As promised in the previous lecture, we will now prove the Cook-Levin theorem, which is restated below.

**Theorem 1 (Cook-Levin)** 3-sat is \( NP \)-complete.

To prove this theorem, we will reduce \( \text{TM-sat} \) to \( 3\text{-sat} \). However, this is difficult to do directly, so we will introduce an intermediate language, called \( \text{Circuit-sat} \). This requires us to first define the notion of a boolean circuit.

**Definition 2** A boolean circuit is a directed acyclic graph with the following properties:

1. It has \( n \) sources \( \{u_1, \ldots, u_n\} \) and one sink output, with \( d_{\text{in}}(\text{output}) = 1 \).
2. All the vertices that are not sources or sinks are called gates. Each gate is labeled with a symbol from \( \{\land, \lor, \neg\} \).
3. Every gate \( v \) satisfies \( d_{\text{out}}(v) = 1 \).
4. If a gate \( v \) is labeled with \( \land \) or \( \lor \), then \( d_{\text{in}}(v) = 2 \). If \( v \) is labeled with \( \neg \) then \( d_{\text{in}}(v) = 1 \).

A boolean circuit \( c \) with \( n \) sources computes a boolean function in the following natural way. If values \( (b_1, \ldots, b_n) \in \{0, 1\}^n \) are provided, label each edge with \( \{0, 1\} \) so that (1) every edge emerging from the source \( u_i \) is labeled with \( b_i \) and (2) the edge emerging from each gate evaluates the appropriate boolean operator on the labels of the edges entering that gate. Then the label of the edge leading to output is the output of the boolean function; we will denote this \( c(b_1, \ldots, b_n) \). It is left as an exercise to show that this function is well-defined.

**Definition 3** The size of a boolean circuit is the number of gates in that circuit.

**Definition 4** The depth of a boolean circuit is the length of a longest path in that circuit.

Having described boolean circuits, we can now define an associated \( NP \)-complete language, \( \text{Circuit-sat} \).

**Definition 5** The language \( \text{Circuit-sat} \) is defined as follows:

\[
\text{Circuit-sat} = \{ \varphi : \varphi \text{ is a satisfiable boolean circuit} \}.
\]

**Proposition 6** \( \text{Circuit-sat} \in NP \).

**Proof** If \( \varphi \) is a satisfiable boolean circuit, any satisfying assignment serves as a witness to this fact. The satisfying assignment has size equal to the number of inputs of \( \varphi \), which is polynomial in the size of \( \varphi \) (since a boolean circuit with \( n \) inputs has at least \( n - 1 \) gates).

The following result showcases the importance of boolean circuits, and will be used in the proof of the Cook-Levin theorem.

**Theorem 7 (Universality of circuits)** For every boolean function \( f : \{0, 1\}^\ell \rightarrow \{0, 1\} \), there is an \( \ell\text{-CNF} \) circuit \( \varphi \) of size \( O(\ell \cdot 2^\ell) \) such that \( \varphi(u) = f(u) \) for all \( u \in \{0, 1\}^\ell \).
Proof. Consider \(S = \{u : f(u) = 0\}\). For each \(u \in S\) write the clause \(C_u\) such that \(C_u(u) = 0\) and \(C_u(v) = 1\) for every \(v \neq u\). Let \(\varphi = \bigwedge_{u \in f^{-1}(0)} C_u\). Then \(\varphi\) computes \(f\) and is of the right size. ■

We are now ready to begin the proof of the Cook-Levin theorem. As noted above, there are two main lemmas.

**Lemma 8** \(\text{TM-SAT} \leq_P \text{CIRCUIT-SAT} \) (and therefore \(\text{CIRCUIT-SAT}\) is \(\text{NP-complete}\)).

**Proof.** We first show the following fact: Given any Turing machine \(M\) that runs in time \(T(n)\) (where \(n\) is the size of the input), there exists an \(O(T(n)^2)\)-size family of circuits \(\{c_n\}_{n \in \mathbb{N}}\) such that for all \(n \in \mathbb{N}\) and \(x \in \{0, 1\}^n\), \(c_n(x) = M(x)\). Furthermore, if \(T(n)\) is polynomial in \(n\), then \(\{c_n\}_{n \in \mathbb{N}}\) can be constructed in polynomial time.

