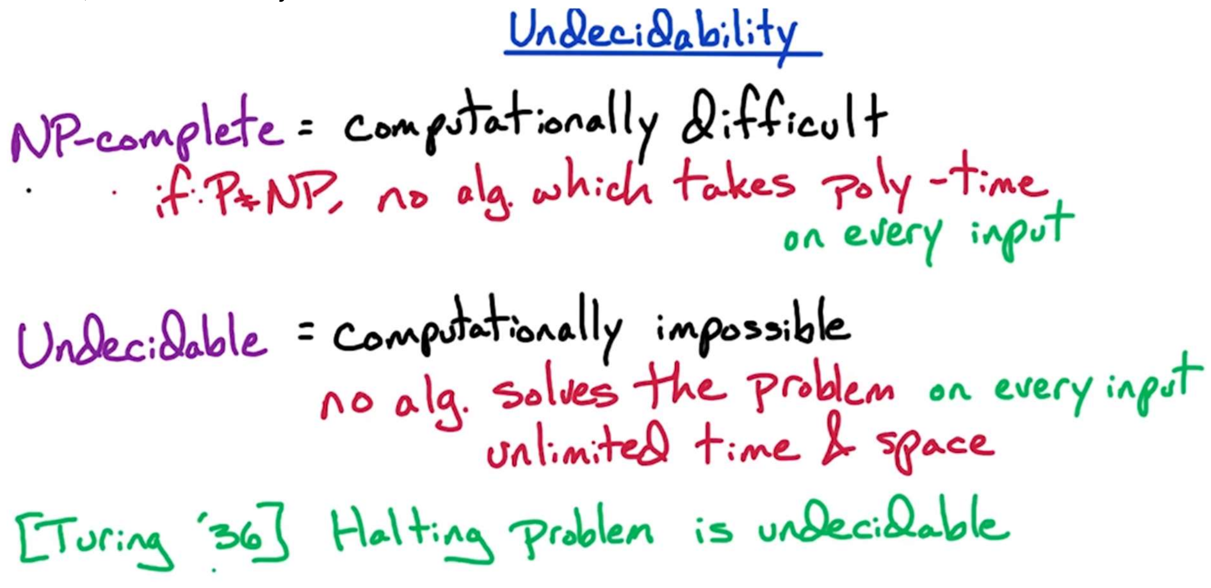
LESSON: NP5: Halting problem

***

(Slide 1) Undecidability



Undecidability

NP-complete = computationally difficult

if P≠NP, no alg. which takes poly-time on every input

Undecidable = computationally impossible

no alg. solves the problem on every input

unlimited time & space

[Turing '36] Halting problem is undecidable

We've now seen many NP-complete problems. When a problem is NP-complete, it signifies to us that it's computationally difficult. Formally what does that mean? That means it is the most difficult problem in the class NP. So, if we can solve this NP-complete problem in polynomial time, we can solve all problems in the class NP in polynomial time. And, now, since there are literally thousands of problems in the class NP (from all scientific fields), it's unlikely that we're going to derive a polynomial time algorithm for our NP-complete problem.

To be precise, if P is not equal to NP, then that implies that there's no algorithm which can run in polynomial time on every input for this NP-complete problem. Notice the important distinction is on every input. We may have an algorithm which takes polynomial time on some inputs, or even on most inputs or almost every input; but, there's no algorithm which is guaranteed to take polynomial time on every input.

Now we're going to look at the class of **undecidable** problems. These are problems which are computationally impossible. For an NP-complete problem, it's unlikely to have an algorithm which solves the problem in polynomial time. In contrast, for an undecidable problem, there is no algorithm which solves the problem on every input regardless of the running time of the algorithm - you run polynomial time, exponential time - there's no algorithm which is going to solve it on every input.

