What if P equals NP?

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# What if P equals NP?

### 

* Explore the consequences of
* *Search-to-decision* reduction: transform algorithms that solve decision version to search version for -complete problems.
* Optimization and learning problems
* Quantifier elimination and solving problems in the polynomial hierarchy.
* What is the evidence for vs ?

*“You don’t have to believe in God, but you should believe in The Book.”*, Paul Erdős, 1985.[[1]](#footnote-22)

*“No more half measures, Walter”*, Mike Ehrmantraut in “Breaking Bad”, 2010.

*“The evidence in favor of [] and [ its algebraic counterpart ] is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures.”*, Volker Strassen, laudation for Leslie Valiant, 1986.

*“Suppose aliens invade the earth and threaten to obliterate it in a year’s time unless human beings can find the [fifth Ramsey number]. We could marshal the world’s best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the [sixth Ramsey number], however, we would have no choice but to launch a preemptive attack.”*, Paul Erdős, as quoted by Graham and Spencer, 1990.[[2]](#footnote-23)

We have mentioned that the question of whether , which is equivalent to whether there is a polynomial-time algorithm for , is the great open question of Computer Science. But why is it so important? In this chapter, we will try to figure out the implications of such an algorithm.

First, let us get one qualm out of the way. Sometimes people say, *“What if but the best algorithm for 3SAT takes time?”* Well, is much larger than, say, for any input smaller than , as large as a harddrive as you will encounter, and so another way to phrase this question is to say “what if the complexity of 3SAT is exponential for all inputs that we will ever encounter, but then grows much smaller than that?” To me this sounds like the computer science equivalent of asking, “what if the laws of physics change completely once they are out of the range of our telescopes?”. Sure, this is a valid possibility, but wondering about it does not sound like the most productive use of our time.

So, as the saying goes, we’ll keep an open mind, but not so open that our brains fall out, and assume from now on that:

* There is a mathematical god,

and

* She does not “pussyfoot around” or take “half measures”.

What we mean by this is that we will consider two extreme scenarios:

* **3SAT is very easy:** has an or time algorithm with a not too huge constant (say smaller than .)
* **3SAt is very hard:** is exponentially hard and cannot be solved faster than for some not too tiny (say at least ). We can even make the stronger assumption that for every sufficiently large , the restriction of to inputs of length cannot be computer by a circuit of fewer than gates.

At the time of writing, the fastest known algorithm for requires more than to solve variable formulas, while we do not even know how to rule out the possibility that we can compute using gates. To put it in perspective, for the case our lower and upper bounds for the computational costs are apart by a factor of about . As far as we know, it could be the case that -variable can be solved in a millisecond on a first-generation iPhone, and it can also be the case that such instances require more than the age of the universe to solve on the world’s fastest supercomputer.

So far, most of our evidence points to the latter possibility of 3SAT being exponentially hard, but we have not ruled out the former possibility either. In this chapter we will explore some of the consequences of the “ easy” scenario.

## Search-to-decision reduction

A priori, having a fast algorithm for 3SAT might not seem so impressive. Sure, such an algorithm allows us to decide the satisfiability of not just 3CNF formulas but also of quadratic equations, as well as find out whether there is a long path in a graph, and solve many other decision problems. But this is not typically what we want to do. It’s not enough to know *if* a formula is satisfiable: we want to discover the actual satisfying assignment. Similarly, it’s not enough to find out if a graph has a long path: we want to actually *find* the path.

It turns out that if we can solve these decision problems, we can solve the corresponding search problems as well:

### 

Suppose that . Then for every polynomial-time algorithm and ,there is a polynomial-time algorithm such that for every , if there exists satisfying , then finds some string satisfying this condition.

### 

To understand what the statement of search-dec-thm means, let us look at the special case of the problem. It is not hard to see that there is a polynomial-time algorithm such that if and only if is a subset of ’s vertices that cuts at least edges. search-dec-thm implies that if then there is a polynomial-time algorithm that on input outputs a set such that if such a set exists. This means that if , by trying all values of we can find in polynomial time a maximum cut in any given graph. We can use a similar argument to show that if then we can find a satisfying assignment for every satisfiable 3CNF formula, find the longest path in a graph, solve integer programming, and so and so forth.

### 

The idea behind the proof of search-dec-thm is simple; let us demonstrate it for the special case of . (In fact, this case is not so “special” since is -complete, we can reduce the task of solving the search problem for or any other problem in to the task of solving it for .) Suppose that and we are given a satisfiable 3CNF formula , and we now want to find a satisfying assignment for . Define to output if there is a satisfying assignment for such that its first bit is , and similarly define if there is a satisfying assignment with . The key observation is that both and are in , and so if then we can compute them in polynomial time as well. Thus we can use this to find the first bit of the satisfying assignment. We can continue in this way to recover all the bits.

