Part 01 - Computability and Complexity Theory

Course on the negative side of AI: because nobody talks about the dark side → is there any intrinsic limit to what we can do with AI and ML, or to the accuracy or efficiency of algorithms solving a given problem?

Historical, Conceptual and Mathematical Preliminaries

@February 27, 2023

A Bit of History

The notion of computation has existed for thousands of years, which is prior to the advent of computers. Euclid's algorithm for computing the maximum common divisor between two numbers is an effective computational model and it's about 2300 yo.
Before the 20th century, Computation as the process of producing outputs from inputs hadn't a formal scientific treatment: you could stumble on a random mathematician working on it, but there was no formal method. Wasn't consistent or adequately precise: it was seen more as a form of art, rather than a scientific concept.
The modern Theory of Computation (from now on ToC) starts last century (late Twenties, beginning of the Thirties). It's the Golden Age of Computer Science. Finally a new definition was formalized, together with many results about it. and all the definition proposals were essentially equivalent (which is nice…). From the Fifties-Sixties the ToC has evolved into a fully fledged scientific field. Its outfall is twofold:

- to the other Sciences, for example Biology (e.g. recognizing hardness in sequencing problems), Physics (Quantum Computation)…
- to ICT → the theoretical notions of computation were already there when the physical computer was born, so it influenced design.

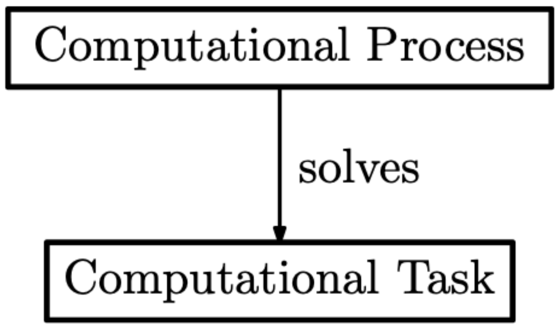
In the last thirty years one can see that the concepts from ToC have gained more and more influence on the other sciences. Also, problems from ToC are considered interesting and worth being solved by from a broader point of view (mainly research in Mathematics and Physics)

@February 28, 2023

At the beginning (Turing, Church,…) ToC was concerned with Computability (whether a given meaningful problem is computable or no - if a problem which in principle could be solvable by computers is in fact solvable). If a task is not solvable by computer, it is deemed uncomputable. ToC also tries to study how computable a problem is (hierarchy).

From the Sixties-Seventies (Hartmanis, Stearns, Cobham), ToC can be also seen as a way to study efficiency (is a certain task solvable in a reasonable amount of time/space?). Definition of what a solvable problem is slightly changes: it exists an algorithm which is efficient in some sense. This is what is called Computational Complexity Theory. It is concerned with classifying problems w.r.t. amount of resources spent.

Modeling Computation



Dichotomy between Processes and Tasks:

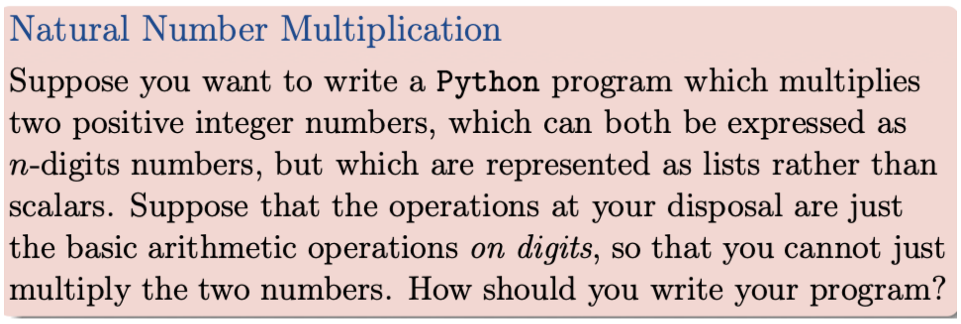
- Task → is the problem, the objective, what we're trying to solve.
- Process → tool you use to solve the task (series of steps). Whatever you do to solve the task

Meaningful → conceivable in principle to be solvable by computer

A computational process can solve only one task, but a task can be solved by different computational processes (it's not a 1:1 relation - you can modify programs to be equivalent using meaningless instructions). It can happen that the task is unsolvable.

