Complexity Class NP:

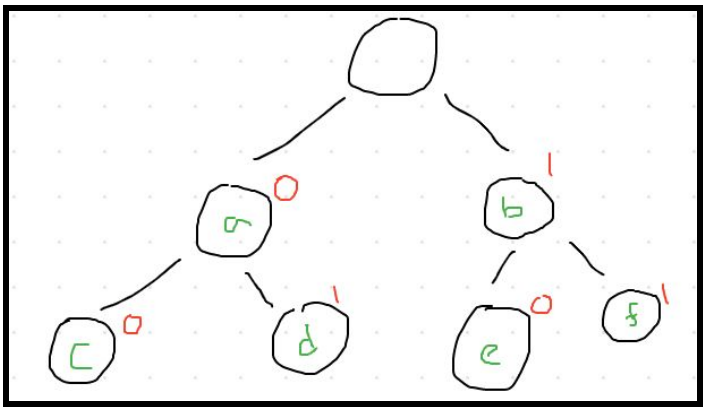
- **NP** is the set of languages/decision problems that can be decided by NTMs with polynomial running time.
- Recall that one of the theoretical justifications for the polynomial time thesis had an exception to it. That exception is that if an NTM takes polynomial time, we don't know if its equivalent regular TM takes polynomial time.
- The running time of an NTM M on input x is the maximum number of moves in any branch of M's computation of input x. Essentially, this is the height of the tree.
- The running time of an NTM M is the function T: N → N s.t. T(n) = max running time of M on any input of size n.
- **Theorem 8.1:** P ⊆ NP

  Proof:
  A TM is a special NTM where we always only have 1 choice.
  Hence, the set of languages that can be decided by a TM with polynomial running time is a subset of the set of languages that can be decided by an NTM with polynomial running time.

  Conjecture 8.2: P ≠ NP and hence P ⊂ NP
  This conjecture states that P is a proper subset of NP.
- **Thm 8.3:** Recall TSP-DEC and IS-DEC from lecture 7. TSP-DEC and IS-DEC are in NP.

  Proof that IS-DEC is in NP:
  We need to show/describe an NTM that runs in polynomial time and accepts yes-instances of the IS-DEC problem.

  A polytime NTM for IS-DEC does:
  1. Nondeterministically choose a set of b nodes of the given graph.
     Suppose there are n nodes in total in the graph.
     We can represent each node as a string of bits of length $\log_2 n$.
     Hence, a set of b nodes would have $b*\log_2 n$ bits.
     Nondeterministically choosing a set of b nodes means write down a set of b nodes.
     Here is an example of how we can nondeterministically choose a set of b nodes:



     The node with "a" in it means that the string starts with 0.
     The node with "b" in it means that the string starts with 1.

The node with “c” in it means that the string starts with 00.
The node with “d” in it means that the string starts with 01.
The node with “e” in it means that the string starts with 10.
The node with “f” in it means that the string starts with 11.
After $b*\log_2 n$ branches, we have all $b*\log_2 n$ possible strings.

2. Verify that no two nodes in the chosen set have an edge between them.
   If so, accept.
   Else, reject.

Step 1 takes polynomial time with respect to the size of the input, which is the number of nodes + the number of edges.
If we have m edges and n nodes, then the size of the input is m+n.
At most, b can be n. (b is n if we choose all n nodes.) So, at most, it takes $n*\log_2 n$ steps.
This is polynomial with respect to the size of the input.

Step 2 is completely deterministic.
Once we fix the set, we can look at every pair of nodes in the set to see if any two have an edge between them.
Hence, step 2 also takes polynomial time with respect to the size of the input.

Since each step takes polynomial time with respect to the size of the input, put together, it still takes polynomial time with respect to the size of the input.
Hence, IS-DEC is in NP.

