LESSON: NP1: Definitions

***

(Slide 1) NP: Overview

How do we prove that a problem is computationally difficult? … meaning that it is hard to devise an efficient algorithm to solve it for all inputs?

To do this, we'll prove that the problem will then be **NP-complete**. In this section, we'll look at the definition of the class **NP** and what exactly it means for a problem to be NP-complete.

We'll first detail the notion of a **reduction**. We've actually seen several examples of reductions in this class before. For example, we reduced the 2-SAT problem to the Strongly Connected Components (SCC) problem. We'll formalize this concept of reduction.

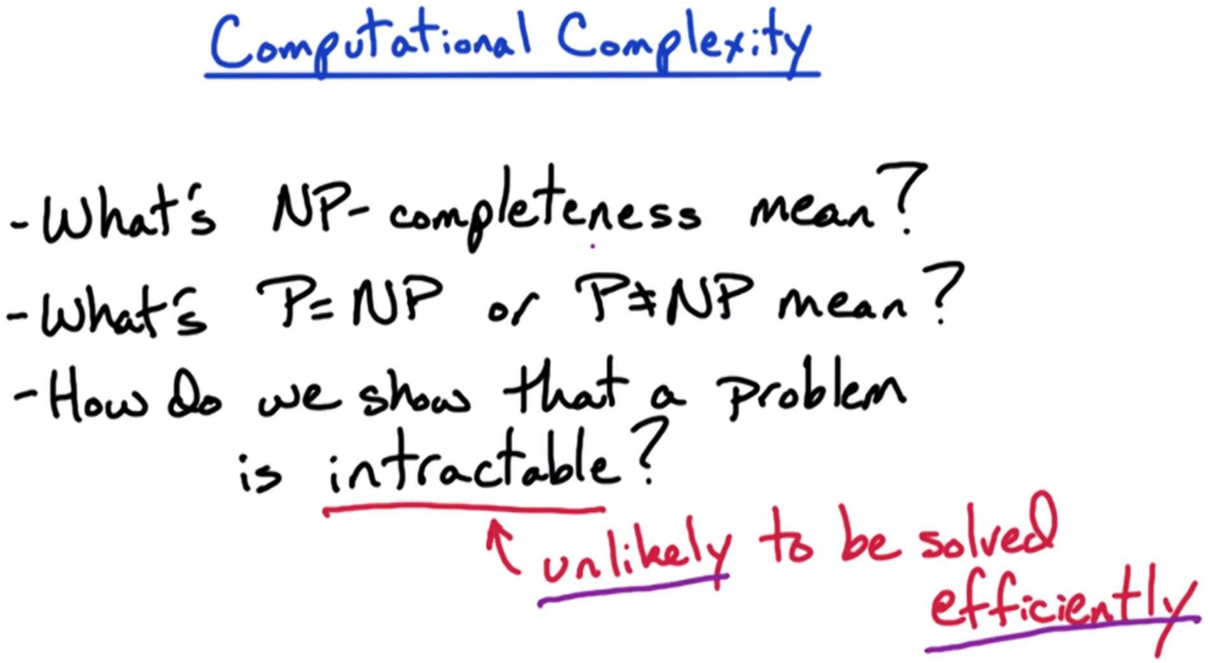
Then we'll see what it means that a problem is **NP-complete** and how to prove that it is NP-complete.

We'll look at a bunch of examples of NP-completeness proofs: 3-SAT, Graph Problems (for example: Independent Set, Clique, Vertex cover), and the classic Knapsack problem.

Finally, we'll see Alan Turing's proof that the halting problem is not just computationally difficult but, in fact, it's not solvable in general. Formally, it's **undecidable**.

***

(Slide 2) NP1: Lecture Outline



Computational Complexity

- What's NP-completeness mean?
- What's P=NP or P≠NP mean?
- How Do we show that a problem is intractable?

↑ unlikely to be solved efficiently

In this section, we're going to study computational complexity.

We're going to address the following question: “What precisely does **NP-completeness** mean and how do we prove that a problem is **NP-complete**?”

Many of you have probably heard of the equation **P=NP** or **P!=NP**. We're going to take a look at what precisely this question means.

Finally we're going to look at how do we show that a particular problem is **intractable**.
Now what exactly do we mean by intractable? … we mean that it is unlikely to be solved efficiently. Now we'd love to say that it definitely cannot be solved efficiently, instead of unlikely to be solved efficiently. But that's more than we can achieve at this time.

Now what exactly do we mean by “**efficiently**”? … well, by that we mean that is **polynomial in the input size**. Can we solve this particular problem, in time polynomial in the input size?

Now, in order to show that a particular problem is intractable, what we're going to do is, we're going to prove that it's NP-complete. So we're going to learn how to prove that a particular problem is NP-complete. So let's dive in now to see what P and NP mean.

***

(Slide 3) Complexity Classes

We're going to define NP as the class of all **search problems**. What exactly do we mean by search problems? -- well, we'll look at that in more detail in a second. Now, there's one important distinction that we need to bring out at the beginning.

We're using the class of search problems to define the class NP. Now, traditionally, what many of you may have seen in your undergraduate course is that the NP is defined with the class of decision problems instead of search problems. This is a somewhat minor distinction.

But, for consistency in this class, we're going to use search problems for NP. Why am I using search problems instead of decision problems? Well, it's somewhat more natural to use search problems. In addition, the group of textbooks also defines NP with respect to search problems.

Now, if you've seen decision problems before, the appeal of search problems is that it gets rid of the need for a **witness** for particular instances. In any case, let's forget about the notion of decision problems and let's focus on search problems in this course.

What do we mean by search problems? Let's look at a **rough definition of search problems** and then in the next slide, we will look at a formal definition of search problems.

Now, roughly, a search problem is **a problem where we can efficiently verify solutions.** So, if I give you particular input and I give you a solution to that input, then I can verify that that solution is in fact a solution to that input. I can check solutions in polynomial time. So, the time it takes to generate the solution doesn't have anything to do with it. All that matters is if I give you an input and I give you a solution, then you can verify that solution in polynomial time.

Now, if I want to know about the time it takes to generate a solution then look at the class P. The class P are those search problems which can be solved in polynomial time so I can generate a solution in polynomial time. Whereas, NP just says that it can verify solutions in polynomial time.

Now, of course, if I can generate a solution in polynomial time then I can also verify solutions in polynomial time. So, P is a subset of NP. Any problem that can be solved in P can also be solved in NP.

Formally, we have that P as a subset of NP. Why is that? Well, NP is a class of all search problems so these are problems where we can verify solutions in polynomial time. P is a class of search problems that can be solved in polynomial time. A problem in P means that we can verify solutions in polynomial time and we can generate solutions in polynomial time.

***

(Slide 4) Comparing P and NP

Now, we'll take a look at the formal definition of search problems momentarily. But, let's first take a look at the intuitive difference between P and NP.

Now, roughly, **a search problem** is a problem where given an input and as proposed solution, we can verify that the solution is in fact a solution in polynomial time - where this is polynomial in the input size. **P** are those search problems where we can also generate solutions in polynomial time.

Now what is P=NP or P != NP, what do they mean? **P** is the class of problems that we **can solve** in polynomial time, while **NP** are those problems where we **can verify** solution in polynomial time.

So, what does the P=NP question mean? What if P=NP? That means that it's as difficult to solve a problem as it is to verify the solution. Think about this in this context of proofs that are trying to prove a theorem. NP says that if I give you the proof for a theorem, then you can check it in polynomial time. That's sort of like going line by line in the proof and checking each line in the proof. Doesn't sound too difficult.

Now, what's the analogue for P? While solving a problem, the analogue for a proof is generating the proof of the theorem. So, is generating the proof for of the theorem as hard as verifying a proof for theorem? So, if I can check the proof line by line, is that as hard as generating the actual proof? -- it seems much more difficult to generate the proof or generate a solution to a problem than to just verify that a solution or verify that a proof is correct.

So, if we show that P=NP, that implies that if I can verify a solution in polynomial time, then I can also solve the problem in polynomial time – so, whenever I can check a proof, I can also generate a proof for that theorem.

For many of us, it seems much more plausible that P is not equal to NP. Why? Because it seems much easier to verify a solution, to check a proof than to actually solve the problem or to generate the proof for the theorem. Now let’s formally define search problems.

***

(Slide 5) Search Problems

A search problem is a problem in the following form. First time given an **instance** I. What is I? I is **the input to the problem**. For example, it might be a draft if I'm doing a graph problem, or I might be a formula, if I'm doing a problem such as satisfiability.

Now, if I has a solution for this problem, then I want to find such a solution. I can output any solution S which is a solution for I. But if I has no solutions, then I simply output NO. So, if this input has a solution, then I output any such solution. If it has no solutions, I output NO.

Now, this is the basic form of a search problem, but I have a further requirement in order to be a search problem. Now, let me rephrase this requirement. So, for a particular problem to be a search problem, it first has to be of this form.

- So, if it's a yes instance so there is a solution to this input, then we have to output some such solution.
- And if there are no solutions then we simply output NO.

Furthermore, we have the following requirement: for any input that has a solution, if I give you a solution -- it doesn't have to be the one that's generated by some particular algorithm -- I give you any solution -- then we need to be able to verify in polynomial time that S, this proposed solution, is in fact a solution to this instance.

So, we just have to be able to check the proof. So if I give you a proof, then I can check that proof line by line that it is a proof to this input, to this theorem. Now if this input is a NO instance -- so there are no solutions -- I don't have to do anything in that case. Simply on yes instances, then I have to be able to verify solutions. But I only have to verify solutions when I'm given the solution. So I don't have to generate the solutions at all. Simply if I'm given the solution, then I can verify that it is a solution to the input.

And it's important to stress that what we mean by **efficiently** verify, we mean in time **polynomial in the input size**.

Now, how would I show that a particular problem is a search problem? Well, I will show you an algorithm which can verify solutions in polynomial time. So I would show you an algorithm which would take as input an instance I, which is an input to the original search problem, and a proposed solution to that input.

Now given I and S, now what should my algorithm do? Well my algorithm is going to verify that S is in fact a solution to the I, and it's going to do so in time polynomial in the size of I.

Typically, it's very easy to show such an algorithm. So let's take a look at a few example problems that we've seen so far in this course and we will verify that they are in fact search problems and therefore they are in NP.

***

(Slide 6) SAT Problem

I will start with the satisfiability problem. This was one of the original NP complete problems. Therefore, it has some special significance in the theory of NP completeness. Now this is the SAT problem which we defined before when we looked at graph algorithms. And we saw an algorithm for 2-SAT.

Now, the input of SAT is a Boolean formula f in conjunctive normal form, CNF. And we'll say that f has n variables, x_1 through x_n, and it has m clauses. Now the output is a satisfying assignment if one exists. What does that mean? That means an assignment to the variables, true or false, for each of the variables so that when we evaluate the formula, it evaluates to True.

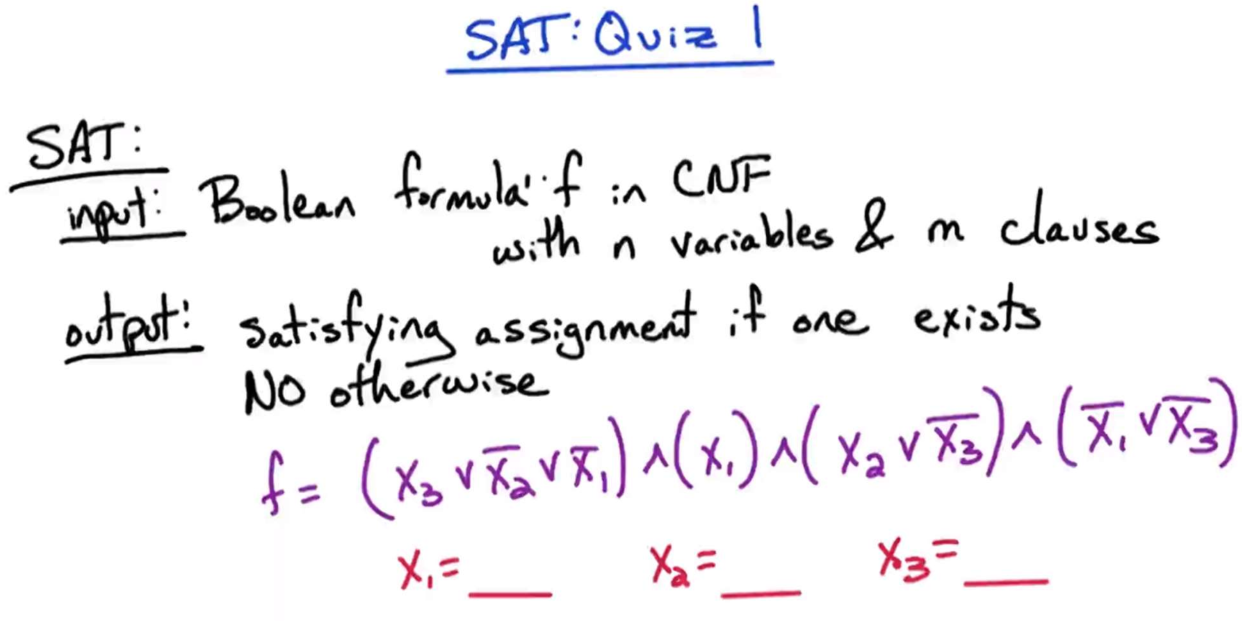
Now, if there is no satisfying assignment -- so there's no way to assign True or False to each of the variables so that the formula evaluates to True -- then we simply output no.

Now here's an example input to SAT with three variables and four clauses. So n equals three and m equals four.

***

(Slide 7) SAT Example

NP1: Definitions: SAT Example



SAT: Quiz 1

SAT:

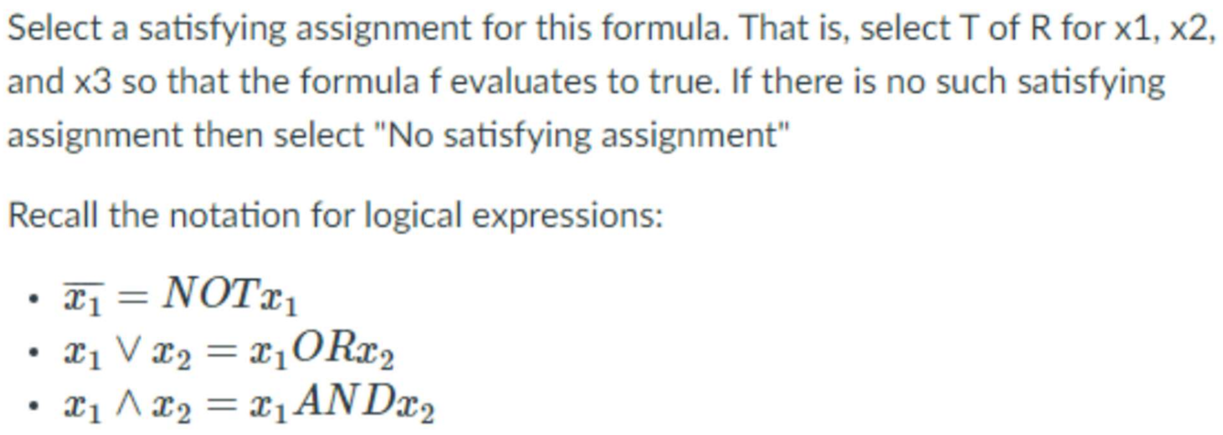
input: Boolean formula f in CNF
with n variables & m clauses

output: satisfying assignment if one exists
NO otherwise

$f = (x_3 \vee \overline{x_2} \vee \overline{x_1}) \wedge (x_1) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3})$

$x_1 =$ ___ $x_2 =$ ___ $x_3 =$ ___

Let's make sure that you recall the notation for the SAT problem. So why don't you go ahead and write a satisfying assignment for this formula. So, give an assignment for x1, x2 and x3 so that the formula evaluates to True. And, if there is no such satisfying assignment, go ahead and enter NO.