Example 1: The Multiplication Problem

The task is:



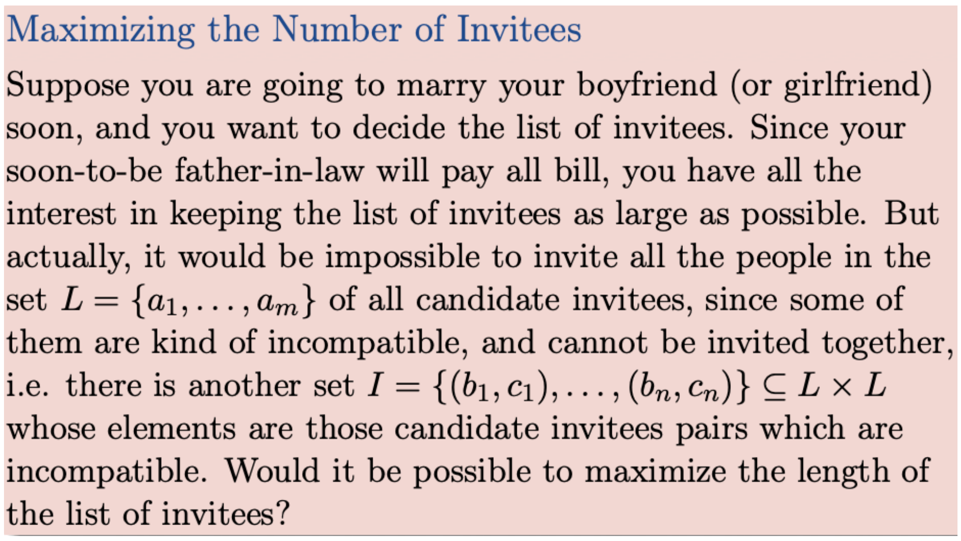
Natural Number Multiplication

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?

One process that solves the task could be what we will call `RepeatedAddition`: we reduce multiplication to addition (addition of two n-digit numbers can be carried out in a linear amount of steps, the total being $b \cdot n$. Another way could be multiplying a and b with the direct algorithm learned in elementary school, so called `GridMethod`, that takes an amount of steps proportional to $n \cdot n$.

The first process is simply impractical independently of the hardware. Indeed, b can be exponential in n (i.e. at most $10^n - 1$). Which means that if $n = 100$, for example, and each basic instruction takes a millisecond, `GridMethod` takes one second, `RepeatedAddition`… years (and a lot of them). This means that I have distinct processes that can solve the same task in many different ways, not all of them acceptable. `GridMethod` is a witness of the fact that this task is in a class P (Poly-time Computable Problems)

Example 2: The Wedding Problem



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 $L = \{a_1, \dots, a_m\}$ of all candidate invitees, since some of them are kind of incompatible, and cannot be invited together, i.e. there is another set $I = \{(b_1, c_1), \dots, (b_n, c_n)\} \subseteq L \times L$ whose elements are those candidate invitees pairs which are incompatible. Would it be possible to maximize the length of the list of invitees?

The "easy way" to solve this problem (and is quite natural as a solution) is to check all the possible subsets of L that respects the constraints of the problem. Sooner or later you will find the best one. The point is that the number n of elements L determines the number of subsets (which is 2^n - out of reach). The algorithm is a witness to the fact that the problem belongs to the class NP (Non-Polynomial Time Computable Problems).

The question is now: is there a more clever algorithm? We don't know. Working on this topic (which you can spend a whole career on and get a million dollars for solving it) would lead to the answer to the question of whether NP is equal to P.

What is a Process → in ToC a process is an algorithm. Giving an actual precise definition of **algorithm** is not possible now, but now we can say it is a process of computational nature that satisfies:

- finite description of a series of computational steps → must be finitely representable; each step must be of a computational nature
- each step must be elementary (atomic operations) → we want to be able to account for efficiency, because having non-elementary steps could hide complexity inside of them (on the surface, you have a much simpler problem than it actually is).
- the way the next step is determined must be deterministic → there is just one way of resolving the choice, without ambiguities

The code we write satisfy 1. and 3., but not necessarily (it is a bit opaque) the 2. → the description we write is high-level, and the resources are hard to estimate. You would be forced to work with a small subset of Python to respect all the requisites.

