulties of these commands are to be: They are whatever follows from their execution in machine language. Thus, since it is the machine language that assigns the difficulties, we might hope that those assignments will be the natural ones. Of course, it would still be required that we decide how to compare number of steps as carried out by machine language(n) with number as carried out by machine language(n'), for $n' \neq n$. It might be interesting to see if this scheme could be implemented.

14 Quantum Mechanics

This section is a very short course in quantum mechanics — for people who already know quantum mechanics.

A *Hilbert space* is a complex vector space, equipped with an inner product that is antilinear in the first factor and linear in the second, such that the associated norm is positive-definite[5]. Vectors in Hilbert spaces are usually written, e.g., as $|\alpha\rangle$, where α is some symbol or word that describes the vector; and the inner product of vectors $|\alpha\rangle$ and $|\beta\rangle$ is usually written $\langle\alpha|\beta\rangle$. *The states of a quantum system are described by nonzero vectors in a suitable Hilbert space.*

Let H and H' be Hilbert spaces. The *tensor product* of H and H' is

[5]The full definition of a Hilbert space includes an additional condition of completeness, but all our Hilbert spaces will be finite-dimensional, and in that case completeness follows automatically

a certain Hilbert space obtained by taking linear combinations of formal products, where each product is of one vector in H with one vector in H'. The tensor product is written $H \otimes H'$, and has dimension given by the product of the dimensions of H and H'. For $|\alpha\rangle \in H$ and $|\alpha'\rangle \in H'$, the corresponding formal product, in $H \otimes H'$, is written $|\alpha\rangle|\alpha'\rangle$. *Now consider two quantum systems, whose states are described by respective Hilbert spaces H and H'. Regard these two separate systems as one. Then the Hilbert space of states of the combined system is $H \otimes H'$. Indeed, $|\alpha\rangle|\alpha'\rangle$ represents that state of the combined system with the H-system in state $|\alpha\rangle$ and the H'-system in state $|\alpha'\rangle$. Since the Hilbert space $H \otimes H'$ allows linear combinations of these simple products, not every state of the combined system is one in which each of the original systems is in a particular state.*

An *operator* on a (finite-dimensional) Hilbert space H is a linear mapping from H to itself. For example, the identity, I, is an operator, as is, for any $|\alpha\rangle \in H$, the map, written $|\alpha\rangle\langle\alpha|$, with action $|\alpha\rangle\langle\alpha|\ (|\beta\rangle) = |\alpha\rangle\ (\langle\alpha|\beta\rangle)$. For A an operator and $|\alpha\rangle$ a vector in the Hilbert space, we sometimes write $|A\alpha\rangle$ for $A(|\alpha\rangle)$. For A and A' operators on Hilbert spaces H and H', respectively, we write $A \otimes A'$ for the operator on $H \otimes H'$ with action $(A \otimes A')(|\alpha\rangle|\alpha'\rangle) = |A\alpha\rangle|A'\alpha'\rangle$ (extended to all of $H \otimes H'$ by linearity). We shall sometimes not distinguish between an operator A' acting on H' and the operator $I \otimes A'$ acting on $H \otimes H'$.

An operator U on a Hilbert space is called *unitary* if it is inner-product preserving, i.e., if $\langle U\alpha|U\beta\rangle = \langle\alpha|\beta\rangle$ for every α, β. For example, if $|\alpha\rangle$ is unit, then $I - 2|\alpha\rangle\langle\alpha|$ is unitary. *The evolution of a quantum system through time is described by a unitary operator U: Initial state $|\psi\rangle$ evolves to $|U\psi\rangle$.*

An operator on a Hilbert space is called *Hermitian* if it satisfies $\langle A\alpha|\beta\rangle = \langle\alpha|A\beta\rangle$ for every α, β. For example, I and $|\gamma\rangle\langle\gamma|$ are Hermitian. In the finite-dimensional case, every Hermitian operator has a finite number of eigenvalues, all real, and the corresponding eigenspaces span the entire Hilbert space. *Observations on quantum systems are described by Hermitian operators. Let a system, initially in state given by unit $|\psi\rangle$, be observed via Hermitian A. Then the "result" of the observation is one of the eigenvalues of A; the state of the system after the observation is the projection of $|\psi\rangle$ into the corresponding eigenspace; and the probability of that result is the squared-norm of that projection. Given a basis for H, by an observation* via that basis *we mean an observation via a Hermitian operator whose eigenspaces are those generated by the individual basis vectors.*

15 Grover Construction

We now begin a new subject: quantum-assisted computing. Our strategy will be first to consider, in some detail, one particular example. We shall then generalize. We choose for our example what is called the Grover construction [3][8][9], for it has a number of attractive features: It is very simple; it illustrates most of the constructs and ideas of quantum-assisted computing; and it holds out realistic hope of generating an example in which the quantum-assist provides a genuine reduction in difficulty.

Consider the challenge of finding a needle in a haystack. Fix an integer N (which you should think of as containing, say, 100 digits). The haystack is the N integers $0, 1, \cdots, (N-1)$; and the needle is a specific one of those integers, say k_o. We suppose that we have a computer that allows us to search for the needle in the following manner. The computer accepts as input any integer k with $0 \leq k \leq (N-1)$, and returns either "no" (if $k \neq k_o$) or "yes" (if $k = k_o$). We wish to find the needle. The obvious way to do this is to run the computer for various k-values as input. Thus, to be certain of finding k_o we would have to run the computer a total of N times; while a mere 50% chance would require only $N/2$ runs. The issue is whether we can discover a way to find the needle in substantially fewer runs.

Here is a corresponding quantum system. Let there be given an N-dimensional Hilbert space, H_{in}, with orthonormal basis $|0\rangle, |1\rangle, \cdots |N-1\rangle$: This is the quantum system in which the input will be registered. And, similarly, let there be given 2-dimensional Hilbert space, H_{out}, with orthonormal basis $|\text{no}\rangle, |\text{yes}\rangle$, to register the output. Then the Hilbert space with which the computer (and we) interact is $H_{\text{in}} \otimes H_{\text{out}}$. We represent the action of the computer by the following unitary operator[6] on this Hilbert space:

$$V(|k\rangle|\text{no}\rangle) = |k\rangle|\text{no}\rangle \ (k \neq k_o) \qquad V(|k_o\rangle|\text{no}\rangle) = |k_o\rangle|\text{yes}\rangle, \tag{4}$$

$$V(|k\rangle|\text{yes}\rangle) = |k\rangle|\text{yes}\rangle \ (k \neq k_o) \qquad V(|k_o\rangle|\text{yes}\rangle) = |k_o\rangle|\text{no}\rangle. \tag{5}$$

That is, if the input register is in any state other than $|k_o\rangle$, then V does nothing; while if it is in state $|k_o\rangle$, then V flips the output state. This unitary operator V is a reasonable rendition of what the computer might do. Indeed, suppose we have agreed to start the system with the output register in state

[6]The action of V on the linear combinations of these simple product states is, of course, fixed by linearity.

$|\text{no}\rangle$. Then Eqn. (4) above specifies that V records the correct answer (for the given $|k\rangle$) in H_{out}. And (5) is the simplest way to extend this V, as a unitary operator, to all of $H_{\text{in}} \otimes H_{\text{out}}$.

Let us pause at this point to see how we might search for the needle under this setup. First select any candidate k, then begin with the registers in the corresponding initial state, $|k\rangle|\text{no}\rangle$, and then run the computer (i.e., apply V). When the computer is finished (with final register-state that given in (4)-(5)), make an observation, on H_{out}, via the basis $|\text{no}\rangle, |\text{yes}\rangle$. If the result is "no" (which it will be, with probability $(N-1)/N$), then we know that our trial k was not the needle; while if it is "yes" (probability $1/N$) then we have found our k_o. This will be recognized as just the original search, cloaked in a thin veneer of quantum mechanics.

Let us now change things slightly. Set $|\phi\rangle = \frac{1}{\sqrt{N}}(|0\rangle + |1\rangle + \cdots + |N-1\rangle)$, a unit vector in H_{in}. This is a state which combines all possible inputs, equally weighted. Let us now begin with state $|\phi\rangle|\text{no}\rangle$. Then the running of the computer produces

$$V(|\phi\rangle|\text{no}\rangle) = \frac{1}{\sqrt{N}} \{|0\rangle + \cdots + |k_o - 1\rangle + |k_o + 1\rangle + \cdots + |N-1\rangle\} |\text{no}\rangle$$

$$+ \frac{1}{\sqrt{N}} |k_o\rangle|\text{yes}\rangle.$$

Again, let us see what we can learn from this final state. We first make an observation on H_{out} via its basis. With probability $(N-1)/N$ we will obtain "no", in which case we have learned nothing whatever (not even, as in the previous paragraph, a k known not to be the needle). But, one time out of N, we will get lucky and obtain "yes". In this case, we proceed to make an observation on H_{in} via its basis $|0\rangle, |1\rangle, \cdots, |N-1\rangle$. The result (since now the H_{in}-state is simply $|k_o\rangle$) will tell us what k_o is. But note that even this procedure, using the state $|\phi\rangle \in H_{\text{in}}$, hasn't gained us anything: This is still basically the original search, the only essential difference being that now quantum mechanics is "choosing" our trial k's for us.

Let us now make still another change, this time to the output register. Let us now choose as our initial state $|\phi\rangle \frac{1}{\sqrt{2}} \{|\text{no}\rangle - |\text{yes}\rangle\}$. In this case, the running of the computer produces

$$V(|\phi\rangle \frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\})$$

$$= \frac{1}{\sqrt{N}} \{|0\rangle + \cdots + |k_o - 1\rangle - |k_o\rangle + |k_o + 1\rangle + \cdots + |N-1\rangle\} \frac{1}{\sqrt{2}} \{|\text{no}\rangle - |\text{yes}\rangle\}.$$

That is, the output register is now always in the state $\frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}$ — both before and after the running of the computer. All the computer does, now, is reverse of the sign of the $|k_o\rangle$-term in the input register. What can we learn by making our observations on *this* final state? Absolutely nothing. An observation on H_{out}, via its basis, will give equal probability for "no" and "yes"; and an observation on H_{in}, via its basis, will return each $k = 0, 1, \cdots, (N-1)$ with equal probability. It looks as though we have gone backward.

Undaunted, we set $V_{\text{in}} = I - 2|k_o\rangle\langle k_o|$, a unitary operator (reflection across the plane orthogonal to $|k_o\rangle$) on H_{in}. Then the result of the previous paragraph can be summarized as follows: For any $|\psi\rangle \in H_{\text{in}}$,

$$V(|\psi\rangle \frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}) = |V_{\text{in}}\psi\rangle \frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}.$$

That is, provided the H_{out}-state is set to $\frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}$, the action of V (the run-the-computer operator) on $H_{\text{in}} \otimes H_{\text{out}}$ is represented by the action of this V_{in} on H_{in}, the H_{out}-state never changing. Next, set $W = I - 2|\phi\rangle\langle\phi|$, another unitary operator (reflection across the plane orthogonal to $|\phi\rangle$) on H_{in}. Note that W does *not* involve knowing which $|k\rangle$ is the needle in the haystack. We now have, by an easy calculation,

$$-WV_{\text{in}}|\phi\rangle = \frac{N-4}{N} |\phi\rangle + \frac{2}{\sqrt{N}} |k_o\rangle. \tag{6}$$

Thus, we are now working solely in H_{in}, for we begin with H_{out}-state $\frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}$, and this state never changes. Eqn. (6) gives the result of starting with state $|\phi\rangle \in H_{\text{in}}$, then running the computer (i.e., applying unitary V_{in}), and then applying unitary W.

Again, let us pause to interpret this equation. Let us make an observation, on the state given by the right side of (6), via our basis, $|0\rangle, |1\rangle, \cdots, |N-1\rangle$, for H_{in}. We find (taking the inner product of that right side with $|k_o\rangle$ and squaring the result) that the probability of obtaining k_o is $(3N-4)^2/N^3$ (the rest of the probability being distributed equally over the other k's). For large N, this probability is about $9/N$. After observing via this basis (obtaining a k-value), we may of course check directly, by running our classical computer,

whether that k is actually the needle. Nine times out of N, we will in this way find the needle. Note that this is nine times the a priori probability of finding k_o by merely guessing a k-value. It may look as though we are making some real progress here, but this appearance is misleading. Even a factor of nine in the probability for success still means that, in order to find the needle, we must carry out a number of runs proportional to N. But suppose that, instead of observing the state (6) immediately, we repeat the operation: Apply $-WV_{\rm in}$ again, and only then observe via the $|k\rangle$-basis and check the k-value that results? Our probability of success will then turn out to be twenty-five times the a priori probability. These remarks motivate what follows.

Now comes the key step: To look, from a geometrical viewpoint, at what we have just done. Consider the 2-plane in $H_{\rm in}$ spanned by $|k_o\rangle$ and $|\phi\rangle$. Each of the operators of interest, $V_{\rm in}$ and W, when acting on any vector orthogonal to this 2-plane, is the identity. Thus, all the action is taking place within this 2-plane. Let us choose an orthonormal basis for this 2-plane consisting of $|k_o\rangle$ and $|k_o\rangle^\perp$, where the latter is that linear combination of $|k_o\rangle$ and $|\phi\rangle$ that is unit and orthogonal to $|k_o\rangle$. Denote by θ the angle that $|\phi\rangle$ makes with $|k_o\rangle^\perp$. Then $\sin\theta = \langle k_o|\phi\rangle = \frac{1}{\sqrt{N}}$.

Now, each of $V_{\rm in}$ and $-W$ is a certain reflection within this plane (about vectors $|k_o\rangle^\perp$ and $|\phi\rangle$, respectively). But the composition of two reflections in a plane is a rotation. The angle of rotation is given by cos(angle) $= \langle\psi|(-WV_{\rm in})\psi\rangle$, where $|\psi\rangle$ is any unit vector in our 2-plane. Choosing $|\psi\rangle = |\phi\rangle$ (or $|k_o\rangle$, if you prefer), we find that this angle is precisely 2θ.

So, vector $|\phi\rangle$ starts out making angle θ with $|k_o\rangle^\perp$; and each application of $-WV_{\rm in}$ increases that angle by 2θ. So, if we apply $-WV_{\rm in}$ to $|\phi\rangle$ a total of s times, the resulting vector will make angle $(2s+1)\theta$ with $|k_o\rangle^\perp$. Now apply the operator $-WV_{\rm in}$ to $|\phi\rangle$ that number s of times such that $(2s+1)\theta$ is closest to $\pi/2$. Then this number of times will satisfy $s \leq \pi/(4\theta) \leq (\pi/4)\sqrt{N}$, where in the second inequality we used $\theta \geq \sin\theta = \frac{1}{\sqrt{N}}$. Having applied $-WV_{\rm in}$ to $|\phi\rangle$ this many times, the resulting vector in this plane will be within angle θ of $|k_o\rangle$. Let us now make an observation on this final vector, via the $|k\rangle$-basis for $H_{\rm in}$. The probability that this observation results in k_o, by what we just observed, is $\geq \cos^2\theta = 1 - \frac{1}{N}$. That is, our chances are excellent that this single observation on $H_{\rm in}$ will find the needle.

So, to summarize, if we apply, to initial state $|\phi\rangle\ \frac{1}{\sqrt{2}}\{|{\rm no}\rangle - |{\rm yes}\rangle\}$ in

$H_{\rm in} \otimes H_{\rm out}$, the operator $-WV$ a number of times not exceeding $\frac{\pi}{4}\sqrt{N}$, and then observe the resulting state via the $|k\rangle$-basis, we will, with probability at least $1 - \frac{1}{N}$, obtain the needle, k_o. Note that we only have to run the computer (i.e., apply V) a number of times proportional to $\sqrt{N}$ — *not* to N itself. It does indeed appear that there has been a significant gain in efficiency. This is an example of a quantum-assisted computation.

Note that if you are impatient — insisting on making $|k\rangle$-basis observations between the computer runs ("just to see how things are going"), then you will destroy this effect. This is similar to the familiar "watched pot never boils" parable in quantum mechanics.

16 Grover Construction: Six Issues

In the previous section, we gave an example of a construction that appears to show quantum mechanics providing a clear gain in efficiency over a non-quantum computation. We here discuss six issues pertaining to that construction.

16.1 Initial State

The construction requires that the registers be placed, initially, in state $|\phi\rangle \ \frac{1}{\sqrt{2}}\{|\text{no}\rangle - |\text{yes}\rangle\}$. Is it feasible to build this state?

The state of $H_{\rm out}$ would not seem to be much of a problem: After all, this is merely a 2-dimensional Hilbert space. So, for example, we could represent this space physically as the spin-states of a spin-1/2 particle, designating $|\text{no}\rangle$ and $|\text{yes}\rangle$ as the states corresponding to the spin aligned or anti-aligned in a given direction. Then $\frac{1}{\sqrt{2}}\ \{|\text{no}\rangle - |\text{yes}\rangle\}$ would be the state in which the spin is aligned in a certain orthogonal direction.