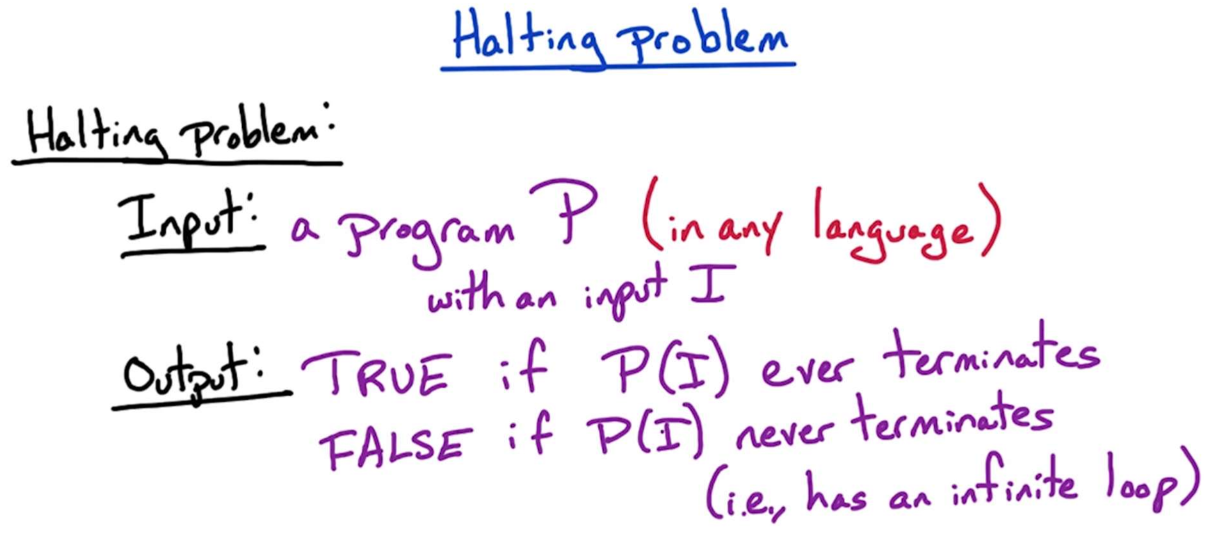
Now, in 1936, the great Alan Turing proved that the Halting problem is undecidable. And we're going to see the idea of that result now. Now, this paper by Turing in 1936 introduced the notion which we now refer to as a Turing Machine. What Turing showed is that the Halting problem is undecidable on a Turing Machine and a Turing Machine captures the power of a conventional computer.

Now, by convention, we're excluding things like quantum computers.   Now, later, many other problems were shown to be undecidable; but, the Halting problem is quite nice, so we're going to dive into that proof.

***

LESSON: NP5: Halting problem

(Slide 2) Halting problem



Halting problem

Halting problem:

Input: a program P (in any language)
with an input I

Output: TRUE if P(I) ever terminates
FALSE if P(I) never terminates
(i.e., has an infinite loop)

Now let's formally define the Halting problem:

- The input to the Halting problem is a program P, with an input I for that program P.

  Now how is this program P given to you? Well, we can restrict it to any language we want. We can say it's in pseudocode; we can say it's in C, Python, etc.; but, we can say it's in an arbitrary language.

