# Languages and Algorithms for Artificial Intelligence (Third Module)

## Introduction

Ugo Dal Lago



University of Bologna, Academic Year 2019/2020

Section 1

# The Logistics

# Organisation

- **Webpage**
  - At IOL: `http://iol.unibo.it/`
- **Email**
  - `ugo.dallago@unibo.it`
- **Office Hours**
  - *Where*: see `http://www.cs.unibo.it/~dallago`
  - *When*: there is no fixed office hours, just write me an email and we will fix an appointment.
- **Teaching Assistant**
  - Francesco Gavazzo, `francesco.gavazzo2@unibo.it`
  - Please contact both of us if you need help, this way your request will be taken into account ASAP.

# Structure and Schedule

- The *schedule* of this module's lectures will vary along the coming weeks.
  - Please keep an eye to to the online schedule.
- **Content**: the module is divided into two parts:
  1. The very basics of the theory of *computational complexity*, with implications for AI.
  2. Some rudiments of *computational learning theory*.
- **Requirements**:
  - We assume all students know what a *program* is and what an *algorithm* is.
  - We also assume familiarity with the basics of *probability theory*, *discrete mathematics* (natural numbers, induction, graphs, and the like) and *mathematical logic* (propositional logic and its semantics).
- All lectures will be given by Ugo Dal Lago.
- Some of the exercise sessions will be given by Francesco Gavazzo

# Textbooks and Exams

- **Textbooks**

  The module will be as self contained as possible, but there are plenty of nice books from which you can learn more than what we will do, if you are interested.

  - Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press. 2007.
  - Christos Papadimitriou. *Computational Complexity*. Addison-Wesley. 1994.
  - Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: from Theory to Algorithms* Cambridge University Press. 2014.
  - Michael Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory* The MIT Press. 1994.

- **Exams**

  - You need to pass a *written exam*.
  - As you know, you will get just *one* mark for the whole of this course, and the three modules have the same weight.

# Section 2

## What? Why?

# Is There any Limit on what AI can do?

- **AI and ML are Everywhere**.
  - The great impact to our society of technologies derived from Artificial Intellingence and Machine Learning is in front of us.
  - Learning the way tools from AI and ML can be employed in a variety of application scenarios (vision, social-media, recommendation systems, etc.) is of course important for you and your career.
  - A number of courses in the Master Degree you are enrolled in are specifically about the above.

# Is There any Limit on what AI can do?

- **AI and ML are Everywhere**.
  - The great impact to our society of technologies derived from Artificial Intellingence and Machine Learning is in front of us.
  - Learning the way tools from AI and ML can be employed in a variety of application scenarios (vision, social-media, recommendation systems, etc.) is of course important for you and your career.
  - A number of courses in the Master Degree you are enrolled in are specifically about the above.
- **This is *not* What this Module is About**!
  - The question we will try to ask you is rather the following: *is there any intrinsic limit to what we can do with AI and ML, or to the accurancy or efficiency of algorithms solving a give problem?*

# A Paradigm Shift

**"Algorithm $X$ solves Problem $Y$ Efficiently and with Great Accuracy"**

# A Paradigm Shift

**"Problem $Y$ Cannot be Solved by any Algorithm $X$".**

$$\Uparrow$$

**"Algorithm $X$ solves Problem $Y$ Efficiently and with Great Accuracy"**

# A Paradigm Shift

**"Problem $Y$ Cannot be Solved by any Algorithm $X$".**

$\Uparrow$

**"Algorithm $X$ solves Problem $Y$ Efficiently and with Great Accuracy"**

$\Downarrow$

**"Any Algorithm Solving $Y$ is bound to be Inefficient, or not Accurate".**

# The Value of Negative Results

## A Quote from Thomas A. Edison

*"Negative results are just what I want. They're just as valuable to me as positive results. I can never find the thing that does the job best until I find the ones that don't."*

# The Value of Negative Results

## A Quote from Thomas A. Edison

*"Negative results are just what I want. They're just as valuable to me as positive results. I can never find the thing that does the job best until I find the ones that don't."*

- If we know that the task we want to accomplish is intrinsically *hard*, and we know *in what* its hardness resides, we can:
  1. Direct our work towards methodologies *specifically designed* for hard tasks.
  2. *Avoid* spending our time trying to design solutions that cannot exist.
  3. Inform the stakeholders *in advance* that the solution we are going to design will likely be inefficient.
  4. Decide that the task is *too difficult*, and that it cannot be accomplished the way we want.
  5. Decide that it is better to switch to a *relaxed* version of the task, which instead admits reasonable solutions.

# Example Problems - I

## Natural Number Multiplication

Suppose you want to write a `Python` program which multiplies two positive integer numbers, which can both be expressed as $n$-digits numbers, but which are represented as lists rather than scalars. Suppose that the operations at your disposal are just the basic arithmetic operations *on digits*, so that you cannot just multiply the two numbers. How should you write your program?

# Example Problems - II

## Maximizing the Number of Invitees

Suppose you are going to marry your boyfriend (or girlfriend) soon, and you want to decide the list of invitees. Since your soon-to-be father-in-law will pay all bill, you have all the interest in keeping the list of invitees as large as possible. But actually, it would be impossible to invite all the people in the set $L = \{a_1, \ldots, a_m\}$ of all candidate invitees, since some of them are kind of incompatible, and cannot be invited together, i.e. there is another set $I = \{(b_1, c_1), \ldots, (b_n, c_n)\} \subseteq L \times L$ whose elements are those candidate invitees pairs which are incompatible. Would it be possible to maximize the length of the list of invitees?

# Example Problems - III

## Learning Programs with Finite Inputs and Outputs

Suppose your ML teacher asks you to write a learning algorithm capable of reconstructing a program by querying it as a black box, without looking at the code. You can assume that the program takes in input a list of booleans of a fixed (rather big) length $n$, and produces in output one boolean value. How would you proceed? On which input would you query the program? Would it be possible to query the program on significantly less inputs than the $2^n$ possible ones?

# The Module's Structure

1. **Introduction to Computational Complexity**
   - Historical, Conceptual, and Mathematical Preliminaries
   - The Computational Model, and Uncomputability
   - The Class P
   - The Class NP and NP-Completeness

# The Module's Structure

1. **Introduction to Computational Complexity**
   - Historical, Conceptual, and Mathematical Preliminaries
   - The Computational Model, and Uncomputability
   - The Class P
   - The Class NP and NP-Completeness
2. **Some Rudiments of Computational Learning Theory**
   - From Computation to Learning
   - Positive and Negative Results about the Learning Process
   - The VC Dimension

# Thank You!

Questions?

# Languages and Algorithms for Artificial Intelligence (Third Module)

## Historical, Conceptual, and Mathematical Preliminaries

Ugo Dal Lago



University of Bologna, Academic Year 2019/2020

Section 1

# A Bit of History

# Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.

# Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.

# Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.
- The modern **theory of computation** (ToC) is one of the many gifts humanity has received from science during the first half of the twentieth century.
  - A *precise* definition of the computable has been given, together with many results about it.
  - Actually, many alternative definitions of the computable have been given, but have been proved to be essentially *equivalent*.

# Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.
- The modern **theory of computation** (ToC) is one of the many gifts humanity has received from science during the first half of the twentieth century.
  - A *precise* definition of the computable has been given, together with many results about it.
  - Actually, many alternative definitions of the computable have been given, but have been proved to be essentially *equivalent*.
- In the second half of the twentieth century, ToC has evolved into a **fully fledged** scientific field.

# Computation, Science and Technology

- The outfalls of modern theory of computation, since the 1940s, have been twofold:
  1. **From ToC to the other Sciences**
     - Examples: Genomics, Physics, ...
     - Results from the other sciences have had a very strong impact to the theory of computation, as well.
  2. **From ToC to ICT**
     - The theoretical notion of computation was already there when the first modern, electronic, computers appeared.
     - Concepts from the theory of computation (e.g. universality) informed the design of computer from the very beginning.
     - Computation theory has since been catching up with the way ICT is done, in practice.

# Computation, Science and Technology

- The outfalls of modern theory of computation, since the 1940s, have been twofold:
  1. **From ToC to the other Sciences**
     - Examples: Genomics, Physics, ...
     - Results from the other sciences have had a very strong impact to the theory of computation, as well.
  2. **From ToC to ICT**
     - The theoretical notion of computation was already there when the first modern, electronic, computers appeared.
     - Concepts from the theory of computation (e.g. universality) informed the design of computer from the very beginning.
     - Computation theory has since been catching up with the way ICT is done, in practice.
- One can easily observe a trend, in the last thirty years:
  - **Concepts** from ToC have more and more influence on the other sciences.
  - **Problems** from ToC are considered worth being solved by, e.g., researchers in mathematics and physics.

# Computability and Complexity

- Until the Late 60s, ToC was mainly concerned with understanding **computability**.
  - Main Question: is a certain task computable?
  - If the answer is negative, the task is said to be *uncomputable* (and there can be many ways in which this can happen).
  - This is the so-called **computability theory**.

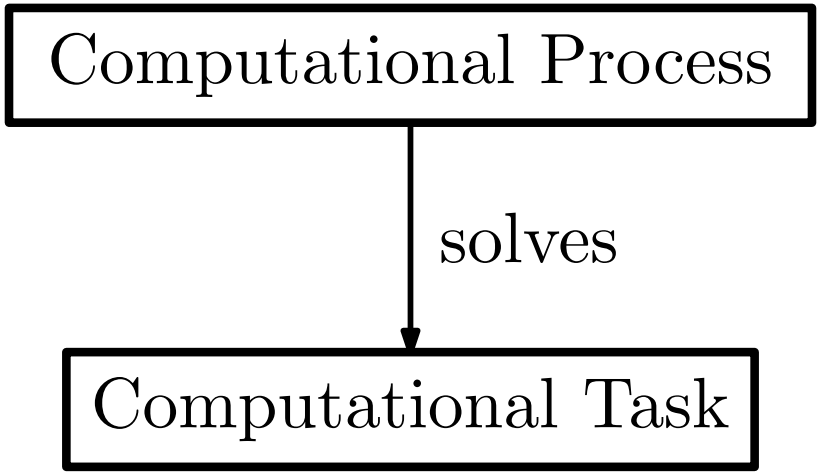
# Computability and Complexity

- Until the Late 60s, ToC was mainly concerned with understanding **computability**.
  - Main Question: is a certain task computable?
  - If the answer is negative, the task is said to be *uncomputable* (and there can be many ways in which this can happen).
  - This is the so-called **computability theory**.
- Since the pioneering works by Hartmanis, Stearns, and Cobham (all from the late 60s), a new branch of ToC has emerged, which deals with **efficiency**.
  - Main Question: is a certain task solvable *in a reasonable amount* of time, or space (i.e. working memory)?
  - If the answer is negative, the task is maybe computable, but requires so much time or space, that it pecomes *practically* uncomputable.
  - This new branch of ToC has been named **computational complexity theory**.
- This module will mainly deal with computational complexity theory.

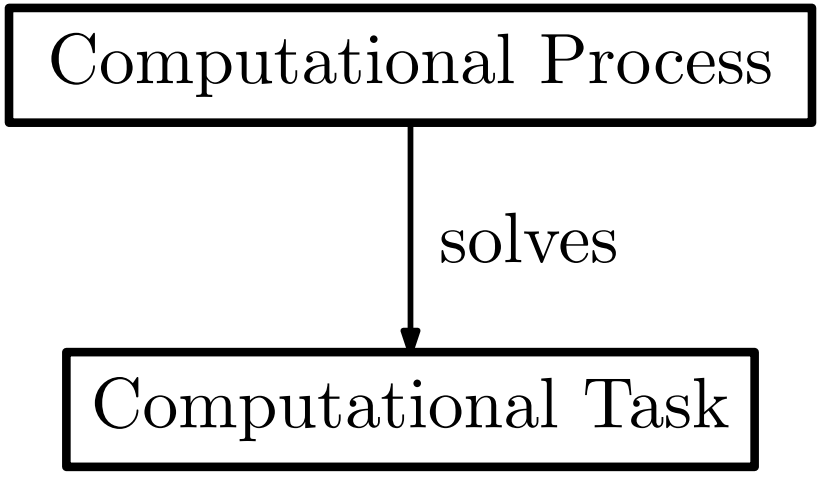
# Section 2

# Modelling Computation

# The Standard Paradigm



# The Standard Paradigm




- Processes and tasks **are different**.
- In some cases, there could be **many** distinct processes solving the same tasks.
- In some other cases, the task is so hard that **no** process can solve it.

# The Multiplication Problem

- **Task**: multiplying two natural numbers $a$ and $b$ both expressible with $n$ digits, by performing operations on digits.

# The Multiplication Problem

- **Task**: multiplying two natural numbers $a$ and $b$ both expressible with $n$ digits, by performing operations on digits.
- One **process** solving the task above consists in reducing multiplication to addition:

$$a \cdot b = \underbrace{a + a + \ldots + a}_{b \text{ times}}$$

Since addition of two $n$-digit numbers is a relatively easy process can easily be carried out in a linear amount of steps, the total number of steps is proportional to $b \cdot n$. Call this process `RepeatedAddition`.

# The Multiplication Problem

- **Task**: multiplying two natural numbers $a$ and $b$ both expressible with $n$ digits, by performing operations on digits.
- One **process** solving the task above consists in reducing multiplication to addition:

$$a \cdot b = \underbrace{a + a + \ldots + a}_{b \text{ times}}$$

  Since addition of two $n$-digit numbers is a relatively easy process can easily be carried out in a linear amount of steps, the total number of steps is proportional to $b \cdot n$. Call this process `RepeatedAddition`.
- Another **process** solving the task above consists in rather multipling $a$ and $b$ with the elementary school direct algorithm, which takes an amount of steps which is proportional to $n \cdot n$. Call this process `GridMethod`

# The Multiplication Problem

- There is an exponential difference between the performances of `RepeatedAddition` and `GridMethod`.
  - Indeed, $b$ can be exponential in $n$, i.e. at most $10^n - 1$.
  - When, e.g., $n$ is 100, there is a huge different between $100 \cdot 100$ and $(10^{100} - 1) \cdot 100$.
  - If each basic instruction takes a millisecond, `GridMethod` would take one second, while `RepeatedAddition` would take more than $10^{80}$ years.
- Distinct processes can solve the same task in very different ways, and not all of them are acceptable.
- `GridMethod` witnesses the fact that the task is in a class of tasks called **P**

# The Wedding Problem

- **Task**: given a set $L$ of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of $L$ composed of compatible invitees.

# The Wedding Problem

- **Task**: given a set $L$ of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of $L$ composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of $L$, and check for the presence of incompatibilities in each of them, at the same tracking their size.

# The Wedding Problem

- **Task**: given a set $L$ of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of $L$ composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of $L$, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If $L$ has $n$ elements, how many subset does $L$ have?
  - This turns out to be $2^n$, which is impractical.

# The Wedding Problem

- **Task**: given a set $L$ of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of $L$ composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of $L$, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If $L$ has $n$ elements, how many subset does $L$ have?
  - This turns out to be $2^n$, which is impractical.
- The problem is thus known to be in the class **NP**.

# The Wedding Problem

- **Task**: given a set $L$ of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of $L$ composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of $L$, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If $L$ has $n$ elements, how many subset does $L$ have?
  - This turns out to be $2^n$, which is impractical.
- The problem is thus known to be in the class **NP**.
- The real question is **can we do better**?
  - If we manage to do significantly better, we would have solved the question of whether **NP** is equal to **P**.

# Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.

# Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.

# Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.
- We are **very rarely** able to prove the nonexistence of efficient algorithm.
  - Finding mathematical techniques which allow us to do so is *the crux* of computational complexity theory.

# Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.
- We are **very rarely** able to prove the nonexistence of efficient algorithm.
  - Finding mathematical techniques which allow us to do so is *the crux* of computational complexity theory.
- Rather than proving the non-existence of certain algorithms, complexity theory **interrelates** different tasks.

