

# SAT, NP, NP-Completeness

Lecture 23

Nov 19, 2019

# Part I

## Reductions Continued

# Polynomial Time Reduction

## Karp reduction

A **polynomial time reduction** from a *decision* problem  $X$  to a *decision* problem  $Y$  is an *algorithm*  $\mathcal{A}$  that has the following properties:

- 1 given an instance  $I_X$  of  $X$ ,  $\mathcal{A}$  produces an instance  $I_Y$  of  $Y$
- 2  $\mathcal{A}$  runs in time polynomial in  $|I_X|$ . This implies that  $|I_Y|$  (size of  $I_Y$ ) is polynomial in  $|I_X|$
- 3 Answer to  $I_X$  YES *iff* answer to  $I_Y$  is YES.

Notation:  $X \leq_P Y$  if  $X$  reduces to  $Y$

## Proposition

If  $X \leq_P Y$  then a polynomial time algorithm for  $Y$  implies a polynomial time algorithm for  $X$ .

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.

# A More General Reduction

## Turing Reduction

### Definition (Turing reduction.)

Problem  $X$  polynomial time reduces to  $Y$  if there is an algorithm  $\mathcal{A}$  for  $X$  that has the following properties:

- 1 on any given instance  $I_X$  of  $X$ ,  $\mathcal{A}$  uses polynomial in  $|I_X|$  “steps”
- 2 a step is either a standard computation step, or
- 3 a sub-routine call to an algorithm that solves  $Y$ .

This is a **Turing reduction**.

# A More General Reduction

## Turing Reduction

### Definition (Turing reduction.)

Problem  $X$  polynomial time reduces to  $Y$  if there is an algorithm  $\mathcal{A}$  for  $X$  that has the following properties:

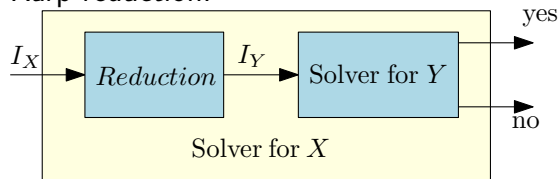
- 1 on any given instance  $I_X$  of  $X$ ,  $\mathcal{A}$  uses polynomial in  $|I_X|$  “steps”
- 2 a step is either a standard computation step, or
- 3 a sub-routine call to an algorithm that solves  $Y$ .

This is a **Turing reduction**.

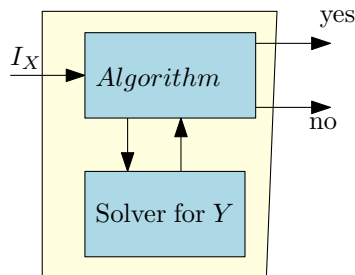
**Note:** In making sub-routine call to algorithm to solve  $Y$ ,  $\mathcal{A}$  can only ask questions of size polynomial in  $|I_X|$ . Why?

# Comparing reductions

## 1 Karp reduction:



## 2 Turing reduction:



## Turing reduction

- 1 Algorithm to solve  $X$  can call solver for  $Y$  many times.
- 2 Conceptually, every call to the solver of  $Y$  takes constant time.

# Relation between reductions

Consider two problems  $X$  and  $Y$ . Which of the following statements is correct?

- (A) If there is a Turing reduction from  $X$  to  $Y$ , then there is a Karp reduction from  $X$  to  $Y$ .
- (B) If there is a Karp reduction from  $X$  to  $Y$ , then there is a Turing reduction from  $X$  to  $Y$ .
- (C) If there is a Karp reduction from  $X$  to  $Y$ , then there is a Karp reduction from  $Y$  to  $X$ .
- (D) If there is a Turing reduction from  $X$  to  $Y$ , then there is a Turing reduction from  $Y$  to  $X$ .
- (E) All of the above.

# Example of Turing Reduction

## Problem (Independent set in circular arcs graph.)

**Input:** *Collection of arcs on a circle.*

**Goal:** *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

## Problem (Independent set of intervals.)

**Input:** *Collection of intervals on the line.*

**Goal:** *Compute the maximum number of non-overlapping intervals.*

How? Used algorithm for interval problem multiple times.



# Turing vs Karp Reductions

- 1 Turing reductions more general than Karp reductions.
- 2 Turing reduction useful in obtaining algorithms via reductions.
- 3 Karp reduction is simpler and easier to use to prove hardness of problems.
- 4 Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Completeness** proofs.
- 5 Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

# Propositional Formulas

## Definition

Consider a set of boolean variables  $x_1, x_2, \dots, x_n$ .

- 1 A **literal** is either a boolean variable  $x_i$  or its negation  $\neg x_i$ .
- 2 A **clause** is a disjunction of literals.  
For example,  $x_1 \vee x_2 \vee \neg x_4$  is a clause.
- 3 A **formula in conjunctive normal form (CNF)** is propositional formula which is a conjunction of clauses
  - 1  $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$  is a **CNF** formula.

