#1 Verifiers and NP

Today we will talk about our *second* major super-duper important complexity class.

Definition 1. *A verifier for a language A is a TM V, where*

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

We measure the time of a verifier only in terms of the length of w, so 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.
The string c is called the certificate or proof of membership in A, and c must be polynomial in the length of w.

#2 Problems in NP

Here is an example problem in NP.

$$\textsc{composites} = \{x \mid x = pq, \text{ for integers } p, q, > 1\}$$

(In other words, x is not prime.) A verifier for COMPOSITES would be a TM V that takes in a pair $\langle x, r\rangle$ where r divides x. V should then be able to determine, in time polynomial w.r.t. the length of x, if r is a divisor of x. If so, $x \in$ COMPOSITES.
What does such a V look like? We can surely think of something—perhaps considering all multiples of $r, 2r, 3r, \ldots$ until we reach a multiple that is greater than or equal to x (we should only need to check up to $r\sqrt{x}$, iirc.)

#3 NDTMs and NTIME

I omitted these definitions from Tuesday's lecture.

Definition 2. *Let N be a deciding NDTM. The running time of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.*

I will draw Sipser Figure 7.10 in class for the above.

Next, we can now define what it means to be "computable in polynomial time" on an NDTM.

$$\textsc{ntime}(t) = \{L \mid L \text{ is a language decided by an } \mathcal{O}(t) \text{ time NDTM.}\}$$

Finally, we have the following important result.

Theorem 1.

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

That is, a language is in NP iff it is decided by some NDTM in polynomial time.

Proof. We show how to convert a poly-time verifier to an equivalent poly-time NTM and vice versa. In one direction, let $A \in NP$ and suppose V is its verifier, running in time n^k. Construct N as follows:

$N =$ "On input w of length n :

1. nondeterministically select string c of length at most n^k
2. Run V on input $\langle w, c\rangle$
3. if V accepts, accept; otherwise, reject. "

In the other direction, suppose A is decided by a poly-time NTM and construct verifier V as follows:

$V =$ "On input $\langle w, c\rangle$, where w and c are strings :

1. Simulate N on w, treating each symbol of c as a description of the nondeterministic choice to make at
2. If this branch of N's computation accepts, accept; otherwise, reject."

□

#4 Polynomial-time reducibilitity

We next adapt our notions of computable functions and mapping reductions to incorporate a time component.

Definition 3. *A function $f : \Sigma^* \to \Sigma^*$ is computable in polynomial time if it's a computable function decided by a poly-time TM.*

And next,

Definition 4. *Language A is polynomial time reducible to language B, written $A \leq_p B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ is a mapping reduction from A to B.*

This theorem is important.

Theorem 2. *If $A \leq_p B$ and $B \in P$, then $A \in P$.*

I will do a proof in class if I have time.

#5 NP-Completeness

Here is our big goal:

Definition 5. *A language B is called NP-Complete if it satisfies two conditions:*

1. *$B \in NP$, and*
2. *for all $A \in NP$, we have $A \leq_p B$.*

And here is the big theorem to go with NP-completeness.

Theorem 3. *If B is NP-complete and $B \in P$, then $P = NP$.*

#6 SAT

The first problem to be shown to be NP-complete is called SAT, or the satisfiability problem. This is a very important problem. We will take it slowly.

First consider the grammar of boolean formulae, with start symbol B. Let Σ be an alphabet of character symbols. Let's introduce some shorthand: we write $V \in \Sigma^*$ to mean, for each $\sigma_i \in \Sigma$:

$$V \to \sigma_1 V \mid \sigma_2 V \mid \ldots \mid \sigma_n V \mid \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n$$

Now, define the grammar of boolena formulae as follows.

$$\begin{aligned}
&V \in \Sigma^* \\
&B \to 0 \mid 1 \mid V \mid \neg B \mid B \vee B \mid B \wedge B
\end{aligned}$$

A string generated by B is called a boolean formula. The operations of disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$) are called boolean operations. These operations have the usual semantics.

We call a boolean formula ϕ satisfiable if an assignment of 0s and 1s exist to its variables that makes the formula ϕ evaluate to 1. For example:

$$\phi = (\neg x \wedge y) \vee (x \wedge \neg z)$$

ϕ has a satisfying assignment of $x = 0$, $y = 1$, and $z = 0$.

The satisfiability problem is:

$$\text{SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula } \}$$

Finally, here is our big theorem:

Theorem 4. SAT $\in P$ *iff* $P = NP$.

And we use NP-completeness to show this fact. (This is where we're headed.)