Rashid Lasker

Kate Kostelnik

ENWR 2520

10 May 2017

P ≟ NP

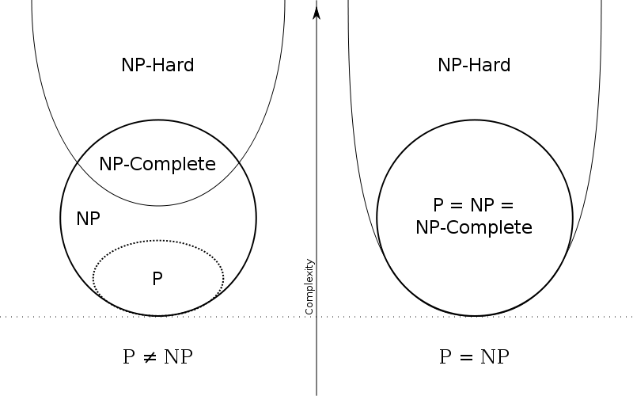
Introduction/Hook:

If you have ever spent a lazy Sunday afternoon playing with puzzles, you’ve come across a puzzle known as Sudoku. The goal is simply to fill a 9 by 9 grid with single digit numbers in a way that each digit is not repeated in each column, row, and 3 by 3 square. The underlying concept of this puzzle is not profoundly complex. All it is rearranging numbers in a grid. However, it’s still a challenging mind game. Perhaps more interestingly, it also plays a key role in one of computer science’s most interesting problems, P vs. NP.

Solving sudoku puzzles is far more difficult than it seems at first glance. As you scale up the difficulty by expanding the grid beyond 16x16 grids, the problem becomes exponentially more complex, taking much longer to solve. In fact, solving sudoku is an NP-complete problem (Yato, 2003). NP-complete is a specific problem class that essentially means a problem is significantly more difficult to solve than to check. This means sudoku puzzles are on the same plane of complexity as protein folding problems. Somehow, this simple puzzle is as difficult as something we are using to understand and combat cancer. This wide spectrum of problems within the same complexity level is why the P vs. NP problem is so important.

Explanation of the problem:

The P vs. NP problem is one of the Clay Mathematics Institute’s Millennium Problems. It’s frequently called one of the most important questions in theoretical computer science, and will net you $1,000,000 for solving it (http://www.claymath.org/millennium-problems/p-vs-np-problem). The specific parameters are fairly nuanced (Cook), but ultimately the problem boils down to proving either P = NP or P ≠ NP.



**Algorithms:**

Before going into the problem itself, you need to know what algorithms in computer science mean. An algorithm is a set of steps conducted in a sequence to reach some solution. In that regard, a program that finds the best move in a game of Othello is an algorithm, as is a program that does simple addition.

**Computational classes:**

P and NP are classes of problems within computer science. P, which stands for polynomial time, represents the set of problems that are reasonably easy to solve. This notion of “easy” is based on how many computations a computer must do given a certain input. An example of a problem in P is two-integer addition. 2+2 is very easy to solve. However, this property scales up as well. It’s very easy for a computer to add up 2,124,542,352 and 113,134,589,238 as well. Modern computers can do addition for numbers that are beyond the scope of the physical universe with relative ease. Therefore, two-integer addition is in P.