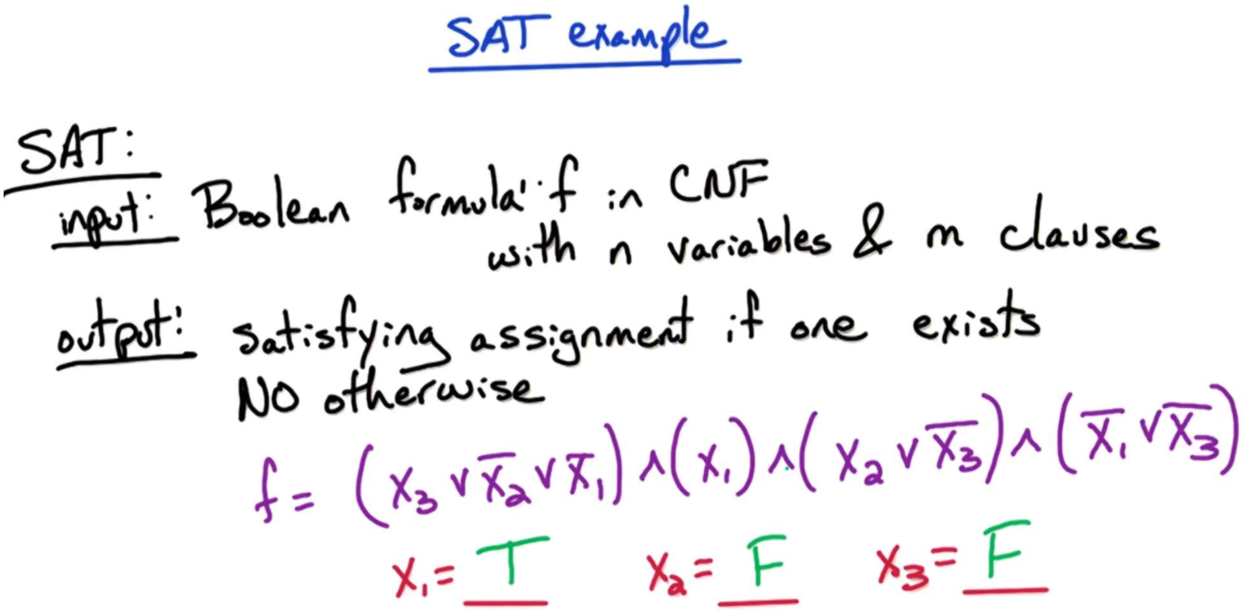
Select a satisfying assignment for this formula. That is, select T of R for x1, x2, and x3 so that the formula f evaluates to true. If there is no such satisfying assignment then select "No satisfying assignment"

Recall the notation for logical expressions:

- $\overline{x_1} = NOT x_1$
- $x_1 \vee x_2 = x_1 OR x_2$
- $x_1 \wedge x_2 = x_1 AND x_2$

***

(Slide 7) SAT Example (Answer)



SAT example

SAT:

input: Boolean formula f in CNF
with n variables & m clauses

output: satisfying assignment if one exists
NO otherwise

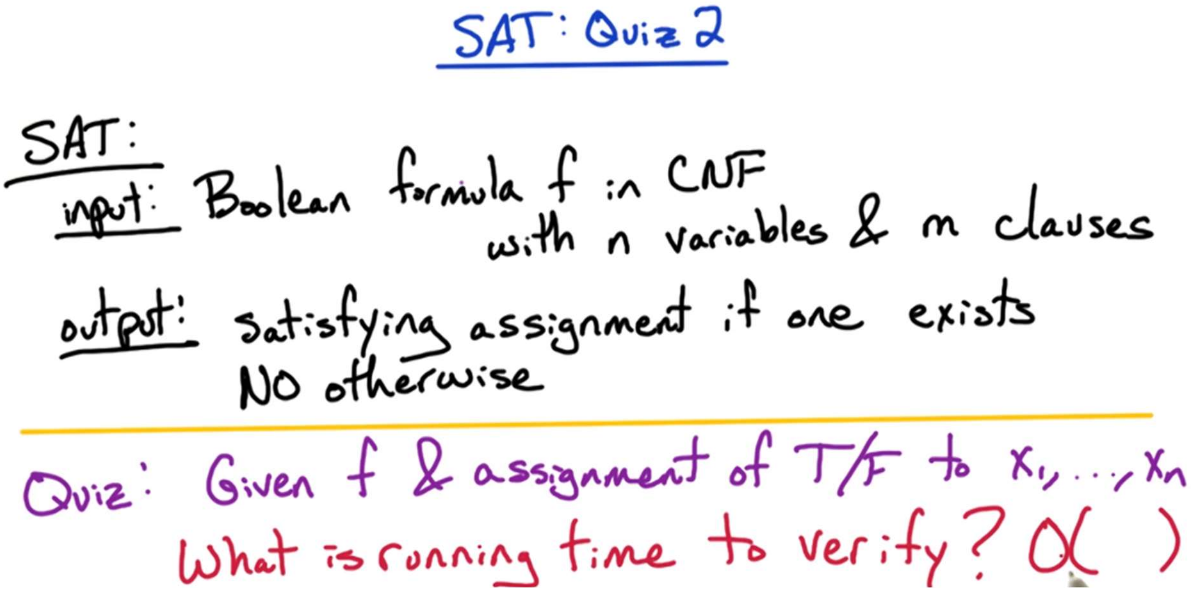
$f = (x_3 \vee \overline{x_2} \vee \overline{x_1}) \wedge (x_1) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3})$

$x_1 =$ T $x_2 =$ F $x_3 =$ F

Now this particular input has a satisfying assignment. I can set x1 = T, x = F, and x3 = F. Let's check it. For the first clause, I have x2 = T, so this is our satisfied variable. For the second clause, I have x1 is True, so this is a satisfied literal. For the third clause, I have that x3 is False, so this is a satisfied clause. For the last clause, I have that x3 is False, so this is a satisfied literal. So each clause has at least one literal satisfied.

***

(Slide 8) SAT in NP



SAT: Quiz 2

SAT:

input: Boolean formula f in CNF with n variables & m clauses

output: satisfying assignment if one exists
NO otherwise

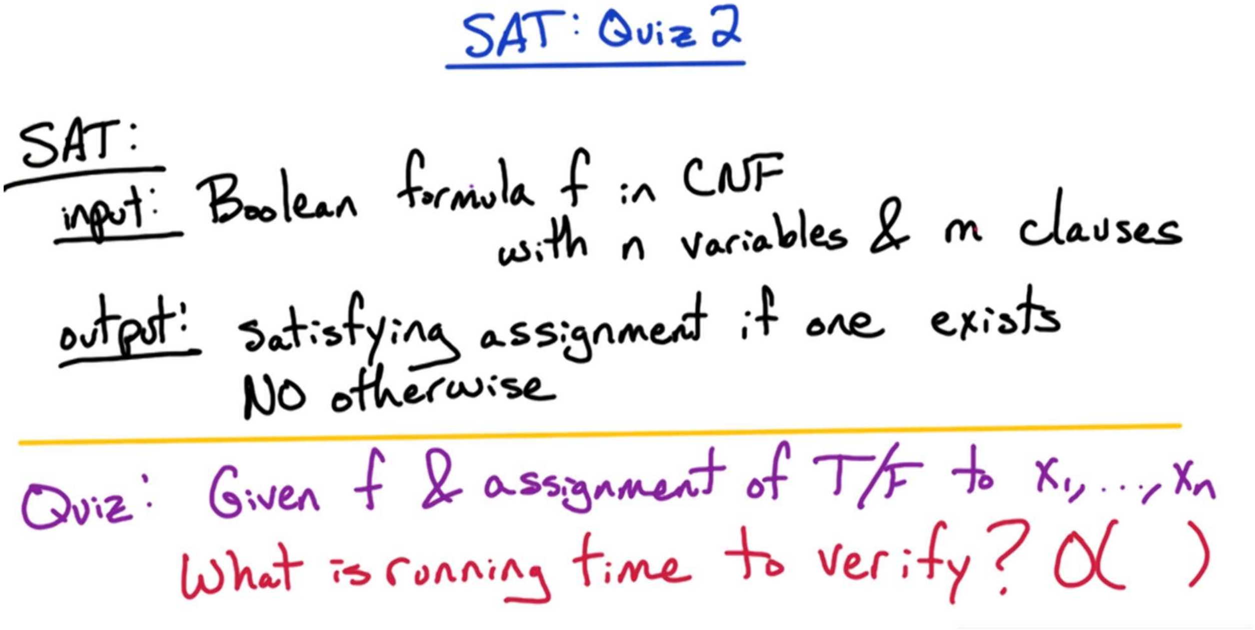
Quiz: Given f & assignment of T/F to $x_1, \ldots, x_n$
What is running time to verify? O( )

Now we just saw a particular input formula and we saw a satisfying assignment for it. And then we checked that that satisfying assignment was in fact satisfying this formula f. So, when we plugged in the assignments of True and False to the variables, the formula evaluated to True.

Now, let's look at the general problem. Suppose I give you a formula f and give you an assignment of True or False to each of the variables, what is the running to verify that the assignment actually evaluates this formula to be True? So, this assignment satisfies the formula. As a function of n and m, what is the running time to do that? Go ahead and write the running time in big-O notation.

***

NP1: Definitions: SAT in NP



SAT: Quiz 2

SAT:

input: Boolean formula f in CNF with n variables & m clauses

output: satisfying assignment if one exists
NO otherwise

Quiz: Given f & assignment of T/F to $x_1, \ldots, x_n$
What is running time to verify? $O(\quad)$

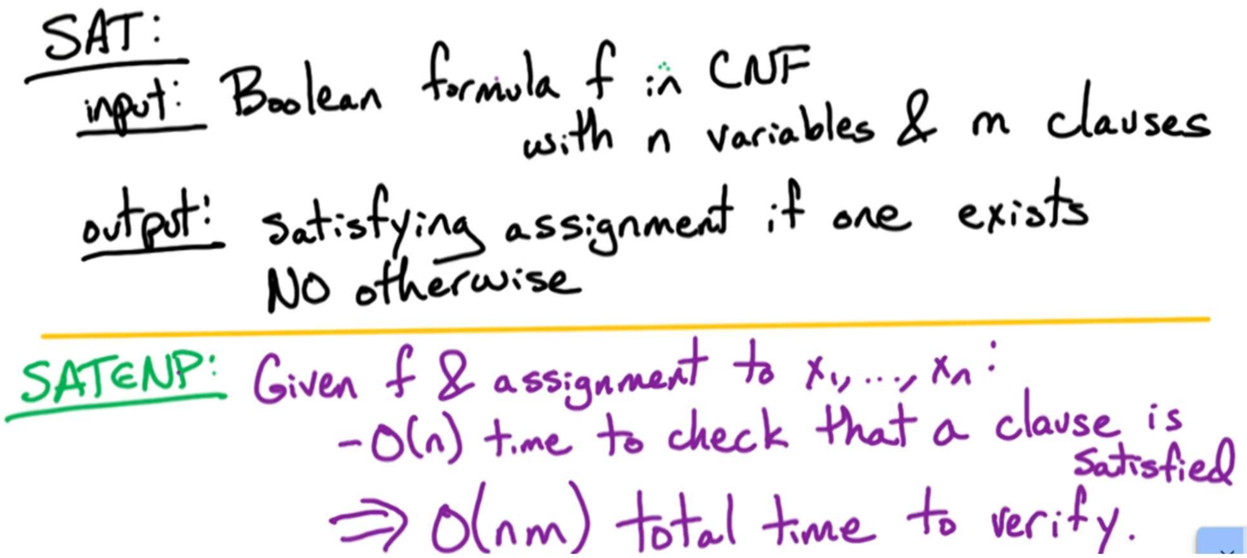
Question

Hint: How long does it take to verify if a formula with *n* variables and *m* clauses evaluates to True or False, given a specific assignment for each variable?

Give your answer in big O notation.

***

(Slide 8) SAT in NP (Answer)



SAT:

input: Boolean formula f in CNF with n variables & m clauses

output: satisfying assignment if one exists
NO otherwise

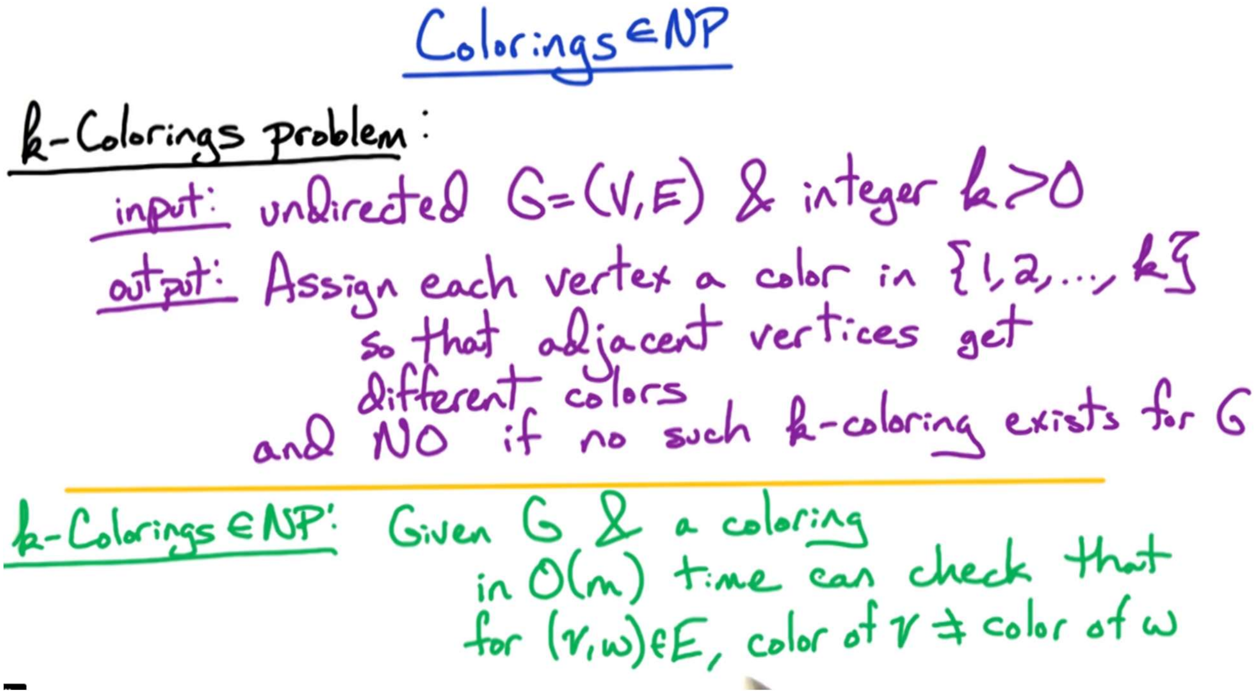
SAT∈NP: Given f & assignment to $x_1, \ldots, x_n$:
- $O(n)$ time to check that a clause is satisfied

$\Rightarrow O(nm)$ total time to verify.