But the state $|\phi\rangle \in H_{\rm in}$ is more complicated. After all, this is a superposition of N states. To construct these states one at a time, and then "superpose them" (whatever that means) is a job that threatens to have difficulty N, i.e., to overwhelm the difficulty of running the computer $\sqrt{N}$ times. Here is a device — common in this subject — to avoid this problem. Fix, once and for all, a 2-dimensional Hilbert space, with basis $|0\rangle, |1\rangle$ (not to be confused with the vectors of the same name in $H_{\rm in}$). So, e.g., this H might be the spin-states of a spin-1/2 system. Let $N = 2^n$ for some positive integer n. [To

achieve this — at most a doubling of the number of input states — should not cause too much additional complication.] Now set

$$H_{\text{in}} = H \otimes H \otimes \cdots \otimes H, \tag{7}$$

where a total of n copies of H appear on the right[7]. Note that this gives the correct dimension for H_{in}. Now consider a typical state, e.g., $|0\rangle|1\rangle|1\rangle|0\rangle\cdots|0\rangle|1\rangle$ (total of n factors) in the Hilbert space on the right. We identify this with the state $|k\rangle$ of H_{in}, where $k = 0110\cdots01$ in base 2. The k's that result in this way range from 0 (for $00\cdots0$) up $2^n - 1$ (for $11\cdots1$); and so we indeed obtain in this way the basis we want for H_{in}. Under this identification, the construction of the state $|\phi\rangle \in H_{\text{in}}$ is quite easy: It is a simple product

$$|\phi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\ \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\ \cdots\ \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

of the states $\frac{1}{\sqrt{2}}\ (|0\rangle + |1\rangle)$ for each of the H-factors. This follows, expanding the right side, and using the definitions of $|k\rangle$ and $|\phi\rangle$. Thus, with this choice of how H_{in} is to be structured, the construction of the state $|\phi\rangle$ should be relatively easy. We note that this construction could have been carried out[8] with any fixed dimension for the factor-Hilbert spaces H.

16.2 Final Observation on H_{in}

The construction requires that, at the end of the computer-runs, an observation be made on H_{in} via the $|k\rangle$-basis. Is it feasible to make such an observation?

Yes, it is. Denote by A the Hermitian operator $|0\rangle 0\langle 0| + |1\rangle 1\langle 1|$ on H (so observation of A is observation of H via its natural basis). For example, if H is spin-states, and the basis is spin-component in a certain direction, then

[7]Note that we could not, e.g., let the H's be simply the spin-states for n identical spin-1/2 particles, because the states on the right in (7) are not in general antisymmetric under particle-interchange. However, we could, e.g., have a system of n electrons occupying n energy levels (say, in an atom); where each H, referring some energy level, gives the spin-state of the occupant of that level.

[8]In fact, the different H's in the product could, if we so desired, be assigned different dimensions. **Exercise.** Set up a system of arithmetic in which each of the various digits of an integer refers to different number-system base. Figure out how to add in this system (which turns out to be quite simple!).

A would be the observation of spin-component in that direction, plus 1/2. Now consider the following Hermitian operator on $H \otimes \cdots \otimes H$: Operator $(2^{n-1}A)$ applied to the first H-factor, plus operator $(2^{n-2}A)$ applied to the second H-factor, and so on, until reaching finally operator $(1\ A)$ applied to the last H-factor. [Here, we are regarding these operators on the H-factors as operators on the tensor product in the manner described in Sect. 14.] The resulting sum can, via (7), be regarded as an operator on H_{in}; and we note that it does indeed have the $|k\rangle$ as its eigenstates. [In physical terms, observe the first H-component and multiply by 2^{n-1}; the second, by 2^{n-2}; etc., and add. The result will be precisely the k-value of that state.] We would expect to have no difficulty in making an observation of H via this operator A; and, therefore, no difficulty in making an observation of H_{in}, so constructed, via its $|k\rangle$-basis.

16.3 Building the Operator W

The construction requires that we apply unitary operator $W = I - 2|\phi\rangle\langle\phi|$ to H_{in}. Is it feasible to build and apply such an operator?

Note that this by no means follows immediately from the prior point: The mere fact that we feel capable to placing H_{in} in state $|\phi\rangle$ does not lead directly to an interaction on H_{in} that shifts each state $|\psi\rangle \in H_{\text{in}}$ to state $W|\psi\rangle \in H_{\text{in}}$. In order to build the operator W, we proceed as follows.

We first require a few preliminaries. We introduce a more convenient basis for H: $|\alpha\rangle = \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle)$, $|\beta\rangle = \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle)$. In terms of this basis, we have $W = I - 2|\alpha\rangle\cdots|\alpha\rangle\langle\alpha|\cdots\langle\alpha|$. Next, we introduce a unitary operator T on $H \otimes H \otimes H$, with the following action: $T(|\alpha\rangle|\alpha\rangle|\alpha\rangle) = |\alpha\rangle|\alpha\rangle|\beta\rangle$, $T(|\alpha\rangle|\alpha\rangle|\beta\rangle) = |\alpha\rangle|\alpha\rangle|\alpha\rangle$, while T the identity on the other six basis elements of $H \otimes H \otimes H$. That is, this operator T, which is called the *Toffoli gate*, flips the third H-state if and only if the first two H-states are[9] both $|\alpha\rangle$; and so, e.g., we have $T^2 = I$. Let us denote by $H_1, H_2, \cdots, H_n$ the n H's in the tensor product that is H_{in}. We now introduce a second Hilbert space,

[9]The general state in $H \otimes H \otimes H$ is, of course, not one in which each of the H's is in a particular state ($|\alpha\rangle$ or $|\beta\rangle$), but rather is a superposition of all eight possible combinations of individual H-states. We often describe operators, such as this T, by giving their actions on each of the combinations that appear in this superposition. Thus, when we say, e.g., "the first two H-states are ...", we really mean "that term, in the superposition, in which the first two H-states are ..."

$H_{\text{scratch}} = \tilde{H}_3 \otimes \tilde{H}_4 \otimes \cdots \otimes \tilde{H}_{n+1}$, where each of the $\tilde{H}$ in this tensor product is also a copy of our basic Hilbert space H. This is the Hilbert space in which we shall carry out scratch work. Thus, our full Hilbert space is now $H_{\text{in}} \otimes H_{\text{scratch}}$, a tensor product of $2n-1$ copies of H. Now consider the following operator on this tensor product:

$$\mathcal{W} = T(H_n, \tilde{H}_n, \tilde{H}_{n+1})T(H_{n-1}, \tilde{H}_{n-1}, \tilde{H}_n) \cdots T(H_3, \tilde{H}_3, \tilde{H}_4) \times T(H_1, H_2, \tilde{H}_3). \tag{8}$$

We note that this operator, as a composition of unitary operators, is unitary. Let us now begin with an arbitrary state in H_{in}, but with H_{scratch} in the state $|\tau\rangle = |\beta\rangle|\beta\rangle \cdots |\beta\rangle \, \frac{1}{\sqrt{2}}(|\alpha\rangle - |\beta\rangle)$. Let us apply to this state the operator (8), and see what happens. The rightmost operator T in this composition will place $\tilde{H}_3$ (which began in state $|\beta\rangle$) in state $|\alpha\rangle$ if and only if the H_1- and H_2-states are both $|\alpha\rangle$. The next T, reading from right to left, will place $\tilde{H}_4$ in state $|\alpha\rangle$ if and only if H_3 and $\tilde{H}_3$ are both in state $|\alpha\rangle$, i.e., if and only if H_1, H_2, and H_3 are all in state $|\alpha\rangle$. And, similarly, the next T will place $\tilde{H}_5$ in state $|\alpha\rangle$ if and only if all four of H_1, H_2, H_3 and H_4 are in state $|\alpha\rangle$. Continue in this way, working from right to left in (8). Recall that the last $\tilde{H}$, $\tilde{H}_{n+1}$ begins in state $\frac{1}{\sqrt{2}}(|\alpha\rangle - |\beta\rangle)$ rather than $|\alpha\rangle$. Thus, in the last step, an attempt to "flip" the $\tilde{H}_{n+1}$-state will merely introduce a minus sign. We conclude: The operator $\mathcal{W}$ of (8), acting on a state $|\psi\rangle|\tau\rangle \in H_{\text{in}} \otimes H_{\text{scratch}}$, where $|\psi\rangle$ is any state in H_{in}, and $|\tau\rangle$ is the state in H_{scratch} given above, indeed generates a sign change if all the H's are in state $|\alpha\rangle$, and no sign change otherwise.

The operator $\mathcal{W}$, so constructed, is our candidate for W. Of course, it acts, not merely on the Hilbert space H_{in} (as W does), but rather on $H_{\text{in}} \otimes H_{\text{scratch}}$ Nevertheless, it does seem to have the right action and so, it appears, would seem to suffice for the Grover construction.

But this appearance is misleading: The above candidate, $\mathcal{W}$, will not work as a proxy for W, for the following reason. Some scratch work for this calculation was left in the auxiliary Hilbert space H_{scratch}. That is, the final state, after application of $\mathcal{W}$ is an entanglement of H_{in}-states and H_{scratch}-states. Consider, for example, $n = 4$. Then if the initial state of H_{in} was $|\alpha\rangle|\alpha\rangle|\beta\rangle|\alpha\rangle$, say, then the final state of H_{scratch} will be $|\alpha\rangle|\beta\rangle \, \frac{1}{\sqrt{2}}(|\alpha\rangle - |\beta\rangle)$; while if the initial state of of H_{in} was $|\alpha\rangle|\beta\rangle|\beta\rangle|\alpha\rangle$, then the final state of H_{scratch} will be $|\beta\rangle|\beta\rangle \, \frac{1}{\sqrt{2}}(|\alpha\rangle - |\beta\rangle)$. This entanglement, we claim, will destroy

the working of the Grover construction. To see this, consider Eqn. (6), which gives the result of the first application of $-WV_{\text{in}}$ to $|\phi\rangle$: a rotation $|\phi\rangle$ through angle 2θ. The key to the construction is that the next application of $-WV_{\text{in}}$ (as well as each successive application) must rotate through an additional angle 2θ. But, in order for this to happen, there must occur cancellation between the $|\phi\rangle$'s and $|k_o\rangle$'s that arise from application of $-WV_{\text{in}}$ to the two terms on the right in (6). Now consider what happens if the W on the left in (6) is replaced by $\mathcal{W}$. Then the terms on the right side of this equation will become entangled with various elements of H_{scratch}. Therefore, on the next application of $-WV_{\text{in}}$ the necessary cancellations on the right will not take place. The Grover construction will thus fail.

In order to obtain an effective W, we proceed as follows. Set

$$\mathcal{W}' = T(H_3, \tilde{H}_3, \tilde{H}_4) \cdots T(H_{n-1}, \tilde{H}_{n-1}, \tilde{H}_n)\ \mathcal{W}. \tag{9}$$

That is, $\mathcal{W}'$ first applies $\mathcal{W}$, and then applies all the operators of $\mathcal{W}$, save the leftmost, in reverse order. It is easy to check that this procedure undoes the entanglement. That is, we have $\mathcal{W}'(|\psi\rangle|\tau\rangle) = |W\psi\rangle|\tau\rangle$ for any $|\psi\rangle \in H_{\text{in}}$, where $|\tau\rangle \in H_{\text{scratch}}$ is the initial state given above. *This* $\mathcal{W}'$, then, can be used in place of W in the Grover construction.

We conclude, then, that the operator W on H_{in} in the Grover construction can indeed be built, by introducing an auxiliary Hilbert space H_{scratch}, and applying the Toffoli gate (an operator on $H \otimes H \otimes H$) a total of $2n-3$ times. So, it would seem that the operator W is feasible — provided the operator T is feasible. We shall return to this last issue shortly.

16.4 Building the Operator V

No real computer, it might be argued, operates by applying some unitary operator V to $H_{\text{in}} \otimes H_{\text{out}}$, as in the Grover construction. After all, real computers use irreversible operations (such as placing bits in locations). How, then, are we to construct and interpret the operator V?

Here is a model for how a computer might operate. We introduce an additional Hilbert space, H_{com}, to represent the computer states. Then the total Hilbert space is $H_{\text{in}} \otimes H_{\text{out}} \otimes H_{\text{comp}}$. The running of the computer will then be represented by some unitary operator, $\mathcal{V}$, on this Hilbert space. Let us fix a vector $|\psi_{\text{init}}\rangle \in H_{\text{comp}}$, to represent the initial state of the computer.

Then, in the Grover case (i.e., with $H_{\rm in}$ spanned by $|0\rangle \cdots |N-1\rangle$ and $H_{\rm out}$ by $|{\rm no}\rangle, |{\rm yes}\rangle$)), the action of a suitable $\mathcal{V}$ would be as follows:

$$\mathcal{V}(|k\rangle|{\rm no/yes}\rangle|\psi_{\rm init}\rangle) = |k\rangle|{\rm no/yes}\rangle|\psi_k\rangle, \tag{10}$$

where the $H_{\rm out}$-state on the right is $|{\rm no}\rangle$ or $|{\rm yes}\rangle$ depending on whether the $H_{\rm out}$-state on the left is $|{\rm no}\rangle$ or $|{\rm yes}\rangle$, and also on whether or not $k = k_o$. The $|\psi_k\rangle \in H_{\rm comp}$ on the right in (10) is the final state in which the computer finds itself, depending on the k-value on the left (and also on the choice of initial state in $H_{\rm out}$, which we suppress). This operator $\mathcal{V}$ is unitary, and so invertible. Thus, we are suggesting, the operation of any computer must always be reversible. [Indeed, in a world governed by quantum mechanics, this is necessary, for dynamics therein is described by an (invertible) unitary operator.] Things don't appear to be this way in practice only because we fail to take into account how large and complicated $H_{\rm comp}$ can be. It includes not only the states of the chips, wires, fan, etc within the box, but also (if, say, the computer is plugged in) the states of the electric company, and then of its employees, etc. By the time all this dust settles, things look pretty irreversible.

Unfortunately, the operator $\mathcal{V}$ of (10) will not serve as a proxy for the operator V of the Grover construction. The problem is that $\mathcal{V}$ introduces entanglements between $H_{\rm in} \otimes H_{\rm out}$ on the one hand and $H_{\rm comp}$ on the other, as reflected in the dependence of the final computer state, $|\psi_k\rangle$, in (10) on k. These entanglements, in the same manner as for $\mathcal{W}$ in the discussion above, will interfere with the cancellation that must take place in Eqn. (6), and will thereby cause the Grover construction to fail. In order to avoid these entanglements, we must, e.g., so design our computer that the final computer state, say $|\psi_{\rm final}\rangle$, is independent of $|k\rangle$. Then, when it comes time to repeat the computation, we could either apply some special treatment to the computer to restore its initial state to $|\psi_{\rm init}\rangle$, or discard that computer entirely, bringing in another with the initial state $|\psi_{\rm init}\rangle$ already preinstalled[10]. Best

[10]Why not get rid of these awkward entanglements, not by searching for a clever $\mathcal{V}$, but rather by simply discarding the computer after each run, bringing in a new computer, with $|\psi_{\rm init}\rangle$ preinstalled, for the next run? The problem with this maneuver is that the act of discarding a system entangled with another places the latter system in a mixed state, as described by a density operator. But a mixed state for $H_{\rm in}$ destroys the cancellation, and so the Grover construction, as surely as does entanglement

would be if we could arrange that $\mathcal{V}$ automatically, at the end of each run, returns the computer to state $|\psi_{\text{init}}\rangle$, ready for the next run.

So, in any case, in order to carry out the computation implicit in the Grover construction, we shall have to produce a computer that does not introduce entanglements between computer states and in-out states. This is definitely *not* the computer on your desk! We shall have to build our computer anew. The danger we face is that the building and operating of such computers consumes resources — in particular, time — and we must be careful that this consumption does not overwhelm the apparent savings we derive from using quantum mechanics.

Recall that the Hilbert space $H_{\text{in}} = H \otimes \cdots \otimes H$, in the Grover construction, has large dimension, 2^n. Our computer must interact with this large Hilbert space, but do so relatively efficiently. It would be of great help if we could design our computer to interact, not with all n of the H's at once, but rather with only a few at a time. Does this restriction entail a restriction on the possible unitary operators we can generate on $H_{\text{in}} \otimes H_{\text{out}}$? The following shows that it does not.

Theorem. Let H be a finite-dimensional Hilbert space. Then any unitary operator on $H \otimes H \otimes \cdots \otimes H$ is equal to a product of unitary operators, each of which acts on at most two of the H-factors in this tensor product.

Of course, different combinations of the two H-factors are allowed for the different unitary operators in this product. Our proof of the theorem will make use of three facts.

Lemma 1. Every Hermitian operator on $H \otimes \cdots \otimes H$ is a linear combination of operators of the form $A \otimes \cdots \otimes B$, where $A, \cdots, B$ are Hermitian operators on H.
Lemma 2. Every Hermitian operator on a Hilbert space is a linear combination of commutators of Hermitian operators, and the identity I.
Lemma 3. Fix a connected Lie group G, and a collection of one-parameter subgroups of G. If the generators of these subgroups generate the entire Lie algebra of G, then the subgroups themselves generate the entirety of G.

Lemma 1 is easy to prove by a dimensional argument, using that the dimension of the (real) vector space of Hermitian operators on a Hilbert

space is equal to the square of the dimension of that (complex) space. Let there be n H's in the tensor product, each of dimension m. Then the Hilbert space $H \otimes \cdots \otimes H$ has dimension m^n, and so the vector space of Hermitian operators on this space has dimension $(n^m)^2$. The Hermitian operators of the form given in the Lemma form a subspace of this space, and it has dimension (dimension of Hermitian operators on $H)^m = (n^2)^m$. These dimensions are equal, and so the subspace is the entire vector space[11]. Lemma 2 (which, apparently, has little independent interest) follows by direct construction. For $m = 2$, for example, it is the statement that any linear combination of spin-operators is a commutator of two such linear combinations. In Lemma 3, the Lie algebra of a Lie group is the tangent space at its identity element. The "generator" of a one-parameter subgroup is that element of the Lie algebra given by the tangent to that curve at the identity. The Lie algebra "generated" by these generators is the collection of all elements that can be obtained by using linear combinations and brackets on the generators of the one-parameter subgroups. And, finally, that the elements of these subgroups "generate" the entirety of G means that every element of G can be written as a (finite) product of such subgroup-elements. This Lemma, in other words, states that if you can get the entire group from the subgroups "infinitesimally close to the identity", then you can indeed get the entire group from the subgroups "everywhere"[12]. [This is the sort of thing that would normally be used, without mention, in a physics course.]

