On the cruel irony of the P-NP problem

18 Dec 2023

Part 1: Basics for Those Who Slept in Class

What is the P-NP Problem?

If you're not familiar with the P-NP problem, it's one of the major unsolved conundrums in computer science, challenging experts for over fifty years. It delves into the realms of time complexity and the intrinsic difficulty of computational problems. Often, media outlets simplify this by referring to the "difficulty" of solving a problem, labeling NP-hard as the toughest category. However, it's not about the intellectual challenge per se; rather, it's about how the time required for a program to solve a problem escalates with the size of the input.

> Throughout this post, I'll use "algorithm" and "problem" somewhat interchangeably.

An algorithm, in essence, is a method or a process for solving a problem. To prove that P equals NP, one needs to **reduce** a problem classified as NP-complete to a problem within the P class. In layman's terms, this means demonstrating they are essentially equivalent.

The significance of the P-NP problem extends into the practical world, particularly concerning our digital security. Most password systems rely on the premise that they cannot be cracked in a reasonable timeframe. Technically, all passwords are breakable, but if it takes a century on a standard computer, they're deemed secure. However, if P were equal to NP, this security assumption would be upended, rendering all passwords vulnerable.

What is Time Complexity?

Time complexity is a measure of how the execution time of a program changes as the size or scale of the input increases. For example, consider a program that finds the maximum value from a set of 100 stock options, taking 100 milliseconds (ms) on a certain computer. The process involves examining each stock option once and comparing each one to the current maximum.

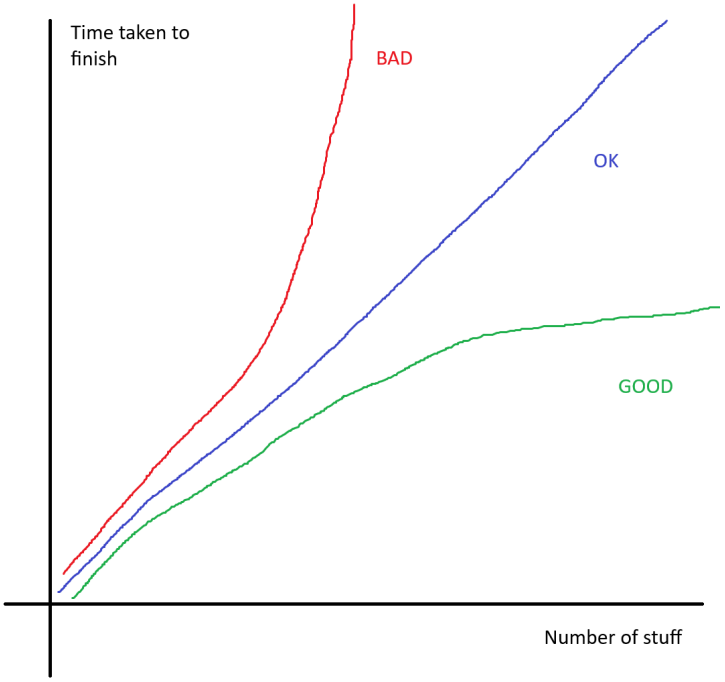
The total time taken can be expressed as:

```
total_time_taken = time_spent_per_stock * number_of_stocks +
time_spent_starting_the_script
```

When the number of stocks is relatively small, even inefficient code can handle the task without much trouble, making detailed analysis unnecessary. However, for large-scale inputs, time complexity becomes critical, especially when dealing with NP-hard problems.

The term "P" stands for polynomial time, which means the time taken by an algorithm as a function of input size can be represented by a polynomial equation, like `y = mx + c` or `y = ax^2 + bx + c`.

