The P vs. NP problem

Madhu Sudan*

May 17, 2010

Abstract

The resounding success of computers has often led to some common misconceptions about "computer science" — namely that it is simply a technological endeavor driven by a search for better physical material and devices that can be used to build smaller, faster, computers. Without trivializing this important quest, this article would like to highlight the complementary science behind computing: namely the search for efficient algorithms (or programs or procedures or software, depending on the readers preference) and the profound implications of this science, not only to the technology of computing, but almost all other disciplines including mathematics, physics and other physical sciences, as well as the social sciences including economics, psychology and indeed, even philosophy!

In this article we will attempt to introduce some of the very basic elements of the theory of computing starting with elementary examples of computational tasks that everyone is familiar with. We will then describe Turing's contributions in the 1930s that led to the development of both computers and a theory of computers. We will conclude with a description of the more modern theory of efficient computing from the 1970s leading to the question "Is P = NP?". We will explain the question, its significance and the important consequences of its resolution (depending on how the question is resolved).

1 Introduction

DISCLAIMER: *As with all other surveys I have written, this one was finished in a hurry; and as a result might suffer from a lot of rough edges. Please feel free to email me your comments if you have any; but please accept my apologies first for any errors, misappropriations and in general lack of taste and judgement. That said, lets get on with our story . . .*

The modern story of computing starts well before there was any realistic hope of building a physical computer, with the work of Alan M. Turing [12] in the 1930s. The idea of calculating devices, and indeed even some notion of programmable calculating devices had existed even before. But a key piece of insight was still missing in all of these works. Note that for most design/engineering tasks, one tends to build a device **after** the task has been fixed. Indeed one looks carefully at the task to be solved, and then carefully comes up with a potential solution that applies in this case. In particular, different tasks require different solutions. Computing, and this may be considered the heart of the science behind computing, is a radically different discipline in this regard. One can build/buy a computer today; and program it to solve any computational

*Microsoft Research New England, One Memorial Drive, Cambridge, MA 02142, USA. `madhu@microsoft.com`. Eventually it is hoped that some version of this article will appear in some form or other in some publication.

task later. (This may require some minimal augmentation in the form of large secondary storage; but that is all.) This discovery about computation is the essence of Turing's contribution to the science of computing — popular as the "universal computer". Till Turing's work came along it was unclear how powerful any *single* computing device could be: Would it always be the case that for every computer one conceives of and builds, there would be some computing tasks that they simply cannot solve? Turing showed remarkably that this was not the case. He proposed a formal (mathematical) model of what a computer ought to be, and then showed there was one computer (the "universal" one) to which one could describe every other computer so that the universal computer could simulate its actions. By doing so he laid the seed for the modern computer which adopts the same principles (though with significant differences aimed at practical efficiency). But Turing himself was not aiming to build these machines - his aim was more to capture logic and human reasoning. The work in mathematical logic at the point had been discussing what a "theorem" is, and what "proofs" are, and assertions may never find proofs (even though they may be true). The crux of all this work was modelling the notion of proofs attempting to extract this notion led Turing to lay down some axioms about human reasoning that form the basis of his model of a computer. Turing's work thus formed the basis of the theory of computing which formalizes what can, or can not be done by a computer. More modern work focusses on "how quickly" can some problem be solved on computers, and which problems seem to be inherently slow. It should be stressed that all this research is independent of the actual technology and if computers speed up to twice their current speed (as they seem to do every three years), the distinction between quickly solvable problems and those that are not would not change! Thus the "complexity" of a problem is really a fundmental notion, and in this article we attempt to describe this notion to the reader.

2 Examples of Computational Problems

We now jump from the very high-level description of computing in the previous section to some very concrete examples. We hope these will be familiar to every reader and thus allow us to explain basic language such as "what is a 'computational problem?" and "what does a solution to a problem (known as an algorithm) look like?".

Example 1: Addition. We start with a very simple problem, familiar to most school children, namely that of adding two numbers. Say we would like to add the number 10234843 to the number 2345639. In elementary school we teach children how to do this. "You start by writing the two numbers one above the other, with both numbers ending in the same column. Then starting with the rightmost column, you add the two digits in the column. If this leads to a number less than ten, then write the number below the two digits and repeat on the next column to the left, else write the last digit of the sum below the two digits, and then repeat with the next column to the left, retaining a carry of 1. Keep repeating the procedure till you are done processing every column. The number on the bottom row is the result you desire."

The description above is roughly what an algorithm is (despite several defects due to informalities that we will get to shortly). It is a finite length description, that explains how to add two *arbitrary* positive numbers, and thus explains how to deal with an infinitely large scope of possible input numbers, and which terminates on every finite length integer. Of course, one doesn't need to mention the words "algorithm" or "infinite many inputs" or "termination" to teach this to a child. But in retrospect these phrases comes in very useful in explaining the power of this method, and its surprising effectiveness: it deals with an infinite scope of possibilities with a finite set of rules!