- Here’s an alternative characterization of NP:

Consider TSP-DEC from lecture 7.
At the present time, there is no substantially better way of solving this problem than trying out every possible permutation of the n nodes in the graphs and seeing if there is a permutation that corresponds to a tour within the budget.
This method cannot be done in polynomial time.
However, if you are given a particular tour, you can easily verify if it is a proper tour and if it is within the budget. The difficult part is finding the tour.
There are a lot of problems with this property. It’s hard to find the solution, but easily to verify if a proposed solution is a valid solution or not.
All problems in NP have the property that if given a proposed solution which solves the problem in polynomial time, can be verified, in polynomial time, if it is an actual solution to the problem.

We will define a proposed solution to be a **certificate**.

TSP-DEC has a short and efficiently checkable certificate.
The certificate has a length of $n\log_2 n$ given there are n nodes in the graph. (short)
It just checks that the given nodes define a tour and that the sum of the weights of the edges is less than or equal to the budget. This takes polynomial time. (efficiently checkable)

Note: Certificates must be:

1. Comprehensive, meaning that all yes-instances have one.
2. Sound, meaning that all no-instances do not have one.
3. Short.
4. Efficiently checkable.

Definition: A polytime verifier for a language L is a polynomial time algorithm for a deterministic TM, V, s.t. $x \in L$ iff there exists a certificate for x, denoted as c_x, s.t. $|\langle c_x \rangle| = O(|\langle x \rangle^k|)$ for some constant k and V accepts $\langle x, c_x \rangle$.

You give the problem, x, and a certificate, c_x, to V and V will verify if that certificate is an actual solution to the problem.

Example: $\langle G, wt, b \rangle \in$ TSP-DEC iff there exists a list of edges s.t.

1. The list of edges form a tour of the graph.
2. The sum of the weights of the edges is at most b.

In this case our list of edges is our certificate. Steps 1 and 2 is what our TM V would do to check.

- **Theorem 8.4:** $L \in NP$ iff L has a polynomial time verifier.

Proof:

(=>)

If $L \in NP$, then it has a polynomial time verifier.

→ By definition, if a language is in NP, then there exists a polytime NTM that decides it.

→ Hence, there's a polytime NTM, M, that decides L.

→ If $x \in L$ and M is a decider for L, then there must be a sequence of configurations that accepts x. Hence, we define c_x as an accepting computation of M on x.

I.e. $c_x = C0 \vdash C1 \vdash ... \vdash Cl$ s.t. C0 is in the initial configuration of M on x and Cl is the accepting configuration of M on x.

→ $|\langle c_x \rangle|$ is short as it is polynomial in respect to $|\langle x \rangle|$

→ The verifier checks:

1. C0 is the initial configuration.
2. Cl is the accepting configuration.
3. $Ci \vdash Ci+1 \ \forall i \ 1 \leq i \leq l$

(<=)

If L has a polynomial time verifier, then $L \in NP$.

→ By definition, $x \in L$ iff there exists a certificate for x, denoted as c_x, s.t. $|\langle c_x \rangle| = O(|\langle x \rangle^k|$, for some constant k, and verifier V accepts $\langle x, c_x \rangle$ in polynomial time.

→ Construct a NTM M that on input x does the following:

1. Non-deterministically write certificate $\langle c_x \rangle$.
2. Run V on $\langle x, c_x \rangle$
    a. If V accepts, then accept.
    b. If V rejects, then reject.

M can do step 1 in polynomial time because c_x is short.
M can do step 2 deterministically in polynomial time.
Polynomial + polynomial is still polynomial.
Hence, the overall time it takes is polynomial.

- We have 2 ways of arguing that a language/decision problem is in NP:
    1. Show a polynomial time NTM for L.
    2. Show a polynomial time deterministic verifier for L.

  Note: The 2 ways are really the same thing.
- Here are 4 steps that we can use to prove that a language/decision problem is in NP:
    1. Show how to generate certificates.
       We can use an NTM to generate all of the strings and say that each one is a certificate.
    2. Argue that the certificate is short in size, meaning polynomial in size.
    3. Explain how the verifier works to validate input.
    4. Argue the verifier works in polynomial time. This is why we need the certificate to be short.

Satisfiability Problem (SAT):

- Suppose we have a proposition formula, Φ.
  A **truth assignment** tells us for every variable in the formula is it true or false.
  Once we know whether each variable is true or false, we know if the formula is true or false.

