A Hitchhiker's Guide to P and NP

Tianchen Liu

You start to study *Computer Science* in UC Berkeley. You want to do something other people call "s-wee" or "em-ell" or whatever those are. You finished your lower-div requirements. You ask your friends what's a good upper-div to take first. Your friends recommend CS 170 and say it preps you for the coding interviews. You enroll in CS 170. Divide and Conquer, Graph algorithms, Dynamic Programming... Not bad, you think, this class really helps me in doing LeetCode.

And then the unthinkable happened.

This class starts to talk about something called "Complexity Theory"[3] and "Randomized Algorithms"[4][7]. No recruiters will ever ask me this! You say angrily while watching lecture. And that's the best part about CS 170: lure students in with the promise of making them better at coding interviews, but we make them better *Computer Scientists* instead. After all, that's the name of the major you're in, right?

With that being said, this note is used as an intuitive guide for the most basic parts of complexity theory - **P** and **NP**. Therefore this note will **not** be rigorous nor formal. I aim to strike a balance: I will try to make this note as easy to read as possible with the condition that that you can do homework problems and exam problems if you just comprehend this note entirely.

But there's so much fun and beauty in complexity theory! I put materials that are out-of-scope but super cool in grey boxes, and I encourage you to read them in your spare time. Specifically, I talk about how there's three different names for a same concept - coming from three perspectives. Then we ruminate on what would happen if there's a polynomial time decider for an **NP**-Complete problem.

1 Motivation

After studying so many efficient algorithms for problems in this class, a question arises naturally: are there problems without efficient algorithms? In other words, are there *intractable* problems? In order to answer this question we first need a way to rigorously define the *difficulty of a problem*, and then we can analyze what it means to be in different "difficulty levels". But how are we even going to start talking about difficulities of problems, when there's so many types of problems to start with? Can we start with the easiest type of problems - the one where we just require a simple yes/no as an answer?

2 Decision Problems

Definition 1. A *decision problem* is a problem that can be posed as a yes–no question of the input values.

Example. Primality testing is a decision problem: decide if the input is a prime number or not.

Given the boolean nature of decision problems, we can think of decision problems in the following way: suppose there's a set of strings that contains all the string encodings of *accepting instances* of a decision problem - input instances of a decision problem whose answer is "yes" - and nothing else. In the case of primality testing, this would just be the set of all prime numbers. We call a set of strings a *language*. A decision problem is equivalent as querying the input's membership in the corresponding language. Therefore in complexity theory, **P** and **NP** can also be described as complexity classes of *languages*.
Example. Deciding if an input is prime is equivalent as testing the input's membership in the language of all prime numbers.

It may seem like decision problems are way easier than other type of problems, but this is true only to a certain extent. Almost all[2] search problems can be reduced to their decision counter parts in polynomial time (Can you think of some examples?). Therefore, when we talk about complexity classes of problems, we **only consider decision problems**.

3 P and NP

Now we are ready to talk about the two complexity classes.

Definition 2. **P** is the class of decision problems that can be **decided** (solved) in polynomial time.

Example. A decision problem in **P**: Does the input directed graph contain a cycle?

Very informal definition. Intuitively, **NP** is the class of decision problems that can be **verified** in polynomial time.

Hold on a bit, you may ask, what is there to verify? We're talking about decision problems, so what does it mean to "verify" a yes/no? If I can "verify" that this input instance would output "yes", isn't that equivalent as solving the problem?

You are absolutely right: verifying the boolean output indeed makes no sense at all. Instead we're verifying something that could *testify* the fact that the input instance is indeed a accepting instance. What would you call something that could testify the truth and nothing but the truth?

Definition 3. **NP** is the class of decision problems with a polynomial time verifier $V(x, w)$, such that

- If x is an accepting instance, then there exists a string w such that $V(x, w)$ will output 1 in $O(\mathsf{poly}(|x|))$ time, where $|x|$ denotes the length of the string encoding of the input instance. This w is called the *witness*.
- If x is a rejecting instance, then for all w, $V(x, w)$ would output 0 in $O(\mathsf{poly}(|x|))$ time.

Obviously the witness needs to be at most polynomial-sized otherwise the verifier won't even have time to read the witness in its entirety.

Remark. In a lot of cases, and almost all cases in CS 170, **this witness is simply a *candidate solution*** for the search version of the problem.

Example. The travelling salesman decision problem in **NP**: Given $G = (V, E)$ and k, does G have a path with length at most k that visits all vertices in V?