Before moving to the next example, let us perform a simple "analysis" of the above "algorithm" for adding numbers. Suppose the numbers to be added are each at most n digits long, then how many simple steps does computing their sum take? The answer, of course, depends on how you define a simple step and depending on this you may say it takes about n steps (one for each column of the input) or $n + 1$ steps (to write down the possibly $n + 1$ digits of the output) or $5n + 1$ steps where each column of the input took at most five elementary calculations (adding the digits, adding the carry-in if necessary, seeing if the sum is less than ten or more, splitting the result into two digits, writing the lesser digit and moving the rest into the carry) and the final column of the output took one step. Despite this seeming arbitrariness of the number of steps, it turns out no matter how we try to write this expression it comes out into a "linear" form, i.e., of the form $a \cdot n + b$ where a and b are some constants. What is important is that the answer can not be found in less than n steps (not say $\sqrt{n}$ steps which would be much smaller than n) nor does it require n^2 steps. This leads to a crude, but extremely useful, approximation to the *time complexity* of this algorithm - it takes $\Theta(n)$ steps (where $\Theta(n)$ roughly suppresses the a and the b above and other such "lesser order terms" and just describes the leading term).

Addition is thus our first example of a "computational problem". The elementary school procedure for long addition is our first "algorithm". And its "time complexity" is linear, i.e., $\Theta(n)$. Linear time solutions are obvously the fastest possible since it takes that much time to read the input or write the output, and so addition turns out to be one of the most "efficiently solved" computational problems. Let us now move to a slightly more complex example.

Example 2: Multiplication. Now lets consider a somewhat more complex "computational problem", that of multiplying two numbers. So the input to the task is two numbers, say X and Y, each at most n digits long. The goal of the task is to produce or "output" the product $X \times Y$, the result of multiplying X and Y.

The naive method, used in the definition of the product of $X \times Y$, is to report $X + X + \cdots + X$ where there are Y copies of X in the list. Even in elementary school we dismiss this approach as too inefficient. Let us see why: Such a computation would involve Y additions of numbers that are between n and $2n$ digits long (Note $X \times Y$ is at most $2n$ digits long.) But Y is a number that is exponentially large in n. In particular $10^{n-1} \leq Y < 10^n$, if Y is n-digits long. So this method of multiplication requires roughly 10^n additions, each of which takes roughly n units of time leading to a complexity of $\Theta(n \cdot 10^n)$ (dropping leading constants and lesser order terms). This is woefully slow. To multiply two 10 digit numbers, this would take 100 billion steps! (A schoolchild would easily reach retirement age adding two such numbers, and this can't be good.)

Fortunately, the "long multiplication" technique taught in elementary school speeds things up rapidly. This is the method which roughly computes n numbers, the first being the product of X with the least significant digit of Y, the second being the product of X with its ten times its next significant digits and so on, and then adds these n numbers together. A quick calculation shows that this method takes $\Theta(n^2)$ steps, with the computation being dominated by the time to write the n intermediate numbers above. Two multiply two 10-digit numbers, this is roughly a 100 steps, which is immensely better than 100 billion! But notice it does take much more time than adding two 10-digit numbers.

Given that there are at least two different ways to multiply n digit numbers, and one of them is much faster, and hence desirable, than the other; it makes sense to ask - can it be that there are even faster procedures to multiply numbers? Faster than $\Theta(n^2)$? The natural, and first reaction to these questions is: Surely not! Surprisingly, this is not true! One can actually multiply n-digit numbers in time much closer to $\Theta(n)$. (We won't go into the exact complexity since it requires more notation, but the running time can be made smaller than $\Theta(n^{1.1})$ or even $\Theta(n^{1.001})$ etc. - any number strictly greater than 1 in the exponent of n will do.)

Yet no algorithm running in time $\Theta(n)$ has been found, suggesting multiplication may indeed be more "complex" than addition, in the number of steps required. This would correspond to our intuitive feel that mutliplication is more complex; but note that our feel was based on a completely misguided belief that multplication ought to take $\Theta(n^2)$ steps. To really "prove" that multiplication is more complex than addition, one would have to examine every algorithm that multplies two numbers (and there are infinitely many such algorithms!) and prove each one takes more than $\Theta(n)$ time. Such complexity lower bounds (on complexity) are rare for any computational problem; and for the specific case of multiplication, this question remains wide open.

Finally let us quickly introduce two more problems to expose the reader to more "complexity" (i.e., more complex problems).

Example 3: Factoring. Our next problem is simply the "inverse" of the problem above. I.e., now somebody gives us $X \times Y$ and we would like to roughly, report X and Y. Stated this way, the problem is not well-defined (given 1001, would you like as output 77 and 13, or 91 and 11 etc.). So let us define it more carefully. A number Z greater than one is a *prime* if it cannot be written as $X \times Y$ where X and Y are both positive and less than Z. The factoring problem is simply the following: Given as "input" Z report two numbers X and Y smaller than Z such that $X \times Y = Z$, if such a pair exists, else report "Z is prime".

A simple algorithm to do this, would be to try every number X between 2 and $Z - 1$ and see if X divides Z. If Z is an n-digit number, the complexity of such a procedure is at least 10^n, i.e., exponentially large in n. Despite several centuries of effort on this problem, the fastest solutions remain exponentially slow in n, leading to a widespread belief that no solutions running in time $\Theta(n)$ or $\Theta(n^{10})$ or "polynomial time" (i.e., $\Theta(n^c)$ for any constant c) exist. This problem is deeply interesting in that if this belief were found to be wrong and someone found an efficient solution for this problem (say running in time $\Theta(n^{10})$ our entire current system of electronic commerce and its security would crumble (as would any other infrastructure that depends on security on the internet).

Example 4: Divisors. A very closely related problem to the one above is the following: Given three n-digit numbers Z, L and U, report a number X such that X divides Z and $L \leq X \leq U$, if such an X exists.