  E.g. Suppose $\Phi = (x \vee y) \wedge (\neg z \wedge \neg(x \vee \neg y))$
  Here is a truth assignment:
  Let x = true. Let y = true. Let z = false.
  Then:
  $(x \vee y) \rightarrow (T \vee T) \rightarrow T$
  $(\neg z \wedge \neg(x \vee \neg y)) \rightarrow (\neg F \wedge \neg(T \vee \neg T)) \rightarrow (T \wedge \neg(T \vee F)) \rightarrow (T \wedge \neg T) \rightarrow (T \wedge F) \rightarrow F$
  $T \wedge F \rightarrow F$
  Hence, in this case, Φ is False.

  Here's another truth assignment:
  Let x = false. Let y = true. Let z = false.
  Then:
  $(x \vee y) \rightarrow (F \vee T) \rightarrow T$
  $(\neg z \wedge \neg(x \vee \neg y)) \rightarrow (\neg F \wedge \neg(F \vee \neg T)) \rightarrow (T \wedge \neg(F \vee F)) \rightarrow (T \wedge \neg F) \rightarrow (T \wedge T) \rightarrow T$
  $T \wedge T \rightarrow T$
  Hence, in this case, Φ is True.

  A formula is **satisfiable** if there exists a truth assignment that makes the formula true.
  In our example above, Φ is satisfiable because there exists at least 1 truth assignment that makes the formula true.
- **SAT Decision Problem**
  Instance: ⟨Φ⟩, where Φ is a propositional formula.
  Question: Is Φ satisfiable?
  No one knows a polynomial time algorithm for this.

The most obvious algorithm is to test every truth assignment. However, if there are n variables, and each variable has 2 values (T or F), it would take 2^n time to test every truth assignment.

Since this is a decision problem, we need to encode the alphabet.
Some symbols we know that are in this alphabet are (,), ¬, ∧, ∨.
However, a problem arises with the variables. This is a problem because although an individual propositional formula has a finite number of variables, the set of all propositional formulas has an infinite number of variables.
We can fix this problem by representing each variable in binary with $\log_2 n$ bits, where n is the number of variables.

- **Theorem 8.5:** SAT ∈ NP

Proof:
The certificate in this case is a truth assignment that makes the formula true.
The verifier evaluates the formula given the certificate.

NP Completeness:

- In the world of computability, we divided languages/decision problems into “easy”, meaning decidable, and “hard”, meaning undecidable.
Previously, we established one “hard” problem, U. To prove that U is “hard”, we used diagonalization. Then, we proved that problem X is “hard” by proving that $U \leq_m X$.
In the world of complexity, we divide languages/decision problems into “easy”, meaning that the language is in P, and “hard”, meaning that the language is in !P∩NP.
Note: If P = NP, then !P∩NP is empty. (We don’t know if P = NP).
So, instead we use the term **NP Complete** or **NPC** to describe these problems.
NPC problems are believed to be hard and are the hardest problems in NP.

Here’s a chart to show dividing of languages in computability and complexity.

	Easy	**Hard**
Computability	Decidable	Undecidable
Complexity	In P	In NPC

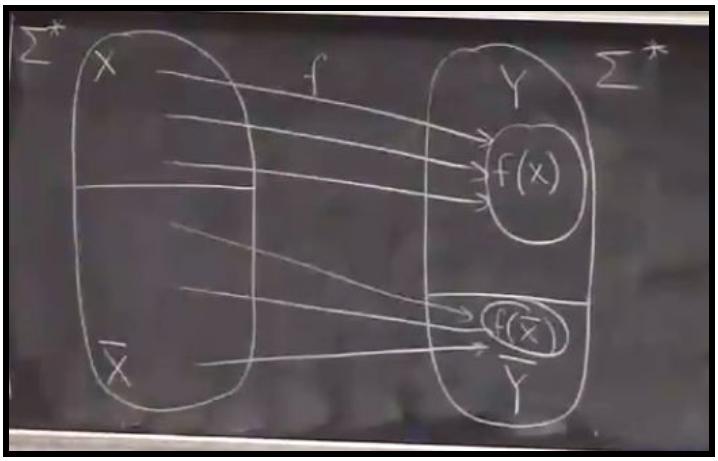
- There are 2 notations of time complexity reduction that roughly correspond to Turing reduction and Mapping reduction.
    1. **Cook Reduction:**
    - Let X, Y be problems (not necessarily decision problems). X **cook reduces** to Y , denoted as $X \rightarrow_P Y$ if there exists a polynomial time algorithm A that solves X given an oracle (blackbox subroutine) for Y where each use of the Y-oracle counts as 1 step.
      Note: While using the Y-oracle counts as 1 step per use, A has to prepare input(s) to the Y-oracle. This preparation is not part of the 1 step and is counted separately.
    - Cook reduction corresponds to Turing reduction.