Why do I care about the running time to verify a solution? -- well, this is a key step for proving that SAT is in NP. So, let’s go ahead and formally prove that SAT is in NP, and then as part of the proof, we’ll see the answer to this question.

***

(Slide 9) Colorings in NP



Colorings $\in$ NP

k-Colorings problem:

input: undirected $G=(V,E)$ & integer $k>0$

output: Assign each vertex a color in $\{1,2,\ldots,k\}$
so that adjacent vertices get
different colors
and NO if no such k-coloring exists for G

k-Colorings $\in$ NP: Given G & a coloring
in $O(m)$ time can check that
for $(v,w)\in E$, color of $v \neq$ color of w

Let's look at the k-colorings problem. This is a natural graph problem that many of you've probably seen before but I don't think we've looked at this particular problem in this course yet. You've probably heard of the four-color theorem. This is the formulation of the general problem.

The input to the problem is an undirected graph G and an integer K. K is a number of colors in our palette. Our goal is to assign colors to the vertices so their neighboring vertices have different colors. So, if there is a proper coloring, we're going to output an assignment of colors to the vertices. So, each vertex is going to get a color between one through K. These are the names of the colors and this is a proper coloring if adjacent vertices get different colors. Now, if this G has no such coloring, then we output NO. So, if there is a proper vertex K coloring for this graph G, then we output such a coloring. And if there is no such proper K coloring, then we output NO.

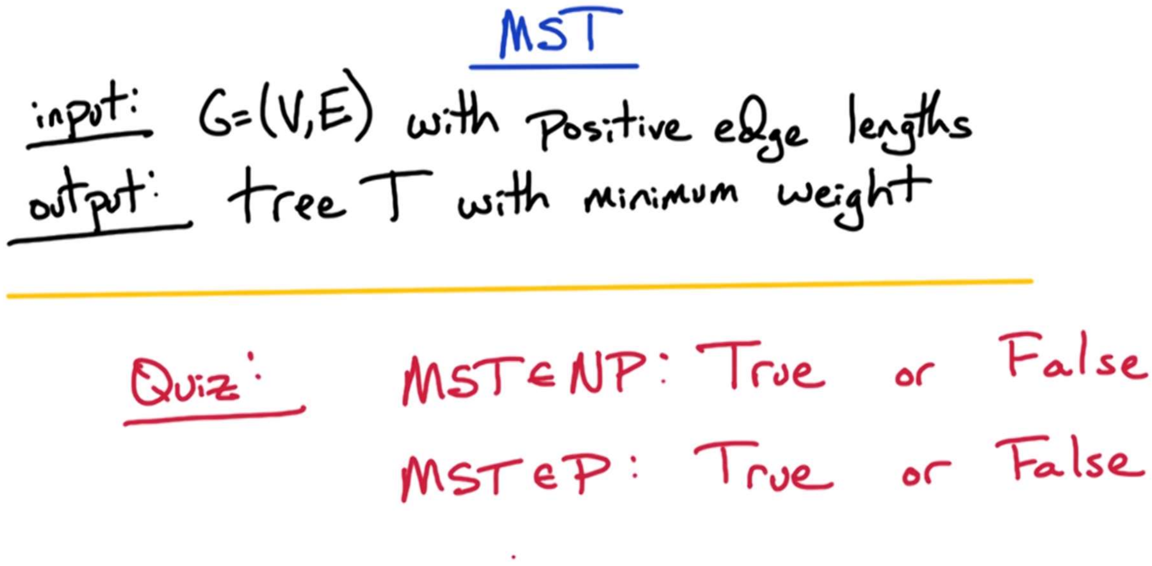
Now, the K-colorings problem is in NP. Why? Well, first off, it's the correct form when there is a solution. When there is a proper coloring, we output such a coloring. And if there is no such coloring, then we simply output NO. Now, the non-trivial step is that we have to show that we can verify solutions.

So, suppose that were given a graph G and we were given a K-coloring for this graph. So, we're given assignment of the vertices to each of the one of the K colors. Now, how do we check that this coloring is a proper coloring? Well, we go through the edges of the graph and we check that each edge is not monochromatic.

In particular, in O(m) time (which is the number of edges), we can check for every edge (v,w) that the color assignment for v is different than the color assignment for w. So, for every edge, the two endpoints have different colors. That's it. That's all we have to do to show that the K-colorings problem is in NP. We simply have to show an algorithm which can verify solutions in polynomial time. And it's a trivial algorithm, so it's a simply one sentence proof that the K-colorings problem is in NP.

***

(1st Slide 10) MST



MST

input: G=(V,E) with Positive edge lengths

output: tree T with minimum weight

Quiz: $MST \in NP$: True or False

$MST \in P$: True or False

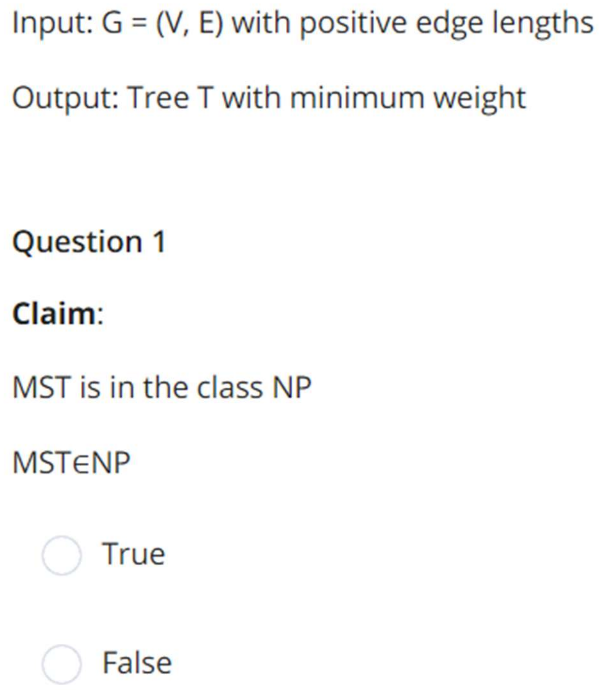
Let’s recall the MST problem. The input to the problem is an undirected graph G with positive edge lengths, and the output is a tree T with minimum weight.

Now, let’s test your knowledge of P and NP. So, the first question is “Is the MST problem in NP?”   Mark whether this statement is True or False.

The second question is whether the MST problem is in the class P.   Is that statement True or False?

***

NP1: Definitions: MST



Input: G = (V, E) with positive edge lengths

Output: Tree T with minimum weight

Question 1

Claim:

MST is in the class NP

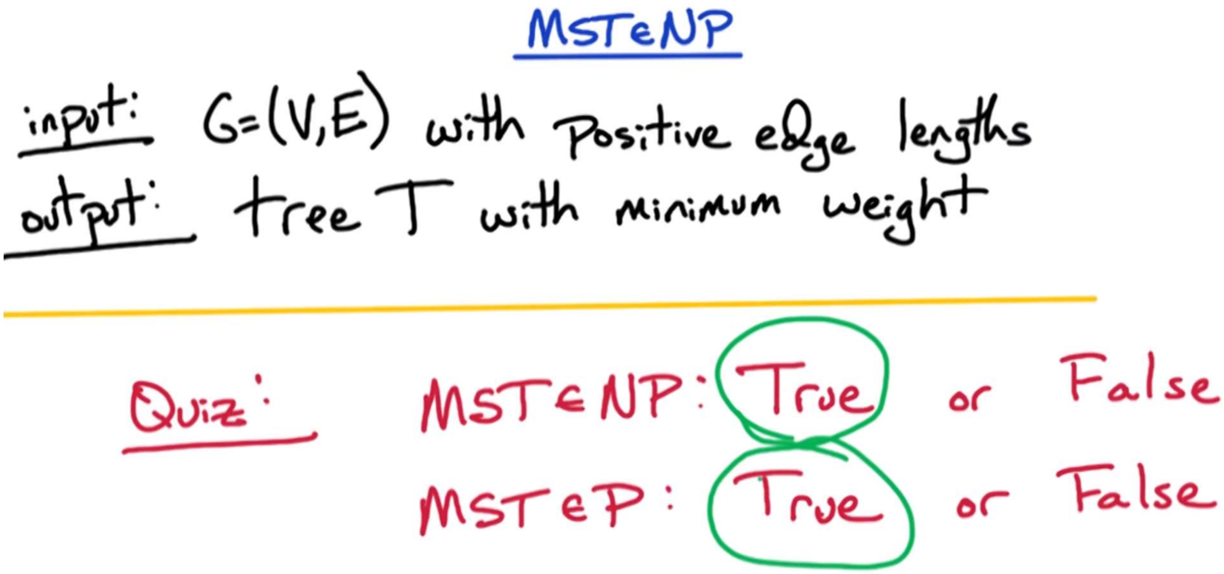
MST∈NP

○ True

○ False

***

(2nd Slide 10) MST (Answer)



MST∈NP

input: G=(V,E) with Positive edge lengths

output: tree T with minimum weight

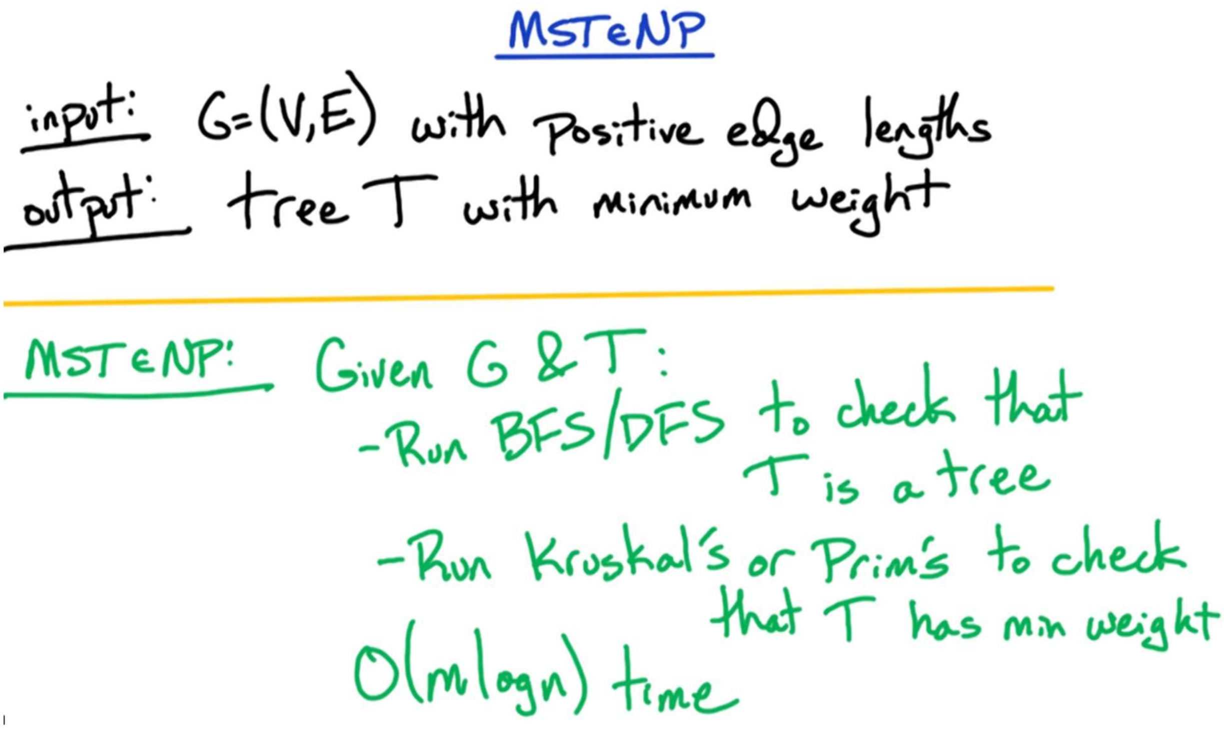
Quiz: MST∈NP: [✓ True] or False

MST∈P: [✓ True] or False

Well, in fact, both statements are True. The MST problem is in the class NP, and the MST problem is, in fact, in P. Let's go ahead and see why both of these are True.

***

(Slide 11) MST in NP



MST $\in$ NP

input: $G=(V,E)$ with Positive edge lengths

output: tree T with minimum weight

MST $\in$ NP: Given G & T:

-Run BFS/DFS to check that T is a tree

-Run Kruskal's or Prim's to check that T has min weight

$O(m \log n)$ time

Let's see why the MST problem lies in the class NP. First off, is the MST problem of the correct form? If there is a solution are we outputting a solution? And if there is no solution, we're outputting NO? Well, the way the problem is formulated, we're always outputting a solution. We're always outputting a tree T, which is guaranteed to be a minimum weight. And the way the problem is formulated. there always is a solution. We're simply looking for the tree of minimum weight. So there was always a tree of minimal weight, so we're always outputting a solution. There's never a NO instance. There's never no tree of minimum weight. Some tree has to be of minimum weight. So there's always a solution to this problem and we're always outputting a solution.