Of course, a solution to the divisor problem would also immediately lead to a solution to the factoring problem (just use $L = 2$ and $U = Z - 1$). And the naive algorithm for factoring above would also solve divisor in exponential time. So in several senses the two problems seem the same. But the problems do turn out to be quite different (at least to the current state of knowledge). The divisor problem is essentially as complex a search problem as any (in later sections we will describe describe a class of problems called "NP-complete" problems, to explain this notion more formally). Among other features, "divisor" has the distinguishing feature that if the answer to the search question ("find X") is negative - no such X exists - we don't know how to give a succinct proof of this statement. (In contrast, in the factoring problem, if the answer is negative and no such X, Y exist, then Z is a prime and one can prove this succinctly, though describing the proof is not within the scope of this article.)

Example 5: Solutions to a polynomial system. This example is perhaps the most complex of the problems we describe, both in terms of ”how quickly it can be solved” and also to describe, so we will ask for the reader’s patience.

A large class of mathematical problems asks the question, does a given polynomial equation have a solution among the integers. A classical example is that of Fermat’s last “theorem” - which considers the equations of the form $X^n + Y^n - Z^n = 0$. For $n = 2$, $X = 3$, $Y = 4$, and $Z = 5$ is one of infinitely many solutions. Fermat asserted (claiming a “proof that didn’t fit the margin of his page”) that this equation does not have a solution with X, Y and Z being integers for any integer $n \geq 3$. It was only recently (2000) that the problem was finally resolved by Andrew Wiles. But even slight variations lead to major challenges to mathematicians. For instance Poonen [11] asserts that if we asked “Are there integers X, Y, Z satisfying $X^3 + Y^3 - Z^3 = 33$?” we still don’t know the answer.

We could imagine turning over such questions to a computer to solve! This would lead to the problem: Given a polynomial equation in several variables, does it have an integer solution? Not surprisingly, (given we don’t know if a specific polynomial in 3 variables X, Y, Z of degree 3 has a solution or not) we don’t have any finite time algorithms for such problems. Even worse, Matiyasevich [10] proved that no finite procedure can even exist.

Thus moving slowly from computational problems that we solve in elementary school, we have reached problems that require all the ingenuity of the best mathematicians to solve, and even that may fail. Thus seeing some of the scope of computational problems via examples, let us now try to understand some of the concepts more formally.

3 Computational complexity: Some formalisms

Let us start with some terminology we have already been using. A *computational problem* is given by specifying a desired *relation* between *inputs* and *outputs*. The goal for any such problem is to produce, given an input, to produce an output that satisfies this desire. (In mathematical terminology, the desire is simply a relation on the cartesian product of the space of inputs and the space of outputs.)

In the case of integer addition and multiplication, for any one input there was only one correct output - such problems fall in the special class of “functional computations”. For others, such as factoring and divisor, sometimes several answers are all correct and the task simply requires the output to be any one of the possibilities.

Specifying a solution to a computational problem, namely an algorithm is much trickier. We attempted to do so informally for integer addition, but even there our description was vague. It didn’t define “carry”. Its description of how to repeat the procedure was by analogy, rather than precision. And its description of the termination condition and the solution were all vague.

All the objections above could be easily remedied, Perhaps the most serious objection would be that we never completely specified what tasks were achievable in one step, and what means can be used to specify what step to perform next, and how much of the past can be remembered in determining the future course of actions. An *algorithm* needs to specify all of this and completely unambiguously to a computer. This was the challenge facing Turing as he wrote his monumental paper. He described a solution which even with the benefit of hindsight is unbelievably slick and subtle. Before describing his formalism of an algorithm, let us lay out some desiredarata on what an algorithm should look like.

To motivate some of these, think about the algorithm for multiplying two numbers. As mentioned earlier, we think a small (finite) set of "rules" suffice to explain the procedure, despite the fact that the set of possible inputs is (infinitely) large. For instance children need to learn multiplication tables for every digit in order to start multiplying larger numbers. (If one were to work in binary less memorization would be required, but then the length of the numbers would increase roughly by a factor of three so one compromises time to multiply for memorization). In any case after some memorization (finite amounts of it) the child seems to be doing something very "local" at each point of time. In addition to the multiplication tables, the children does have to does have to remember some other things in the intermediate stages (like which digit of Y it is currently multiplying with X, and what the results of the previous multiplications were). To do so, it does need some "scratch paper" — or "memory". But once one decides what to write down on the scratch paper/memory, the algorithms next steps are really simple and do not get more complex as the amount of stuff it has committed to memory grows larger. Keeping these aspects in mind, we lay out the following desiderata that the definition of an algorithm should possess.

1. An algorithm should be finitely described (else it wouldn't fit in this book!).
2. At the same time, it should be capable dealing with infinitely many possible inputs and produce infinitely many different outputs.
3. It should be allowed to use scratch paper (extra memory) to store intermediate results of computation.
4. Its actions at any point of time may depend on the past, the input and the scratch contents, but only in some finite way.
5. It should model every (physically realizable) computational process.

Items (1)-(4) are restrictive and limit the actions of an algorithm. Item (5) on the other hand is expansive in its desire. It would like the formal definition of the algorithm to include every possible mental image of an algorithm or a computational process. Furthermore, it is not formal and not even formalizable till we have a description of "all physically realizable computational processes". Indeed this item is a wishful thought - it would be nice to have a definition of an algorithm which is not immediately challenged by a counterexample of a natural process that fits our intuition but does not match the formal definition.

