P vs NP

Alyssa Lytle

Fall 2025

1 P

Recall:

Previously we defined the complexity class P:

Definition: P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k \text{TIME}(n^k).$$

[Sip96]

Also recall the definition of a time complexity class:

Definition: Time Complexity Class

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the *time complexity class*, $\text{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing Machine. [Sip96]

We also talked about the basic steps of showing a problem is in P:

1. Express our problem as a language A
2. Present a Turing machine computable algorithm M that decides A
3. Argue the runtime of M

Let's do one more example of a problem that we can show is in P, and then we will talk about the other, more challenging problems!

Example: Relatively Prime Numbers

Say our problem is: I want to know if two integers x and y are relatively prime.
We can define our language as $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$.
An now we can show $RELPRIME \in P$ by presenting a Turing machine computable algorithm M that decides A and arguing that the runtime of M is polynomial.
To come up with our algorithm, let's recall what it means to be relatively prime. This means that 1 is the largest number that divides x and y. In other words, $\text{GCD}(x, y) = 1$.
We know that we can find the GCD of two numbers using the Euclidean algorithm, so let's use that information to help us out!
First, let's recall the euclidean algorithm.
(Note: this algorithm assumes $x \geq y$.)

1. Divide x by y If the remainder r is zero, y is your GCD. (Note that $r = x \bmod y$.)
2. If the remainder r is not zero. Replace x with y and y with r.
3. Repeat steps 1 and 2 until the remainder is zero. The last non-zero remainder is your GCD.

So, if this algorithm returns 1, then we know x and y are relatively prime.
Let's express this algorithm in a more computation-friendly way. (This is assuming we have a polynomial-time way to perfom the mod operation.)
While the remainder variable r is helpful in understanding the algorithm, we don't actually need it for our implementation. In other words, we can just think about it as:

1. If $x \bmod y = 0$, then y is your GCD.
2. Otherwise, $x = y$ and $y = x \bmod y$.
3. Repeat steps 1 and 2 until remainder ($x \bmod y$) is zero. The last non-zero remainder is your GCD.

Since $y = x \bmod y$, this means that you want to exit when $y = 0$. Additionally, since you're setting x to equal the previous value for y, that means x will have the value of the last non-zero remainder when the loop exits.
The other nuance to consider is when we're actually computing this, we don't want to overwrite the previous value for y when we compute $y = x \bmod y$, so we let $x = x \bmod y$ and $y = y$, then *swap* the values.
So we can write our algorithm as:

- Repeat until $y = 0$:
- Assign $x \leftarrow x \bmod y$
- Swap x and y
- Output x

Again, what we're trying to solve is whether or not x and y are relatively prime. So we want to check if the output is 1.
Now we can use this to build our machine:

$R =$ "On input $\langle x, y\rangle$,
1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$
3. Swap x and y
4. If x is 1, *accept*. Otherwise, *reject*."

2 NP

So now, let's talk about the *other* problems... the ones we are potentially unable to show are in P. Can we still say something about them?

Let's talk about the "solvability" of a problem in a different way. Maybe we can't produce an algorithm that solves it in polynomial time, but perhaps we can produce an algorithm that *verifies* it in polynomial time.

In other words, maybe we don't have an algorithm that produces a solution, but we have one that checks if a solution is valid.

Definition: Verifier

A *verifier* for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

A *polynomial time verifier* runs in polynomial time in the length of w.
A language A is *polynomially verifiable* if it has a polynomial time verifier.

Example: PATH

Recall

$$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

We showed that we could build a machine that decides $PATH$ in polynomial time.
If we wanted to build a machine V that *verifies* $PATH$, you could say

$$PATH = \{\langle G, s, t\rangle \mid V \text{ accepts } \langle G, s, t, e\rangle, \text{ where } e \text{ is a path from } s \text{ to } t \text{ over } G\ \}$$

So V should be able to take graph G, nodes s and t and a path/set of edges e and verify that it is e is indeed a set of edges from G that forms a directed path from s to t.

Now that we have this definition of a polynomial time verifier, we can define NP.