# Some Research Questions in Complexity Theory

- The **P** vs. **NP** question.
- Can the use of randomness help in speeding up computation?
- Can hard tasks become easier if we allow algorithms to err on a relatively small subset of the inputs?
- Can we exploit the hardness of certain tasks?
- Can we make use of some counterintuitive properties of quantum mechanics to build faster computing devices?

# Section 3

# Some Mathematical Preliminaries

# Sets and Numbers

- The cardinality of any set $X$ is indicated as $|X|$.
  - It can be finite or infinite.
- We will mainly work with discrete numbers.
  - $\mathbb{N}$ is the set of natural numbers, while $\mathbb{Z}$ is the set of integers.
- We say that a condition $P(n)$ depending on $n \in \mathbb{N}$ holds for **sufficiently large** $n$ if there is $N \in \mathbb{N}$ such that $P(n)$ holds for every $n > N$.
- Some common (and useful notation):
  - For a given real number $x$, $\lceil x \rceil$ is the the smallest element of $\mathbb{Z}$ such that $\lceil x \rceil \geq x$. Whenever a real number $x$ is use in place of anatural number, we implicitly read it as $\lceil x \rceil$
  - For a natural number $n$, $[n]$ is the set $\{1, \ldots, n\}$.
  - Contrarily to the common mathematical notation, $\log x$ is **base** 2 logarithm.

# Strings

- If $S$ is a finite set, then a **string** over the alphabet $S$ is a finite, ordered, possibly empty, tuple of elements from $S$.
  - Most often, the alphabet $S$ will be the set $\{0, 1\}$.
- The set of all strings over $S$ of length exactly $n \in \mathbb{N}$ is indicated as $S^n$ (where $S^0$ is the set containing only the empty string $\varepsilon$).
- The set of *all strings* over $S$ is $\bigcup_{n=0}^{\infty} S^n$ and is indicated as $S^*$.
- The concatenation of two strings $x$ and $y$ over $S$ is indicated as $xy$. The string over $S$ obtained by concatenating $x$ with itself $k \in \mathbb{N}$ times is indicated as $x^k$.
  - Strings form a monoid!
- The **length** of a string $x$ is indicated as $|x|$.
- Examples

$$\{0, 1\}^2 = \{\varepsilon, 0, 1, 00, 01, 10, 11\} \qquad |000101| = 6$$

# Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?

# Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.

# Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.
- How about the emphasis on strings?
  - If the input and/or the output are not strings, but they are taken from a discrete set, they can be **represented** as strings, following some encoding, which however should be as simple as possible.

# Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.
- How about the emphasis on strings?
  - If the input and/or the output are not strings, but they are taken from a discrete set, they can be **represented** as strings, following some encoding, which however should be as simple as possible.
- Summing up, we always assume that the task we want to solve **is given** as a function $f : A \to B$ where both the domain $A$ and $B$ are discrete (i.e. countable) sets, sometime leaving the encoding of $A$ and $B$ into $\{0,1\}^*$ implicit.
- The encoding of any element $x$ of $A$ as a string is often indicated as $\llcorner x \lrcorner$ or simply as $x$.

# Representing Objects as Strings: Examples

- Strings in $S^*$, where $S = \{a, b, c\}$.
  - $a$ can be encoded as 00, $b$ becomes 01 and $c$ becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.

# Representing Objects as Strings: Examples

- Strings in $S^*$, where $S = \{a, b, c\}$.
  - $a$ can be encoded as 00, $b$ becomes 01 and $c$ becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.

# Representing Objects as Strings: Examples

- Strings in $S^*$, where $S = \{a, b, c\}$.
  - $a$ can be encoded as 00, $b$ becomes 01 and $c$ becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.
- Pairs of binary strings, i.e. $\{0, 1\}^* \times \{0, 1\}^*$.
  - One can encode the pair $(x, y)$ as the string $x\#y$ over the alphabet $\{0, 1, \#\}$, and the proceed as before.

# Representing Objects as Strings: Examples

- Strings in $S^*$, where $S = \{a, b, c\}$.
  - $a$ can be encoded as 00, $b$ becomes 01 and $c$ becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.
- Pairs of binary strings, i.e. $\{0, 1\}^* \times \{0, 1\}^*$.
  - One can encode the pair $(x, y)$ as the string $x\#y$ over the alphabet $\{0, 1, \#\}$, and the proceed as before.

$\vdots$

# Languages and Decision Problems

- An important class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ are those whose range are strings of length exactly one, called **boolean functions**.
  - They are characteristic functions.

# Languages and Decision Problems

- An important class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ are those whose range are strings of length exactly one, called **boolean functions**.
  - They are characteristic functions.
- We identify such a function $f$ with the subset $\mathcal{L}_f$ of $\{0,1\}^*$ defined as follows:

$$\mathcal{L}_f = \{x \in \{0,1\}^* \mid f(x) = 1\}.$$

- This way, a **decision problem** for a given language $\mathcal{M}$ (i.e. does $x \in \{0,1\}^*$ is in $\mathcal{M}$?) can be seen as the task of computing $f$ such that $\mathcal{M} = \mathcal{L}_f$.

# Asymptotic Notation

- A function $f : \mathbb{N} \to \mathbb{N}$ is $O(g)$ if there is a positive real constants $c$ such that $f(n) \leq c \cdot g(n)$ for sufficiently large $n$.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $O(n^2)$, but also $O(n^3)$, and certainly $O(2^n)$. It is not, however, $O(n)$.
- A function $f : \mathbb{N} \to \mathbb{N}$ is $\Omega(g)$ if there if a positive real constants $c$ such that $f(n) \geq c \cdot g(n)$ for sufficiently large $n$
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $\Omega(n^2)$, but also $\Omega(n)$, but not $\Omega(n^3)$.
- A function $f$ is $\Theta(g)$ if $f$ is both $O(g)$ and $\Omega(g)$.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n + 7$ is $\Theta(n^2)$.

# Thank You!

Questions?

# Languages and Algorithms for Artificial Intelligence (Third Module)

## The Computational Model

Ugo Dal Lago



University of Bologna, Academic Year 2019/2020

# The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!

# The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.

# The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.
- The only way out is to define a **model of computation** in the form of an abstract machine, keeping in mind that:
  - It should be as simple as possible, to *facilitate proofs*.
  - It must be able to simulate with reasonable overhead *all physically realistic* machines.

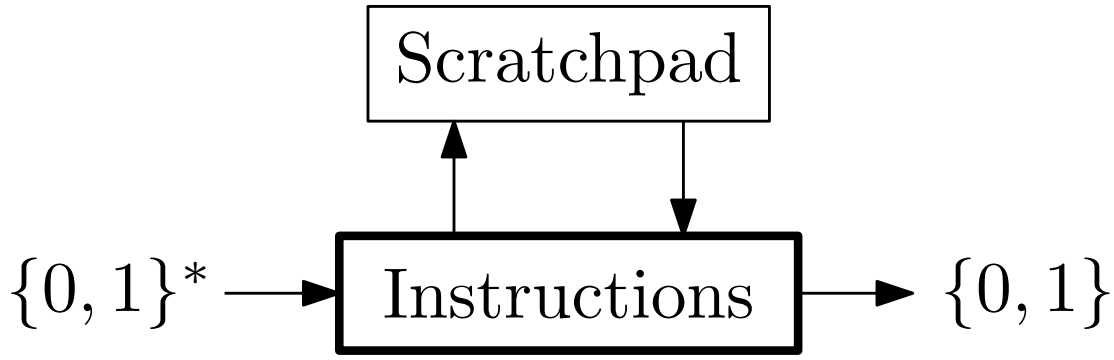
# The Need of a Model

- Giving **positive** results about the feasibility of certain computation task is relatively easy:
  - You implement your algorithm, you run it on a powerful machine, and that's it!
- But how about **negative** results?
  - Which machine should we choose?
  - If we choose one specific, concrete machine without relating to the other ones, then our negative results would be *vacuous*.
- The only way out is to define a **model of computation** in the form of an abstract machine, keeping in mind that:
  - It should be as simple as possible, to *facilitate proofs*.
  - It must be able to simulate with reasonable overhead *all physically realistic* machines.
- There is a universally accepted model of computation, that we will take as our reference model, namely the **Turing Machine**.
  - This part of the module is specifically about it.

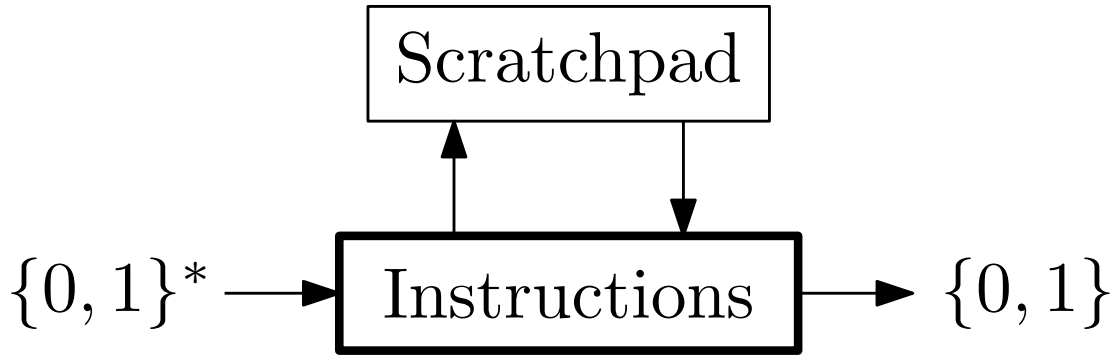
# Part I

# The Model, Informally

# How to Compute $f : \{0,1\}^* \to \{0,1\}$



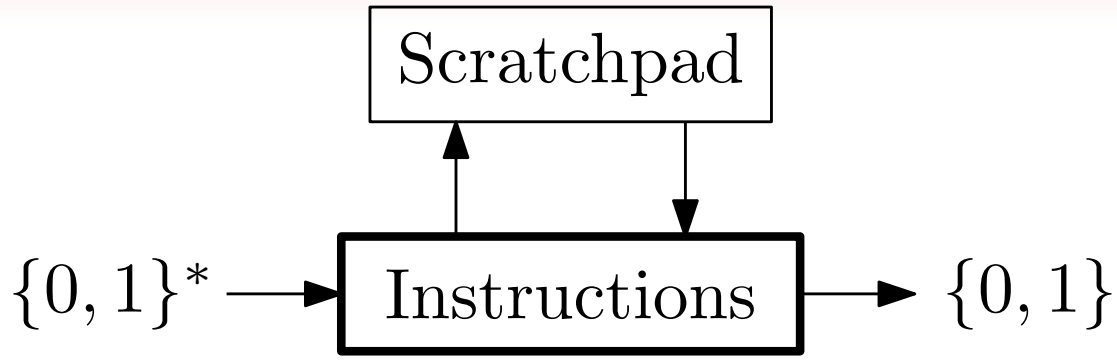
# How to Compute $f : \{0,1\}^* \to \{0,1\}$




- The set of instructions to be followed is fixed (and should work for every input $x$), and finite.
- The same instruction can be used potentially many times.
- Every instruction proceeds by:
  - Reading a bit of the input;
  - Reading a symbol from the scratchpad;

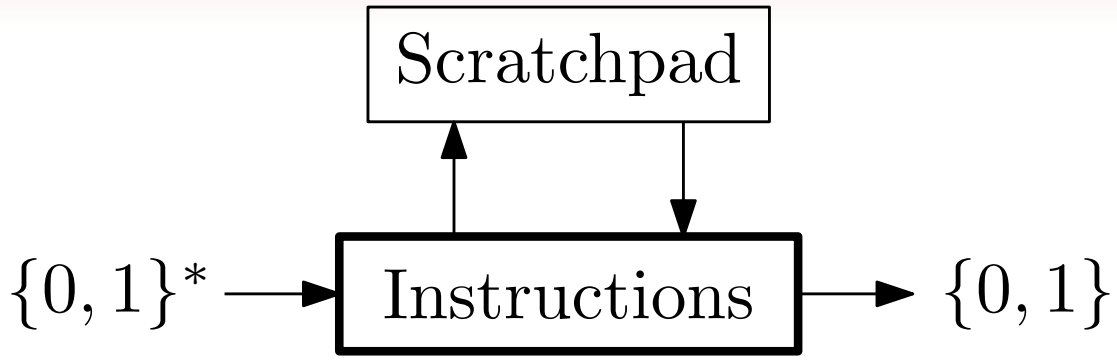
  and based on that decide what to do next, namely:
  - either write symbol to the scratchpad, and proceed to another instruction;
  - or declare the computation finished, by stopping it and outputting either 0 or 1.

# How to Compute $f : \{0,1\}^* \to \{0,1\}$



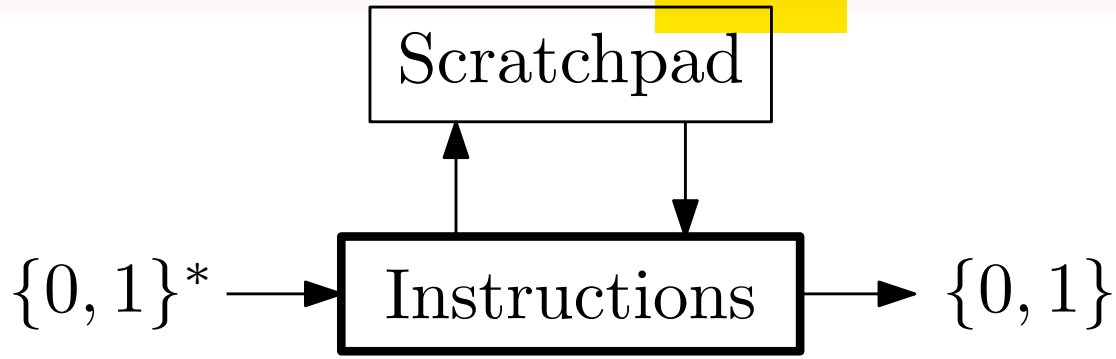

- The **running time** of this machine/process/algorithm on $x$ is simply the number of these basic instructions which are executed on a certain input $x$.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* $n$.

# How to Compute $f : \{0,1\}^* \to \{0,1\}$



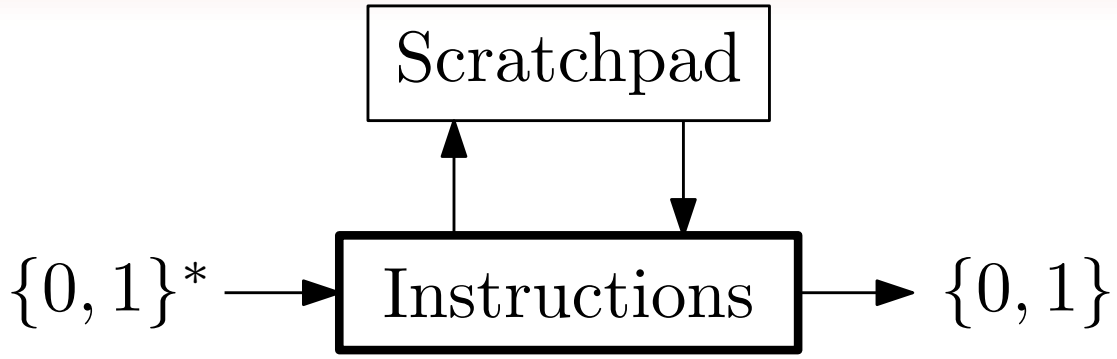

- The **running time** of this machine/process/algorithm on $x$ is simply the number of these basic instructions which are executed on a certain input $x$.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* $n$.
- The model is **robust** to many tweaks in the definition, (e.g. changing the alphabet, allowing multiple scratchpads rather than one): the simplest model can simulate the more complicated with *a polynomial overhead* in time.

