




# Computational Complexity Theory

## Lecture 2: Class P & NP, Karp reductions, NP-completeness

Department of Computer Science,  
Indian Institute of Science

# Recap: Turing Machines

- An algorithm is a set of instructions or rules.
- To understand the performance of an algorithm we need a model of computation. Turing machine is one such *natural* model (introduced by Alan Turing in 1936).
- A TM consists of:
  - Memory tape(s)
  - A finite set of rules
- Turing machines  A mathematical way to describe algorithms.

# Recap: Turing Machines

- **Definition.** A  $k$ -tape Turing Machine  $M$  is described by a tuple  $(\Gamma, Q, \delta)$  such that
- $M$  has  $k$  memory tapes (input/work/output tapes) with *heads*;
- $\Gamma$  is a finite set of alphabets. (Every memory cell contains an element of  $\Gamma$ )
- $Q$  is a finite set of states. (special states:  $q_{\text{start}}$ ,  $q_{\text{halt}}$ )
- $\delta$  is a function from  $Q \times \Gamma^k$  to  $Q \times \Gamma^k \times \{L, S, R\}^k$



known as **transition function**; it captures the dynamics of  $M$

# Recap: TM Computation

- Start configuration.
  - All tapes other than the input tape contain blank symbols.
  - The input tape contains the input string.
  - All the head positions are at the start of the tapes.
  - The machine is in the start state  $q_{\text{start}}$ .
- Computation.
  - A **step of computation** is performed by applying  $\delta$ .
- Halting.
  - Once the machine enters  $q_{\text{halt}}$  it stops computation.

# Recap: TM Running time

- Let  $f: \{0,1\}^* \rightarrow \{0,1\}^*$  and  $T: \mathbb{N} \rightarrow \mathbb{N}$  and  $M$  be a Turing machine.
- **Definition.** We say  $M$  **computes**  $f$  if on every  $x$  in  $\{0,1\}^*$ ,  $M$  halts with  $f(x)$  on its output tape beginning from the start configuration with  $x$  on its input tape.
- **Definition.**  $M$  computes  $f$  in  $T(|x|)$  **time**, if for every  $x$  in  $\{0,1\}^*$ ,  $M$  halts within  $T(|x|)$  steps of computation and outputs  $f(x)$ .

# Recap: Uncomputability

- In this course, we would be dealing with
  - Turing machines that halt on every input.
  - Computational problems that can be solved by Turing machines.
- Can every computational problem be solved using Turing machines?

# Recap: Uncomputability

- There are problems for which there exists **no** TM that halts on every input instances of the problem and outputs the correct answer.
  - **Input:** A system of polynomial equations in many variables with integer coefficients.
  - **Output:** Check if the system has **integer solutions** .
  - **Question:** Is there an algorithm to solve this problem?
- **Theorem.** There doesn't exist any algorithm (realizable by a TM) to solve this problem. (Davis, Putnam, Robinson, Matiyasevich 1970)

# Recap: Power of Turing Machines

- TMs are natural and intuitive.
- **Church-Turing thesis:** *“Every physically realizable computation device – whether it’s based on silicon, DNA, neurons or some other alien technology – can be simulated by a Turing machine”.*
  - [quoted from Arora-Barak’s book]
- Several other computational models can be simulated by TMs.



# Recap: Power of Turing Machines

- TMs are natural and intuitive.
- **Strong Church-Turing thesis:** “Every *physically realizable computation device* – whether it’s based on silicon, DNA, neurons or some other alien technology – can be simulated *efficiently* by a Turing machine”.

Possible exception: Quantum machines!