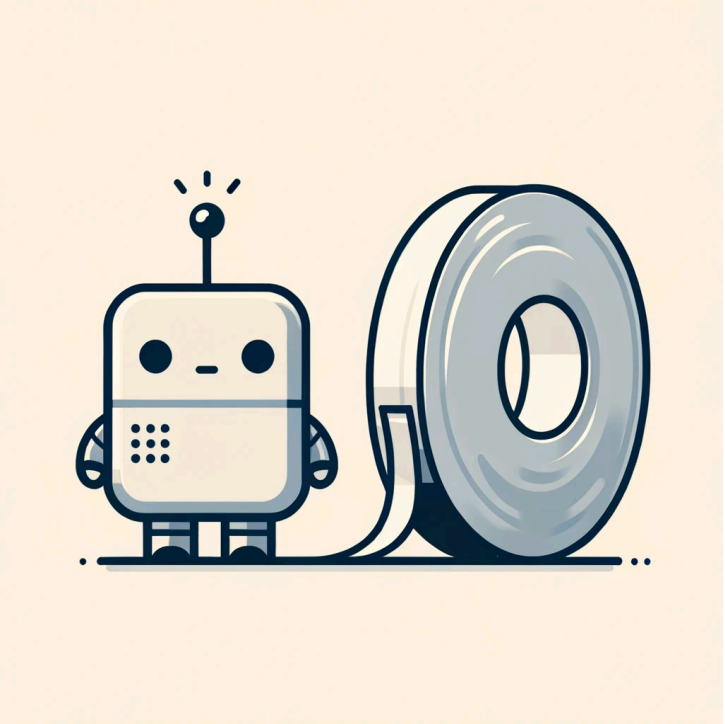
Contrary to common misconception, "NP" does not stand for "Not Polynomial". It actually refers to "Non-deterministic Polynomial time". This describes algorithms that would run in polynomial time on a theoretical non-deterministic Turing machine. While it's true that many NP problems exhibit exponential time complexity on conventional computers, it's important not to conflate NP with exponential time directly.



Behold my crudely drawn time complexity curves

What are Turing Machines?

A brief but essential concept to grasp before diving deeper is Turing machines. Introduced by Alan Turing, these theoretical constructs are fundamental to computer science. They demonstrate that virtually *anything computable can be computed* with just an infinitely long tape and a finitely sized algorithm. We'll explore Turing machines in more detail in another post. For now, just picture a robot working with a really long tape.



He likes collecting tapes

What is Non-Determinism?

To understand non-determinism, let's revisit our stock example. Imagine a computer with enough processing cores that each stock can be assigned to a separate core. In this scenario, all cores can process their respective stocks in parallel. Consequently, regardless of the number of stocks, the processing time remains constant, assuming we have sufficient cores.

The equation for total time taken in a parallel processing scenario simplifies to:

> total_time_taken_for_parallel_script = time_spent_starting_the_script + time_spent_for_all_processes_to_finish

This is effectively constant time, as the bulk of the work happens simultaneously across different cores.

Non-determinism, in a computational sense, can be visualized as a scenario where, at every decision point, a machine can simultaneously explore multiple paths. Unlike sequential machines that must complete one path before starting another, a non-deterministic machine can pursue multiple paths concurrently. However, it's crucial not to confuse non-determinism with quantum entanglement; they are distinct concepts. While parallel processes in non-deterministic systems can be synchronized through various techniques, by default, actions in one do not directly influence the others.

Think of non-deterministic Turing machines as standard Turing machines, but with the ability for the 'robot' to split into multiple versions of itself and follow different paths simultaneously whenever it encounters a decision point.

Lets put together P, NP-complete, NP-hard

1. **P**: Algorithms that solve problems in polynomial time on deterministic Turing machines.
2. **NP-Complete**: Problems solvable in polynomial time on non-deterministic Turing machines, with solutions verifiable in polynomial time on deterministic machines.
3. **NP-Hard**: Problems as hard as the hardest in NP, not necessarily solvable or verifiable in polynomial time.

Key point: All NP problems can be reduced to any NP-complete problem. If one NP-complete problem is shown to be in P, it implies P=NP.

Another key point: Not all NP-hard problems are in NP. There are indeed peculiar problems within NP-hard, such as the halting problem, but those are topics for another discussion.

Part 2: Mind-Blowing Starts Here

Proving a Negative is (Not) Always Impossible

The common belief is that proving a negative is impossible — for example, proving the non-existence of a purple squirrel with two tails. This belief stems from the fact that it's impractical to search everywhere to verify such a claim. This concept underpins legal principles like "innocent until proven guilty," where the burden of proof is on the accuser to demonstrate a positive (i.e., the occurrence of a crime).