In the rest of this section we describe the technical formalism that Turing proposed for his notion of an algorithm, which manages to satisfy items (1)-(4) and seems to also satisfy (5), modulo details of the laws of physics.

3.1 Turing's formalism

This section will be somewhat technical and use basic set theoretic notation. (A reader not familiar with such notation may simply skip this section and jump to the next.)

All algorithms, according to Turing, are given by a "finite state control" an infinite tape, and a "head". Different algorithms differ only in the finite state control, and otherwise their operations go along the same fixed set of rules.

The control is described by a six tuple $(Q, \Sigma, q_0, f, \sqcup, \delta)$ where

- Q is a finite set (describing the set of states the control can be in).
- Σ is a finite set (describing the set of letters than can be written on the tape).
- $q_0, f \in Q$ (are the initial state and the halting state).
- $\sqcup \in \Sigma$ (is a special symbol).
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{+1, 0, -1\}$ (is the state transition function).

The "tape" formally is a sequence of functions $\text{tape}_t : \mathbb{Z}^+ \to \Sigma$ for every $t \geq 0$ and the head locations are given by a sequence $\text{head}_t \in \mathbb{Z}^+$.

The operation of the Turing algorithm given by the control above is the following:

Initially, at $t = 0$, the algorithm is in state q_0 and $\text{head}_0 = 1$. The tape at $t = 0$, tape_0, contains the input to the algorithm; i.e., given a desired input $x = x_1, \ldots, x_n \in (\Sigma - \{\sqcup\})^n$, it is provided to the algorithm by setting $\text{tape}_0(i) = x_i$ for $i \leq n$ and $\text{tape}_0(i) = \sqcup$ for $i > n$.

At any given time step t, the algorithm is in some state $q_t \in Q$ and its tape has some information given by the function tape_t, though the algorithm only sees the current contents at the tape head where the location of this head is given by $\text{head}_t \in \mathbb{Z}^+$.

If the state of the algorithm at time t is the halting state (i.e., $q_t = f$), then the algorithm halts, and the tape contents tape_t is the output of the algorithm.

If the state q_t is not f, then algorithm evolves to time $t+1$ where the evolution of the algorithm to time $t+1$ is given by the transition function applied to the current state and symbol on tape. Let $(q', \gamma, b) = \delta(q_t, \text{tape}_t(\text{head}_t))$. Then the state of the algorithm at time $t+1$ is given by $q_{t+1} = q'$. The location of the head $\text{head}_{t+1} = \text{head}_t + b$. The tape contents remain mostly unchanged except at the head location and evolve as follows. $\text{tape}_{t+1}(\text{head}_t) = \gamma$ and $\text{tape}_{t+1}(i) = \text{tape}_t(i)$ for every $i \neq \text{head}_t$.

3.2 Implications of the formal definition

Remarkably, with total mathematical precision (and with very little mathematical notation), the above completely describes a formalism of an algorithm. It satisfies our desires as listed in items (1)-(4) and seems thus far to satisfy the final wish that it should model every intuitive notion of an algorithm. This assertion is now known as the Church-Turing thesis.

Thesis 3.1 *Every physically realizable computation can be simulated on a Turing machine.*

(We will remark on the challenges brought up by randomness and quantum physics later.)

Of course, the definition is not a very convenient one to describe algorithms in a natural fashion. Indeed it may take a long while before a college student in mathematics, who is very familiar with programming and, of course, elementary school arithmetic, might be able come up with a "Turing algorithm" for multiplying two integers! But the main point is that it can be done, and so can any other natural process we may conjure. And among all such possibilities it remains one of the simplest to reason about mathematically.

The main advantage of the ability to model all computational processes mathematically is that we can now reason about the limits of computation. Indeed this was one of the main points of Turing's work. But first he proved a result which was fundamental to the development of the modern computer.

He proved that in his formalism there existed a single algorithm, the "universal computer", that could take as input a description $< A >$ of another algorithm A, and an input x and report the output of A on x. In other words, the universal computer could "run" every algorithm.

Today, this is a totally non-surprising result. Most people own computers that can be programmed to execute any program, and thus any algorithm. This is really a consequence of Turing's universalilty theorem. Turing's work was the first to realize that there is a duality between "data" and "algorithms"; the universal computer converts every algorithm A into some data $< A >$ that becomes input to some other algorithm (namely the universal computer). This idea inspired the contributions of John von Neumann, architect of the "stored-program computer", which suggested a somewhat more practical, but mathematically less simple, model of a computer which could deal with any program as data, and simulate the effect of any algorithm on any input.

4 Computational Complexity

A formal definition of an algorithm is of course somewhat helpful in the design of algorithms but is not essential. The definition really becomes useful when one tries to understand the limits of algorithmic power. Indeed this was one of the original motivations behind the formal definition. Using the fact that every algorithm is potentially an input to other algorithms, Turing described a computational problem that could provably not be solved by any algorithm.

The *halting problem* is the computational problem whose inputs are pairs $(< A >, x)$ where $< A >$ is the description of some algorithm A and the desired output is true is A halts in finite time on input x and false if it does not halt. Turing showed that the halting problem can not be solved by any algorithm.

This result and strengthenings led over the years to an immense collection of results showing the limits of algorithms. It became clear that the power of the universal computer - the fact that it can simulate any algorithm - was also its bane. It is too easy for computational tasks to become self-referential, and this leads to many limitation results. Often the self-reference emerges in very surprisingly simple ways. A highlight result in this direction is that of Matiyasevich [10] who showed that the problem in our Example 5 above (given a polynomial equation, does it have a zero among the integers?) was also "undecidable" (i.e., unsolvable by any algorithm)!