We will use a different, more abstract and low level definition

Our objective is to try to prove that a certain task that we're interested about is not solvable by any processes beyond a certain level of efficiency. It can become extremely difficult as a task. We can't proceed by exhaustive enumeration (I don't have upper bounds). Computational Complexity is rather young. So we're not actually in a position to prove the non-existence of efficient algorithms (this is our holy grail). Up to now, we don't actually classify problems, but we interrelate different problems (e.g. one problem is harder than another). It is a young field, where many crucial problems remain still unsolved. Cryptography is actually a subfield of Complexity Theory as it is based on the hardness of certain tasks.

Some Mathematical Preliminaries

We work with Set Theory.

Sets and Numbers

The cardinality (number of elements) of any set X is indicated as $|X|$. It can be finite or infinite. There are different cardinalities for different infinite sets (e.g. $|\mathbb{N}| < |\mathbb{R}|$). Indeed, we will mainly work with discrete numbers. We will work with numerical sets in $\mathbb{N}$ (Natural Numbers) or $\mathbb{Z}$ (Integer Numbers).

(Used Extensively) We say that a condition (predicate) $P(n)$ depending on $n \in \mathbb{N}$ holds for sufficiently large n if there is $N \in \mathbb{N}$ such that $P(n)$ holds for any $n > N$

Common notation:

- For a given real number x, $\lceil x \rceil$ is the smallest element of $\mathbb{Z}$ s.t. $\lceil x \rceil \geq x$. Whenever a real number x is used in place of a natural number, we implicitly read it as $\lceil x \rceil$
- For a natural number n, $[n]$ is the set $\{1, ..., n\}$

- Peculiar use of the logarithm function: $\log x$ stands for the base 2 logarithm.

Strings

Strict meaning: mathematical concept. If S is a finite set and you consider it as your alphabet, then a string over the alphabet S is a finite, ordered, possibly empty, tuple of elements from S. Most often, the alphabet S is the set $\{0,1\}$

The set of all strings over S of length exactly $n \in \mathbb{N}$ is indicated as S^n, with S^0 being the set containing only the empty string. The empty string is indicated as ε.

The set of all strings over S is $\bigcup_{n=0}^{\infty} S^n$ and is indicated as S^* (arbitrary but finite concatenation of objects).

The operation of concatenation of two strings x and y over S is xy with nothing in between (might find a $\cdot$ sometimes). If the string is concatenated with itself $k \in \mathbb{N}$ times is x^k. As one can expect, $x^0 = \varepsilon$. This means that the string is a monoid. The length of a string x is indicated $|x|$.

Tasks as Functions

Any tasks of interest consist in computing a function from $\{0,1\}^*$ to itself. They are functions from the binary String. The notion of function is sufficiently general, as most (but not all) tasks consist in turning an input to an output (computing a mere function). A learning model doesn't actually fit the definition.

The emphasis on strings lies in the fact that most discrete sets can be easily represented as strings, following some encoding, that however should be as simple as possible.

Summarizing, 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 sets, which means that we could potentially represent them as strings, sometimes leaving the encoding of A and B into $\{0,1\}^*$ implicit.

Encoding of element $x \in A$ as a string: $\llcorner x \lrcorner$

Languages and Decision Problems

Boolean functions are special kinds of functions in which the possible outcomes are either 0 or 1; more formally, class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ to strings of length exactly one. This might be the case of Binary Classifiers. These can be seen as characteristic functions of languages.

We identify the function f with the subset $\mathcal{L}_f = \{x \in \{0,1\}^* | f(x) = 1\}$

Any subset of S^* is called a language: mathematical formalization of the concept of decision problem. This way, a decision problem for a given language $\mathcal{M}$ 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 constant c such that $f(n) \leq c \cdot g(n)$ for sufficiently large n

The dual concept is $\Omega(g)$ → $f(n) \geq c \cdot g(n)$ for sufficiently large n

$\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.

I can study the relation between two functions f and g by studying $\lim_{n\to\infty} \frac{f(n)}{g(n)}$

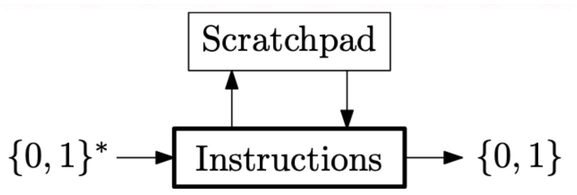
The Computational Model

@March 6, 2023

Giving positive results about the feasibility of a certain computation task is relatively easy: this amounts to implement the algorithm without giving it a formal status and run it on a relatively powerful machine. If this is not there, the negative result is different to obtain: the same method for feasibility is not acceptable. We want our model to be general, universal. We don't want to work with complex models (→ proofs of the negative results would be unmanageable).