Let be some polynomial time algorithm and some constants. Define the function as follows: For every and , if and only if there exists some (where such that . That is, outputs if there is some string of length such that and the first bits of are . Since, given as above, we can check in polynomial time if , the function is in and hence if we can compute it in polynomial time.

Now for every such polynomial-time and , we can implement as follows:

INPUT: $x\in \{0,1\}^n$  
OUTPUT: $z\in \{0,1\}^{an^b}$ s.t. $V(xz)=1$, -if such $z$ exists. Otherwise -output the empty string.  
  
Initially $z\_0=z\_1=\cdots=z\_{an^b-1}=0$.  
For{$\ell=0,\ldots,an^b-1$}  
Let $b\_0 \leftarrow STARTSWITH\_V(xz\_{0}\cdots z\_{\ell-1}0)$.  
Let $b\_1 \leftarrow STARTSWITH\_V(xz\_{0}\cdots z\_{\ell-1}1)$.  
If{$b\_0=b\_1=0$}   
Return "" # Can't extend $xz\_0\ldots z\_{\ell-1}$ to an accepting input of $V$  
Endif  
If{$b\_0=1$}  
 $z\_\ell \leftarrow 0$ # Can extend $xz\_0\ldots x\_{\ell-1}$ with $0$ to accepting input  
Else  
 $z\_\ell \leftarrow 1$ # Can extend $xz\_0\ldots x\_{\ell-1}$ with $1$ to accepting input  
Endif  
Endfor  
Return $z\_0,\ldots,z\_{an^b-1}$

To analyze searchtodecisionalg, note that it makes invocations to and hence if the latter is polynomial-time, then so is searchtodecisionalg Now suppose that is such that there exists *some* satisfying . We claim that at every step , we maintain the invariant that there exists whose first bits are s.t. . Note that this claim implies the theorem, since in particular it means that for , satisfies .

We prove the claim by induction. For , this holds vacuously. Now for every , if the call returns , then we are guaranteed the invariant by definition of . Now under our inductive hypothesis, there is such that . If the call to returns then it must be the case that , and hence when we set we maintain the invariant.

## Optimization

search-dec-thm allows us to find solutions for problems if , but it is not immediately clear that we can find the *optimal* solution. For example, suppose that , and you are given a graph . Can you find the *longest* simple path in in polynomial time?

### 

This is actually an excellent question for you to attempt on your own. That is, assuming , give a polynomial-time algorithm that on input a graph , outputs a maximally long simple path in the graph .

The answer is *Yes*. The idea is simple: if then we can find out in polynomial time if an -vertex graph contains a simple path of length , and moreover, by search-dec-thm, if does contain such a path, then we can find it. (Can you see why?) If does not contain a simple path of length , then we will check if it contains a simple path of length , and continue in this way to find the largest such that contains a simple path of length .

The above reasoning was not specifically tailored to finding paths in graphs. In fact, it can be vastly generalized to proving the following result:

Suppose that . Then for every polynomial-time computable function (identifying with natural numbers via the binary representation) there is a polynomial-time algorithm such that on input ,

Moreover under the same assumption, there is a polynomial-time algorithm such that for every , outputs such that .

### 

The statement of optimizationnp is a bit cumbersome. To understand it, think how it would subsume the example above of a polynomial time algorithm for finding the maximum length path in a graph. In this case the function would be the map that on input a pair outputs if the pair does not represent some graph and a simple path inside the graph respectively; otherwise would equal the length of the path in the graph . Since a path in an vertex graph can be represented by at most bits, for every representing a graph of vertices, finding corresponds to finding the length of the maximum simple path in the graph corresponding to , and finding the string that achieves this maximum corresponds to actually finding the path.

### 

The proof follows by generalizing our ideas from the longest path example above. Let be as in the theorem statement. If then for every for every string and number , we can test in in time whether there exists such that , or in other words test whether . If is an integer between and (as is the case in the example of longest path) then we can just try out all possibilities for to find the maximum number for which . Otherwise, we can use *binary search* to hone down on the right value. Once we do so, we can use search-to-decision to actually find the string that achieves the maximum.

For every as in the theorem statement, we can define the Boolean function as follows.

Since is computable in polynomial time, is in , and so under our assumption that , itself can be computed in polynomial time. Now, for every and , we can compute the largest such that by a binary search. Specifically, we will do this as follows:

1. We maintain two numbers such that we are guaranteed that .
2. Initially we set and where is the running time of . (A function with running time can’t output more than bits and so can’t output a number larger than .)
3. At each point in time, we compute the midpoint and let .
   1. If then we set and leave as it is.
   2. If then we set and leave as it is.
4. We then go back to step 3, until .

