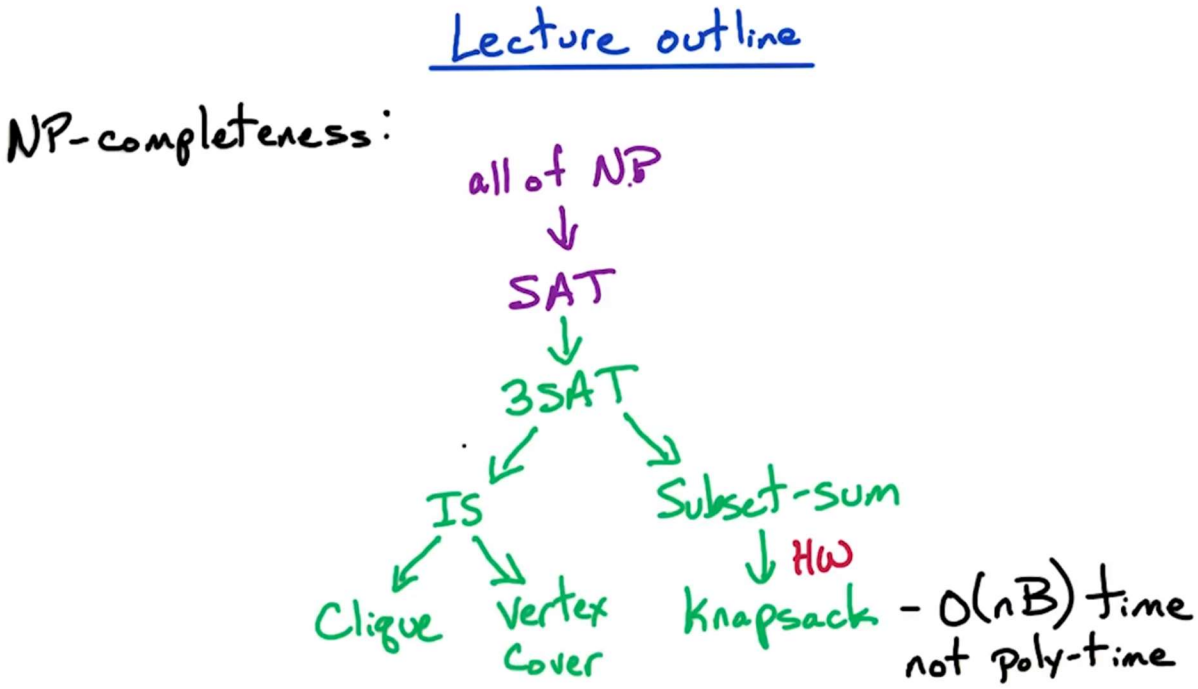
LESSON: NP4: Knapsack

***

(Slide 1) Knapsack Lecture Outline



Let's give a quick recap of what we've seen so far.

- We took for granted that SAT is NP-complete. This means that all the problems in the class NP can be reduced to SAT. So, if we can solve SAT in polynomial time, then we can solve any problem in NP in polynomial time.
- And we saw that 3SAT is NP-complete. This involved a reduction from SAT to 3SAT.
- Then we proved that the Independent Set problem is NP-complete. This involved a reduction from 3SAT to the Independent Set problem. This was a quite interesting reduction because it involved a transformation from a logic problem,
- Then we proved that the Clique problem and Vertex cover problems were NP-complete. These involve more trivial reductions from the Independent Set problem.

What we are going to do now is we're going to prove that the Knapsack problem is NP-complete. As an intermediate step, we're going to look at the Subset-sum problem. We're going to prove that a Subset-sum problem is NP-complete. This is going to involve a reduction from the 3SAT problem to Subset-sum problem. This, again, is a quite non-trivial reduction because it involves transforming a logic problem 3SAT into this optimization problem Subset-sum.

Once we prove that Subset-sum is NP-complete, it will be quite easy to prove that the Knapsack problem is NP-complete. We'll leave this last step as a homework problem - proving that Knapsack is NP-complete once we know that Subset-sum is NP-complete.

If you recall from the beginning of the course, we studied the Dynamic Programming algorithm for the Knapsack problem. And, there, we saw an O(nB) time algorithm for the Knapsack problem, and we pointed out why this is not polynomial time.

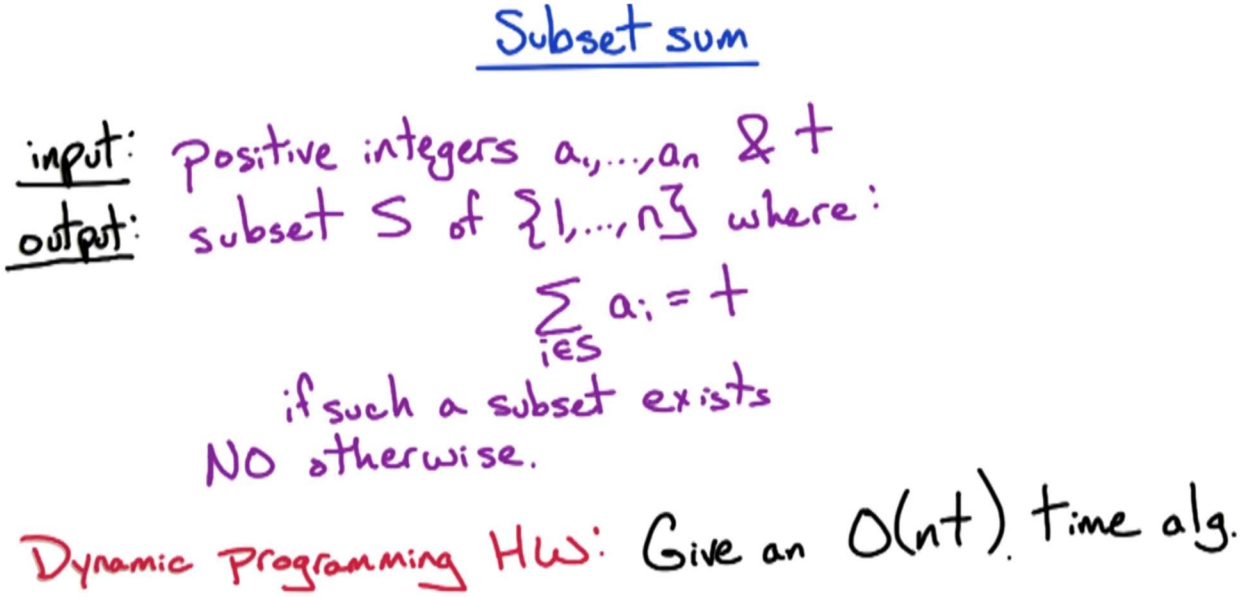
Why is this running time not polynomial in the input size? In order to be polynomial in the input size, it has to be polynomial in n and log(B). Why this is not polynomial is somewhat of a subtle point. I think it becomes much clearer when you see the reduction from 3SAT to Subset-sum, or 3SAT to Knapsack. What you see in this reduction is that capital B will be exponential in the size of the input formula for the 3SAT problem. So, if we use this Algorithm to solve 3SAT, this would give us an exponential time algorithm of 3SAT. It's a quite clever reduction so let's dive into it. Let's prove that the Subset-sum problem is NP-complete.

Let's start with the definition of a Subset-sum.