# Propositional Formulas

## Definition

Consider a set of boolean variables  $x_1, x_2, \dots, x_n$ .

- 1 A **literal** is either a boolean variable  $x_i$  or its negation  $\neg x_i$ .
- 2 A **clause** is a disjunction of literals.  
For example,  $x_1 \vee x_2 \vee \neg x_4$  is a clause.
- 3 A **formula in conjunctive normal form (CNF)** is propositional formula which is a conjunction of clauses
  - 1  $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$  is a **CNF** formula.
- 4 A formula  $\varphi$  is a **3CNF**:  
A **CNF** formula such that every clause has **exactly** 3 literals.
  - 1  $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$  is a **3CNF** formula, but  $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$  is not.

# Satisfiability

## Problem: SAT

**Instance:** A CNF formula  $\varphi$ .

**Question:** Is there a truth assignment to the variables of  $\varphi$  such that  $\varphi$  evaluates to true?

## Problem: 3SAT

**Instance:** A 3CNF formula  $\varphi$ .

**Question:** Is there a truth assignment to the variables of  $\varphi$  such that  $\varphi$  evaluates to true?

# Satisfiability

## SAT

Given a **CNF** formula  $\varphi$ , is there a truth assignment to variables such that  $\varphi$  evaluates to true?

## Example

- 1  $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$  is satisfiable; take  $x_1, x_2, \dots, x_5$  to be all true
- 2  $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$  is not satisfiable.

## 3SAT

Given a **3CNF** formula  $\varphi$ , is there a truth assignment to variables such that  $\varphi$  evaluates to true?

(More on **2SAT** in a bit...)

# Importance of **SAT** and **3SAT**

- ① **SAT** and **3SAT** are basic constraint satisfaction problems.
- ② Many different problems can be reduced to them because of the simple yet powerful expressiveness of logical constraints.
- ③ Arise naturally in many applications involving hardware and software verification and correctness.
- ④ As we will see, it is a fundamental problem in theory of **NP-Completeness**.

# 3SAT $\leq_P$ SAT

① 3SAT  $\leq_P$  SAT.

② Because...

A 3SAT instance is also an instance of SAT.

# $SAT \leq_p 3SAT$

Claim

$SAT \leq_p 3SAT$ .



# SAT $\leq_P$ 3SAT

## Claim

SAT  $\leq_P$  3SAT.

Given  $\varphi$  a SAT formula we create a 3SAT formula  $\varphi'$  such that

- 1  $\varphi$  is satisfiable iff  $\varphi'$  is satisfiable.
- 2  $\varphi'$  can be constructed from  $\varphi$  in time polynomial in  $|\varphi|$ .

# SAT $\leq_P$ 3SAT

## Claim

SAT  $\leq_P$  3SAT.

Given  $\varphi$  a SAT formula we create a 3SAT formula  $\varphi'$  such that

- 1  $\varphi$  is satisfiable iff  $\varphi'$  is satisfiable.
- 2  $\varphi'$  can be constructed from  $\varphi$  in time polynomial in  $|\varphi|$ .

**Idea:** if a clause of  $\varphi$  is not of length 3, replace it with several clauses of length exactly 3.

# SAT $\leq_p$ 3SAT

## How SAT is different from 3SAT?

In SAT clauses might have arbitrary length: **1, 2, 3, ...** variables:

$$(x \vee y \vee z \vee w \vee u) \wedge (\neg x \vee \neg y \vee \neg z \vee w \vee u) \wedge (\neg x)$$

In 3SAT every clause must have **exactly 3** different literals.

To reduce from an instance of SAT to an instance of 3SAT, we must make all clauses to have exactly **3** variables...

## Basic idea

- 1 Pad short clauses so they have **3** literals.
- 2 Break long clauses into shorter clauses.
- 3 Repeat the above till we have a **3CNF**.

**Note:** Need to add new variables.

# What about **2SAT**?

**2SAT** can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

## Why the reduction from **3SAT** to **2SAT** fails?

Consider a clause  $(x \vee y \vee z)$ . We need to reduce it to a collection of **2CNF** clauses. Introduce a fresh variable  $\alpha$ , and rewrite this as

$$\begin{array}{ll} (x \vee y \vee \alpha) \wedge (\neg \alpha \vee z) & \text{(bad! clause with 3 vars)} \\ \text{or } (x \vee \alpha) \wedge (\neg \alpha \vee y \vee z) & \text{(bad! clause with 3 vars).} \end{array}$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

# What about **2SAT**?

A challenging exercise: Given a **2SAT** formula show to compute its satisfying assignment...

Look in books etc.

# Independent Set

**Problem:** Independent Set

**Instance:** A graph  $G$ , integer  $k$ .

**Question:** Is there an independent set in  $G$  of size  $k$ ?

# 3SAT $\leq_P$ Independent Set

## The reduction 3SAT $\leq_P$ Independent Set

**Input:** Given a 3CNF formula  $\varphi$

**Goal:** Construct a graph  $G_\varphi$  and number  $k$  such that  $G_\varphi$  has an independent set of size  $k$  if and only if  $\varphi$  is satisfiable.