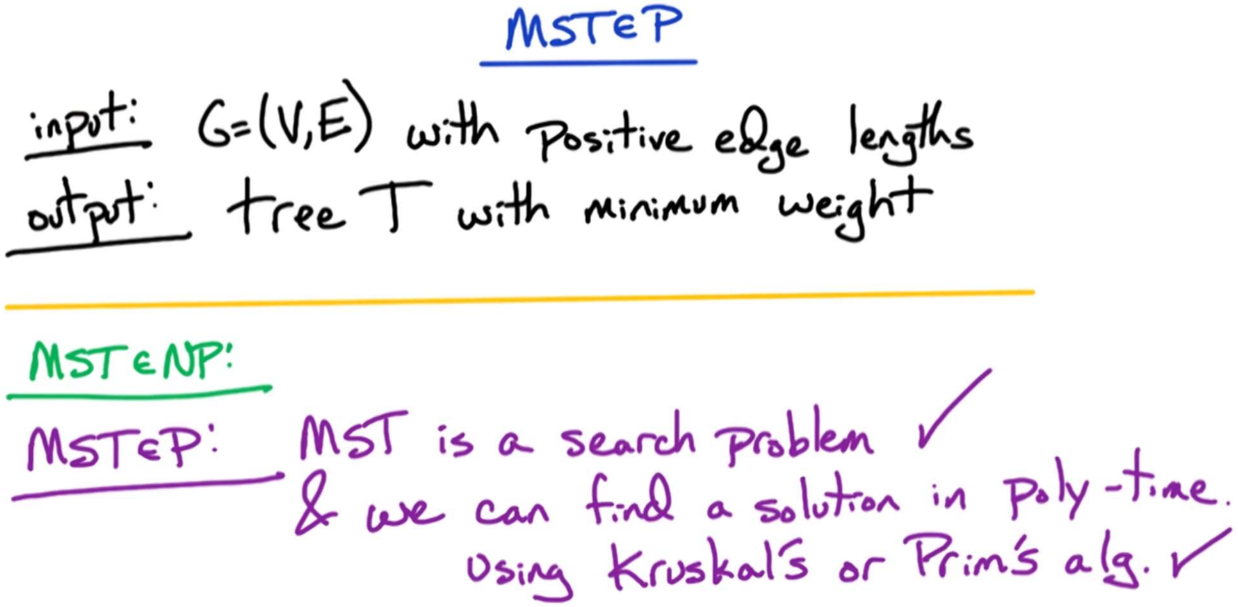
Now, suppose we're given a graph G, which is our input to the MST problem and we're given a tree T, which we claim is a solution to the MST problem. How do we verify that T is in fact a solution to the MST problem for G? Well, first we have to verify that T is a tree. Or, we can simply run the BFS or DFS algorithm to check that T is a tree. To check that T connects up the graph G and that T has no cycles.

Now, how do we check that T is of minimum weight? Well, we can run Kruskal's or Prim's algorithm, that's going to give it an MST. Then, for that given MST that's outputted by

Kruskal's or Prim's algorithm, we check its weight and then we check that T has the same weight as that output from Kruskal's or Prim's algorithm. There's no reason why Kruskal's or Prim's algorithm might generate this particular tree T, but they're going to output a MST and we checked the weight of that MST outputted by these algorithms, and then we compare that to the weight of the tree T. And therefore, we can verify that T has minimum weight and that T is a tree. And therefore, T is an MST. It takes us O(n+m) time to run BFS or DFS and it takes us O(mlogn) time to run Kruskal's or Prim's algorithm. So the total time of our verification algorithm is O(mlogn). This shows that the MST problem is in NP.

***

(Slide 12) NST in NP



MST $\in$ P

input: $G=(V,E)$ with positive edge lengths

output: tree T with minimum weight

MST $\in$ NP:

MST $\in$ P: MST is a search problem ✓
& we can find a solution in poly-time.
using Kruskal's or Prim's alg. ✓

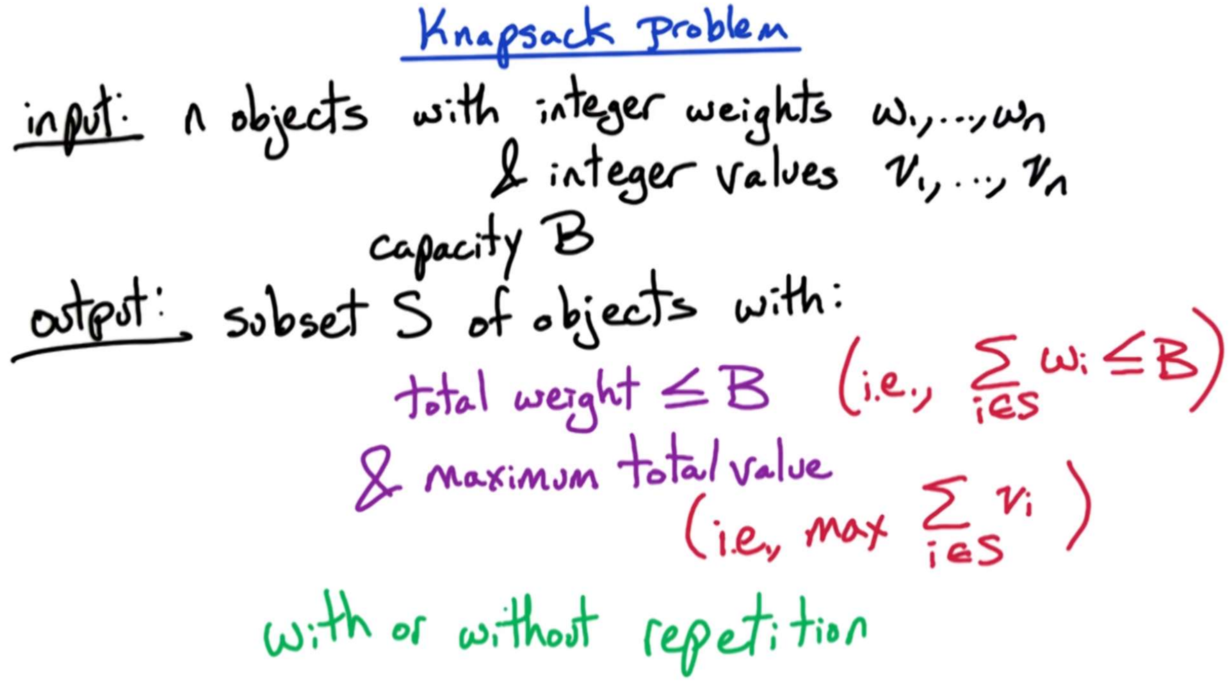
Now, we just saw that the MST problem is an NP. So, we've seen that MST is a search problem. Now, let's prove that MST is in P. And to show that the MST problem is in P, there's two parts.

First, we have to show that the MST problem is a search problem. Well, that's what we just did to show that MST is an NP. So we've done that first part.

The second part is to show that we can find a solution in polynomial time. How do we find a solution in polynomial time? We simply use Kruskal or Prim's algorithm. This shows the second part. So this completes the proof that MST is NP.

***

(Slide 13) Knapsack Problem



Knapsack Problem

input: n objects with integer weights $w_1, \dots, w_n$

& integer values $v_1, \dots, v_n$

capacity B

output: subset S of objects with:

total weight $\le B$ (i.e., $\sum_{i \in S} w_i \le B$)

& maximum total value

(i.e., $\max \sum_{i \in S} v_i$)

with or without repetition

We've looked so far at the satisfiability problem, SAT, the colorings problem, and MST (minimum spanning tree) problem. And we've seen why each of these are in the class NP. Now let's take a look at the Knapsack problem which we studied in the dynamic programming section. Let me remind you about the formulation of the knapsack problem.

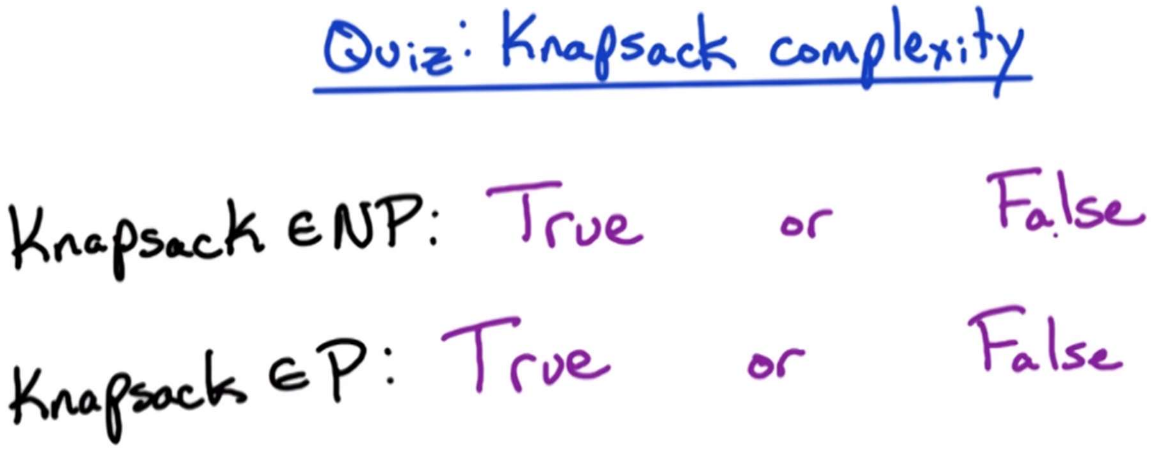
Now, the input to the knapsack problem are n objects. For each object, we’re given an integer weight, and we are given the integer values. The weights are denoted as w_1 thru w_n, and the values are denoted by v_1 thru v_n. And in addition, we're given a total capacity B for the backpack or knapsack. Now the output for the knapsack problem is a subset of objects which fit in the backpack, so their total weight is at most B. And we want the subset which maximizes the total value.

Let's make this a little bit more mathematically precise. By “total weight at most B”, we mean that if you sum over the objects in the subset S, their weights sum up to at most B. Our objective is to maximize the total value. The total value for a subset S is the sum (over the objects in the subset) of their values. And we're trying to find the subset S which maximizes this sum of the total value, subject to the constraint that the sum of their weights is at most B.

Now if you recall there were two variants of the knapsack problem that we studied in dynamic programming section. There was the version with repetition – So, there was unlimited supply of each object. Or, there was a version without repetition – So, we had at most one copy of each object. Now which of these two variants should you consider? It doesn't matter. For the following discussion, it holds for both of these versions.

***

(Slide 14) Knapsack Complexity



Quiz: Knapsack complexity

Knapsack ∈ NP: True or False

Knapsack ∈ P: True or False

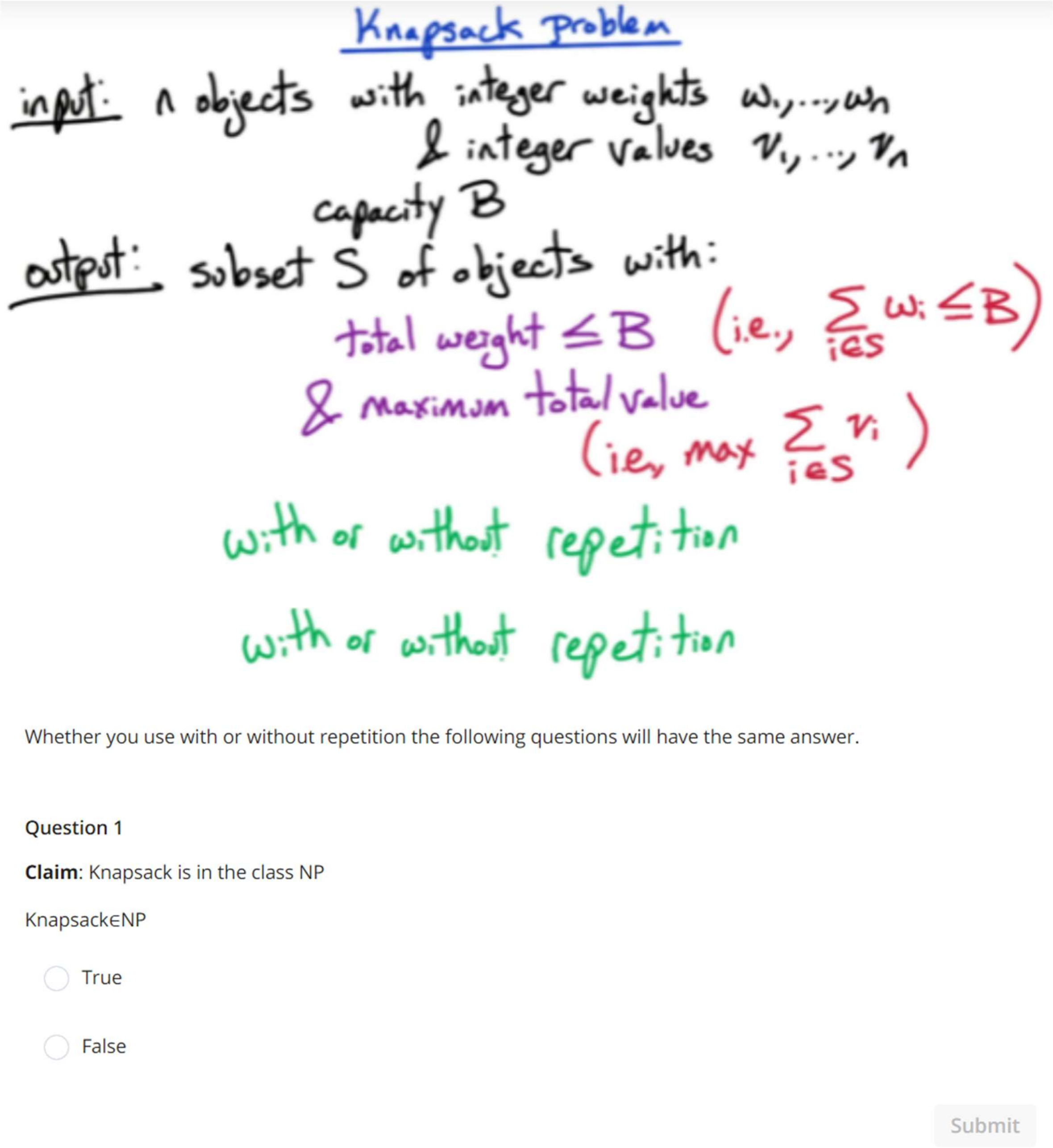
For the Knapsack problem that we just formulated, let’s address the following two questions:

- Knapsack ∈ NP?
- Knapsack ∈ P?

For each of these statements, why don’t you mark whether it is True or False?

***

NP: Definitions: Knapsack



Knapsack Problem

input: n objects with integer weights $w_1, \dots, w_n$
& integer values $v_1, \dots, v_n$
capacity B

output: subset S of objects with:
total weight $\leq B$ (i.e., $\sum_{i \in S} w_i \leq B$)
& maximum total value
(ie, max $\sum_{i \in S} v_i$)

with or without repetition

with or without repetition

Whether you use with or without repetition the following questions will have the same answer.