Since shrinks by a factor of , within steps, we will get to the point at which , and then we can simply output . Once we find the maximum value of such that , we can use the search to decision reduction of search-dec-thm to obtain the actual value such that .

One application for optimizationnp is in solving *optimization problems*. For example, the task of *linear programming* is to find that maximizes some linear objective subject to the constraint that satisfies linear inequalities of the form . As we discussed in mincutsec, there is a known polynomial-time algorithm for linear programming. However, if we want to place additional constraints on , such as requiring the coordinates of to be *integer* or  *valued* then the best-known algorithms run in exponential time in the worst case. However, if then optimizationnp tells us that we would be able to solve all problems of this form in polynomial time. For every string that describes a set of constraints and objective, we will define a function such that if satisfies the constraints of then is the value of the objective, and otherwise we set where is some large number. We can then use optimizationnp to compute the that maximizes and that will give us the assignment for the variables that satisfies our constraints and maximizes the objective. (If the computation results in such that then we can double and try again; if the true maximum objective is achieved by some string , then eventually will be large enough so that would be smaller than the objective achieved by , and hence when we run procedure of optimizationnp we would get a value larger than .)

In many examples, such as the case of finding longest path, we don’t need to use the binary search step in optimizationnp, and can simply enumerate over all possible values for until we find the correct one. One example where we do need to use this binary search step is in the case of the problem of finding a maximum length path in a *weighted* graph. This is the problem where is a weighted graph, and every edge of is given a weight which is a number between and . optimizationnp shows that we can find the maximum-weight simple path in (i.e., simple path maximizing the sum of the weights of its edges) in time polynomial in the number of vertices and in .

Beyond just this example there is a vast field of [mathematical optimization](https://en.wikipedia.org/wiki/Mathematical_optimization) that studies problems of the same form as in optimizationnp. In the context of optimization, typically denotes a set of constraints over some variables (that can be Boolean, integer, or real valued), encodes an assignment to these variables, and is the value of some *objective function* that we want to maximize. Given that we don’t know efficient algorithms for complete problems, researchers in optimization research study special cases of functions (such as linear programming and semidefinite programming) where it *is* possible to optimize the value efficiently. Optimization is widely used in a great many scientific areas including: machine learning, engineering, economics and operations research.

### Example: Supervised learning

One classical optimization task is *supervised learning*. In supervised learning we are given a list of *examples* (where we can think of each as a string in for some ) and the *labels* for them (which we will think of simply bits, i.e., ). For example, we can think of the ’s as images of either dogs or cats, for which in the former case and in the latter case. Our goal is to come up with a *hypothesis* or *predictor* such that if we are given a new example that has an (unknown to us) label , then with high probability will *predict* the label. That is, with high probability it will hold that . The idea in supervised learning is to use the *Occam’s Razor principle*: the simplest hypothesis that explains the data is likely to be correct. There are several ways to model this, but one popular approach is to pick some fairly simple function . We think of the first inputs as the *parameters* and the last inputs as the example data. (For example, we can think of the first inputs of as specifying the weights and connections for some neural network that will then be applied on the latter inputs.) We can then phrase the supervised learning problem as finding, given a set of labeled examples , the set of parameters that minimizes the number of errors made by the predictor .[[3]](#footnote-39)

In other words, we can define for every set as above the function such that . Now, finding the value that minimizes is equivalent to solving the supervised learning problem with respect to . For every polynomial-time computable , the task of minimizing can be “massaged” to fit the form of optimizationnp and hence if , then we can solve the supervised learning problem in great generality. In fact, this observation extends to essentially any learning model, and allows for finding the optimal predictors given the minimum number of examples. (This is in contrast to many current learning algorithms, which often rely on having access to an extremely large number of examples far beyond the minimum needed, and in particular far beyond the number of examples humans use for the same tasks.)

### Example: Breaking cryptosystems

We will discuss *cryptography* later in this course, but it turns out that if then almost every cryptosystem can be efficiently broken. One approach is to treat finding an encryption key as an instance of a supervised learning problem. If there is an encryption scheme that maps a “plaintext” message and a key to a “ciphertext” , then given examples of ciphertext/plaintext pairs of the form , our goal is to find the key such that where is the encryption algorithm. While you might think getting such “labeled examples” is unrealistic, it turns out (as many amateur home-brew crypto designers learn the hard way) that this is actually quite common in real-life scenarios, and that it is also possible to relax the assumption to having more minimal prior information about the plaintext (e.g., that it is English text). We defer a more formal treatment to chapcryptography.