First, simulate \(M\) by an oblivious Turing machine \(M'\) that runs in \(O(T(n)^2)\) time (this is proved possible in Arora and Barak). On input \(x\) to \(M'\), suppose that \(M'\) halts in time \(t\). Then let \(z_t, \ldots, z_i\) be “local snapshots” of the computation of \(M'(x);\) that is, \(z_i = (s_i, h_i)\), where \(s_i\) is the state of \(M'\) at time \(i\) and \(h_i\) is the symbol read by the head at time \(i\). Represent \(z_i\) by a fixed-length binary string.

Now, because \(M'\) is oblivious, \(z_i\) is a boolean function of \(z_{i-1}\) and \(z_j\), where \(j\) is the most recent time before \(i\) that \(M'\) visited the current location on the tape. In particular, note that the number of bits required to determine \(z_i\) is a fixed number that depends only on \(M'\) (not \(x\)). By the universality of circuits, this means that \(z_i\) can be computed by a circuit of size \(O(1)\). By appropriately “composing” these circuits for all \(i \in \{1, \ldots, t\}\), \(z_c\) can be computed by a circuit of size \(O(t) = O(T(n)^2)\). Furthermore, by following the above procedure, this circuit can be constructed in time polynomial in \(n\).

Having proved the fact, we now apply it to reduce \(\text{TM-SAT} \leq_P \text{CIRCUIT-SAT}\). Let \((\alpha, x, 1^n, 1^t)\) be given.

Let \(M\) be the Turing machine that, given \(y\), computes \(M_\alpha(x, y)\) for \(t\) steps and accepts if and only if \(M_\alpha\) accepted. Construct the circuit family \(\{c_n\}_{n \in \mathbb{N}}\) corresponding to \(M\) using the fact. Then the size of \(c_n\) is \(O(t^2)\), and \(c_n\) is satisfiable if and only if \((\alpha, x, 1^n, 1^t) \in \text{TM-SAT}\), so we have reduced \(\text{TM-SAT} \leq_P \text{CIRCUIT-SAT}\).

Because we previously showed (Proposition 6) that \(\text{CIRCUIT-SAT}\) was in \(\text{NP}\), and we have now proved it to be \(\text{NP-hard}\), we also see that \(\text{CIRCUIT-SAT}\) is \(\text{NP-complete}\). ■

**Lemma 9** \(\text{CIRCUIT-SAT} \leq_P \text{3-SAT}\)

**Proof.** Given a circuit \(\varphi\) with input variables \(u_1, \ldots, u_n\), for each \(\lor\) or \(\land\) gate \(g_i \in \varphi\) introduce a new variable \(v_{g_i}\) that represents the output of that gate. For each such gate, the inputs are either of the form \(u_i, v_{g_i}, \) or \(\neg v_{g_i}\). If \(v_{g_k}\) is a \(\land\) gate with inputs \(w_i\) and \(w_j\) (of the above form), let

\[
C_k = (\neg v_{g_k} \lor w_i \lor w_j) \land (\neg v_{g_k} \lor w_i \lor \neg w_j) \land (\neg v_{g_k} \lor \neg w_i \lor w_j) \land (v_{g_k} \lor \neg w_i \lor \neg w_j).
\]

If \(v_{g_k}\) is a \(\lor\) gate, let

\[
C_k = (\neg v_{g_k} \lor w_i \lor w_j) \land (v_{g_k} \lor w_i \lor \neg w_j) \land (v_{g_k} \lor \neg w_i \lor w_j) \land (v_{g_k} \lor \neg w_i \lor \neg w_j).
\]

Then let \(\psi = \bigwedge_k C_k\). Observe that \(\psi\) is satisfiable if and only if \(\varphi\) is. This is the desired reduction. ■

From these lemmas, it is now easy to assemble the

**Proof of the Cook-Levin theorem:**

By Lemma 8, \(\text{TM-SAT} \leq_P \text{CIRCUIT-SAT}\). By Lemma 9, \(\text{CIRCUIT-SAT} \leq_P \text{3-SAT}\). Hence \(\text{TM-SAT} \leq_P \text{3-SAT}\); this means that \(3\text{-SAT}\) is \(\text{NP-hard}\).