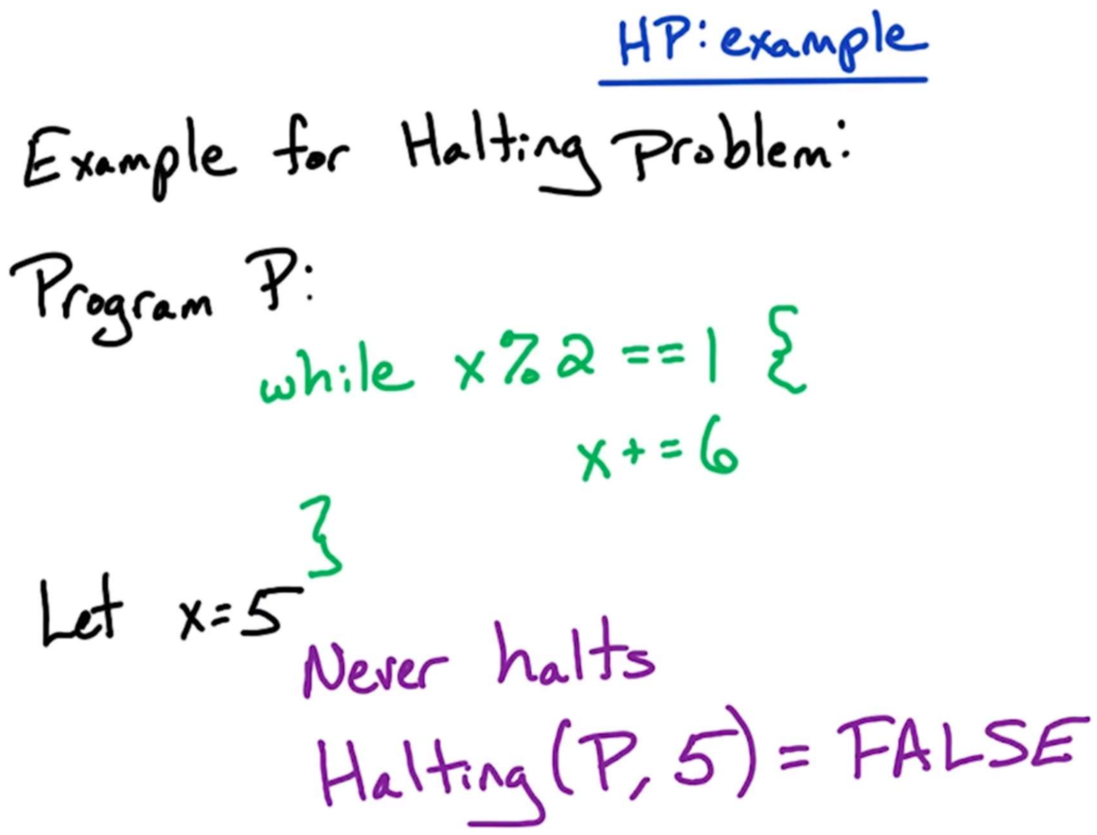
- Now what is the output of the Halting problem? Well, think of the basic task of a compiler. Given a program and an input, we want to figure out if this program terminates on this input, or does it have an infinite loop? That's the task of the Halting problem - to figure out whether this program on this particular input runs forever, or ever terminates.

  So if the program P on input I ever terminates - so it stops eventually – then, we output True. We're not trying to figure out whether it runs correctly, whether it gives a correct solution on this input. We're just asking whether the program P ever stops when we run it on input I.

  On the other side, we output false if the program P on input I never terminates. In other words, it has an infinite loop.

To summarize, I'm giving you a particular program P, and a specific input I, and on this specific input I, does the program P have an infinite loop? Or does it eventually stop?

***
LESSON: NP5: Halting problem
(Slide 3) HP: Example



HP: example

Example for Halting Problem:

Program P:

```
while x%2 == 1 {
        x += 6
}
```

Let x=5

Never halts

Halting(P, 5) = FALSE

Let's look an example instance for the Halting problem. Let's look at the following simple program P. And I'll attempt to write it in C.