# Basic facts about TMs

# Recap: Time constructible functions

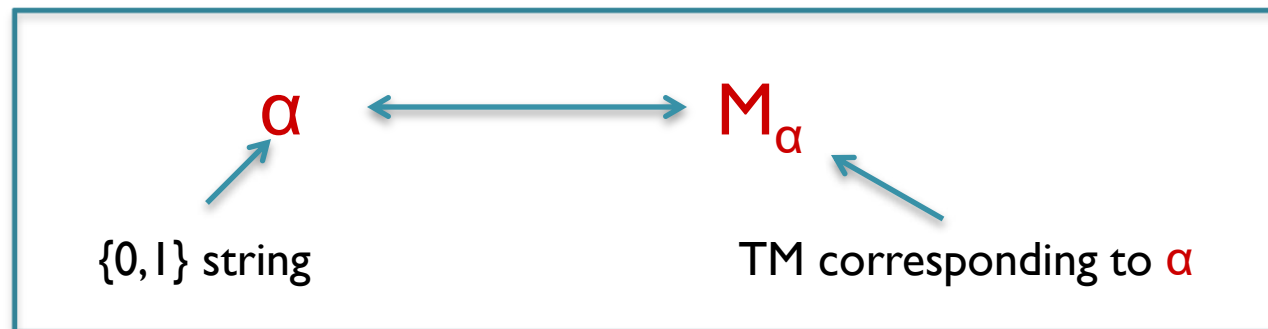
- **Time constructible functions.** A function  $T: \mathbb{N} \rightarrow \mathbb{N}$  is time constructible if  $T(n) \geq n$  and there's a TM that computes the function that maps  $x$  to  $T(\underbrace{|x|}_{\text{in binary}})$  in  $O(T(|x|))$  time.
- Examples:  $T(n) = n^2$ , or  $2^n$ , or  $n \log n$

# Recap: TM Robustness

- Let  $f: \{0,1\}^* \rightarrow \{0,1\}^*$  and  $T: \mathbb{N} \rightarrow \mathbb{N}$  be a time constructible function.
- Binary alphabets suffice.
  - If a TM  $M$  computes  $f$  in  $T(n)$  time using  $\Gamma$  as the alphabet set, then there's another TM  $M'$  that computes  $f$  in time  $4 \cdot \log |\Gamma| \cdot T(n)$  using  $\{0, 1, \text{blank}\}$  as the alphabet set.
- A single tape suffices.
  - If a TM  $M$  computes  $f$  in  $T(n)$  time using  $k$  tapes then there's another TM  $M'$  that computes  $f$  in time  $5k \cdot T(n)^2$  using a single tape that is used for input, work and output.

# Recap: TM as strings

- Every TM can be represented by a finite string over  $\{0,1\}$ .
- Every string over  $\{0,1\}$  represents some TM.
- Every TM has infinitely many string representations.



# Recap: TM as strings

- Every TM can be represented by a finite string over  $\{0,1\}$ .
- Every string over  $\{0,1\}$  represents some TM.
- Every TM has infinitely many string representations.
- A TM (i.e., its string representation) can be given as an input to another TM !!

# Recap: Universal Turing Machines

- **Theorem.** There exists a TM  $U$  that on every input  $x$ ,  $\alpha$  in  $\{0,1\}^*$  outputs  $M_\alpha(x)$ .
- Further, if  $M_\alpha$  halts within  $T$  steps then  $U$  halts within  $C \cdot T \cdot \log T$  steps, where  $C$  is a constant that depends only on  $M_\alpha$ 's alphabet size, number of states and number of tapes.
- Physical realization of UTMs are modern day electronic computers.

Complexity class P



# Decision Problems

- In the initial part of this course, we'll focus primarily on **decision problems**.

# Decision Problems

- In the initial part of this course, we'll focus primarily on **decision problems**.
- Decision problems can be naturally identified with **Boolean functions**, i.e., functions from  $\{0,1\}^*$  to  $\{0,1\}$ .

# Decision Problems

- In the initial part of this course, we'll focus primarily on **decision problems**.
- Decision problems can be naturally identified with **Boolean functions**, i.e., functions from  $\{0,1\}^*$  to  $\{0,1\}$ .
- Boolean functions can be naturally identified with sets of  $\{0,1\}$  strings, also called **languages**.

# Decision Problems

Decision problems  $\leftrightarrow$  Boolean functions  $\leftrightarrow$  Languages


- **Definition.** We say a TM  $M$  decides a language  $L \subseteq \{0,1\}^*$  if  $M$  computes  $f_L$ , where  $f_L(x) = 1$  if and only if  $x \in L$ .

The characteristic function of  $L$ .

# Complexity Class P

- Let  $T: \mathbb{N} \rightarrow \mathbb{N}$  be some function.
- **Definition:** A language  $L$  is in  $\text{DTIME}(T(n))$  if there's a TM that decides  $L$  in time  $O(T(n))$ .
- **Definition:** Class  $P = \bigcup_{c > 0} \text{DTIME}(n^c)$ .

# Complexity Class P

- Let  $T: \mathbb{N} \rightarrow \mathbb{N}$  be some function.
- **Definition:** A language  $L$  is in  $\text{DTIME}(T(n))$  if there's a TM that decides  $L$  in time  $O(T(n))$ .
- **Definition:** Class  $P = \bigcup_{c > 0} \text{DTIME}(n^c)$ .  


