

[Sign In](#)[Get started](#)

Published in Towards Data Science

You have **2** free member-only stories left this month.
[Sign up for Medium and get an extra one](#)



David Chong

[Follow](#)Dec 2, 2019 · 7 min read ★ · [Listen](#)

The Aged P versus NP Problem

Why is $P=NP$ such a big deal that it warrants a \$1 million prize?

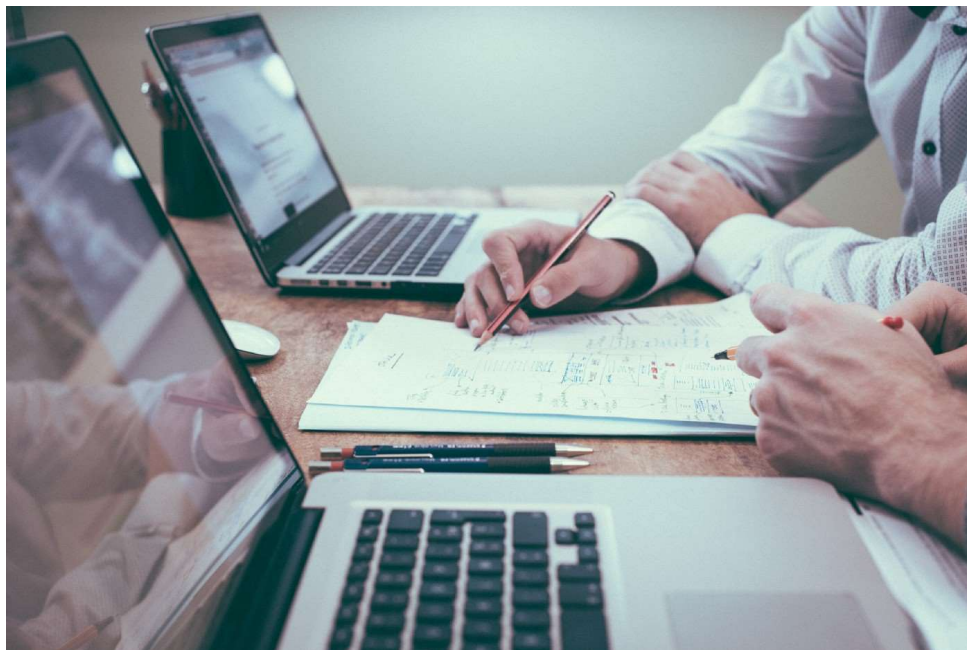


Photo by [Helloquence](#) on [Unsplash](#)

P versus NP. Is it even solvable?

It is one of the seven [Millennium Prize Problems](#) selected by the [Clay Mathematics Institute](#), each of which carries a US\$1,000,000 prize for the first correct solution. It is the most recently conceived problem of the seven (in 1971) and also the easiest to explain (hopefully).

What is it all about?

Before we deep dive, I hope it is safe to assume that those who clicked this article





regarding the technical details but provide some background to those non-technical folks out there.



Essential ideas to know for the non-programming folks

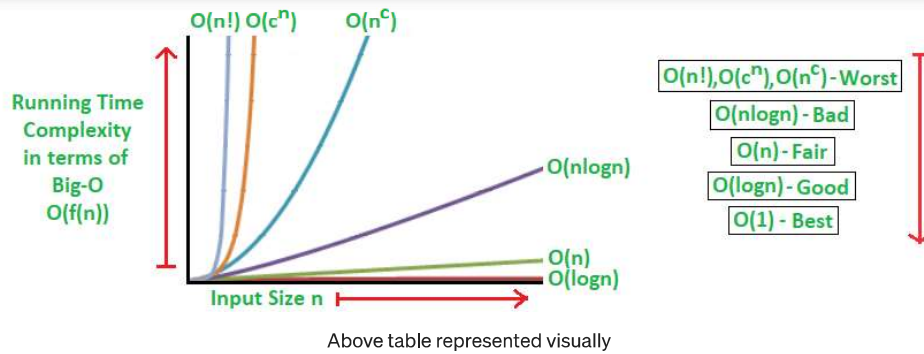
(Those familiar with time and space complexity can skip this section)

- **Algorithms are basically just a set of instructions written in a programming language**
- **Algorithms are the ‘brains’ behind computer programs** — they essentially help solve our day-to-day problems such as vehicle routing, protein folding, sorting, finding prime numbers, graph searching, etc.
- **Every algorithm has a time and space complexity** — this is essentially a technically cooler way of saying that every algorithm takes some amount of time to run and takes some amount of memory on our computer when executed

increasing order of complexity ↓		Big O	Remarks
	Constant	$O(1)$	not affected by input size n
	Logarithmic	$O(\log n)$	e.g. binary search
	Linear	$O(n)$	proportional to input size n
	Linearithmic	$O(n \log n)$	e.g. merge sort, heap sort
	Polynomial	$O(n^k)$	k is some constant, e.g., $k = 1$ is linear, $k = 2$ is quadratic, $k = 3$ is cubic
	Exponential	$O(k^n)$	k is some constant
	Factorial	$O(n!)$	within the exponential family