***

LESSON: NP4: Knapsack

(Slide 2) Subset-Sum



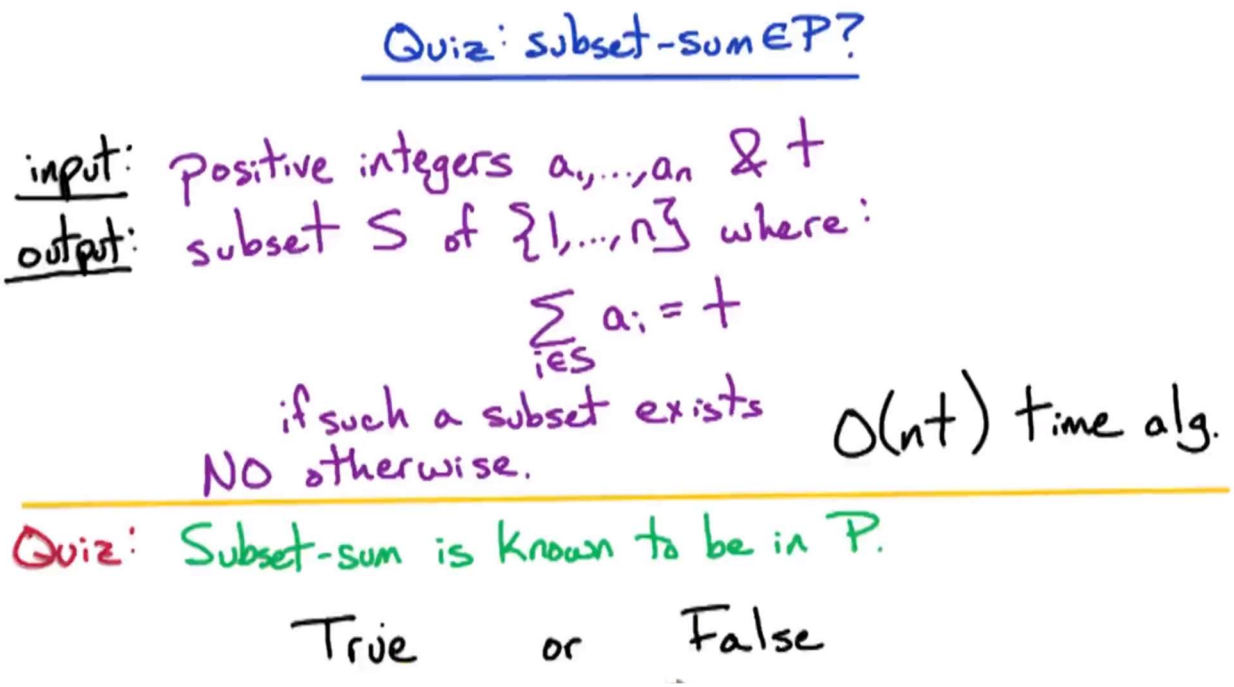
The input to the Subset-sum problem are n positive integers which we'll denote as a1 thru an, and one additional integer, t. Our goal is to find a subset of numbers which sum up to exactly t. So we want the sum over the indices in this subset of the corresponding numbers to be equal to exactly t. We output such a subset if it exists and otherwise, we output NO.

This is a great practice problem for dynamic programming. Using dynamic programming, we can solve this problem in O(nt) time. If you want practice with dynamic programming, I suggest you try that.

***

LESSON: NP4: Knapsack

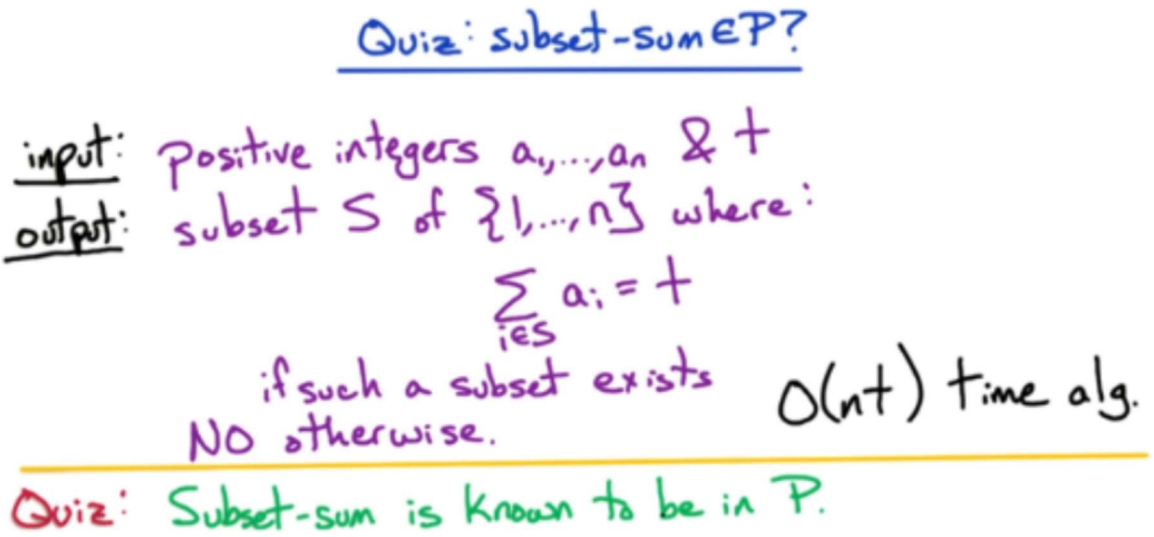
(1st Slide 3) Subset-Sum in P?



Let's take for granted that there's an O(nt) algorithm for the Subset-sum problem. Now, given that fact, "Subset-sum is known to be in P" – is that True or False?

***

NP4: Knapsack: Subset-Sum in P?

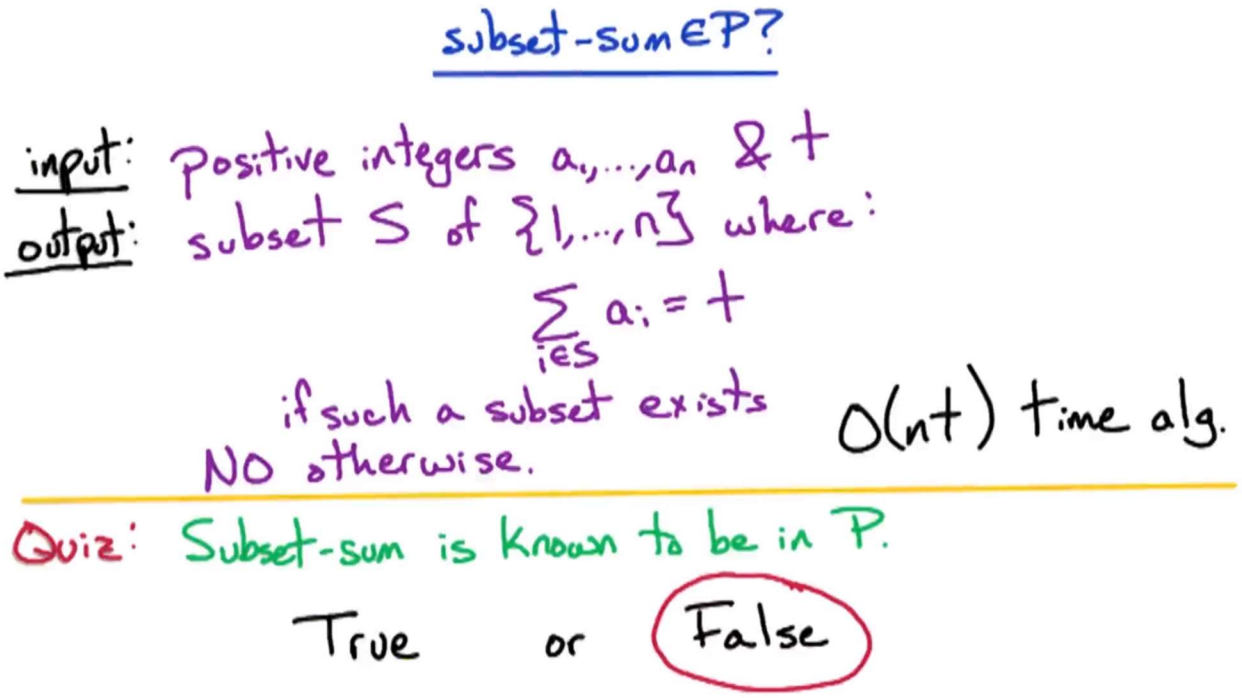


Question: Is the claim that Subset-Sum is in P True or False?