# How to Compute $f : \{0,1\}^* \to \{0,1\}$



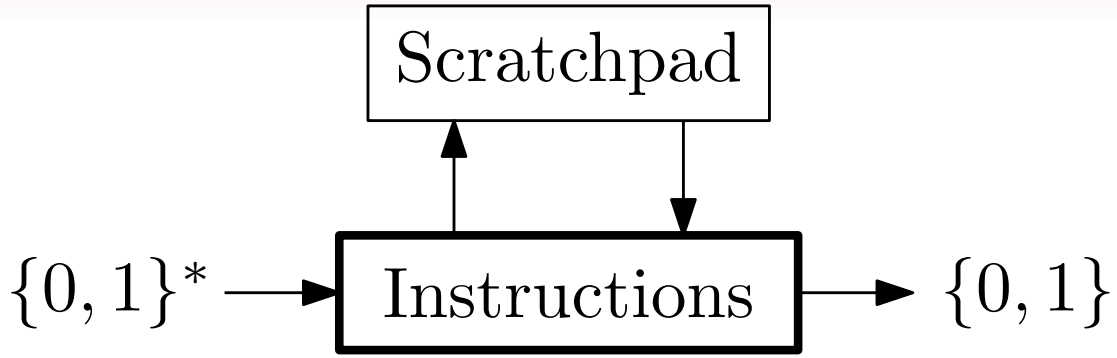

- The **running time** of this machine/process/algorithm on $x$ is simply the number of these basic instructions which are executed on a certain input $x$.
- We say that the machine **runs in time** $T(n)$ if it performs *at most* $T(n)$ instructions on input strings *of length* $n$.
- The model is **robust** to many tweaks in the definition, (e.g. changing the alphabet, allowing multiple scratchpads rather than one): the simplest model can simulate the more complicated with *a polynomial overhead* in time.
- Since there are finitely many instructions, machine descriptions can be **encoded as binary strings** themselves.

# How to Compute $f : \{0,1\}^* \to \{0,1\}$




- Given a string $\alpha$, we indicate as $\mathcal{M}_\alpha$ the Turing Machine $\alpha$ encodes.

# How to Compute $f : \{0,1\}^* \to \{0,1\}$

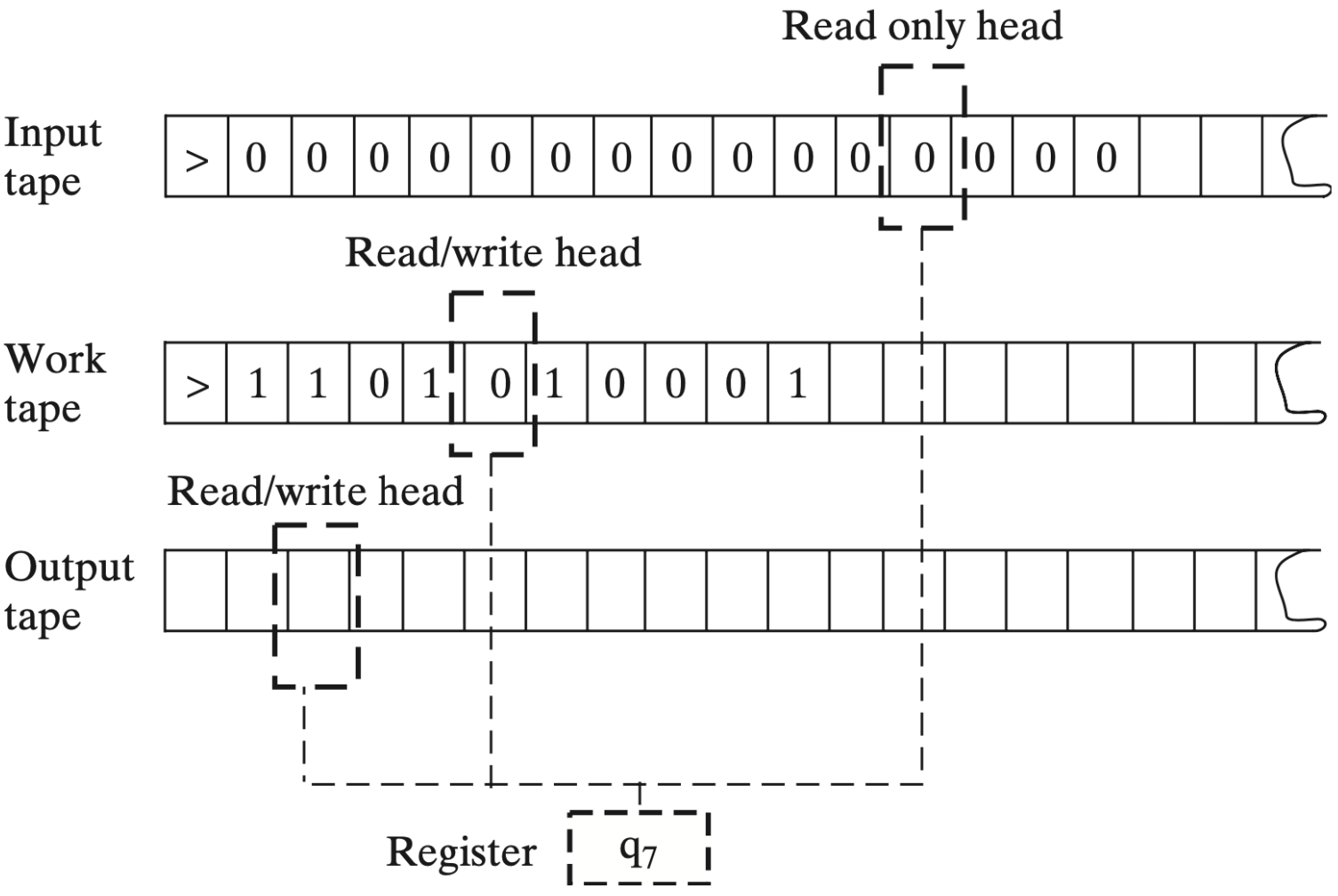



- Given a string $\alpha$, we indicate as $\mathcal{M}_\alpha$ the Turing Machine $\alpha$ encodes.
- There is a so-called **Universal** Turing Machine $\mathcal{U}$, which simulates any other Turing Machine given its string representation: from a pair of strings $(x, \alpha)$, the machine $\mathcal{U}$ simulates the behaviour of $\mathcal{M}_\alpha$ on $x$.
  - The simulation is very efficient: if the running time of $\mathcal{M}_\alpha$ were $T(|x|)$, then $\mathcal{U}$ would take time $O(T(|x|) \log T(|x|))$.
- There are functions $f : \{0,1\}^* \to \{0,1\}$ which are **intrinsically uncomputable** by Turing Machines, and this can be proved formally.
  - This has intimate connections to Gödel's famous **incompleteness theorem**.

# Part II

# The Model, Formally

# A More Detailed View



# The Scratchpad(s)

- It consists of $k$ **tapes**, where a tape is an infinite one-directional line of cells, each of which can hold a simbol from a finite alphabel $\Gamma$, the *alphabet* of the machine.
- Each tape is equipped with **tape head**, which can read or write symbols *from* or *to* the tape. Each head can move left or right.
- The first tape is designated as **input tape**, and is read-only.
- The last tape is the **output tape**, and contains the result of the computation.
  - This slightly deviates from what we have said in our informal account, and is needed to compute arbitrary functions on $\{0,1\}^*$.

# The Instructions

- The machine has a finite set of *states*, called $Q$, which determine the action to be taken at the next step.
- At *each step*, the machine:
  1. **Read** the symbols under the $k$ tape heads.
  2. For the $k-1$ read-write tapes, **replace** the symbol with a new one, or leave it unchanged.
  3. **Change** its state to a new one.
  4. **Move** each of the $k$ tape heads to the left or to the right (or stay in place).

# The Instructions

- The machine has a finite set of *states*, called $Q$, which determine the action to be taken at the next step.
- At *each step*, the machine:
  1. **Read** the symbols under the $k$ tape heads.
  2. For the $k-1$ read-write tapes, **replace** the symbol with a new one, or leave it unchanged.
  3. **Change** its state to a new one.
  4. **Move** each of the $k$ tape heads to the left or to the right (or stay in place).
- These instructions are of course very basic, and far from being close to the kind of instructions programming languages offer.
  - The point here is to have a *simple*, but *expressive* model.

# The Formal Definition

A **Turing Machine** (TM for short) working on $k$ tapes is described as a triple $(\Gamma, Q, \delta)$ containing

- A finite set $\Gamma$ of **tape symbols**, which we assume contains the *blank symbol* $\square$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.

# The Formal Definition

A **Turing Machine** (TM for short) working on $k$ tapes is described as a triple $(\Gamma, Q, \delta)$ containing

- A finite set $\Gamma$ of **tape symbols**, which we assume contains the *blank symbol* $\square$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.
- A finite set $Q$ of **states** which includes a designated *initial state* $q_{\texttt{init}}$ and a designated final state $q_{\texttt{halt}}$.

# The Formal Definition

A **Turing Machine** (TM for short) working on $k$ tapes is described as a triple $(\Gamma, Q, \delta)$ containing

- A finite set $\Gamma$ of **tape symbols**, which we assume contains the *blank symbol* $\square$, the *start symbol* $\triangleright$, and the binary digits 0 and 1.
- A finite set $Q$ of **states** which includes a designated *initial state* $q_{\texttt{init}}$ and a designated final state $q_{\texttt{halt}}$.
- A **transition function**

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\texttt{L}, \texttt{S}, \texttt{R}\}^k$$

describing the instructions regulating the functioning of the machine at each step.

  - When the first parameter is $q_{\texttt{halt}}$, then $\delta$ cannot touch the tapes nor the heads:

$$\delta(q_{\texttt{halt}}, (\sigma_1, \ldots, \sigma_k)) = (q_{\texttt{halt}}, (\sigma_2 \ldots, \sigma_k), (\texttt{S}, \ldots, \texttt{S}))$$

and the machine is *stuck*.

# Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on $k$ tapes:

- A **configuration** consists of
  - The current state $q$.
  - The contents of the $k$ tapes.
  - The positions of the $k$ tape heads.

  One such configuration will be indicated as $C$.

# Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on $k$ tapes:

- A **configuration** consists of
  - The current state $q$.
  - The contents of the $k$ tapes.
  - The positions of the $k$ tape heads.

  One such configuration will be indicated as $C$.
- The **initial configuration** for the input $x \in \{0,1\}^*$ is the configuration $\mathcal{I}_x$ in which:
  - The current state is $q_{\texttt{init}}$.
  - The first tape contains $\triangleright x$, followed by blank symbols, while the other tapes contain $\triangleright$, followed by blank symbols.
  - The tape heads are positioned on the first symbol of the $k$ tapes.

# Machine Configurations and Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on $k$ tapes:

- A **configuration** consists of
  - The current state $q$.
  - The contents of the $k$ tapes.
  - The positions of the $k$ tape heads.

  One such configuration will be indicated as $C$.
- The **initial configuration** for the input $x \in \{0,1\}^*$ is the configuration $\mathcal{I}_x$ in which:
  - The current state is $q_{\texttt{init}}$.
  - The first tape contains $\triangleright x$, followed by blank symbols, while the other tapes contain $\triangleright$, followed by blank symbols.
  - The tape heads are positioned on the first symbol of the $k$ tapes.
- A **final configuration** for the output $y \in \{0,1\}^*$ is any configuration whose state is $q_{\texttt{halt}}$ and in which the content of the output tape is $\triangleright y$, followed by blank symbols.

# Machine Computations

Given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on $k$ tapes:

- Given any configuration $C$, the transition function $\delta$ determines in a natural way the next configuration $D$, and we write $C \xrightarrow{\delta} D$ if this is the case.
- We say that $\mathcal{M}$ **returns** $y \in \{0,1\}^*$ **on input** $x \in \{0,1\}^*$ **in** $t$ **steps** if

$$\mathcal{I}_x \xrightarrow{\delta} C_1 \xrightarrow{\delta} C_2 \xrightarrow{\delta} \ldots \xrightarrow{\delta} C_t$$

  where $C_t$ is a final configuration for $y$. We write $\mathcal{M}(x)$ for $y$ if this holds.
- Finally, we say that $\mathcal{M}$ **computes** a function $f : \{0,1\}^* \to \{0,1\}^*$ iff $\mathcal{M}(x) = f(x)$ for every $x \in \{0,1\}^*$. In this case, $f$ is said to be **computable.**
  - Beware: for the moment, we do not put any constraint *on the number of steps* $\mathcal{M}$ needs to compute $f(x)$ from $x$.

# The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.

# The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.
- Usually, functions on binary strings are shown to be computable by informally describing *algorithms* or *programs* computing the function.
  - This is fine, *provided* program or algorithm instructions can be **simulated** by TMs.

# The Expressive Power of TMs

- Please take a loot at

  `http://turingmachinesimulator.com`

- Explicitly constructing TMs is **very tedious**.
  - One needs to give the set of states, the set of instructions, etc.
  - One also needs to *prove* that the construction is correct.
- Usually, functions on binary strings are shown to be computable by informally describing *algorithms* or *programs* computing the function.
  - This is fine, *provided* program or algorithm instructions can be **simulated** by TMs.
- There are many other formalisms which are perfectly equivalent to TMs as for the class of computable functions they induce.
  - Examples: Random Access Machines, the $\lambda$-calculus, URMs, Partial Recursive Functions, ...

# Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input $x$ in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, $f$ is said to be **computable** in time $T$.

# Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input $x$ in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, $f$ is said to be **computable** in time $T$.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time $T$ iff $f$ is computable in time $T$.

# Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input $x$ in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, $f$ is said to be **computable** in time $T$.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time $T$ iff $f$ is computable in time $T$.
- Examples.
  - The set of palindrome words is decidable in time $T(n) = 3n$.
  - Computing the parity of binary strings requires time $T(n) = n + 2$.
  - Basic operations like addition and multiplication are computable in polynomial time.

# Efficiency and Runtime

- A TM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ **in time** $T : \mathbb{N} \to \mathbb{N}$ iff $\mathcal{M}$ returns $f(x)$ on input $x$ in a number of steps smaller or equal to $T(|x|)$ for every $x \in \{0,1\}^*$. In this case, $f$ is said to be **computable** in time $T$.
- A language $\mathcal{L}_f \subseteq \{0,1\}^*$ is **decidable** in time $T$ iff $f$ is computable in time $T$.
- Examples.
  - The set of palindrome words is decidable in time $T(n) = 3n$.
  - Computing the parity of binary strings requires time $T(n) = n + 2$.
  - Basic operations like addition and multiplication are computable in polynomial time.
- A function $T : \mathbb{N} \to \mathbb{N}$ is **time-constructible** iff the function on $\{0,1\}^*$ defined as $x \mapsto \llcorner T(|x|) \lrcorner$ is computable.

# On the Robustness of Our Definition

- Question: *why is our definition the right one?*

# On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.

# On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.
- Examples:
  - Rather than an arbitrary tape alphabet $\Gamma$, **restrict to** $\{0, 1, \triangleright, \square\}$.
  - Just **one tape**, rather than many.
  - Tapes can be infinite **in both directions**.

# On the Robustness of Our Definition

- Question: *why is our definition the right one?*
- Actually, there are many details in our definion of a TM which are arbitrary: **many** alternative definitions are available in the literature.
- Examples:
  - Rather than an arbitrary tape alphabet $\Gamma$, **restrict to** $\{0, 1, \triangleright, \square\}$.
  - Just **one tape**, rather than many.
  - Tapes can be infinite **in both directions**.
- In all the cases above (and in many others), one can prove that the more restrictive notion of machine **simulates** the more general one **with polynomial overhead**.
  - We do not have time to see all that, but you are encouraged to take a look at [AroraBarak2009], Section 1.3.1.

# Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of $\delta$, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

# Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of $\delta$, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

- Any parismomious encoding of $\delta$ as a binary string in $\{0,1\}$ thus constitutes an acceptable encoding $\llcorner \mathcal{M} \lrcorner$ of $\mathcal{M}$, provided the following two conditions are satisfied:
  1. **Every string** in $\{0,1\}^*$ represents a TM, i.e. for every $x \in \{0,1\}^*$ there is $\mathcal{M}$ such that $x = \llcorner \mathcal{M} \lrcorner$.
  2. Every TM $\mathcal{M}$ is represented by an **infinitely many strings** (although exactly one is the "canonical" representation $\llcorner \mathcal{M} \lrcorner$ of $\mathcal{M}$.