# 3SAT $\leq_P$ Independent Set

## The reduction 3SAT $\leq_P$ Independent Set

**Input:** Given a 3CNF formula  $\varphi$

**Goal:** Construct a graph  $G_\varphi$  and number  $k$  such that  $G_\varphi$  has an independent set of size  $k$  if and only if  $\varphi$  is satisfiable.

$G_\varphi$  should be constructable in time polynomial in size of  $\varphi$



# 3SAT $\leq_P$ Independent Set

## The reduction 3SAT $\leq_P$ Independent Set

**Input:** Given a 3CNF formula  $\varphi$

**Goal:** Construct a graph  $G_\varphi$  and number  $k$  such that  $G_\varphi$  has an independent set of size  $k$  if and only if  $\varphi$  is satisfiable.

$G_\varphi$  should be constructable in time polynomial in size of  $\varphi$

**Importance of reduction:** Although 3SAT is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

# Interpreting 3SAT

There are two ways to think about 3SAT

# Interpreting 3SAT

There are two ways to think about **3SAT**

- 1 Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

# Interpreting 3SAT

There are two ways to think about **3SAT**

- 1 Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
- 2 Pick a literal from each clause and find a truth assignment to make all of them true

# Interpreting 3SAT

There are two ways to think about **3SAT**

- 1 Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
- 2 Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in **conflict**, i.e., you pick  $x_i$  and  $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

# The Reduction

- 1  $G_\varphi$  will have one vertex for each literal in a clause

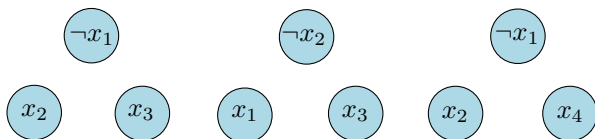


Figure: Graph for

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

# The Reduction

- 1  $G_\varphi$  will have one vertex for each literal in a clause
- 2 Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

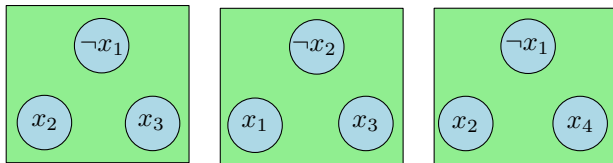


Figure: Graph for

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

# The Reduction

- 1  $G_\varphi$  will have one vertex for each literal in a clause
- 2 Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

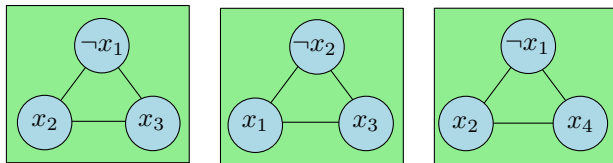


Figure: Graph for

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



# The Reduction

- 1  $G_\varphi$  will have one vertex for each literal in a clause
- 2 Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
- 3 Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

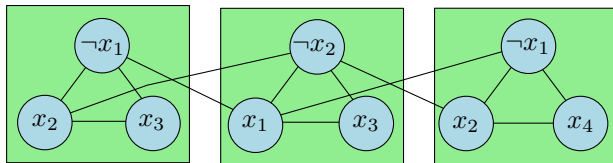


Figure: Graph for

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

# The Reduction

- 1  $G_\varphi$  will have one vertex for each literal in a clause
- 2 Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
- 3 Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 4 Take  $k$  to be the number of clauses

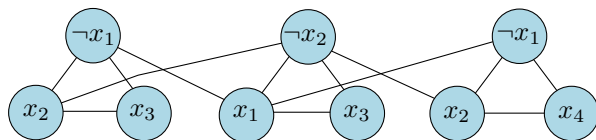


Figure: Graph for

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

## Proposition

$\varphi$  is satisfiable iff  $G_\varphi$  has an independent set of size  $k$  (= number of clauses in  $\varphi$ ).

## Proof.

$\Rightarrow$  Let  $a$  be the truth assignment satisfying  $\varphi$

## Proposition

$\varphi$  is satisfiable iff  $G_\varphi$  has an independent set of size  $k$  (= number of clauses in  $\varphi$ ).

## Proof.

$\Rightarrow$  Let  $\mathbf{a}$  be the truth assignment satisfying  $\varphi$

- 1 Pick one of the vertices, corresponding to true literals under  $\mathbf{a}$ , from each triangle. This is an independent set of the appropriate size □

# Correctness (contd)

## Proposition

$\varphi$  is satisfiable iff  $G_\varphi$  has an independent set of size  $k$  (= number of clauses in  $\varphi$ ).