The theorem is very easy to prove from the three Lemmas. Consider, say, $n = 3$. We have, for A, B, C, and D any Hermitian operators on H, and a any real number

$$[A \otimes I \otimes C,\ I \otimes B \otimes D] + a\ A \otimes B \otimes I = A \otimes B \otimes ([C, D] + aI). \qquad (11)$$

where "[,]" denotes i times the commutator. Each of the operators that appears on the left contains an "I", and so is a generator of unitary operators

[11] Lemma 1 is not quite as empty as it may appear at first sight. To see this, you might try to write the Hermitian operator that switches the two H-states in $H \otimes H$ in the form guaranteed by the Lemma.

[12] As an example, let G be the rotation group, let one one-parameter subgroup be the rotations about some vector $\vec{s}$, and let the another be rotations about some other, independent, vector $\vec{t}$. Then, since the Lie bracket of the corresponding infinitesimal rotations is simply an infinitesimal rotation about $\vec{s} \times \vec{t}$, the hypothesis of Lemma 3 is satisfied. The Lemma asserts in this case that *every* rotation can be written as a some product of various rotations about $\vec{s}$ and $\vec{t}$. This fact is the basis of Euler angles.

on $H \otimes H \otimes H$ that act on only two factors. By Lemma 2, the right side of (11) includes the general tensor product of three Hermitian operators on $H \otimes H \otimes H$; and, by Lemma 1, these span all Hermitian operators on the tensor product. The result (for $n = 3$) now follows from Lemma 3. The case of general n is by induction on n, repeating the construction of (11) at each step.

It seems likely that the product (whose existence is guaranteed by the Theorem) involves no more than 5^{n-2} factors (perhaps substantially fewer), by an argument that traces the mechanism of Lemma 3. Unfortunately, this number grows quickly with n. The Theorem is also true (suitably modifying Lemma 1) when the H's in the tensor product have different dimensions.

So, we may expect to build our computer out of operators that act on H-factors two at a time. But is it feasible to construct even these operators? Let us take, as an example, the case in which H is 2-dimensional, representing the spin-states of an electron. Then the general Hermitian operator on H is $\vec{s} \cdot \vec{\sigma} + b\ I$, where $\vec{s}$ is any vector in 3-space, $\vec{\sigma}$ is the vector (Pauli) spin operator, and b is any real number. [Note that these form a 4-dimensional vector space, as required.] The Hermitian operator $\vec{s} \cdot \vec{\sigma}$ generates the family of unitary operators, written $e^{i\ \vec{s} \cdot \vec{\sigma}}$, that correspond to rotations in space about the vector $\vec{s}$ as axis; while $b\ I$ generates the family, written e^{ib}, that correspond to overall phase-changes (which have no physical significance).

The unitary operators on a single H can be constructed physically as follows. The unitary operators corresponding to rotations about $\vec{s}$ result from applying to the electron a magnetic field in the $\vec{s}$-direction, for such a field causes the electron, by virtue of its angular momentum and magnetic moment, to precess about the magnetic-field direction. The product of the field-strength and the time for which the interaction is turned on determine the magnitude of this rotation.

But, in order to invoke the Theorem, we must also construct the unitary operators on $H \otimes H$, i.e., on the two-electron system. It should be clear that merely subjecting the two electrons, each to its own magnetic field, will not suffice. We must introduce some sort of direct interaction between the two electrons. One such is what is called the spin-spin interaction. The corresponding Hermitian operator on $H \otimes H$ is $\vec{\sigma_1} \cdot \vec{\sigma_2}$ where $\vec{\sigma_1}$ and $\vec{\sigma_2}$ denote the spin operator acting on the first and second factor in $H \otimes H$, respectively. [Strictly speaking, we should include a "$\otimes$" between the two σ's in this expression; but the dot gets in the way.] This particular interaction actually

occurs in nature: If the two electrons are merely brought close together[13], then, by virtue of the electromagnetic interaction between their magnetic moments, the electrons interact in just the manner we have described. The corresponding unitary operator may be written $e^{ia\ \vec{\sigma_1}\cdot\vec{\sigma_2}}$, where the number a is determined by how close together the electrons are placed, and for how long.

That these two physical operations — placing one or both electrons in a magnetic field, and allowing the electrons to interact electromagnetically — suffice to generate all possible two-electron interactions now follows from:

Theorem. Let H be a 2-dimensional Hilbert space. Then every unitary operator on $H \otimes H$ is equal to some (finite) product of the operators e^{ib}, $I \otimes e^{i\ \vec{s}\cdot\vec{\sigma_1}}$, $I \otimes e^{i\ \vec{s}\cdot\vec{\sigma_2}}$, and $e^{ia\ \vec{\sigma_1}\cdot\vec{\sigma_2}}$, where $\vec{s}$ is any vector in 3-space and a and b are any real numbers.

The proof is virtually identically to that of the earlier Theorem, using the Lemmas in the same way. In this case, Eqn. (11) is replaced by

$$-[\vec{t}\cdot\vec{\sigma_1}\otimes I,\ [\vec{s}\cdot\vec{\sigma_1}\otimes I,\ \vec{\sigma_1}\cdot\vec{\sigma_2}]] + (\vec{s}\cdot\vec{t})\ \vec{\sigma_1}\cdot\vec{\sigma_2} = (\vec{s}\cdot\vec{\sigma_1})\otimes(\vec{t}\cdot\vec{\sigma_2}), \quad (12)$$

where we have used the fact that $[\vec{s}\cdot\vec{\sigma},\ \vec{t}\cdot\vec{\sigma}] = i(\vec{s}\times\vec{t})\cdot\vec{\sigma}$. Taking linear combinations involving the right side of (12) and the Hermitian operators $\vec{s}\cdot\vec{\sigma_1}\otimes I$, $I\otimes\vec{s}\cdot\vec{\sigma_2}$, and $I\otimes I$ reproduce the entire Lie algebra of $H\otimes H$; which is just what we need to complete the proof.

Here are a couple of examples. Consider the operator W on $H_{\rm in} = H\otimes\cdots\otimes H$, where H is taken as the two-dimensional Hilbert space of spin-1/2 states. It follows, from the two theorems above, that this W is equal to a composition of our basic unitary operators: That (on a single H) generated by a magnetic field, and that (on two H's) generated by the spin-spin interaction. It seems likely that the number of such basic operators that must be composed to construct W in this manner increases exponentially in n. Note that this construction of W is different from that of Sect. 16.3, for there we made use of an auxiliary Hilbert space $H_{\rm scratch}$, whereas here there is none. Next, note, again from the two theorems above, that the Toffoli

[13]This could be done, for example, by keeping the electrons in boxes, with H representing the spin-state of the occupant of a given box; and then moving the boxes into close proximity.

operator, T, on $H \otimes H \otimes H$ is also equal to a product of the basic operators on H. Having constructed T from the basic operators, we may then proceed to construct W from the basic operators, using the strategy of Sect. 16.3. While this alternative construction of W requires an auxiliary Hilbert space H_{scratch}, it does have the advantage that the number of basic operators required grows only linearly with n. We may, in addition to W, also construct V_{in}, in the following manner. Denote by U the unitary operator on H with action $U|0\rangle = |1\rangle, U|1\rangle = |0\rangle$. Suppose that the needle, written in base 2, is, say, $k_o = 01001 \cdots 01$. Then set $\mathcal{U} = U \otimes I \otimes U \otimes U \otimes I \cdots \otimes U \otimes I$, where the U's and I's on the right correspond to the digits in this expression for k_o. We then claim that $V_{\text{in}} = \mathcal{U}W\mathcal{U}$. This is easy to check: $\mathcal{U}$ sends $|k_o\rangle$ to $|1\rangle|1\rangle \cdots |1\rangle$ (but other $|k\rangle$'s to something else); and then W produces a minus sign (but, for other $|k\rangle$'s, a plus sign); and then the final $\mathcal{U}$ restores the original state. Note that, in this construction of V_{in}, the number of a number of operators that must be composed grows only linear in n.

The discussion above shows that there will in general be a variety of ways to construct a given unitary operator out of some set of basic operators. Some ways may involve an auxiliary Hilbert space (in which case we must avoid entanglement) and some not; some may involve the composition of a large number of basic operators and some a smaller number. But, unfortunately, none of this is what we really want for the Grover construction. The V_{in} above, for example, is completely useless for our purposes, because it requires that you already "know" k_o, and this is exactly what you are not supposed to know. What we need is, not a variety of ways to construct some "given" unitary operator on a Hilbert space out of basic operators, but rather a way to convert programs into operators. In the Grover case, for example, we begin with a computer program that checks whether or not a given k is the needle; and we wish to convert that program to a suitable unitary operator V_{in}. Here is a general summary of what we are looking for.

> Let there be given a program P that accepts as input nonnegative integers, and returns nonnegative integers. We may register the inputs and outputs in Hilbert spaces H_{in} and H_{out}, each of which is a (finite) tensor product of some finite-dimensional Hilbert space H with itself. We wish to convert the program P into a unitary operator on $H_{\text{in}} \otimes H_{\text{out}}$, such that this operator "computes" the output from the input in the manner of P, and

> does so with substantially the same difficulty function as that of P. This unitary operator may require an auxiliary Hilbert space H_{scratch}, but if it does then the operator must be such that the $H_{\text{in}} \otimes H_{\text{out}}$-part of the final state is not entangled with the H_{scratch}-part.

It is not clear how to make this summary into a precise statement. What, for example, does "... in the manner of ..." mean; and what is the "difficulty function" of a unitary operator? We have in mind some sort of compiler, which turns command-lines in the program into compositions of some basic unitary operators on the Hilbert spaces. But it is not clear how this is to work. What, for example, are the operator-equivalents of APPEND and DELETE? Even more difficult would be to find an operator-equivalent of IF (LAST $C(S) == x$) THEN SKIP n LINES. In any case, having constructed such a compiler, then we might be able to define a suitable difficulty function in terms of the number of basic operators in the composition. And even after all this, we would have to contend with the fact that *programs* accept as input arbitrary integers, while our *operators* act on just finite tensor products. All this is somewhat reminiscent of the issue, discussed in Sect. 13, of compiling programs in machine language. It might be worthwhile to try to resolve that issue first, as a prerequisite to this one.

There seems to be a sense, in this field, that there may exist some sort of construction along the lines outlined above.

16.5 Errors

Errors abound in the Grover construction. They can appear in the setting up of the initial state, in the application of the operators W and V, and in the observation on the final state. Errors could arise, for example, from imperfections in the apparatus; or from quantum tunneling causing interactions between the H-states and the thermal fluctuations in the outside world. How shall we take such errors into account?

Of course, errors abound everywhere in physics. But here, because of the kinds of questions we are asking, this issue seems particular compelling. Consider a computation, and suppose that, on the initial run, the input string S is such that we require just 10 steps. Then we can afford, for this run, to be relatively cavalier about errors. But the next run, for another S, might

involve 1,000 steps, requiring, in order to keep the effects of errors in check, that we purchase new and better equipment. And still another run might involve 1,000,000,000 steps, requiring that we cool the entire Earth down to 0.03^o K and move the Sun over to another part of the Galaxy. All of these extra precautions take time and effort, and so should be taken into account in the difficulty function. But how will we be able ever to include such things? How, for example, does the difficulty of these various precautions scale with the number of steps in the computation? Similar issues arise already in ordinary computing: Bits are sometimes recorded incorrectly; and the longer the calculation the greater the care that must be exercised in this regard.

We shall simply ignore the effects of errors, not out of any conviction these effects are likely to be unimportant, but rather because we do not know how to do anything else.

16.6 What Is The Problem?

The Grover construction, as you have undoubtedly noticed, does not compute any problem, as we have defined that term. A "problem" entails an output for *every* possible input string; while the Grover construction searches for the needle in a finite haystack. Finite input sets are not very interesting: All problems based on them are computable, and all difficulty functions on them are equivalent.

The obvious way to respond to this situation would be to modify the construction to apply to a variety of haystacks. Given n, we first construct the Hilbert space H_{in} (as a tensor product of n copies of a 2-dimensional H), then build our computer (i.e, construct our operator V). We are now prepared to apply the Grover construction to find the needle. Of course, the building of the computer (i.e., of the operator V) is an additional burden, which would, presumably, be included in the difficulty function for this computation.

So let us suppose, then, that we have suitably modified the Grover construction along these lines. We would then be in a position to ask the key question: Is there any problem for which the Grover construction, so configured, is more efficient than any computation-method not using quantum mechanics? Here is a precise mathematical assertion that reflects these ideas.

Assertion. Let P be a program (written, say, in the language of Sect. 12) that accepts as input a pair, (n, k), where n is a positive integer and k is

an integer with $0 \leq k \leq N-1$ (where we have set $N = 2^n$); and returns either "yes" or "no". Let this P have the following property: For each n, there is one and only one k-value (say, k_o) for which P returns "yes". Denote the difficulty function of P by $f(n,k)$; and set $h(n) = \max_k f(n,k)$. Then there exists a program P' that accepts as input any positive integer n, and returns, for each n, precisely that k_o; and whose difficulty function, f', satisfies $f'(n) \leq \sqrt{N}h(n)$ (where "$\leq$" is in the sense of difficulty functions).

The idea of this assertion is the following. We imagine that we have a family of needle-in-the-haystack challenges, one for each value of n. The program P answers this challenge in the following manner. For each fixed n, P will check whether or not a given k, with $0 \leq k \leq N-1$ is the needle. So, for fixed n, P is prepared to run a total of $N = 2^n$ checks on needle-candidates, in order to find this needle. The function h gives the difficulty of the hardest of those N checks. The hardest check could be the successful one (i.e., that for $k = k_o$), or it could be an unsuccessful one. Now, the program P' represents a shortcut for finding the needle. It accepts as input n (i.e.,which haystack-challenge is under consideration), and returns the needle (k_o) directly (not necessarily by doing the sort of exhaustive search of which P is so fond).

Let a program P, as in the assertion, be given. Here is one way to build a program P'. Given any n, let P' simply simulate the running of P on input (n,k), for all N possible values of k. Eventually, P' will find the k for which P returns "yes", and will report that k-value. This is the most naive possible P', for it merely piggybacks on P, using trial-and-error. Let us now determine the difficulty function, f', of this naive program P'. Fix n. Then the simulation of P on a candidate, (n,k) will require no more than $h(n)$ steps, and so, since P' must simulate P on a total of at most N such candidates, we shall have $f'(n) \leq Nh(n)$.

We conclude: If the inequality at the end of the assertion above were changed to "$f'(n) \leq Nh(n)$", then that assertion would be true (since the naive P' of the previous paragraph would do the trick). The assertion as it stands, then, says that there exists a shortcut P' that is *better*, in a suitable sense, than this naive P'. It says that you can always discover some way of finding the needle with difficulty not exceeding the maximum P-difficulty to check one candidate, multiplied by the *square root* of the number of candidates that P' would have to check. The assertion asserts, in other words, that

given any family of needle-challenges, and a way to meet those challenges by trial-and-error, then there exists a way to meet those challenges that is more efficient than trial-and-error, by a factor of $\sqrt{N}$.

Why this $\sqrt{N}$? It comes from the Grover construction! Imagine that we had somehow come up with a counterexample to the assertion above. That is, we have a program P of the indicated type, and there does not exists (or at least, there has not been found) any shortcut program P', in the sense of the assertion. Fix n, and consider the action of P on (n, k), for that n. Let us next imagine that we were able to simulate this action of P by a suitable unitary operator, V, on $H \otimes \cdots \otimes H$ (n times), and that the total "difficulty" required to apply this operator was just $h(n)$, i.e., the maximal difficulty that the ordinary program P encounters for the various (n, k), with this fixed n. And finally, let us suppose further that the additional difficulty of building the quantum system is sufficiently small that it may be disregarded. Then the Grover construction would find the needle (with very large probability) with a total difficulty of $\sqrt{N}h(n)$ (since, as we saw in Sect. 15, the computer would have to be run only $\sqrt{N}$ times). But we began with the assumption that this P is a counterexample to the assertion, i.e., that it is is such such that exists no shortcut P' with $f'(n) \leq \sqrt{N}h(n)$. What this means, in other words, is that there is no regular program that solves this problem more efficiently than does the Grover construction.

We conclude: A counterexample to the assertion above would provide a road map for finding (via Grover) an example in which a quantum-assisted computation is more efficient than any computation of the same problem without a quantum-assist. If the assertion were true, on the other hand, then this result would considerably diminish the prospects for using the Grover construction in this way.

We emphasize that the assertion above is a statement in mathematics: It does not involve quantum mechanics, nor vague idea about computation. It is either true or false. I have neither a proof nor a counterexample to this assertion. Below are three examples of (failed) attempts to construct a counterexample, which are intended to give a sense of how the assertion works.

For the first example, let, given n, the needle be $k_o = N/2$. Thus, program P would, on receiving (n, k), multiply k by 2, and see if the result is 2^n. The difficulty function is n (independent of k); and so we have $h(n) = n$. This program P is not a counterexample. Let P' be the program that accepts

positive integer n, computes $N/2 = (2^n)/2$, and returns that integer. The difficulty function for this P' is $f'(n) = n$, and so we certainly have $f'(n) \leq \sqrt{N}h(n)$. This candidate for a counterexample was hopeless right from the beginning. Anytime the structure of P is "do some computation involving n, and then check to see if the result matches k", then you will never end up with a counterexample. Program P' will overhear this strategy, and proceed to compute the needle k_o directly from n in the same manner, ending up with difficulty function given by $f'(n) = h(n)$, and thus satisfying the inequality of the assertion.