Question 1

Claim: Knapsack is in the class NP

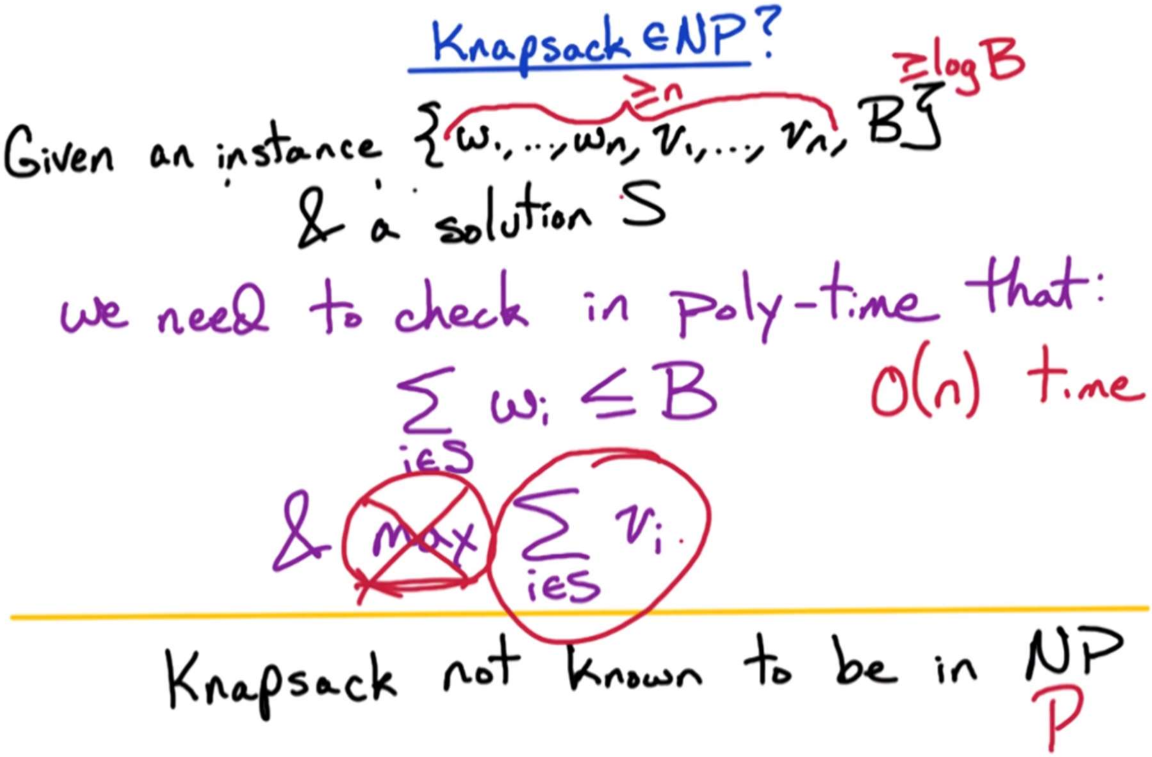
Knapsack$\in$NP

○ True

○ False

***

[Slide 15] Solution: Knapsack Complexity



Knapsack ∈ NP?

Given an instance $\{\overbrace{w_1,\dots,w_n, v_1,\dots,v_n}^{\geq n}, \overset{\geq \log B}{B}\}$

& a solution S

we need to check in poly-time that:

$\sum_{i\in S} w_i \leq B$ O(n) time

& $\cancel{\max} \sum_{i\in S} v_i$ ..

Knapsack not known to be in NP

P

Let's first address whether a knapsack problem lies in the class NP or not. In order to lie in the class NP, we have to be able to verify solutions. We have to be able to do the following efficiently:

> We have to take a particular input to the knapsack problem. An input is given by a set of weights, a set of values, and a total capacity to the for the backpack, and we have to take a solution for that input.
> Now, in polynomial time, can we check whether this solution is in fact correct for this input?

In order for S to be a solution for this input, what do we need to check?

- Well, we have the constraint that the total weight of the objects in this solution is at most B so we have to check that the sum of objects in this subset - of the individual weights - is at most B. Now, this is quite straightforward to do. We're just summing up at most n numbers so this takes at most O(n) time.

- The second thing that we need to check is that this subset is optimal in the following sense: that it maximizes the sum of the values.

It's easy to check what the total value is, we just sum up the values, but how do we check that has maximum total value? How do we check that the value of this subset is better or at least as good as any other subset?

Now, for the MST problem, we could check that it has optimal value. It had minimum weight tree by running a polynomial time algorithm such as Kruskal's or Prim's. We checked the output of that algorithm and we look at the total weight of the tree produced and we compare that to the weight of the tree that we're considering in our proposed solution.

Now, we had an algorithm for the knapsack problem that we looked at and the dynamic programming section. The running time of our dynamic programming algorithm was O(nB). The problem is that this running time is not polynomial time. It's not polynomial in the input size.

Recall the input of size at least n to represent these n objects. And, how big is the representation of this one number B? Well, that takes log B bits. In order to be polynomial in the input size, we need an algorithm which runs in time polynomial in n and log B.

So, the dynamic programming algorithm is actually exponential in the input size. We need a way to efficiently check solutions, so we can't run that dynamic programming algorithm because that's exponential time.

So, we have no way of verifying that our proposed solution is optimal - that it's a maximum total value. So, what is our conclusion? Is the knapsack problem in NP or not? Well, the final statement is a bit subtle. If you said that True, Knapsack is in NP, that is incorrect. We don't know how to show that knapsack is in NP.

Now is Knapsack not in NP? Well we can't prove that at the moment. If we could prove that knapsack is not in NP, that would imply that P is not equal to NP. It could be the case that there is a polynomial time algorithm for the Knapsack problem and then we could check this maximization in polynomial time.

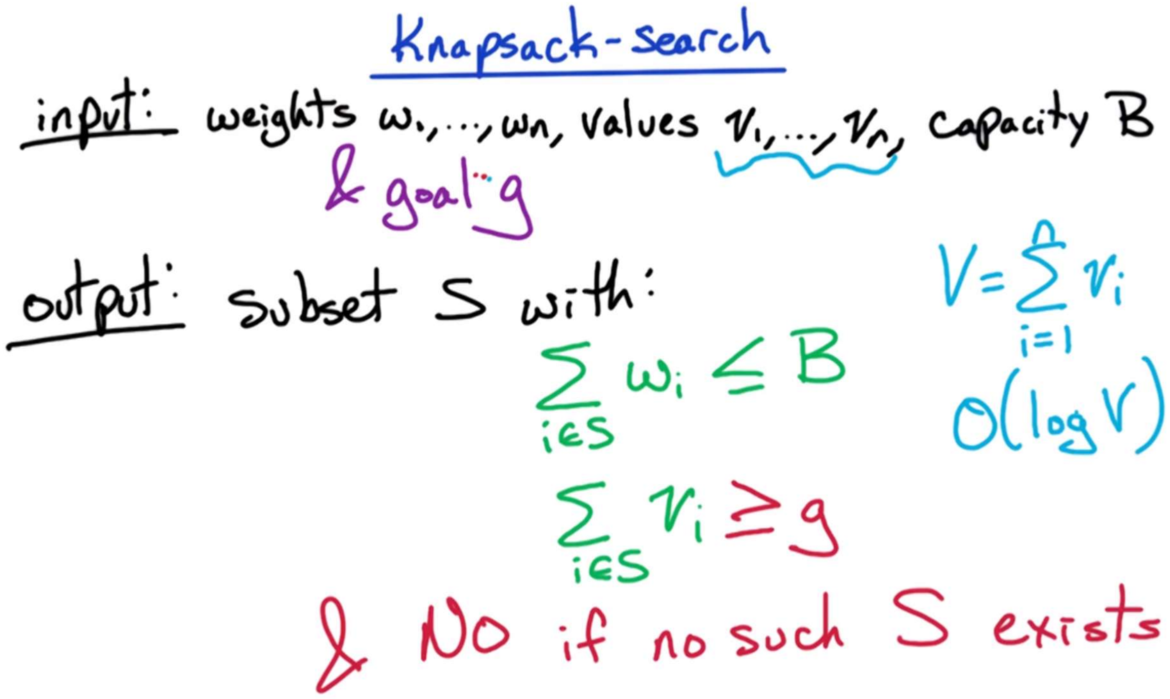
What can we say? We can say that if you said Knapsack is in NP, that is incorrect and as for a Knapsack not being in NP, well, we don't know whether or not that's true. So the best we can say is that as far as we know right now Knapsack is not known to be in the class NP. Similarly, if you ask for the class P, well, as far as we know, Knapsack is not known to be in the class P.

Could it be that knapsack is in P? Sure. it might be. There might be a polynomial time algorithm for Knapsack; P might equal NP, in which case, Knapsack will lie in NP, but as of right now, Knapsack is not known to be in the class P.

There is a simple variant of the Knapsack problem, which is in the class NP. Let's take a look at that variant. What we're going to have to do is drop this optimization part and we're going to add in another input parameter. That additional input parameter is going to be our goal for the total value and then we're going to check whether our total value of our subset is at least the goal. Then, we can do binary search on that additional parameter, the goal for the total value. Let's go ahead and formalize that now.

***

(Slide 16) Knapsack Search



Knapsack-search

input: weights $w_1, \ldots, w_n$, values $\underbrace{v_1, \ldots, v_n}$, capacity B

& goal g

output: subset S with:

$$\sum_{i \in S} w_i \le B$$

$$\sum_{i \in S} v_i \ge g$$

& NO if no such S exists

$$V = \sum_{i=1}^{n} v_i$$

$$O(\log V)$$

Now, we're going to look at the version of Knapsack which is a search problem. The input starts off the same as before. So, we'll have integer weights w1 thru wn, integer values v1 thru vn and a total capacity, B. All of these are the same as before for the original Knapsack problem; but, there's an additional input parameter which is our goal which will denote as g. Our output is a subset, just as before, where the total weight of this subset S is at most B. So, if I sum over the objects in the subset of their individual weights that's at most B. That's just as before and we look at the total value of this subset.

Previously, we were trying to maximize the total value, but now we're going to just check whether this total value meets our goal. How does it meet our immediate goal? Well, we just want that the total value is at least g. We're trying to maximize the total value. So, we wanted to be “at least” our goal. If we made it “at most” our goal, that would be easy. That would be trivial. We could just output the empty set. And if we do better than our goal that's even better. That's why we make it at least g.

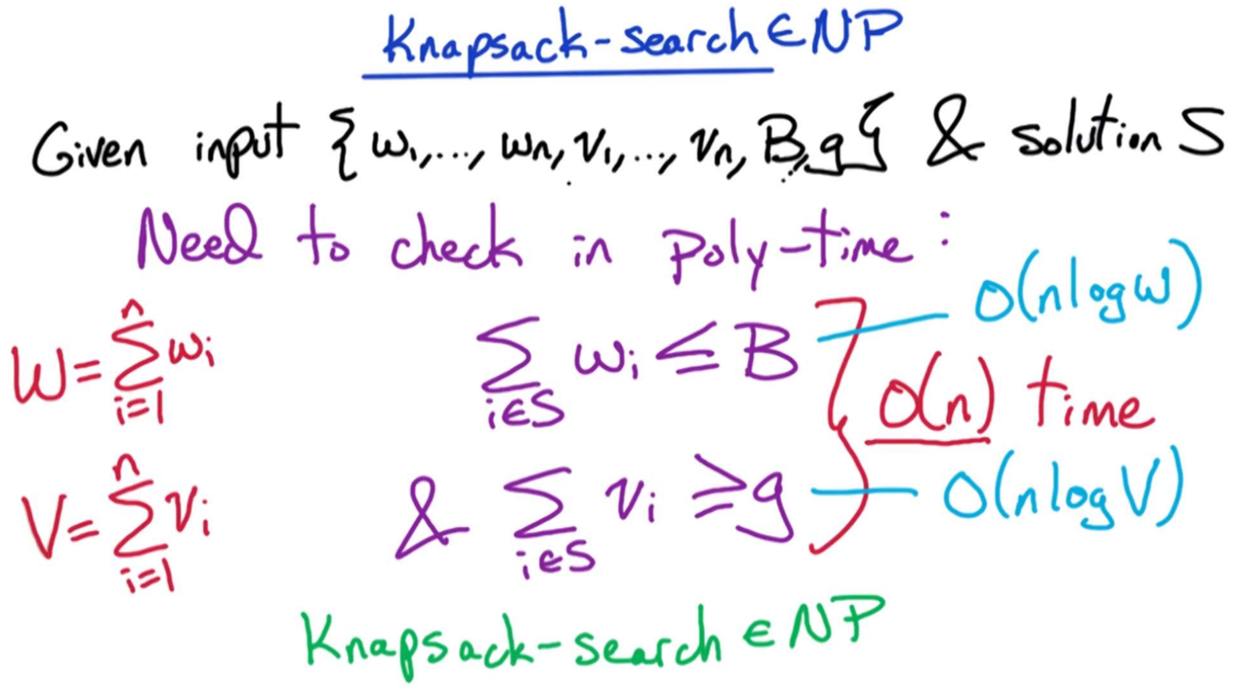
But, now, what happens if there's no subset which meets our goal? Well, if there's no solution to this problem, so there's no subset which meets this goal, then we simply output "NO".

This is the correct form for our search problem. Notice that if we can solve this Knapsack search version in polynomial time, then we can solve the original optimization version in polynomial time. How? Well, in the original version we were maximizing this sum of the total value. Here we're simply finding a subset whose total value is at least little g. Now, suppose that we could solve this version in polynomial time. How do we solve the maximization version? Well, we can simply do binary search over g and by doing binary search over g, we find the maximum g which has a solution, and then that tells us the maximum total value which we can achieve.

How many rounds in our binary search are we going to have to run? How many of these Knapsack search version problems are we going to have to run in order to solve the optimization problem? Well, let's look at the sum of these values. Let's let capital V be the sum of all the values. Certainly the max of this total value that we can achieve is at most capital V. So, we're going to do binary search on little g ranging between one and capital V. So, the total number of rounds in our binary search algorithm is going to be at most O(log V). What is the size of the input to represent these numbers? Well, it's log V. So, this algorithm, this binary search algorithm, is polynomial in the input size. So, if we can solve the Knapsack search version in polynomial time, then we can solve the optimization version in polynomial time.

***

(Slide 17) Knapsack Search in NP



Knapsack-search $\in$ NP

Given input $\{w_1,\ldots,w_n, v_1,\ldots,v_n, B, g\}$ & solution S

Need to check in poly-time:

$W=\sum_{i=1}^{n} w_i$ $\qquad \sum_{i\in S} w_i \le B$ — $O(n\log W)$

$O(n)$ time

$V=\sum_{i=1}^{n} v_i$ $\qquad$ & $\sum_{i\in S} v_i \ge g$ — $O(n\log V)$

Knapsack-search $\in$ NP

Now, this new version of the Knapsack problem does lie in the class NP. We can prove that now.