Mixed approach: we will use the Turing Machine, and then on top of it, we will introduce pseudocode.

The Model, Informally



For the sake of simplicity, we stick to decision problems

The scratchpad is used to store intermediate results. The arrows are bidirectional in this case. The set of instructions is finite (one of the postulates of algorithms) and should work for any input (dichotomy between the inputs and the set of instructions). The same instruction can be used multiple times. The instructions read the next symbol from the input (they access the input one bit after the other), but they could also access a fixed (in advance) number of symbols from the scratchpad. Based on these inputs, the instructions decide what to do: they can update the scratchpad or, if the machine is “happy” (the computation is over) it stops the computation and outputs either 0 or 1.

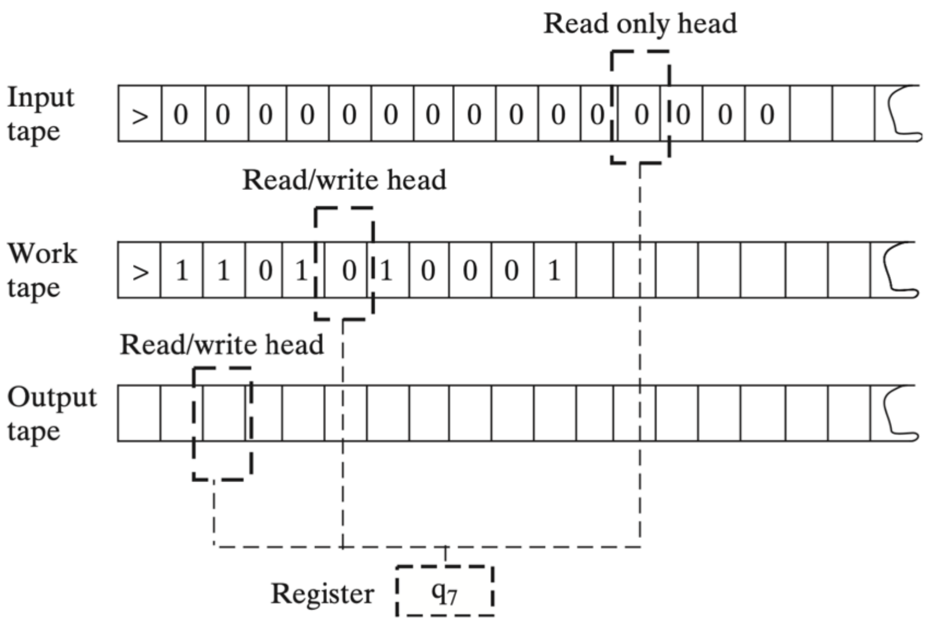
The running time is easy to define: it is the number of steps needed to process the input and solve the problem. Crucial definition: the machine M runs in time $T(n)$ if the time the machine needs to handle input strings of length n is at most $T(n)$. Is the model robust? Yes, it is robust in the sense that the set of problems that can be handled by the model does not change if we variate the definition. The time however changes. All the machines can simulate each other with at most a polynomial overhead in time.

The Turing Machine we have in front of us is completely described by its instructions. The scratchpad is just a memory, it starts empty. Every different input can give rise to a different content for the scratchpad. If you get any string α, we can indicate $\mathcal{M}_\alpha$ as a Turing Machine (every string can describe a Turing Machine). You can define one specific machine (Universal Turing Machine $\mathcal{U}$) that simulates the rest of Turing Machines. It is an algorithm whose purpose is to simulate other programs. It takes in input pairs (x, α) and checks whether α is a legal description of a Turing Machine and if true it simulates the instruction x on $\mathcal{M}_\alpha$. This actually shows us that limits to this model exist. This has ties to the Gödel Incompleteness Theorem (there are formulas in logic which are true but cannot be proven with any formal system).

The Model, Formally

@March 7, 2023

What comes later is just a natural consequence of this lecture.



