CMSC 451: The classes P and NP

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 8.3 of *Algorithm Design* by Kleinberg & Tardos.

Computational Complexity

- We've seen algorithms for lots of problems, and the goal was always to design an algorithm that ran in polynomial time.

- Sometimes we've claimed a problem is NP-hard as evidence that no such algorithm exists.

- Now, we'll formally say what that means.

Decision Problems

Decision Problems:

- Usually, we've considered optimization problems: given some input instance, output some answer that maximizes or minimizes a particular objective function.
- Most of computational complexity deals with a seemingly simpler type of problem: the decision problem.
- A decision problem just asks for a `yes` or a `no`.
- We phrased Circulation with Demands as a decision problem.

Optimization $\rightarrow$ Decision

Recall this problem from a few weeks ago:

Weighted Interval Scheduling

Given a set of intervals $\{I_i\}$ each with a nonnegative weight w_i, find a subset of intervals S such that $\sum_{i \in S} w_i$ is maximized.

We can change this into a decision problem by asking:

Weighted Interval Scheduling

Given a set of intervals $\{I_i\}$ each with a nonnegative weight w_i, is there a subset of intervals S such that $\sum_{i \in S} w_i$ is greater than C.

Decision is no harder than Optimization

The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of Weighted Interval Scheduling?

Decision is no harder than Optimization

The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of Weighted Interval Scheduling?

- If you could solve the optimization version and got a solution of value M, then you could just check to see if $M > C$.
- If you can solve the optimization problem, you can solve the decision problem.
- If the *decision* problem is hard, then so is the optimization version.

Decision → Optimization

We can often also go from decision to optimization:

Independent Set

Given graph G and a number k, does G contain a set of at least k independent vertices?

How can we find the largest k for which G contains an independent set of size k?

Decision → Optimization

We can often also go from decision to optimization:

Independent Set

Given graph G and a number k, does G contain a set of at least k independent vertices?

How can we find the largest k for which G contains an independent set of size k?

Try every possible value of k — there are only n of them.
Or even better use binary search.

Encoding an Instance

We can encode an instance of a decision problem as a string.

Example. The encoding of a Weighted Interval Set instance with 3 intervals might be:

$$s_1, e_1, w_1; s_2, e_2, w_2; s_3, e_3, w_3; ; C$$

More explictly,

`1,10,5;3,7,20;12,15,1;;10`

How do we "know" intuitively that all of the problems we've considered so far can be encoded as a single string?

Encoding an Instance

We can encode an instance of a decision problem as a string.

Example. The encoding of a Weighted Interval Set instance with 3 intervals might be:

$$s_1, e_1, w_1; s_2, e_2, w_2; s_3, e_3, w_3; ; C$$

More explictly,

`1,10,5;3,7,20;12,15,1;;10`

How do we "know" intuitively that all of the problems we've considered so far can be encoded as a single string?

Because we can represent them in RAM as a string of bits!

Decision Problems and Languages

A decision problem X is really just sets of strings:

String	$\in X$?
`1,10,5;3,7,20;12,15,1;;10`	Yes
`1,10,5;3,7,20;12,15,1;;100`	No
⋮	⋮

Def. A language is a set of strings.
(Analogy: English is the set of valid English words.)

Hence, any decision problem is equivalent to deciding membership in some language.

We talk about "decision problems" and "languages" pretty much interchangeably.

Recap

Computational complexity primarily deals with decision problems.

A decision problem is no harder than the corresponding optimization problem.

A decision problem can be thought of as a set of the strings that encode "`yes`" instances.

Such sets are called languages.

How can we say a decision problem is hard?

A Model of Computation

Ultimately, we want to say that "a computer can't recognize some language efficiently."