> My program P consists of one input variable, x, and it consists of one while loop. The while loop checks whether x mod 2 is equal to one. So, it checks whether x is odd. If x is odd, then it adds 6 to X and it repeats. And it keeps going until x is even.
>
> Now let's look at this program P on input X equals five. Now for this simple program, it's easy to see what will happen:
>
> > x starts at five, and then it'll be 11, and then 17, 23, and so on. x will always be odd.
>
> So the program never halts. This is going to be an infinite loop. Now of course, this is

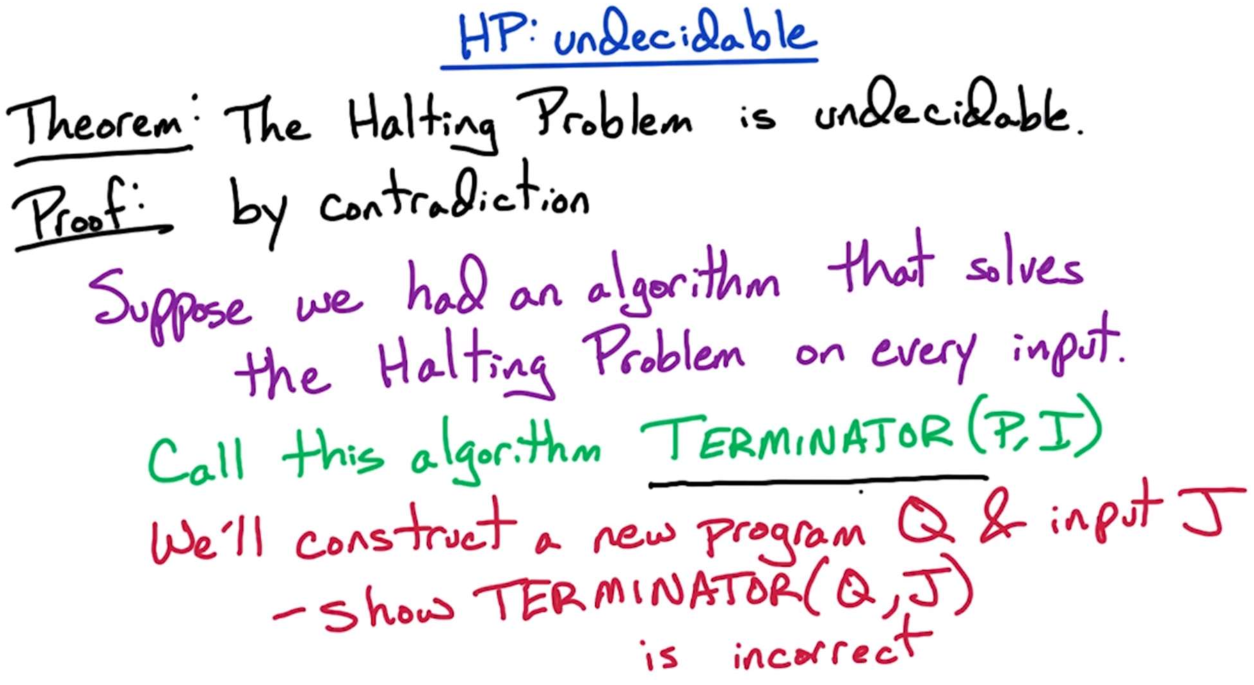
assuming infinite memory – so, there's no overflows or anything like that.

Therefore, if we look at Halting with this pair of inputs, program P and this input x = 5, then this program has an infinite loop on this particular input. Therefore, Halting(P,5) is False because the program P never terminates on this particular input (5).

***

LESSON: NP5: Halting problem

(Slide 4) HP: Undecidable



HP: undecidable

Theorem: The Halting Problem is undecidable.

Proof: by contradiction

Suppose we had an algorithm that solves the Halting Problem on every input.

Call this algorithm TERMINATOR(P, I)

We'll construct a new program Q & input J

- show TERMINATOR(Q, J) is incorrect

Now, let's prove the theorem that the Halting problem is undecidable.

How can we hope to prove this theorem? We can't expect to come up with an incredibly difficult program for which no algorithm can solve the Halting problem on that program. It's more like, for every algorithm, there is a program for which the algorithm fails. So, the way we go about proving this theorem is by contradiction.