New Details of the Scratchpad

The scratchpad now consists of k **tapes** (fixed constant), which are infinite one-directional lines of cells, each of which can hold a symbol from a finite alphabet Γ, which will be called the “alphabet of the machine”. Each tape has its own **tape head**, which is a pointer to one of the cells in each tape and indicates what symbols the machine is using at a certain moment. It can read or write one symbol at a time and can move either left or right. Computation happens locally through tapes and heads (basic principles).

- Input Tape → the machine cannot write it, it is supposed to store the input. The machine reads it. The number of input tapes is fixed once and for all once you start dealing with the specific task.
- Output Tape → dual of the Input Tape, it’s written by the machine after computations happen. We can see it as part of the scratchpad and then feed it out. In principle you can have more than one, but practically it is just one. Should contain the result of the computation. It can be actually absent, in this case the role can be filled by the work tape or the register (state) keeps the actual output.
- Work Tape → can be accessed in both ways. Stores the intermediate results
- Register → interface between the scratchpad and the instructions. It contains a state: it tells what is the next task to be executed. The machine is described by the finite set of states Q.

Assumption: the tapes are limited on the left-side ($>$ symbol) but there’s no a priori limit on how far you can go on the right-side. They are potentially infinite, but at any time during the computation only a finite portion is used. They are accessed through heads that can move in more than one direction.

Based on the current state:

1. Read the symbols under the k tape heads and decides what to do
2. For the $k-1$ read-write tapes, can replace the symbol under the head with a new one, or leaving it unchanged
3. It can decide to change the internal state
4. Each computation step is over after deciding how to move (and if move) all the heads.

The way the machine decides how to move the heads, etc… must be very simple, but expressive.

Formal Definition

A Turing Machine (TM) working on k tapes is described as a triple (Γ, Q, δ) containing:

- A finite set Γ of tape symbols, that we assume contains the blank symbol $\square$, the start symbol $\triangleright$, binary digits 0 and 1
- A finite set Q of states which include designated initial (q_{init}) and final (q_{halt}) states
- A transition function $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{L, S, R\}^k$
  with finite domain and codomain. For every state Q and for every k symbols it reads from the tape heads, the machine returns something (states, head tapes modified (I have at least one input tape) and a set of directions (in which the heads will move) in the form {Left, Stable/Still → the head doesn't move, Right}). It's more of an if-else statement.

When the first parameter is q_{halt} then δ cannot touch the state nor the heads and therefore the machine is stuck.

A configuration is not an element from a finite domain, in principle it can be any sequence of values you can put on the tapes. It formally consists of:

- the current state q
- the content of the k tapes
- the positions of the k tape heads

It is indicated with the meta-variable C.

The starting configuration is non-ambiguously defined once the input is fixed. You define it for that specific input. So, formally, the initial configuration for the input $x \in \{0,1\}^*$ is the configuration $\mathcal{I}_x$, in which:

- q_{init} is my current state
- first tape contains $\triangleright x$ followed by blank symbols, while the others $\triangleright$ followed by blank symbols.
- tape heads are positioned on the first symbol of the k tapes

A final configuration for the output $y \in \{0,1\}^*$ is the one where final state is q_{halt} and the encoding is the representation of a string, the content of the output state being $\triangleright y$ followed by blank symbols.

Thanks to configurations, we can express mathematically computations using the transition function. The notion of computability is agnostic to the amount of resources used.
Formally, given a TM $\mathcal{M} = (\Gamma, Q, \delta)$ working on k tapes:

- Given any configuration C, the transition function δ 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 $\mathcal{I}_x \xrightarrow{\delta} C_1 \xrightarrow{\delta} C_2 \xrightarrow{\delta} \dots \xrightarrow{\delta} C_t$, where C_t is the final configuration for y. We write $\mathcal{M}(x)$ for y if this holds.
- 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.
    - Keep in mind that this definition does not put any kind of constraints on the number of steps $\mathcal{M}$ needs to compute $f(x)$ from x.

http://turingmachinesimulator.com

Explicitly constructing TMs is tedious and not the optimal model when you're looking at positive examples. There are other formalisms that are perfectly equivalent to TMs and probably more suited to different cases.

Efficiency and Runtime

Turing Machine has a built in Time indication (function $T : \mathbb{N} \to \mathbb{N}$) and it is tied to the input given to the machine. 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. For every function T there's a number of Algorithms and Languages that can be decided in a time bounded by T

Again, many aspects of the definitions of a TM we're given are arbitrary. You can have many alternative definitions (e.g. the simulator).
In all the cases, there is something that stays invariant. The class of functions you can solve does not depend on these details (number of tapes, length of tapes…). The class of problems solvable in Linear/Quadratic etc… Time is a fragile concept. However, Polynomial Time is extremely robust (reference notion of what can be computed efficiently). All the variations of the notion of TMs can simulate each other with polynomial overhead.