## Finding mathematical proofs

In the context of Gödel’s Theorem, we discussed the notion of a *proof system* (see godelproofdef). Generally speaking, a *proof system* can be thought of as an algorithm (known as the *verifier*) such that given a *statement* and a *candidate proof* , if and only if encodes a valid proof for the statement . Any type of proof system that is used in mathematics for geometry, number theory, analysis, etc., is an instance of this form. In fact, standard mathematical proof systems have an even simpler form where the proof encodes a *sequence* of lines (each of which is itself a binary string) such that each line is either an *axiom* or follows from some prior lines through an application of some *inference rule*. For example, [Peano’s axioms](https://en.wikipedia.org/wiki/Peano_axioms) encode a set of axioms and rules for the natural numbers, and one can use them to formalize proofs in number theory. Also, there are some even stronger axiomatic systems, the most popular one being [Zermelo–Fraenkel with the Axiom of Choice](https://en.wikipedia.org/wiki/Zermelo%E2%80%93Fraenkel_set_theory) or ZFC for short. Thus, although mathematicians typically write their papers in natural language, proofs of number theorists can typically be translated to ZFC or similar systems, and so in particular the existence of an -page proof for a statement implies that there exists a string of length (in fact often or ) that encodes the proof in such a system. Moreover, because verifying a proof simply involves going over each line and checking that it does indeed follow from the prior lines, it is fairly easy to do that in or (where as usual denotes the length of the proof ). This means that for every reasonable proof system , the following function is in , where for every input of the form , if and only if there exists with s.t. . That is, if there is a proof (in the system ) of length at most bits that is true. Thus, if , then despite Gödel’s Incompleteness Theorems, we can still automate mathematics in the sense of finding proofs that are not too long for every statement that has one. (Frankly speaking, if the shortest proof for some statement requires a terabyte, then human mathematicians won’t ever find this proof either.) For this reason, Gödel himself felt that the question of whether has a polynomial time algorithm is of great interest. As Gödel wrote [in a letter to John von Neumann](https://rjlipton.wordpress.com/the-gdel-letter/) in 1956 (before the concept of or even “polynomial time” was formally defined):

One can obviously easily construct a Turing machine, which for every formula in first order predicate logic and every natural number , allows one to decide if there is a proof of of length (length = number of symbols). Let be the number of steps the machine requires for this and let . The question is how fast grows for an optimal machine. One can show that [for some constant ]. If there really were a machine with (or even ), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem,[[4]](#footnote-46) the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number so large that when the machine does not deliver a result, it makes no sense to think more about the problem.

For many reasonable proof systems (including the one that Gödel referred to), is in fact -complete, and so Gödel can be thought of as the first person to formulate the vs question. Unfortunately, the letter was [only discovered in 1988](https://www.win.tue.nl/~gwoegi/P-versus-NP/sipser.pdf).

## Quantifier elimination (advanced)

If then we can solve all *search* and *optimization* problems in polynomial time. But can we do more? It turns out that the answer is that *Yes we can!*

An decision problem can be thought of as the task of deciding, given some string the truth of a statement of the form

for some polynomial-time algorithm and polynomial . That is, we are trying to determine, given some string , whether *there exists* a string such that and satisfy some polynomial-time checkable condition . For example, in the *independent set* problem, the string represents a graph and a number , the string represents some subset of ’s vertices, and the condition that we check is whether and there is no edge in such that both and .

We can consider more general statements such as checking, given a string , the truth of a statement of the form

which in words corresponds to checking, given some string , whether *there exists* a string such that *for every* string , the triple satisfy some polynomial-time checkable condition. We can also consider more levels of quantifiers such as checking the truth of the statement

and so on and so forth.

For example, given an -input NAND-CIRC program , we might want to find the *smallest* NAND-CIRC program that computes the same function as . The question of whether there is such a that can be described by a string of at most bits can be phrased as

which has the form existsforalleq.[[5]](#footnote-51) Another example of a statement involving levels of quantifiers would be to check, given a chess position , whether there is a strategy that guarantees that White wins within steps. For example is we would want to check if given the board position , *there exists* a move for White such that *for every* move for Black *there exists* a move for White that ends in a a checkmate.

It turns out that if then we can solve these kinds of problems as well:

If then for every , polynomial and polynomial-time algorithm , there is a polynomial-time algorithm that on input returns if and only if

where and is either or depending on whether is odd or even, respectively.[[6]](#footnote-53)

### 

To understand the idea behind the proof, consider the special case where we want to decide, given , whether for every there exists such that . Consider the function such that if there exists such that . Since runs in polynomial-time and hence if , then there is an algorithm that on input outputs if and only if there exists such that . Now we can see that the original statement we consider is true if and only if for every , , which means it is false if and only if the following condition holds: there exists some such that . But for every , the question of whether the condition is itself in (as we assumed can be computed in polynomial time) and hence under the assumption that we can determine in polynomial time whether the condition , and hence our original statement, is true.