Proof by contradiction:

> Suppose that we had an algorithm that solves the Halting problem on every input. Now we're going to derive a contradiction and therefore, our assumption that there exists an algorithm which solves the Halting problem is not true, and, therefore, there is no algorithm which solves the Halting problem.
>
> What we're going to do is for this particular algorithm which solves the Halting problem, we're going to construct an input for which this algorithm is incorrect. And therefore, our assumption that this algorithm solves the Halting problem on every input is incorrect. And we have a contradiction and therefore, that will prove the theorem.

Now let's give a name for this algorithm. Now this algorithm is determining whether a particular program on a specific input terminates or not. So, let's call this algorithm Terminator.

Terminator takes a pair of inputs, P and I.   P is a program, I is an input for this program, and Terminator outputs True or False depending on whether this program P, on this particular input I, terminates eventually or not. If it eventually terminates, then it outputs True. If it has an infinite loop, then it outputs False. And we're assuming that Terminator is correct. It solves the Halting problem for every program P, and every input I.

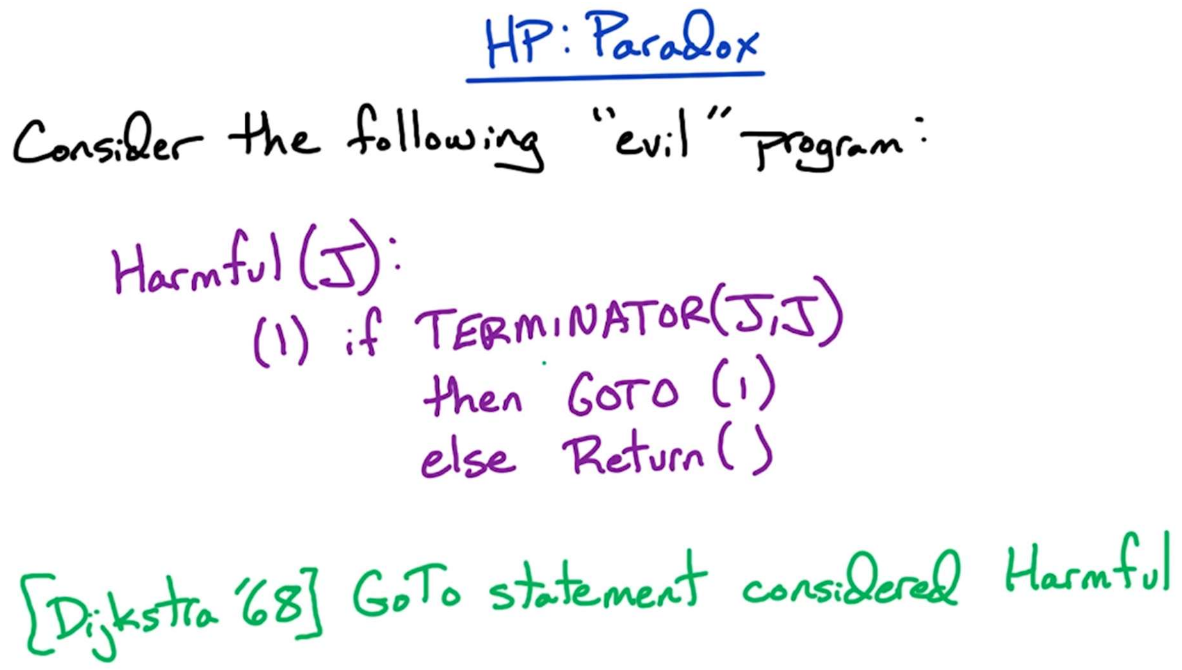
Now we're going to construct a program Q and an input J, and we're going to show that when we run Terminator on this input pair Q,J, then its output is incorrect. Since Terminator is incorrect on this pair of inputs, therefore, Terminator does not solve the Halting problem on every input. So this will give us our contradiction, and therefore, that would complete the proof by contradiction.