As a consequence of our definition, any TMs is completely determined from the graph of δ. This graph is finite because both the domain and codomain are finite. If you have a parsimonious encoding of δ as a binary string in $\{0,1\}^*$, the encoding $\llcorner \mathcal{M} \lrcorner$ of $\mathcal{M}$ is acceptable provided that:

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

The two conditions are not essential, but here are technically crucial

Theorem (Universal Turing Machine, 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 α. 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 $\mathcal{M}_\alpha$

T is a bound on the runtime of α. We won't see the proof. One has to encode configurations of TMs as strings and prove that $\mathcal{U}$ can simulate $\mathcal{M}_\alpha$ for every α

Theorem (Existence of Uncomputable Functions)
There exists a function $uc : \{0,1\}^* \to \{0,1\}^*$ that is not computable by any TM

@March 13, 2023

Due to Turing. There is a task that no process whatsoever can solve. You can actually prove by "counting argument": the function goes from binary strings to binary strings, a huge set. We, though, don't have many TMs (we have a Natural Number), while the quantity of possible functions is more alike to the Real Numbers.

Here follows the proof, which is constructive. We're considering one specific function. $uc(\alpha)$ (the input is a program). It returns 0 if the TM with index α returns 1 when fed with α itself (Diagonalisation - use the string both as a program and as an input. Concept actually used by Gödel to prove the incompleteness of First Order Arithmetics, and by Bertrand Russel).

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

It's a R.A.A. proof. If uc were computable, we would be able to come up with a machine $\mathcal{M}$ s.t. $\mathcal{M}(\alpha) = uc(\alpha)$. In particular, this would hold also when $\alpha = \llcorner\mathcal{M}\lrcorner$, which is the encoding of $\mathcal{M}$ itself. And here lies the contradiction.

$$uc(\llcorner\mathcal{M}\lrcorner) = 1 \iff \mathcal{M}(\llcorner\mathcal{M}\lrcorner) \neq 1 \iff uc(\llcorner\mathcal{M}\lrcorner) = 0$$

We can in fact build a chain of false double implications, as uc cannot be equal to 0 and to 1 at the same time. The left-hand side implication comes from the definition of uc (the condition is false, otherwise uc would be 0). But $\mathcal{M}$ is a machine computed on uc, so we have contemporarily that uc = 0 (here we exploit the nature of $\mathcal{M}$).

The Halting Problem

We want to write an interpreter that does not compute the program as it is, but determines if the program halts on input.

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

This means we would be able to check for eventual termination of algorithms.

👩🏻‍💻 **Theorem: Uncomputability of halt**

The function halt is not computable by any Turing Machine

This theorem, even more than the previous, is related to Gödel's Incompleteness Theorem.

See Booklet of Exercises for a proof.

Diophantine Equations

Equations are actually computational problems. Diophantine is a polynomial equality with integer coefficients and finitely many unknowns.

MDPR Theorem
The problem of determining whether an arbitrary diophantine equation has a solution is undecidable.

We’re only asking if the solution exists, not which it is. The proof is non-trivial.

Rice’s Theorem

Class of understandability results, along the lines of the halting problem, but more general. Can deal with even infinite classes of languages.

A language $\mathcal{L} \subseteq \{0,1\}^*$ is semantic when all the elements of the set are encodings of Turing Machines and if the encoding of the Turing Machine $\mathcal{M}$ is part of the language and another TM $\mathcal{N}$ computes the same function, also the encoding of $\mathcal{N}$ is part of the language.

This definition captures the extensional properties of programs (e.g. the set of all machines which compute a certain function, the set of all terminating programs, the set of all programs which never output a certain bad string…). All verification problems are bound to be undecidable.