## Proof.

← Let  $S$  be an independent set of size  $k$

- 1  $S$  must contain exactly one vertex from each clause
- 2  $S$  cannot contain vertices labeled by conflicting clauses
- 3 Thus, it is possible to obtain a truth assignment that makes in the literals in  $S$  true; such an assignment satisfies one literal in every clause □

# Transitivity of Reductions

## Lemma

$X \leq_P Y$  and  $Y \leq_P Z$  implies that  $X \leq_P Z$ .

**Note:**  $X \leq_P Y$  does not imply that  $Y \leq_P X$  and hence it is very important to know the FROM and TO in a reduction.

To prove  $X \leq_P Y$  you need to show a reduction FROM  $X$  TO  $Y$   
In other words show that an algorithm for  $Y$  implies an algorithm for  $X$ .

# Part II

## Definition of NP

## Problems

- 1 Independent Set
- 2 Vertex Cover
- 3 Set Cover
- 4 SAT
- 5 3SAT



## Problems

- 1 Independent Set
- 2 Vertex Cover
- 3 Set Cover
- 4 SAT
- 5 3SAT

## Relationship

**3SAT**  $\leq_P$  **Independent Set**

## Problems

- 1 Independent Set
- 2 Vertex Cover
- 3 Set Cover
- 4 SAT
- 5 3SAT

## Relationship

$$3SAT \leq_P \text{Independent Set} \begin{matrix} \leq_P \\ \geq_P \end{matrix} \text{Vertex Cover}$$

## Problems

- 1 Independent Set
- 2 Vertex Cover
- 3 Set Cover
- 4 SAT
- 5 3SAT

## Relationship

$3SAT \leq_P Independent\ Set \begin{matrix} \leq_P \\ \geq_P \end{matrix} Vertex\ Cover \leq_P Set\ Cover$

## Problems

- 1 Independent Set
- 2 Vertex Cover
- 3 Set Cover
- 4 SAT
- 5 3SAT

## Relationship

$3SAT \leq_P \text{Independent Set} \stackrel{\leq_P}{\geq_P} \text{Vertex Cover} \leq_P \text{Set Cover}$   
 $3SAT \leq_P \text{SAT} \leq_P 3SAT$

# Problems and Algorithms: Formal Approach

## Decision Problems

- 1 **Problem Instance:** Binary string  $s$ , with size  $|s|$
- 2 **Problem:** A set  $X$  of strings on which the answer should be "yes"; we call these YES instances of  $X$ . Strings not in  $X$  are NO instances of  $X$ .

## Definition

- 1  $A$  is an **algorithm for problem  $X$**  if  $A(s) = \text{"yes"}$  iff  $s \in X$ .
- 2  $A$  is said to have a **polynomial running time** if there is a polynomial  $p(\cdot)$  such that for every string  $s$ ,  $A(s)$  terminates in at most  $O(p(|s|))$  steps.

# Polynomial Time

## Definition

**Polynomial time** (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

# Polynomial Time

## Definition

**Polynomial time** (denoted by  $\mathbf{P}$ ) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

## Example

Problems in  $\mathbf{P}$  include

- 1 Is there a shortest path from  $s$  to  $t$  of length  $\leq k$  in  $G$ ?
- 2 Is there a flow of value  $\geq k$  in network  $G$ ?
- 3 Is there an assignment to variables to satisfy given linear constraints?

# Efficiency Hypothesis

*A problem  $X$  has an efficient algorithm iff  $X \in P$ , that is  $X$  has a polynomial time algorithm.*

Justifications:

- 1 Robustness of definition to variations in machines.
- 2 A sound theoretical definition.
- 3 Most known polynomial time algorithms for “natural” problems have small polynomial running times.



# Problems with no known polynomial time algorithms

## Problems

- 1 **Independent Set**
- 2 **Vertex Cover**
- 3 **Set Cover**
- 4 **SAT**
- 5 **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

**Question:** What is common to above problems?

# Efficient Checkability

Above problems share the following feature:

## Checkability

*For any YES instance  $I_X$  of  $X$  there is a proof/certificate/solution that is of length  $\text{poly}(|I_X|)$  such that given a proof one can efficiently check that  $I_X$  is indeed a YES instance.*

# Efficient Checkability

Above problems share the following feature:

## Checkability

*For any YES instance  $I_X$  of  $X$  there is a proof/certificate/solution that is of length  $\text{poly}(|I_X|)$  such that given a proof one can efficiently check that  $I_X$  is indeed a YES instance.*

Examples:

- 1 **SAT** formula  $\varphi$ : proof is a satisfying assignment.
- 2 **Independent Set** in graph  $G$  and  $k$ : a subset  $S$  of vertices.

## Definition

An algorithm  $C(\cdot, \cdot)$  is a **certifier** for problem  $X$  if for every  $s \in X$  there is some string  $t$  such that  $C(s, t) = \text{"yes"}$ , and conversely, if for some  $s$  and  $t$ ,  $C(s, t) = \text{"yes"}$  then  $s \in X$ . The string  $t$  is called a **certificate** or **proof** for  $s$ .