Now, how can we hope to construct this program Q? Well, one important piece is that we're assuming the existence of this program, Terminator. So we can use this algorithm, Terminator as a subroutine in our new algorithm Q. Now I don't know anything about the inner workings of Terminator so I have to use it as a black box, but I can use this as a subroutine. So we're going to use it as a subroutine to get our paradox or contradiction.

***

LESSON: NP5: Halting problem

(Slide 5) HP: Paradox



HP: Paradox

Consider the following "evil" program:

```
Harmful(J):
    (1) if TERMINATOR(J,J)
        then GOTO (1)
        else Return( )
```

[Dijkstra '68] GoTo statement considered Harmful

So consider the following evil program, it's evil in the sense that Terminator algorithm that we assumed existed is going to fail on this program.

This new program that I'm going define now. I'm going to call Harmful. Harmful has one input J.

```
Harmful(J):
    1. If TERMINATOR(J,J)
        Then GOTO (1)
        else Return()
```

First line of Harmful is an if statement. We run this Terminator Algorithm which we assumed existed on input J and J. So we use this input for the Harmful program as the program P and the input I for the Terminator algorithm. Now Terminator returns True or False. Now if it returns True then I'm going to go to step (1). So I have this loop.

If Terminator returns False, then I simply exit the procedure. Now why did I call this procedure Harmful – well, this was just a reference to Dijkstra's article from 1968, which was titled "GoTo statement is considered Harmful". Anyways I'm allowed to use whatever Programming language I want and any programming style I want. So I choose to use this GoTo statement. It's going to make the loop more apparent in the algorithm.

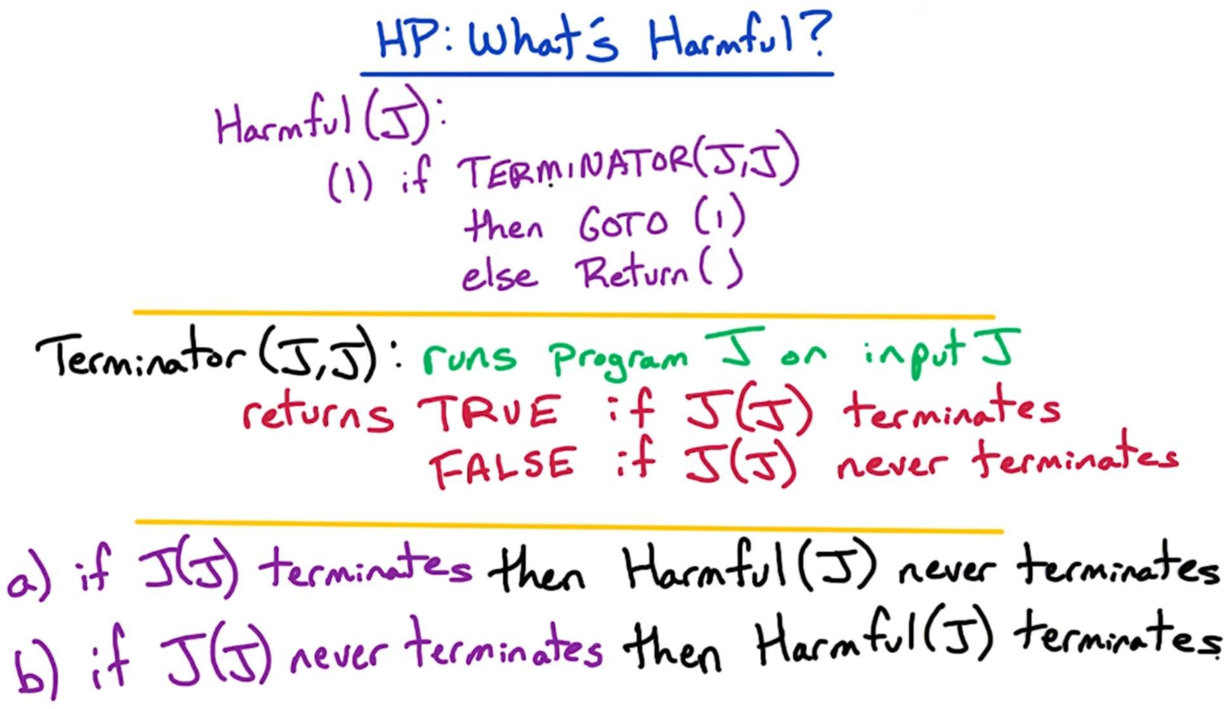
Let's summarize what we just did. We assume the existence of this algorithm Terminator which solves the Halting problem on every input. Now we're going to construct a new program which we call Harmful for which the Terminator will show fails. This new program Harmful simply has one line. It's just an if-then-else statement.