***

LESSON: NP4: Knapsack

(2nd Slide 3) Subset-Sum in P? (Answer)

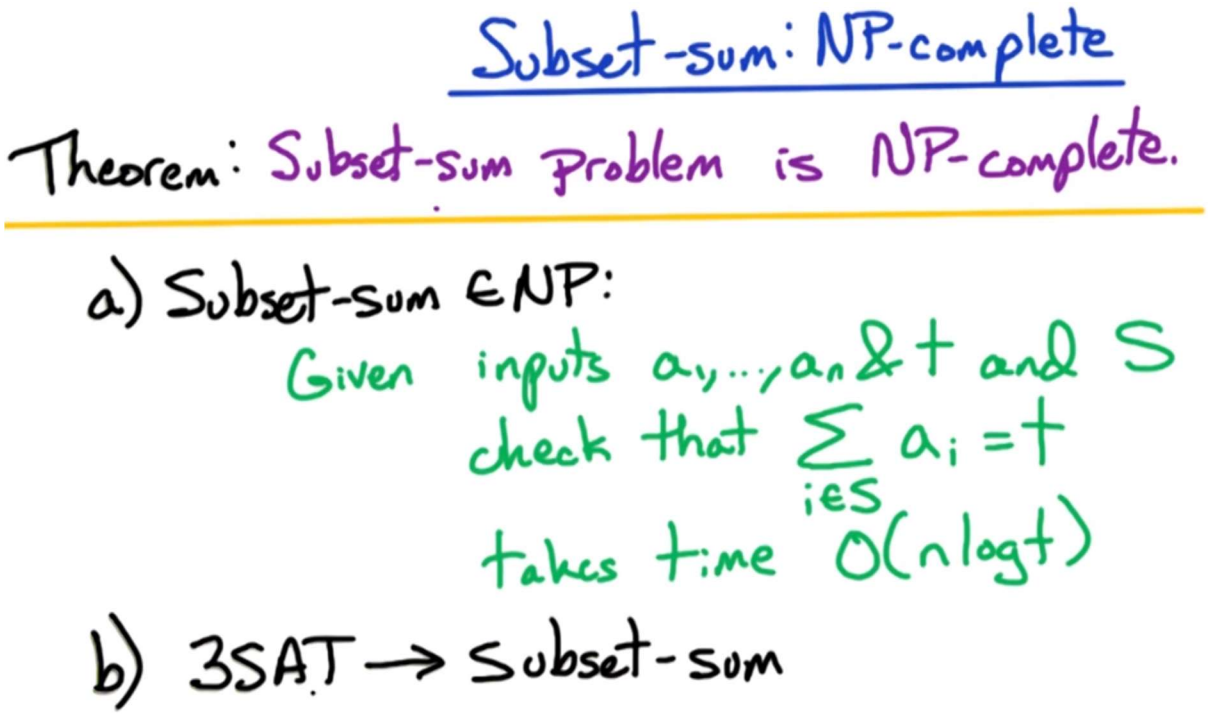


This is False. Just like with Knapsack problem, this is not polynomial in the input size. To be polynomial in the input size, the running time would have to be n times polynomial in log t.

***

LESSON: NP4: Knapsack

(Slide 4) Subset-Sum NP-Complete



We're going to prove now that's the Subset-sum problem is NP-complete. Again, there are two parts of the proof.

Subset-sum ∈ NP:

First, we have to prove that the subset-sum problem is in the class NP. So we take an input instance to the Subset-sum problem. This is specified by n numbers a1 thru an, and t, and we take a proposed solution S. Then we check that the sum over the integers in that subset equals exactly t.

How long does it take to compute the sum? Well, there is at most n numbers there, so that's O(n) time.

And, then how long is each number? Well, each number at most log t bits. So to sum these up takes at most O(n log t). Notice this as polynomial in the input size, so we can verify solutions in polynomial time.

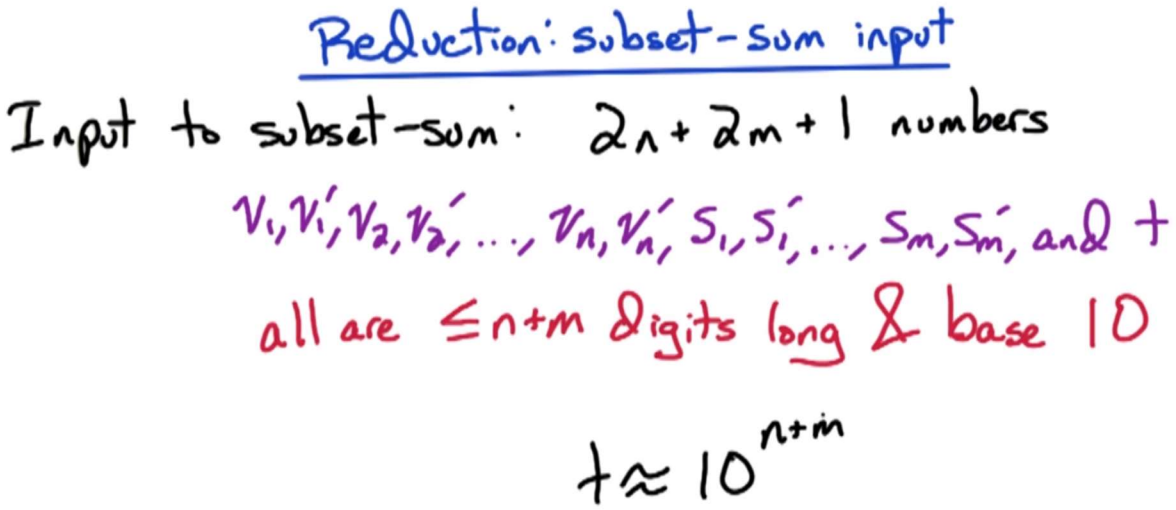
Next we have to show that the Subset-sum problem is at least as hard as every problem in the class NP. So we have to take a known NP-complete problem and reduce it to the Subset-sum problem.

Which problem do we choose? Well, we're going to choose the 3SAT problem, and we're going to reduce it to the Subset-sum problem. The reduction is very clever, and somewhat involved; but, it doesn't involve any heavy math or anything like that. And, it really illustrates why the O(nt) time algorithm is not efficient for this subset-sum problem.

***

LESSON: NP4: Knapsack

(Slide 5) 3SAT Subset-Sum: Input



The input to the Subset-sum problem will consist of 2n + 2m + 1 numbers. There are n variables and 2n literals, so we'll have a number for each literal. There are m clauses, so we'll have two numbers for each clause. And then we'll have one additional number which is our desired sum.