Rice’s Theorem
Any decidable language $\mathcal{L}$ which respects functions (i.e. is semantic) is trivial, i.e. either $\mathcal{L} = \emptyset$ or $\mathcal{L} = \{0,1\}^*$

This means you cannot check it (and usually, you’re not interested). It is undecidable

How to Prove a Problem to be (un)computable

Task we’ve performed actually many times. The hard way to solve the task of proving a problem to be computable is to use a Turing Machine. As soon as the problem starts to be not so easy anymore, the hard way becomes impractical (too complicated: need more tapes, back&forth…). So you can describe in a more informal way an algorithm that describes a function. Designing an algorithm means writing down a series of informal instructions that should be able to solve the problem. If you do so, need to be sure that all steps/instructions perform (are perfect and elementary). You can use other algorithms as subroutines if you know that they solve the sub-problem (the overall task becomes more complicated). Here we look for positive results.

The usual way to prove un-computability of a problem is more difficult than the opposite in CS. We proceed by reduction, so we turn it into an existential statement. We want to prove

that the problem under consideration $\mathcal{L}$ is at least as hard as a problem we know to be undecidable, say $\mathcal{G}$. We define a computable function ϕ so that a string in $\mathcal{G}$ is turned into an instance of $\mathcal{L}$
$s \in \mathcal{G} \iff \phi(s) \in \mathcal{L}$
This way, any hypothetical algorithm for $\mathcal{L}$ that proves solvable would prove the solvability of $\mathcal{G}$, which cannot be possible.

This assumes though, that someone has already proven the un-computability of the function $\mathcal{G}$. Otherwise, you can use Rice's Theorem if the problem falls into particular categories.

Polynomial Time Computable Problems

@March 14, 2023

A complexity class is a set of tasks (not a set of programs) that we can solve (subclass of solvable problems) and solve within a prescribed resource bound. With the complexity classes we try to dig deeper into solvable problems, to find more meaningful subsets. It is not a set of Turing Machines, although it is defined based on Turing Machines. From now on, we will be mainly concern about decision problems (i.e. subsets of$\{0,1\}^*$).

First approach: fix a function $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class $\mathbf{DTIME}(T(n))$ if and only if there is a TM deciding $\mathcal{L}$ and running in a time $n \mapsto c \cdot T(n)$ for some constant c. We fix the function and we define the class around it. Issue (and this is why this approach is not used anymore): this class is not robust and depends on the computational model used $\Rightarrow$ the nature of the class can change, by changing the model. We need stability to build a theory about the intrinsic difficulty of a problem.

"D" stands for deterministic; machines on which the class is based are deterministic automata.

The Class P

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

$\mathbf{P}$ includes all those languages $\mathcal{L}$ which can be decided by a Turing Machine, working in time P, where P is any polynomial. For any polynomial P there are $c, d > 0$ s.t. $P(n) \leq c \cdot n^d$ for sufficiently large n. The c and d can be arbitrarily large (we don't care about the degree of the polynomial)

$\mathbf{P}$ is considered as the class of efficiently decidable languages.

Church-Turing Thesis

Why are we basing complexity theory on Turing Machines?

The Church-Turing Thesis (Postulate)
Every physically realizable computer can be simulated by a Turing Machine with a (possibly very large) overhead in time.

The class of computable problems would not be larger if formalized differently, but it would actually be equal.

The Strong Church-Turing Thesis (Postulate)
Every physically realizable computer can be simulated by a Turing Machine with a polynomial overhead in time (n steps on a computer requires n^c on Turing Machines, where c only depends on the computer) and viceversa.

This means that the class $\mathbf{P}$ is very robust, as it does not depend on the machine. Though this topic is rather controversial, as in Quantum Computing would not hold (in Quantum Computing can apparently theoretically solve in Polynomial Time a problem - factoring of integers - that in Classic Computing can't do - it is not proven unconditionally). Though, there's no evidence of it being false.

Why the class of Polynomials?

- This is the smallest class of functions, which makes $\mathbf{P}$ a robust class.
- It is not so big. Experimentally, once a problem is proved to be solvable in Polynomial time, is very often the case that the degree of polynomials is rather small (though in principle the exponent c bounding the time of any machine deciding $\mathcal{L} \in \mathbf{P}$ can be huge).
- The class is manageable in terms of whomever wants to prove theorems about the class. The class is closed w.r.t. various operations on programs, e.g. composition (intersection, union…) 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.

Criticism:

- The definition of $\mathbf{P}$ is intrinsically based on worst-case complexity (you consider the worse behavior of the machine to determine complexity, not the average behavior). It is good enough even if our problem takes little time on the types of input which arise in practice and not on all of them. Solution: Approximation Algorithms, Average-case Complexity.