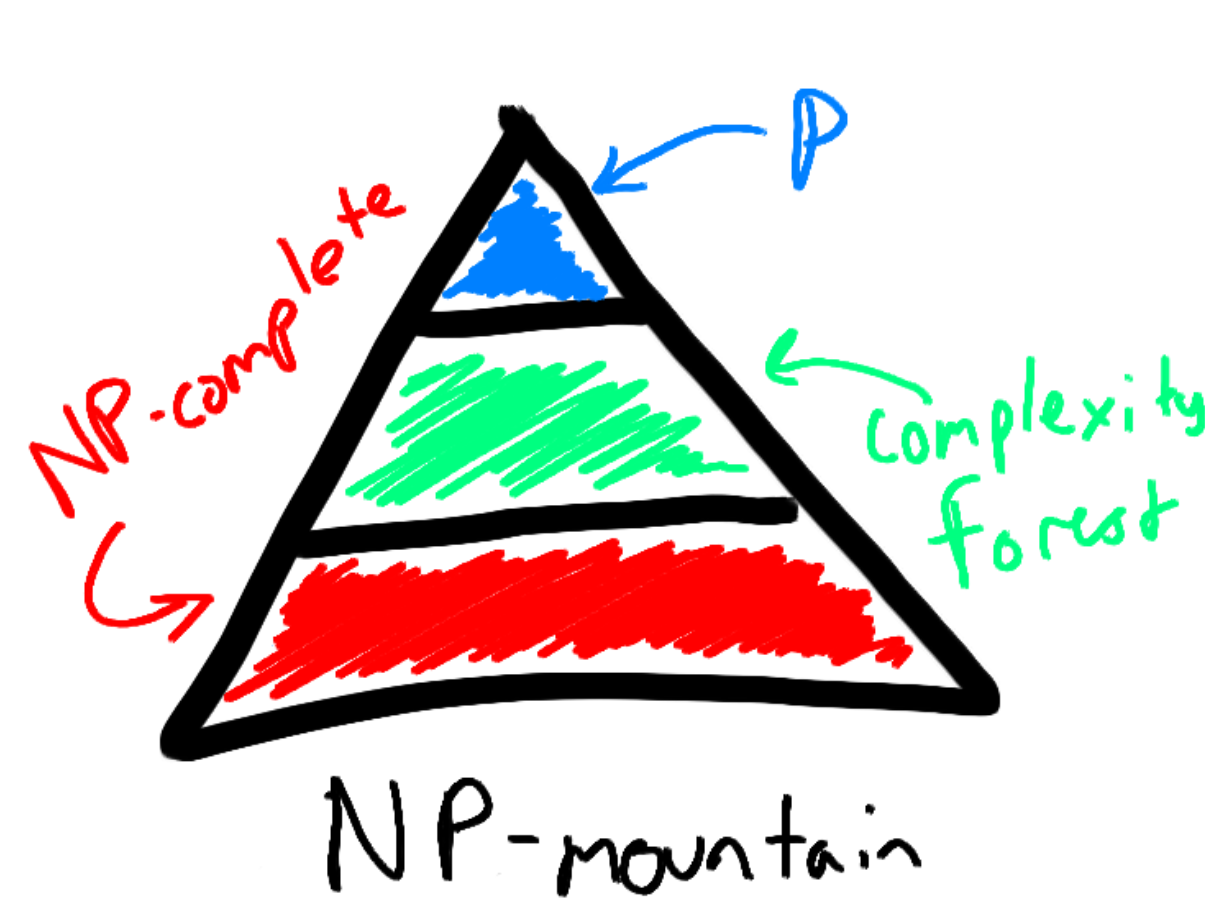
NP, which stands for nondeterministic polynomial time, represents the set of problems that are not necessarily easy to solve, but reasonably easy to check. The sudoku puzzle is in NP because to check it, you just need to check each row, column, and 3x3 square. The characteristic of being easy to check is very important to the relationship between P and NP. By definition, something that is easy to solve is easy to check, because all you need to do is resolve the problem to check it. Therefore, NP includes all of P.

These two sets do not represent all problems however. An example of a problem that is neither in P nor NP is finding the best move in a game of chess. For one, it’s hard to do because of how many possible future states there are in a game of chess. Moreover, it’s hard to check whether you made the best move because the only way to do so is to resolve the problem, which we know is a “hard” process.