4.1 Complexity Measures

Subsequent explorations of computational complexity focussed on the amount of "resources" would be needed to solve a desired problem on a computer. The main resources of interest are the "time" that an algorithm would take to solve a given problem, and the "memory" or "space" that it needs to solve a given problem.

For a specific algorithm, and a specific input, these quantities are easy to define based on Turing's formal definition. (Space is the largest index s for which $\text{tape}_t(s) \neq \sqcup$ for some t, and Time is the first index t for which $q_t = f$.) However, how does one define the complexity of an algorithm on a "problem" (when one has not fixed a particular input). More crucially, given two different algorithms whose time-complexity is different on different inputs, how does one compare them?

Over the years, the theory of computing has converged to two principal "simplifications" to allow such comparisons, leading to the "worst-case" "asymptotic" complexity measures, which roughly studies the complexity as the input gets "larger". Let us explain these simplifications in the case of time-complexity. But before doing so, let us note that the every input to a computer has a natural "length" associated with it. (Formally, it is the minimum n such that $\text{tape}_0(n+1) = \sqcup$.)

Given an algorithm A, its worst-case time complexity $T_A(n)$, is the maximum over all inputs x of length n of the time A takes to solve the problem and halt on input x. Worst-case complexity already simplifies the description of time complexity. Rather than describing running time on each possible input, we only have to describe it for each possible input length.

But now consider two algorithms for multiplication of n-digit numbers: with the A having time-complexity $T_A(n) = n^3$, while B having time-complexity $T_B(n) = 3000n^2$. Which of these is better? To make comparisons between such we use the asymptotic comparisons, which say that if one of the running time bounds is better than the other by more than a fixed multiplicative factor for large enough n, then such a bound is better. (So $T_B(n)$ would be better than $T_A(n)$ by this measure since for every c, $T_B(n) < T_A(n)/c$ for every $n > 3000c$.)

Using such measures computational complexity manages to make comparisons between most natural algorithms for natural problems. And the first findings are not surprising: For every "natural" time complexity function $t(n)$, there are problems that are solvable in time $t(n)$, but not much faster. So there are problems that for which *every* algorithm takes time at least n^2 asymptotically (i.e., on large enough inputs), and which can be solved in time n^3 (for instance).

Unfortunately, the problems for which we have such precise knowledge are somewhat unnatural. For natural problems our precise knowledge is certainly very weak. For instance for integer addition (Example 1), we know no algorithm can run in time faster than n and this is also achieved by the natural algorithm. But this is about the limit for problems without a natural lower bound. For multiplication (Example 2) we don't know of an $\Theta(n)$ time algorithm but don't know that this is not possible either. Even worse we don't know of any "polynomial time" algorithm for solving the factoring problem (Example 3). The best algorithms still run in time "exponential" in some small power of n. The divisor problem appears even harder though still solvable in exponential time.

4.2 Polynomial and Exponential Time Complexities

Two very broad classes of running time complexity that turn out to take a central role in computing are "polynomial time computable problems" (also known by the single letter P) and "exponential time computable problems". The former include all problems that can be solved in time $\Theta(n^c)$ for some constant c (independent of the input size n), while the latter is a broader class and includes all functions that can be solved in time $2^{\Theta(n^c)}$ for constant c.

To understand the difference between the two cases, consider some problem X whose complexity is in P (polynomial time). Let n_0 be the largest input length for which this problem can be solved in an hour on a computer. Now suppose we double the amount of time available. How will the size of the largest solvable input grow? The feature of P is that this input size will grow by a constant *multiplicative* factor. For problems with algorithms in time $\Theta(n)$, the n_0 will double. For problems with algorithms in time $\Theta(n^3)$, the n_0 will only grow by a factor of $\sqrt[3]{2}$. If this seems bad, now consider what happens to a problem that is only solvable in exponential time. For such a problem the corresponding input length would only grow additively, when the running time grows by a multiplicative factor. Indeed for algorithms running in time 2^n, doubling the running time only adds 1 to n_0.

For such reasons alone, polynomial running times are vastly preferrable to exponential running times. But for many classes of problems polynomial running time often offers a "clever alternative", or an "insightful solution" to the "brute force" exponential time solution. As a result the search for polynomial time algorithms is a major focus of research in the theory of computing.

5 The P vs. NP question

One of the most interesting mathematical questions of the current era stems from the gap between exponential time and polynomial time complexity. There is a wide variety, a "class", of natural problems for every one of which there is a natural algorithm running in exponential time. However for some specific problems in the class, one can also find other algorithms, sometimes very clever and unnatural ones, that run in polynomial time. The question as to whether such clever algorithms for every problem in the class is what leads to the famed "Is P = NP?" question developed in the early 70s in the works of Cook [2], Levin [9] and Karp [8].

To parse the notation of the sentence, let us first clarify that the two sides of the equation refer to "complexity classes", i.e., an infinitely large collection of computational problems. P is the class already described informally above, i.e., all computational problems that can be solved in polynomial time. (Strictly speaking the class P only contains computational problems described by Boolean functions, i.e., by functions whose output is either 0 or 1, but the difference is not significant to this article.)

NP similarly describes another complexity class, the class of "search problems", i.e., problems where the goal is to *search* for some solution that satisfies various constraints (and optimizes some objective function), where it is easy to check if a solution is valid. Let us start with some examples.