# Machines as Strings

- One of the very nice consequences of keeping our definition of a Turing Machine very simple is that any machine $\mathcal{M} = (\Gamma, Q, \delta)$ is in fact copmletely determined from the graph of $\delta$, seen as a subset of

$$Q \times \Gamma^k \times Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$$

- Any parismomious encoding of $\delta$ as a binary string in $\{0,1\}$ thus constitutes an acceptable encoding $\llcorner \mathcal{M} \lrcorner$ of $\mathcal{M}$, provided the following two conditions are satisfied:
  1. **Every string** in $\{0,1\}^*$ represents a TM, i.e. for every $x \in \{0,1\}^*$ there is $\mathcal{M}$ such that $x = \llcorner \mathcal{M} \lrcorner$.
  2. Every TM $\mathcal{M}$ is represented by an **infinitely many strings** (although exactly one is the "canonical" representation $\llcorner \mathcal{M} \lrcorner$ of $\mathcal{M}$.
- The two conditions above are not essential in any other contexts (i.e. when the encoded data is not a program), but are technically crucial here.

# The Universal Turing Machine

**Theorem (UTM, Efficiently)**

*There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, it holds that $\mathcal{U}(x, \alpha) = \mathcal{M}_\alpha(x)$, where $\mathcal{M}_\alpha$ denotes the TM represented by $\alpha$. Moreover, if $\mathcal{M}_\alpha$ halts on input $x$ within $T$ steps then $\mathcal{U}(x, \alpha)$ halts within $CT\log(T)$ steps, where $C$ is independent of $|x|$ and depending only on $\mathcal{M}_\alpha$.*

# The Universal Turing Machine

**Theorem (UTM, Efficiently)**

*There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, it holds that $\mathcal{U}(x,\alpha) = \mathcal{M}_\alpha(x)$, where $\mathcal{M}_\alpha$ denotes the TM represented by $\alpha$. Moreover, if $\mathcal{M}_\alpha$ halts on input $x$ within $T$ steps then $\mathcal{U}(x,\alpha)$ halts within $CT\log(T)$ steps, where $C$ is independent of $|x|$ and depending only on $\mathcal{M}_\alpha$.*

- A proof of the Theorem above in its full generality requires quite a bit of work.
- We will see the proof of a version of it in which the $O(T\log(T))$ bound is replaced by a polynomial one.

# Uncomputability

**Theorem (Uncomputable Functions Exist)**

*There exists a function* $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

# Uncomputability

**Theorem (Uncomputable Functions Exist)**

*There exists a function* $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

# Uncomputability

**Theorem (Uncomputable Functions Exist)**

*There exists a function* $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

- Indeed, if $uc$ were computable, there would be a TM $\mathcal{M}$ such that $\mathcal{M}(\alpha) = uc(\alpha)$ for every $\alpha$, and in particular when $\alpha = \llcorner \mathcal{M} \lrcorner$.

# Uncomputability

**Theorem (Uncomputable Functions Exist)**

*There exists a function* $uc : \{0,1\}^* \to \{0,1\}^*$ *that is not computable by any Turing Machine.*

- The proof of the Theorem above is constructive. It suffices to consider the function

$$uc(\alpha) = \begin{cases} 0 & \text{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

- Indeed, if $uc$ were computable, there would be a TM $\mathcal{M}$ such that $\mathcal{M}(\alpha) = uc(\alpha)$ for every $\alpha$, and in particular when $\alpha = \llcorner \mathcal{M} \lrcorner$.
- This would be a contraddiction, because by definition

$$uc(\llcorner \mathcal{M} \lrcorner) = 1 \Leftrightarrow \mathcal{M}(\llcorner \mathcal{M} \lrcorner) \neq 1$$

# The Halting Problem

- The function $uc$ we proved uncomputable does not represent an interesting computationl task.

# The Halting Problem

- The function $uc$ we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function $halt$ defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute $halt$ would mean being able to check algorithms for termination.

# The Halting Problem

- The function $uc$ we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function $halt$ defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute $halt$ would mean being able to check algorithms for termination.

**Theorem (Uncomputability of $halt$)**

*The function halt is not computable by any TM.*

# The Halting Problem

- The function $uc$ we proved uncomputable does not represent an interesting computationl task.
- Consider, instead, the function $halt$ defined as follows:

$$halt(\llcorner(\alpha, x)\lrcorner) = \begin{cases} 1 & \text{if } \mathcal{M}_\alpha \text{ halts on input } x; \\ 0 & \text{otherwise} \end{cases}$$

- Being able to compute $halt$ would mean being able to check algorithms for termination.

**Theorem (Uncomputability of *halt*)**

*The function halt is not computable by any TM.*

- This result could be seen as a way to reinterpret Gödel's first incompleteness theorem "computationally".

# Thank You!

Questions?

# Languages and Algorithms for Artificial Intelligence (Third Module)

## Polynomial Time Computable Problems

Ugo Dal Lago



University of Bologna, Academic Year 2019/2020

# Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
  - It is *not* a set of TMs, although it is defined based on TMs.
  - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).

# Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
  - It is *not* a set of TMs, although it is defined based on TMs.
  - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).
- Let $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class $\mathbf{DTIME}(T(n))$ iff *there is* a TM deciding $\mathcal{L}$ and running in time $n \mapsto c \cdot T(n)$ for some constant $c$.

# Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
  - It is *not* a set of TMs, although it is defined based on TMs.
  - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).
- Let $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class $\mathbf{DTIME}(T(n))$ iff *there is* a TM deciding $\mathcal{L}$ and running in time $n \mapsto c \cdot T(n)$ for some constant $c$.
- The letter "D" in $\mathbf{DTIME}(\cdot)$ refers to *determinism*: the machines on which the class is based work deterministically.
- Should we study efficiently solvable tasks by way of classes in the form $\mathbf{DTIME}(T(n))$?
  - The answer is bound to be negative, because these classes are not **robust**, they depends too much on
  - We need a larger class.

# The Class **P**

- The class **P** is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class **P** includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time $P$;
  3. where $P$ is a any polynomial.

# The Class **P**

- The class **P** is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class **P** includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time $P$;
  3. where $P$ is a any polynomial.
- Indeed, for any any polynomial $P$ there are $c, d > 0$ such that $P(n) \geq c \cdot n^d$ for sufficiently large $n$.

# The Class **P**

- The class **P** is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class **P** includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time $P$;
  3. where $P$ is a any polynomial.
- Indeed, for any any polynomial $P$ there are $c, d > 0$ such that $P(n) \geq c \cdot n^d$ for sufficiently large $n$.
- Please observe that $c$ and $d$ can be arbitrarily large, so a TM deciding $\mathcal{L}$ and working in time $10^{20} \cdot n^{10^{30}}$ is a witness of $\mathcal{L}$ being in **P**.

# The Class **P**

- The class **P** is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class **P** includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time $P$;
  3. where $P$ is a any polynomial.
- Indeed, for any any polynomial $P$ there are $c, d > 0$ such that $P(n) \geq c \cdot n^d$ for sufficiently large $n$.
- Please observe that $c$ and $d$ can be arbitrarily large, so a TM deciding $\mathcal{L}$ and working in time $10^{20} \cdot n^{10^{30}}$ is a witness of $\mathcal{L}$ being in **P**.
- **P** is generally considered as *the* class of efficiently decidable languages.

# The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!

# The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!
- **The Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a (possibly *very large*) overhead in time.
  - The class of computable tasks *would not be larger* (actually, equal!) if formalized in a realistic way, but differently.
  - Most scientists believe in it.

# The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!
- **The Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a (possibly *very large*) overhead in time.
  - The class of computable tasks *would not be larger* (actually, equal!) if formalized in a realistic way, but differently.
  - Most scientists believe in it.
- **The Strong Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a *polynomial* overhead in time. ($n$ steps on the computer requires $n^c$ on TMs, where $c$ only depends on the computer), and viceversa.
  - The class **P** would be *the same* if defined based on other realistic models of computation.
  - This is more controversial (due to, e.g., quantum computation).

# Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.

# Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.
- **Exponents are Often Small**
  - In principle, the exponent $c$ bounding the time of any machine deciding $\mathcal{L} \in \mathbf{P}$ can be huge.
  - For many problems of interest and in **P**, there are TMs working within quadratic or cubit bounds.

# Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.
- **Exponents are Often Small**
  - In principle, the exponent $c$ bounding the time of any machine deciding $\mathcal{L} \in \mathbf{P}$ can be huge.
  - For many problems of interest and in **P**, there are TMs working within quadratic or cubit bounds.
- **Nice Closure Properties**
  - The class is closed various operations on programs, e.g. composition and bounded loops (with some restrictions!).
  - As a consequence, it is relatively easy to prove that a given problem/task is *in* the class: it suffices to give an algorithm solving the problem and working in polynomial time, without constructing the TM explicitly.

# Some Criticisms on **P**

- **Worst-Case is Not Realistic**
  - The definition of **P** is intrinsically based on worst-case complexity: there must be *a* polynomial and *a* TM such that *for every input*...
  - It is good enough is our problem takes little time *on the types of inputs* which arise in practice, and not on *all* of them.
  - Solutions: Average-case Complexity, Approximation Algorithms

# Some Criticisms on **P**

- **Worst-Case is Not Realistic**
  - The definition of **P** is intrinsically based on worst-case complexity: there must be *a* polynomial and *a* TM such that *for every input...*
  - It is good enough is our problem takes little time *on the types of inputs* which arise in practice, and not on *all* of them.
  - Solutions: Average-case Complexity, Approximation Algorithms
- **Alternative Computational Models**
  - Feasibility can be also be defined for classes dealine with arbitrary precision computation, with randomized computation, or with quantum computation.
  - Solutions: the class **P** can be spelled out with other computational models in mind, giving rise to other classes (e.g. **BPP** or **BQP**).
- **Why Just Decision Problems?**
  - As alreday pointed out, not all tasks can be modeled this way.

# Thank You!

Questions?

# The Complexity Class **FP**

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function $f$ is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing $f$ and running in time $n \mapsto c \cdot T(n)$ for some constant $c$.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

# The Complexity Class **FP**

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function $f$ is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing $f$ and running in time $n \mapsto c \cdot T(n)$ for some constant $c$.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

- For every $\mathcal{L} \in \mathbf{P}$, the characteristic function $f$ of $\mathcal{L}$ is trivially in **FP**.

# The Complexity Class **FP**

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function $f$ is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing $f$ and running in time $n \mapsto c \cdot T(n)$ for some constant $c$.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

- For every $\mathcal{L} \in \mathbf{P}$, the characteristic function $f$ of $\mathcal{L}$ is trivially in **FP**.
- For certain classes of functions (e.g. those corresponding to optimization problems), there are canonical ways to turn a function $f$ into a language $\mathcal{L}_f$
  - In general, however, it is not true that $f \in \mathbf{FP}$ implies $\mathcal{L}_f \in \mathbf{P}$.

# Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.

# Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.

# Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.

# Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.
- **Strings**: string matching, approximate string matching, etc.

# Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.
- **Strings**: string matching, approximate string matching, etc.
- **Optimization Problems**: linear programming, maximum cost flow, etc.

# How to Prove a Task Being in **P** or **FP**

- In theory, one should give a TM working within some polynomial bounds, and prove that the machines decides the language (or computes the function).

# How to Prove a Task Being in **P** or **FP**

- In theory, one should give a TM working within some polynomial bounds, and prove that the machines decides the language (or computes the function).
- This is however too cumbersome, and instead of going through TMs, one often goes informal and uses the so called pseudocode.
- Example.
  - Suppose you want to show the following problem to be computable in polynomial time: given two strings $x, y \in \{0,1\}^*$. determine if the $x$ contains an instance of $y$.
  - A pseudocode solving the problem above is the following:

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

# How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

# How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

- The input can be easily encoded as a binary string.

# How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

- The input can be easily encoded as a binary string.
- The total number of instruction is polynomially bounded.
  - Indeed it is $O(|x|)$.

# How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

- The input can be easily encoded as a binary string.
- The total number of instruction is polynomially bounded.
  - Indeed it is $O(|x|)$.
- All intermediate results are polynomially bounded in length.
  - Indeed, $i$ cannot be greater than $O(|x|)$, thus its length is $O(\lg |x|)$.