For the second example, let, given n, the needle k_o be the largest prime $\leq (N-1)$. Thus, program P would, on receiving (n, k), first check to see if k is prime (reporting "no" if it is not), then check the integers from $k+1$ to to $N-1$ for primeness (reporting "no" if any is prime), and otherwise reporting "yes". The difficulty function, $f(n, k)$, of this program has complicated k-dependence. But in any case, denote by $h(n)$ be the greatest difficulty encountered as k ranges from 0 to $(N-1)$. This program P is not a counterexample. Let P' be the program that works downward from $N-1$, checking each integer for primeness, and reporting the first prime it finds. The number of steps required by P' will be the same as for P to check candidate k_o, i.e., we have $f'(n) = f(n, k_o)$. Hence, $f' \leq h$, and so certainly the inequality of the assertion will be satisfied. This candidate for a counterexample was not much more promising. Any time the structure of P is "check to see if k is the largest integer $\leq (N-1)$ such that . . . ", then P' will overhear this strategy, and proceed to find the needle directly by starting at $(N-1)$ and working down. Similarly for "smallest"; and any other "-est" that P' can figure out how to exploit.

The third example is the following. For n any positive integer, denote by p_n the integer obtained by writing out the digits of π (314159265...), and stopping as soon as you arrive at the largest integer less than N^2. [For example, $p_5 = 314$.] Now let the needle, for this haystack n, be the following. If p_n is not a product of two primes, then $k_o = 0$. But if p_n is a product of two primes, then k_o is the smaller prime factor. [The idea here is that the digits of π are "pretty random"; and that it is (or at least, used to be) hard to find prime factors other than by trial-and-error.] Now, most of the time (i.e., for most n) p_n will have many factors, and in these cases it will be easy to find the needle. Program P' will have a field day in these cases, easily achieving $f'(n) \leq \sqrt{N}h(n)$. But every so often (at least, we *hope* so — we,

of course, have no theorem to this effect) p_n will turn out to be a product of two primes, and now the needle is harder to find. In this case, program P will have a relatively easy job of it: Given candidate k, P need only test to see whether or not k divides p_n. The shortcut program P', by contrast, has the duty to find the needle for this n — and it is hard to see how P' is going to do this other than testing various k to see if they divide p_n. So, here is an example in which (at least, *sometimes*) there doesn't appear a viable shortcut over the method of trial-and-error. So, is this a counterexample to the assertion? Probably not. The problem is that P must do more than merely check whether k divides p_n — it must also check whether or not p_n is a product of two primes (in order to know whether or not $k_o = 0$). The difficulty (for P) of doing this is comparable to the difficulty P' experiences in finding the needle in this case.

Either the assertion above is true; or it is false. It would be of great help in thinking about this subject, in my opinion, if we knew which.

17 Quantum-Assisted Computing

The discussion of the previous two sections suggests that the use of quantum mechanics may indeed gain efficiency for certain computations. But there remain at least three issues. First, as discussed in Sect. 16.4, our ability to apply quantum mechanics to specific problems appears to depend on finding a suitable technique for converting conventional computer programs to unitary operators. By "suitable", we mean a technique that results in no substantial loss of efficiency, and is such that no entanglements are created with any scratch Hilbert spaces that must be introduced. Second, we must make allowance for the fact that, while programs act on arbitrary strings (and thus are suitable for computing real problems), our unitary operators always act on *finite* tensor products of H's. And, finally, we must find a suitable definition of "difficulty" for unitary operators. We now introduce a general framework for computations using quantum mechanics, a scheme that, among other things, addresses these three issues.

Fix, once and for all, the following objects: i) a finite-dimensional Hilbert space H, ii) a unit vector $|\psi_o\rangle$ in H, iii) a finite list of unitary operators, each of which acts on some finite tensor product, $H \otimes \cdots \otimes H$, of H's, and iv) a

finite list of projection operators[14], each of which acts on some finite tensor product of H's. The individual unitary and projection operators in these lists may operate on tensor products with different numbers of H-factors, e.g., some may act on a single H, some on $H \otimes H$, etc. We label each unitary operator of ii) and each projection operator of iii) by a nonempty string (e.g., as U_S and $P_{S'}$, respectively); and, for later convenience, we do not use the same string to label both a unitary and a projection operator. [We shall later impose a further condition on this arrangement; but for the moment it is convenient to keep things general.]

We now introduce some terminology. First, we introduce a separator-character, $*$, in the manner we have done, occasionally, before. Next, we call a string $\tilde{S}$ a *unitary operation* if it is of the form $\check{S} * S_1 * \cdots * S_k$, i.e., consists of $(k+1)$ strings (each nonempty and containing no $*$; and with $S_1 \cdots S_k$ distinct), such that: The first of these strings, $\check{S}$, labels some unitary operator, $U_{\check{S}}$, in our list, and that $U_{\check{S}}$ acts on a tensor product of precisely k factors of H. Thus, beyond the first string, it is only the *number* of additional strings, and not what those strings are, that counts. For example, if, among the unitary operators, there is one labeled U_{8k}, and if it acts on $H \otimes H$, then "$8k * yzr * 8\$9Q$" would be a unitary operation; whereas "$8k * yzr$" would not. And, similarly, we call string $\tilde{S}$ a *projection operation* if it is again of the above form, $\tilde{S} = \check{S} * S_1 * \cdots * S_k$, such that $P_{\check{S}}$ appears on our list of projection operators, and it acts on the tensor product of exactly k factors of H. Note that no string is both a unitary operation and a projection operation; and that the problem of deciding whether a string $\tilde{S}$ is a unitary operation, a projection operation, or neither is computable and has difficulty function $L(\tilde{S})$.

We now introduce a new computer language. As before, we have storage locations, each of which is labeled by a string and contains a string (where, as before, $C(S)$ denotes the string contained in location S). There is a total of seven commands in this language, consisting of the five we introduced in Sect. 12 — INPUT, OUTPUT, APPEND, DELETE, and IF — together with two new ones:

6. APPLY $C(S)$.

7. OBSERVE $C(S)$, APPEND RESULT TO $C(S')$.

[14] A *projection* operator, P, is self-adjoint operator satisfying $P \circ P = P$, i.e., a self-adjoint operator having no eigenvalues other than 0 and 1.

where, as before, S and S' denote arbitrary strings. Here is what these commands "do". In addition to the storage locations, there will now be a separate quantum system. The Hilbert space, $\mathcal{H}$, of states of this system will be, at any one moment during the operation of the computer, some tensor product of H's, where each factor of H is this tensor product is labeled by a string. Thus, we might have, at one moment, $\mathcal{H} = H_8 \otimes H_{abc} \otimes H_{Q3}$, a tensor product of three copies of H. The state of this quantum system, at that moment, will be given by some vector, say $|\Psi\rangle$, in the Hilbert space $\mathcal{H}$. Now, here is what is to be done in response to the command APPLY $C(S)$):

1. If $C(S)$ is not a unitary operation, then do nothing.

2. If $C(S)$ ($= \check{S} * S_1 * \cdots * S_k$, say) *is* a unitary operation, and each of the strings $S_1, \cdots, S_k$ is already represented by an H-factor in the tensor product that is $\mathcal{H}$, then apply to the state $|\Psi\rangle \in \mathcal{H}$ the unitary operator $U_{\check{S}}$ on $H_{S_1} \otimes \cdots \otimes H_{S_k}$. [That is, the unitary operator that is applied to $\mathcal{H}$ is the operator $U_{\check{S}}$ applied to the factors $H_{S_1} \otimes \cdots \otimes H_{S_k}$, and "$I$" applied to the remaining factors.]

3. If $C(S)$ ($= \check{S} * S_1 * \cdots * S_k$, say) is a unitary operation, and some of the strings $S_1, \cdots, S_k$ are not represented by H-factors in the tensor product that is $\mathcal{H}$, then proceed as follows. First, enlarge $\mathcal{H}$ to include those H-factors (i.e., replace $\mathcal{H}$ by its tensor product with the missing H_S). Next, replace the state $|\Psi\rangle$ by the result of taking the tensor product of this state with one copy of $|\psi_o\rangle$ for each new H-factor introduced. And finally, apply $U_{\check{S}}$ to this state in $\mathcal{H}$ (so enlarged) as in instruction 2.

Here is an example of these rules. Let, at some moment, $\mathcal{H} = H_{19} \otimes H_{yzr}$, let the state be $|\Psi\rangle \in \mathcal{H}$, let our list of unitary operators include an operator U_{8k} that acts on $H \otimes H$, and let $C(S) = 8k * yzr * Q9$. Then APPLY $C(S)$ would replace this $\mathcal{H}$ by $H_{19} \otimes H_{yzr} \otimes H_{Q9}$ and state $|\Psi\rangle$ by $|\Psi\rangle|\psi_o\rangle$; and would then apply $I \otimes U_{8k}$ to this state. Similar rules apply to OBSERVE $C(S)$, APPEND RESULT TO $C(S')$. If $C(S)$ is not a projection operation, do nothing. Otherwise, proceed as follows. First, enlarge $\mathcal{H}$ by including, as necessary, additional H-factors, labeled by any strings in $C(S)$ (other than the first) not already so represented. Next, replace the state $|\Psi\rangle$ by that state in this enlarged Hilbert space obtained by taking one tensor product with a $|\psi_o\rangle$ for each new factor of H. Then, make an observation on this new state of this expanded Hilbert space, via the self-adjoint operator $P_{\check{S}}$.

The result of this observation must be either 0 or 1, since $P_{\check{S}}$ is a projection. Append this result (suitably encoded, if necessary) to the string in location $C(S')$. [Usually, we would have previously SET $C(S') = \emptyset$, to avoid clutter.] After this OBSERVation, the state of our quantum system will, of course, be replaced by its projection into the appropriate eigenspace, i.e., by either $P_{\check{S}}$ or $(I - P_{\check{S}})$ applied to that state, according to whether the observation resulted in 1 or 0, respectively.

In physical terms, what we are doing here is quite simple. We have some basic quantum system, described by Hilbert space H. What we call a projection operation, for example, is a string that describes which projection operator is to be applied, and to which combination of copies the basic system. If any of the required copies are missing, we simply supply them, by ordering new copies of that system (which come with state $|\psi_o\rangle$ preinstalled) through the catalog, and placing those new system-copies next to the old system-copies. When all the necessary copies of our basic system have been assembled, we observe via the appropriate (projection) operator, and append the result to location S'. We remark that no generality has been lost by our making all observations via projection operators (rather than the more general self-adjoint operators). This is a consequence of the following fact: Every self-adjoint operator A on a finite-dimensional Hilbert space can be written as a sum, $a_1P_1 + \cdots + a_sP_s$, where the a_i are the eigenvalues of A, and the P_i project into the corresponding eigenspaces (so, in particular, the various P_i commute with each other). By virtue of this fact, we may, instead of observing via self-adjoint A, observe via each the P_i, noting that, by commutativity, the order of the latter observations is irrelevant. These two operations will always produce the same result, in terms of the outcomes and their probabilities as well as the final state of the system.

A *quantum-assisted program* is a finite, ordered list of commands, the first command of which is INPUT and the last of which is OUTPUT, each of these appearing nowhere else in the program. We run a program just as before, starting with each storage location containing the empty string; and with $\mathcal{H}$ the (one-dimensional Hilbert space of) the complexes. We then execute the commands in order, except as directed by IF. The various commands will then manipulate strings; or operate on, expand, or observe $\mathcal{H}$. So, for example, the first APPLY or OBSERVE command will require that $\mathcal{H}$ be expanded to include the appropriate copies of H as a tensor product. If and when the program reaches OUTPUT, it halts, allowing us to read the output string.

When a given quantum-assisted program is run with a given input string, it must either halt, with some output string returned, or never halt (which we denote, as before, by "$*$"). From the laws of quantum mechanics, there will be probabilities for these various outcomes, i.e., we will have a probability distribution, p, on $\mathcal{S} \cup \{*\}$. We say that a quantum-assisted program *computes* problem π if, for every input string S, the program, run on that string, has $p(*) = 0$, and $p(\pi(S)) > p(S')$ for every $S' \neq \pi(S)$. That is, the probability of not halting must be zero (although, as we have already seen, it may still be possible that the program fail to halt), and the probability of the "correct answer", $\pi(S)$, must exceed that of every other possible output.

As an example of all this, let us consider the Grover construction. Let H be the two-dimensional Hilbert space of spin states of an electron, and let $|\psi_o\rangle$ be the state we earlier designated $|1\rangle$. Let U_1 act on H by $U_1|1\rangle = \frac{1}{\sqrt{2}}\{|1\rangle + |0\rangle\}$, $U_1|0\rangle = \frac{1}{\sqrt{2}}\{|1\rangle - |0\rangle\}$; U_2 by $U_2|1\rangle = -|1\rangle$, $U_2|0\rangle = |0\rangle$; and U_3 by $U_3|1\rangle = i|1\rangle$, $U_3|0\rangle = |0\rangle$. [A few more U's on H might also be required.] Let U_4 act on $H \otimes H$ by $U_4 = \exp(i(\pi/2)\vec{\sigma_1} \cdot \vec{\sigma_2})$. Finally, let there be a single projection operator, P_5, acting on a single H via $P_5|1\rangle = |1\rangle$, $P_5|0\rangle = 0$.

The program will accept as input a positive integer n. It will then place, in location S, the string "$1*1$", execute APPLY C(S), then place "$1*2$" there, then execute APPLY C(S) again, etc, a total of n times. This will set up our Hilbert space, $H_{\rm in} = H_1 \otimes \cdots \otimes H_n$, with each H_i-state given by $\frac{1}{\sqrt{2}}\{|1\rangle + |0\rangle\}$. [Note how we set up initial states, by starting with $|\psi_o\rangle = |1\rangle$ and applying the appropriate operator.] We next introduce three subroutines. The first applies to this $H_{\rm in}$ the operator $V_{\rm in}$ (possibly by introducing, into $\mathcal{H}$, an auxiliary Hilbert space for scratch work). The second subroutine applies W: We use our U's to construct the Toffoli operator, and then we apply it to suitable combinations of H-factors, introducing the auxiliary Hilbert space of Sect. 16.2. And finally, we introduce a subroutine that observes $H_{\rm in}$ via the k-basis. This will, e.g., set $C(S) = 5 * 1$, execute OBSERVE $C(S)$, APPEND RESULT TO $C(S')$ and multiply the result (in $C(S')$) by 2^{n-1}; then set $C(S) = 5 * 2$, execute OBSERVE $C(S)$, APPEND RESULT TO $C(S')$, multiply the result (in $C(S')$) by 2^{n-2} and add to the previous result; and so on, for n times around. This subroutine, then, will end up producing some k-value, where $0 \leq k \leq N - 1$, stored in some memory location.

There are at least three different programs that could be constructed from these pieces. First, we could simply apply the $V_{\rm in}$ subroutine, then

the W subroutine, and finally the observation subroutine, storing the result in location S'. Then execute OUTPUT $C(S')$. For this program, we would have $p(*) = 0$ (and, in fact, outcome $*$ would be impossible), $p(k_o) = 9/N$ (for large N), with the probabilities for the other k-values correspondingly reduced. This quantum-assisted program indeed computes our problem. For the second program, we would run $-WV_{\text{in}}$ the correct number (which is within one or two of $(\pi/4)\sqrt{N}$) of times (where the regular commands and storage locations are used to keep track of these numbers), then run the observation subroutine once, and finally execute OUTPUT to return whatever k that results. For this program, we would again have $p(*) = 0$ (and, again, outcome $*$ would be impossible). Now, however, $p(k_o) > 1 - 1/N$, with the remaining k taking up the rest of the probabilities. This quantum-assisted program also computes our problem. For our final program, we first do the computation of the previous program, but instead of reporting the k that results from the observation subroutine, we instead run the non-quantum check program on that k, to find out if if is indeed the needle. If it is, we report it. Otherwise, go through the entire procedure again, finding a new k and checking that one. Continue in this way until we find the k that checks out as the needle. For this program, we again have $p(*) = 0$, but now the outcome $*$ is possible (for we could to go on indefinitely, being unlucky time after time). We also have $p(k_o) = 1$; and $p(k) = 0$ for all other k. Thus, this quantum-assisted program also computes our problem.

Thus, we have three separate quantum-assisted programs, each of which computes the problem (whatever it is). In these examples, the OBSERVE commands generally come after the APPLY commands have already been executed. But that, of course, needn't be the case in general: These commands could very well be intermingled. Note that the brain of this program is the original five commands and the storage locations with which they interact: These keep track of what is going on, decide when APPLY and OBSERVE are to be carried out, decide what to do with the results, handle the INPUT and OUTPUT, etc. The Hilbert space of the quantum system serves as a glorified storage register, into which we may place data (via APPLY), within which we may manipulate data (via APPLY), and from which we may extract data (via OBSERVE). We could also arrange, in effect, for there to be several different Hilbert spaces to handle data. These would be represented by different parts of the tensor product that is $\mathcal{H}$, and we would simply manipulate and access whatever part we wish to use at any one time, ignoring (i.e., applying the

identity to, and not observing) the other parts. At any one moment in the program, the Hilbert space of states of the quantum system, $\mathcal{H}$ is a finite tensor product of H's; although, of course, the number of H's in this product is not limited. Had we, for example, replaced each "$C(S)$" in the APPLY and OBSERVE commands with "S", then there would, for any given program, be a limit (independent of the input string) on this number. Replacing each "$C(S)$" in these commands by "$C(C(S))$" would not make any difference, since we have a subroutine SET $C(S') = C(C(S))$. Note that the structure of quantum-assisted programs does not require, necessarily, that entanglements be avoided or undone. We simply APPLY certain unitary operators and OBSERVE certain projection operators; and whatever follows follows, whether there is entanglement or not. [Of course, it may be necessary to avoid entanglement in order that the program compute what we want it to compute.]