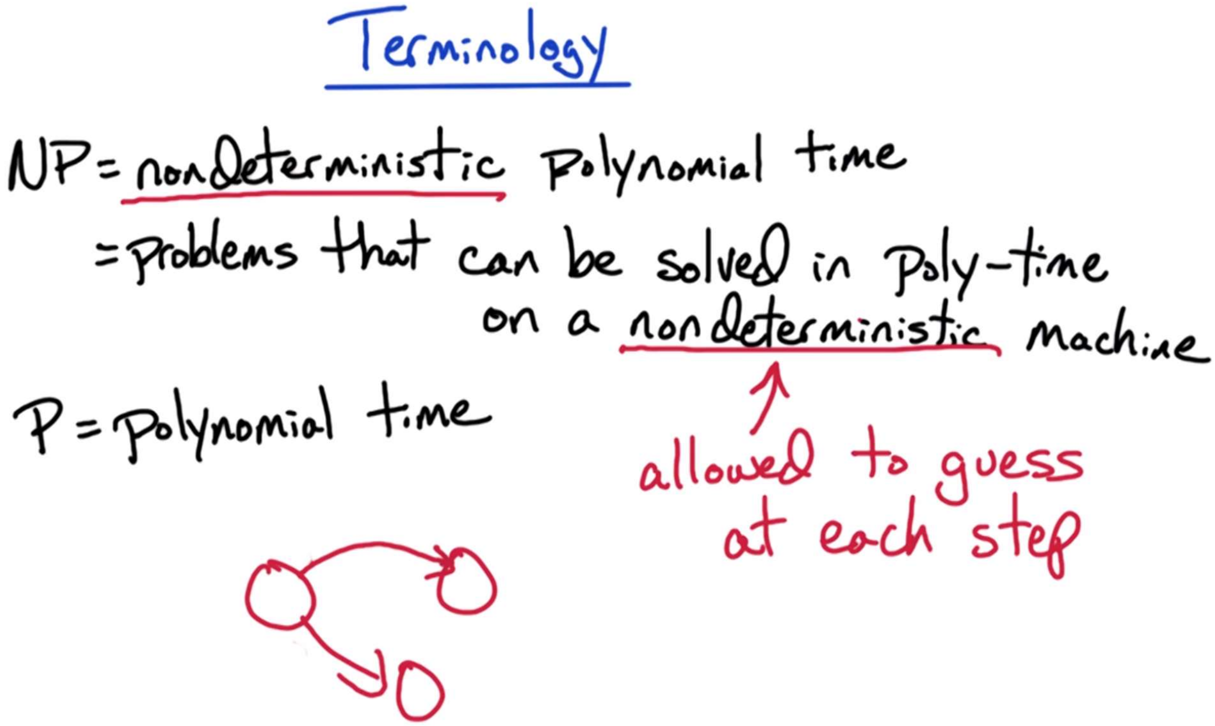
So, consider a particular input to the Knapsack search version. We have the integer weights, integer values, total capacity, and our goal, and consider a particular solution. In order to check that this is, in fact, a solution, what do we need to do? We need to check that the total weight is at most capital B. We saw how to do that before. That's easy. We're just summing up at most O(n) numbers. And, now, instead of checking that it has maximum total value, we just have to check that the total value is at least a little g. This again involves just adding up O(n) numbers. So we can check these two in O(n) time.

We can check a proposed solution in polynomial time. Therefore, this version of Knapsack, the Knapsack search version does, in fact, lie in the class NP. Now, let me just be a little bit more precise about one aspect which may be confusing some people.

You may ask, "Does this really take O(n) time to sum these numbers?" Well, if you really look at it in terms of the magnitude of these numbers, we have V as the sum of all values. Let’s let W denote the sum of all weights. Certainly, to add up all these numbers, the time required is at most the number of bits. The number of bits in each of these numbers is at most log(W). So, the time required to compute this sum is at most O(n logW). To compute the sum of all values is at most O(n logV). Since the input size is log(W) (weights) and log(V) (values), this (pointing to the O(n log W) and O(n logV) time estimates) is still polynomial in the input size. So it is correct that Knapsack-search version does lie in the class NP. We just verified this.

***

(Slide 18) NP acronym for?



Terminology

NP = nondeterministic Polynomial time

= problems that can be solved in Poly-time on a nondeterministic machine

P = Polynomial time

allowed to guess at each step

Now, we have these two complexity classes we'll consider so far, P and NP. What do these abbreviations stand for?

- P stands for polynomial time. Why? Because this is the class of search problems that can be solved in polynomial time.
- Now you might think that NP stands for not polynomial time. That's incorrect.

  It may be the case that we can solve NP problems in polynomial time as well. It may be the case that P = NP. We don't know.

  > So what does NP stand for? NP stands for **nondeterministic polynomial time**. What does nondeterministic mean? This is a class of problems that can be solved in polynomial time on a nondeterministic machine.
  >
  > That doesn't do much good, I still haven't defined this term non-nondeterministic.

  By nondeterministic, I mean a machine where I'm allowed to guess at each step and there is a series of guesses. And, there is a series of choices which leads to an accepting state.

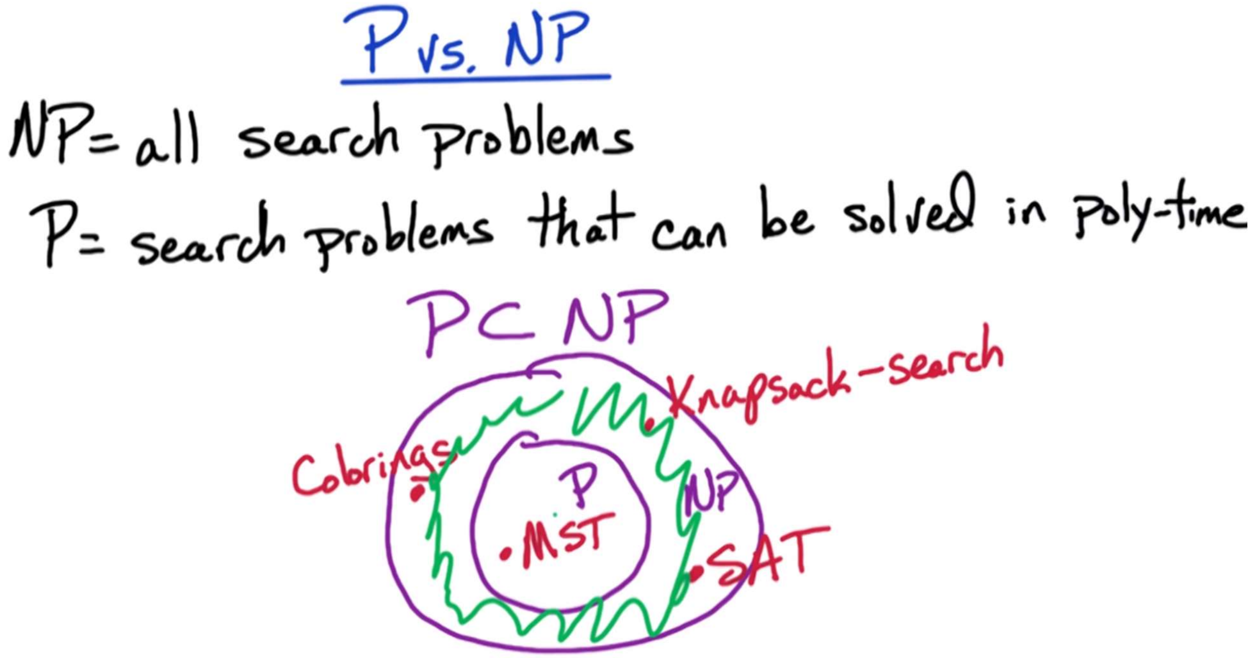
You think of it in terms of an automaton. Typically for an automaton, I specify a particular input, say 1. I move from a first state to this (second) state, and maybe on a particular input 0, I moved move from the first state to this other state.

But now, I don't need to specify for which particular input, I move from this state to this other state. I just specify that sometimes I might move from this state to this state and sometimes, from this state to this other state. And for the case of an automaton, I just need that there is a path in this automaton that goes to an accepting state. So there is a choice of branchings which leads me to an accepting state.

I can't really explain it much more than what I just did, because to be honest, I don't understand it much more than that. But the important thing to remember is that NP does not stand for not polynomial time. What it stands for is nondeterministic polynomial time.

***

(Slide 19) P vs NP



P vs. NP

NP= all search problems

P= search problems that can be solved in poly-time

$P \subset NP$

Let's go back and have a somewhat philosophical discussion on the P verses NP question.

Recall that NP stands for the class of all search problems. Meanwhile, P is the class of those search problems that can be solved in polynomial time. So, every search problem lies in NP and P are those search problems which can also be solved in polynomial time. So P has an extra restriction. So, P is a subset of NP.

So, if we look at it graphically, we have this class of problems in P, and then we have this class NP which is a superset of P. Now, we have some examples of problems which lie in P. For example, the MST problem that lies in P and it lies in NP.

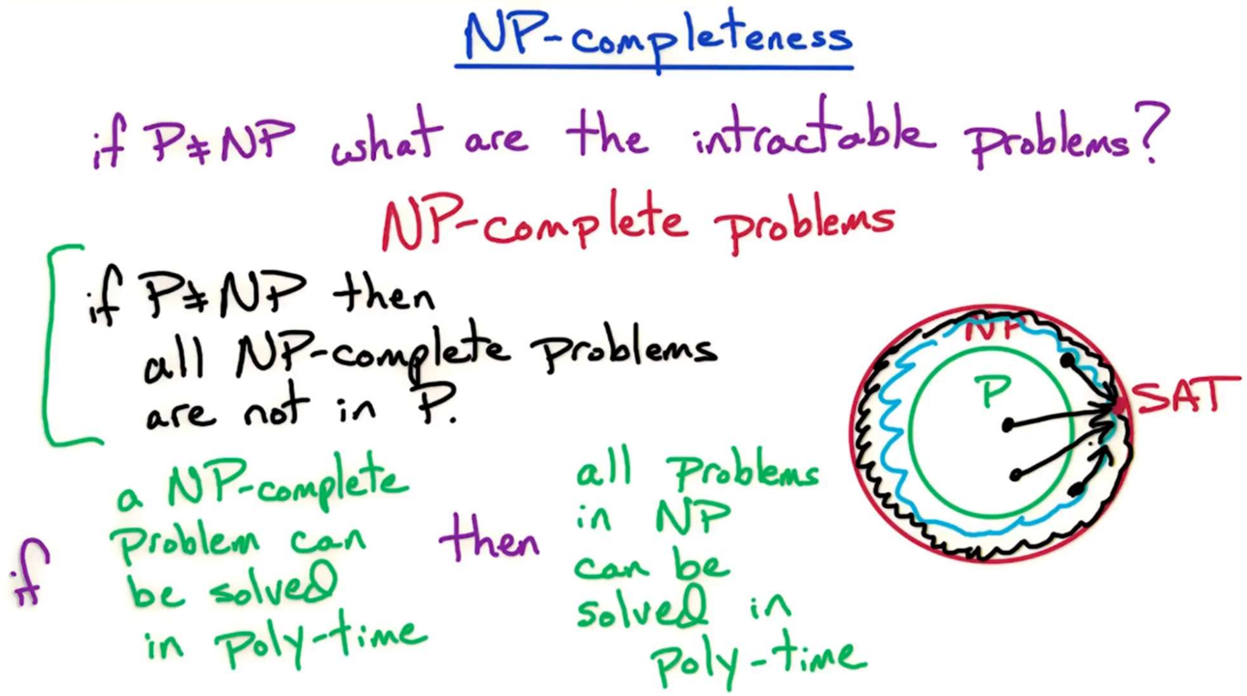
We just saw that the search version of Knapsack problem lies in NP. We saw that the satisfiability problem lies in NP. We saw that the colorings problems lies in NP.

Now, is there a separation here? Are there problems which lie in NP but do not lie in P? We don't know whether that's the case.

- [P = NP]: It may be that all these problems in this band actually lie in P - that these two classes are the same (P = NP).
- [P != NP]: Or, it may be the case that there are problems which lie in NP which do not lie in P - that means that P != NP, and there are some problems that we can not solve in polynomial time.

***

(Slide 20) NP Completeness



NP-completeness

if $P \neq NP$ what are the intractable problems?

NP-complete problems

if $P \neq NP$ then
all NP-complete problems
are not in P.

if a NP-complete problem can be solved in poly-time then all problems in NP can be solved in poly-time

I suppose that P is not equal to NP. Now, looking at our diagram from before, that means that this donut, this separation between P and NP is non-empty. Therefore there are some such problems which can't be solved in polynomial time. They don't lie in the class P.

What are the hard problems that lie in this donut, these intractable problems, these problems that we can't solve in polynomial time? The answer is NP-complete problems. Now there may be other problems that lie in this class NP which do not lie in P. So. they lie in this donut but NP-complete problems are guaranteed to lie in this donut, if it exists.

NP-complete problems are the hardest problems in the class NP. Let me repeat that because it's very key: NP-complete problems are the hardest in the class NP. Now, what exactly do we mean by the hardest in the class? We mean that if P is not equal to NP then the following holds: Then, all the NP complete problems are not in the class P. So, all NP-complete problems cannot be solved in polynomial time if P is not equal to NP.

So if P is not equal to NP, this donut is not empty - there are some problems in NP which don't lie in P, so we can't solve them in polynomial time. And what does NP-complete mean? Well, these are the hardest in the class NP, so you can think of them as sort of as the outer layer of this donut. This outer layer are the NP-complete problems. If there is some separation between these sets, then these NP-complete problems are guaranteed to be in the difference.

Now, let's look at this statement more carefully. An equivalent statement is the contra-positive of it. Now, what's the complement of “all NP-complete problems are not in P”? Well, the compliment is that there exists an NP-complete problem in P. In other words a NP-complete problem can be solved in polynomial time. So, the complement of this conclusion that all NP-complete problems are not in the class P is that there exists an NP-complete problem that can be solved in polynomial time. If this holds, then the complement of the hypothesis holds. The complement of the hypothesis is that P equals NP. If P equals NP, that means that all problems in the class NP can be solved in polynomial time. An equivalent statement to this one is that if there exists an NP-complete problem that can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

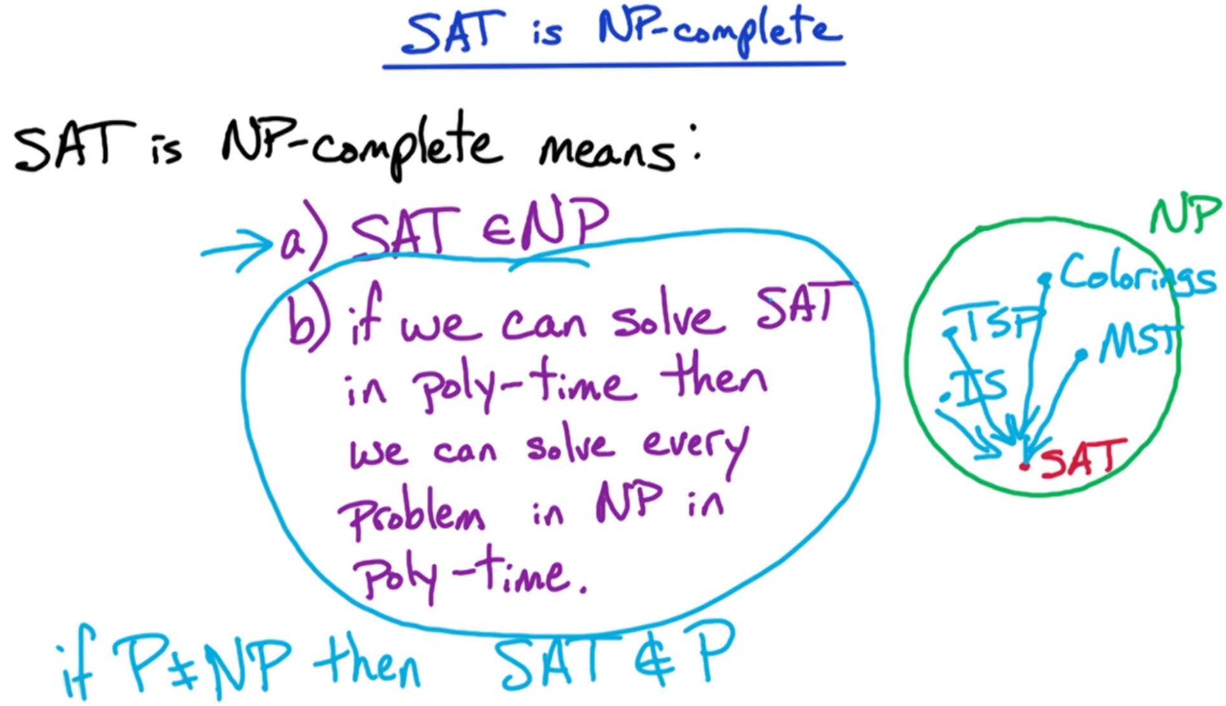
Hence to show that a problem such as SAT lies in this, the complete portion of the donut, we have to show that if there is a polynomial time algorithm for SAT, then there is a polynomial time algorithm for all problems in the class NP. We have to show that if there is an algorithm for SAT which works in polynomial time, we can use it as the black box to solve all problems NP in polynomial time.