It remains only to show that \(\text{3-SAT}\) is in \(\text{NP}\). This is easy because if \(\varphi\) is a \(3\text{-CNF formula}\), then a satisfying assignment is a polynomial-size witness to the satisfiability of \(\varphi\).

Combining these facts, \(3\text{-SAT}\) is \(\text{NP-complete}\). ■

2
Why do we study 3-SAT and 3-CNF Formulas? For one, although TM-SAT is an important problem in that we used it to establish the existence of NP-complete problems, it is less accessible in that it is intimately tied to Turing Machines. 3-SAT, however, is an easier problem to work with as it is divorced from Turing Machines, asking only about the satisfiability of certain boolean expressions. This makes it useful for reductions. Indeed, SAT and/or 3-SAT are the initial problems from which a myriad of other important problems are shown to be NP-complete. See figure 2.4 on page 51 of Arora and Barak. Additionally, 3-SAT is an important problem in that CNF expressions are already intensively studied in logic and have been for a long time.

Now we give an example of an NP-complete language for which it is not hard to see that is NP-complete having established 3-SAT as NP-complete. A 0/1 integer program is a system of m linear inequalities in n variables with rational coefficients. We seek an assignment of either 0 or 1 to the variables that satisfies all the equations. We define a language “0/1 int prog” to be those systems which have a solution in 0 and 1.

**Theorem:** 0/1 Integer Programming is NP-complete.

**Proof:** First we observe that 0/1 int prog ∈ NP. This is easy to see. The witness is the satisfying boolean assignment which is a sequence of n bits. This is clearly polynomial in the size of the input. Checking the witness involves checking m linear inequalities, which can be done in time poly(n, m).

Next, we want to show that 3-SAT \( \leq_p \) 0/1 int prog. This will show that 0/1 int prog is in NP. So suppose we are given a 3-SAT \( \psi \). To each clause of \( \psi \) we associate an inequality. This is best illustrated by example: \( u_i \lor u_j \lor \overline{u_k} \rightarrow [u_i + u_j + (1 - u_k) \geq 1] \). In general if \( u_i \) appears in the clause we associate the variable \( u_i \) in an inequality in our integer program and if \( \overline{u_i} \) appears we associate \((1 - u_i)\) in the inequality. Since the clause is a sequence of ORs we need only one to be true and if \( u_i \geq 1 \) if \( u_i = 1 \) and \((1 - u_i) \geq 1\) if \( u_i = 0\). So the resulting system of m inequalities (where m is the number of clauses in our 3-CNF expression) is solvable with 0 and 1 if and only if the 3-CNF expression is satisfiable.

It is also worth noting that the language int-prog is NP-complete. Since 0/1 int prog is a sub problem it is clear that int-prog is NP-hard. To see that int-prog ∈ NP takes a little more work. It is not as obvious that the witness (the satisfying integer values of the n variables) must be poly(n, m) in size, but this turns out to be true.

From studying the many interesting problems in NP that turn out to be NP-complete, one might think that any problem in NP that is not in P might be NP-complete. This turns out not to be true (if P ≠ NP):

**Theorem (Ladner):** If P ≠ NP then there exists \( L \in NP \) such that \( L \notin P \) and \( L \) is not NP-complete.

There are two natural and important languages in NP that are conjectured to be of this type (not in P but not NP-complete). Note, that they are of course not known to be of this type as this would imply P ≠ NP.

- Graph-Isomorphism
- Factoring. Note that given n, determining the factorization of n is not a decision problem. However, we can pose the following decision problem: given n and bounds a and b, does n have an integer factor between a and b. If this can be computed in time \( T(n) \) then employing a binary search, n can be factored in roughly \( T(n) \log(n) \) steps. So it is highly doubtful that the factoring decision problem is in P. However, subexponential factoring algorithms have been discovered suggesting that this problem might be “easier” than NP-complete problems.