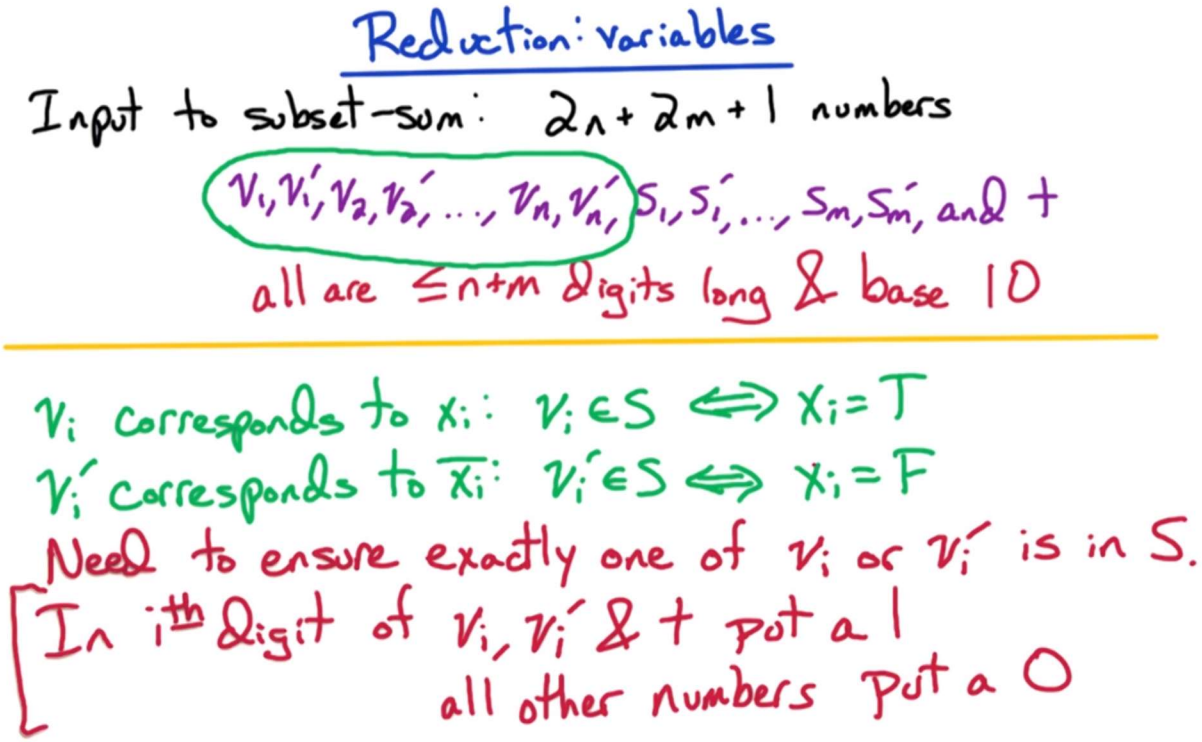
We'll have v1 and v1' corresponding to x1 and $\overline{x1}$, v2 and v2' for x2 and $\overline{x2}$. And vn and vn' for xn and $\overline{xn}$. We'll have these two additional numbers, s1 and s1', for the first clause, and so on for the m clauses. And finally, we'll have the desired sum.

Now these numbers are all huge. They're going to be at most n+m digits long. And, we're going to work in base 10. This is so that if we add up any subset of numbers, there'll be no carries between the digits. Note these numbers are huge, for instance, t is on the order of 10^{n+m}. This illustrates why our O(nt) algorithm is a somewhat terrible algorithm.

***

LESSON: NP4: Knapsack

(Slide 6) 3SAT Subset-Sum: Variables



Now, let's take a look at the actual specification of the input to the subset sum problem. Let's start with these numbers corresponding to the 2n literals.

In particular vi corresponds to xi. So we're going to include this number vi in the subset S if and only if the literal xi is set to true. Similarly vi' corresponds to xi_bar and we will include vi' in this subset S if and only if the variable xi is set to false. And the assignment for the 3SAT formula either sets xi to true or xi to false.

Therefore, we need for the Subset-sum problem that either we include vi in S or vi' in S. Well, we can't include both and we can include neither, because then we don't know how to set xi. So we need to ensure that exactly one of vi or vi' is in S. How do we achieve this? Well, in the ith digit of these three numbers vi, vi' and t, we put a 1. In all the other numbers, we put a 0 in the ith digit.

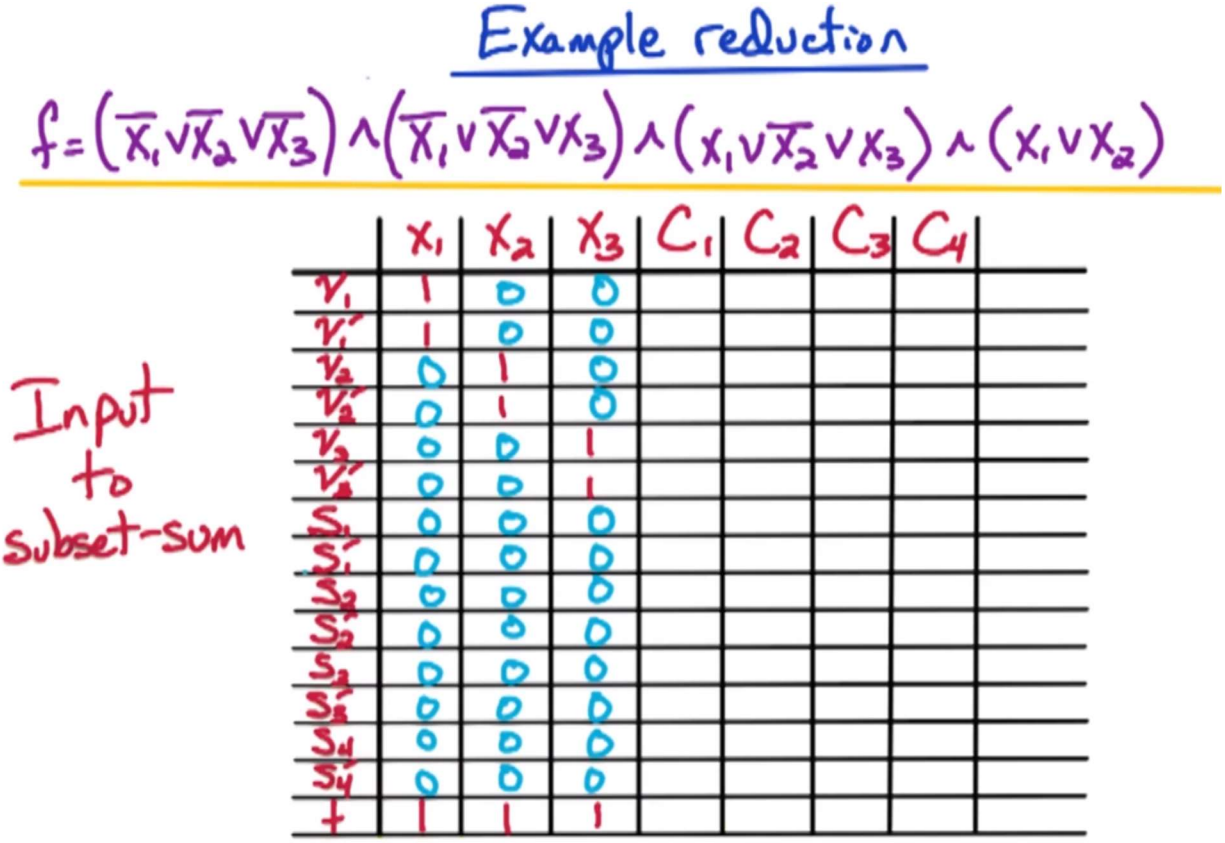
Now by using base 10, we will ensure that there is no carries between the digits. So each digit is going to behave independently of each other. Recall that t is our desired sum. So, the only way to achieve a desired sum that has a 1 in the ith digit is to include either vi or vi' in S – but, not both and not neither.