To do that, we're going to take every problem in the class NP - now, there are many problems in this class - I've just marked a few of them; but, we have to take all such problems. So we take all such problems and we have to show that each of these such problems have a reduction to SAT.

If we can solve SAT in polynomial time, we can solve every such problem in the class NP in polynomial time. Let's look at this more precisely on the next two slides.

***

(Slide 21) SAT is NP-Complete



SAT is NP-complete

SAT is NP-complete means:

a) SAT $\in$ NP

b) if we can solve SAT in poly-time then we can solve every problem in NP in poly-time.

if P $\neq$ NP then SAT $\notin$ P

What precisely does it mean for a problem to be NP-complete? Let's define it now and let's use SAT as our example.

So, what does it mean for SAT to be NP-complete? SAT is NP-complete means the following:

- SAT ∊ NP: First off, we need that SAT lies in the class NP. So if we think as a class of problems that lie in NP, then the first requirement is that SAT lies in this class NP. So, we have checked that for the SAT problem - we can efficiently verify solutions. And, we've seen how to do that before. We saw how to prove that SAT lies in the class NP.
- Now the second requirement is that SAT is the hardest problem in this class. What exactly do we mean by that? Well, we're looking at these problems in terms of whether we can efficiently solve them or not. So for SAT to be the hardest in this class, we mean that it's the least likely to have an efficient solution.

  What's another way of saying that it's the least likely to have an efficient solution? Is that if we have an efficient solution for SAT, then we have an efficient solution for every other problem in NP.

So let us write this down. So if we can solve SAT in polynomial time then we can solve every problem in NP in polynomial time. What does this mean? This means for every problem in NP, for example:

- the MST problem,
- the colorings problem,
- there are other problems such as the TSP Traveling Salesman Problem,
- there's the Independent Set problem which we'll see shortly.

For all these problems in NP, there is a reduction or transformation to the SAT problem.

So we can take an input for the MST problem and we can transform it to an input to the SAT problem. And then if we have an algorithm which efficiently solves SAT - so it solves a SAT problem in polynomial time - then we can use that algorithm to solve MST.

How? We take an input for the MST problem, we reduce it to an input for the SAT problem, and then we run our algorithm on the SAT problem and that gives us an output - a solution to the MST problem. And we can reduce all the problems in NP to SAT.

Therefore, if we have an algorithm to solve SAT in polynomial time, we get an algorithm for all of the problems in NP. Now the point is that if P is not equal to NP, then we know that there are some problems in NP which cannot be solved in polynomial time. And therefore, we can't solve SAT in polynomial time because, if we could solve SAT in polynomial time, then we can solve all of the problems in NP in polynomial time. Since there are some problems in NP that we can't solve in polynomial time under the assumption that P is not equal to NP, then therefore SAT is not in polynomial time.

So SAT is not in the class P. So, this means that SAT is the computationally most difficult problem in the class NP. So if you believe that P is not equal to NP, then there is no polynomial time algorithm for SAT. Or alternatively, if you believe that nobody knows how to prove that P equals NP, then nobody knows a polynomial time algorithm for SAT. So it's a reasonably fair assumption that as of right now there is no polynomial time algorithm for SAT.

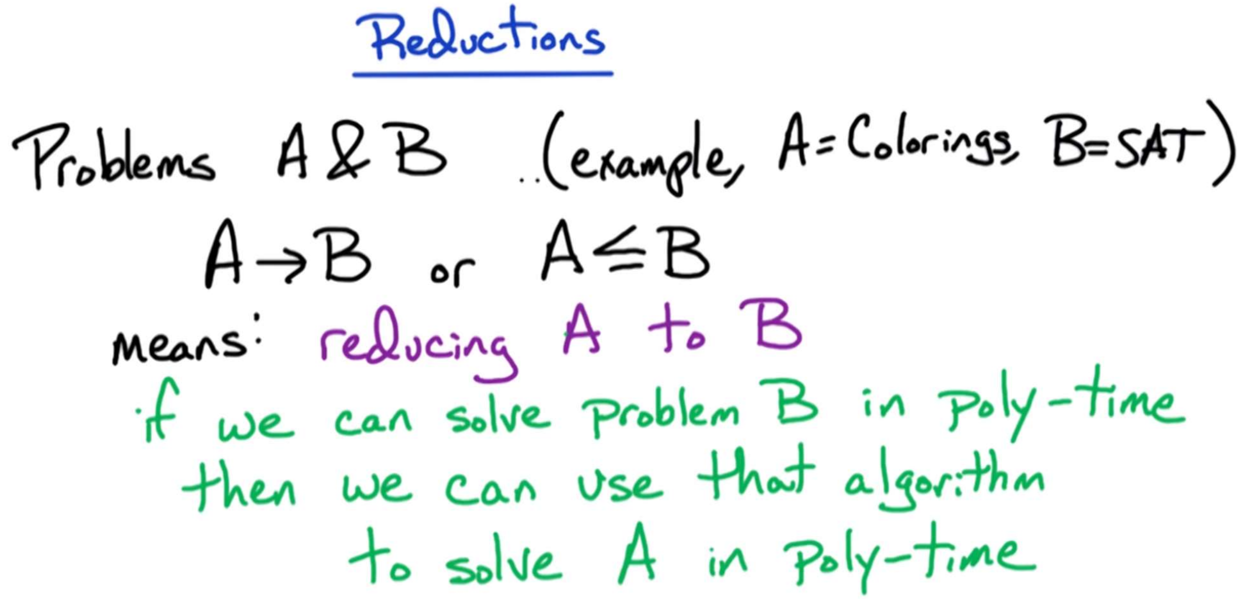
Why? Because if somebody had a polynomial time algorithm for SAT, then they could prove that P equals NP. And, if they can prove that P equals NP, they could get a fair bit of money from some mathematical prizes, and they get some fame from proving that P equals NP. They get a Fields Medal and so on.

Now let's formalize this a bit more. We know how to do this step (SAT $\in$ NP). We know how to prove that SAT lies in the class NP. But how do we do this second step? How do we prove that if there is a polynomial time algorithm for SAT, then there is a polynomial time algorithm for every problem in NP? So what we're going to do now is formalize this notion of a reduction.

We want to show a reduction from problems such as MST and colorings to the SAT problem. What exactly do we mean by reducing this colorings problem to the SAT problem?

***

(Slide 22) Reductions



Reductions

Problems A & B ..(example, A=Colorings B=SAT)

$A \to B$ or $A \leq B$

means: reducing A to B

if we can solve problem B in poly-time
then we can use that algorithm
to solve A in poly-time

Now, let's consider problems A and B, and I want to show a reduction from A to B. And I want us to see what exactly that means.

Now, as an example, we can think of A, problem A as the colorings problem, and problem B as the satisfiability problem. Now, A -> B is a notation that we're going to use for a reduction. This is the same notation that's used in the Dasgupta textbook. Now, an alternative notation for reduction is that A ≤ B. What does this notation mean? It means that when we show a reduction from A to B, we're showing that B is at least as hard computationally as A, because if we can solve B then we can solve A.

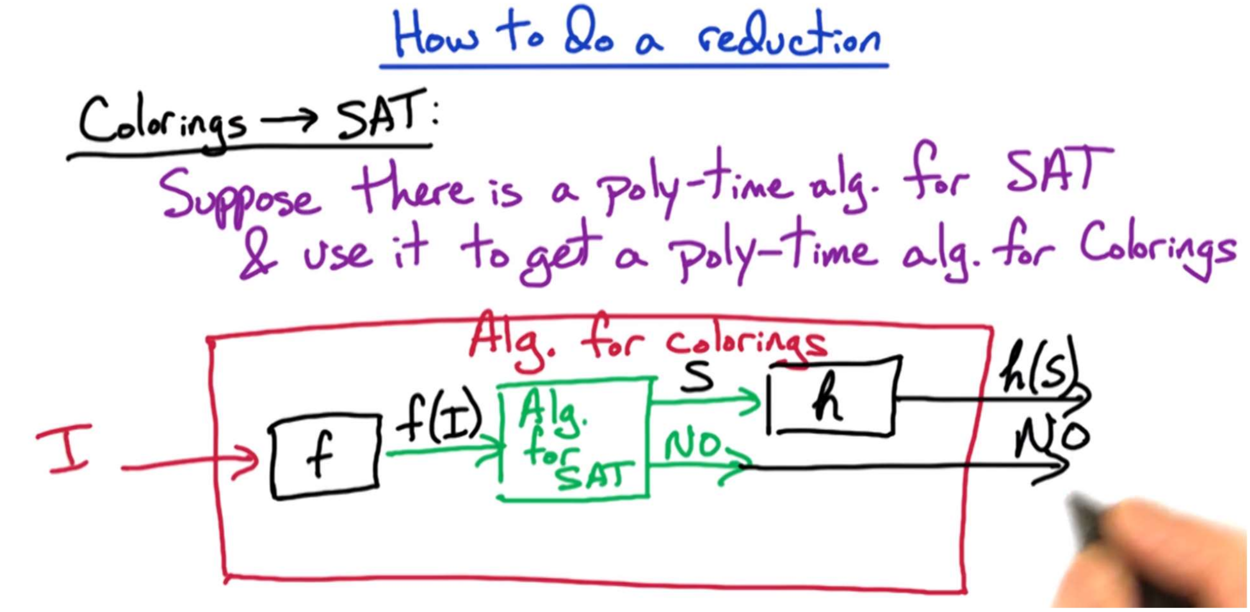
Now, if we're trying to devise an algorithm, it'll be easier to devise an algorithm for A. Why? Because if we devise an algorithm for B, then not only do we solve B, but we also solve A. Whereas if we solve A, we may just solve A - we may not necessarily solve B at the same time.

Now, let's formalize what exactly we mean by a reduction. So, a reduction from A to B means the following. So, verbally, this notation (A -> B) means reducing A to B. Formerly, this means that if we have an algorithm which solves problem B in polynomial time - so in this case, the SAT problem - then we can use that algorithm for problem B. So, in this case the SAT algorithm, we can use it to solve A in polynomial time. So, a reduction from the colorings problem to the SAT problem (Colorings -> SAT) means that if we have an algorithm to solve satisfiability in polynomial time, then we can use that algorithm to solve the colorings problem in polynomial time.

Okay, let's dive into this and see how we actually show a reduction. How do we show a reduction from colorings to satisfiability?

***

(Slide 23) How to do a Reduction



How to Do a reduction

Colorings → SAT:

Suppose there is a poly-time alg. for SAT & use it to get a poly-time alg. for Colorings

Now, let's suppose I want to show a reduction from Colorings to SAT. How do I do it?

Well if we reduce Colorings to SAT, what we're showing is that if there is a polynomial time algorithm for SAT, then we can use it to get a polynomial time algorithm for Colorings.

So what do we do? We suppose that there is a polynomial time algorithm for SAT. And we're going to treat this algorithm like a blackbox. What does it mean to treat this algorithm as a blackbox?

> [**blackbox algorithm**] Well, we don't know how the algorithm works, but we can use this as a basic subroutine.

We can put in some input for SAT and we can get out some output and it takes polynomial time for that algorithm to run. And we're going to use this blackbox algorithm as a subroutine to get a polynomial time algorithm for Colorings problem.

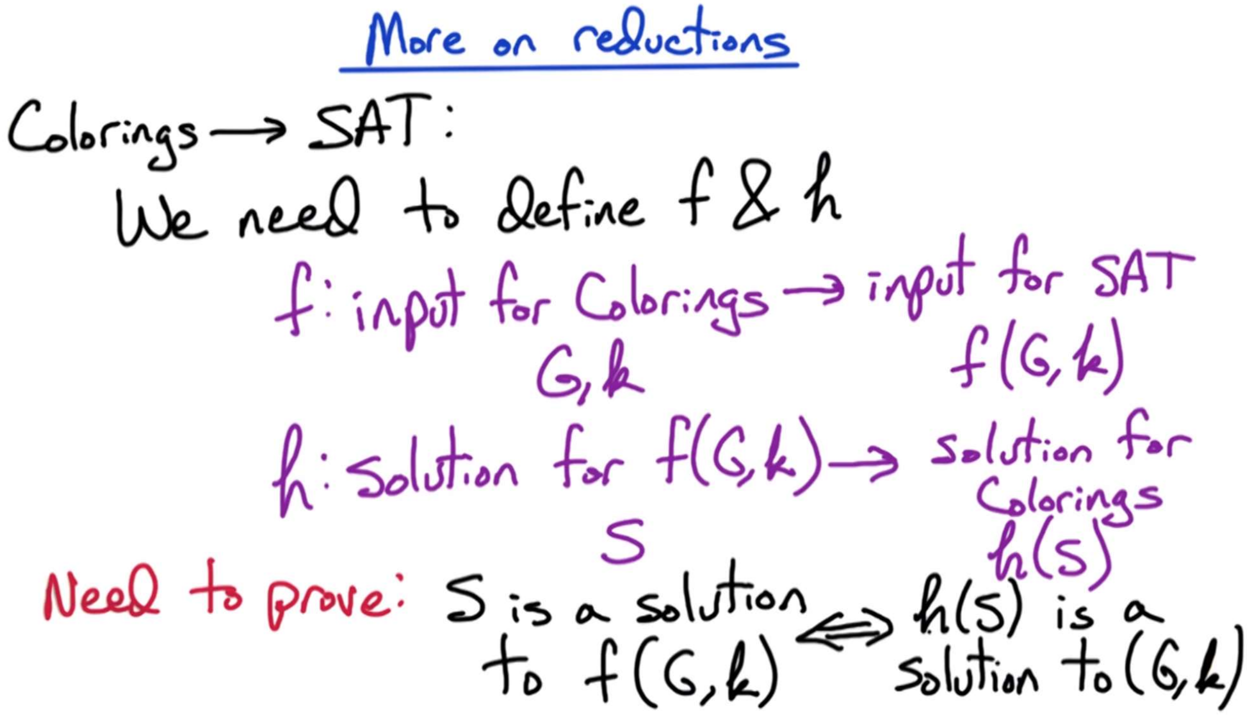
So let's look at it pictorially. So here's our blackbox algorithm for SAT. We can give it some input and it produces some output. Either it gives us a solution, or it outputs NO if there's no solution for that particular input.