# How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        i ← i + 1
    else
        return True
    end
end
return False
```

- The input can be easily encoded as a binary string.
- The total number of instruction is polynomially bounded.
  - Indeed it is $O(|x|)$.
- All intermediate results are polynomially bounded in length.
  - Indeed, $i$ cannot be greater than $O(|x|)$, thus its length is $O(\lg |x|)$.
- Each instruction takes polynomial time to be simulated.
  - Comparing two strings of length $|y|$ can be done in polynomial time in $|y|$, thus polynomial in $|\llcorner(x, y)\lrcorner|$.

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \quad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \quad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \qquad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.
- Of course, it holds that

$$\mathbf{P} \subseteq \mathbf{EXP} \qquad \mathbf{FP} \subseteq \mathbf{FEXP}$$

# The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \qquad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.
- Of course, it holds that

$$\mathbf{P} \subseteq \mathbf{EXP} \qquad \mathbf{FP} \subseteq \mathbf{FEXP}$$

**Theorem**

*The two inclusions above are strict.*

# Thank You!

Questions?

# Languages and Algorithms for Artificial Intelligence (Third Module)

## Between the Feasible and the Unfeasible

Ugo Dal Lago



University of Bologna, Academic Year 2019/2020

# The Border Between the Tractable and the Intractable

- Up to now, we have encountered (essentially speaking) two complexity classes, namely:
  - **P**, which contains those problems which can be solved in polynomial time, so the **tractable** ones.
  - **EXP**, which contains the whole of **P**, but also some problems which *cannot* be solved in polynomial time, intrinsically requiring exponential time (and as such, **intractable**).

# The Border Between the Tractable and the Intractable

- Up to now, we have encountered (essentially speaking) two complexity classes, namely:
  - **P**, which contains those problems which can be solved in polynomial time, so the **tractable** ones.
  - **EXP**, which contains the whole of **P**, but also some problems which *cannot* be solved in polynomial time, intrinsically requiring exponential time (and as such, **intractable**).
- It's now time to study the "border" between tractability and intractability:
  - Between **P** and **EXP**, one can define *many other* classes. i.e. there are many ways of defining a class **A** such that

$$\mathbf{P} \subseteq \mathbf{A} \subseteq \mathbf{EXP}$$

  - This is a formidable tool to classify those problems in **EXP** for which *we do not know* whether they are in **P** (and there are *so many* of them).

# Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.

# Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elments of $\mathcal{L}$ are those strings for which we can find a *certificate* $y$ (of polynomial length) such that the pair $(x, y)$ passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.

# Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elments of $\mathcal{L}$ are those strings for which we can find a *certificate* $y$ (of polynomial length) such that the pair $(x, y)$ passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?

# Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elments of $\mathcal{L}$ are those strings for which we can find a *certificate* $y$ (of polynomial length) such that the pair $(x, y)$ passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?
  - *Not necessarily*: given $x$, we can check whether $x \in \mathcal{L}$ by $(x, y) \in \mathcal{A}$ for all possible possible $y$ such that $|y| \leq p(|x|)$, of which however there are *exponentially* many.
  - Of course this does *not* rule out other strategies to decide $\mathcal{L}$.

# Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elments of $\mathcal{L}$ are those strings for which we can find a *certificate* $y$ (of polynomial length) such that the pair $(x, y)$ passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?
  - *Not necessarily*: given $x$, we can check whether $x \in \mathcal{L}$ by $(x, y) \in \mathcal{A}$ for all possible possible $y$ such that $|y| \leq p(|x|)$, of which however there are *exponentially* many.
  - Of course this does *not* rule out other strategies to decide $\mathcal{L}$.
- The *take-away message* is thus the following: **crafting** a solution for the problem $x$ (i.e., finding $y$) can potentially be more difficult than just **checking** $y$ to be a solution to $x$.

# The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

# The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - $M$ is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner (x,y) \lrcorner) = 1$ is said to be a **certificate** for $x$.

# The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - $M$ is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner (x,y) \lrcorner) = 1$ is said to be a **certificate** for $x$.
- Differently from **P** and **EXP**, the class **NP** does not have a natural counterpart as a class of *functions*.

# The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - $M$ is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner (x,y) \lrcorner) = 1$ is said to be a **certificate** for $x$.
- Differently from **P** and **EXP**, the class **NP** does not have a natural counterpart as a class of *functions*.

**Theorem**

**P $\subseteq$ NP $\subseteq$ EXP**

# Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least $k$.

# Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least $k$.
   - Certificate: the certificate here is just $W$. Indeed, checking whether $W$ is independent can easily be done in polynomial time.

# Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least $k$.
   - Certificate: the certificate here is just $W$. Indeed, checking whether $W$ is independent can easily be done in polynomial time.
2. **Subset Sum**
   - It asks whether, given a sequence of natural numbers $n_1, \ldots, n_m$ and a number $k$, there exists a subset $I$ of $\{1, \ldots, m\}$ such that $\sum_{i \in I} n_i = k$.

# Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least $k$.
   - Certificate: the certificate here is just $W$. Indeed, checking whether $W$ is independent can easily be done in polynomial time.
2. **Subset Sum**
   - It asks whether, given a sequence of natural numbers $n_1, \ldots, n_m$ and a number $k$, there exists a subset $I$ of $\{1, \ldots, m\}$ such that $\sum_{i \in I} n_i = k$.
   - Certificate: again, the certificate here is just $I$: checking whether $\sum_{i \in I} n_i = k$ just amounts to some additions and comparisons.

# Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether $n$ is composite (i.e., not prime).

# Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether $n$ is composite (i.e., not prime).
   - Certificate: it is the factorization of $n$, a pair $(m, l)$ of natural numbers (greater than 2) such that $n = m \cdot l$.

# Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether $n$ is composite (i.e., not prime).
   - Certificate: it is the factorization of $n$, a pair $(m, l)$ of natural numbers (greater than 2) such that $n = m \cdot l$.
4. **Factoring**
   - Given three numbers $n, m, l$, it asks whether $n$ has a prime factor in the interval $[m, l]$.

# Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether $n$ is composite (i.e., not prime).
   - Certificate: it is the factorization of $n$, a pair $(m, l)$ of natural numbers (greater than 2) such that $n = m \cdot l$.
4. **Factoring**
   - Given three numbers $n, m, l$, it asks whether $n$ has a prime factor in the interval $[m, l]$.
   - Certificate: it can be taken to be the prime number $p$: checking that $p \in [m, l]$ and that $p$ divides $n$ is easy. Instead, checking that $p$ is prime requires a lot of work, but can indeed be done in polynomial time.

# Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of $m$ linear inequalities with rational coefficients over $n$ variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.

# Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of $m$ linear inequalities with rational coefficients over $n$ variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.

# Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of $m$ linear inequalities with rational coefficients over $n$ variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.
4. **Decisional 0/1 Linear Programming**
   - Given a sequence of linear inequalities as above, decide whether there is an assignment *of zeros and ones* to the variables $x_1, \ldots, x_n$ rendering all the inequalities true.

# Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of $m$ linear inequalities with rational coefficients over $n$ variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.
4. **Decisional 0/1 Linear Programming**
   - Given a sequence of linear inequalities as above, decide whether there is an assignment *of zeros and ones* to the variables $x_1, \ldots, x_n$ rendering all the inequalities true.
   - Certificate: again, the assignments suffices.

# Examples Problems in **NP**

- Some of the aforementioned problems are also in **P**:
  - *Decisional linear programming* can be proved to be in **P** thanks to, e.g., the Ellipsoid algorithm.
  - The *composite numbers* problem can be proved itself to be in **P**, thanks to a breakthrough recent result, namely the so-called AKS algorithm.

# Examples Problems in **NP**

- Some of the aforementioned problems are also in **P**:
  - *Decisional linear programming* can be proved to be in **P** thanks to, e.g., the Ellipsoid algorithm.
  - The *composite numbers* problem can be proved itself to be in **P**, thanks to a breakthrough recent result, namely the so-called AKS algorithm.
- All the other problems are currently **not known** to be in **P**.
  - Are they all equivalent in terms of their inherent computational difficulty?
  - Is there any way isolate those problems in **NP** whose difficult is maximal, i.e. they are at least as hard as all other problems in **NP**?

# Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.

# Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.
- The only differences between a NDTM and an ordinary TM is that the former has:
  - Two transition functions $\delta_0$ and $\delta_1$ rather than just one. *At every step*, the machine chooses nondeterministically one between the two transition functions and proceed according to it
  - A special state $q_{\texttt{accept}}$.

# Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.
- The only differences between a NDTM and an ordinary TM is that the former has:
  - Two transition functions $\delta_0$ and $\delta_1$ rather than just one. *At every step*, the machine chooses nondeterministically one between the two transition functions and proceed according to it
  - A special state $q_{\texttt{accept}}$.
- We say that a NDTM $M$:
  - **Accepts** the input $x \in \{0,1\}^*$ iff *there exists* one among the many possible evolutions of the machine $M$ when fed with $x$ which makes it reaching $q_{\texttt{accept}}$.
  - **Rejects** the input $x \in \{0,1\}^*$ iff *none* of the aformentioned evolutions leads to $q_{\texttt{accept}}$.

# Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.
- The only differences between a NDTM and an ordinary TM is that the former has:
  - Two transition functions $\delta_0$ and $\delta_1$ rather than just one. *At every step*, the machine chooses nondeterministically one between the two transition functions and proceed according to it
  - A special state $q_{\texttt{accept}}$.
- We say that a NDTM $M$:
  - **Accepts** the input $x \in \{0,1\}^*$ iff *there exists* one among the many possible evolutions of the machine $M$ when fed with $x$ which makes it reaching $q_{\texttt{accept}}$.
  - **Rejects** the input $x \in \{0,1\}^*$ iff *none* of the aformentioned evolutions leads to $q_{\texttt{accept}}$.