We prove the theorem by induction. We assume that there is a polynomial-time algorithm that can solve the problem eq:QBF for and use that to solve the problem for . For , iff which is a polynomial-time computation since runs in polynomial time. For every , define the statement to be the following:

By the definition of , for every , our goal is that if and only if there exists such that is true.

The *negation* of is the statement

where is if was and is otherwise. (Please stop and verify that you understand why this is true, this is a generalization of the fact that if is some logical condition then the negation of is .)

The crucial observation is that is exactly a statement of the form we consider with quantifiers instead of , and hence by our inductive hypothesis there is some polynomial time algorithm that on input outputs if and only if is true. If we let be the algorithm that on input outputs then we see that outputs if and only if is true. Hence we can rephrase the original statement eq:QBF as follows:

but since is a polynomial-time algorithm, equivalentqbfinducteq is clearly a statement in and hence under our assumption that there is a polynomial time algorithm that on input , will determine if equivalentqbfinducteq is true and so also if the original statement eq:QBF is true.

The algorithm of PH-collapse-thm can also solve the search problem as well: find the value that certifies the truth of eq:QBF. We note that while this algorithm is in polynomial time, the exponent of this polynomial blows up quite fast. If the original NANDSAT algorithm required time, solving levels of quantifiers would require time .[[7]](#footnote-56)

### Application: self improving algorithm for

Suppose that we found a polynomial-time algorithm for that is “good but not great”. For example, maybe our algorithm runs in time for some not too small constant . However, it’s possible that the *best possible* SAT algorithm is actually much more efficient than that. Perhaps, as we guessed before, there is a circuit of at most gates that computes 3SAT on variables, and we simply haven’t discovered it yet. We can use PH-collapse-thm to “bootstrap” our original “good but not great” 3SAT algorithm to discover the optimal one. The idea is that we can phrase the question of whether there exists a size circuit that computes 3SAT for all length inputs as follows: *there exists* a size circuit such that *for every* formula described by a string of length at most , if then *there exists* an assignment to the variables of that satisfies it. One can see that this is a statement of the form existsforallexistseq and hence if we can solve it in polynomial time as well. We can therefore imagine investing huge computational resources in running one time to discover the circuit and then using for all further computation.

## Approximating counting problems and posterior sampling (advanced, optional)

Given a Boolean circuit , if then we can find an input (if one exists) such that . But what if there is more than one like that? Clearly we can’t efficiently output all such ’s; there might be exponentially many. But we can get an arbitrarily good multiplicative approximation (i.e., a factor for arbitrarily small ) for the number of such ’s, as well as output a (nearly) uniform member of this set. The details are beyond the scope of this book, but this result is formally stated in the following theorem (whose proof is omitted).

Let be some polynomial-time algorithm, and suppose that . Then there exists an algorithm that on input , runs in time polynomial in and outputs a number in satisfying

$$(1-\epsilon)COUNT\_V(x,m,\epsilon) \leq \Bigl|\{ y \in \{0,1\}^m \;:\; V(xy)=1 \} \Bigr| \leq (1+\epsilon)COUNT\_V(x,m,\epsilon) \;.
$$

In other words, the algorithm gives an approximation up to a factor of for the number of *witnesses* for with respect to the verifying algorithm . Once again, to understand this theorem it can be useful to see how it implies that if then there is a polynomial-time algorithm that given a graph and a number , can compute a number that is within a factor equal to the number of simple paths in of length . (That is, is between to times the number of such paths.)

**Posterior sampling and probabilistic programming.** The algorithm for counting can also be extended to *sampling* from a given posterior distribution. That is, if is a Boolean circuit and , then if we can sample from (a close approximation of) the distribution of uniform conditioned on . This task is known as *posterior sampling* and is crucial for Bayesian data analysis. These days it is known how to achieve posterior sampling only for circuits of very special form, and even in these cases more often than not we do have guarantees on the quality of the sampling algorithm. The field of making inferences by sampling from posterior distribution specified by circuits or programs is known as [probabilistic programming](https://en.wikipedia.org/wiki/Probabilistic_programming_language).

## What does all of this imply?

So, what will happen if we have a algorithm for ? We have mentioned that -hard problems arise in many contexts, and indeed scientists, engineers, programmers and others routinely encounter such problems in their daily work. A better algorithm will probably make their lives easier, but that is the wrong place to look for the most foundational consequences. Indeed, while the invention of electronic computers did of course make it easier to do calculations that people were already doing with mechanical devices and pen and paper, the main applications computers are used for today were not even imagined before their invention.