So, this specification ensures that our solution to the Subset-sum problem is going to correspond to an assignment. Whether it's satisfying or not is another issue; but, for now, at least we know that we have an assignment. So, either we're going to correspond to xi being set to true or xi being set to False.

***

LESSON: NP4: Knapsack

(Slide 7) 3SAT: Subset-Sum: Example



To illustrate the reduction, it'll be useful to have a running example and we can fill in the numbers as we go along.

So let's consider this input formula to the 3SAT problem:

f = (x1 ∨ x2 ∨ x3) ∧ (x1 ∨ x2 ∨ x3) ∧ (x1 ∨ x2 ∨ x3) ∧ (x1 ∨ x2)

It consists of three variables and four clauses.

- For variable x1, there'll be two numbers, the v1 and v1′,
- Similarly for x2, there'll be v2 and v2′
- and for x3, there'll be v3 and v3′

In addition there'll be two numbers for each clause, so we have

- s1 and s1′ for clause one (C1),
- s2 and s2′ for clause two (C2),
- s3, s3′, s4, s4′ for the last two clauses (C3 and C4)

- and finally we have t which is our desired sum.

These 15 numbers specify the input to the subset-sum problem. These numbers are going to be seven digits long. The digits correspond to the variables - three variables - and the clauses - four clauses.

The first three digits correspond to the three variables, x1, x2, x3. The next four digits correspond to the four clauses.

Now an assignment to f either sets x1 to True or False. So, we wanted that a subset-sum solution either includes v1 or v1′ in the subset. How do we ensure that? We put a 1 in the first digit for v1, v1′ and t. For all other numbers, we put a 0 in the first digit.

Now assuming there's no carry from the other digits, how can our subset of numbers achieve a sum of 1 in the first digit? Well, the only way is to include either v1 or v1′, but not both and not neither. So, we have to include exactly one of these two in order to achieve a sum of 1 in the first digit.

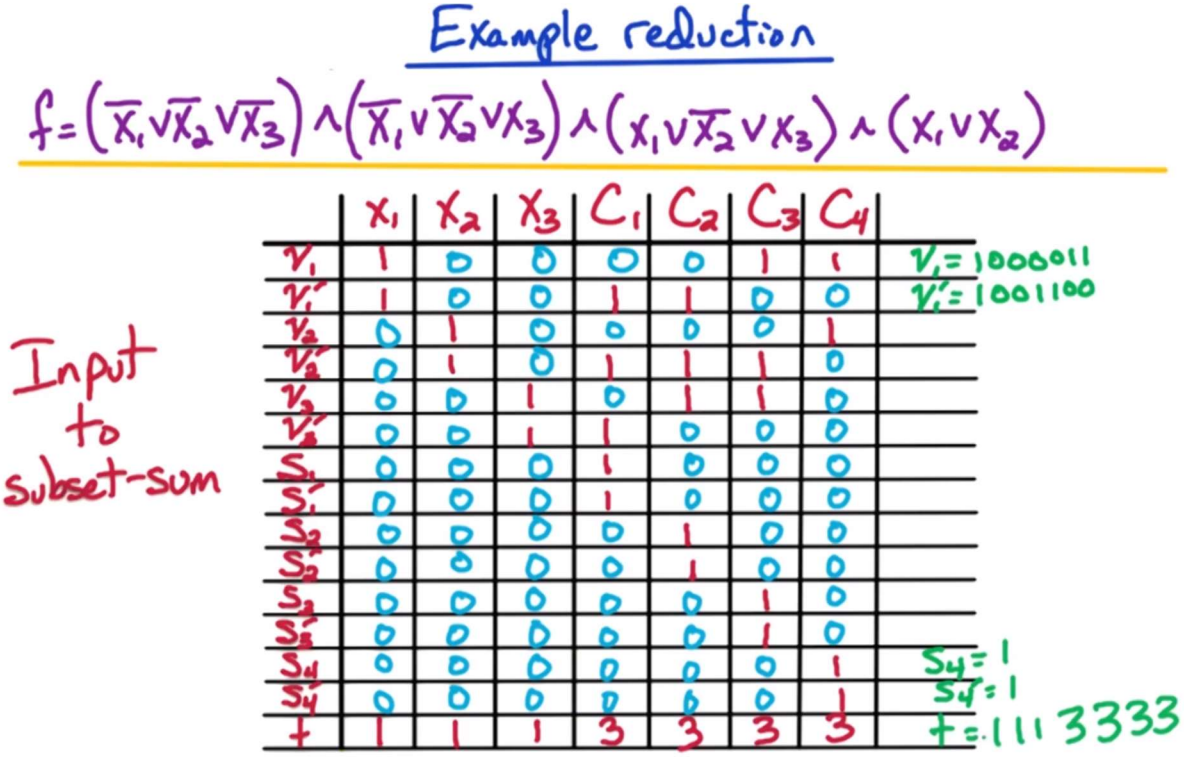
Similarly we put a 1 in the second digit for v2 and v2′ and for t, and in the third digit we put a 1 for v3, v3′ and t. Then we put a 0 for all other numbers in the second and third digits. This ensures that exactly one of v2 and v2′ is in the subset and exactly one of v3 and v3′ is in the solution to the Subset-sum problem.

So now we know that any solution to the Subset-sum problem corresponds to an assignment. Now, we have to ensure that it corresponds to a satisfying assignment. So, for that we use these remaining four digits.

***

LESSON: NP4: Knapsack

(Slide 8) 3SAT Subset-Sum: Clauses



The first n digits correspond to the variables. The next m digits correspond to the clauses. So, digit n plus j is going to correspond to clause Cj.

If the literal xi appears in the clause Cj, well, xi, this literal, corresponds to this number of vi. So. we're going to put a 1 in digit n+j for this number vi. Similarly, if the literal xi_bar appears in this clause, well, in this case, this literal xi_bar corresponds to this number vi' prime. So, we put a 1 in digit n+j for this number, vi'. What this does is, it encodes the clauses in these numbers, vi and vi'.

Let's go back to our running example.

- Our first clause contains x1_bar, x2_bar, and x3_bar. This clause corresponds to the fourth digit so we're going to put a 1 in v1' and put a 1 in v2', and a 1 in v3'.
- Similarly, for the second clause, we put a 1 in v1' for x1_bar, v2' and v3.
- For clause three, we put a 1 in v1, v2', and v3.
- Clause four, v1 and v2.

Now for the first clause, we want to ensure that either v1', v2', or v3' are included in the subset in the solution to the Subset-sum problem. We just need that at least one of these is included.

We don't need that exactly one so we can't just put a 1 here in t. We want that either 1, 2, or 3 of these numbers are included in the solution so we put a 3 in the fourth digit of t, fifth digit, sixth digit, and seventh digit.

Now, if we include all three of these numbers in our solution, then we're okay, but what if we only include one of these digits? What if only one of these literals is satisfied?

