# *Lecture 6: Non-deterministic Polynomial Time*

*Anup Rao*

*April 15, 2021*

In the last class, we introduced the concept of *complexity classes*. We saw the classes $P, L, E, EXP$ and $PSPACE$. These classes were obtained by considering functions that can be computed with limited time or limited space. Today, we explore a different kind of class, the class $NP$.

Recall: $L \subseteq P \subseteq PSPACE \subseteq EXP$.

$NP$ is interesting chiefly because many problems that we would like to solve efficiently with a computer, but cannot solve, belong to $NP$. The list of such problems includes essentially all problems solved today with machine learning, and many other practically important problems. Before giving the definition of $NP$, let us see some examples of problems in $NP$.

*Independent Set* Given a graph $G$ and a number $k$, does the graph have an independent set of size $k$? Let $ISet(G, k) = 1$ if the graph has an independent set, and 0 otherwise.

Recall that an independent set is a set of nodes that does not contain any edges.

*Subset sum* : Given a list of numbers $a_1, \ldots, a_\ell, t$, is there some subset of the numbers $a_1, \ldots, a_\ell$ that sums to $t$? Let $SubSum(a_1, \ldots, a_\ell, t) = 1$ if there is such a subset, and 0 otherwise.

*Composite numbers* : Given a number $N$, decide if it is composite or not. Let $Comp(N) = 1$ if $N$ is composite, and 0 otherwise.

*Matching* : Given a graph $G$ and a number $k$, are there $k$ disjoint edges in the graph? Let $Match(G, k)$ be 1 if there are $k$ such edges, and 0 otherwise.

All of these problems have something in common: although it may be hard to efficiently compute the functions they define, it is very easy to *check* a solution if one is given to us! For example, if $ISet(G, k) = 1$, then there is a an independent set $S$ of size $k$, and given $G, S, k$, one can check that $S$ is an independent set of size $k$ in polynomial time. Similarly, if $SubSum(a_1, \ldots, a_\ell, t) = 1$, then there is a subset of the numbers $S \subseteq \{a_1, \ldots, a_\ell\}$, that if given as input can be verified to have the sum $t$.

$NP$ is the class of all functions $f$ that have the above property, where if $f(x) = 1$, then this can be checked efficiently by an efficient verifier:

**Definition 1.** $f : \{0,1\}^* \to \{0,1\}$ *is in* **NP** *if there exists a polynomial* $p$

*and a polynomial time machine $V$ such that for every $x \in \{0,1\}^*$,*

$$f(x) = 1 \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)}, V(x,w) = 1$$

$V$ is usually called the *verifier* and $w$ is usually called the witness or certificate or proof. For example, in the independent set problem above, the witness $w$ would correspond to an independent set, and the verifier $V$ would be the program that checks that $w$ is in fact an independent set of size $k$ in the input graph.

The witness $w$ is restricted to being of polynomial length to ensure that the running time of $V$ is actually polynomial in the length of $x$. If we allowed the witness to be arbitrarily long, then $V$ would be allowed to run very long computations on $x$.

Many important combinatorial optimization problems can be cast as problems in **NP**.

## *P, NP and EXP*

**Fact 2.** *$P \subseteq NP \subseteq EXP$.*

To see the first containment, observe that if $f \in P$, there is a polynomial time Turing machine $M$ with $M(x) = f(x)$. But $M$ itself is a verifier for $f$ (with a witness of length 0) proving that $f \in NP$.

For the second containment, if $f \in NP$, then $f$ has a verifier $V(x,w)$. Consider the algorithm that on input $x$ runs over all possible $w$ and checks if $V(x,w) = 1$. If any witness makes $V(x,w) = 1$, the algorithm outputs 1, otherwise it outputs 0. This algorithm computes $f$ and runs in exponential time, so $f \in EXP$.

## *Nondeterministic Machines, and a Hierarchy Theorem*

The original definition of **NP** was by considering Turing machines that are allowed to make non-deterministic choices: namely after each step, the machine is allowed to make a guess about which state to transition to in the next step. The machine computes 1 if there is a single accepting computational path, and 0 otherwise.

