**DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

The goal of this lecture is to introduce the formal definition of non-deterministic poly time. To do this, we first need to formalize what it means for a Turing machine to run in a certain amount of time, and then formalize non-determinism. Combining these then leads us to the notion of non-deterministic time complexities.

## 1 Time Complexity

The running time of a Turing machine is a function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that the Turing machine can take on an input of size $n$.

This in turn leads to the definition of time complexity classes. We use $\text{Time}(g(n))$ to denote the set of all languages decidable by $O(g(n))$ time Turing machines.

Then $P$, the set of all poly time decidable languages, is given by

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

Note that this definition fits well with Turing machines because operations such as rewinding the tape give extra factors of $n$ (actually, the length of the input, which may be as big as $n^k$) in the running times. Treating such overheads as in the same class allows us to ignore such details when considering algorithms.

We show a language is in $P$ by exhibiting a Turing machine that decides it in $O(n^k)$ time. Examples in the text book include:

1. $\{0^k1^k \mid k \geq 0\} \in \text{Time}(n^2)$.
2. $\text{PATH} \in P$.
3. $\text{RELATIVEPRIME} \in P$.
4. ANY context-free language $L$ is in $P$ (via dynamic programming).

The much more tricky examples of problems in $P$ are PRIME and COMPOSITE. $P$ is an extremely powerful definition because most of the modifications to Turing machines remain closed within the class. This includes:

1. Increasing the alphabet size.
2. Increasing the number of tapes.

3. Running TM $M$ (on some input) inside a universal Turing machine.

The one exception is non-determinism, where the machine itself can choose between multiple outcomes from the same state/tape symbol.

## 2 Non-Determinism

To define $NP$, we need to first introduce the notion of non-determinism.

A non-deterministic operation is simply where $\delta(q, x)$ can give multiple outcomes. Such a Turing machine accepts an input if and only if there is a choice of choices on which the input is accepted.

The running time of such a machine however is defined by the maximum number of steps required to terminate on an input of length $n$.

We use $\text{NTIME}(g(n))$ to denote the set of all languages decidable by $O(g(n))$ time non-deterministic Turing machines. It also gives the definition of $NP$, non-deterministic polynomial time:

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

Examples of problems in NP discussed in the textbook are CLIQUE and HAMILTON-PATH. For each such problem, the following alternate characterization of NP is useful:

**Definition 2.1.** A verifier of a language $A$ is an algorithm $V$ where

$$A = \{ w \mid V \text{ accepts } <w, c> \text{ for some string } c \}.$$

**Theorem 2.2.** $NP$ is precisely the languages with poly-time verifiers. That is, a verifier $V$ that runs in time $O(n^k)$ when $w$ has length $n$.

Note that the poly-time behavior of $V$ also means the length of $c$ must also be poly $n$. This equivalence is proven by:

1. (for showing a language with a poly-time verifier is in $NP$) Having a non-deterministic Turing machine that first writes down the string $c$, and then invokes $V$. This is akin to ‘trying all strings’.

2. (for showing any language in $NP$ has a poly-time verifier) Letting the string $c$ denote the non-deterministic choices of the Turing machine made during its operations, and letting $V$ be the (deterministic) simulation of the non-deterministic Turing machine using the choices given by $c$. 


Note that a direct implication for this is

\[ NP \subseteq \bigcup_k \text{Time}\left(2^k\right). \]

This is proven by enumerating over all secret strings of length \(n^k\), and running the verifier. Note that while the number of such strings is \(\Gamma^{n^k}\), it is still \(2^{O(k)}\) due to \(|\Gamma|\) being a constant. Somewhat surprisingly, this is the best that’s known to date in terms of the relation between \(P\) and \(NP\).

3 Polytime Reducibility

Instead, our justification of the hardness of many problems in \(NP\) is via the the Cook-Levin theorem. It states that there are explicitly given problems that are harder than any problem in \(NP\).

To formalize hardness, we need to give the notion of ‘harder. This is done through reductions.

**Definition 3.1.** Language \(A\) is poly-time reducible to language \(B\), written \(A \leq_P B\) if there is a poly-time function \(f : \Sigma^* \rightarrow \Sigma^*\) such that

\[ w \in A \iff f(w) \in B. \]

The function is called the polynomial time reduction of \(A\) to \(B\).

The immediate implication is:

**Lemma 3.2.** If \(A \leq_P B\), and \(B \in P\), then \(A \in P\).

The hard problem is 3-SAT. This problem takes a set of variables \(x_1,\ldots,x_n\), and defines:

- A clause \(c_j\) is the disjunction (“or”) of three literals, \(l_{j,1} \lor l_{j,2} \lor l_{j,3}\).
- a literal \(l_{j,k}\) is either a variable \(x_i\) or its negation \(\neg x_i\).

The 3-SAT problem asks, does there exist an assignment of either \(T\) or \(F\) to each \(x_i\) so that each clause has at least one literal true.

The Cook-Levin theorem then gives that 3-SAT is \(NP\)-hard.

**Theorem 3.3** (Cook-Levin Theorem). For any problem \(A\) in \(NP\), we have \(A \leq_P 3\text{-SAT}\).

As 3-SAT is in \(NP\), it is also \(NP\)-complete.

We can then see how that SAT is also \(NP\)-complete. Here the main idea is that a clause with two literals can be turned into a clause with three literals via

\[ L_1 \lor L_2 = L_1 \lor L_2 \lor L_3 \]
while a clause with four literals

\[ L_1 \lor L_2 \lor L_3 \lor L_4 \]

can be turned into

\[
L_1 \lor L_2 \lor \neg y \\
y \lor L_3 \lor L_4
\]

This conversion works because \( y \) cannot be \( T \) when both \( L_1 \) and \( L_2 \) are false: it would make \( \neg y \) false, thus making everything in the first clause false.

When we have more variables, we can similarly create new variables that can only be \( T \) if one of the two variables is \( T \).

We will also briefly discuss how the encodings of these SAT instances can be converted, and why such a Turing machine runs in poly time.