However, this notion changes in a closed domain. Take the revised statement: "There is no purple squirrel with two tails **in this room**." In this confined space, it's feasible to search thoroughly in a resonable time (foreshadowing) and confirm the absence of such a squirrel, thereby proving a negative.

Its not purple its magenta

Think Meta

To tackle the P=NP question, a straightforward approach is to find an NP-complete problem that can be reduced to a P problem, thereby proving P=NP. But what about proving the opposite, that P ≠ NP? This would involve demonstrating that no problem in P can be reduced to an NP-complete problem.

However, there's an inherent challenge: the sheer number of potential problems in P makes it virtually impossible to check each one individually within a reasonable timeframe.

Here's where the paradox kicks in: if P=NP were true, then theoretically, we could devise an efficient method to go through all possible P problems and determine whether they can be reduced to an NP problem. In other words, having a solution to P=NP would actually make it feasible to construct a proof for P≠NP in a tractable amount of time.

This circularity underscores the complexity and intrigue of the P=NP problem. It's a bit like needing the answer to solve the problem and needing to solve the problem to get the answer.

Gödel's Incompleteness Theorem

Gödel's Incompleteness Theorem is a landmark in mathematical logic. It posits that in any sufficiently complex mathematical system, there are statements that are true but cannot be proven within the system. This theorem challenges the notion of absolute completeness in mathematics, suggesting the existence of truths beyond formal proof.

Relating this to our discussion on the P vs NP problem, it brings up an ironic possibility. Imagine if the only way to prove that P is not in NP (P≠NP) is if P were actually in NP, because that would allow us to generate a proof in a computationally feasible way. In essence, the ability to prove P≠NP might hinge on the very condition (P=NP) that we're trying to disprove. This echoes the kind of paradoxical scenarios Gödel's theorem implies — that certain truths might exist beyond our current capability to formally establish them.

While this post won't venture into proofs of such complexity, it's intriguing to consider how Gödel's theorem could intersect with one of the biggest unsolved questions in computer science.

Is This a Proof by Contradiction?

What we're dealing with in the P vs NP dilemma isn't exactly a traditional proof by contradiction. In such a proof, you'd start by assuming the opposite of what you want to prove and then show that this assumption leads to an inconsistency or contradiction. However, in the case of P vs NP, the twist is different.

While trying to prove that P is not equal to NP (P≠NP), if we were to enumerate all possible problems in P, we might, by some extraordinary chance, discover a problem that does reduce to an NP-complete problem. This would inadvertently prove that P equals NP (P=NP)!

This scenario makes the P vs NP problem resemble more of a *conspiracy theory* rather than a conventional proof. Conspiracy theories often present plausible scenarios but lack definitive verification due to pervasive obstructive forces. In the P vs NP context, it's like a universal conspiracy theory — the only way to conclusively prove P≠NP false is if it were actually true.

Thus, the cruel irony of the P-NP problem lies in its conditional nature: the possibility of proving it false hinges on the truth of the very statement we're trying to disprove.

Cruel squirrel's thesis

What do you think?

0 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments Login ▾

Start the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS

Name

♡ • Share Best Newest Oldest

All articles

The Photonic Transformer Hypothesis: Rethinking Photosynthesis and Methane Consumption in Plants

19 Nov 2025

The Scalable Quasi-Perpetual Photonic Machine
17 Nov 2025

How to Make a Ball Orbit Itself
08 Nov 2025

The Boron-Nitrogen Catastrophe: A Critical Gap in Beta Decay Verification
28 Oct 2025

X-Ray Data Pipeline: Ultra-High-Speed Communication Through Limestone Tubes
26 Oct 2025

The Universal Approximation Theorem Is Right. You're Using It Wrong.
19 Oct 2025

The Power Law Illusion: A Measurement Artifact Hypothesis
11 Oct 2025

Fractional Gamma Function via Fractional Derivatives
18 Sep 2025

On the cruel irony of the P-NP problem
18 Dec 2023

The Journey from India to Germany: A Guide for IT Professionals
20 Jun 2023

How does AutoGPT work under the hood?
03 May 2023

Unit Test Recorder - Automatically generate unit tests as you use your application

25 Jun 2020

[How to migrate from vanilla Kubernetes to Istio service mesh?]
14 Oct 2019