Big O Notation and Orders of Complexity

- **Polynomial time is slow** —  127 |  1 out time complexity (i.e. how much time an algorithm takes to run), the programming community measures the time an algorithm takes based on input size n . The mathematical notation that describes the limiting behaviour of a function when an argument tends towards a particular value or infinity is called **Big O notation**. Generally, any algorithm that runs in polynomial time (i.e. takes n^k time given input of size n) is considered slow but still ‘accepted’ by the community as the upper limit. Anything slower is deemed not usable by the programming community as time taken scales up too fast relative to input size.



So what then, is the P versus NP problem?

For the record, the status quo is that $P \neq NP$.

P (polynomial time) refers to the class of problems that can be solved by an algorithm in polynomial time. Problems in the **P** class can range from anything as simple as multiplication to finding the largest number in a list. They are the relatively 'easier' set of problems.

NP (non-deterministic polynomial time) refers to the class of problems that can be solved in polynomial time by a non-deterministic computer. This is essentially another way of saying "If I have unlimited compute power (i.e. as many computers as I need), I am capable of solving any problem in at most polynomial time". More intuitively though, it refers to the class of problems that currently, has no way of finding a quick (polynomial time) enough answer, BUT can be quickly **verified** (in polynomial time) if one provides the solution to the problem. The term verified here means that one is able to check that the solution provided is indeed correct.

Clearly, based on the definition above, $P \subseteq NP$. Let's take a look at an example to illustrate this abstract problem.

One of the most common yet effective examples is Sudoku. Given an unsolved Sudoku grid (9 x 9 for example), it would take an algorithm a fair amount of time to solve one. However, if the 9 x 9 grid increases to a 100 x 100 or 10,000 x 10,000 grid, the time it would take to solve it would increase **exponentially** because the problem itself becomes significantly harder. However, given a solved Sudoku grid (of 9 x 9), it is fairly straightforward to verify that the particular solution is indeed correct even if the size scales to 10,000 by 10,000. It would be slower, but the time to check a solution increases at a slower rate.





There are many other **NP** problems out there, including the Knapsack problem and the Traveling Salesmen problem, and they are similar in that they are hard to solve but quick to verify. The fundamental problem we are trying to solve here is:

Does being able to quickly recognize *NP* correct answers mean that there is also a quick way to find them?

If so (i.e. $P=NP$), this could greatly change how we look at these **NP** problems because that means there is a quick way to solve all these problems, just that we haven't been able to figure out how, YET.

If this isn't complicated enough, let's spice it up.

NP-Complete and NP-Hard

Amongst these **NP** problems, there exists a King of all problems which researchers call **NP-Complete** problems. Formally, they are a set of problems to each of which any other **NP** problem can be **reduced** (*addressed below*) in polynomial time and whose solution may still be verified in polynomial time. This means that *any* **NP** problem can be transformed into a **NP-Complete** problem.

Informally, they are the “hardest” of the **NP** problems. Thus if any *one* **NP-Complete** problem can be solved in polynomial time, then *every* **NP-Complete** problem can be solved in polynomial time, and *every* problem in **NP** can be solved in polynomial time (i.e. $P=NP$). The most famous example would be the Traveling Salesmen problem.



Photo by [JESHOOOTS.COM](https://www.jeshoots.com) on [Unsplash](https://unsplash.com)

There also exists a set of problems called ***NP-Hard*** problems. These problems are at least as hard as ***NP*** problems, but **without** the condition that requires it to be solved in polynomial time. This suggests that ***NP-Hard*** problems may not necessarily be part of the ***NP*** class. An example would be solving a chess board — given a state of a chess board, it is almost impossible to tell if a given move at the given state, is in fact the optimal move. Formally, there exists no polynomial time algorithm to *verify* a solution to a ***NP-Hard*** problem.