# Time-Bounded NDTMs

- We say that a NDTM $M$ *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, $M$ reaches either the halting state or $q_{\texttt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.

# Time-Bounded NDTMs

- We say that a NDTM $M$ *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, $M$ reaches either the halting state or $q_{\texttt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM $M$ working in time $T$ and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

# Time-Bounded NDTMs

- We say that a NDTM $M$ *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, $M$ reaches either the halting state or $q_{\texttt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM $M$ working in time $T$ and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

**Theorem**

$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NDTIME}(n^c)$

# Time-Bounded NDTMs

- We say that a NDTM $M$ *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, $M$ reaches either the halting state or $q_{\texttt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM $M$ working in time $T$ and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

**Theorem**

$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NDTIME}(n^c)$

- All this being said, NDTMs, contrarily to TMs, are *not* meant to model any form of physically realisable machine.

# Thank You!

Questions?

# Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.

# Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.

# Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should be* different.

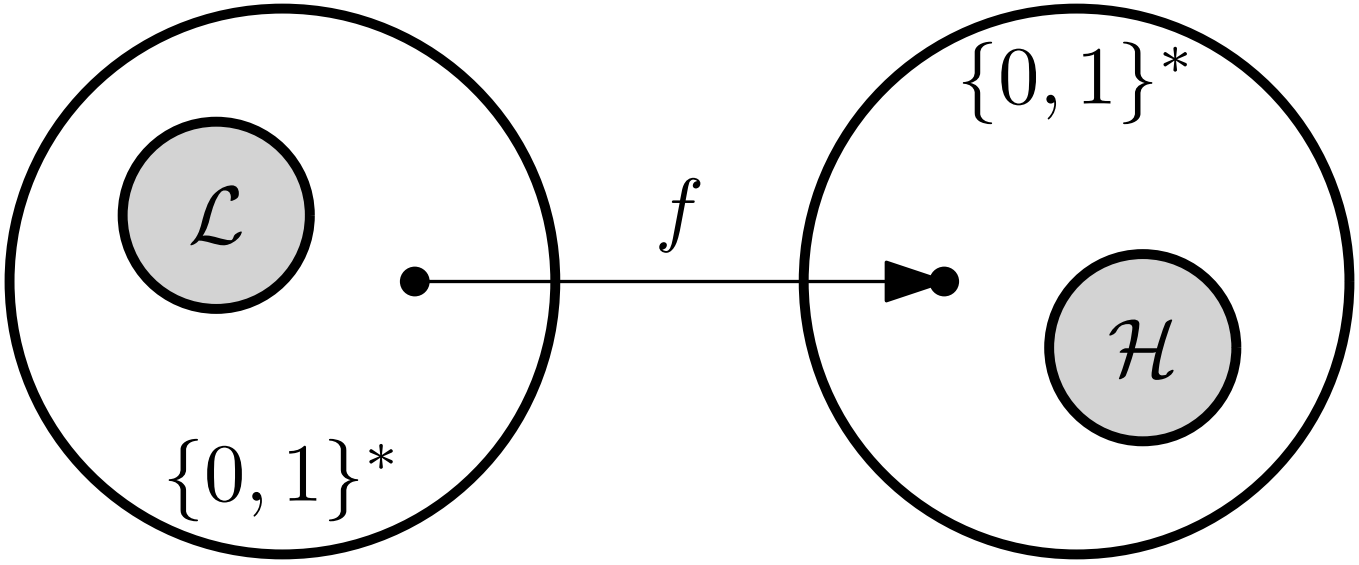
# Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should b*e different.
- What can we thus conclude from the fact that a language $\mathcal{L}$ is **in** the class **NP**?
  - Not much, actually! We can only conclude that it is not *too* complicated to solve it.

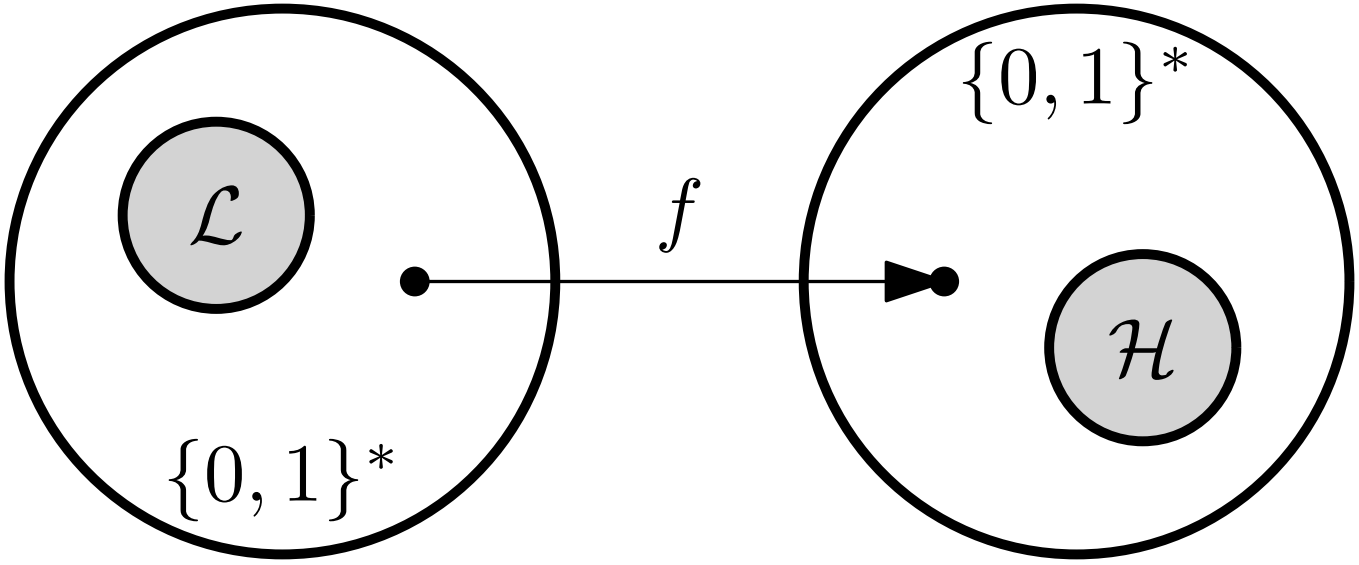
# Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should b*e different.
- What can we thus conclude from the fact that a language $\mathcal{L}$ is **in** the class **NP**?
  - Not much, actually! We can only conclude that it is not *too* complicated to solve it.
- We need something else, namely a (pre-order) relation between languages such that two languages being in relation tells us something precise about the **relative** difficulty of deciding them.

# Reductions



# Reductions




- The language $\mathcal{L}$ is said to be **polynomial-time reducible** to another language $\mathcal{H}$ iff there is a polytime computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in \mathcal{L}$ iff $f(x) \in \mathcal{H}$.
- In this case, we write $\mathcal{L} \leq_p \mathcal{H}$.

# Reductions and Complexity

- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.

# Reductions and Complexity

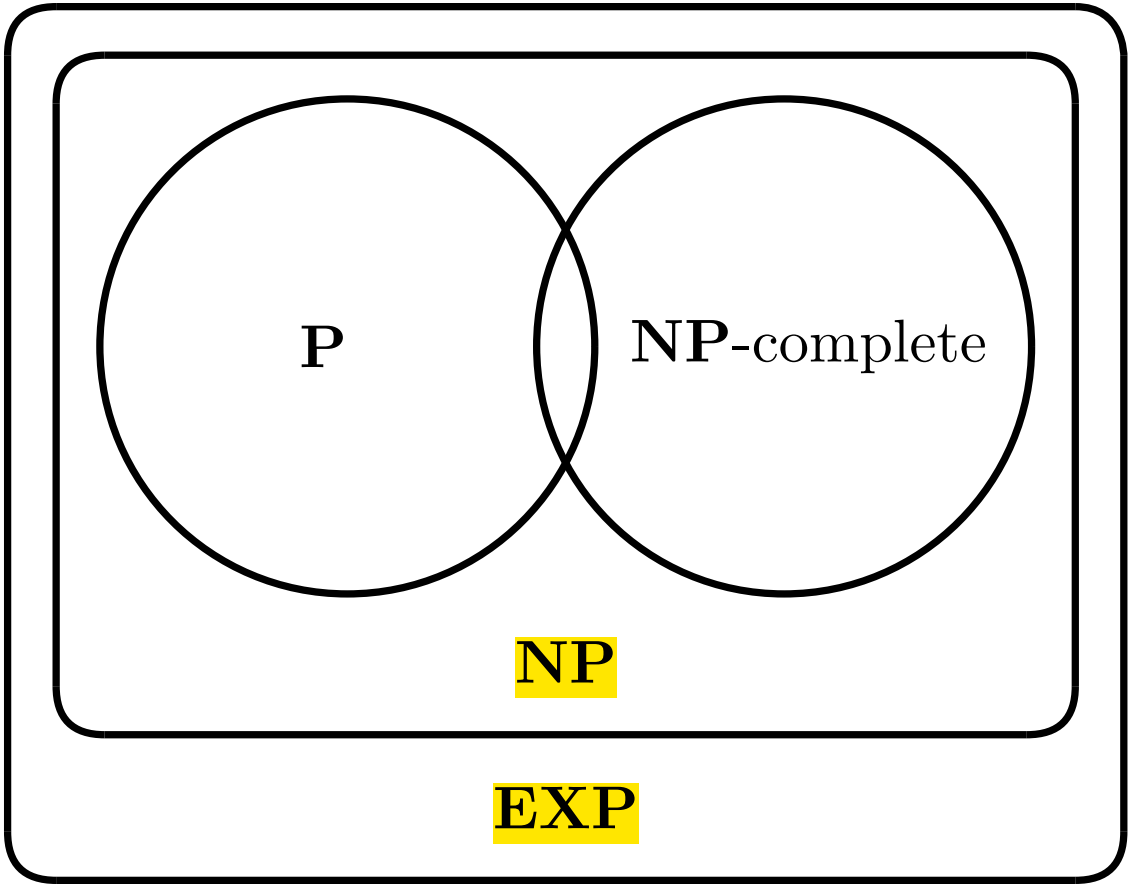
- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.
- A language $\mathcal{H} \subseteq \{0,1\}^*$ is said to be:
  - **NP**-hard if $\mathcal{L} \leq_p \mathcal{H}$ for every $\mathcal{L} \in \mathbf{NP}$.
  - **NP**-complete if $\mathcal{H}$ is **NP**-hard, and $\mathcal{H} \in \mathbf{NP}$.

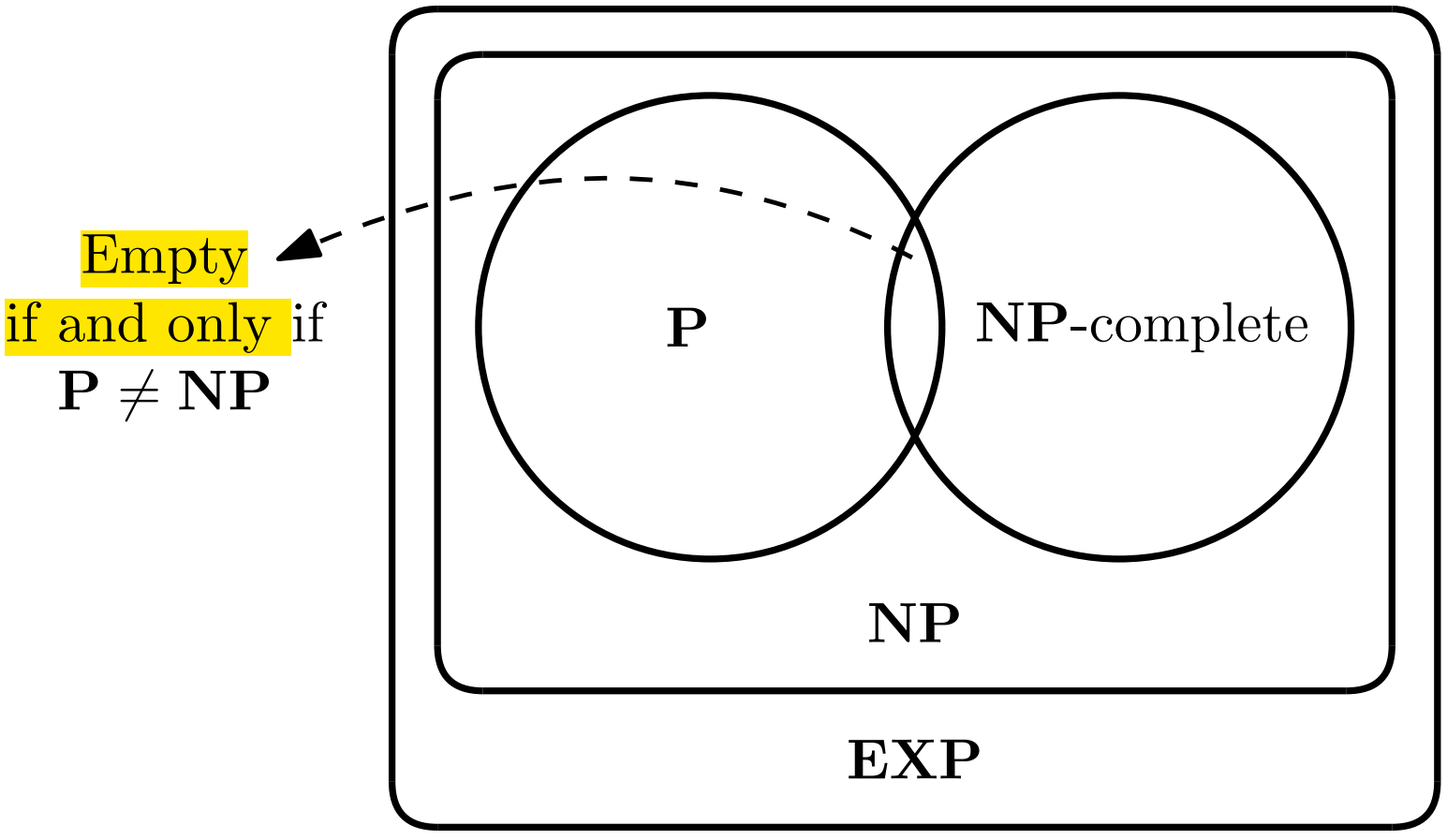
# Reductions and Complexity

- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.
- A language $\mathcal{H} \subseteq \{0,1\}^*$ is said to be:
  - $\mathbf{NP}$-hard if $\mathcal{L} \leq_p \mathcal{H}$ for every $\mathcal{L} \in \mathbf{NP}$.
  - $\mathbf{NP}$-complete if $\mathcal{H}$ is $\mathbf{NP}$-hard, and $\mathcal{H} \in \mathbf{NP}$.

**Theorem**

1. *The relation $\leq_p$ is a pre-order (i.e. it is reflexive and transitive).*
2. *If $\mathcal{L}$ is $\mathbf{NP}$-hard and $\mathcal{L} \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.*
3. *If $\mathcal{L}$ is $\mathbf{NP}$-complete, then $\mathcal{L} \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$.*


P
NP-complete
NP
EXP


Empty
if and only if
$\mathbf{P} \neq \mathbf{NP}$
P
NP-complete
NP
EXP

# The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine, more or less the way we have done it while proving the Hierarchy Theorem.

# The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine, more or less the way we have done it while proving the Hierarchy Theorem.
- Let `TMSAT` be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^*. \mathcal{M}_\alpha \text{ outputs 1 on input } (x, u) \text{ within } t \text{ steps}\}$$

# The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine, more or less the way we have done it while proving the Hierarchy Theorem.
- Let `TMSAT` be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^*.\mathcal{M}_\alpha \text{ outputs } 1 \text{ on input } (x,u) \text{ within } t \text{ steps}\}$$

**Theorem**

`TMSAT` *is* **NP**-*complete.*

# The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine, more or less the way we have done it while proving the Hierarchy Theorem.
- Let `TMSAT` be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^*. \mathcal{M}_\alpha \text{ outputs } 1 \text{ on input } (x,u) \text{ within } t \text{ steps}\}$$

**Theorem**

`TMSAT` *is* **NP***-complete.*

- Although interesting from a purely theoretical perspective, the language `TMSAT` is very specifically tied to Turing Machines, and thus of no practical importance.

# A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,

# A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.

# A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula $F$ and an assignment $\rho$ of elements from $\{0, 1\}$ to the propositional variables in $F$, one can define the **truth value** for $F$, indicated as $[\![F]\!]$, by induction on $F$:

$$[\![X]\!] = \rho(X) \qquad [\![F \vee G]\!] = [\![F]\!] + [\![G]\!]$$
$$[\![F \wedge G]\!] = [\![F]\!] \cdot [\![G]\!] \qquad [\![\neg F]\!] = 1 - [\![F]\!]$$

# A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula $F$ and an assignment $\rho$ of elements from $\{0, 1\}$ to the propositional variables in $F$, one can define the **truth value** for $F$, indicated as $[\![F]\!]$, by induction on $F$:

$$[\![X]\!] = \rho(X) \qquad [\![F \vee G]\!] = [\![F]\!] + [\![G]\!]$$
$$[\![F \wedge G]\!] = [\![F]\!] \cdot [\![G]\!] \qquad [\![\neg F]\!] = 1 - [\![F]\!]$$

- Examples: $[\![X \vee \neg X]\!] = 1$ for every $\rho$, while the truth value $[\![X \wedge (Y \vee \neg Z)]\!]$ equals 1 only for some of the possible $\rho$.

# A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula $F$ and an assignment $\rho$ of elements from $\{0, 1\}$ to the propositional variables in $F$, one can define the **truth value** for $F$, indicated as $[\![F]\!]$, by induction on $F$:

$$[\![X]\!] = \rho(X) \qquad [\![F \vee G]\!] = [\![F]\!] + [\![G]\!]$$
$$[\![F \wedge G]\!] = [\![F]\!] \cdot [\![G]\!] \qquad [\![\neg F]\!] = 1 - [\![F]\!]$$

- Examples: $[\![X \vee \neg X]\!] = 1$ for every $\rho$, while the truth value $[\![X \wedge (Y \vee \neg Z)]\!]$ equals 1 only for some of the possible $\rho$.
- A formula $F$ is **satisfiable** iff there is one $\rho$ such that $[\![F]\!] = 1$.

# The Cook-Levin Theorem

- A propositional formula $F$ is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.

# The Cook-Levin Theorem

- A propositional formula $F$ is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.
- The disjunctions in a CNF are said to be **clauses**, and a $k$**CNF** is a CNF whose clauses contains at most $k \in \mathbb{N}$ literals. Examples: the two formulas $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are 2CNFs, but not 1CNFs.

# The Cook-Levin Theorem

- A propositional formula $F$ is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.
- The disjunctions in a CNF are said to be **clauses**, and a $k$**CNF** is a CNF whose clauses contains at most $k \in \mathbb{N}$ literals. Examples: the two formulas $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are 2CNFs, but not 1CNFs.

**Theorem (Cook-Levin)**

*The following two languages are* **NP***-complete:*

$$\texttt{SAT} = \{\llcorner F \lrcorner \mid F \text{ is a satisfiable CNF}\}$$

$$\texttt{3SAT} = \{\llcorner F \lrcorner \mid F \text{ is a satisfiable 3CNF}\}$$

# Thank You!

Questions?

# The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.

# The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.

# The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.
  - To do that, we consider any possible polynomial $p$ and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$

# The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.
  - To do that, we consider any possible polynomial $p$ and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$
  - We define a polynomial-time transformation $x \mapsto \varphi_x$ from strings to CNFs such that

$$\varphi_x \in \texttt{SAT} \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1$$

    - This is the crucial step, and requires quite some work. It amounts to showing that computation in TM is *inherently local*.

# The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.
  - To do that, we consider any possible polynomial $p$ and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$
  - We define a polynomial-time transformation $x \mapsto \varphi_x$ from strings to CNFs such that

$$\varphi_x \in \texttt{SAT} \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1$$

    - This is the crucial step, and requires quite some work. It amounts to showing that computation in TM is *inherently local*.
  - Finally, we need to show that $\texttt{SAT} \leq_p \texttt{3SAT}$.

# Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.

# Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.

# Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.
- By proving that $\mathcal{L} \in \mathbf{NP}$?
  - Again, this **does not** mean much.

# Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.
- By proving that $\mathcal{L} \in \mathbf{NP}$?
  - Again, this **does not** mean much.
- By proving that $\mathcal{L}$ is **NP**-complete?
  - **Yes**, this way you prove that the problem is not so hard (being in **NP**), but not so easy either (unless $\mathbf{P} = \mathbf{NP}$).

# Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:

# Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:
  1. That $\mathcal{L}$ is in **NP**.
     - This amounts to showing that there are $p$ polynomial and $\mathcal{M}$ polytime TM such that $\mathcal{L}$ can be written as

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1\}$$

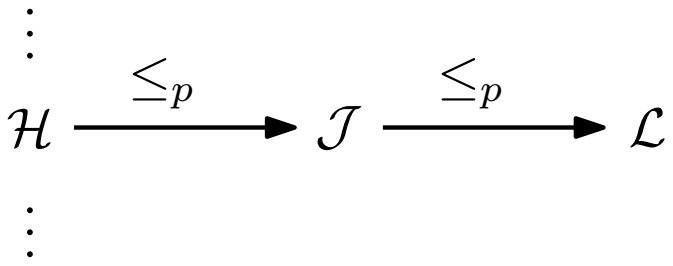
     - This is typically rather easy.

# Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:
  1. That $\mathcal{L}$ is in **NP**.
     - This amounts to showing that there are $p$ polynomial and $\mathcal{M}$ polytime TM such that $\mathcal{L}$ can be written as

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1\}$$

     - This is typically rather easy.
  2. That any other language $\mathcal{H} \in \mathbf{NP}$ is such that $\mathcal{H} \leq_p \mathcal{L}$.
     - We can of course prove the statement directly.
     - More often (e.g. when showing `3SAT` **NP**-complete), one rather proves that $\mathcal{J} \leq_p \mathcal{L}$ for a language $\mathcal{J}$ which is already known to be **NP**-complete.
     - This is correct, simply because $\leq_p$ is transitive:



# **NP**-complete Problems: Examples

**Theorem**

*The Maximum Independent Set Problem* `INDSET` *is* **NP***-complete.*

# **NP**-complete Problems: Examples

**Theorem**

*The Maximum Independent Set Problem* `INDSET` *is* **NP***-complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

# **NP**-complete Problems: Examples

**Theorem**

*The Maximum Independent Set Problem* `INDSET` *is* **NP***-complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

**Theorem**

*The Subset Sum Problem* `SUBSETSUM` *is* **NP***-complete.*

- There are polytime reductions from `3SAT` to a variation `OL3SAT` of it, and from the latter to `SUBSETSUM`.