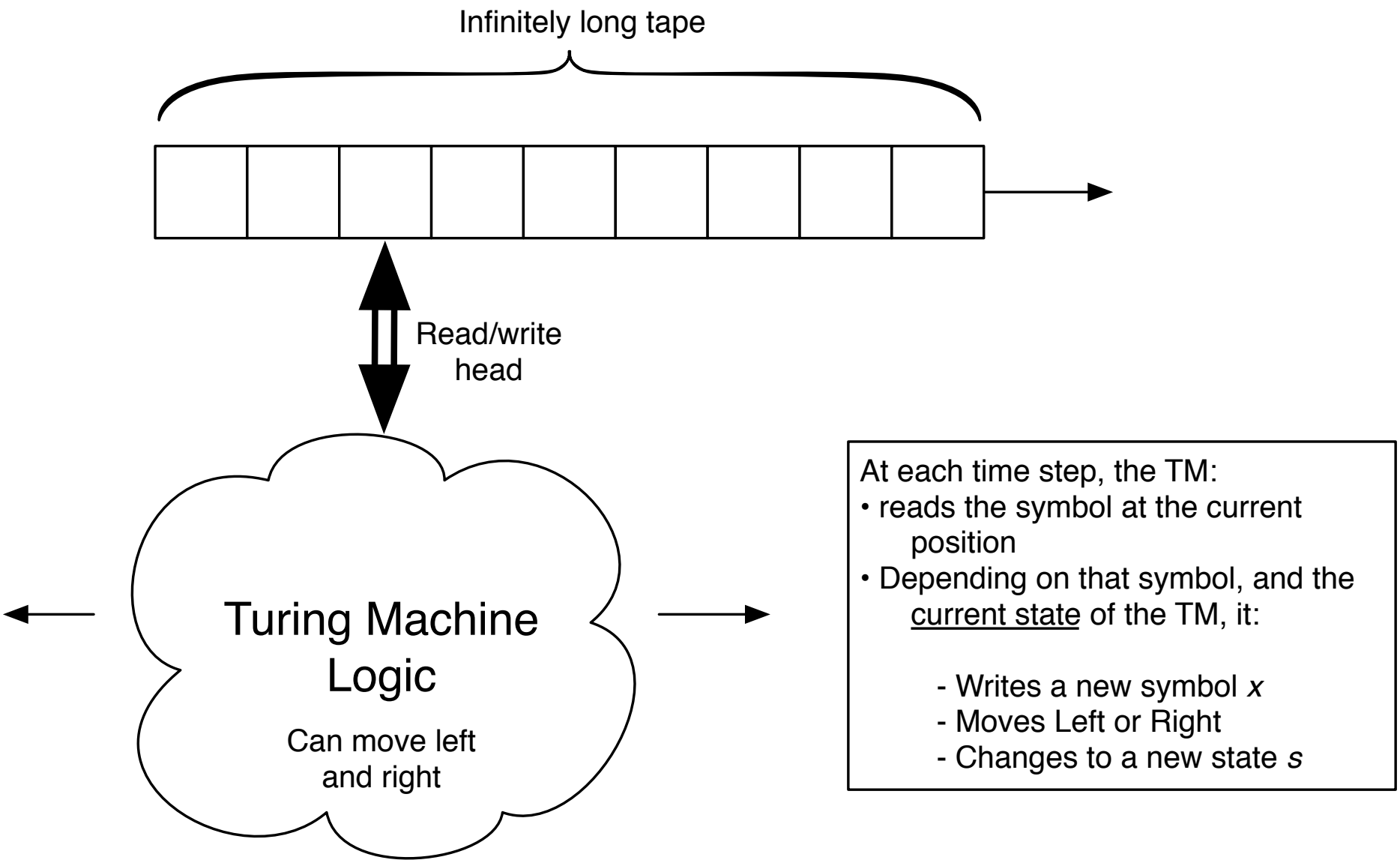
To do that, we have to decide what we mean by a *computer*.

We will mean a Turing Machine.

Church-Turing Thesis

Everything that is efficiently computable is efficiently computable on a Turing Machine.

Turing Machine



At each time step, the TM:

- reads the symbol at the current position
- Depending on that symbol, and the current state of the TM, it:

  - Writes a new symbol *x*
  - Moves Left or Right
  - Changes to a new state *s*

The class **P**

P is the set of languages whose memberships are decidable by a Turing Machine that makes a polynomial number of steps.

By the Church-Turing thesis, this is the "same" as:

P is the set of decision problems that can be decided by a computer in a polynomial time.

From now on, you can just think of your normal computer as a Turing Machine — and we won't worry too much about that formalism.

The Class NP

Now that we have a different (more formal) view of **P**, we will define another class of problems called **NP**.

We need some new ideas.

Certificates

Recall the independent set problem (decision version):

Independent Set

Given a graph G, is there set S of size $\geq k$ such that no two nodes in S are connected by an edge?

Finding the set S is hard (we will see).

But if I give you a set S^*, checking whether S^* is the answer is easy: check that $|S| \geq k$ and no edges go between 2 nodes in S^*.

S^* acts as a certificate that $\langle G, k \rangle$ is a `yes` instance of Independent Set.

Efficient Certification

Def. An algorithm B is an efficient certifier for problem X if:

1. B is a polynomial time algorithm that takes two input strings I (instance of X) and C (a certificate).
2. B outputs either `yes` or `no`.
3. There is a polynomial $p(n)$ such that for every string I:

   $I \in X$ if and only if there exists string C of length $\leq p(|I|)$ such that $B(I, C) =$ `yes`.

B is an algorithm that can decide whether an instance I is a `yes` instance if it is given some "help" in the form of a polynomially long certificate.

Certifiers and Brute Force

Let's say you had an efficient certifier B for the Independent Set problem.

How could you use it in a brute force algorithm on instance I?

Certifiers and Brute Force

Let's say you had an efficient certifier B for the Independent Set problem.

How could you use it in a brute force algorithm on instance I?

Try every string C of length $\leq p(|I|)$ and ask is $B(I, C) = \texttt{yes}$?

The class NP

NP is the set of languages for which there exists an efficient certifier.