An exponentially faster algorithm for all problems would be no less radical an improvement (and indeed, in some sense would be more) than the computer itself, and it is as hard for us to imagine what it would imply as it was for Babbage to envision today’s world. For starters, such an algorithm would completely change the way we program computers. Since we could automatically find the “best” (in any measure we chose) program that achieves a certain task, we would not need to define *how* to achieve a task, but only specify tests as to what would be a good solution, and could also ensure that a program satisfies an exponential number of tests without actually running them.

The possibility that is often described as “automating creativity”. There is something to that analogy, as we often think of a creative solution as one that is hard to discover but that, once the “spark” hits, is easy to verify. But there is also an element of hubris to that statement, implying that the most impressive consequence of such an algorithmic breakthrough will be that computers would succeed in doing something that humans already do today. In fact, creativity already is to a large extent automated or minimized (e.g., just see how much popular media content is mass-produced), and as in most professions we should expect to see the need for humans diminish with time even if .

Nevertheless, artificial intelligence, like many other fields, will clearly be greatly impacted by an efficient 3SAT algorithm. For example, it is clearly much easier to find a better Chess-playing algorithm when, given any algorithm , you can find the smallest algorithm that plays Chess better than . Moreover, as we mentioned above, much of machine learning (and statistical reasoning in general) is about finding “simple” concepts that explain the observed data, and if , we could search for such concepts automatically for any notion of “simplicity” we see fit. In fact, we could even “skip the middle man” and do an automatic search for the learning algorithm with smallest generalization error. Ultimately the field of Artificial Intelligence is about trying to “shortcut” billions of years of evolution to obtain artificial programs that match (or beat) the performance of natural ones, and a fast algorithm for would provide the ultimate shortcut.[[8]](#footnote-62)

More generally, a faster algorithm for problems would be immensely useful in any field where one is faced with computational or quantitative problems which is basically all fields of science, math, and engineering. This will not only help with concrete problems such as designing a better bridge, or finding a better drug, but also with addressing basic mysteries such as trying to find scientific theories or “laws of nature”. In a [fascinating talk](http://www.cornell.edu/video/nima-arkani-hamed-morality-fundamental-physics), physicist Nima Arkani-Hamed discusses the effort of finding scientific theories in much the same language as one would describe solving an problem, for which the solution is easy to verify or seems “inevitable”, once found, but that requires searching through a huge landscape of possibilities to reach, and that often can get “stuck” at local optima:

*“the laws of nature have this amazing feeling of inevitability… which is associated with local perfection.”*

*“The classical picture of the world is the top of a local mountain in the space of ideas. And you go up to the top and it looks amazing up there and absolutely incredible. And you learn that there is a taller mountain out there. Find it, Mount Quantum…. they’re not smoothly connected … you’ve got to make a jump to go from classical to quantum … This also tells you why we have such major challenges in trying to extend our understanding of physics. We don’t have these knobs, and little wheels, and twiddles that we can turn. We have to learn how to make these jumps. And it is a tall order. And that’s why things are difficult.”*

Finding an efficient algorithm for amounts to always being able to search through an exponential space and find not just the “local” mountain, but the tallest peak.

But perhaps more than any computational speedups, a fast algorithm for problems would bring about a *new type of understanding*. In many of the areas where -completeness arises, it is not as much a barrier for solving computational problems as it is a barrier for obtaining “closed-form formulas” or other types of more constructive descriptions of the behavior of natural, biological, social and other systems. A better algorithm for , even if it is “merely” -time, seems to require obtaining a new way to understand these types of systems, whether it is characterizing Nash equilibria, spin-glass configurations, entangled quantum states, or any of the other questions where is currently a barrier for analytical understanding. Such new insights would be very fruitful regardless of their computational utility.

If , we can efficiently solve a fantastic number of decision, search, optimization, counting, and sampling problems from all areas of human endeavors.

## Can be neither true nor false?

The [Continuum Hypothesis](https://en.wikipedia.org/wiki/Continuum_hypothesis) is a conjecture made by Georg Cantor in 1878, positing the non-existence of a certain type of infinite cardinality. (One way to phrase it is that for every infinite subset of the real numbers , either there is a one-to-one and onto function or there is a one-to-one and onto function .) This was considered one of the most important open problems in set theory, and settling its truth or falseness was the first problem put forward by Hilbert in the 1900 address we mentioned before. However, using the theories developed by Gödel and Turing, in 1963 Paul Cohen proved that both the Continuum Hypothesis and its negation are consistent with the standard axioms of set theory (i.e., the Zermelo-Fraenkel axioms + the Axiom of choice, or “ZFC” for short). Formally, what he proved is that if ZFC is consistent, then so is ZFC when we assume either the continuum hypothesis or its negation.