18 Quantum-Assisted Computability

We introduced, in the previous section, the notion of a problem being computable by a quantum-assisted computer; and we already have, from Sect. 6, the notion of a problem being computable by a regular computer. Clearly, every regular-computable problem is quantum-assisted-computable (since every regular program *is* a quantum-assisted program, just one happening to have no APPLY or OBSERVE commands). Is the converse true? That is, is it true that every problem that is computable with quantum-assist is also computable in the regular sense?

This converse, as stated, is false. Here is an example. Let the Hilbert space H be 2-dimensional, with basis $|\psi_o\rangle$ (the initial state) and $|\psi_o\rangle^\perp$. Let there be just one unitary operator in the list, acting on a single copy of H by $U_\theta|\psi\rangle = |\psi_o\rangle\ \cos\theta + |\psi_o\rangle^\perp \sin\theta,\ U_\theta|\psi_o\rangle^\perp = |\psi_o\rangle^\perp \cos\theta - |\psi_o\rangle\ \sin\theta$, where θ is some fixed number. Let there be just one projection operator in the list, also acting on a single copy of H, by $P|\psi_o\rangle = |\psi_o\rangle, P|\psi_o\rangle^\perp = 0$. [You will note that this is a pretty poor excuse for quantum-assistance: There are no operators in our lists that act on two or more H's, and so we will never produce, by means of these operators, interestingly entangled tensor-product states.] The program accepts as input a positive integer n, and proceeds as follows. It first executes APPLY U_θ to H_1; OBSERVE P on H_1, and accumulate the result (0 or 1) in $C(a)$. It then repeats this subroutine,

with "H_1" replaced by "H_2", and so on, all the way up to "H_{n^4}". At this point, $C(a)$ will contain an integer (the number of times "1" was returned by OBSERVE during all n^4 runs of the subroutine). Finally, the program returns, via OUTPUT, the rational number $C(a)/n^4$. What is this program doing? Well, on each run through the subroutine, either 1 will be added to the integer in $C(a)$ (probability $\cos^2\theta$), or it will not (probability $\sin^2\theta$). Thus, what the program returns at the end is a Monte-Carlo estimate of the value of $\cos^2\theta$, based on these n^4 runs. But the fractional error in such an estimate goes down, as the number of runs increases, as the reciprocal of the square root of that number. Hence, the estimate this program produces will have an error of the order of $1/n^2$. So, except possibly for the first few n-values, we have a high probability that our estimate will be within $1/n$ of the actual value of $\cos^2\theta$.

What this program above does, in other words, is compute the number $\cos^2\theta$, in the sense of Sect. 7. What we have shown, then, is that for any number θ we can write a quantum-assisted program that computes $\cos^2\theta$. Now choose for θ that value such that $\cos^2\theta = c$, where c denotes the non-computable number given by Eqn. (2) of Sect. 7. Thus, we have produced a quantum-assisted program that computes a number that is not computable with any regular program.

The discussion of the previous two paragraphs will fool nobody. It is absurd to take seriously a unitary operator U_θ that claims to carry out a rotation through a noncomputable angle: We would not expect to be able to buy and operate a machine that would apply any such U_θ to any real quantum system. Suppose, for example, that we acquire a machine that is capable of carrying out a rotation in the $|\psi_o\rangle - |\psi_o\rangle^\perp$ plane through any angle θ, where the value of θ is set by adjusting a knob. Then, as we fine-tune this machine, we shall be called upon to determine, more and more precisely, what the number θ is; i.e., we will have to decide whether or not more and more complicated Turing machines will halt. One of these, for example, is the machine that searches for a counterexample to the Goldbach conjecture — and so, at this point, the proper adjustment of the knob will require that we settle this conjecture, one way or the other. And, as soon as we are finished with this one, we will be asked to resolve some other, even harder, conjecture in mathematics. How, in this atmosphere, are ever going to get this experiment finished? It is silly to call this quandary a "piece of laboratory equipment". Note that the fact that there are rational numbers

arbitrarily close to the noncomputable number c is of no help to us here. The issue is one of *determining* what c is (in practice, in the lab), not one of *approximating* c (in principle, in mathematics-land). To know c within 1%, let us say, will tell us whether or not the Goldbach conjecture is true. How helpful, in this circumstance, is it to be reassured that there does indeed exist a rational number within one-tenth of 1% of c?

> **Exercise.** Consider the unitary operator U_θ above, but let us select the angle θ "randomly, by spinning, and then stopping the wheel". Then the probability is one that we shall end up with a noncomputable number (since the computable numbers in $[0, 2\pi]$ are measurable, with measure zero). So here is an example in which a quantum-assisted computer computes a (regularly-) noncomputable number. Respond.

So, we might ask, which unitary operators are, and which are not, "physically realistic"? In fact, the problem goes a little deeper than even this question suggests. Consider, for example, two unitary operators, U and U', each of which carries out a 90^o rotation in the (3-dimensional, say) Hilbert space H, but such that the planes in which these rotations take place make angle c with each other. Although each of these two unitary operators, by itself, is quite innocent-looking, together they allow, in the same manner as above, a computation of c. The lesson here is that we cannot look at the state $|\psi_o\rangle$, the unitary operators that appear on our list, and the projection operators that appear on our list, in isolation. It is the *entire list* — consisting of $|\psi_o\rangle$, the U's and the P's; all taken together — that must rise or fall. So, we might ask, how do we identify which such lists are, and which are not, physically realistic?

Here is a possible answer to this question. Let us imagine that the box in which the Hilbert space H is shipped has printed on it a standard basis for this Hilbert space, to be used for reference purposes. Then from this basis we acquire a standard basis for each tensor product, $H \otimes \cdots \otimes H$. Now, before purchasing a unitary operator U (on some tensor product of H's), we want to know what it is we are buying. This is to be provided by the manufacturer, in the form of a program, printed in the owner's manual, that accepts as input any positive integer n, and returns rational numbers each within $1/n$ of the respective matrix element of U in this standard basis. We call this a program

that *computes* U. Without such a program, we simply don't know what U "is". We have, similarly, the notion of a program that *computes* a projection operator (on a tensor product of H's), and of a program that *computes* the initial state $|\psi_o\rangle$. We now regard H, state $|\psi_o\rangle$, and the lists of unitary and projection operators as "physically realistic" if, for some H-basis, there exist programs that compute all these objects. This appears to be a rather mild condition.

Some terminology will allow us to formulate the idea of the previous paragraph more neatly. Fix, as before, i) a finite-dimensional Hilbert space H, ii) a unit vector $|\psi_o\rangle$ therein, iii) a finite list of unitary operators (labeled by strings) on various H-tensor products, and iv) a finite list of projection operators (labeled by strings) on various H-tensor products. We call a string $\mathbf{S}$ a *history* if it is of the form $S_1 * *S_2 * * \cdots * *S_k$, where each of the strings $S_1, \cdots, S_k$ is either i) a unitary operation (string), or ii) a projection operation (string) to which either "$*0$" or "$*1$" has been appended. Thus, a typical history string might be "$k9 * yzr * 0 * *B * xx * ABC$": The rightmost entry represents the unitary operator U_B applied to $H_{xx} \otimes H_{ABC}$; the other entry the projection operator P_{k9} applied to H_{yzr}; with "$*0$" appended.

Now consider the running of some quantum-assisted program. Each time an APPLY command is executed, some things will be done with respect to the Hilbert space $\mathcal{H}$: This Hilbert space will be expanded (if necessary) by taking additional tensor products with H-copies; the state $|\Psi\rangle$ will be adjusted (if necessary) with $\otimes|\psi_o\rangle$'s to lie in this expanded Hilbert space; and a certain unitary operator will be applied to this state. The same goes for an OBSERVE command, except that in this case either the projection operator P (from the list) is applied to the state (the case in which the observation yielded "1"), or the projection operator $(I-P)$ is applied (observation yielded "0"). The other commands, INPUT, OUTPUT, DELETE, APPEND, and IF, do nothing with respect to $\mathcal{H}$. Thus, as of any one moment during the running of the program, $\mathcal{H}$ will have been subjected to some finite number of such operations, in some order. But this is precisely the information contained in a history. In other words, a history string provides a complete summary of what has happened with respect to $\mathcal{H}$ as of a certain moment. Perhaps "virtual history" would be a better term, for we admit as histories *all* strings formed by the rules of the previous paragraph, whether or not they happen to represent what has actually occurred. From the history string we may determine what the Hilbert space $\mathcal{H}$ is at that moment, what state

(in $\mathcal{H}$) the quantum system is in, and what information (0's and 1's) has been passed so far from the quantum system to storage locations. From the history "$\mathbf{S} = k9 * yzr * 0 * *B * xx * ABC$" of the previous paragraph, for example, we would determine that $\mathcal{H} = H_{yzr} \otimes H_{xx} \otimes H_{ABC}$, that the state is $((I - P_{k9}) \otimes I \otimes I)((I \otimes U_B)(|\psi_o\rangle|\psi_o\rangle|\psi_o\rangle))$, and that value "0" was returned on the execution of the one OBSERVE command. The idea, in short, is to reflect the entirety of the quantum part of quantum-assisted computing by a string, something we can easily dissect.

For $\mathbf{S}$ any history, denote by $\gamma(\mathbf{S})$ the squared norm of the state (as determined by $\mathbf{S}$) in the Hilbert space (as determined by $\mathbf{S}$). Thus, for example, we have $\gamma(\mathbf{S}) = 1$ if the history $\mathbf{S}$ contains no projection operations (since $|\psi_o\rangle$ is unit, and unitary operators are norm-preserving); and, for a general history $\mathbf{S}$, $0 \leq \gamma(\mathbf{S}) \leq 1$. This real-valued function γ on histories carries all the relevant information about the workings of quantum mechanics within our quantum-assisted language, in a sense that will become clear shortly.

Now suppose that, with respect to some standard basis for H, there exist a program that computes the initial state $|\psi_o\rangle$, as well as ones that compute each of the unitary and projection operators in our lists, in the sense described above. Then we may combine these to produce a program that, given any history $\mathbf{S}$, will compute the components, in terms of this standard basis, of the state determined by this $\mathbf{S}$. It is now a simple matter to write a program that computes the function γ, in the following sense. This program accepts as input any history $\mathbf{S}$ and positive integer n, and returns some rational number within $1/n$ of $\gamma(\mathbf{S})$. The existence of such a program implies, of course, that each number $\gamma(\mathbf{S})$ is computable, but it also implies somewhat more: It means that there is a *single* rule that suffices to provide an approximation for *every* $\gamma(\mathbf{S})$.

We now return to the issue raised at the beginning of this section. We claim: Any number that is computable by a quantum-assisted program, using computable initial state and operators, is also computable by a regular program. The proof, much like that of the similar result for probabilistic programs, is by simulation.

Fix H, and let $|\psi_o\rangle$ and the labeled unitary and projection operators be computable, in the sense above. Now fix a quantum-assisted program P_{quant}, together with the input string, S_{in}, on which this program is to be run. We are going to construct a regular program, P_{reg}, that will simulate the running of

P_{quant} on S_{in}. Suppose that P_{quant} has been run for a few steps, encountering an APPLY command or two, but no OBSERVE command. Then the entire state of the computer (including the quantum system) can be expressed, as of this moment, by giving three pieces of information: i) which command in the list P_{quant} is slated to be executed next; ii) which string is stored in each (nonempty) storage location; and iii) the history, $\mathbf{S}$, of the quantum system. Let us now carry P_{quant} through the next step (say, a non-OBSERVE one). To simulate this step, we update the three pieces of information in the obvious way: For i), we now indicate what is the new next command; for ii) we make an adjustment (required only if that step was APPEND or DELETE; and then to only one of the stored strings) to reflect the new string-storage; and for iii) we add an entry (required only if that step was APPLY) to the history. In this way, we continue our simulation of P_{quant}, step by (non-OBSERVE) step. What happens when we reach an OBSERVE command? Now there will be two possible outcomes, depending on whether the observation returns 1 or 0. We reflect this state of affairs by splitting our description into two branches, each of which carries three pieces of information as above. In one branch, corresponding to the observation returning "1", the three pieces of information read: for i), the next command to be executed; for ii), the same stored strings, but with "1" appended to one particular string; and, for iii) the same history string, but with the addition a certain projection operation and " $*1$". In the other branch, we enter, similarly, the three pieces of information appropriate to the case in which observation returns "0". We now continue to simulate the behavior of P_{quant} in each of these two branches separately. As more OBSERVE commands are executed, the number of branches will grow, as will the burden of separately simulating what happens within each branch. But at every stage in this simulation, we shall have a finite number of branches, each described by just these three pieces of information.

So, P_{reg} simulates P_{quant}, in this way. Every so often, one of the branches being simulated by P_{reg} will reach an OUTPUT command (after which there is nothing more to simulate). When that occurs, the program P_{reg} reads the output string S and the final history $\mathbf{S}$ for that branch, stores this information in a special section, and drops that branch from further consideration. [We, of course, know that $\gamma(\mathbf{S})$ is the probability that the actual program P_{quant} will take this branch, reaching this OUTPUT and returning this S.] Thus, as the simulation by P_{reg} continues, the list of string-history pairs in this special

section will grow. Our program P_{reg} will include, furthermore, a subroutine, which accepts as input a positive integer n, and operates as follows. The subroutine goes to this special section, takes each of the history strings in that section, and computes γ of that history, within error $1/n$ (here using the program that we constructed from the reference manuals). It then totals these numbers, for each output string listed in that section. Finally, the subroutine checks to see if any one output string, say S, has emerged as a clear winner (i.e., is such that no other string S' will ever be able to achieve a total exceeding that for S, even if we allocate to S' all the so-far unallocated probability, and even if we assume that all the $1/n$-errors in these probabilities are resolved in S''s favor). If the subroutine finds such a clear winner S, then it causes P_{reg} to halt immediately, giving that S as output. If there is no clear winner, then the subroutine returns P_{reg} to its simulation.

The full program P_{reg} now operates as follows: Every so often (say, after every hundred steps of simulation), P_{reg} runs the subroutine, using an n-value one higher than that for the previous run. So, this program P_{reg} will continue to run in this way: continuing to carry out the simulation of P_{quant}, continuing to store the results for halted branches in the special section, and continuing to make ever-finer checks on the status of that special section. Now suppose that the program P_{quant} computes a problem π, i.e., that, for every input string S_{in}, P_{quant} has probability zero of failing to halt, and has probability of halting with output $\pi(S_{\text{in}})$ that exceeds that of every other possible output. In this case, our simulation P_{reg} must eventually halt, for eventually it will have accounted for sufficient probability to identify $\pi(S_{\text{in}})$ as the clear winner. At this point, P_{reg} will return $\pi(S_{\text{in}})$.

What we have shown, then, is that, given any quantum-assisted program that computes a problem, we can, by simulating it in this manner, build a regular program that computes the same problem. Note that P_{reg} *always* halts, giving the correct answer, $\pi(S_{in})$. The quantum-assisted program P_{quant}, by contrast, gives this answer only probabilistically. Note also that this argument makes essential use of the program that computes the function γ. In any case, we conclude that any problem that is computable by a quantum-assisted program, with computable $|\psi_o\rangle$ and operators, is also computable by a regular program.

The result of this section is of perhaps only mild interest. What is important, and what we shall use extensively in the what follows, is three notions: that of a history; that of the function γ; and the present strategy for simu-

lating a quantum assisted computation.

19 Quantum-Assisted Difficulty Functions

In Sect. 10, we defined a difficulty function for any program that computes some problem, π. This positive, real-valued function represents the amount of "computer time", as a function of the input string S_{in}, required to compute $\pi(S_{\text{in}})$. We now wish to do the same thing for any quantum-assisted program that computes a problem. We shall do so in two steps. First, we assign a difficulty to each individual command in our quantum-assisted language. This will entail a certain restriction on the unitary and projection operators that go into the language. Second, we adapt the notion of a difficulty function to take into account the fact that quantum-assisted computing provides answers only probabilistically.

There are seven types of commands that may appear in a quantum-assisted program. Five of these — INPUT, OUTPUT, APPEND, DELETE, and IF — are the original commands we introduced in Sect. 12; and it is natural to assign to these commands the difficulties we already chose earlier. But what of the two new commands — APPLY and OBSERVE? Note that we have finite lists of the unitary and projection operators that are permitted to appear in these commands. So, in effect, each of APPLY and OBSERVE represents a finite number of physical operations. A natural choice would thus seem to be: Assign, to each APPLY and OBSERVE command, difficulty one. But, unfortunately, things are not that simple, as the following example illustrates.

Consider a problem, π, that accepts as input a positive integer n, returning either 0 or 2; and is such that every program that computes this π has difficulty function f that grows very quickly with n (say, faster than 2 to the power of 2 to the power of 2, etc, for n iterations). We have seen in Sect. 11 that there does indeed exist such a (computable) problem. Now set $c' = \sum_n \pi(n)/3^n$, a certain real number. Thus, this c' is constructed in the same manner as the noncomputable number c of Sect 6, but, in contrast to that c, is a computable number (since the problem π is). The point, however, is that c' is hard to compute: The number of computer-steps required to approximate c' within $1/n$ grows very quickly with n. Let us now consider a quantum-assisted language in which the Hilbert space H is 2-dimensional,

and the list of unitary operators includes a U_θ that applies a rotation, to a single H, through angle θ, just as in the previous section. Now, however, we choose $\cos^2\theta = c'$. We next write a quantum-assisted program in this language similar to that of the previous section. That is, this program repeatedly applies U_θ and makes an observation, resulting in a Monte-Carlo estimate of $\cos^2\theta$. In order to compute $\pi(n)$, we must estimate the value of c' to within $1/(2*3^n)$. But the error in a Monte-Carlo estimate decreases as the reciprocal of the square root of the number of runs. Thus, we need about $(2*3^n)^2 = 4*9^n$ applications of U_θ to have a reasonable chance of recovering the value of $\pi(n)$. For given n, carry out 10^{n+1} applications (just to be on the safe side): Then we shall have a probability of correctly determining $\pi(n)$ that is high and increasing with n. Thus, we have written a quantum-assisted program that computes this problem π with, presumably, difficulty function 10^{n+1} — much less than the difficulty function of any regular program that computes π.