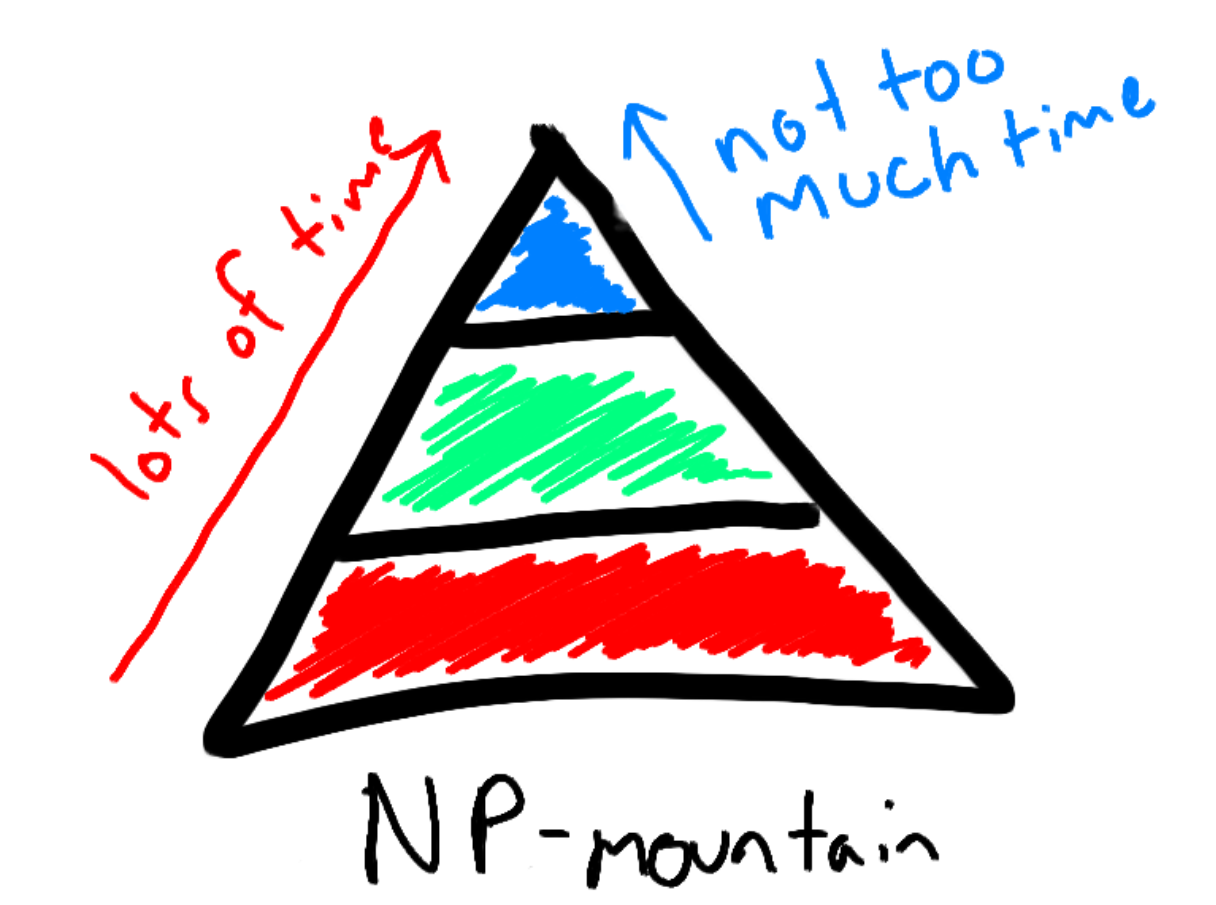
There are other classes of problems relating to P and NP. NP-hard is a class that essentially includes all problems that are at least as hard to solve as the hardest NP problems. This definition leads us to the class of NP-complete. Problems that are NP-complete are both NP and NP-hard. Therefore, they are the hardest NP problems to solve. NP-complete and P essentially represent opposite ends of the NP spectrum, and are therefore the most interesting groups to compare.

An analogy:

In an attempt to visualize the P vs. NP problem, let’s imagine ourselves on a mountain. We might not be mountaineers in our free time, but the goal from any starting point is to reach the summit. The distance to the summit varies on where we start. Let’s imagine the mountain in the following way:



The three zones are separated as follows. The P-zone is the section closest to the summit. From here, it’s possible to reach the top of the summit in a reasonable amount of time. The other two zones are the NP-complete-zone and the Complexity Forest. Essentially, the Complexity Forest separates the P-zone and the NP-complete-zone. As a result, if you start off in the NP-complete-zone, it takes a significantly longer time to reach the summit.