Today, many (though not all) mathematicians interpret this result as saying that the Continuum Hypothesis is neither true nor false, but rather is an axiomatic choice that we are free to make one way or the other. Could the same hold for ?

In short, the answer is *No*. For example, suppose that we are trying to decide between the “3SAT is easy” conjecture (there is an time algorithm for 3SAT) and the “3SAT is hard” conjecture (for every , any NAND-CIRC program that solves variable 3SAT takes lines). Then, since for , , this boils down to the finite question of deciding whether or not there is a -line NAND-CIRC program deciding 3SAT on formulas with variables. If there is such a program then there is a finite proof of its existence, namely the approximately 1TB file describing the program, and for which the verification is the (finite in principle though infeasible in practice) process of checking that it succeeds on all inputs.[[9]](#footnote-67) If there isn’t such a program, then there is also a finite proof of that, though any such proof would take longer since we would need to enumerate over all *programs* as well. Ultimately, since it boils down to a finite statement about bits and numbers; either the statement or its negation must follow from the standard axioms of arithmetic in a finite number of arithmetic steps. Thus, we cannot justify our ignorance in distinguishing between the “3SAT easy” and “3SAT hard” cases by claiming that this might be an inherently ill-defined question. Similar reasoning (with different numbers) applies to other variants of the vs question. We note that in the case that 3SAT is hard, it may well be that there is no *short* proof of this fact using the standard axioms, and this is a question that people have been studying in various restricted forms of proof systems.

## Is “in practice”?

The fact that a problem is -hard means that we believe there is no efficient algorithm that solve it in the *worst case*. It does not, however, mean that every single instance of the problem is hard. For example, if all the clauses in a 3SAT instance contain the same variable (possibly in negated form), then by guessing a value to we can reduce to a 2SAT instance which can then be efficiently solved. Generalizations of this simple idea are used in “SAT solvers”, which are algorithms that have solved certain specific interesting SAT formulas with thousands of variables, despite the fact that we believe SAT to be exponentially hard in the worst case. Similarly, a lot of problems arising in economics and machine learning are -hard.[[10]](#footnote-69) And yet vendors and customers manage to figure out market-clearing prices (as economists like to point out, there is milk on the shelves) and mice succeed in distinguishing cats from dogs. Hence people (and machines) seem to regularly succeed in solving interesting instances of -hard problems, typically by using some combination of guessing while making local improvements.

It is also true that there are many interesting instances of -hard problems that we do *not* currently know how to solve. Across all application areas, whether it is scientific computing, optimization, control or more, people often encounter hard instances of problems on which our current algorithms fail. In fact, as we will see, all of our digital security infrastructure relies on the fact that some concrete and easy-to-generate instances of, say, 3SAT (or, equivalently, any other -hard problem) are exponentially hard to solve.

Thus it would be wrong to say that is easy “in practice”, nor would it be correct to take -hardness as the “final word” on the complexity of a problem, particularly when we have more information about how any given instance is generated. Understanding both the “typical complexity” of problems, as well as the power and limitations of certain heuristics (such as various local-search based algorithms) is a very active area of research. We will see more on these topics later in this course.

[[11]](#footnote-70)

## What if ?

So, would give us all kinds of fantastical outcomes. But we strongly suspect that , and moreover that there is no much-better-than-brute-force algorithm for 3SAT. If indeed that is the case, is it all bad news?

One might think that impossibility results, telling you that you *cannot* do something, is the kind of cloud that does not have a silver lining. But in fact, as we already alluded to before, it does. A hard (in a sufficiently strong sense) problem in can be used to create a code that *cannot be broken*, a task that for thousands of years has been the dream of not just spies but of many scientists and mathematicians over the generations. But the complexity viewpoint turned out to yield much more than simple codes, achieving tasks that people had previously not even dared to dream of. These include the notion of *public key cryptography*, allowing two people to communicate securely without ever having exchanged a secret key; *electronic cash*, allowing private and secure transaction without a central authority; and *secure multiparty computation*, enabling parties to compute a joint function on private inputs without revealing any extra information about it. Also, as we will see, computational hardness can be used to replace the role of *randomness* in many settings.

Furthermore, while it is often convenient to pretend that computational problems are simply handed to us, and that our job as computer scientists is to find the most efficient algorithm for them, this is not how things work in most computing applications. Typically even formulating the problem to solve is a highly non-trivial task. When we discover that the problem we want to solve is -hard, this might be a useful sign that we used the wrong formulation for it.