# **NP**-complete Problems: Examples

**Theorem**

*The Maximum Independent Set Problem* `INDSET` *is* **NP***-complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

**Theorem**

*The Subset Sum Problem* `SUBSETSUM` *is* **NP***-complete.*

- There are polytime reductions from `3SAT` to a variation `OL3SAT` of it, and from the latter to `SUBSETSUM`.

**Theorem**

*The Decisional 0/1 Linear Programming Problem* `ILP` *is* **NP***-complete*

- There is an easy polytime reduction from `SAT` to `ILP`.

# The Graph of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

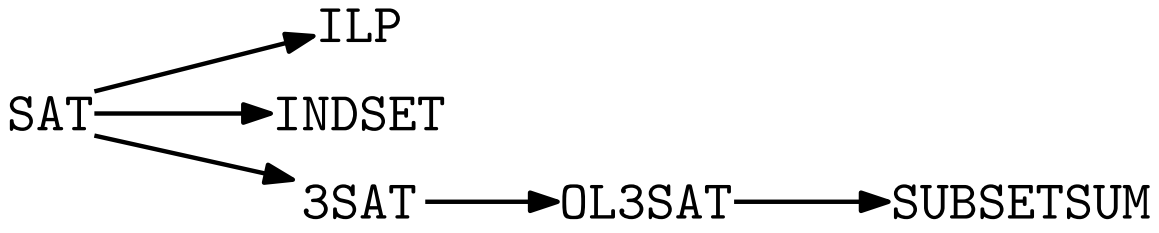
  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.

# The Graph of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.
- This being said, defining a reduction from $\mathcal{L}$ to $\mathcal{H}$ is sometimes easy, and sometimes very difficult, and it is instructive to think at **NP**-complete problems as forming a graph, where edges are *natural* polytime reductions
  - A fragment, the one we have encountered so far is the following

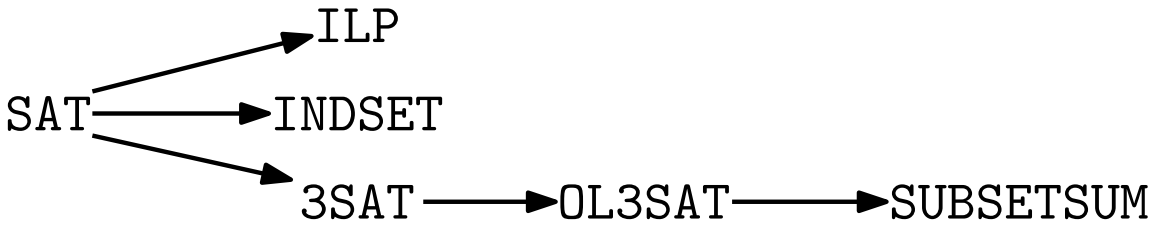


# The Graph of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.
- This being said, defining a reduction from $\mathcal{L}$ to $\mathcal{H}$ is sometimes easy, and sometimes very difficult, and it is instructive to think at **NP**-complete problems as forming a graph, where edges are *natural* polytime reductions
  - A fragment, the one we have encountered so far is the following




  - In fact, this graph is **huge**: thousands of different problems are known to be **NP**-complete

# Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?

# Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.

# Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.
- On the other hand, given that $\mathcal{L}$ is in **NP**, and that `SAT` is **NP**-complete, we know that $\mathcal{L} \leq_p$ `SAT`, i.e. that instances of $\mathcal{L}$ can be efficiently translated into instances of `SAT`.

# Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.
- On the other hand, given that $\mathcal{L}$ is in **NP**, and that `SAT` is **NP**-complete, we know that $\mathcal{L} \leq_p$ `SAT`, i.e. that instances of $\mathcal{L}$ can be efficiently translated into instances of `SAT`.
- This is often **very useful**, because specialised tools for `SAT`, called SAT-solvers do exist.
  - They do not work in polynomial time.
  - Concretely, they work extremely well on a relatively large class of formulas.

# Thank You!

Questions?

# Languages and Algorithms for Artificial Intelligence (Third Module)

## A Glimpse into Computational Learning Theory

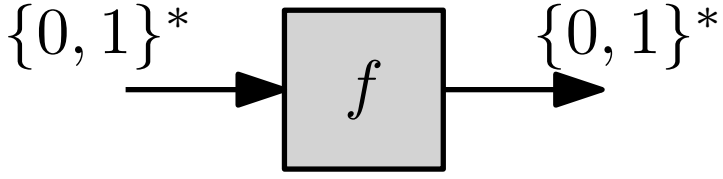
Ugo Dal Lago

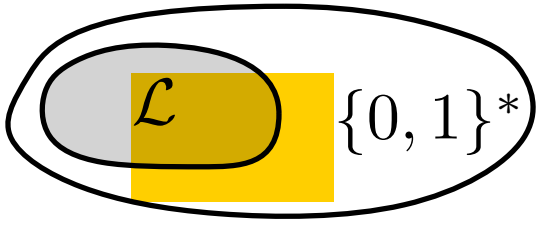


University of Bologna, Academic Year 2019/2020

# Learning Problems as Computational Problems

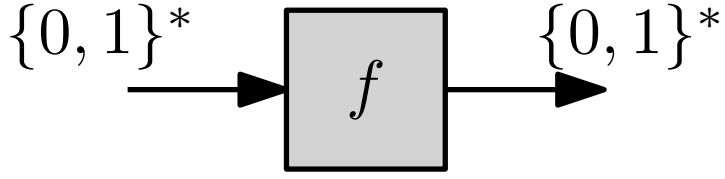
- We have so far taken functions and languages as our notions of **computational tasks**:

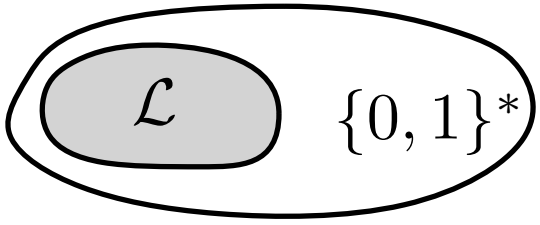


# Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:

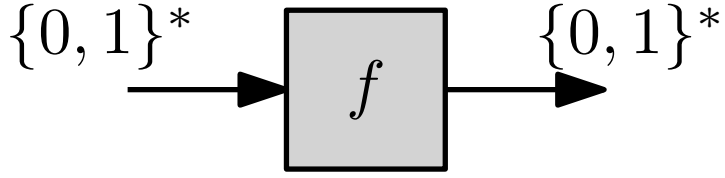


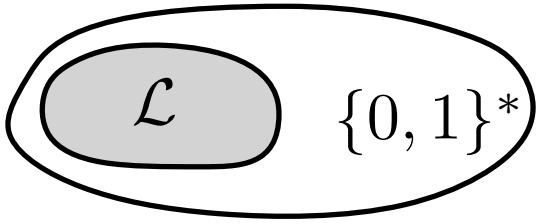




- Is it that **learning problemes**, among the most crucial tasks in AI, can be seen as computational problems?

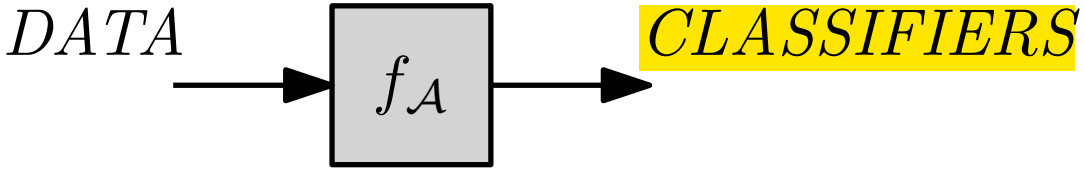
# Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:



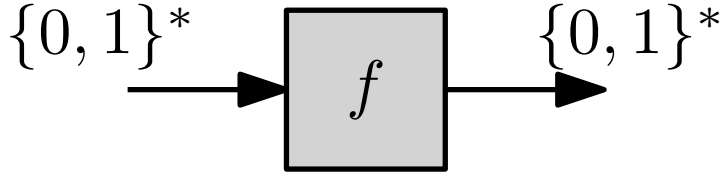



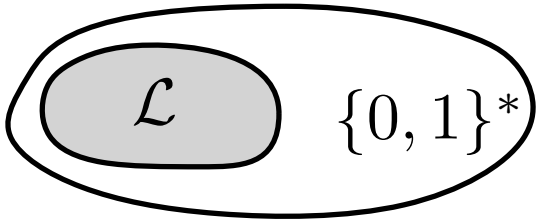

- Is it that **learning problemes**, among the most crucial tasks in AI, can be seen as computational problems?
- The answer is positive: any learning algorithm $\mathcal{A}$ actually computes a function $f_{\mathcal{A}}$ whose input is a finite sequence of *labelled data* and whose output can be seen as a *classifier*:



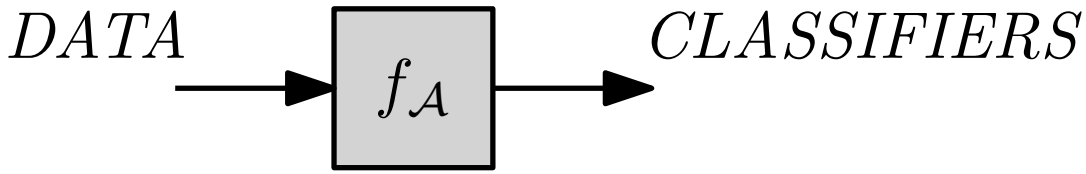
# Learning Problems as Computational Problems

- We have so far taken functions and languages as our notions of **computational tasks**:



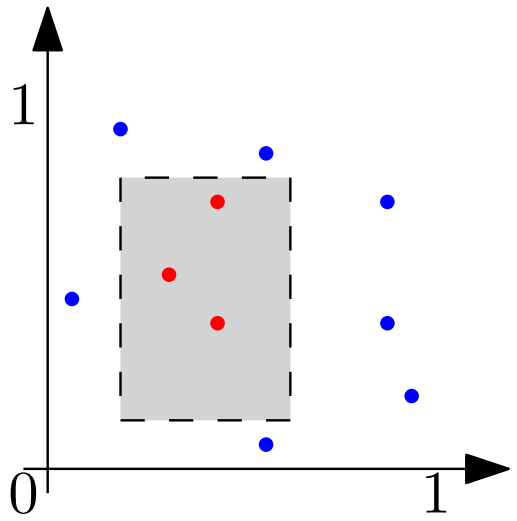

- Is it that **learning problemes**, among the most crucial tasks in AI, can be seen as computational problems?
- The answer is positive: any learning algorithm $\mathcal{A}$ actually computes a function $f_{\mathcal{A}}$ whose input is a finite sequence of *labelled data* and whose output can be seen as a *classifier*:




  - Could data and classifiers be encoded as strings, thus turning $f_{\mathcal{A}}$ as a function of the kind we know?
  - How could we formalize the fact that $\mathcal{A}$ correctly solves a given learning task?

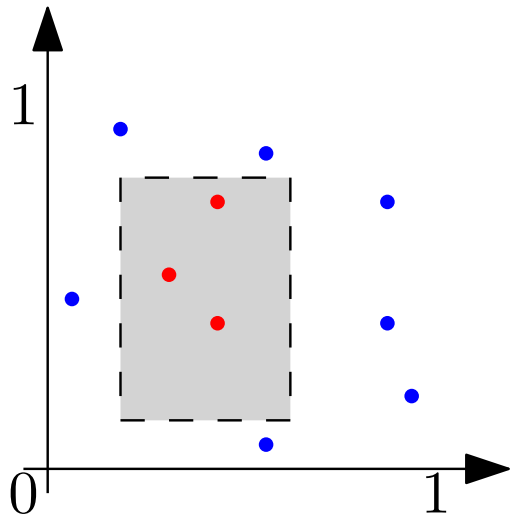
# An Example Problem

Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle



# An Example Problem

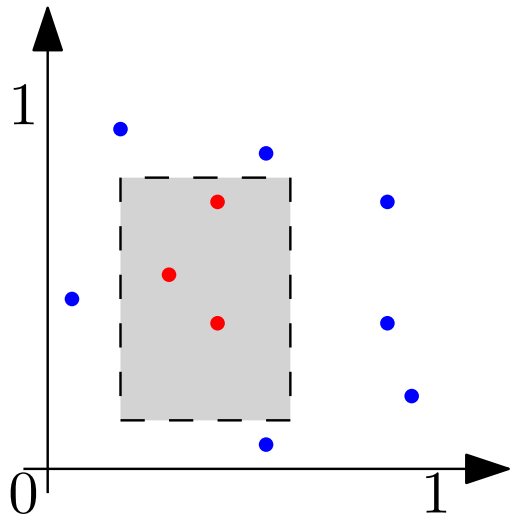
Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle




- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.

# An Example Problem

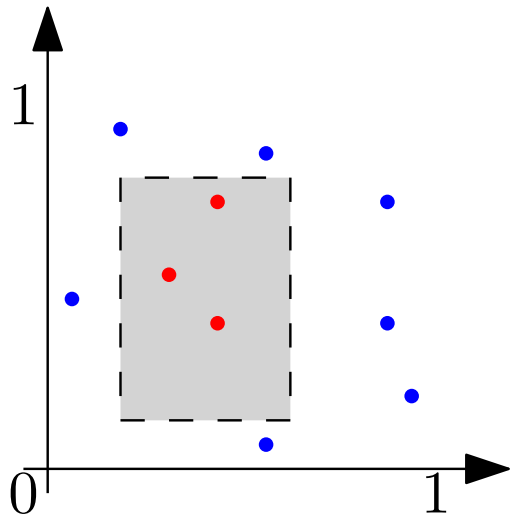
Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle




- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.
- It knows that the data are labelled according to a rectangle, $R$ but it does not know *which rectangle* is being used.

# An Example Problem

Suppose that the data the algorithm $\mathcal{A}$ takes in input are points $(x, y) \in \mathbb{R}_{[0,1]} \times \mathbb{R}_{[0,1]}$ (where $\mathbb{R}_{[0,1]}$ is the set of real numbers between 0 and 1). These are labelled as positive or negative depending on they being inside a rectangle




- The algorithm $\mathcal{A}$ should be able to guess a classifier, namely a *rectangle*, based on the labelled data it received in input.
- It knows that the data are labelled according to a rectangle, $R$ but it does not know *which rectangle* is being used.
- The algorithm $\mathcal{A}$ cannot guess the rectangle $R$ with perfect accuracy if the data it receives in input are too few. As the data in $D$ grow in number, we would expect the rectangle $f_{\mathcal{A}}(D)$ to converge to $R$, wouldn't we?

# The Rules of the Game

- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
  - But it *does not know* which one!

# The Rules of the Game

- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
  - But it *does not know* which one!
- $\mathcal{A}$ *does not know* the distribution $\mathbf{D}$ from which the points $(x, y)$ are drawn.
  - It is supposed to "do the job" for each possible distribution $\mathbf{D}$.

# The Rules of the Game

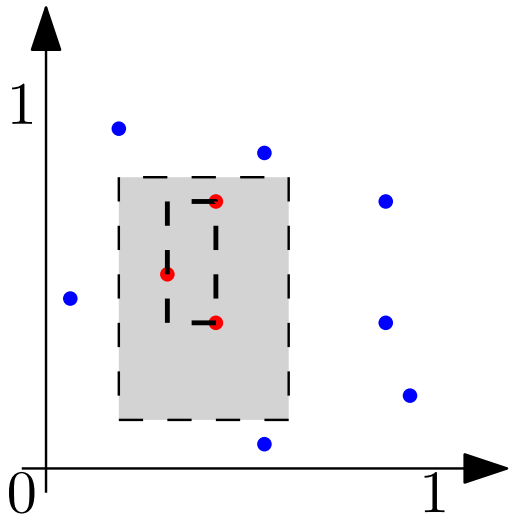
- Again, $\mathcal{A}$ *knows* that the the way input data are labelled is by way of a rectangle (whose sides are parallel to the axes).
  - But it *does not know* which one!
- $\mathcal{A}$ *does not know* the distribution $\mathbf{D}$ from which the points $(x, y)$ are drawn.
  - It is supposed to "do the job" for each possible distribution $\mathbf{D}$.
- $\mathcal{A}$ is an ordinary algorithm.
  - Ultimately, it can be seen as a TM.
  - We thus assume that real numbers can be appropriately approximated as binary strings.
  - In some cases, it is useful to assume $\mathcal{A}$ to have the possibility to "flip a coin", i.e., to be a randomized algorithm.

# The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle $R$ including all the positive instances;
  3. Return $R$.

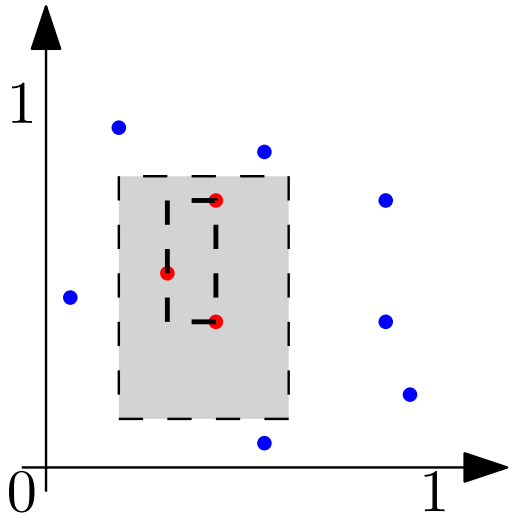
# The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle $R$ including all the positive instances;
  3. Return $R$.
- In the probelm instance from the previous slides, one would get, when running $\mathcal{A}_{\mathsf{BFP}}$, the little bold rectangle. Of course, the result is always a sub-rectangle of the target rectangle.



# The Algorithm $\mathcal{A}_{\mathsf{BFP}}$

- We could define an Algorithm $\mathcal{A}_{\mathsf{BFP}}$ as follows:
  1. Given the data $((x_1, y_1), p_1), \ldots, ((x_n, y_n), p_n)$;
  2. Determine the smallest rectangle $R$ including all the positive instances;
  3. Return $R$.
- In the probelm instance from the previous slides, one would get, when running $\mathcal{A}_{\mathsf{BFP}}$, the little bold rectangle. Of course, the result is always a sub-rectangle of the target rectangle.




- The output classifier is a rectangle, which can be easily represented as a pair of coordinates.

# Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?

# Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.

# Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.
- For a given rectangle $R$ and a target rectangle $T$, the *probability of error* in using $R$ as a replacement of $T$ (when the distribution is $\mathbf{D}$) is
  $error_{\mathbf{D},T}(R) = \Pr_{x \sim \mathbf{D}}[x \in (R - T) \cup (T - R)]$.

# Is $\mathcal{A}_{\mathsf{BFP}}$ (Approximately) Correct?

- The output of $\mathcal{A}_{\mathsf{BFP}}$ can be very different from the target rectangle.
  - Is the algorithm balantantly incorrect, then?
- The answer is **negative**.
- For a given rectangle $R$ and a target rectangle $T$, the *probability of error* in using $R$ as a replacement of $T$ (when the distribution is $\mathbf{D}$) is
  $error_{\mathbf{D},T}(R) = \Pr_{x \sim \mathbf{D}}[x \in (R - T) \cup (T - R)]$.
- As the number of samples in $D$ grows, the result $\mathcal{A}_{\mathsf{BFP}}(D)$ does *not* necessarily approach the target rectangle, but its probability of error approaches zero.

**Theorem**

*For every distribution* $\mathbf{D}$, *for every* $0 < \varepsilon < \frac{1}{2}$ *and for every* $0 < \delta < \frac{1}{2}$, *if* $m \geq \frac{4}{\varepsilon} \ln\left(\frac{4}{\delta}\right)$, *then*

$$\Pr_{D \sim \mathbf{D}^m}[error_{\mathbf{D},T}(\mathcal{A}_{\mathsf{BFP}}(T(D)) < \varepsilon] > 1 - \delta$$

# The General Model — Terminology

- We assume to work within an **instance space** $X$.
  - $X$ is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.

# The General Model — Terminology

- We assume to work within an **instance space** $X$.
  - $X$ is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.
- **Concepts** are subsets of $X$, i.e. collections of objects. These should be thought of as properties of objects.
  - In the example, concepts are arbitrary subsets of $X = \mathbb{R}^2_{[0,1]}$, i.e. arbitrary regions within $\mathbb{R}^2_{[0,1]}$.

# The General Model — Terminology

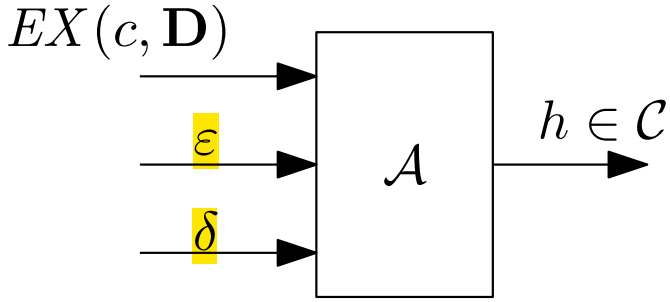
- We assume to work within an **instance space** $X$.
  - $X$ is the set of (encodings) of instances of objects the learner wants to classify.
  - Data from the instance spaces are generated through a distribution $\mathbf{D}$, unknown to the learner.
  - In the example, $X = \mathbb{R}^2_{[0,1]}$.
- **Concepts** are subsets of $X$, i.e. collections of objects. These should be thought of as properties of objects.
  - In the example, concepts are arbitrary subsets of $X = \mathbb{R}^2_{[0,1]}$, i.e. arbitrary regions within $\mathbb{R}^2_{[0,1]}$.
- A **concept class** $\mathcal{C}$ is a collection of concepts, namely a subset of $\mathcal{P}(X)$. These are the concepts which are sufficiently simple to describe, and that algorithms can handle.
  - The concept class $\mathcal{C}$ we work with in the example is the one of rectangles whose sides are parallel to the axes.
  - The **target concept** $c \in \mathcal{C}$ is the concept the learner wants to build a classifier for.

# The General Model — The Learning Algorithm $\mathcal{A}$

- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it does not know the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.

# The General Model — The Learning Algorithm $\mathcal{A}$

- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it does not know the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.
- The interface of any learning algorithm $\mathcal{A}$ can be described as follows:

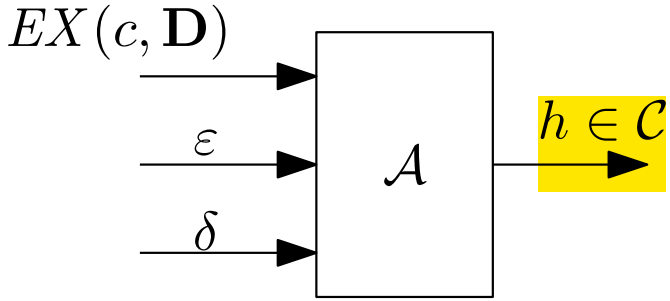



where:

- $\varepsilon$ is **error parameter**, while $\delta$ is the **confidence parameter**;