Well, that's where we use these buffer numbers, s1 and s1'. We're going to put a 1 in the fourth digit for these two numbers. The point is that if only one of these three literals is satisfied, then we get up the sum up to 3 by including both of these buffer numbers. So. we get a 1 from one of these three plus 1 plus 1 - and that gives us 3. What if all three of these literals are satisfied? Then, we don't have to use either of these buffer numbers.

What if exactly two of these literals are satisfied? Then we use one or the other of these buffer numbers. But, if none of these literals are satisfied, then using both of the buffer numbers only gets us up to a sum of 2 so we can't satisfy t. We can't get a solution to the Subset-sum problem. The only way to get a solution to the Subset-sum problem is to have at least one of these literals satisfied and then we can use the buffer numbers to get the desired sum in this digit.

So, similarly, for clause two, we put a one in this digit for s2 and s2', s3 and s3', s4 and s4'. All the other numbers have zeros in these digits.

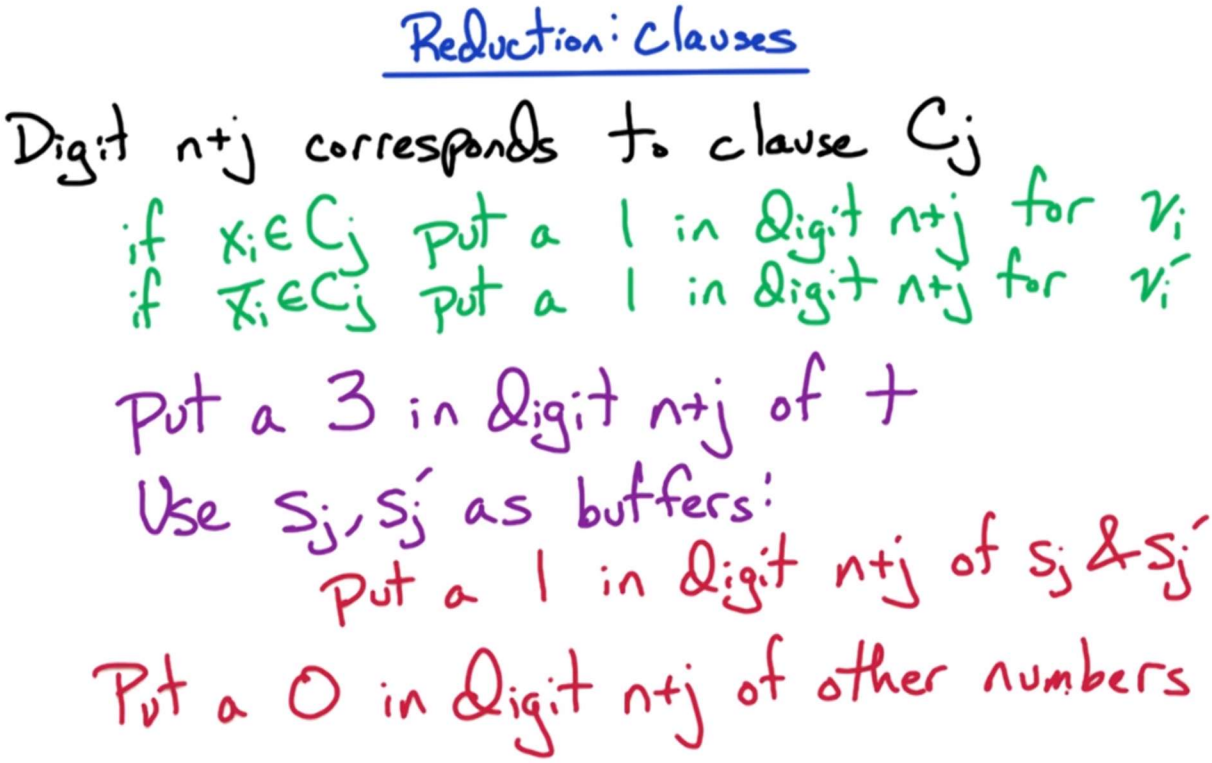
This specifies the reduction from the 3SAT formula to the Subset-sum problem. Note that

- v1 is the number 1000011
- v1' is the number 1001100
- s4 and s4' are 1
- t is the number 1113333.

***

LESSON: NP4: Knapsack

(Slide 9) 3SAT: Subset-Sum: Buffers



Let's go back and specify the remainder of the reduction in general.

For each clause, we encoded the literals which appear in that clause using vi and vi'. To ensure that at least one of these literals is satisfied, we put a 3 in digit n+j of t. Then, if all of the literals in this clause are satisfied, we get the desired sum in this digit.

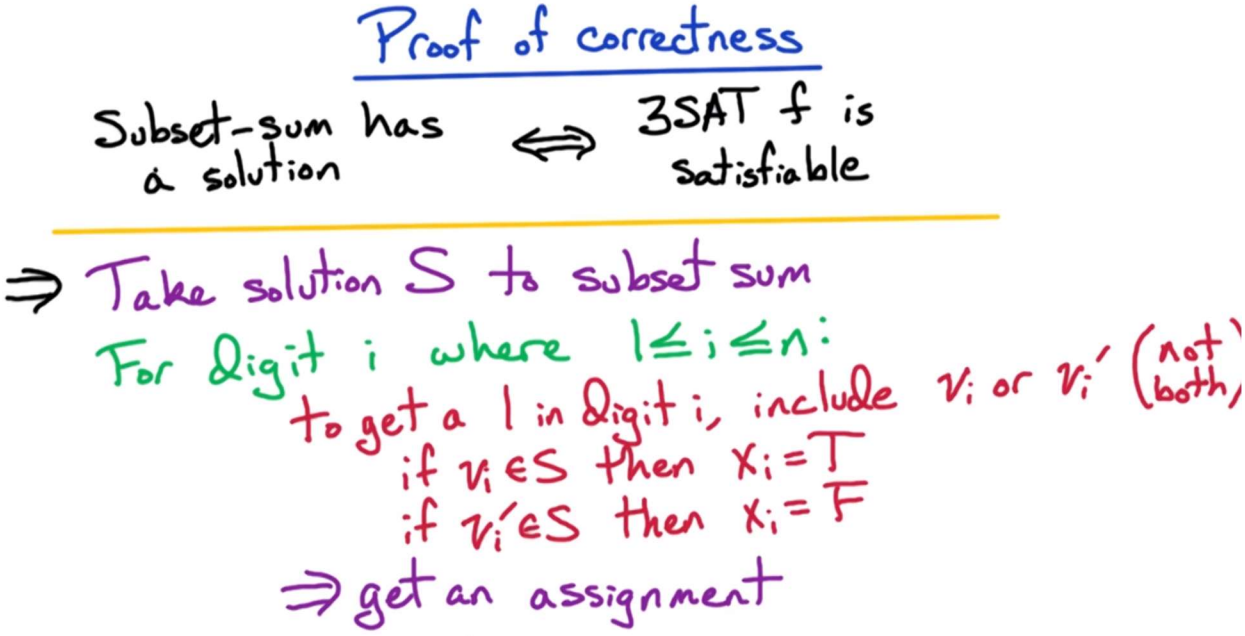
What if only one or two of the literals in this clause are satisfied? How do we get the desired sum? Well, that's where we use sj and sj' to act as buffers. We put a 1 in digit n+j of both of these numbers, and finally put a 0 in digit n+j of all the other numbers which we haven't specified so far.

The main point is that, if all three of the literals in this clause are satisfied, then we get the desired sum of 3 in this digit. If exactly one or two of these literals are satisfied, then using these buffer numbers, we get the desired sum of 3 in this digit. But, if none of the literals in this clause are satisfied, then there's no way to achieve a sum of 3 in this digit because the sum of these buffer numbers only adds up to 2.