This was a foolish argument in the previous section, and it is no better this time around. What this argument does show, however, is that we must be prepared to exercise some care as to which unitary (and projection) operators will be allowed in quantum-assisted programs, and as to what their difficulties are to be.

It is tempting to take the position that, since this U_θ is apparently such a terribly difficult operator, we can make things right by merely assigning a large difficulty to to the corresponding APPLY command. But this isn't going to work: There is just one of these U_θ's, and just one corresponding APPLY command, and so there is just a one number for us to assign. Changing the difficulty of this single command from 1 to 1000, for example, will not undermine the above argument. In fact, it appears that any attempt to preserve this particular U_θ in our list of unitary operators will return us to the issue of how we take into account errors. What is problematic about this operator U_θ appears only in attempts to "approximate" it. Imagine a new kind of quantum-assisted program that, when commanded to APPLY this U_θ, actually applies an operator that is only a rough approximation to U_θ, but which (by virtue of the roughness of the approximation) is also low in difficulty. If and when, as the running of the program proceeds, a more accurate application of U_θ becomes necessary, then our program would go back and redo the original APPLY command, but this time applying something that is closer to the actual U_θ (and carries a larger difficulty).

So, it appears that the only way we can avoid a very complicated programming environment, in which errors must constantly be taken into account, is to banish this U_θ from out list of unitary operators. But this is a slippery slope: Will there be allowed other unitary operators, based on numbers that are a little easier to compute, but still pretty hard? Where do we draw the line? Our approach will be to go ahead and slide down the slope, i.e., to rule out all but the "simplest" U's.

Fix an m-dimensional Hilbert space H, a unit vector $|\psi_o\rangle$ in H, and finite lists of unitary and of projection operators, each acting on on some finite tensor product of H's. We say that this arrangement is *simple* if, for every history string $\mathbf{S}$, the number $\gamma(\mathbf{S})$ is rational; and furthermore there exists a (regular) program that computes this γ, with difficulty function f satisfying

$$f(\mathbf{S}) \leq N_{\rm op}(\mathbf{S})\ m^{N_{\rm H}(\mathbf{S})}. \tag{13}$$

Here, $N_{\rm op}(\mathbf{S})$ denotes the total number of (unitary or projection) operations represented by the history string $\mathbf{S}$, $N_{\rm H}$ denotes the total number of copies of the Hilbert space H that appear in the final tensor product (within which $\gamma(\mathbf{S})$ is computed), and m is the dimension of H. Note that simplicity implies immediately that γ is computable, in the sense of the previous section.

Here is why this definition is what it is. Fix some basis for H, say $|\alpha_1\rangle, |\alpha_2\rangle, \cdots, |\alpha_m\rangle$. Then, as we have seen, we may construct from this H-basis a basis for every tensor product, $H \otimes H \otimes \cdots \otimes H$, of H's. For n H's in the tensor product, this basis will contain m^n vectors, each of which is a product of some n vectors in our H-basis, e.g., $|\alpha_i\rangle|\alpha_j\rangle \cdots |\alpha_k\rangle$. Now suppose that, with respect to this basis, the components of $|\psi_o\rangle$ and of all the unitary and projection operators in our list are rational numbers. This is about as simple as $|\psi_o\rangle$ and the U's and P's could possibly be. Now, in this case each value of γ will certainly be rational. Furthermore, we can easily write a program that computes γ. This program would take the history $\mathbf{S}$ and express explicitly, in terms of our basis, the result of applying in succession each operation contained in $\mathbf{S}$. The program then takes the resulting final state, again expressed in terms of components in this basis, and computes its squared norm.

What is the difficulty function of this program? Consider one of the operations — say, application of some unitary U — in the history string $\mathbf{S}$, and suppose that, at the point at which this U is applied, the Hilbert space is

a tensor product of n copies of H. Then, at that point, the dimension of this Hilbert space will be m^n, and so the current state will have m^n components, and so the record of this state in our program will require that m^n entries be stored. To apply the operator U to this state will entail replacing each of these entries by a linear combination of other entries. That is, in order to compute the effect of this U we shall have to carry out a number of arithmetic operations given by a small multiple of m^n. Note — and this is a key point — that there is no savings from the fact that U actually operates on a small number of H's in that tensor product. For example, say that H has basis $|0\rangle, |1\rangle$, and let U act on a single H, by $U|1\rangle = \frac{4}{5}|1\rangle + \frac{3}{5}|0\rangle$, $U|0\rangle = \frac{4}{5}|0\rangle - \frac{3}{5}|1\rangle$. Let the current tensor product consist of a large number n of H's, and suppose that this U is acting on the $73^{\rm rd}$ one. Consider any two basis-elements,

$$|0\rangle|1\rangle|1\rangle|0\rangle\cdots|1\rangle\cdots|0\rangle|1\rangle,$$

$$|0\rangle|1\rangle|1\rangle|0\rangle\cdots|0\rangle\cdots|0\rangle|1\rangle.$$

differing only in their entry for the $73^{\rm rd}$ copy of H. Now, the action of U will mix these two elements. Thus, to compute how this U acts, we will have to carry out a small arithmetic computation involving the component-values stored in these two locations. But the same is true for all m^n component-values stored. So, the order of m^n arithmetic operations must be carried out. And, apparently, there is available no shortcut, by which multiple entries can be calculated or stored all in one shot. The next application of a U may involve the $194^{\rm th}$ H in the tensor product, and to compute its action will again involve the entries in *all* the m^n locations, grouping those entries in a different way from that of the previous application of U.

So, under the assumption of rational components in a certain basis, the difficulty required to compute the effect of application of one U or P in our list is a small multiple of m^n. So, the total difficulty to compute the rational number $\gamma(\mathbf{S})$ is a sum of terms, one for each operation in the history $\mathbf{S}$ and each of the form m^n, where "n" is the then-number of H's in the Hilbert space. Eqn. (13) is a simpler, and somewhat weaker, expression of this bound. We conclude: In the case in which $|\psi_o\rangle$ and the U's and P's have rational coefficients in some H-basis, that arrangement is simple as defined above. In fact, a few other cases are also allowed by the definition, e.g., that in which U is of the form $U|1\rangle = \frac{1}{\sqrt{2}}\{|1\rangle + |0\rangle\}$, $U|0\rangle = \frac{1}{\sqrt{2}}\{|1\rangle - |0\rangle\}$. The

definition of "simple" as given has the advantage that it allows these other cases, and also that it makes no reference to any basis.

So, in short, a system — of $|\psi_o\rangle$, some unitary U's and some projection P's — is simple if the only thing that counts in computing $\gamma(\mathbf{S})$ is the number of operations represented by $\mathbf{S}$ and the size of the Hilbert spaces on which these operations act. There is no factor to represent "how hard" the arithmetic manipulations are. Simplicity means, in effect, that the operators require only "easy" arithmetic.

We are now in a position to appreciate the key difference between a quantum-assisted program and a regular program. The quantum-assisted program can apply one of its unitary or projection operators in a single step. This is because the operators themselves are rather simple, and each of them applies to only a few H's. The quantum-assisted computer simply assembles the appropriate two or three H's, and applies the operator — all without even knowing about any other H's that may be involved in the tensor product. But, in order for a regular program to see what is happening, it is necessary for that program to consider *all* the H's in the tensor product: It cannot simply ignore those H's to which the operator does not apply. In short, quantum mechanics is able to *do* (easily; probabilistically) what non-quantum mechanics can only *compute* (with much more difficulty; numerically). This state of affairs is reflected by the fact that the regular program ends up with a difficulty function for γ satisfying Eqn. (13), whereas the analogous inequality for a quantum-assisted program would read: $f_{\text{quant}}(\mathbf{S}) \leq N_{\text{ops}}(\mathbf{S})$. What quantum mechanics has going for it, in short, is the tensor product.

So, we have decided what combinations of H, $|\psi_o\rangle$, U's, and P's (namely the simple ones) to allow in our quantum-assisted programs; and what difficulty (namely, one each) to assign to the new commands, APPLY and OBSERVE, in that language. We must now contend with the probabilistic aspect of quantum-assisted computing.

Fix a quantum-assisted program, P_{quant}, that computes some problem, π, in the sense of Sect. 17. Thus, for every input string, S_{in}, we have a probability distribution p on the possible outcomes with this S_{in}; and these satisfy $p(*) = 0$, and $p(\pi(S_{\text{in}})) > p(S')$ for every $S' \neq \pi(S_{\text{in}})$. We wish to assign a difficulty function to this entire program.

Fix an input string, S_{in}, and run P_{quant} for that string. Then during this run, various commands will be executed, and to each of these we have assigned a difficulty. Let us keep track of the cumulative total difficulty

during the running of the program. Now should it happen, on this particular run of P_{quant}, that the program fails to halt, then the cumulative difficulty will, of course, grow without bound. But if P_{quant} does halt, then there will be some total cumulated difficulty, ν, as of that halt. We shall have some probability distribution on the possible cumulated difficulties, i.e., for each ν, we have a number $p(\nu) \geq 0$, such that $\sum_\nu p(\nu) = 1$. Denote by D the mean total difficulty: $D(S_{\text{in}}) = \sum_\nu \nu\ p(\nu)$. This is the difficulty that would be experienced "on the average" in one run of P_{quant} with input string S_{in}. Of course, it is only an average: On any given run, it is entirely possible that the actual cumulated difficulty turn out to be much greater than $D(S_{\text{in}})$ — or much less. Note that the sum defining $D(S_{\text{in}})$ need not converge: The difficulty ν could grow very much more quickly than $p(\nu)$ approaches zero. [**Exercise:** Find an example.] If this should occur, then we assign P_{quant} infinite difficulty for the input string S_{in}, and give up on further efforts to find a difficulty function for this program. Note that, by simulating, as in the previous section, the running of P_{quant} on input string S_{in}, we could compute an increasing sequence of rational numbers that converges to $D(S_{\text{in}})$ (or, in the case in which $D(S_{\text{in}}) = \infty$, that grows without bound). It seems unlikely, nevertheless, that D is always computable, in the sense that there always exists a (regular) program that, given $P_{\text{quant}}, S_{\text{in}}$, and positive integer n, returns a rational within $1/n$ of $D(S_{\text{in}})$. Indeed, even the problem of whether or not $D(S_{\text{in}})$ is finite is probably also not computable.

In any case, we have the notion of the mean difficulty, $D(S_{\text{in}})$, for one run of P_{quant} with input string S_{in}. But, unfortunately, a single run doesn't give us the answer, but only a probability distribution on possible outputs. The "real" answer, of course, is buried in there, in the form of the most likely output; and in order to figure out what that answer is, we must run P_{quant} repeatedly. What me must determine, then, by what factor to multiply the mean difficulty, $D(S_{\text{in}})$, to correct for this probabilistic character. To this end, let us run this program a total of r times, keeping a record of the various outputs that result. At the end of all these runs, we announce as the answer that output that occurred most frequently. Sometimes we will announce the correct answer, $\pi(S_{\text{in}})$, and sometimes the wrong answer. Denote by $\kappa(r)$ the probability that our announcement is wrong. The following Lemma states, roughly speaking, that, as the number r of runs increases, this probability $\kappa(r)$ goes to zero as e^{-Kr}, for a certain number K:

Lemma. Consider a collection of positive numbers, with sum one. Denote the largest by p and the next largest by p', with $p > p'$. Carry out r runs in the corresponding probability distribution, and denote by $\kappa(r)$ the probability that the most frequent single outcome is not the most probable outcome (i.e., not the p-outcome). Then the limit of $-\log \kappa(r)/r$, as $r \to \infty$, exists, and has value $K = (p - p')^2/2[(p + p') - (p - p')^2]$.

The proof uses two facts. First, for large r, any other outcome, say with probability $p'' < p'$ has negligible probability, compared with that of p', of being the most frequent outcome. And, second, the difference between the numbers of p-outcomes and p'-outcomes is, for large r, normally distributed, with mean $r(p - p')$ and variance $(r[(p + p') - (p - p')^2])^{1/2}$. For the present application, the p of the Lemma is $p(\pi(S_{\text{in}}))$, and the p' is the probability of the next-most-likely outcome. Note that the number K of the Lemma here depends on the input string S_{in} (through the dependence of the probabilities p, p' on S_{in}).

Now fix a small number $p_o > 0$, which we shall interpret as the "largest probability of error that we are willing to tolerate", i.e., as a confidence limit. Let us run our program a number $r = r_o$ of times, such that the probability of an erroneous announcement is within our tolerance, i.e., r_o is the smallest positive integer such that $\kappa(r_o) \leq p_o$. The mean total difficulty, for these r_o runs, is given by $f_{p_o}(S_{\text{in}}) = r_o D(S_{\text{in}})$. The limit of small p_o is the limit of large r_o, i.e., the limit described in the Lemma. We thus conclude from the Lemma that $f_{p_o}(S_{\text{in}})$ is approximated by $[D(S_{\text{in}})/K(S_{\text{in}})]\ (-\log p_o)$ in this limit. Note that the probability p_o appears in this formula only in a common overall factor. But a common overall factor is precisely what our equivalence relation on difficulty functions is designed to ignore. Thus, up to equivalence, p_o (in the limit of small p_o) drops out! It makes no difference, up to equivalence, how small is the probability of error that you are willing to tolerate. In the argument above, we (for simplicity) ignored the fact that there must always be executed at least one run of P_{comp}. To take this fact into account, we may simply add the difficulty function for a single run, $D(S_{\text{in}})$, to the difficulty function, $D(S_{\text{in}})/K(S_{\text{in}})$, that results from this argument. But $D(S_{\text{in}})/K(S_{\text{in}}) + D(S_{\text{in}})$ is equivalent to $D(S_{\text{in}})(p + p')/(p - p')^2$, where we have substituted for K from the Lemma. We summarize this discussion as follows:

> Let P_{quant} be a quantum-assisted program that computes some problem, π. Then we shall assign to this program the difficulty function given by $f_{\text{quant}}(S_{\text{in}}) = D(S_{\text{in}})(p + p')/(p - p')^2$, where $D(S_{\text{in}})$ is the mean difficulty for running P_{quant} on input string S_{in}, p is the probability that that run results in output $\pi(S_{\text{in}})$, and $p' < p$ is the probability of the next most probable output.

The factor, $(p + p')/(p - p')^2$, by which $D(S_{\text{in}})$ is multiplied reflects the increase in difficulty due to the fact that P_{quant} computes our problem only probabilistically. This factor is always at least one, and for $p = 1$ (and so $p' = 0$) this factor is exactly one, as expected. When p and p' are very close, the factor is large, reflecting the fact that there must be carried out many runs of P_{quant}, on the given input string S_{in}, in order to have a decent chance of announcing the correct value of $\pi(S_{\text{in}})$. In most cases of interest, the probability for the correct outcome dominates the other probabilities, in the following sense: There exists a positive number, independent of S_{in}, such that $p(\pi(S_{\text{in}}))$ exceeds the next-highest probability by at least that number. [This condition is always satisfied, e.g., if $p(\pi(S_{\text{in}})) \geq 3/4$. Indeed, one would perhaps not even regard P_{quant} as "really computing" the problem if this condition were not satisfied.] In any case, whenever this condition is satisfied, then the factor $(p + p')/(p - p')^2$ is bounded above, and so in this case the difficulty function, up to equivalence, is given simply by $D(S_{\text{in}})$. That is: Under this rather weak condition, we may assign to a program P_{quant} that solves a problem the difficulty function whose value on any input string is the mean difficulty of running that program on that input string.

> **Exercise.** Let P_{quant} and P'_{quant}, with respective difficulty functions f_{quant} and f'_{quant}, both compute the same problem. Is there a way to alternate between these two programs, constructing a program P''_{quant} that also computes this problem, with difficulty function given by $f''_{\text{quant}}(S_{\text{in}}) = \min(f_{\text{quant}}, f'_{\text{quant}})$?

As an example of these ideas, consider again the Grover construction, as reflected by the three distinct quantum-assisted programs introduced in Sect. 17. We now determine the difficulty function for each of these programs. For input n, a positive integer, denote by $h(n)$ the difficulty encountered by a quantum-assisted subroutine in applying the entire operator WV to the

tensor product, $H \otimes \cdots \otimes H$, of n H's, so $h(n) \geq n$. Then, as part of the lore of this construction, this same $h(n)$ will be the largest difficulty encountered by a regular program making a check to see whether a single k is the needle in the n-haystack. Thus, a regular program can compute this problem with difficulty function $f_{\text{reg}}(n) = Nh(n)$, where $N = 2^n$ is the total number of needles in the haystack.

In the first quantum-assisted program, there is made a single iteration of WV, followed by a series of n OBSERVations. There results a candidate k for the needle, which is then immediately reported using OUTPUT. The probability (p) that this k is the actual needle is about $9/N$; while the remaining k-values share the rest of the probabilities (so each p' is about $1/N$). The mean difficulty in this case is $D(n) = h(n) + n$ (since there is performed a single iteration followed by n observations). Substituting into our formula, we obtain a difficulty function (for large N) $f_{\text{quant}}(n) = (10/64)(h(n)+n)N$, which is equivalent to the difficulty function, above, of the regular program. It should come as no surprise that *this* strategy for a quantum-assisted computation brings no advantage.