Now, we want to use this black box algorithm or subroutine for satisfiability problem and we want to use it to construct an algorithm for the Colorings problem. So, this red box denotes our subroutine for the Colorings problem. So, the input to our colorings problem is a particular I (I for Instance), which in this case is a graph and a number of colors K. Now, what we need to do is to define a transformation f. We have to show how to transform the input to the Colorings problem and produce an input to the satisfiability problem. So we have to change this graph and this number of colors into a boolean formula. Then we plug that boolean formula into our SAT solver and our SAT solver either gives us a solution (in this case it's an assignment) which satisfies this formula or it outputs NO, there's no solution for this input.

Now, we have to transform this solution into a solution to the original Colorings input. So we have defined another transformation which we denote as h. This h takes as input a solution to our SAT input and it transforms it to get a solution to our original Colorings instance. So if S is a solution to f(I), then h(S) is the solution to I. And if our SAT input had no solution, then we'll say that our Colorings input had no solution. So if we get output NO, we output NO.

***

(Slide 24) More on Reductions



More on reductions

Colorings → SAT:

We need to define f & h

f: input for Colorings → input for SAT

G,k $\qquad$ f(G,k)

h: solution for f(G,k) → solution for Colorings

S $\qquad$ h(S)

Need to prove: S is a solution to f(G,k) ⟺ h(S) is a solution to (G,k)

To reduce Colorings to SAT, what do we need to show? We need to find these two functions f and h:

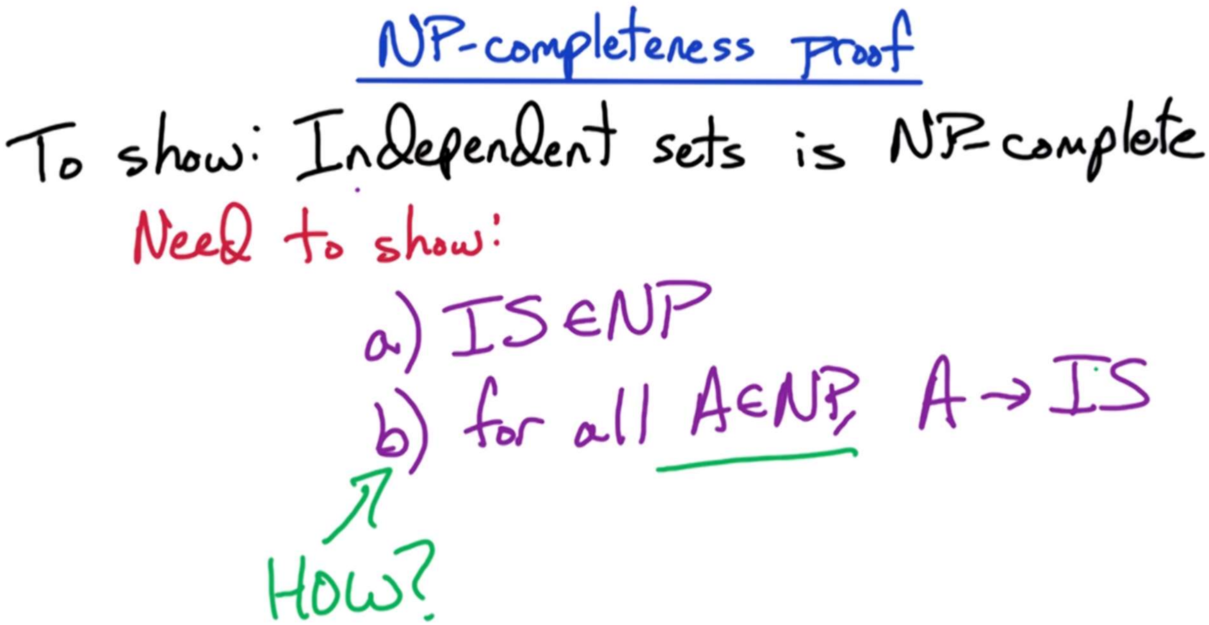
- f takes an input for the Colorings problem and produces an input for the SAT problem. If you think of the input for the Colorings problem as a graph and a number of colors, so it's a G and a k, then the input for the SAT problem is f(G,k).
- Now, h takes a solution to this SAT problem – this f(G,k) is a Boolean formula – So, it's going to take an assignment, a satisfying assignment for this boolean formula, and it's going to produce a solution to the original Colorings problem on (G, k). So. it's going to take an assignment S which satisfies this boolean formula and it's going to produce h(S) which is a coloring for this graph using k colors.

We need to define these two functions f (which transforms inputs to Colorings to inputs to SAT) and h (which transform solutions to SAT into solutions for Colorings).

Now, what do we need to prove in order to prove that this reduction is valid? Well, we need to prove that, if S was a solution to f, then h(S) is a solution to the original G. But we also need that if there was no solution to f, then there is no solution to the Colorings problem. So we need to show that if S was a solution to this SAT input, then h(S) is a solution to this Colorings input and vice-versa. We need to show this equivalence, this if and only if statement because we need to know that the solutions here map to solutions here and no instances here are mapped to no instance here. We need both directions. We need solutions here (i.e., a solution to f(G,k)) if and only if solutions here (i.e., h(S) is a solution to (G,k)).

***

(Slide 25) NP-Completeness Proof



NP-completeness proof

To show: Independent sets is NP complete

Need to show:

a) $IS \in NP$

b) for all $A \in NP$, $A \rightarrow IS$

How?

Now let's go back and see how we show that a particular problem is NP-complete. Later we're going to consider this problem, Independent Sets problem, and we're going to prove that the Independent Sets problem is NP-complete.

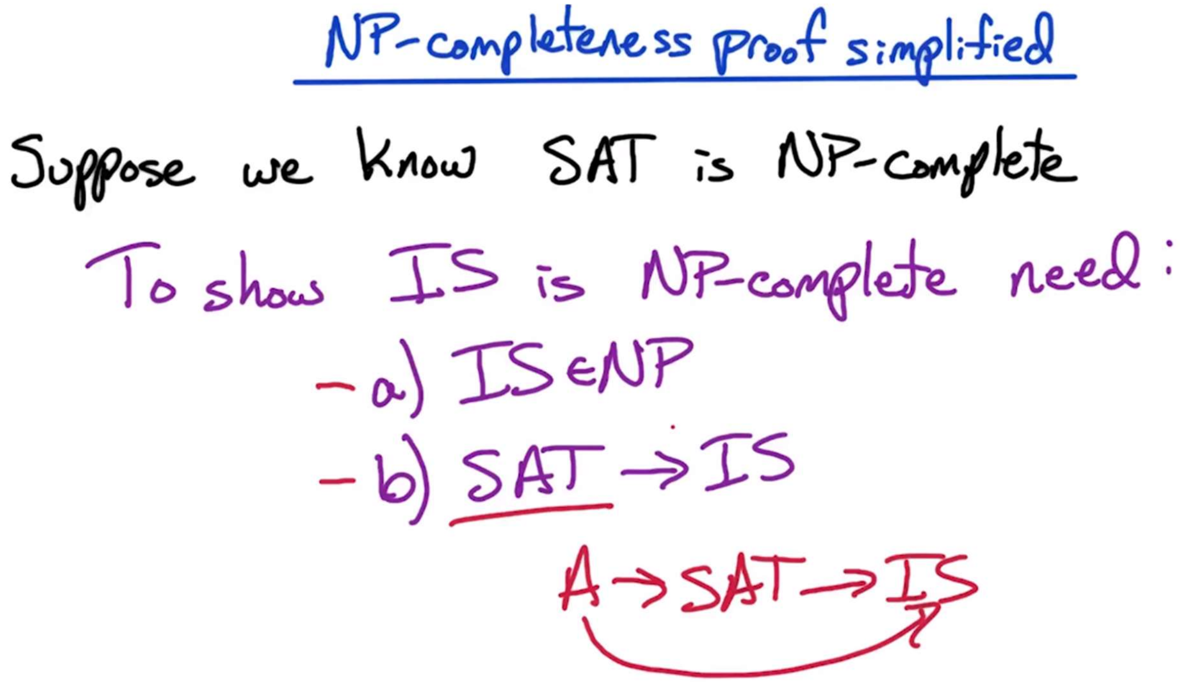
Now the details of the Independent Sets problem is not important at this point. But, what exactly do we need to show in order to show that Independent Sets problem is NP-complete? We need to show the following:

- (I'm a bit lazy, so let me denote Independent Sets problem by IS). First, we need to show that the Independent Sets problem is in the class NP - that we know how to do - We need to show that we can verify solutions in polynomial time.
- The second thing we need to show is that if we can solve Independent Sets problem in polynomial time, then we can solve every problem in NP in polynomial time. Now the second thing we need to show is that for every problem A in the class NP, there is a reduction from A to the problem Independent Sets. That means that if we have a polynomial time algorithm for the Independent Sets problem, then we can use that as a subroutine to solve the problem A in polynomial time. And since we can do this for every problem A in the class NP, then, if there is a polynomial time algorithm for the Independent Sets problem, then there is a polynomial time algorithm for every problem in NP.

  But how can we do this second step? We need to take every problem in NP, and show reduction from every problem in NP to the Independent Sets problem.

***

(Slide 26) Simpler Proof Approach



NP-completeness proof simplified

Suppose we know SAT is NP-complete

To show IS is NP-complete need:

- a) $IS \in NP$
- b) $\underline{SAT} \to IS$

$$\begin{tikzcd} A \arrow[r] \arrow[rr, bend right] & SAT \arrow[r] & IS \end{tikzcd}$$

Now, suppose that we know that the SAT problem is NP-complete. Now, in fact, we do know this. What exactly does this mean? This means that SAT lies in the class NP. And, in addition, it means that, for all problems A in the class NP, we know reduction from A to the SAT problem. Now, suppose we show that the SAT problem reduces to the Independent Sets problem (A -> IS). Then, what do we know? We know that every problem A ∈ NP reduces to SAT. So A reduces to SAT (A -> SAT), and SAT reduces to the Independent Set problem (SAT -> IS). So, by just composing these reductions or composing these functions, we have a reduction from A to Independent Sets problem (A -> SAT -> IS). And this holds for every A and NP.

Therefore, for every A ∈ NP, we have a reduction from A to the Independent Sets problem. If we have a known NP-complete problem such as SAT, then to show that the Independent Sets problem is NP-complete, we just have to show that the Independent Sets problem lies in the class NP, and that there is a reduction from SAT to Independent Sets.

To recap, to show that the Independent Sets problem is NP complete, we need to show the following.

- The Independent Sets problem lies in the class NP, that's easy to do,

- and we have to take a known NP-complete problem such as SAT (satisfiability) and we have to show how to reduce satisfiability to the Independent Sets problem. We have to take the **known** NP-complete problem and reduce it to our **new** NP-complete problem (known problem -> new problem). Since every problem A $\in$ NP reduces to SAT because SAT is NP-complete, and we just showed this reduction from SAT to Independent Sets problem. therefore, we know how to take every problem A and NP and reduce it to the Independent Sets problem and we can start from any known NP-complete problem.

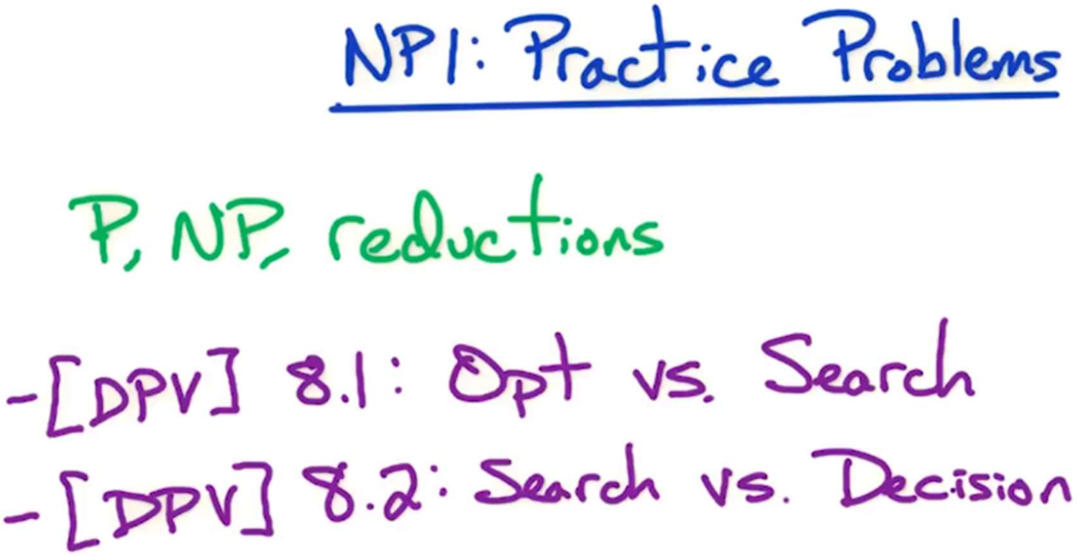
At the beginning, we'll start with SAT and we'll build up our repertoire of NP-complete problems and then we can take any of these known NP-complete problems and reduce it to our new problem, which we're trying to show as NP-complete. Now, it sounds much more tangible.

[Recap] To prove that a problem such as Independent Set problem is NP-complete, we have to just show that we can verify solutions in polynomial time. That's the first step and then we have to take a known and complete problem and reduce it to this new problem. Now, the second step is what confuses students many times. Often, they'll show the reduction in the other direction. That's not really proving anything. We already know that every problem in NP can be reduced to this known NP-complete problem. And even if they marked it this way, that they're trying to do a reduction from set Independent Sets, actually, in the proof, they actually do the reduction the other way. It's meaningless if you do the reduction in the other way. So this is the difficult part for students sometimes. You have to make sure that you're doing a reduction **from the known** NP-complete problem **to this new problem** that you're trying to show as NP-complete because you want to show that you can reduce every problem and NP to this new problem in order to prove that this new problem is computationally difficult. It's NP-complete.

So next, we're going to dive in and we're going to prove some problems such as Independent Set problem are, in fact, NP-complete.

***

(Slide 27) Practice Problems



NP1: Practice Problems

P, NP, reductions

-[DPV] 8.1: Opt vs. Search

-[DPV] 8.2: Search vs. Decision

Now, at this point, you should understand the notions of the class P and the class NP and the notion of reductions. Now, we're going to see a lot of examples of proofs that a problem is NP complete.

So I don't expect you to be comfortable with that at this point. But just based on these definitions, there are a few practice problems you can try from the textbook. The first two problems in Chapter 8 of the textbook, Dasgupta, Papadimitriou, and Vazirani, are relevant at this point:

- The first problem looks at reductions between optimization problems and search problems,
- and the second problem looks at reduction between search problems and decision problems.

Now, in the next lecture, we're going to do our first NP-completeness proof. We're going to prove that 3SAT is NP-complete using the fact that SAT is NP.