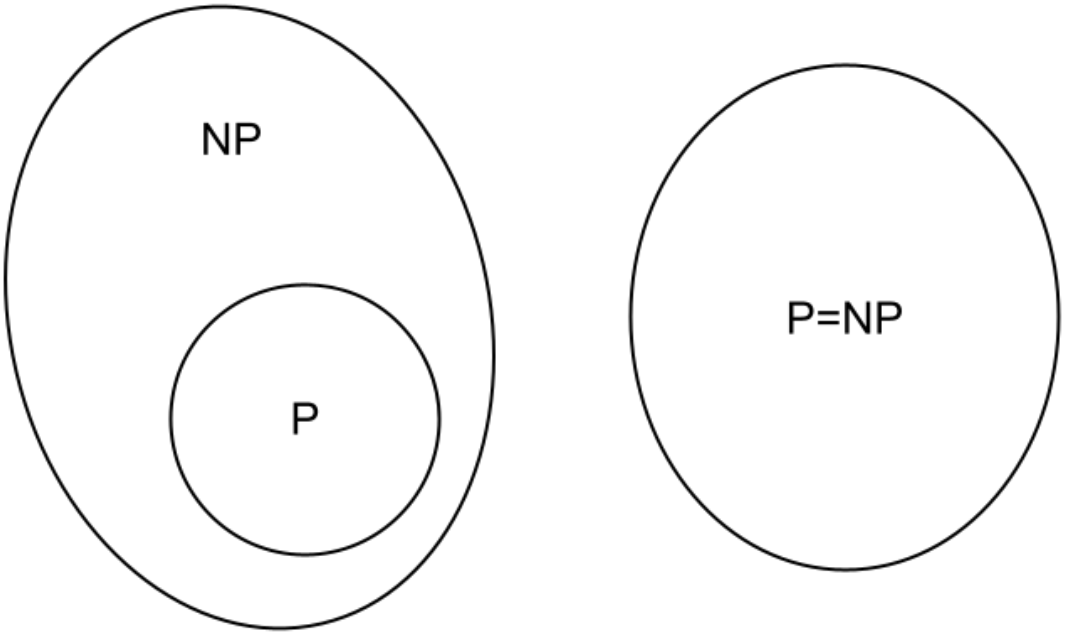
Definition: NP

NP is the class of languages that have polynomial time verifiers.

Note that we have discussed how $PATH \in P$ and $PATH \in$ NP. In fact, it is true that P $\subseteq$ NP.

In other words, every problem that can be decided in polynomial time can also be verified in polynomial time.

What about the reverse? Can every problem that can be verified in polynomial time also be decided in polynomial time? This is an open question. If this were true, then it would be the case that NP $\subseteq$ P, and therefore P = NP.



This figure illustrates the two possibilities this creates. Either P is strictly a subset of NP (meaning there are problems in NP that cannot be decided in polynomial time) or they are the same set.

2.1 Nondeterministic Polynomial Time + Turing Machines

You may be wondering where the term "NP" comes from, and it's shorthand for *Nondeterministic polynomial time*. In fact, any NP problem *could* be decided in polynomial time on a nondeterministic Turing machine.

Definition: NP

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Definition: Nondeterministic Turing Machine

A nondeterministic Turing Machine is defined the same as a deterministic one, except the transition function is defined as a probability mapping.

$$\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$$

Another definition that may be useful to know is that of NTIME:

Definition: NTIME

$$\mathrm{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an O(t(n)) time nondeterministic Turing machine}$$

This allows us to define NP as

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$$

This helps us with the question: how do I argue that a problem is in NP? For this, you could either design a nondeterministic turing machine that decides it in polynomial time *or* you can design a deterministic turing machine that verifies it in polynomial time.

Example: CLIQUE

A clique in an undirected graph is a subgraph where every node is connected by an edge. A k-clique is a clique that contains k nodes.
Let $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
Show $CLIQUE$ is in NP.
(Solution on page 296 of book [Sip96].)

Example: SUBSETSUM

Say we have a collection of numbers $S = \{x_1, x_2, \ldots, x_k\}$ and we want to know if a subset of the elements of S can be summed to equal a specific value t. For example, can I sum a subset of $\{1, 2, 3, 4\}$ to get 7?
This is how we define our problem:

$$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, x_2, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq S, \sum y_i = t\}$$

Show $SUBSETSUM$ is in NP.
(Solution on page 297 of book [Sip96].)

References

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.