Example 6: Map Coloring: This is the cartographer's problem: Given a map of potential nations in some region of a world, can you color all nations with one of three colors so that nations sharing a border not have the same color? To feed such a problem as input to a computer, one only needs to list all the nations, and for each nation, list all other nations that share a border with this nation. It is well known (a major result of 20th century mathematics) that if nations are connected regions then every map can be colored with four colors. But in some cases maps can be colored with three or fewer colors, and in others four colors are necessary. The *decision* version of the problem simply asks the computer to output true if the map is colorable with 3 colors and false otherwwise. The *search* problem would ask for a coloring of the nations with the three colors when the decision version returns true.

Example 7: Travelling Salesperson Problem: In modern language this could be called the "super-GPS" problem. Many of the readers are probably familiar with the "GPS"-based navigation devices - you get into a car and punch in a destination and the GPS device (a computer) returns the shortest path to the destination. Now consider a more sophisticated problem. You have a set of places you would like to go to, and have no desires on which order you would like to visit them. Can you punch in the list of all destinations and ask the device to return the shortest tour that will take you to all places on your list and bring you back to your starting point? Such a device would be the "super-GPS" device, and as it turns out it would be solving a significantly harder computational problem than the standard GPS device solves - the underlying problem is the TSP (for Travelling Salesperson Problem). The input in the TSP is a list of "cities" (or locations one would like to visit), along with a distance table (matrix, if you prefer) that tells the distance between each pair of cities. The decision version of the problem is also given a limit D on the total distance one is willing to travel. The decision problem's output should be true if there is a tour that visits every city and returns to the origin with total distance being at most D, and false otherwise. In the search problem the output should include the order in which the cities should be visited (in the former case).

Example 8: Shortest Supersequence: The next problem came up as part of the program to sequence the human genome. As the reader may know the "genome" (or the entire genetic information) of any human can be expressed as a long sequence of letters from the alphabet $\{A, C, G, T\}$. The sequencing challenge faced by the biologists at the end of the twentieth century was to find this sequence exactly. Over the years the biolgists had found ways of finding moderate length "substrings" (description of the sequence for a contiguous portion of the genome) of the genome, but now the challenge is/was: how can one glue all the little pieces together (that are overlapping) to get the full sequence out. A plausible assumption (not always valid, but often helpful) was that the original sequence was the *smallest* possible sequence that contained all the given moderate length sequences as substrings. If so, can we find the supersequence? This the Shortest supersequence problem. The input here is a collection of strings $S_1, \ldots, S_n$ over some finite alphabet (say $\{A, C, G, T\}$), and possibly a target length L. The decision version of the problem asks: Is there a string T of length at most L such that each of the strings $S_1, \ldots, S_n$ appears as a contiguous subsequence of T. The search problem would require that the output includes such a string T if it exists.

All the above problems, as also the problem of Factoring (Example ???) and Divisor (Example ???) share some common features. In each case, there is an underlying decision question. If the answer to the decision question is "true", then a more refined search question can be posed. And, crucially, the solution to the search question would "certify" the truth. For example in the map-coloring problem, if a map is 3-colorable, the answer to the search problem would certify it by assigning colors to each of the nations, and then it would be "easy" for us to *verify* the validity of the solution - we simply have to make sure that for each pair of nations, if they share a border, then their assigned colors are different. For n nations, this takes at most n^2 time. In contrast the number of possible colorings is 3^n (three choices for each nation) and for all we know, only one of these might be valid! Thus, naively it seems searching for a solution is *hard*, while verifying a solution is *easy*. This is indeed the heart of the complexity class NP. Indeed the reader is encouraged to see that each of the problems listed above have "easy" verification procedures given the output of the search algorithm, when the decision problem has as output true,

Formally, a decision problem Π (one where the goal is to output some true/false value) is said to be in NP, if there is a polynomial time verification algorithm V and polynomial f such that Π outputs true on some input x of length n if and only if there is some auxiliary input y (the solution) of length at most $f(n)$ such that V also outputs true on input x *and* y. [Note, that even if Π outputs true on x there may exist y such that V outputs false on input x and y. This simply means that y is not the right solution to x; but not that x does not have a solution.]

By definition (and by formalizing one's intuition about the definition) every problem in NP can be solved in exponential time. But this does not imply that every problem in NP *requires* exponential time to solve. Indeed NP includes as a subclass the set of problems P which can be solved in polynomial time. (In some cases, the algorithm that places some problem in P is highly non-trivial and counterintuitive.) The main question about NP, encapsulated by the compressed question “Is P = NP?” is: Can every problem in NP be solved in polynomial time? This question remains open to this day, and is a major challenge to mathematics.

The more common belief is that NP does not equal P. This belief is supported by the fact that some problems, like factoring, have been considered by mathemticians for centuries and they have resisted attacks so far and no polynomial time algorithm has been found. We will focus on the implications of P being equal or unequal to NP later, but let us turn to what we do know.

One of the most interesting features of NP is that we do know which problems are the “hardest” within the class, in a rough sense. There is a large class of such problems and these are referred to as the NP-complete problems, and the problems listed above in Examples 6, 7, and 8 are all examples of NP-complete problems. In what sense are they the hardest? If any of these turn out to to have a polynomial time algorithm, then we know NP = P, and so every one of these and every other problem in NP would have a polynomial tie algorithm. Of course, if any of these do not have a polynomial time algorithm, then NP $\neq$ P (since the problem is in NP and would not be in P). It is perhaps interesting to note that one problem we’ve considered so far, namely Factoring, is not known to be in P but not known to be NP-complete either (and suspected not to be NP-complete). We will see the relevance below. We also not that in contrast the Divisor problem is very likely to be NP-complete (i.e., it is NP-complete is we assume certain very likely, but unproven, statements about the density of prime numbers). For the discussion below, we will simply regard it as NP-complete.

