P, NP, and all that Jazz![1]

The goal of this supplement is to familiarize you (a bit) with what $\mathbf{P}$, $\mathbf{NP}$, $\mathbf{NP}$-hard, and $\mathbf{NP}$-complete really mean. This is more of an *amuse-bouche* to the course COSC 39 which really gets to the heart of the question: "What is Computation?"

Let us start with a definition of *decision problems*.

Definition 1. Informally, any problem Π is a decision problem if its solutions are either YES or NO. Formally, a problem Π is a decision problem if all instances $\mathcal{I} \in \Pi$ can be *partitioned* into two classes YES-Instances and NO-instances, and given an instance $\mathcal{I}$, the objective is to figure out which class it lies in.

Not all problems are decision problems. There are *optimization* problems which ask us to maximize or minimize things. For example, find the largest independent set in a graph is an optimization problem. Most optimization problems, however, have *decision* versions. For example, the problem IS in the main notes is a *decision* version of the optimization problem.

Next, let me formally define what *polynomial time reductions* are. We essentially saw this in the main lecture, but this puts a bit of formalism in.

Definition 2. A decision problem Π_A **polynomial time reduces** to a decision problem Π_B if there exists an algorithm $\mathcal{A}$ such that

- $\mathcal{A}$ takes input an *instance* $\mathcal{I}$ of Π_A.
- $\mathcal{A}$ outputs an *instance* $\mathcal{J}$ of Π_B.
- There exists a polynomial $p(\cdot)$ such that the running time of $\mathcal{A}$ on $\mathcal{I} \in \Pi_A$ is at most $p(|\mathcal{I}|)$ where $|\mathcal{I}|$ is the size of the instance. This necessarily implies $|\mathcal{J}| \leq p(|\mathcal{I}|)$.
- $\mathcal{I}$ is a YES-instance of Π_A **if and only if** $\mathcal{J}$ is a YES-instance of Π_B.

In that case, we say $\Pi_A \preceq_{\text{poly}} \Pi_B$.

In the main lecture, we basically argued the following two lemmas. It is a good exercise to formally prove them.

Lemma 1. If $\Pi_A \preceq_{\text{poly}} \Pi_B$, and there exists a polynomial time algorithm to solve Π_B, then there is a polynomial time algorithm to solve Π_A.

Lemma 2. If $\Pi_A \preceq_{\text{poly}} \Pi_B$ and $\Pi_B \preceq_{\text{poly}} \Pi_C$, then $\Pi_A \preceq_{\text{poly}} \Pi_C$.

Now, we are ready to explain what $\mathbf{P}$ and $\mathbf{NP}$ are. They are just classes (subsets) of decision problems. The definition of $\mathbf{P}$ is straightforward.

Definition 3. $\mathbf{P}$ is the class of decision problems which can be solved in polynomial time. More precisely, $\Pi \in \mathbf{P}$ if there exists an algorithm $\mathcal{A}$ and a polynomial $p()$, such that

- For every YES-instance $\mathcal{I} \in \Pi$, $\mathcal{A}(\mathcal{I})$ runs in time $p(|\mathcal{I}|)$ and returns YES.
- For every NO-instance $\mathcal{I} \in \Pi$, $\mathcal{A}(\mathcal{I})$ runs in time $p(|\mathcal{I}|)$ and returns NO.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

The definition of **NP** is more interesting. **NP** is the class of decision problems which can be *verified* in polynomial time. Let us elaborate in the definition below.

Definition 4. A problem Π lies in **NP** if there exists an algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$, such that

- For every YES-instance $\mathcal{I} \in \Pi$, there *exists* a "solution" $\mathcal{S}$ with $|\mathcal{S}| \leq p(|\mathcal{I}|)$ such that $\mathcal{A}(\mathcal{I}, \mathcal{S})$ runs in $\leq p(|\mathcal{I}| + |\mathcal{S}|)$ time and returns YES.
- For every NO-instance $\mathcal{I} \in \Pi$, and for *every* purported "solution" $\mathcal{S}$ of size $|\mathcal{S}| \leq p(|\mathcal{I}|)$, $\mathcal{A}(\mathcal{I}, \mathcal{S})$ runs in $\leq p(|\mathcal{I}| + |\mathcal{S}|)$ time and returns NO.

The above definition is deep, so let's first look at an example and then we see other ways of interpreting it.

Claim 1. SAT lies in the class **NP**.

Proof. We need to come up with an algorithm $\mathcal{A}$ which reads an instance of SAT, that is a formula ϕ, and a "solution" $\mathcal{S}$ and says YES or NO. Here is one observation: we can assume that the solution $\mathcal{S}$ be any given "format"; our algorithm $\mathcal{A}$ will say NO immediately if the format is not satisfied.

So here is what our "verifier" $\mathcal{A}$ expects: $\mathcal{I}$ should be a formula ϕ, and the solution $\mathcal{S}$ should be an assignment of truth values to the variables of ϕ. Anything else, $\mathcal{A}$ will reject outright (that is, says NO). Furthermore, if the format is correct, the algorithm just verifies if the truth value expressed in $\mathcal{S}$ satisfies the formula ϕ. If so, it says YES.

Now we see that the conditions of the definition are trivially true. If ϕ is indeed satisfiable, then $\mathcal{S}$ be the assignment which satisfies ϕ. We can't stress this enough: the algorithm $\mathcal{A}$ doesn't need to *find* $\mathcal{S}$; it just needs to *verify* it. A much easier proposition. In any case, if ϕ is satisfiable then the $\mathcal{S}$ described would make the algorithm $\mathcal{A}$ accept. All this takes linear time (so the polynomial is just a linear function).