Now we're using this supposed algorithm Terminator as a blackbox in our Harmful algorithm so Harmful takes an input J. What we do is we run Terminator on this input pair (J,J). So J is a program and J is the input to the program J. And we run Terminator on this pair. If Terminator returns True on this input pair then our program goes to (1). So we get this loop. If Terminator returns False then we simply exit this procedure and we exit the program Harmful.

***

LESSON: NP5: Halting problem

(Slide 6) HP: What’s Harmful?



HP: What's Harmful?

```
Harmful(J):
    (1) if TERMINATOR(J,J)
            then GOTO (1)
            else Return()
```

Terminator(J,J): runs program J on input J
returns TRUE if J(J) terminates
FALSE if J(J) never terminates

a) if J(J) terminates then Harmful(J) never terminates

b) if J(J) never terminates then Harmful(J) terminates

Let's detail once again what's happening in this simple program.

We're running Terminator on input (J, J). What does that do? That runs program J on input J, and Terminator returns True if program J on input J terminates.

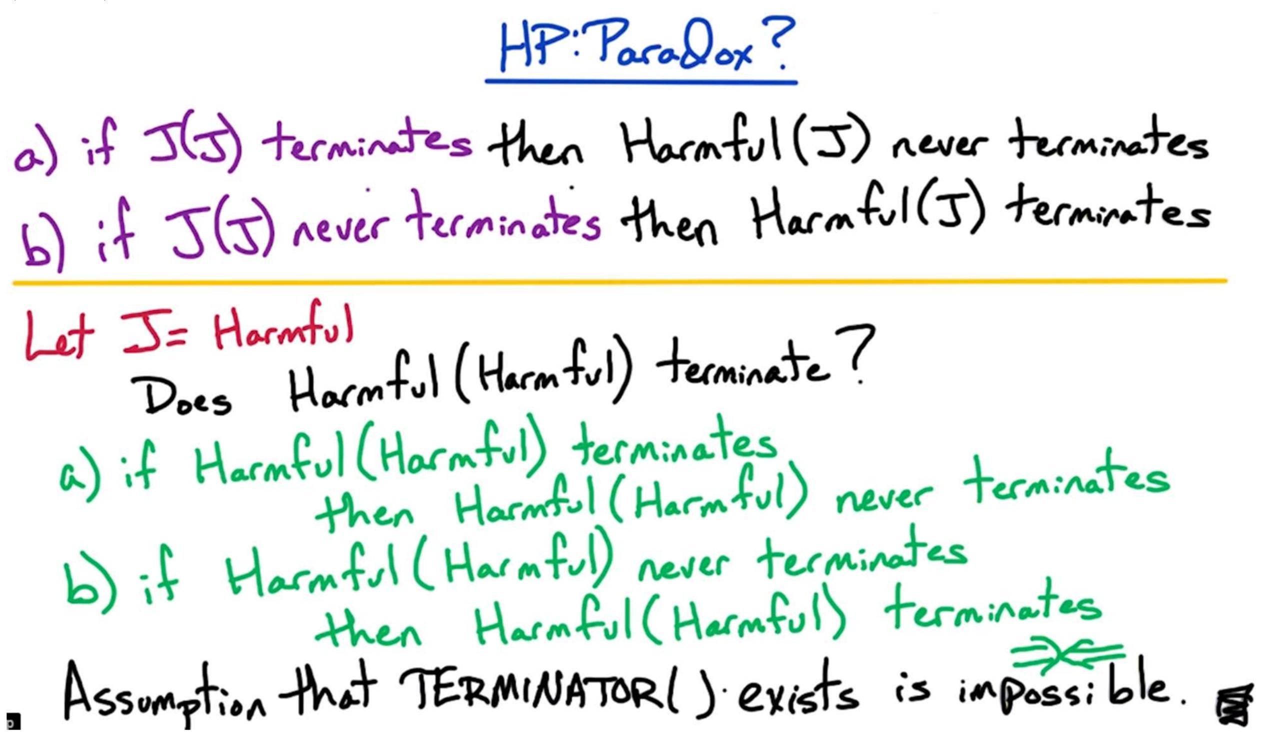
It eventually halts. And it returns False if J on input J never terminates. It has an infinite loop. So there are two cases - either Terminator(J, J) returns True or False. In which case, we either go to the THEN statement or the ELSE statement.