Beyond all these, the quest to understand computational hardness including the discoveries of lower bounds for restricted computational models, as well as new types of reductions (such as those arising from “probabilistically checkable proofs”) has already had surprising *positive* applications to problems in algorithm design, as well as in coding for both communication and storage. This is not surprising since, as we mentioned before, from group theory to the theory of relativity, the pursuit of impossibility results has often been one of the most fruitful enterprises of mankind.

* The question of whether is one of the most important and fascinating questions of computer science and science at large, touching on all fields of the natural and social sciences, as well as mathematics and engineering.
* Our current evidence and understanding supports the “SAT hard” scenario that there is no much-better-than-brute-force algorithm for 3SAT or many other -hard problems.
* We are very far from *proving* this, however. Researchers have studied proving lower bounds on the number of gates to compute explicit functions in *restricted forms* of circuits, and have made some advances in this effort, along the way generating mathematical tools that have found other uses. However, we have made essentially no headway in proving lower bounds for *general* models of computation such as Boolean circuits and Turing machines. Indeed, we currently do not even know how to rule out the possibility that for every , restricted to -length inputs has a Boolean circuit of less than gates (even though there *exist* -input functions that require at least gates to compute).
* Understanding how to cope with this computational intractability, and even benefit from it, comprises much of the research in theoretical computer science.

## Exercises

## Bibliographical notes

As mentioned before, Aaronson’s survey [@aaronson2016p] is a great exposition of the vs problem. Another recommended survey by Aaronson is [@aaronson2005physicalreality] which discusses the question of whether complete problems could be computed by any physical means.

The paper [@buchfuhrer2011complexity] discusses some results about problems in the polynomial hierarchy.

1. Paul Erdős (1913-1996) was one of the most prolific mathematicians of all times. Though he was an atheist, Erdős often referred to “The Book” in which God keeps the most elegant proof of each mathematical theorem. [↑](#footnote-ref-22)
2. The -th Ramsey number, denoted as , is the smallest number such that for every graph on vertices, both and its complement contain a -sized independent set. If then we can compute in time polynomial in , while otherwise it can potentially take closer to steps. [↑](#footnote-ref-23)
3. This is often known as [Empirical Risk Minimization](https://goo.gl/F9AgG8). [↑](#footnote-ref-39)
4. The undecidability of [Entscheidungsproblem](https://en.wikipedia.org/wiki/Entscheidungsproblem) refers to the uncomputability of the function that maps a statement in [first order logic](https://en.wikipedia.org/wiki/First-order_logic) to if and only if that statement has a proof. [↑](#footnote-ref-46)
5. Since NAND-CIRC programs are equivalent to Boolean circuits, the search problem corresponding to circmineq known as the [circuit minimization problem](https://goo.gl/iykqbh) and is widely studied in Engineering. You can skip ahead to selfimprovingsat to see a particularly compelling application of this. [↑](#footnote-ref-51)
6. For the ease of notation, we assume that all the strings we quantify over have the same length , but using simple padding one can show that this captures the general case of strings of different polynomial lengths. [↑](#footnote-ref-53)
7. We do not know whether such loss is inherent. As far as we can tell, it’s possible that the *quantified boolean formula* problem has a linear-time algorithm. We will, however, see later in this course that it satisfies a notion known as -hardness that is even stronger than -hardness. [↑](#footnote-ref-56)
8. One interesting theory is that and evolution has already discovered this algorithm, which we are already using without realizing it. At the moment, there seems to be very little evidence for such a scenario. In fact, we have some partial results in the other direction showing that, regardless of whether , many types of “local search” or “evolutionary” algorithms require exponential time to solve 3SAT and other -hard problems. [↑](#footnote-ref-62)
9. This inefficiency is not necessarily inherent. Later in this course we may discuss results in program-checking, interactive proofs, and average-case complexity, that can be used for efficient verification of proofs of related statements. In contrast, the inefficiency of verifying *failure* of all programs could well be inherent. [↑](#footnote-ref-67)
10. Actually, the computational difficulty of problems in economics such as finding optimal (or any) equilibria is quite subtle. Some variants of such problems are -hard, while others have a certain “intermediate” complexity. [↑](#footnote-ref-69)
11. Talk more about coping with NP hardness. Main two approaches are *heuristics* such as SAT solvers that succeed on *some* instances, and *proxy measures* such as mathematical relaxations that instead of solving problem (e.g., an integer program) solve program (e.g., a linear program) that is related to that. Maybe give compressed sensing as an example, and least square minimization as a proxy for maximum apostoriori probability. [↑](#footnote-ref-70)