We can define $\mathsf{NTIME}(t(n))$ in the same way as $\mathsf{DTIME}(t(n))$, it is the set of functions computable by non-deterministic machines in time $O(t(n))$, and then you can check that $\mathbf{NP} = \bigcup_c \mathsf{NTIME}(n^c)$. Just as for deterministic time, there is a non-deterministic time hierarchy theorem:

**Theorem 3.** *If $r, t$ are time-constructible functions satisfying $r(n+1) = o(t(n))$, then*

$$\mathsf{NTIME}(r(n)) \subsetneq \mathsf{NTIME}(t(n)).$$

## *Polynomial time Reductions*

One of the central questions in complexity theory is whether or not $\mathbf{P} = \mathbf{NP}$. Although we don't know the answer to this question, we can prove a lot about the class **NP**, via the concept of polynomial time reductions:

**Definition 4.** *A function $f$ is polynomial time reducible to a function $g$ if there is a polynomial time computable function $h$ such that $f(x) = g(h(x))$. We write $f \leq_P g$.*

Note that the above definition is not the only one that makes sense. In general it makes sense to allow our reductions to make multiple calls to the problem being reduced to. However, we will be able to prove many of our results using the stronger notion above, so that is what we shall use.

**Definition 5.** *We say $f$ is* **NP**-*hard if $g \leq_P f$ for every $g \in \mathbf{NP}$. We say $f$ is* **NP**-*complete if $f$ is* **NP**-*hard and $f \in \mathbf{NP}$.*

**Theorem 6.** *Here are some easy facts that one can prove about reductions:*

- *If $f \leq_P g$ and $g \leq_P h$, then $f \leq_P h$.*
- *If $f$ is* **NP**-*hard and $f \in \mathbf{P}$, then $P = NP$.*
- *If $f$ is* **NP**-*complete, then $\mathbf{P} = \mathbf{NP}$ if and only if $f \in P$.*

## **NP**-*complete problems*

The above definitions make sense because we do know of examples of **NP**-complete problems.

### *Circuit-Sat*

**Definition 7.** *$CircuitSat : \{0,1\}^* \to \{0,1\}$ is the function that views its input as a circuit $C$ and outputs $1$ iff $\exists x$ such that $C(x) = 1$.*

I have claimed in class that circuits can simulate Turing Machines. Here is what you can actually prove in this regard:

**Theorem 8.** *If a function $f : \{0,1\}^* \to \{0,1\}$ can be computed in time $t(n)$ by a Turing machine, then for every $n$ there is a circuit of size $O(t(n) \log t(n))$ that computes $f$ restricted to the inputs of size $n$.*

Although we did not prove this theorem in class, we sketched how you could find a circuit of size $O(t(n)^2)$ that computes $f$. The idea was to add a layer of gates that maintains the entire state of the Turing machine—contents of all tapes, pointers, and the line of code being executed. Then we add a new layer that computes this configuration after one execution step of the Turing machine, using the earlier configuration as input. A single configuration can be written down using $O(t(n))$ gates since we only need to write down the values of the tapes up to $O(t(n))$ coordinates. The new configuration can be computed from the old one with $O(t(n))$ gates as well. After repeating this $O(t(n))$ times, we obtain the final configuration of the Turing machine, which must include the value of $f(x)$.

**Theorem 9.** *CircuitSat is* **NP***-complete.*

**Proof** It is clear that *CircuitSat* is in **NP**. Next we show that for every $f \in \mathbf{NP}$, $f \leq_P \mathit{CircuitSat}$. Let $V$ be a verifier for $f$. Then to compute $f(x)$, the reduction will build the circuit $C_x(w)$ that computes $V(x, w)$, where here $w$ are the input variables to the circuit and $x$ is the input. Since $f(x) = 1$ if and only if there exists $w$ such that $C_x(w) = 1$, we can determine the value of $f$ by computing $\mathit{CircuitSat}(C_x)$. ■