## Definition

An algorithm  $C(\cdot, \cdot)$  is a **certifier** for problem  $X$  if for every  $s \in X$  there is some string  $t$  such that  $C(s, t) = \text{"yes"}$ , and conversely, if for some  $s$  and  $t$ ,  $C(s, t) = \text{"yes"}$  then  $s \in X$ .

The string  $t$  is called a **certificate** or **proof** for  $s$ .

## Definition (Efficient Certifier.)

A certifier  $C$  is an **efficient certifier** for problem  $X$  if there is a polynomial  $p(\cdot)$  such that for every string  $s$ , we have that

- ★  $s \in X$  if and only if
- ★ there is a string  $t$ :
  - 1  $|t| \leq p(|s|)$ ,
  - 2  $C(s, t) = \text{"yes"}$ ,
  - 3 and  $C$  runs in polynomial time.

# Example: Independent Set

- ① **Problem:** Does  $G = (V, E)$  have an independent set of size  $\geq k$ ?
  - ① **Certificate:** Set  $S \subseteq V$ .
  - ② **Certifier:** Check  $|S| \geq k$  and no pair of vertices in  $S$  is connected by an edge.

# Example: Vertex Cover

- ① **Problem:** Does  $G$  have a vertex cover of size  $\leq k$ ?
- ① **Certificate:**  $S \subseteq V$ .
- ② **Certifier:** Check  $|S| \leq k$  and that for every edge at least one endpoint is in  $S$ .

# Example: SAT

- ① **Problem:** Does formula  $\varphi$  have a satisfying truth assignment?
  - ① **Certificate:** Assignment  $a$  of **0/1** values to each variable.
  - ② **Certifier:** Check each clause under  $a$  and say “yes” if all clauses are true.



# Example: Composites

**Problem:** Composite

**Instance:** A number  $s$ .

**Question:** Is the number  $s$  a composite?

① **Problem:** Composite.

- ① **Certificate:** A factor  $t \leq s$  such that  $t \neq 1$  and  $t \neq s$ .
- ② **Certifier:** Check that  $t$  divides  $s$ .

# Not composite?

## Problem: **Not Composite**

**Instance:** A number  $s$ .

**Question:** Is the number  $s$  not a composite?

The problem **Not Composite** is

- (A) Can be solved in linear time.
- (B) in **P**.
- (C) Can be solved in exponential time.
- (D) Does not have a certificate or an efficient certifier.
- (E) The status of this problem is still open.

# Post Correspondence Problem

Given: Dominoes, each with a top-word and a bottom-word.

<i>b</i>	<i>ba</i>	<i>abb</i>	<i>abb</i>	<i>a</i>
<i>bbb</i>	<i>bbb</i>	<i>a</i>	<i>baa</i>	<i>ab</i>

Can one arrange them, using any number of copies of each type, so that the top and bottom strings are equal?

<i>abb</i>	<i>ba</i>	<i>abb</i>	<i>a</i>	<i>abb</i>	<i>b</i>
<i>a</i>	<i>bbb</i>	<i>a</i>	<i>ab</i>	<i>baa</i>	<i>bbb</i>

# Example: A String Problem

## Problem: PCP

**Instance:** Two sets of binary strings  $\alpha_1, \dots, \alpha_n$  and  $\beta_1, \dots, \beta_n$

**Question:** Are there indices  $i_1, i_2, \dots, i_k$  such that  $\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_k}$

### ① Problem: PCP

- ① **Certificate:** A sequence of indices  $i_1, i_2, \dots, i_k$
- ② **Certifier:** Check that  $\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_k}$

# Example: A String Problem

## Problem: PCP

**Instance:** Two sets of binary strings  $\alpha_1, \dots, \alpha_n$  and  $\beta_1, \dots, \beta_n$

**Question:** Are there indices  $i_1, i_2, \dots, i_k$  such that  $\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_k}$

### ① Problem: PCP

- ① **Certificate:** A sequence of indices  $i_1, i_2, \dots, i_k$
- ② **Certifier:** Check that  $\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!  
Implies no finite bound on length of certificate!

# Nondeterministic Polynomial Time

## Definition

**Nondeterministic Polynomial Time** (denoted by **NP**) is the class of all problems that have efficient certifiers.

# Nondeterministic Polynomial Time

## Definition

**Nondeterministic Polynomial Time** (denoted by **NP**) is the class of all problems that have efficient certifiers.

## Example

**Independent Set**, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, and **Composite** are all examples of problems in **NP**.

# Why is it called...

## Nondeterministic Polynomial Time

A certifier is an algorithm  $C(I, c)$  with two inputs:

- 1  $I$ : instance.
- 2  $c$ : proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about  $C$  as an algorithm for the original problem, if:

- 1 Given  $I$ , the algorithm guesses (non-deterministically, and who knows how) a certificate  $c$ .
- 2 The algorithm now verifies the certificate  $c$  for the instance  $I$ .