Next we discuss another important complexity class, co-NP. First, we define the complement of a language. If \( L \subseteq \{0,1\}^* \) then we define \( \overline{L} = \{0,1\}^* \setminus L \).
Definition: $\text{co-NP} = \{ L : \overline{L} \in \text{NP} \}$.

For example, CNF formulas which are not satisfiable are in co-NP. As an exercise, one can show that the following alternative definition is equivalent.

Definition: An equivalent definition of co-NP is as follows. $L \subseteq \{0,1\}^*$ is in co-NP if there exists a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial time TM $M$ such that for all $x \in \{0,1\}^n$ we have:

$$x \in L \iff \forall y \in \{0,1\}^{p(|x|)}, m(x,y) = 1.$$ 

It is not hard to see that $\text{P} \subseteq \text{co-NP}$. However, it is unresolved if $\text{NP} = \text{co-NP}$, but this is conjectured not to be the case.

Theorem (Time Hierarchy): Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a time-constructable function. Then $\text{DTIME}(f(n)) \neq \text{DTIME}(\omega f(n) \log f(n))$, i.e. the containment is strict. This is a powerful theorem. For example, it shows that in our definition $\text{P} = \bigcup_{c \geq 1} \text{DTIME}(n^c)$. So there are languages in $\text{P}$ for which there is no $O(n^{1000})$ algorithm. We will prove the weaker statement $\text{DTIME}(n) \neq \text{DTIME}(n^{10})$ which gives the main idea, but doesn’t get bogged down in the details.

Proof: We use diagonalization. Enumerate all Turing Machines $M_i$ as follows:

$M_1, M_1, M_2, M_1, M_2, M_3, M_1, M_2, M_3, M_4, \ldots$ We can relabel this sequence as $N_1, N_2, N_3, \ldots$ In fact any sequence $\{N_j\}_{j=1}^\infty$ of TMs such that each $M_i$ appears in the sequence an infinite number of times will work. Now consider the following language, $L$, defined as follows. For input $\alpha$ where $n = |\alpha|$ if $M_\alpha(\alpha)$ accepts within $n^2$ steps then we reject (i.e. $\alpha \notin L$). Otherwise we accept. The simulation of $M_\alpha(\alpha)$ can be done using the universal Turing Machine, U-TM, which induces only a $\log(n)$ slowdown. So it is easy to see that $L \in \text{DTIME}(n^{10})$. Now suppose, $L \in \text{DTIME}(n)$ then there exists a TM $M$ deciding $L$ in time $cn$ for some constant $c$. Since our enumeration $\{N_j\}_{j=1}^\infty$ contained an infinite number of copies of $M$, we have for some $\alpha$ that $M = M_\alpha$ and $c|\alpha| < |\alpha|^2$. But then we have $M(\alpha) \neq M_\alpha(\alpha)$ by our definition of $L$. $\Rightarrow \Leftarrow$.

Next, we develop the notion of non-deterministic Turing Machines and NTIME.

Definition: We say $L \in \text{NTIME}(f(n))$ if there exists a TM $M$ and a constant $c$ such that for all $x \in \{0,1\}^n$:

$$x \in L \iff \exists y \in \{0,1\}^{cf(n)} \text{ such that } M(x,y) \text{ haults in at most } cf(n) \text{ steps and outputs } 1.$$ 

Definition: An alternative definition of NP is then $\text{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$.

The name NP comes from “Nondeterministic polynomial time.” It is those languages which are computable in polynomial time on a nondeterministic Turing machine (NDTM). What is an NDTM and where can I get my hands on one?!

Definition (loose): A nondeterministic Turing machine (NDTM) is defined similarly to a Turing machine except that it has two different transition functions. At each stage of computation it branches and uses each transition function to continue its computation. It returns 1 if there is some sequence along which it haults and outputs 1. If every branch haults without returning 1 then the output is 0. NDTMs are described on page 41 of Arora and Barak.