Proof. Given a candidate solution, which in this case is a path, the verifier can follow down this path and record its length and the set of vertices it has visited. The verifier returns 1 if and only if the length is at most k and the size of the set is $|V|$ when it has reached the end of the path. This verifier runs in $O(|V|+|E|)$ time and is hence a polynomial time algorithm. □

4 Three different terms for *witness*, and consequences if P = NP

In this section, we explore three different perspectives of the complexity class **NP** through the three different terms for the same concept: witness/certificate/proof. This entire section is out of scope.

There are different names for the same thing.

Ben Gibbard

4.1 Certificate

Remember from the earlier box that in complexity theory, **P** and **NP** can also be described as complexity classes of *languages* (sets of strings). Here we give the corresponding definition **NP**.

Definition 4. A language L is in **NP** if and only if there exists a verifier V such that

$$x \in L \iff \exists w : V(x, w) \text{ outputs 1 in } O(\mathsf{poly}(|x|)) \text{ time.}$$

And hence we have an even cleaner definition (without ever touching Turing Machines).

Definition 5. **NP** is the class of languages that could be expressed in this form

$$\{x \mid \exists w : V(x, w) = 1\}$$

where V runs in $O(\mathsf{poly}(|x|))$ time.

In this definition, w verifies the membership of x in the language L. What do you call something that verifies someone's membership? A *certificate*. In complexity theory, the terms *witness* and *certificate* are used interchangeably, along with...

4.2 Proof

Let's start with this question: what is a mathematical theorem?

Well, it's a statement that is guaranteed to be true - and we can verify the correctness of the theorem by verifying a *proof* of the theorem. Specifically,

- True theorems must have a proof that could be verified efficiently;
- False theorems must not have any proofs - you can feed a proof verifier any false proofs and the verifier would reject.

Does this definition sounds familiar?

And this gives us a new perspective of **NP**: **NP** *captures all mathematical theorems with reasonable sized (polynomial sized) proofs.*

This definition of **NP** opens many new doors in complexity theory: what if we allow the verifier to talk back to the prover, instead of just reading a proof? What if the verifier is not deterministic (has randomness)? What if the verifier can randomly read bits of the proof? These are called *Interactive Proofs* and *Probabilistic Checkable Proofs*[5].

4.3 A dire consequence if there's a constructive proof of P = NP

I would like to end this note by discussing about one particular consequence that will happen to humankind if there's a constructive proof for **P** = **NP** (*i.e.* an algorithm that takes anything in **NP** and turns it to a thing in **P**; for example, a polynomial time algorithm for an **NP**-Complete problem[6]).

Recall the third perspective of **NP** as the class of mathematical theorems. With the algorithm, you can essentially take any statements [1], put it in the algorithm, and obtain - within polynomial time - an equivalent statement that can be decided in polynomial time. We then decide the latter statement in polynomial time.

Take a step back and breathe. Think about what we just did. We just gave a method to decide any mathematical statements in polynomial time.

What does this mean? In the words of Scott Aaronson,

> Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett.[1]

Forget the cryptographic protocols crumbling to the ground, forget cybersecurity vanishing, forget internet privacy disappearing, forget blockchains turning into dust - this, in my opinion, is the biggest consequence:

Humans, literally, have nothing left to prove anymore. This is it. All logical truth will be found.

We can roll the credits now.

[1]Specifically, statements with polynomial-sized proofs. But really, if there exists a proof of exponential size for a statement, it really doesn't matter to humans because there's no way we can verify it anyways - if the statement has 128 bits, how are you going to verify a 2^{128}-bit proof?

References

[1] Scott Aaronson. *Reasons to believe.* URL: https://scottaaronson.blog/?p=122.

[2] Mihir Bellare and Shafi Goldwasser. "The Complexity of Decision Versus Search". In: *SIAM J. Comput.* 23 (1994), pp. 97–119.

[3] *CS 278 Computational Complexity Theory.* URL: https://www.avishaytal.org/cs-278-complexity-theory.

[4] *CS271 Randomness & Computation: Spring 2020.* URL: https://people.eecs.berkeley.edu/~sinclair/cs271/s20.html.

[5] *CS294: Foundations of Probabilistic Proofs (F2020).* URL: https://people.eecs.berkeley.edu/~alexch/classes/CS294-F2020.html.

[6] Tianchen Liu. *Reductions / NP Completeness Review Session Slides.* URL: https://tc-liu.github.io/files/cs170/Reductions.pdf.

[7] *Sketching Algorithms.* URL: https://www.sketchingbigdata.org/.