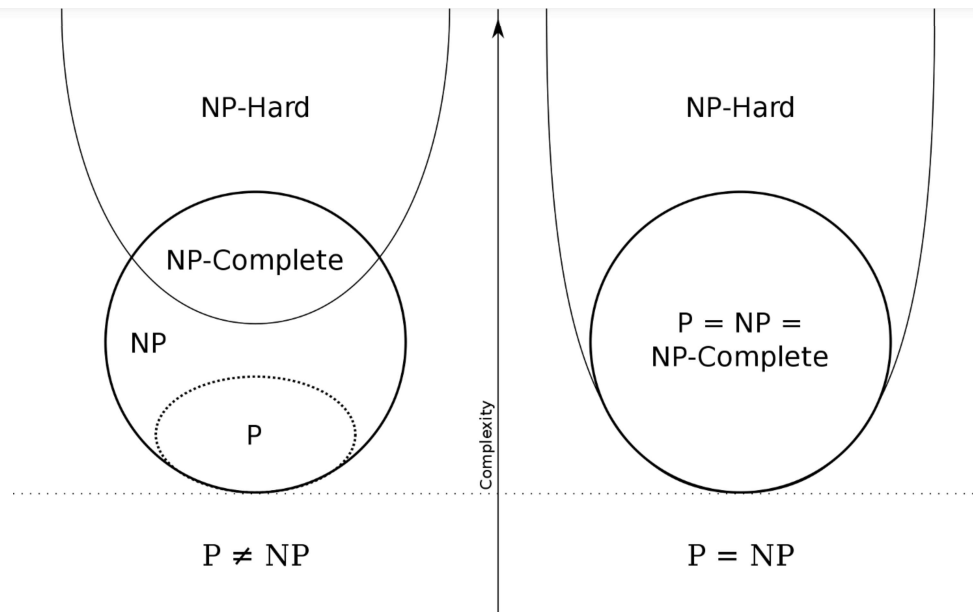
If we put the two together, a ***NP-Complete*** problem implies it being ***NP-Hard***, but a ***NP-Hard*** problem does NOT imply it being ***NP-Complete***.

Defining NP-Completeness

A problem L , is ***NP-Complete*** if:

1. L is NP-Hard
2. L belongs to NP

The diagram below (*focus on the left-hand side*) should make things clearer.



P versus NP diagram (source: https://en.wikipedia.org/wiki/P_versus_NP_problem)

Wait, what does it mean by reducing A to B?

Reduction forms the crux of *NP-Completeness*.

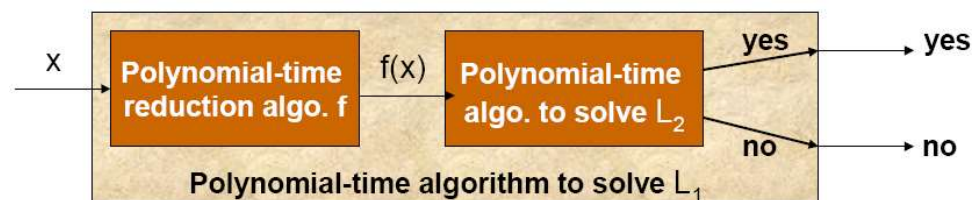
Informally, a problem L_1 can be reduced to another problem L_2 if:

- Any instance of L_1 can be modeled as an instance of L_2
- The solution to the latter provides a solution to the former and vice versa

To understand this intuitively, one can think of it as such:

If L_1 is reducible to L_2 , L_1 must be AT MOST as hard as L_2 . Conversely, L_2 must be AT LEAST as hard as L_1 .

Mathematically, this is denoted as: $L_1 \leq_p L_2$ (read as “ L_1 is polynomially reducible to L_2 ”).



Visual representation of above, where f represents the polynomial-time reduction algorithm (Source: Singapore Management University)



Now that we have defined all these different terms, a more important question to ask is — “Why does all this even matter?”. Having a solution to this problem can have profound consequences not only in academia, but also in practical situations. This includes:

- **Cryptography** — From the many passwords we maintain to the PIN number we use for the ATM, we rely on these codes to govern our everyday lives because they are easy to check but hard for people to crack.
- **Vehicle Routing** — Transportation and the movement of logistics would be optimized across the world, impacting many industries from transport to e-commerce to manufacturing.
- **Protein-Folding** — Understanding or predicting how a given sequence of amino acids (a protein) will fold up to form its 3D shape will help in many areas such as drug design and perhaps even cure cancer.

Concluding Remarks

For me, I’m more of a pragmatist than an idealist. I stand on the side of the fence that **P does not equal** to NP and will probably never will. As much as I would love to uncover the cure for cancer (via fast protein folding) or fast vehicle routing, I believe that the world is meant to be as complex as it is now. Companies exist because they compete with one another to find a better (but never the best) way to solve these problems. These unsolvable problems drive our economy and the world.

If there really IS a solution to this problem, the complexity of the problem warrants a cash prize of a lot more than US\$1,000,000.

Support me! — If you like my content and are *not* subscribed to Medium, do consider supporting me and subscribing via my referral link [here](#) (NOTE: a portion of your membership fees will be apportioned to me as referral fees).

References

1. https://en.wikipedia.org/wiki/P_versus_NP_problem
2. <https://www.youtube.com/watch?v=YX40hbAHx3s>
3. <https://bigthink.com/technology-innovation/what-is-p-vs-np?rebelltitem=1#rebelltitem1>





Sign In

Get started

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

Get this newsletter