In the second program, there is made a total of $\sqrt{N}$ (give or take a couple) iterations of WV, again followed by a series of n OBSERVations, and the reporting of a needle-candidate. Here, the mean difficulty is $D(n) = \sqrt{N}h(n) + n$. The probability that the reported k is the actual needle is now about $p = 1 - \frac{1}{N}$, while the remaining k-values share the rest of the probabilities (so p' is approximately $\frac{1}{N^2}$). Substituting into our formula, we obtain difficulty function $f_{\text{quant}}(n) = (\sqrt{N}h(n) + n)(1 - \frac{1}{N} + \frac{1}{N^2})/(1 - \frac{1}{N} - \frac{1}{N^2})^2$. This function, for large N, is equivalent to $\sqrt{N}h(n)$. Note that the difficulty function for this program is $\ll$ than the difficulty function for the regular program, reflecting a potential advantage for the quantum-assist (which would, perhaps, be a real advantage, if only we had a good candidate for what problem is being computed here).

In the third program, we begin, just as above, with a total of $\sqrt{N}$ iterations of WV, followed by a series of n OBSERVation. But in this case we check, using the regular program, whether or not the k that results is indeed the needle. If it is, report that k (thus incurring total difficulty $\sqrt{N}h(n) + n + h(n)$); but if it is not, go back to the beginning, carrying out the iterations and the OBSERVations again. Repeat until you find the needle. In this case $p = 1, p' = 0$ (since we will either find the needle, or

(with probability zero) continue trying forever). But now the mean difficulty (which really is a mean in this case, for now there is a nontrivial probability distribution on difficulties) is more complicated. The probability that we carry out just one group of $\sqrt{N}$ iterations of WV is $1 - \frac{1}{N}$ (approximately); that we carry out two is $\frac{1}{N}(1 - \frac{1}{N})$; etc. So, the mean difficulty is given by

$$D(n) = (\sqrt{N}h(n)+n+h(n))[1(1-\frac{1}{N})+2\frac{1}{N}(1-\frac{1}{N})+3\frac{1}{N^2}(1-\frac{1}{N})+\cdots]. \quad (14)$$

The sum on the right is $(\frac{N}{N-1})$. Substituting these into our formula, we obtain, for large N and up to equivalence, the difficulty function of this program: $f_{\text{quant}}(n) = \sqrt{N}h(n)$. This is identical to the difficulty function of the previous program.

These are precisely the results that we expect. The first program is not really exploiting the potential advantages of quantum mechanics, and its difficulty function shows it. The last two are essentially the same program. The only difference is that the first program has a fixed difficulty per run (as opposed to a probability distribution in difficulties), but leaves some unfinished business in form of the output-probabilities; while the second yields certainty for the correct output, at the cost of possibly requiring several repetitions. Our definition of the difficulty function of a quantum-assisted program is so constructed to ignore such window-dressing.

20 Limitations on Quantum-Assisted Computing

This completes our formulation of quantum-assisted computing. This formulation begins by fixing a character set; together with a finite-dimensional Hilbert space H, a state in that Hilbert space, and finite lists of unitary and projection operators, each acting on some finite tensor product of H's. On these objects we impose the condition of simplicity. We then introduce a quantum-assisted programming language, consisting of some seven commands. We introduce the notion of a program's computing a problem; as well as the difficulty function associated with such a program. These are the building blocks of quantum-assisted computing. In this section and the next, we compare quantum-assisted programs and regular programs with respect

to their difficulty functions. Here, we obtain a result to the effect that the maximum reduction in difficulty that can be achieved by quantum-assist is logarithmic.

Fix a quantum-assisted program, P_{quant}, that computes some problem π, and denote its difficulty function by f_{quant}. We construct a regular program, P_{reg}, that simulates P_{quant}, in the following manner. Fix the input string, S_{in}. Then P_{reg} simulates the running of P_{quant}, on this input string, in the same manner as in Sect. 18. That is, at any one moment P_{reg} is following a number of "branches" of P_{quant} (each spawned by the simulated execution of an OBSERVE command); and for each of these branches P_{reg} keeps track of three pieces of information: i) what is the next command, in the list P_{quant}, to be executed; ii) what is stored, by P_{quant}, in all nonempty storage locations; and iii) what is the history string $\mathbf{S}$, representing interactions P_{quant} has initiated with the quantum system. We now modify that earlier simulation, in two ways.

First, the earlier simulation (implicitly) proceeded along each branch at the same command-rate. That is, one additional command was executed in every branch; then one more command in every branch, etc. Now, however, we proceed along each branch at the same difficulty-rate. That is, we carry out one unit of P_{quant}-difficulty in each branch; then one more unit in each branch, etc. Thus, branches that involve a great deal of difficulty per command are simulated more slowly than those that involve less.

For the second modification, recall that in the earlier simulation P_{reg} maintained in its memory a special section, which was added to each time a branch under simulation reached a "halt", i.e., a P_{quant}-OUTPUT command. When this occurred, the program P_{reg} stored in this section the current history string $\mathbf{S}$, as of that halt, as well as the string S that would have then been returned by P_{quant}. A branch, once reported in this way, was then abandoned by P_{reg}. The present simulation is a little different. The special section now contains a certain list of strings and, for each such string S, a corresponding rational number. When a branch, while under simulation, reaches a halt, P_{reg} immediately computes the rational number $\gamma(\mathbf{S})$ (where $\mathbf{S}$ is the current history string), adds this number to the number already stored for string S (where S is the string that P_{quant} would have returned), and then again abandons that branch. Thus, the various strings listed in this special section are, as before, the possible outputs from P_{quant} up to this point. But now the (rational) number stored for each string gives the total probability

that P_{quant} would, by this point, have returned that string. In addition, P_{reg} contains a subroutine, which operates as follows: It goes through the list of strings and (rational) probabilities in the special section, and determines whether any output string in that list can be declared a clear winner (i.e., has a total that is greater than that which could be achieved by any other string, even if that string were allocated all so-far unallocated probability). If the subroutine finds a clear winner, then P_{reg} itself halts, returning the winning string. This subroutine is run each time P_{reg} finds itself making an addition to the special section.

So, given the program P_{quant}, we may write this program P_{reg}, which, for every input string, simulates the behavior of P_{quant}, as described above. Clearly, this P_{reg} always halts, and computes the same problem as P_{quant} does. Denote by f_{reg} the difficulty function of P_{reg}. The plan is to use (13) to find an inequality that bounds f_{reg} in terms of f_{quant}.

Fix the input string S_{in}, and denote by p the probability that P_{quant} will return $\pi(S_{\text{in}})$, and by p' the probability of the next-most-likely output, so $p > p'$. Then P_{reg} will be able to declare a clear winner, and so will halt, at least by the time it has accounted for an amount $1 - \frac{p-p'}{2}$ of probability. [To see this, denote by x the amount of probability that P_{reg} has accounted for up to some point. In the worst-case scenario, an amount $1-p$ (the maximum possible) would have been used on the non-p outcomes (including amount p' on the p'-outcome), leaving just $x - 1 + p$ for the p-outcome. So, in order that there be declared a clear winner at this point, p's amount $(x - 1 + p)$ must exceed p''s amount (p') plus the so-far unallocated probability $(1 - x)$.] Denote by $\mathcal{N}$ the total amount of P_{quant}-difficulty that P_{reg} has simulated (in each branch) at the point at which P_{reg} halts. Then we have

$$f_{\text{quant}}(S_{\text{in}}) = D(S_{\text{in}})\frac{p+p'}{(p-p')^2} \geq \{\mathcal{N}\,\frac{p-p'}{2}\}\,\frac{p+p'}{(p-p')^2} \;\geq\; \frac{1}{2}\mathcal{N}. \qquad (15)$$

The first step in (15) is the definition of f_{quant}. The second step uses the fact that P_{reg} has already gone through amount $\mathcal{N}$ of P_{quant}-difficulty, and yet there still remains probability at least $\frac{p-p'}{2}$ that P_{quant} has not halted. This fact alone contributes to the mean total difficulty of P_{quant} an amount equal to the product of these two numbers.

We must now relate f_{reg} to this $\mathcal{N}$. To this end, denote by M_{op} the maximum number of operators that can be applied, in our language, per unit

difficulty; and by $M_{\rm H}$ the maximum number of additional H's that can be introduced into the tensor product per unit difficulty. For example, if each of APPLY and OBSERVE is assigned difficulty one, and if no unitary or projection operator in these lists requires a tensor product of more than three H's, then we would have $M_{\rm op} = 1, M_{\rm H} = 3$. Note that $M_{\rm op}$ and $M_{\rm H}$ depend *only* on the our quantum-assisted language and its difficulty-assignments, and *not* on the particular program under consideration. Returning now to our simulation, since $P_{\rm reg}$ has traversed total $P_{\rm quant}$-difficulty $\mathcal{N}$ in each branch, the total number of operators that have been applied in each branch is bounded by $M_{\rm op}\mathcal{N}$; while the total number of H's that can occur in the tensor product in each branch is bounded by $M_{\rm H}\mathcal{N}$. We have

$$f_{\rm reg} \leq \{2^{M_{\rm op}\ \mathcal{N}}\}\ \{\mathcal{N} + (M_{\rm op}\mathcal{N})^2\ m^{M_{\rm H}\mathcal{N}}\}. \tag{16}$$

The first factor on the right is an upper bound on the total number of branches, where we are using the fact that each OBSERVE command spawns the splitting of one branch into two. The second factor on the right in (16) is an upper bound on the total $P_{\rm reg}$-difficulty of each branch. The first term therein covers the case in which the $P_{\rm quant}$-command simulated is INPUT, OUTPUT, APPEND, DELETE, IF, or APPLY. The second term covers the simulation of an OBSERVE command: The bound in this case is the product of our bound ($M_{\rm op}\mathcal{N}$) on the number of OBSERVE commands in a branch and the difficulty (from (13)) required to compute, for each OBSERVE command, the rational number to include in the special section. Combining (15) and (16), we obtain

$$f_{\rm reg} \leq a^{f_{\rm quant}}, \tag{17}$$

where we have chosen any $a > 2^{2M_{\rm op}} m^{2M_{\rm H}}$. [The little extra in this a takes care of the sums, extraneous factors of $\mathcal{N}$, etc.]

We conclude: Given any quantum-assisted program, there exists a regular program that computes exactly the same problem, and has difficulty function satisfying (17). The benefit in efficiency from the quantum-assist cannot be more than exponential. There is a more elegant, if slightly less informative, way of putting this. First note that if f is any difficulty function that is bounded away from one, then $\log f$ is also a difficulty function; and that if $f \sim f'$, then $\log f \sim \log f'$ also[15]. In light of this observation, we may

[15]There is actually a check to be done here. For example, the same assertion, with "exp" replacing "log", is false.

take the log of (17), to obtain a formula free of constants: $\log f_{\rm reg} \leq f_{\rm quant}$. Of course, these general inequalities are rather coarse. If, in a particular example, a finer inequality is wanted, it usually can be obtained by applying (13) directly.

As an example of these ideas, let us return to the Grover construction. Here $m = 2$. Let us assign to each APPLY and OBSERVE command difficulty one; and suppose that none of these operators require a tensor product of more than two H's. Then $M_{\rm op} = 1$ and $M_{\rm H} = 2$. Choose, for $a > 2^{2M_{\rm op}} m^{2M_{\rm H}}$, the value $a = 65$.

Our quantum-assisted program computes this problem with difficulty function $f_{\rm quant}(n) = \sqrt{N}h(n)$, where $N = 2^n$. The inequality (17) now implies the existence of a regular program to compute this problem, with difficulty function $f_{\rm reg}(n) \leq (65)^{\sqrt{N}h(n)}$. We can find a much better bound than this. This particular program requires, for given n, just n H's in the tensor product, and it applies to this Hilbert space just $h(n)$ operators. Thus, to simulate a single OBSERVE requires of $P_{\rm reg}$ difficulty $h(n)2^n$. There is a total of n such OBSERVE commands to be executed, and so we obtain $f_{\rm reg}(n) \leq nNh(n)$. Recall, by contrast, that the actual regular program for this construction has an even smaller difficulty function, namely $f_{\rm reg}(n) = Nh(n)$. The extra factor of n in the former reflects the fact that our simulation recomputes $\gamma(\mathbf{S})$, from scratch, for each OBSERVation, whereas it is more efficient to carry out these n computations together.

21 Quantum-Assisted Efficiency

This subject cries out for an example — for one clear-cut instance in which the promise of greater efficiency through quantum-assistance is fulfilled.

Challenge. Find an example of a problem π together with a quantum-assisted program, $P_{\rm quant}$ (with difficulty function $f_{\rm quant}$) having the following property: There exists no regular program, $P_{\rm reg}$, that computes this same problem and whose difficulty function $f_{\rm reg}$ satisfies $f_{\rm reg} \leq f_{\rm quant}$.

To meet this challenge requires, in other words, that there be produced a problem and a quantum-assisted computation thereof; together with a *proof* that there exists no regular program does the computation at least

as efficiently[16]. As far as I am aware, this challenge remains open. The basic impediment seems to be, not any defect in the idea of quantum-assisted computing, but rather the fact that we simply do not have good lower bounds on difficulty functions for regular programs.

The obvious way to meet this challenge would be to construct π by a diagonal argument. That is, we would introduce the list of all quantum-assisted programs that compute problems, the list of all regular programs that compute problems, and the list of all input strings. In order to choose what π is on the first string, S_1, we would try a few quantum-assisted programs on this string, as well as a few regular programs, determining what final strings result, and what difficulties are encountered. Then, we would select $\pi(S_1)$ so as to to eliminate the "faster" regular programs as well as the "slower" quantum-assisted programs. Continuing in this way through the list of input strings, we would hope to design a π on which quantum-assisted programs are more efficient. But this line, apparently, has so far not met with success.

Lacking an example of a quantum-assisted program for which we can *prove* that there exists no regular program of comparable (or lower) difficulty, we might try for the next best thing: an example of a problem and a quantum-assisted program that computes it, such that no regular program of comparable difficulty is known; and such that it is very hard to see how any such program would be written. We sketch one candidate for such an example below. There may very well be much better examples.

The plan is to come up with a problem for which quantum-assisted programs are well-suited, but regular programs are not. Consider, for instance, the following: Let π accept as input a pair $(P_{\text{quant}}, S_{\text{in}})$, where P_{quant} is a string representing a quantum assisted program, and S_{in} is any string; and let π, on such input, return the string that is determined by running program P_{quant} on input string S_{in}. Quantum-assisted programs are certainly well set

[16]It is tempting to pose a stronger challenge, namely, that which results from replacing the last part of the sentence with ". . . every regular program P_{reg} that computes this same problem has difficulty function satisfying $f_{\text{quant}} \ll f_{\text{reg}}$." But this challenge is too strong: It is unlikely that it could ever be met. The reason is that there can usually be found a regular program that has greater difficulty than P_{quant} for most input strings, but much less difficulty for just a few. The corresponding f_{reg} will not satisfy $f_{\text{reg}} \ll f_{\text{quant}}$. [We saw an example of this in Sect. 12, in which we introduced the method of determining primeness of a given integer by looking for factors in the usual manner, but first checking whether or not the given integer is a perfect square.]

up for this π! But, unfortunately, this π is not even a problem, for P_{quant}, applied to S_{in}, need not determine any string at all: The program may fail to halt altogether; or, even if it does halt, it may do so such that no one output has a probability strictly greater than that of every other possible output string[17].

A possible line to avoid the difficulty of the previous paragraph would be to focus on the essence of quantum-assisted computing: the function γ. For example, fix the quantum-assisted language, and let the problem be the following: For $\mathbf{S}$ any history string, $\pi(\mathbf{S}) = \gamma(\mathbf{S})$. This is indeed a problem; and, indeed, any regular program apparently would require a relatively large difficulty function to compute it. But this example does not work either, for no quantum-assisted program can (at least, not in any obvious way) do any better! Quantum-assisted programs are very good at taking actions in response to probability $\gamma(\mathbf{S})$, but are not so adept at actually determining the integers in the numerator and denominator of this fraction.

With these comments as motivation, we now give our example. Let H be 2-dimensional, with basis $|\psi_o\rangle, |\psi_o\rangle^\perp$, where $|\psi_o\rangle$ is our initial (unit) state. Let there be two unitary operators in our list. Operator U_{rot} acts on a single H by $U_{\text{rot}}|\psi_o\rangle = \frac{1}{\sqrt{2}}\{|\psi_o\rangle + |\psi_o\rangle^\perp\}$, $U_{\text{rot}}|\psi_o\rangle^\perp = \frac{1}{\sqrt{2}}\{|\psi_o\rangle^\perp - |\psi_o\rangle\}$; and operator U_{Tof} acts on $H \otimes H \otimes H$, by $U_{\text{Tof}}|\psi_o\rangle|\psi_o\rangle|\psi_o\rangle = |\psi_o\rangle|\psi_o\rangle|\psi_o\rangle^\perp$ and $U_{\text{Tof}}|\psi_o\rangle|\psi_o\rangle|\psi_o\rangle^\perp = |\psi_o\rangle|\psi_o\rangle|\psi_o\rangle$, with U_{Tof} the identity on the remaining basis states. Let there be just one projection operator on our list, namely $P = |\psi_o\rangle\langle\psi_o|$, the projection, on a single H, onto state $|\psi_o\rangle$.