On the other hand if the formula is not satisfiable, then if no assignment makes all clauses true. So if $\mathcal{S}$ is not rejected outright, it will be rejected once all the clauses are verified by $\mathcal{A}$. That is, if $\mathcal{I}$ is a NO-instance, no matter what $\mathcal{S}$ you feed $\mathcal{A}$ the algorithm will say NO. □

Here is another interpretation of **NP**. Imagine Luna wants to solve SAT but she has tried hard and failed. So she seeks help from an all powerful *prover* named Albus. More precisely, she seeks advice from Albus regarding an instance ϕ of SAT to decide whether it is satisfiable or not. If ϕ is indeed satisfiable, then Albus can find the solution (he is all powerful, remember) and write it down for Luna. Luna, however, is a bit paranoid and she is worried that Tom may be impersonating as Albus and trying to fool her. So she asks Albus to write down the solution in a precise format (that is, the truth assignment) and if there is any discrepancy, she just rejects. And if it is in the format, then she just checks if it indeed satisfies ϕ.

More generally, **NP** is the set of decision problems for which Luna can ask advice from Albus in a *precise format* such that for YES-instances, Albus can write down a solution which Luna can *quickly* verify and convince herself that the instance is YES, while for NO-instances no solution from Albus, or from any Tom impersonating as Albus, can ever fool Luna into believing that the instance is a YES-instance.

Remark: *"What's that N in NP?"* **NP** *does* **not** *stand for "not polynomial". Rather, it stands for "non-deterministic polynomial". If our algorithm (or Luna) is allowed to make divine guesses (advice from Albus), then for* YES*-instances there is a guess which leads to the correct resolution, while for* NO*-instance all guesses lead to rejection. The concept of* non-determinism *is a deep one and there are some computation models where non-determinism, in fact, doesn't give any extra power. Whether it does for polynomial time algorithms, is, quite literally,* **a million dollar question**.

It is not hard to show $\mathbf{P} \subseteq \mathbf{NP}$. Indeed, if you have understood so far, then you can probably prove it yourself. Using the metaphor above: if Luna can herself solve the problem, then she can just ignore any advice from Albus (or Tom) and convince herself. In particular, the "empty" solution would do.

Question 1.

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

The famous $\mathbf{P}$ vs $\mathbf{NP}$ question asks whether $\mathbf{NP} \subseteq \mathbf{P}$ or not? Is there any problem in $\mathbf{NP}$ that is not in $\mathbf{P}$? Is there any problem for which solutions can be verified efficiently but can't be found efficiently? We don't know.

$\mathbf{NP}$-hardness and $\mathbf{NP}$-completeness

Although we don't know whether $\mathbf{P} = \mathbf{NP}$ or not, we do know the *hardest* problems in $\mathbf{NP}$. This is one of the deepest facts in computer science and is due to Stephen Cook and, independently, Leonid Levin.

Theorem 1 (Cook-Levin Theorem)**.** Let Π be any problem in $\mathbf{NP}$. Then, $\Pi \preceq_{\text{poly}} \text{SAT}$.

We already saw SAT was in the class $\mathbf{NP}$. The above theorem says f*any* problem in $\mathbf{NP}$ is "easier" than SAT. Thus, if we have a polynomial time algorithm for SAT, then there is a polynomial time algorithm for *all* problems in $\mathbf{NP}$. Amazing, isn't it? To prove $\mathbf{P} = \mathbf{NP}$, that is to argue about two sets, we just need to argue about one element in the set $\mathbf{NP}$. On the other hand, if we can prove SAT has no polynomial time algorithms, then we prove $\mathbf{P} \neq \mathbf{NP}$; we have found one element in $\mathbf{NP} \setminus \mathbf{P}$. The Conjecture from the last lecture asserts that $\mathbf{P} \neq \mathbf{NP}$.

But we know there are harder problems than SAT. IS is one, for example. This motivates the following definitions.

Definition 5. A problem Π is $\mathbf{NP}$-hard if $\Pi' \preceq_{\text{poly}} \Pi$ for any $\Pi' \in \mathbf{NP}$.

That is, Π is $\mathbf{NP}$-hard if its harder than all problems in $\mathbf{NP}$. Theorem 1 implies that (a) SAT is $\mathbf{NP}$-hard, and (b) to show Π is $\mathbf{NP}$-hard, all we need to show is $\text{SAT} \preceq_{\text{poly}} \Pi$. Thus, IS is $\mathbf{NP}$-hard as well.

Definition 6. A problem Π is $\mathbf{NP}$-complete if it is (a) in $\mathbf{NP}$ and (b) is $\mathbf{NP}$-hard.

$\mathbf{NP}$-compelete problems form an *equivalence* class. By definition, for any two $\mathbf{NP}$-complete problems Π, Π', we have $\Pi \preceq_{\text{poly}} \Pi'$ and $\Pi' \preceq_{\text{poly}} \Pi$. It is not too hard to show that IS is indeed $\in \mathbf{NP}$ and so we get IS is indeed $\mathbf{NP}$-complete. Which means if you do find a polynomial time algorithm for IS, then you do find a polynomial time algorithm for ***all*** problems in $\mathbf{NP}$.

There are hundreds of problems which are $\mathbf{NP}$-complete. Indeed, most problems you will meet out there are $\mathbf{NP}$-complete. See here and links within for exploring more.