Deterministic polynomial-time

# Complexity Class P : Examples

- Cycle detection (*DFS*)
  - Check if a given graph has a cycle.

# Complexity Class P : Examples

- Cycle detection
- Solvability of a system of linear equations (*Gaussian elimination*)
  - Given a system of linear equations over  $\mathbb{Q}$  check if there exists a common solution to all the linear equations.



# Complexity Class P : Examples

- Cycle detection
- Solvability of a system of linear equations
- Perfect matching (*Edmonds 1965*) (birth of class P)
  - Check if a given graph has a perfect matching

# Complexity Class P : Examples

- Cycle detection
- Solvability of a system of linear equations
- Perfect matching
- Planarity testing (*Hopcroft & Tarjan 1974*)
  - Check if a given graph is planar

# Complexity Class P : Examples

- Cycle detection
- Solvability of a system of linear equations
- Perfect matching
- Planarity testing
- Primality testing (*Agrawal, Kayal & Saxena 2002*)
  - Check if a number is prime

# Polynomial-time Turing Machines

- **Definition.** A TM  $M$  is a *polynomial-time* TM if there's a polynomial function  $q: \mathbb{N} \rightarrow \mathbb{N}$  such that for every input  $x \in \{0,1\}^*$ ,  $M$  halts within  $q(|x|)$  steps.

**Polynomial function.**  $q(n) = O(n^c)$  for some constant  $c$ .

# Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.

# Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the **i-th** bit of the output and make it a decision problem.

(Is the **i-th** bit, on input **x**, **1**?)

# Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the **i-th** bit of the output and make it a decision problem.
- Alternatively, we define a class called **functional P** or **FP**.

# Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the **i-th** bit of the output and make it a decision problem.
- We say that a problem or a function  $f: \{0,1\}^* \rightarrow \{0,1\}^*$  is in **FP** (functional **P**) if there's a polynomial-time TM that computes **f**.



# Complexity Class FP : Examples

- Greatest Common Divisor (*Euclid ~300 BC*)
  - Given two integers **a** and **b**, find their gcd.

# Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG (*homework*)
  - Find the number of paths between two vertices in a directed acyclic graph.

# Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching (*Edmonds 1965*)
  - Find a maximum matching in a given graph

# Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching
- Linear Programming (*Khachiyan 1979, Karmarkar 1984*)
  - Optimize a linear objective function subject to linear (in)equality constraints

# Complexity Class FP : Examples

- Greatest Common Divisor

Not known if LP has a *strongly polynomial-time* algorithm.

- Counting paths in a DAG

**Homework:** Read about the differences between *strongly poly-time*, *weakly poly-time* and *pseudo poly-time* algorithms.

- Maximum matching

- Linear Programming (*Khachiyan 1979, Karmarkar 1984*)

➤ Optimize a linear objective function subject to linear (in)equality constraints

# Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching
- Linear Programming
- Factoring Polynomials (*Lenstra, Lenstra, Lovasz 1982*)
  - Compute the irreducible factors of a univariate polynomial over  $\mathbb{Q}$

Complexity class NP

# Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class **NP** captures the set of decision problems whose solutions are *efficiently verifiable*.



# Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.
- Class **NP** captures the set of decision problems whose solutions are *efficiently verifiable*.

Nondeterministic polynomial-time

# Complexity Class NP

- **Definition.** A language  $L \subseteq \{0,1\}^*$  is in **NP** if there's a polynomial function  $p: \mathbb{N} \rightarrow \mathbb{N}$  and a polynomial-time TM  $M$  (called the verifier) such that for every  $x$ ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

# Complexity Class NP

- **Definition.** A language  $L \subseteq \{0,1\}^*$  is in NP if there's a polynomial function  $p: \mathbb{N} \rightarrow \mathbb{N}$  and a polynomial-time TM  $M$  (called the verifier) such that for every  $x$ ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

$u$  is called a certificate or witness for  $x$  (w.r.t  $L$  and  $M$ ), if  $x \in L$ .

# Complexity Class NP

- **Definition.** A language  $L \subseteq \{0,1\}^*$  is in **NP** if there's a polynomial function  $p: \mathbb{N} \rightarrow \mathbb{N}$  and a polynomial-time TM **M** (called the verifier) such that for every  $x$ ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- It follows that verifier **M** cannot be fooled !