Now let $\mathbf{S}$ be any history string, composed of these three operations, having the following property: The projection operation P appears once and only once in the string, as the last operation to be applied, and this P is paired with outcome "1". Given any such history string $\mathbf{S}$, set $\Pi(\mathbf{S}) = 1$ if $\gamma(\mathbf{S}) \geq 1/2$; and $\Pi(\mathbf{S}) = 0$ if $\gamma(\mathbf{S}) < 1/2$. This is indeed a problem. In essence, Π asks: "Is it at least a 50-50 chance that, if you apply the unitary operators in $\mathbf{S}$ and then observe via P, you will obtain result '1'?"

We next construct a quantum-assisted program, P_{quant}, that computes this problem. Let P_{quant} first parse the string $\mathbf{S}$, to extract the individual

[17]It would not help to modify this example to read: $\pi(P_{\text{quant}}, S_{\text{in}})$ is the empty string in case P_{quant}, applied to S_{in} fails to compute any string; and otherwise is whatever string it does compute. Now we do indeed have a problem π but, unfortunately, it is not a computable one. Indeed, there exists no program that will even decide whether or not a given regular program and input string results in a halt.

operations that it contains; and then apply those operations (building, in the process, the appropriate tensor product of H's). The final operation to be applied will be P (on one of the H's in the tensor product); and let P_{quant} store the result of this observation in some location. If P_{quant} now simply reported that result, it would "practically" compute Π, but for one little thing: It is possible that $\gamma(\mathbf{S})$ might turn out to be exactly $1/2$, and in this case our P_{quant} computes nothing at all (since in this case no single output will have probability strictly greater than all others). We therefore modify our program P_{quant}, to take this into account, in the following way. First note that the number $\gamma(\mathrm{S})$ can be written as a fraction whose denominator is given by $L(\mathbf{S}) = 2^{N_{\text{rot}}(\mathbf{S})}$, where $N_{\text{rot}}(\mathbf{S})$ is the number of U_{rot}-operations that appear in $\mathbf{S}$. [This follows, since each application of U_{rot} introduces a factor of $\frac{1}{\sqrt{2}}$ in the components of the state, while each U_{Tof} has, with respect to this basis, integral coefficients.] Therefore, it is only necessary, in order to handle the case in which $\gamma(\mathbf{S}) = 1/2$, to modify P_{quant} so that it generates a small extra probability, between 0 and $\frac{1}{L}$, of returning "1". To accomplish this, we have our program P_{quant} carry out the following subroutine: Construct a tensor product of $N_{\text{rot}}(\mathbf{S}) + 1$ H's, with state $|\psi_o\rangle \cdots |\psi_o\rangle$ ($N_{\text{rot}}(\mathbf{S}) + 1$ times); then apply $U_{\text{rot}} \otimes \cdots \otimes U_{\text{rot}}$ ($N_{\text{rot}}(\mathbf{S}) + 1$ times) to this state; and finally observe $P \otimes \cdots \otimes P$ ($N_{\text{rot}}(\mathbf{S}) + 1$ times). If this observation returns $(1, 1, \cdots, 1)$ (which will occur with probability $\frac{1}{2L}$) then P_{quant} halts, returning 1; otherwise (probability $(1 - \frac{1}{2L})$), P_{quant} returns the result of its original implementation of the input string $\mathbf{S}$. This quantum-assisted program does indeed compute our problem Π. It always returns either "1" (with probability $\gamma(\mathbf{S})(1 - \frac{1}{2L}) + \frac{1}{2L}$) or "0".

Let us now determine the difficulty function of this program. The original run, implementing $\mathbf{S}$, contributes difficulty $N_{\text{op}}(\mathbf{S})$; while the second run, implementing the additional probability $\frac{1}{2L}$, contributes difficulty $N_{\text{rot}}(\mathbf{S}) \leq N_{\text{op}}(\mathbf{S})$. Thus, the mean difficulty of each run is simply $N_{\text{op}}(\mathbf{S})$. Using the probabilities p and $p' = 1 - p$ of the previous paragraph and the formula from Sect. 19, we obtain the difficulty function of P_{quant}:

$$f_{\text{quant}}(\mathbf{S}) = N_{\text{op}}(\mathbf{S}) \; [\gamma(\mathbf{S})(2 - \frac{1}{L(\mathbf{S})}) + \frac{1}{L(\mathbf{S})} - 1]^{-2}. \tag{18}$$

Note that the second factor on the right in (18) is bounded provided $\gamma(\mathbf{S})$ is bounded away from $1/2$. [For instance, for $\gamma(\mathbf{S}) \leq 1/4$ or $\gamma(\mathbf{S}) \geq 3/4$, say, this factor is always ≤ 16.] But when $\gamma(\mathbf{S})$ gets close to $1/2$, this factor can

become very large. It reaches its maximum, $4L(\mathbf{S})^2$, when $\gamma(\mathbf{S})$ is exactly $1/2$. This reflects the fact that, when $\gamma(\mathbf{S})$ is close to $1/2$, many repetitions of P_{quant} will be necessary in order finally to determine whether the more probable output is "1" or "0".

So, we have now introduced a problem Π, together with a quantum-assisted program, P_{quant} that computes it; and we have determined the difficulty function, f_{quant}, of that program. Does there exist a regular program that computes this same problem, with difficulty function $\leq f_{\text{quant}}$?

We first notice that there does indeed exist a regular program P_{reg} that computes the problem Π, simply by computing $\gamma(\mathbf{S})$. The difficulty function, f_{reg}, of this program is rather complicated: Its value on some given input string $\mathbf{S}$ depends on the details of at what point in that string the various H's are inserted into the tensor product. But Eqn. (13) gives an upper bound for f_{reg} which is the order of the correct answer. Note that this regular program is actually *more* efficient than P_{quant} when $\gamma(\mathbf{S})$ is close to $1/2$, but is considerably less efficient otherwise[18]. This reflects the fact that Monte-Carlo is a poor way to estimate a probability close to $1/2$. But there exist of course many strings $\mathbf{S}$ for which $\gamma(\mathbf{S})$ is, say, greater than $3/4$ or less than $1/4$; and for these $f_{\text{reg}}(\mathbf{S})$ will be much larger than $f_{\text{quant}}(\mathbf{S})$. Thus, this particular regular program does not satisfy $f_{\text{reg}} \leq f_{\text{quant}}$.

The question, then, is whether we can find *any* regular program at all satisfying $f_{\text{reg}} \leq f_{\text{quant}}$. We have neither an example of such a program P_{reg}, nor a proof that none exists. Below, we a plausibility argument that there exists no such regular program.

We begin by introducing a new problem. Fix an integer $n \geq 3$, and denote by Z_n the collection of all strings, composed exclusively of "0" and "1", having length exactly n. So, for instance, one element of Z_5 is "10110". This set Z_n thus has precisely $N = 2^n$ elements. Denote by $C_n \subset Z_n$ the collection of all strings in Z_n whose first character is "1", so C_n consists of exactly half the elements of Z_n. Now let $\mathcal{A}$ be any permutation on the set Z_n, i.e., let $Z_n \stackrel{\mathcal{A}}{\to} Z_n$ be one-to-one and onto. Then this permutation $\mathcal{A}$ sends C_n to some subset, $\mathcal{A}[C_n]$, of Z_n; and this subset again consists of exactly half the elements of Z_n. Denote by $\Gamma(\mathcal{A})$ the fraction of all elements of C_n that arose from applying this $\mathcal{A}$ to C_n. That is, $\Gamma(\mathcal{A})$ is the number of elements

[18] We could, if we wished, modify P_{quant} so that it is *always* more efficient than this P_{reg}, by having P_{quant} alternate its own steps with those of this P_{reg}.

of $C_n \cup \mathcal{A}[C_n]$ divided by the number (namely, $\frac{N}{2}$) of elements of C_n itself. This $\Gamma(\mathcal{A})$ must be between 0 and 1. For example, for $\mathcal{A}$ the permutation that reverses the third and fifth entries of the string, $\Gamma(\mathcal{A}) = 1$ (since this $\mathcal{A}$ leaves C_n invariant; while for $\mathcal{A}$ the permutation that reverses the first and fifth entries of the string, $\Gamma(\mathcal{A}) = \frac{1}{2}$. [Can you think of an $\mathcal{A}$ for which $\Gamma(\mathcal{A}) = 0$?] For n large, and $\mathcal{A}$ "random", $\Gamma(\mathcal{A})$ will be fairly close to $\frac{1}{2}$, although there will always exist $\mathcal{A}$ for which $\Gamma(\mathcal{A})$ is close to zero or close to one.

Next, we consider a special class of permutations on the set Z_n. Fix three distinct positive integers, (a, b, c), none of which exceed n. Let $A_{a,b,c}$ have the following action on strings in Z_n: If the a^{th} and b^{th} entries of the string are both "1", then $A_{a,b,c}$ reverses the c^{th} character of the string (i.e., sends $1 \to 0$ and $0 \to 1$); and otherwise $A_{a,b,c}$ does nothing. Thus, in Z_5, $A_{4,2,1}(10110) = 10110$; while $A_{4,2,1}(11010) = 01010$. Each of these $A_{a,b,c}$ is, of course, a permutation on the set Z_n. Note that each permutation $A_{a,b,c}$ leaves unchanged exactly $\frac{3}{4}$ of the elements of Z_n, while permuting the other $\frac{1}{4}$ among themselves. By applying a number of these A-permutations in succession, one can generate rather complicated permutations on this set Z_n (but not every permutation. For example, $A_{a,b,c}(00\cdots0) = 00\cdots0$ for every (a, b, c).)

Here, finally, is the problem. Let string S_{in} represent some integer $n \geq 3$, together with some finite list of A-permutations on Z_n. Then set $\pi(S_{\text{in}}) = 1$ of $\Gamma(\mathcal{A}) \geq \frac{1}{2}$; and $\pi(S_{\text{in}}) = 0$ if $\Gamma(\mathcal{A}) < \frac{1}{2}$, where $\mathcal{A}$ denotes the permutation that results from applying the A-permutations in succession. Thus, if S_{in} represents the integer 5 together with the single permutation $A_{3,2,1}$, then $\pi(S_{\text{in}}) = 0$ (since $\Gamma(A_{3,2,1}) = \frac{1}{4}$), while if S_{in} represents the list $A_{2,3,4}, A_{1,2,3}$, then $\pi(S_{\text{in}}) = 1$.

Here is a regular program that computes this problem (by brute force). It introduces $N = 2^n$ storage locations, and simply follows explicitly what each $A_{a,b,c}$ does to each element of Z_n. Then, after the A's have all been applied, it computes $\Gamma(\mathcal{A})$, and then $\pi(S_{\text{in}})$. The difficulty function of this program is given by

$$f_{\text{reg}}(S_{\text{in}}) = N(S_{\text{in}})N_A(S_{\text{in}}), \tag{19}$$

where $N(S_{\text{in}}) = 2^n$ is the number of elements of Z_n, and $N_A(S_{\text{in}})$ is the number of A-permutations represented in the string S_{in}.

Does there exist a regular program more efficient than this one? Well,

there certainly are special combinations of A-permutations that are easy to handle directly, without recourse to all this storage and calculation. For example, if $S_{\rm in}$ consists simply of a single $A_{a,b,c}$, repeated any even number of times, then $\Gamma(\mathcal{A}) = 1$, and so $\pi(S_{\rm in}) = 1$. But in general the different A-permutations feed on each other — what a given A will do depends on what has already been done before — and it is very difficult to think of any substantial shortcuts. Even in cases for which Γ is $\geq \frac{3}{4}$ or $\leq \frac{1}{4}$ (i.e., in which there is a substantial overlap or underlap between C_n and $\mathcal{A}[C_n]$) is hard to see how to determine π without resorting to this brute-force method.

This problem π is, of course, a mere subproblem of the problem Π introduced earlier. Let the history string $\mathbf{S}$ first use operations involving $U_{\rm rot}$ to construct the tensor product, $H \otimes H \otimes \cdots \otimes H$, of n H's, placing the quantum system in the state with the first H-state $|\psi_o\rangle$, the remaining H-states $\frac{1}{\sqrt{2}}\{|\psi_o\rangle + |\psi_o\rangle^{\perp}\}$. Then apply $U_{\rm Tof}$-operations, corresponding to the $A_{a,b,c}$ that appear in $S_{\rm in}$. And, finally, make an observation associated with P, applied to the first H in the tensor product. What Π returns for this history string is what π returns for this $S_{\rm in}$. Thus, the quantum-assisted program $P_{\rm quant}$ we gave earlier to compute Π also computes this subproblem π. From (18), the difficulty function of this program is

$$f_{\rm quant}(S_{\rm in}) = N(S_{\rm in})\ [\Gamma(N_A(S_{\rm in}))(2 - \frac{2}{N(S_{\rm in})}) + \frac{2}{N(S_{\rm in})} - 1]^{-2}. \qquad (20)$$

This $f_{\rm quant}(S_{\rm in})$ is far less than the $f_{\rm reg}(S_{\rm in})$ given in (19) when, e.g., $\Gamma(N_A(S_{\rm in}))$ is less than $\frac{1}{4}$ or greater than $\frac{3}{4}$. They differ in this case by a factor of $N(S_{\rm i}) = 2^n$. Thus, our problem Π is a candidate for meeting the Challenge issued at the beginning of this section. In order to defeat this candidate, it is necessary to find a much more efficient regular program for computing this problem. It seems unlikely that there exists any such regular program — but it also seems difficult to prove it.

22 Non-Quantum Assistance

Quantum mechanics clearly has a potential advantage in the battle for efficient computing — the tensor product. This construction allows one to manipulate just n physical systems, thereby carrying out operations on m^n states. Can other physical theories partake of this advantage? That is, can

we, in any other physical theory, carry out a "tensor-product-like" construction?

Consider electromagnetism. Suppose that we were capable of manufacturing small boxes, in which there could be stimulated a total of three possible electromagnetic modes. Thus, the electromagnetic states within each box form a 3-dimensional (real) vector space, V. Now take two such boxes, place them side by side, and regard these two as a single system. What is the space of states of this combination? Well, each of the two boxes carries a field, in some state in V, and so the state of the total system is described by simply specifying these two elements of V. That is, the vector space of states is $V \oplus V$, the direct sum of V and V (with dimension $6 = 3 + 3$). Had this instead been $V \otimes V$, the tensor product (with dimension $9 = 3 \times 3$), then we would have the beginnings of electromagnetic-assisted computing: We would then proceed to introduce various interactions between the boxes, various observations on the boxes, etc.

But tensor products, it appears, do not routinely make an appearance outside of quantum mechanics. Is there some general principle of non-quantum physics that rules out the tensor product, once and for all? Here is an example suggesting that there is no such principle. Consider a one-dimensional "box", of length L, in which the vector space V of allowed states is those resulting from exciting three modes of a field, given, say, by ($\sin \pi x/L$, $\sin 2\pi x/L$, $\sin 3\pi x/L$). To take the "product" of two such boxes, we consider fields in the square of side L. The corresponding modes are arbitrary linear combinations of products, $\sin a\pi x/L \sin b\pi y/L$, where $a, b = 1, 2, 3$. That is, the vector space of such solutions is $V \otimes V$, the (9-dimensional) tensor product of V with itself. And, passing to a cubic box, we obtain a space of states given by $V \otimes V \otimes V$.

But, alas, we all too soon run out of dimensions.

References

[1] Agrawal, M, Neerja, K, Nitin, S, "Primes is in P", Annals of Mathematics 160, 781 (2004). At http://www.math.princeton.edu/~annals/issues/2004/Sept2004/Agrawal.pdf Shows that there exists a program that computes the problem of decid-

ing whether or not an integer n is prime, with difficulty function given by $\leq (\log n)^s$, for every $s > 15/2$.

[2] Blum, Manuel, "A Machine-Independent Theory of the Complexity of Recursive Functions", J. Assoc for Comp Mach 14, 322 (1967). http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=321395 Deals with properties of difficulty that rely only on some general features of the difficulty-measure. There are two very nice results here!

[3] Grover, L, "A Fast Quantum-Mechanical Algorithm for Database Search", in Proc. 28th Annual ACM Symposium on Theory of Computing, ACM, New York (1996).

[4] Hartmanis, J, Hopcroft J.E., "An Overview of the Theory of Computational Complexity", J. Assoc for Comp Mach 18, 444 (1971).

[5] Hennie, F.C., "One-Tape Off-Line Turing Machine Computations", Information and Control 8, 553 (1965). It is alleged that these techniques can be used to show that no Turing machine can compute the palindrome problem with difficulty function $\ll L(S)^2$.

[6] Kelly, John, "General Topology", Springer-Verlag (New York), 1975. An appendix contains the best treatment of axiomatic set theory I have ever seen.

[7] Lenstra, A.K., Lenstra, H.W, eds, in "The Development of the Number-Field Sieve", Lecture Notes in Mathematics 1554, pp 11-42, Springer-Verlag (1993). The sieve method for computing the prime problem.

[8] Mermin, David, "Quantum Computation", Lecture Notes, at http://people.ccmr.cornell.edu/~mermin/qcomp/CS483.html. A beautifully done introduction of this subject. See especially Sect IV. These notes should be a relatively quick read (since they are intended for undergraduates).

[9] Pittenger, Arthur, "An Introduction to Quantum Computing Algorithms", Progress in Computer Science and Applied Logic, Vol 19, Birkhauser. Gives some detail how to do real-world computations with c-not gates.

[10] Unruh, W.G., "Maintaining Coherence in Quantum Computers", Phys Rev A 51, 992 (1995).

[11] Yasuhara, Ann, "Recursive Function Theory and Logic", Academic Press, 1971. This is my favorite book on Turing machines, unsolvable problems, etc.