Now, if program J on input J terminates so we get a True, then, in this case, what happens in the program? So, we go to the THEN statement and we have a GOTO(1) and then we have an infinite loop. So, in this case, Harmful(J) never terminates. It has an infinite loop. Because, if TERMINATOR(J,J) returns true, then we go to (1) and we have this loop. In the other case, if we get False, then we go to the ELSE statement and we exit the program right away. So Harmful(J) terminates.

***

LESSON: NP5: Halting problem

(Slide 7) HP: Paradox derived



HP: Paradox?

a) if J(J) terminates then Harmful(J) never terminates

b) if J(J) never terminates then Harmful(J) terminates

Let J = Harmful

Does Harmful(Harmful) terminate?

a) if Harmful(Harmful) terminates
then Harmful(Harmful) never terminates

b) if Harmful(Harmful) never terminates
then Harmful(Harmful) terminates

Assumption that TERMINATOR( ) exists is im$\overset{\Rightarrow\Leftarrow}{\text{possi}}$ble.

Here's our summary from the previous slide.

If program J on input J terminates, then our new evil program harmful never terminates. It gets into this GOTO loop, and then it has an infinite loop.

On the other hand, if program J on input J never terminates - so it has an infinite loop – then, our new evil program goes into the else statement and it exits right away. And, therefore it terminates.

Now, we need to derive a paradox or a contradiction. What we do is we set the input J to be this program Harmful which we just defined. So Harmful is this short one line program, and we use that as the input to Harmful itself.

Now the question is, does the program Harmful, when it takes itself as input, does it terminate or not? Well there are two possibilities - either it does terminate or it has an infinite loop. Let's consider both possibilities.

- (Case: Harmful terminates) Suppose program Harmful, when it takes itself as input, does terminate.

  Going up to the above summary, if J on J terminates, then harmful on J never terminates because it gets into this GOTO statement. So plugging in J = Harmful, we have that Harmful on Harmful never terminates.

  So if the program Harmful on itself as input terminates, then harmful on itself is input never terminates. That's a contradiction. Therefore this can't be the case.

- (Case: Harmful never terminates) So it must be the case that when we run program Harmful on itself as input, then it never terminates. What happens in this scenario?

  Well, let's look above - in this scenario when J on J never terminates, then Harmful on J terminates because it goes into this else statement, and it exits the program Harmful immediately. Once again, if the program harmful when it takes itself as input never terminates, then what we conclude is that harmful on itself terminates.

  Again, we have a contradiction, so this can't be the case.

So it can't be the case that it terminates, and it can't be the case that it never terminates. Since either case leads to a contradiction, our initial assumption that the program TERMINATOR which solves the Halting problem on every input exists must be impossible. And therefore there does not exist a program which solves the Halting problem on every input.

And that completes the proof of the theorem.