NP-complete problems are possibly the highlights of NP. They come up in a variety of contexts, and seem to reflect very disparate aspects of mathematical/human quests. (Already in the 1980s hundreds of problems were seen to be NP-complete and the work by Garey and Johnson [6] compiled an enormous list of such problems. Today the list is just too diverse to allow a complete enumeration.) Yet, NP-completeness implies that they are really the same problem in many different guises. An efficient solution to *any* one problem immediately solves all the others almost as efficiently. Thus the diversity is just a superficial one, and NP-completeness brings out the hidden commonality.

Concretely, one implication of NP-completeness, is that the decision version of NP-complete problems are as easy/hard as the search versions. This ought to be somewhat surprising. Take for instance the map-coloring problem. Suppose someone manages to come up with an “oracle” that can tell which maps are colorable with 3-colors and which ones are not. The “search-decision equivalence” implies that by repeatedly invoking this oracle (polynomially many times) one can actually come up with a valid 3-coloring of any map that is 3-colorable. Coming up with this procedure to use the oracle is not trivial. On the other hand, for the Divisor problem (where the decision version simply asks if there is a divisor X of a given number Z such that $L \leq X \leq U$), it is easy to find such a divisor using the oracle by a “divide-and-conquer” approach: If the oracle returns true, then see if there is a divisor between L and M where $M \approx (L+U)/2$. If so, search in this smaller range or else search in the range $M+1$ to U. In either case, we’ve reduced the search space by half with one oracle call! Indeed with some care and using the presumed NP-completeness of Divisor, as well as the NP-completeness of map coloring, the reader should be able to see how to obtain a 3-coloring of any 3-colorable map using a map-coloring oracle!

Given the fundamental importance of the P vs. NP question, modern complexity theorists have strengthened the Church-Turing hypothesis into a significantly stronger one: They now assert/believe that any physically implementable computational system can be simulated by a Turing algorithm with only a polynomial slowdown. Combined with the belief that NP does not equal P, this turns into limitations on the power of human reasoning as well as nature, and thus turns into one more “law of nature”. We attempt explain why by exploring the implications of the inequality $P \neq NP$ below.

5.1 Optimization

A vast area where humans, and increasingly computers, apply their analytic power is to optimize the working of some system: This could range from minimizing the size of a computer chip, or designing an efficient airline schedule given aircraft and crew considerations and passenger demand, or the optimal location of emergency services (hospitals, fire stations) so as to minimizee the time to respond to an emergency etc. All such optimization problems tend to have this “verifiability” feature: Given a “purported solution” it is easy to measure how good it is, and if it satisfies all the constraints one has. The only hard part might be searching for such a solution. A strikingly large subset of these problems end up being NP-complete. To understand the implication, note that most of the optimization is currently done carefully, with experts from the domain using all their years of accumulated wisdom to “heuristically” optimize the given system. Doing so often requires significant creativity. If NP were equal to P, there would essentially be an automated way of optimizing every such system. We could replace all the individual experts with one large computer and it would simply optimize every such system. Of course, the belief that NP does not equal P suggests that such hope/optimism is not likely to come true, and we will have to continue to exploit human creativity and accumulated wisdom!

5.2 Market behavior

A central notion in economics is the equilibrium principle: namely that whenever selfish players come together to participate in some joint behaviour (like buying/selling goods, or engaging in joint contracts) then there is an equilibrium behavior by which each player has a strategy for its behavior which is optimal (maximizes its own profit) given the behavior of the other players. If each player could also find its “optimal” strategy, then this could be one way to explain the behavior of economic systems (or markets) and explain how they tend to evaluate the price of various goods. But can an individual player really determine its optimal behavior, or can a system really somehow find the equilibrium behavior? It turns out that the evidence is overwhelmingly negative. If one seeks an equilibrium under even slightly restricted conditions, then the problem turns out to be NP-complete (see e.g., [7]). And very recently, computational complexity results have started to show that finding any form of equilibrium may be hard (though not as hard as a general NP problem) [4].

5.3 Biology

Optimization is not the pursuit only of the most advanced form of intelligent lifeforms. It is a much more basic natural phenomenon: Place a ball on the top of a sloped surface and it will roll down to the bottom to *minimize* its potential energy. More complex behavior of this form is exhibited by molecules fold themselves to minimize their chemical energy; and quite often the behavior of a molecule (especially large ones) is a function of its geometric shape. This is a principle used in the design of advanced drugs, and indeed the behavior of many protiens (which are big complex molecules) is determined by their shape. An early belief in biology was that protiens fold up to a shape that would lead to *the minimum* possible potential energy, minimizing over all possible foldings. But as it turns out, the global minimum is NP-complete to compute (see [1, 3]); and assuming that nature can not do something efficiently when a Turing algorithm can not, we are forced to conclude that protiens probably do not fold to the absolute minimum energy shape, but merely look for a local minimum (where any attempt to change shape starts by increasing the potential energy).