- $EX(c, \mathbf{D})$ should be though as an *oracle*, a procedure that $\mathcal{A}$ can call as many times she wants, and which returns an element $x \sim \mathbf{D}$ from $X$, labelled according to whether it is in $c$ or not.

# The General Model — The Learning Algorithm $\mathcal{A}$

- Every learning algorithm is designed to learn concepts from a concept class $\mathcal{C}$ but it does not know the target concept $c \in \mathcal{C}$, nor the associated distribution $\mathbf{D}$.
- The interface of any learning algorithm $\mathcal{A}$ can be described as follows:




where:

  - $\varepsilon$ is **error parameter**, while $\delta$ is the **confidence parameter**;
  - $EX(c, \mathbf{D})$ should be though as an *oracle*, a procedure that $\mathcal{A}$ can call as many times she wants, and which returns an element $x \sim \mathbf{D}$ from $X$, labelled according to whether it is in $c$ or not.

- The **error** of any $h \in \mathcal{C}$ is defined as $error_{\mathbf{D},c} = \Pr_{x \sim \mathbf{D}}[h(x) \neq c(x)]$.

# The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space $X$. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, then

$$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c,\mathbf{D}),\varepsilon,\delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c,\mathbf{D})$

# The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space $X$. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, then

$$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c,\mathbf{D}),\varepsilon,\delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c,\mathbf{D})$
- If the time complexity of $\mathcal{A}$ is bounded by a polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$, we say that $\mathcal{C}$ is **efficiently PAC learnable**.
  - The complexity of $\mathcal{A}$ is measured taking into account the number of calls to $EX(c,\mathbf{D})$.

# The General Model — PAC Concept Classes

- Let $\mathcal{C}$ be a concept class over the instance space $X$. We say that $\mathcal{C}$ is **PAC learnable** iff there is an algorithm $\mathcal{A}$ such that for every $c \in \mathcal{C}$, for every distribution $\mathbf{D}$, for every $0 < \varepsilon < \frac{1}{2}$ and for every $0 < \delta < \frac{1}{2}$, then

$$\Pr[error_{\mathbf{D},c}(\mathcal{A}(EX(c,\mathbf{D}),\varepsilon,\delta)) < \varepsilon] > 1 - \delta$$

  where the probability is taken over the calls to $EX(c,\mathbf{D})$
- If the time complexity of $\mathcal{A}$ is bounded by a polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$, we say that $\mathcal{C}$ is **efficiently PAC learnable**.
  - The complexity of $\mathcal{A}$ is measured taking into account the number of calls to $EX(c,\mathbf{D})$.

## Corollary

*The concept-class of axis-aligned rectangles over $\mathbb{R}^2_{[0,1]}$ is efficiently PAC-learnable.*

# Thank You!

Questions?

# Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on $c$.

# Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on $c$.
  - We assume concepts in $\mathcal{C}$ can be represented by way of binary strings, and each concept $e \in \mathcal{C}$ requires $size(e)$ bits. We talk of a **representation class**.

# Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on $c$.
  - We assume concepts in $\mathcal{C}$ can be represented by way of binary strings, and each concept $e \in \mathcal{C}$ requires $size(e)$ bits. We talk of a **representation class**.
- Examples
  - $X_n$ could be $\{0,1\}^n$, the set of **boolean vectors** of of (fixed!) length $n$, and $\mathcal{C}_n$ is the set of all subsets of $\{0,1\}^n$ *represented by CNFs.*
  - $X_n$ could rather be $\mathbb{R}^n$, the set of **vectors of real numbers** of length $n$, while $\mathcal{C}_n$ are say, the subsets of $\mathbb{R}^n$ *represented by some form of neural network* with $n$ inputs and 1 output.

# Representation Classes

- In our definition of efficient PAC learning, the algorithm $\mathcal{A}$, having no access to the target concept $c \in \mathcal{C}$, must work in polynomial time **independently** on $c$.
  - We assume concepts in $\mathcal{C}$ can be represented by way of binary strings, and each concept $e \in \mathcal{C}$ requires $size(e)$ bits. We talk of a **representation class**.
- Examples
  - $X_n$ could be $\{0,1\}^n$, the set of **boolean vectors** of of (fixed!) length $n$, and $\mathcal{C}_n$ is the set of all subsets of $\{0,1\}^n$ *represented by CNFs*.
  - $X_n$ could rather be $\mathbb{R}^n$, the set of **vectors of real numbers** of length $n$, while $\mathcal{C}_n$ are say, the subsets of $\mathbb{R}^n$ *represented by some form of neural network* with $n$ inputs and 1 output.
- In many cases (e.g. SGD), one has a *single* learning algorithm that work for every value of $n$. In that case, we allow (in the definition of efficient PAC learning) the algorithm $\mathcal{A}$ to take time polynomial in $n$, $size(c)$, $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$.

# Boolean Functions as a Representation Class

- Suppose your instance class is $X = \cup_{n \in \mathbb{N}} X_n$ where $X_n = \{0,1\}^n$.

# Boolean Functions as a Representation Class

- Suppose your instance class is $X = \cup_{n \in \mathbb{N}} X_n$ where $X_n = \{0,1\}^n$.
- One **first example** of a representation class for $X_n$ is the class $\mathbf{CL}_n$ of all *conjunctions of literals* on the variables $x_1, \ldots, x_n$.
  - As an example, the conjunction

    $$x_1 \wedge \neg x_2 \wedge x_4,$$

    defines a subset of $\{0,1\}^4$.
  - *Not all* subsets of $\{0,1\}^n$ can be captured.

# Boolean Functions as a Representation Class

- Suppose your instance class is $X = \cup_{n \in \mathbb{N}} X_n$ where $X_n = \{0,1\}^n$.
- One **first example** of a representation class for $X_n$ is the class $\mathbf{CL}_n$ of all *conjunctions of literals* on the variables $x_1, \ldots, x_n$.
  - As an example, the conjunction

    $$x_1 \wedge \neg x_2 \wedge x_4,$$

    defines a subset of $\{0,1\}^4$.
  - *Not all* subsets of $\{0,1\}^n$ can be captured.
- A **second example** of a representation class for $X$ is a class we know, namely the class $\mathbf{CNF}_n$ of CNFs over $x_1, \ldots, x_n$, which are conjunction *of disjunctions* of literals.
  - CNFs are normal forms of any boolean functions.
  - *All* subsets of $\{0,1\}^*$ can be captured this way.
  - We could even consider $k\mathbf{CNF}_n$ rather than arbitrary one, but this way we would lose universality.

# Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals $c$ on $n$ variables $x_1, \ldots, x_n$. How could a learning algorithm proceed?

# Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals $c$ on $n$ variables $x_1, \ldots, x_n$. How could a learning algorithm proceed?
- Data are in the form $(s, b)$ where $s \in \{0,1\}^n$ and $b \in \{0,1\}$. The latter is a label telling us whether $s \in c$ or $s \notin c$.
- A learning algorithm could proceed by keeping a conjunction of literals $h$ as its state, initially set to

$$x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge \cdots \wedge x_n \wedge \neg x_n.$$

  and updating it according to positive data (while negative data are discarded).
  - If $n = 3$, the current state of $h$ is $x_1 \wedge x_2 \wedge \neg x_2 \wedge \neg x_3$ and we receive $(101, 1)$, the hypothesis $h$ is updated as $x_1 \wedge \neg x_2$.

# Learning Conjuctions of Literals

- Suppose your target concept is a conjunction of literals $c$ on $n$ variables $x_1, \ldots, x_n$. How could a learning algorithm proceed?
- Data are in the form $(s, b)$ where $s \in \{0,1\}^n$ and $b \in \{0,1\}$. The latter is a label telling us whether $s \in c$ or $s \notin c$.
- A learning algorithm could proceed by keeping a conjunction of literals $h$ as its state, initially set to

$$x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge \cdots \wedge x_n \wedge \neg x_n.$$

  and updating it according to positive data (while negative data are discarded).
  - If $n = 3$, the current state of $h$ is $x_1 \wedge x_2 \wedge \neg x_2 \wedge \neg x_3$ and we receive $(101, 1)$, the hypothesis $h$ is updated as $x_1 \wedge \neg x_2$.

**Theorem**

*The representation class of boolean conjuctions of literals is efficiently PAC-learnable.*

# Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.

# Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.
- Let us take a look at a *slight generalization* of conjunctions of literals as a representation class.
  - A **3-term DNF formula** over $n$ bits is a propositional formula in the form $T_1 \vee T_2 \vee T_3$, where each $T_i$ is a conjunction of literals over $x_1, \ldots, x_n$.
  - In a sense, this class is the *dual* to 3CNFs!
  - As such, it is more expressive than conjunctions of literals, but still not universal.

# Intractability of Learning DNFs

- We know that conjunctions of literals are efficiently learnable. But they are highly incomplete as a way to represent boolean functions.
- Let us take a look at a *slight generalization* of conjunctions of literals as a representation class.
  - A **3-term DNF formula** over $n$ bits is a propositional formula in the form $T_1 \vee T_2 \vee T_3$, where each $T_i$ is a conjunction of literals over $x_1, \ldots, x_n$.
  - In a sense, this class is the *dual* to 3CNFs!
  - As such, it is more expressive than conjunctions of literals, but still not universal.

**Theorem**

*If* **NP** $\neq$ **RP**, *then the representation class of 3-term DNF formulas is not efficiently PAC learnable.*

# Is This the End of the Story?

# Is This the End of the Story?

- **Definitely No!** Actually, we have just *scratched the surface* of computational learning theory.

# Is This the End of the Story?

- **Definitely No!** Actually, we have just *scratched the surface* of computational learning theory.
- Models and results we did not have time to talk about include:
  - The VC Dimension.
  - The Fundamental Theorem of Learning.
  - The No-Free-Lunch Theorem.
  - Occam's Razor.
  - Positive and negative results about neural networks.
  - ...
- More information can be found in of the many excellent books on CLT, e.g.
  - Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar *Foundations of Machine Learning* Second Edition. The MIT Press. 2018
  - Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: from Theory to Algorithms* Cambridge University Press. 2014.
  - Michael Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory* The MIT Press. 1994.

# Example Results about Neural Networks (from Kearns and Vazirani's Book)

**Theorem 3.7** *Let $G$ be any directed acyclic graph, and let $C_G$ be the class of neural networks on an architecture $G$ with indegree $r$ and $s$ internal nodes. Then the number of examples required to learn $C_G$ is*

$$O\left(\frac{1}{\epsilon}\log\frac{1}{\delta}+\frac{(rs+s)\log s}{\epsilon}\log\frac{1}{\epsilon}\right).$$

# Example Results about Neural Networks (from Kearns and Vazirani's Book)

**Theorem 3.7** *Let $G$ be any directed acyclic graph, and let $C_G$ be the class of neural networks on an architecture $G$ with indegree $r$ and $s$ internal nodes. Then the number of examples required to learn $C_G$ is*

$$O\left(\frac{1}{\epsilon}\log\frac{1}{\delta}+\frac{(rs+s)\log s}{\epsilon}\log\frac{1}{\epsilon}\right).$$

**Theorem 6.6** *Under the Discrete Cube Root Assumption, there is fixed polynomial $p(\cdot)$ and an infinite family of directed acyclic graphs (architectures) $G=\{G_{n^2}\}_{n\geq 1}$ such that each $G_{n^2}$ has $n^2$ boolean inputs and at most $p(n)$ nodes, the depth of $G_{n^2}$ is a fixed constant independent of $n$, but the representation class $C_G=\cup_{n\geq 1}C_{G_{n^2}}$ (where $C_{G_{n^2}}$ is the class of all neural networks over $\Re^{n^2}$ with underlying architecture $G_{n^2}$) is not efficiently PAC learnable (using any polynomially evaluatable hypothesis class). This holds even if we restrict the networks in $C_{G_{n^2}}$ to have only binary weights.*

# Thank You!

Questions?