***

LESSON: NP4: Knapsack

(Slide 10) 3SAT: Subset-Sum: Correctness



What we've sketched so far and what we'll formally prove now is that the subset sum instance that we constructed has a solution if and only if the original 3SAT input is satisfiable. It's going to prove that we did a value reduction. Let's start with the forward direction.

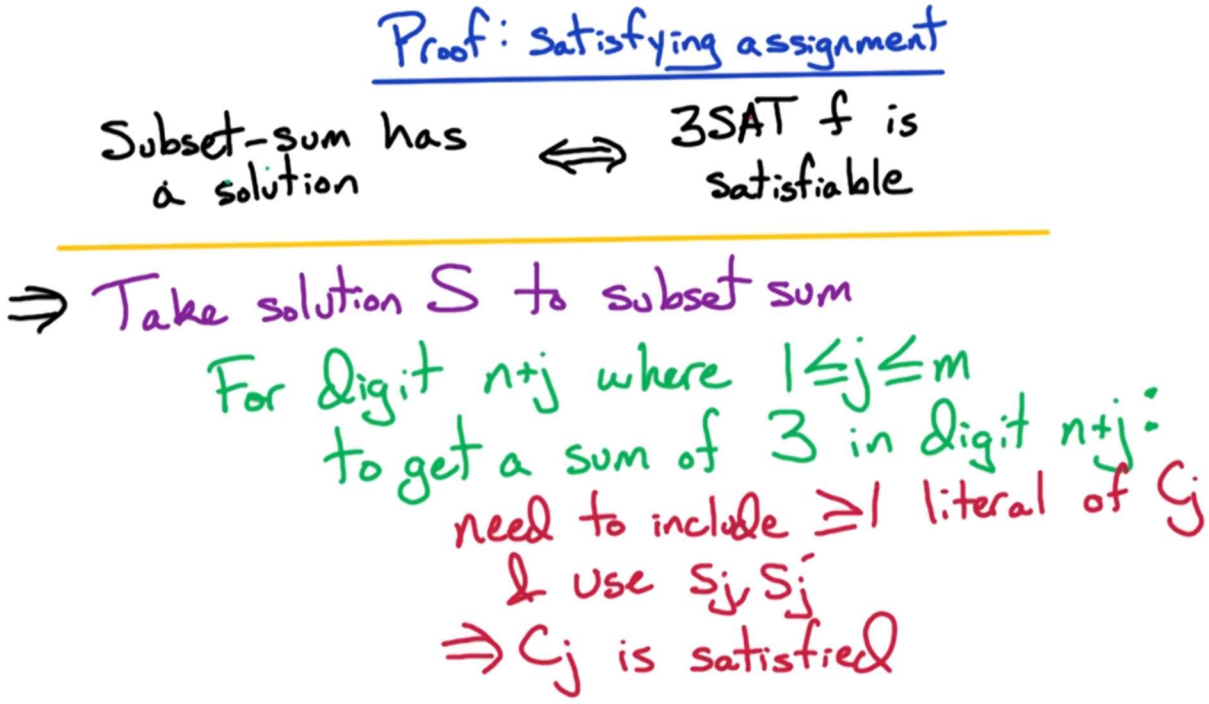
A solution for Subset-sum => A satisfying solution for 3SAT:

> Let's take a solution S to this subset sum instance. So we know that if we sum up the numbers in S, it sums up to exactly t.
>
> Let's first look at the first n digits. These correspond to the n variables x1 through xn. We know that t has a one in digit i. In order to get a 1 in digit i, we have to include vi or vi' in our subset S, but not both. If we include both we get a 2 in digit i. And, if we include neither, we get a sum of 0 in digit i. So the only way to get a sum of exactly 1 in digit i is to include exactly one of vi and vi'.
>
> If we include vi in our subset S, then we set the variable xi to True. If we include vi' in our subset S then we set the variable xi to False. So, from this solution to the subset sum problem, we get an assignment. We get a True/False assignment for the variables. Why is this a satisfying assignment? Well, to see that, we'll look at the remaining digits.

***

LESSON: NP4: Knapsack

(Slide 11) 3SAT: Subset-Sum: Correctness



We just saw, if we take a solution to the Subset-sum instances that we specified, then using the first n digits, that specifies a True/False assignment for the n variables x1 thru xn. Now, using the remaining n digits, we're going to show that that assignment that we just specified corresponds to a satisfying assignment. It satisfies the input formula f.

First n digits correspond to the variables. The next m digits correspond to clauses. So let's look at digit n+j, where j varies between one and n. So digit n+j corresponds to clause Cj. In our definition of the Subset-sum instance, we put a 3 in digit n+j of t. So, we need to achieve a sum of 3 in digit n+j. In order to get a sum of three in digit n+j, we have to include at least one of the numbers corresponding to the literal appearing in clause j.

If we satisfy exactly one literal in Cj, then we use both sj and sj' as buffer numbers and then we get a sum of 3. If we satisfy exactly two literals in this clause, then we either include sj or sj' in our subset S, and that gives us a sum of 3. If we satisfy exactly three literals in this clause, then we don't need to use either of these buffer numbers, sj or sj'. Just using these three satisfied literals gives us a sum of 3 in this digit n+j.

If we satisfy zero of the literals in this clause, then there is no way to achieve a sum of 3 in this

digit, and therefore, we don't have a solution to the subset sum problem. We're assuming we have a solution; therefore, we must have at least one satisfied literal in this clause. Thus, Cj is satisfied and thus, every clause is satisfied.

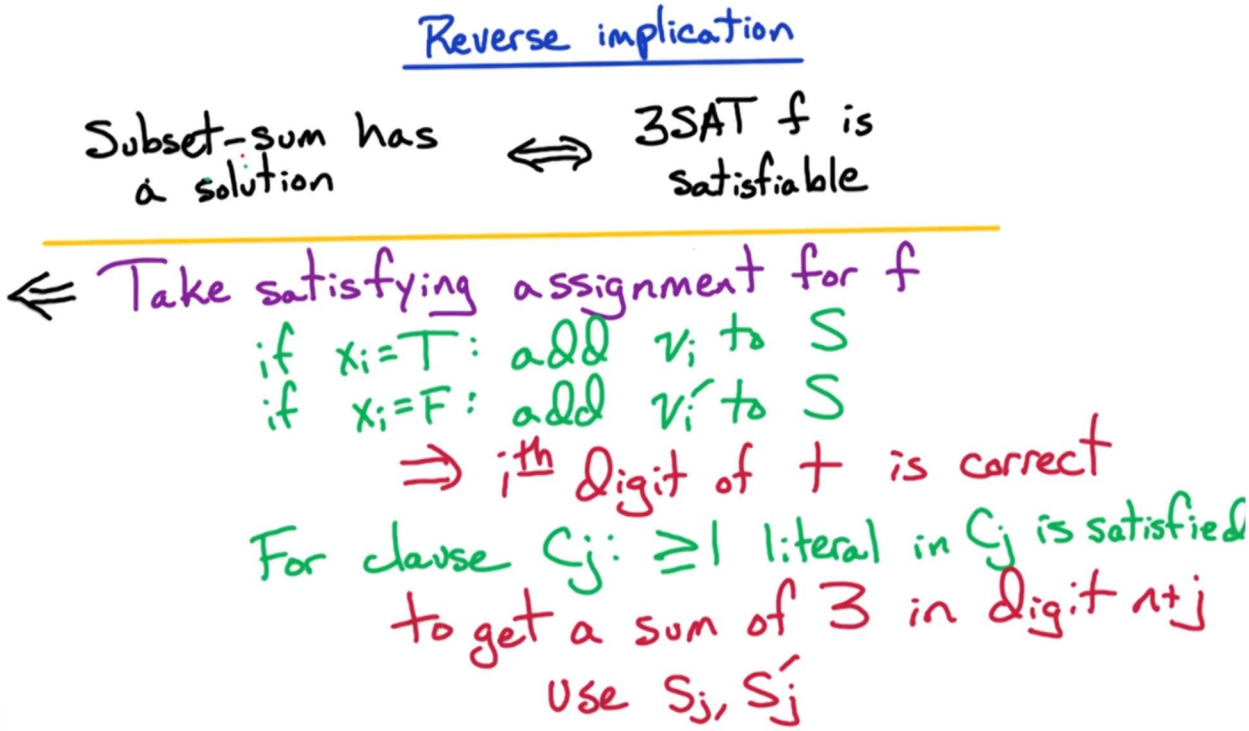
And therefore, we have a satisfying assignment for f. So we've shown that a solution to this Subset-sum instance corresponds to a satisfying assignment for our 3SAT formula. And. we've actually shown how to transform this solution to the subset sum problem to find the satisfying assignment to the 3SAT's formula.