- **Theorem 8.6:** If $X \rightarrow_P Y$ and there exists a polynomial time algorithm for Y, then there exists a polynomial time algorithm for X.

  Proof:
  If $X \rightarrow_P Y$, then there exists a polynomial time algorithm A that solves X, but instead of using a Y-oracle, it uses the polynomial time algorithm for Y.
  A polynomial of a polynomial is still a polynomial.
  Hence, the resulting algorithm is still polynomial time.

2. Karp Reduction:

- Let $X, Y \subseteq \sum^*$ be languages. X **karp-reduces/polytime reduces** to Y, denoted as $X \leq_P Y$, iff there exists a function $f: \Sigma^* \rightarrow \Sigma^*$ that can be completed in polynomial time s.t. $x \in X$ iff $f(x) \in Y$.
- Here's a diagram to help with the definition.



  f maps all the Yes-instances of X to a subset of the Yes-instances of Y.
  Similarly, f maps all the No-instances of X to a subset of the No-instances of Y.

- **Theorem 8.7:** If $X \leq_P Y$ and $Y \in P$ then $X \in P$.

  Proof:
  Similar to the proof of theorem 8.6.
  X-SOLVER on input x does 2 things:
  1. y = f(x)
  2. return Y-SOLVER(y)

  I.e.
  X-SOLVER(x)
  y = f(x)
  return Y-SOLVER(y)

- **Theorem 8.8:** $\leq_P$ is transitive.
  I.e. If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

  Proof:
  Given $x \in X$ iff $f(x) \in Y$ and $y \in Y$ iff $g(y) \in Z$, then $f(x) \in Y$ iff $g(f(x)) \in Z$

- **Definition:** Y is an NP-Complete (NPC) language iff
  a. $Y \in NP$
     I.e. Y is in NP.
  b. $\forall X \in NP, X \leq_P Y$
     I.e. Every problem, X, in NP must be polytime reducible to Y.
     This means that Y is the "hardest" in NP.
- **Theorem 8.9:** If Y is NPC and $Y \in P$ then P=NP.

  Proof:
  Assume that Y is NPC and $Y \in P$.
  Then, by definition, $\forall X \in NP, X \leq_P Y$.
  Then, by theorem 8.7, $X \in P$.
  Hence, every problem in NP is in P.
  Hence, P = NP.
- **Theorem 8.10:** If Y is NPC, $Z \in NP$ and $Y \leq_P Z$ then Z is NPC.

  Proof:
  Since $Z \in NP$, then part (a) of the definition of being in NPC holds.

  To show that part (b) of the definition of being in NPC holds:
  $\forall X \in NP, X \leq_P Y$. (This is because Y is NPC.)
  We also know that $Y \leq_P Z$. (This is by assumption.)
  Hence, by theorem 8.8, $\forall X \in NP, X \leq_P Z$. (This is because $\leq_P$ is transitive.)
  Hence, Z is in NPC.
- Later on we will show SAT is NPC with a specialized argument.
  After, we can use SAT as our "seed" to prove other problems are NPC.
  We will also prove that $SAT \leq_P IS$.

- **Theorem 8.11 (Ladner's Theorem):** If $P \neq NP$ then there are problems in $NP - (NPC \cup P)$.