**NP** can be equivalently described using Turing machines.



# Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

## Example

**SAT** formula  $\varphi$ . No easy way to prove that  $\varphi$  is NOT satisfiable!

More on this and **co-NP** later on.

# P versus NP

Proposition

$P \subseteq NP$ .

# P versus NP

## Proposition

$P \subseteq NP$ .

For a problem in  $P$  no need for a certificate!

## Proof.

Consider problem  $X \in P$  with algorithm  $A$ . Need to demonstrate that  $X$  has an efficient certifier:

- 1 Certifier  $C$  on input  $s, t$ , runs  $A(s)$  and returns the answer.
- 2  $C$  runs in polynomial time.
- 3 If  $s \in X$ , then for every  $t$ ,  $C(s, t) = \text{"yes"}$ .
- 4 If  $s \notin X$ , then for every  $t$ ,  $C(s, t) = \text{"no"}$ . □

# Exponential Time

## Definition

**Exponential Time** (denoted **EXP**) is the collection of all problems that have an algorithm which on input  $s$  runs in exponential time, i.e.,  $O(2^{\text{poly}(|s|)})$ .

# Exponential Time

## Definition

**Exponential Time** (denoted **EXP**) is the collection of all problems that have an algorithm which on input  $s$  runs in exponential time, i.e.,  $O(2^{\text{poly}(|s|)})$ .

Example:  $O(2^n)$ ,  $O(2^{n \log n})$ ,  $O(2^{n^3})$ , ...

# NP versus EXP

## Proposition

$\text{NP} \subseteq \text{EXP}$ .

## Proof.

Let  $X \in \text{NP}$  with certifier  $C$ . Need to design an exponential time algorithm for  $X$ .

- 1 For every  $t$ , with  $|t| \leq p(|s|)$  run  $C(s, t)$ ; answer “yes” if any one of these calls returns “yes”.
- 2 The above algorithm correctly solves  $X$  (exercise).
- 3 Algorithm runs in  $O(q(|s| + |p(s)|)2^{p(|s|)})$ , where  $q$  is the running time of  $C$ . □

# Examples

- ① **SAT**: try all possible truth assignment to variables.
- ② **Independent Set**: try all possible subsets of vertices.
- ③ **Vertex Cover**: try all possible subsets of vertices.

# Is **NP** efficiently solvable?

We know  $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$ .



# Is **NP** efficiently solvable?

We know  $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$ .

## Big Question

Is there are problem in **NP** that **does not** belong to **P**? Is  $\mathbf{P} = \mathbf{NP}$ ?

# If $P = NP$ ...

Or: If pigs could fly then life would be sweet.

- 1 Many important optimization problems can be solved efficiently.

# If $P = NP$ ...

Or: If pigs could fly then life would be sweet.

- 1 Many important optimization problems can be solved efficiently.
- 2 The **RSA** cryptosystem can be broken.

# If $P = NP$ ...

Or: If pigs could fly then life would be sweet.

- 1 Many important optimization problems can be solved efficiently.
- 2 The **RSA** cryptosystem can be broken.
- 3 No security on the web.

# If $P = NP$ . . .

Or: If pigs could fly then life would be sweet.

- 1 Many important optimization problems can be solved efficiently.
- 2 The **RSA** cryptosystem can be broken.
- 3 No security on the web.
- 4 No e-commerce . . .

# If $P = NP$ . . .

Or: If pigs could fly then life would be sweet.

- 1 Many important optimization problems can be solved efficiently.
- 2 The **RSA** cryptosystem can be broken.
- 3 No security on the web.
- 4 No e-commerce . . .
- 5 Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If  $P = NP$  this implies that...

- (A) **Vertex Cover** can be solved in polynomial time.
- (B)  $P = EXP$ .
- (C)  $EXP \subseteq P$ .
- (D) All of the above.

# P versus NP

## Status

Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

**Consensus:** Most people feel/believe  $P \neq NP$ .

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!



# Linear Programming in NP

Is LP in **NP**? Recall LP in (one) standard form is  $\max cx, Ax \leq b$ .

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq K$ ? Input has  $n + mn + m + 1$  numbers.

- What is the certificate?
- What is the certifier?

# Linear Programming in NP

Is LP in **NP**? Recall LP in (one) standard form is  $\max cx, Ax \leq b$ .

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq K$ ? Input has  $n + mn + m + 1$  numbers.

- What is the certificate?
- What is the certifier?

**Certificate:** A solution  $y \in \mathbb{R}^n$  consisting of  $n$  numbers?

**Certifier:** Check that  $Ay \leq b$  and that  $cy \geq K$

# Linear Programming in NP

Is LP in **NP**? Recall LP in (one) standard form is  $\max cx, Ax \leq b$ .

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq K$ ? Input has  $n + mn + m + 1$  numbers.

- What is the certificate?
- What is the certifier?