5.4 Logic and Mathematics

The foundations of mathematics are in formal logic. A system of logic consists of some axioms and derivations rules. Axioms specify which assertion are assumed true, and derivation rules explain how a collection of assertions can imply other assertions. A mathematical theorem would simply be an assertion that can be derived from the axioms by applying the derivation rules in some careful way. A proof could be a sequence of assertions with an explanation describing how the given assertion is obtained from previous assertions and derivation rules. Now the exact nature of the derivation rules (as also axioms) can vary, and so can the exact rules specifying what makes a proof valid, however broadly two things are always implicit in every system of logic: A *proof* should be *easy* to verify, while finding a proof of a *theorem* may be potentially *hard*. While individual derivation rules try to implement specific ways in which "easiness" can be enforced, the principles of computing give a very general abstraction of all such procedures: Verification should be polynomial time computable! And any polynomial time computable verification function should be as good as any other. In this view of the world, proof verification becomes the canonical polynomial time computable task. Howeever finding the proof of a theorem remains hard. Specifically one define the SHORTPROOF? problem, whose input is a purported theorem statement T (written over some alphabet) and the integer n (represented by a sequence of n ones, say), with the decision question being: Is there a proof P of T of length at most n, according to some given verification algorithm V. SHORTPROOF? turns out to an NP-complete problem suggesting that if P = NP then one could have an automated procedure replacing the creativity mathematicians exhibit in finding proofs of theorems. This, of course, feels very counterintuitive; adding to the general belief that perhaps P may not equal NP and mathematicians can't be replaced by computers after all.

5.5 Cryptography

Not all the news coming from the P vs. NP problem, specifically from the assumption P $\neq$ NP, turns into limitations on the power of computing. Perhaps the most notable counterexample to this is the current field of cryptography. If one thinks a little about this: Cryptography is doing something (hopefully) amazing: My internet provider can see every bit of information that flows in and out of my computer, and in particular overhears everything I say to my bank during my electronic banking transactions. Yet the internet service provider (or some disgruntled employee there) can not start pretending to be me when talking to my bank and therein lies the security of electronic commerce. How is this made possible? Modern cryptography has made it possible for two complete strangers to come together and generate a secure "secret" communication channel between them, while an eavesdropper listening to every bit of information exchanged by the two strangers can not figure out their secrets. Such cryptography relies on the hope that P $\neq$ NP (and more), and in particular in the phenomenon that some functions can be very easy to compute but hard to invert. Indeed in their seminal work in 1976 creating such a "secret-exchange" protocol [5], the authors Diffie and Hellman point out that “We are on the brink of a revolution in cryptography …” essentially because of the possibility that P $\neq$ NP. They then go on to construct the secret-exchange protocol which seems secure to this day (no one knows how to break it); which would certainly break if P = NP; but with the possibility remaining alive that P may not equal NP and still their scheme (as also all forms of cryptography) may be breakable. Nevertheless their work and subsequent works in the theory of cryptography have managed to create the possibility a remarkably secure based on just a few assumptions (which are somewhat stronger than assuming $P \neq NP$.

6 Conclusions

We conclude this article by repeating a few points made in detail earlier: Computing is not just the study of what computers do, or can do; it is a much broader study of logical reasoning entirely and offers the foundational framework for a wide variety of human activities. The examples of addition and multiplication of numbers should highlight how many of us “compute” all the time; but more importantly every other form of reasoning we do is a form of computation as well. So in effect the study of computing is one of the purest ways of studying human or more generally behavior of “intelligent” entities. Given its ambititious nature, it should not come as a surprise that many of the tasks one would like to compute become impossible. But given the pervasive nature of computing, such “impossibility”, or in the case of NP-completeness “infeasibility”, results should be thought of as “natural laws” that we can not afford to overlook.

Of course, the science of computing remains intimately connected to the practice of computing; and as computers become more powerful, more interconnected, and more capable of controlling mechanical devices and more ambititous in their attempt to communicate with humans, the nature of the science is also changing with it. A modern theory of “interacting computers” sees many more laws than just “universality” or “irreversibility” (NP-completeness). It should be safe to assume that the course of the twenty-first century will be dictated to a large extent by the new paths that the science and technology take.

Acknowledgments

Thanks to Joe Kilian for permission to use Example 4. Thanks for Scott Aaronson and Albert Meyer for reminding me of Example 5. Thanks to Costis Daskalakis for pointers to works in game theory.

References

[1] Bonnie Berger and Frank Thomson Leighton. Protein folding in the hydrophobic-hydrophilic(hp) model is np-complete. *Journal of Computational Biology*, 5(1):27–40, 1998.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971.

[3] Pierluigi Crescenzi, Deborah Goldman, Christos H. Papadimitriou, Antonio Piccolboni, and Mihalis Yannakakis. On the complexity of protein folding. *Journal of Computational Biology*, 5(3):423–466, 1998.

[4] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *Communications of the ACM*, 52(2):89–97, 2009.

[5] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability.* Freeman, 1979.

[7] Itzhak Gilboa and Eitan Zemel. Nash and correlated equilibria: Some complexity considerations. *Games and Economic Behavior*, 1:80–93, 1989.

[8] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations, (R. Miller, J. Thatcher eds.)*, pages 85–103, 1972.

[9] Leonid A. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.

[10] Yuri Matiyasevich. *Hilbert's 10th Problem.* The MIT Press, 1993.

[11] Bjorn Poonen. Undecidability in number theory. *Notices of the AMS*, 55(3):344–350, March 2008.

[12] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. A correction *ibid*, 43, 544-546.