 From both zones, it’s easy to check if you’re at the summit; you just look down at your feet. However, the time it takes to reach the summit from either zone is very different. Herein lies the main difference between P and NP-complete problems. Both types of problems are easy to check if you’re given a solution already. However, it is far easier to find a solution to a P problem, compared to an NP-complete problem.

The above analogy assumes that P ≠ NP. In this scenario, there’s no easy way to bypass the Complexity Forest. However, there is alternative possibility, where P = NP. This would mean that there is not a difference between P and NP problems. Going back to the analogy, if P = NP, there would be a ski-lift that goes over the Complexity Forest entirely. For every NP-complete problem, there would be a shortcut that turns it into a P problem. Therefore, all NP-complete problems would be able to be solved faster, which has widespread ramifications in today’s world.



History of the problem:

The history of the P vs. NP problem begins with huge advances in the formalization of algorithms in the early 1900s. These early developments characterized the nature of algorithms and introduced the idea of computational complexity, from a mathematical perspective. The rudiments of the P vs. NP problem came within a letter from John Nash to the NSA in 1955. The letter explored the exponential complexity of cracking keys in cryptography, with an implication that P ≠ NP. The following year, Kurt Gödel wrote a letter exploring the possibility of improving the rate at which we can prove theorems, which would have ramifications in the automaticity of writing proofs. This automaticity would then translate into the solvability of all NP problems.

The P vs. NP problem became a greater issue in the 1970s, with the advent of computer science. As computers started to become functional, computer scientists gained the capacity to explore problems they couldn’t have before. They discovered that solving some problems was significantly easier than solving others, and started to put these two types of problems in different “buckets.”



With certain problems, like the Discrete Fourier Transform, computer scientists found shortcuts to solve the problems, which moved the problem from the hard bucket to the easy bucket. Moreover, they were able to determine that certain problems had no shortcuts to make them easier, like finding the best move in chess. This left them with a lot of problems that weren’t confirmed to be in one of the two buckets for sure. The problems that were left could have a shortcut, but no one could prove it. Therein lies the P vs. NP problem. If P = NP, the problems left over would all be put in the easy bucket. If P ≠ NP, nothing really changes, the ambiguous pile of problems left over is still there.

The future of P ≟ NP:

There is a school of thought that believes that the overall result of the P vs. NP problem will be inconsequential in the end. This is because of sizeable majority of computer scientists believe that P ≠ NP as is. If P ≠ NP is proven, then nothing will change because of the assumptions already made within the computer science industry. Advances in theoretical computation will thereby rely on things like quantum computing to radically disrupt the speed at which we solve certain problems.

However, there are several computer scientists that are optimistic about the potential about the P = NP problem:

* “If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument could be Gauss.” – Scott Aaronson, MIT
* “The main argument in favor of P ≠ NP is the total lack of fundamental progress in the area of exhaustive search. This is, in my opinion, a very weak argument. The space of algorithms is very large and we are only at the beginning of its exploration. [...] The resolution of Fermat's Last Theorem also shows that very simple questions may be settled only by very deep theories.” — Moshe Y. Vardi, Rice University

Ultimately, we continue to wait on the answer to the P vs. NP problem. If P were to equal NP, our conception of specific problems would fundamentally change. The largest changes would perhaps be seen in the computer security sector. As of now, encryption is dependent on the fact that certain NP problems are unable to be solved easily. If P = NP, there would be a definite shortcut out there for cracking encryptions online. Even if computers able to harness that shortcut are years away, computer scientists will have to totally rethink the way they encrypt computers.

Resources

Aaronson, Scott. “P ≟ NP.” *The University of Texas*, 2017.

Sipser, Michael. “The History and Status of the P versus NP Question.” *Massachusetts Institute of Technology*, 1992.

Yato, Takayuki. “Complexity and Completeness of Finding Another Solution and its Application to Puzzles.” *The University of Tokyo*, 2003.