**Certificate:** A solution  $y \in \mathbb{R}^n$  consisting of  $n$  numbers?

**Certifier:** Check that  $Ay \leq b$  and that  $cy \geq K$

**Caveat:** What is the representation size of  $y$ ? Are we even guaranteed rational numbers? How many bits do we need to represent  $y$  and is it polynomial in the input size?

# Linear Programming in NP

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq B$ ?

Assume for simplicity that  $Ax \leq b$  defines a bounded polytope

- there is an optimum solution  $x^*$  which is a vertex
- $x^*$  is defined as the unique solution to  $A'x = b'$  where  $A'$  is a full-rank sub-matrix of  $A$  and  $b'$  is the corresponding sub-vector of  $b$
- thus  $x^* = (A')^{-1}b' = \frac{1}{\det(A')}(\text{adjoint}(A'))^T b'$

# Linear Programming in NP

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq B$ ?

Assume for simplicity that  $Ax \leq b$  defines a bounded polytope

- there is an optimum solution  $x^*$  which is a vertex
- $x^*$  is defined as the unique solution to  $A'x = b'$  where  $A'$  is a full-rank sub-matrix of  $A$  and  $b'$  is the corresponding sub-vector of  $b$
- thus  $x^* = (A')^{-1}b' = \frac{1}{\det(A')}(\text{adjoint}(A'))^T b'$

**Main question:** How many bits does  $\det(A)$  have as a function of numbers in  $A$ ?

# Linear Programming in NP

**Main question:** How many bits does  $\det(\mathbf{A})$  have as a function of numbers in  $\mathbf{A}$ ?

One definition of determinant of a  $n \times n$  matrix  $\mathbf{A}$  is:

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^n A_{i\sigma(i)}$$

Here  $S_n$  is the set of all  $n!$  permutations of  $\{1, 2, \dots, n\}$  and  $\text{sign}(\sigma) \in \{-1, 1\}$  is the signature of  $\sigma$  depending on whether  $\sigma$  can be obtained by odd or even number of transpositions.

Therefore  $|\det(\mathbf{A})| \leq n! \times (\max_{ij} |A_{ij}|)^n$  and hence

$$\log |\det(\mathbf{A})| \leq n \log n + n \log(\max_{ij} |A_{ij}|)$$

# Integer Linear Programming in **NP**

Is ILP in **NP**? Recall ILP in (one) standard form is  $\max cx, Ax \leq b, x \in \mathbb{Z}^n$ .

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq K$ ? Input has  $n + mn + m + 1$  numbers.

**Certificate:** A solution  $y \in \mathbb{R}^n$  consisting of  $n$  numbers?

**Certifier:** Check that  $Ay \leq b$  and that  $cy \geq K$

# Integer Linear Programming in NP

Is ILP in **NP**? Recall ILP in (one) standard form is  
 $\max cx, Ax \leq b, x \in \mathbb{Z}^n$ .

Given  $c, A, b$  where  $c \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$  and integer  $K$ , is optimum value  $\geq K$ ? Input has  $n + mn + m + 1$  numbers.

**Certificate:** A solution  $y \in \mathbb{R}^n$  consisting of  $n$  numbers?

**Certifier:** Check that  $Ay \leq b$  and that  $cy \geq K$

**Caveat:** What is the representation size of  $y$ ? How many bits do we need to represent  $y$  and is it polynomial in the input size? Note that unlike LP  $y$  is not necessarily a vertex of the polytope defined by  $Ax \leq b$ . Can be in the interior.

Need some advanced tools to prove that there always exists a  $y$  with representation size polynomial in input size.



## Part III

# NP-Completeness and Cook-Levin Theorem

# “Hardest” Problems

## Question

What is the hardest problem in **NP**? How do we define it?

## Towards a definition

- 1 Hardest problem must be in **NP**.
- 2 Hardest problem must be at least as “difficult” as every other problem in **NP**.

# NP-Complete Problems

## Definition

A problem  $X$  is said to be **NP-Complete** if

- 1  $X \in \text{NP}$ , and
- 2 (**Hardness**) For any  $Y \in \text{NP}$ ,  $Y \leq_P X$ .

# Solving **NP-Complete** Problems

## Proposition

Suppose  $X$  is **NP-Complete**. Then  $X$  can be solved in polynomial time if and only if  $P = NP$ .

## Proof.

$\Rightarrow$  Suppose  $X$  can be solved in polynomial time

- ① Let  $Y \in NP$ . We know  $Y \leq_P X$ .
- ② We showed that if  $Y \leq_P X$  and  $X$  can be solved in polynomial time, then  $Y$  can be solved in polynomial time.
- ③ Thus, every problem  $Y \in NP$  is such that  $Y \in P$ ;  $NP \subseteq P$ .
- ④ Since  $P \subseteq NP$ , we have  $P = NP$ .