# Complexity Class NP

- **Definition.** A language  $L \subseteq \{0,1\}^*$  is in **NP** if there's a polynomial function  $p: \mathbb{N} \rightarrow \mathbb{N}$  and a polynomial-time TM **M** (called the verifier) such that for every  $x$ ,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- Class **NP** contains those problems (languages) which have such efficient verifiers.

# Class NP : Examples

- Vertex cover
  - Given a graph  $G$  and an integer  $k$ , check if  $G$  has a vertex cover of size  $k$ .

# Class NP : Examples

- Vertex cover
- 0/1 integer programming
  - Given a system of linear (in)equalities with integer coefficients, check if there's a **0-1** assignment to the variables that satisfy all the (in)equalities.

# Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
  - Given two numbers  $n$  and  $U$ , check if  $n$  has a prime factor less than or equal to  $U$ .



# Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
  - Given two graphs, check if they are isomorphic.

# Class NP : Examples

- 2-Diophantine solvability

➤ Given three integers  $a$ ,  $b$  and  $c$ , check if the equation  $ax^2 + by + c = 0$  has a solution  $(x, y)$ , where both  $x$  and  $y$  are positive integers.

[Homework]: Show that the above problem is in NP.

Hint: If  $(x, y)$  is a solution, then so is  $(x + b, y - a(2x + b))$ .

# Is $P = NP$ ?

- Obviously,  $P \subseteq NP$ .
- Whether or not  $P = NP$  is an outstanding open question in mathematics and TCS!

# Is $P = NP$ ?

- Obviously,  $P \subseteq NP$ .
- Whether or not  $P = NP$  is an outstanding open question in mathematics and TCS!
- Solving a problem does seem harder than verifying its solution, so most people believe that  $P \neq NP$ .

# Is $P = NP$ ?

- Obviously,  $P \subseteq NP$ .
- Whether or not  $P = NP$  is an outstanding open question in mathematics and TCS!
- $P = NP$  has many weird consequences, and if true, will pose a serious threat to secure and efficient cryptography (and e-commerce).

# Is $P = NP$ ?

- Obviously,  $P \subseteq NP$ .
- Whether or not  $P = NP$  is an outstanding open question in mathematics and TCS!
- Mathematics has gained much from attempts to prove such “negative” statements—Galois theory, Godel’s incompleteness, Fermat’s Last Theorem, Turing’s undecidability, Continuum hypothesis etc.

# Is $P = NP$ ?

- Obviously,  $P \subseteq NP$ .
- Whether or not  $P = NP$  is an outstanding open question in mathematics and TCS!
- Complexity theory has several of such intriguing unsolved questions.

*The history and status of the P versus NP question*

*– survey by Michael Sipser (1992)*

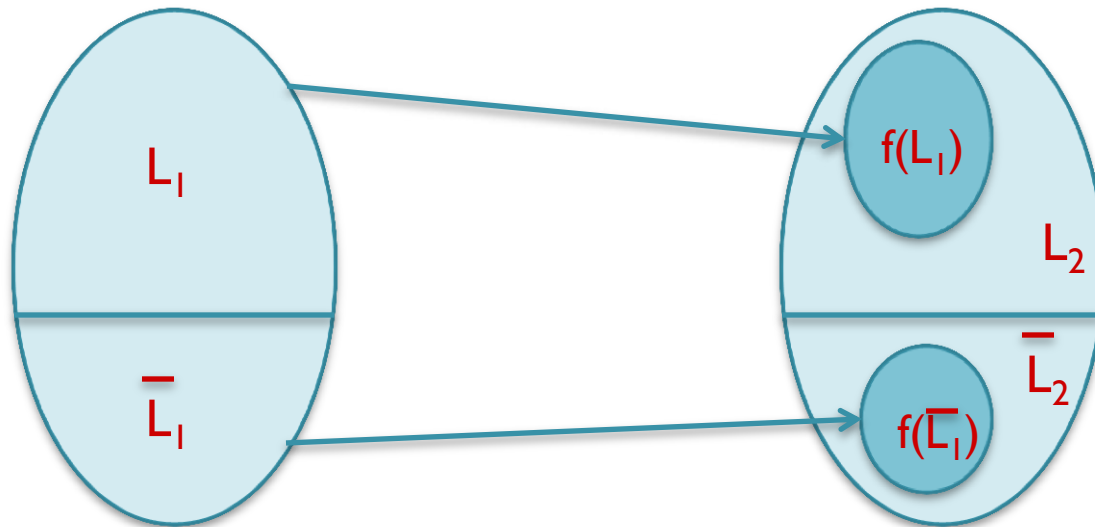
# Reductions



# Polynomial-time reduction

- **Definition.** We say a language  $L_1 \subseteq \{0,1\}^*$  is polynomial-time (Karp) reducible to a language  $L_2 \subseteq \{0,1\}^*$  if there's a polynomial-time computable function  $f$  s.t.