Now let's prove the reverse implication. Let's prove that if we have a satisfying assignment for this 3SAT formula, it corresponds to the solution to the subset sum problem.

***

LESSON: NP4: Knapsack

(Slide 12) Reverse Implication



A satisfying solution for 3SAT => A solution for Subset-sum:

Let's prove the reverse implication that a satisfying assignment for the 3SAT formula corresponds to a solution to the subset-sum instance. So let's take a satisfying assignment for f, and we'll construct a solution to the subset-sum instance in the following manner:

- In our satisfying assignment, if we set the variable xi to True, then we add the number vi to the solution S.
- And if we set the variable xi to False in our satisfying assignment, then we add the number of vi' to our solution S.

We're adding exactly one of vi and vi' to S. This means, in the ith digit, we get exactly 1. So, the ith digit of t is correct.

- Now let's look at the jth clause. We started from a satisfying assignment. Therefore, at least one of the literals in this clause is satisfied. The numbers corresponding to these literals give us a sum of 1, 2, or 3 in the digit n+j. t has 3

in digit n+j. So how do we achieve a sum of 3 in digit n+j? Well, using the numbers corresponding to these literals, we get a sum of 1, 2, or 3, and then we can use sj and/or sj' to get up to a sum of 3.

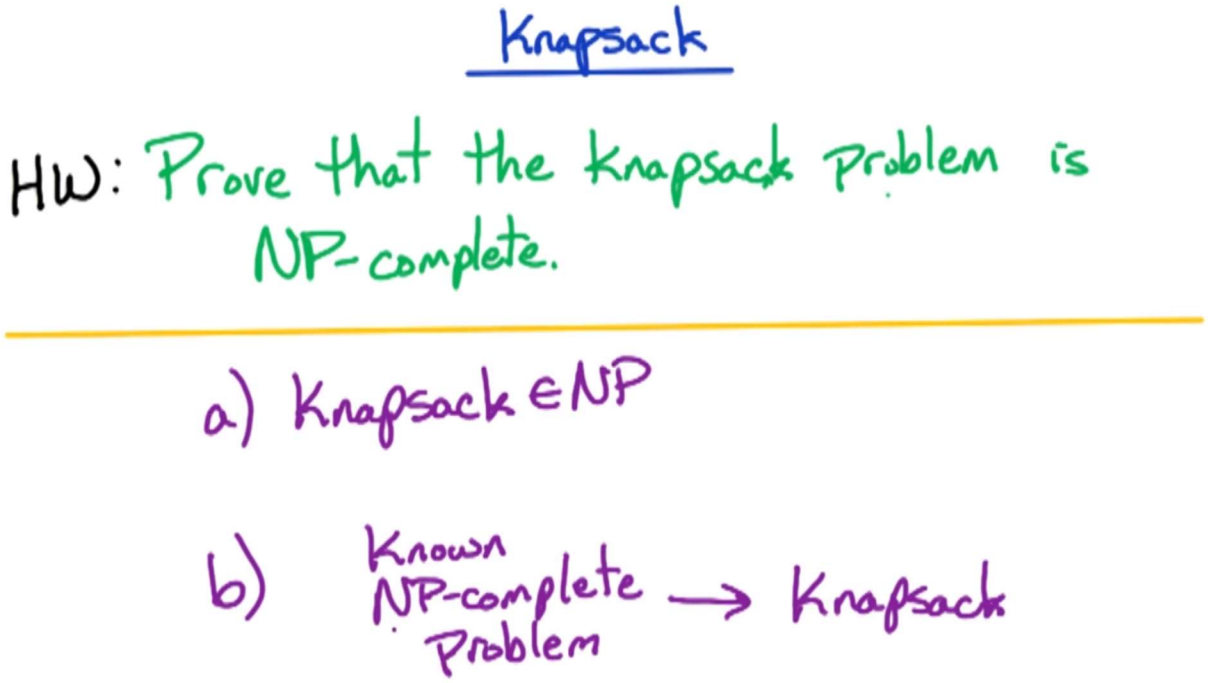
This ensures that digit n+j is correct and therefore, the last m digits are correct and the first digits are correct. And therefore, we have a solution to the subset-sum instance.

So we've proved this equivalence and we've proved that our reduction is correct.

***

LESSON: NP4: Knapsack

(Slide 13) Knapsack is NP-complete



You just proved that the subset sum problem is NP-complete. Now a nice homework problem is to use that fact that subset sum problem is NP-complete and then prove that the Knapsack problem is NP-complete. And make sure that you use the correct version of the Knapsack problem - the version that which lies in the class NP.

Now to prove that the Knapsack problem or any problem is NP-complete, there's two parts:

1. You first have to prove that the desired problem lies in the class NP,
2. And, second, you have to take a known NP-complete problem and you have to reduce the known NP-complete problem to this new problem - in this case, the Knapsack problem.

We know that all problems in NP reduce to this known NP-complete problem, therefore if we show a reduction from this known NP-complete problem to the Knapsack problem, then we know that all problems in NP reduce to the Knapsack problem.

Which problem are you going to use for this part? Well I'm not trying to trick you, so what are you going to use is the problem that looks most similar to the Knapsack problem - in this case, this is Subset-sum problem.

On the homework or exam, if you have a graph problem which you're trying to prove is NP-complete, what are you going to use here? Probably a graph problem which sounds similar or looks similar to the desired problem. If you have a logic problem you're trying to prove is NP-complete and probably you're going to use SAT or 3SAT over here (as the known problem).

Known NP-complete problem -> New Problem!

> The other thing to make sure of is that you're doing the reduction in the correct direction. So you're assuming you have a polynomial time algorithm for this new problem and you are using that to solve this known problem in polynomial time. So you have taken input to this known NP-complete problem and reduce it / transform it into an input for this new problem. Then if we take a solution to this new problem then you can transform it back to a solution to this original problem. So you have to transform input from this known NP-complete problem to input to this new problem.
>
> That's a common mistake that students make, even though they write the direction this way, they actually do the reduction in the other direction. They take inputs for this new problem and transform it to input to this known problem. You have to make sure you're doing it the correct direction. So you take inputs to the known NP-complete problem and transform it to inputs to this new problem.