$\Leftarrow$  Since  $P = NP$ , and  $X \in NP$ , we have a polynomial time algorithm for  $X$ . □

# NP-Hard Problems

## Definition

A problem  $X$  is said to be **NP-Hard** if

- 1 (Hardness) For any  $Y \in \text{NP}$ , we have that  $Y \leq_P X$ .

An **NP-Hard** problem need not be in **NP**!

**Example:** Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

# Consequences of proving **NP-Completeness**

If  $X$  is **NP-Complete**

- 1 Since we believe  $P \neq NP$ ,
- 2 and solving  $X$  implies  $P = NP$ .

$X$  is **unlikely** to be efficiently solvable.

# Consequences of proving **NP-Completeness**

If  $X$  is **NP-Complete**

- 1 Since we believe  $P \neq NP$ ,
- 2 and solving  $X$  implies  $P = NP$ .

$X$  is **unlikely** to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for  $X$ .

# Consequences of proving **NP-Completeness**

If  $X$  is **NP-Complete**

- 1 Since we believe  $P \neq NP$ ,
- 2 and solving  $X$  implies  $P = NP$ .

$X$  is **unlikely** to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for  $X$ .



# Consequences of proving **NP-Completeness**

If  $X$  is **NP-Complete**

- 1 Since we believe  $P \neq NP$ ,
- 2 and solving  $X$  implies  $P = NP$ .

$X$  is **unlikely** to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for  $X$ .

(This is proof by mob opinion — take with a grain of salt.)

# NP-Complete Problems

## Question

Are there any problems that are **NP-Complete**?

## Answer

Yes! Many, many problems are **NP-Complete**.

# Cook-Levin Theorem

## Theorem

**SAT** is **NP-Complete**.

# Cook-Levin Theorem

## Theorem

**SAT** is **NP-Complete**.

Using reductions one can prove that many other problems are **NP-Complete**

# Proving that a problem $X$ is **NP-Complete**

To prove  $X$  is **NP-Complete**, show

- 1 Show  $X$  is in **NP**.
  - 1 certificate/proof of polynomial size in input
  - 2 polynomial time certifier  $C(s, t)$
- 2 Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to  $X$

# Proving that a problem $X$ is **NP-Complete**

To prove  $X$  is **NP-Complete**, show

- 1 Show  $X$  is in **NP**.
  - 1 certificate/proof of polynomial size in input
  - 2 polynomial time certifier  $C(s, t)$
- 2 Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to  $X$

$SAT \leq_P X$  implies that every **NP** problem  $Y \leq_P X$ . Why?

# Proving that a problem $X$ is **NP-Complete**

To prove  $X$  is **NP-Complete**, show

- 1 Show  $X$  is in **NP**.
  - 1 certificate/proof of polynomial size in input
  - 2 polynomial time certifier  $C(s, t)$
- 2 Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to  $X$

$SAT \leq_P X$  implies that every **NP** problem  $Y \leq_P X$ . Why?

Transitivity of reductions:

$Y \leq_P SAT$  and  $SAT \leq_P X$  and hence  $Y \leq_P X$ .

# Integer Linear Programming is NP Complete

ILP in (one) standard form is  $\max cx, Ax \leq b, x \in \mathbb{Z}^n$ .

**Non-trivial statement:** ILP is in **NP**.

Special case of ILP: Boolean ILP where we require  $x \in \{0, 1\}^n$ .



# Integer Linear Programming is NP Complete

ILP in (one) standard form is  $\max cx, Ax \leq b, x \in \mathbb{Z}^n$ .

**Non-trivial statement:** ILP is in **NP**.

Special case of ILP: Boolean ILP where we require  $x \in \{0, 1\}^n$ .

Can easily reduce **3SAT** to Boolean ILP. Also many other standard problems such as **Independent Set** etc.

# NP-Completeness via Reductions

- 1 **SAT** is **NP-Complete**.
- 2 **SAT**  $\leq_P$  **3-SAT** and hence 3-SAT is **NP-Complete**.
- 3 **3-SAT**  $\leq_P$  **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
- 4 **Clique** is **NP-Complete**
- 5 **Vertex Cover** is **NP-Complete**
- 6 **Set Cover** is **NP-Complete**
- 7 **Hamilton Cycle** is **NP-Complete**
- 8 **3-Color** is **NP-Complete**
- 9 **Integer Linear Programming** is **NP-Complete**

# NP-Completeness via Reductions

- 1 **SAT** is **NP-Complete**.
- 2 **SAT**  $\leq_P$  **3-SAT** and hence 3-SAT is **NP-Complete**.
- 3 **3-SAT**  $\leq_P$  **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
- 4 **Clique** is **NP-Complete**
- 5 **Vertex Cover** is **NP-Complete**
- 6 **Set Cover** is **NP-Complete**
- 7 **Hamilton Cycle** is **NP-Complete**
- 8 **3-Color** is **NP-Complete**
- 9 **Integer Linear Programming** is **NP-Complete**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!