The class NP

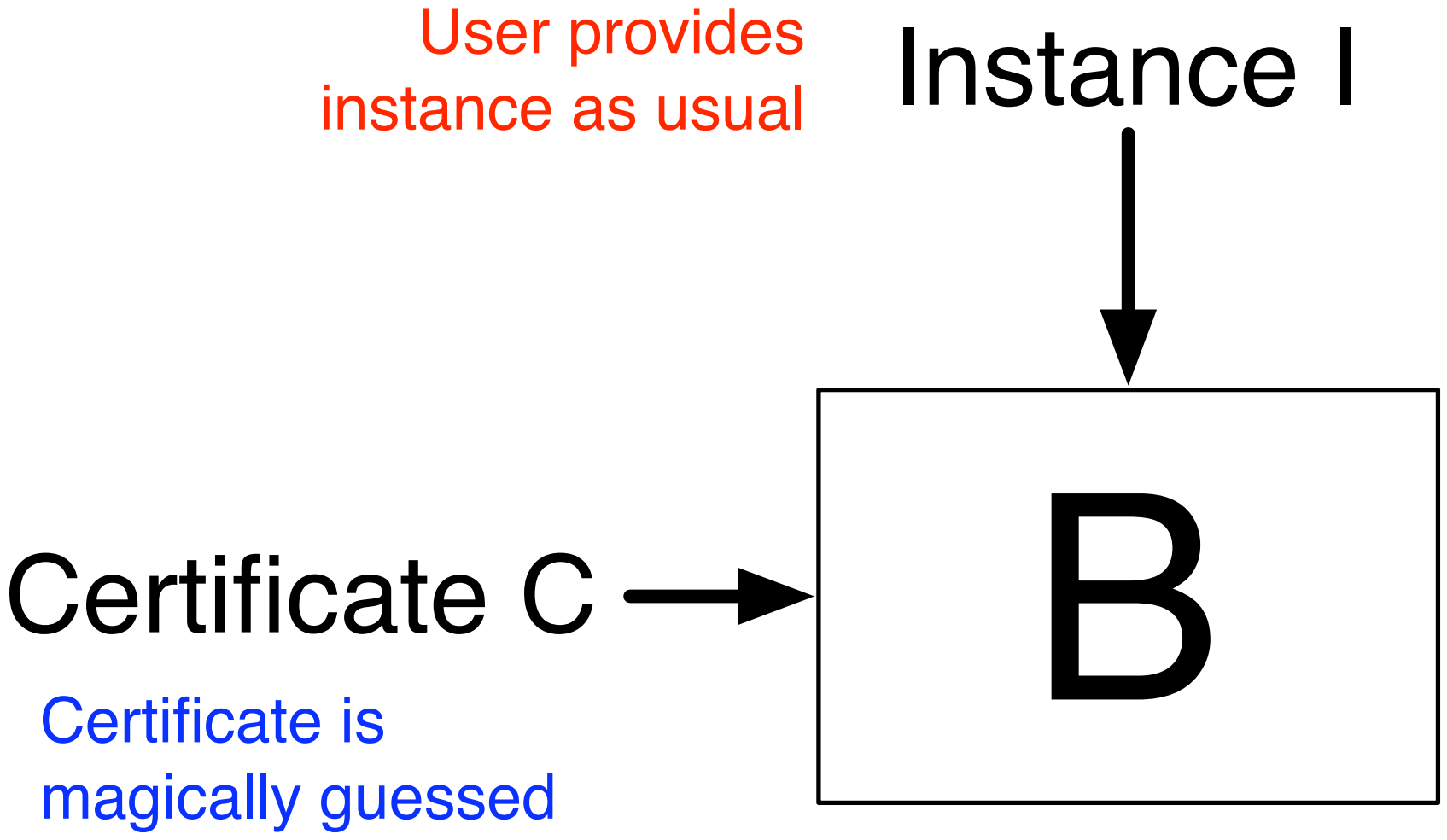
NP is the set of languages for which there exists an efficient certifier.

P is the set of languages for which there exists an efficient certifier that ignores the certificate.

That's the difference:

A problem is in **P** if we can decided them in polynomial time. It is in **NP** if we can decide them in polynomial time, if we are given the right certificate.

Do we have to find the certificates?



$\mathbf{P} \subseteq \mathbf{NP}$

Theorem

$\mathbf{P} \subseteq \mathbf{NP}$

Proof. Suppose $X \in \mathbf{P}$. Then there is a polynomial-time algorithm A for X.

To show that $X \in \mathbf{NP}$, we need to design an efficient certifier $B(I, C)$.

Just take $B(I, C) = A(I)$. $\square$

Every problem with a polynomial time algorithm is in **NP**.

$\mathbf{P} \neq \mathbf{NP}$?

The big question:

$$\mathbf{P} = \mathbf{NP}?$$

We know $\mathbf{P} \subseteq \mathbf{NP}$. So the question is:

Is there some problem in **NP** that is not in **P**?

Seems like the power of the certificate would help a lot.
But no one knows. . . .