$$x \in L_1 \iff f(x) \in L_2$$



# Polynomial-time reduction

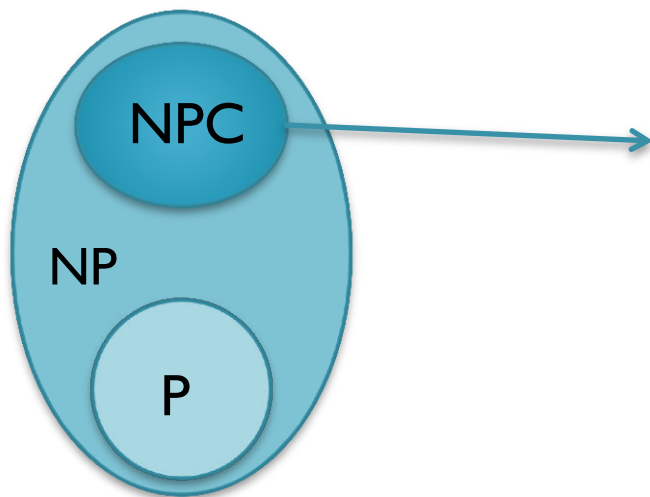
- **Definition.** We say a language  $L_1 \subseteq \{0,1\}^*$  is polynomial-time (Karp) reducible to a language  $L_2 \subseteq \{0,1\}^*$  if there's a polynomial time computable function  $f$  s.t.

$$x \in L_1 \iff f(x) \in L_2$$

- **Notation.**  $L_1 \leq_p L_2$
- **Observe.** If  $L_1 \leq_p L_2$  and  $L_2 \leq_p L_3$  then  $L_1 \leq_p L_3$ .

# NP-completeness

- **Definition.** A language  $L'$  is *NP-hard* if for every  $L$  in  $NP$ ,  $L \leq_p L'$ . Further,  $L'$  is *NP-complete* if  $L'$  is in  $NP$  and is NP-hard.
- **Observe.** If  $L'$  is NP-hard and  $L'$  is in  $P$  then  $P = NP$ . If  $L'$  is NP-complete then  $L'$  in  $P$  if and only if  $P = NP$ .



Hardest problems inside  $NP$  in the sense that if one NPC problem is in  $P$  then all problems in  $NP$  is in  $P$ .

# NP-completeness

- **Definition.** A language  $L'$  is *NP-hard* if for every  $L$  in  $NP$ ,  $L \leq_p L'$ . Further,  $L'$  is *NP-complete* if  $L'$  is in  $NP$  and is NP-hard.
- **Observe.** If  $L'$  is NP-hard and  $L'$  is in  $P$  then  $P = NP$ . If  $L'$  is NP-complete then  $L'$  is in  $P$  if and only if  $P = NP$ .
- **[Homework].** Let  $L_1 \subseteq \{0,1\}^*$  be any language and  $L_2$  be a language in  $NP$ . If  $L_1 \leq_p L_2$  then  $L_1$  is also in  $NP$ .

# Few words on reductions

- As to how we define a reduction from one language to the other (or one function to the other) is usually guided by a question on whether two complexity classes are different or identical.
- For polynomial-time reductions, the question is whether or not  $P$  equals  $NP$ .
- Reductions help us define *complete problems* (the 'hardest' problems in a class) which in turn help us compare the complexity classes under consideration.

# Class NP : Examples

- Vertex cover (NP-complete)
- 0/1 integer programming (NP-complete)
- 3-coloring planar graphs (NP-complete)
- 2-Diophantine solvability (NP-complete)
- Integer factoring (unlikely to be NP-complete)
- Graph isomorphism (Quasi-P) *Babai 2015*

# How to show existence of an NPC problem?

- Let  $L' = \{ (\alpha, x, l^m, l^t) : \text{there exists a } u \in \{0,1\}^m \text{ s.t. } M_\alpha \text{ accepts } (x, u) \text{ in } t \text{ steps} \}$
- **Observation.**  $L'$  is NP-complete.
- The language  $L'$  involves Turing machine in its definition. Next, we'll see an example of an NP-complete problem that is arguably more natural.