- If you change the definition of what your computer can do, also the definition of what's feasible changes. Feasibility can also be defined for classes dealing with arbitrary precision computation, with randomized computation and quantum computation. The class $\mathbf{P}$ can be redefined with other computational models in mind, giving rise to other classes.
- Not all tasks can be modeled as decision problems (complexity theory also deals with functions)

The Class FP

Sometimes, one would like to classify functions rather than languages. We need to slightly generalize.

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 $\mathbf{FP}$ is then defined in a way similar to $\mathbf{P}$:

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

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

Example Problems in $\mathbf{P}$ or $\mathbf{FP}$

- Problems related to Lists (inverting, sorting, finding max or min…)
- Problems related to Graphs (reachability, shortest path, minimum spanning trees…)
- Primality test, exponentiation (counter-intuitive)…
- String matching (also approximate) and other problems related to strings…
- Optimization Problems: LP, maximum cost flow…

How to prove a task being in P or FP

In theory we work the same as deciding if a problem is decidable, one should give a TM working within some polynomial bounds, and prove that the machine decides the language (or computes the function). Can be too bothersome and probably not worth it. Use this method only for decidability. One can also go informal and use pseudocode (similar to python code).

Example: given two strings $x, y \in \{0,1\}^*$ determine if x contains an instance of y. Pseudocode:

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

How can we be sure that the algorithm (which is correct) works in polynomial time?
- Describe the input encoding → The input can be easily encoded as a binary string.
- You have to count the total number of instructions → The total number of instructions is polynomially bounded (linear in the length of x).
- Check that all intermediate variables are under control → All intermediate results are polynomially bounded in length → i cannot be greater than $O(|x|)$, thus its length is $O(\lg |x|)$.
- How much time you need to execute each instruction → 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 $|\llcorner(x, y)\lrcorner|$.

Another example:

```
y <- x
i <- 1
while i < |x| do:
  y <- y * y
  i <- i + 1
end
```

In this case, I don't have intermediate variables under control (y becomes exponentially long). This is not in $\mathbf{P}$.

The Class EXP

@March 20, 2023

What can we find if we try to go beyond the class $\mathbf{P}$, i.e. we allow TMs to have more time at their disposal? (we try to dig into the zoo of complexity classes). 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 $\mathbf{EXP}$ and $\mathbf{FEXP}$ are defined:

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

The 2^{n^c} captures the function multiplication. The tasks in these classes can be solved mechanically, but possibly cannot be solved efficiently. If a problem is in this class, we know that there's an algorithm, a TM that solves it, but it grows real fast and might not possibly be efficiently solvable. The difference with the class $\mathbf{P}$ lies exactly in the potential growth of the problem: $\mathbf{P}$ doesn't grow as much as $\mathbf{EXP}$. It holds that $\mathbf{P} \subseteq \mathbf{EXP}$, $\mathbf{FP} \subseteq \mathbf{FEXP}$. (NB → saying that a program is in $\mathbf{EXP}$ means that there exists a TM that solves it in exponential time, but in the worst case, not always).

Theorem:
The two inclusions above are strict

$\mathbf{EXP}$ and $\mathbf{FEXP}$ are still meaningful as we can find something that it is not in $\mathbf{P}$.

Between the Feasible and the Unfeasible

@March 27, 2023

$\mathbf{P}$ captures the notion of feasible/tractable problem (in a reasonable amount of time). $\mathbf{EXP}$ contains the whole of $\mathbf{P}$ and more (intractable problems). Beyond $\mathbf{EXP}$ it is simply out of reach. In general, we can study the "middle ground" by defining a class $\mathbf{P} \subseteq \mathbf{A} \subseteq \mathbf{EXP}$ (e.g. P-Space, BQP, BPP…). Either one of the inclusions is strict (equality can be on the left or right but not both sides). This way, we can tackle the border between tractable and intractable. Theory of Computation is in its infancy.

But how do we define this class $\mathbf{A}$?. The intuition comes from the dichotomy between creating and verifying/finding. Many problems consist in looking for a solution which can be hard to find, but easier to verify if a guess is correct. We can define the language $\mathcal{L} = \{x \in \{0,1\}^* | \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$ where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings (in itself a language, a language of pairs of strings). The elements 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 $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$. This does not rule that if $\mathcal{A}$ is decidable in poly-time also $\mathcal{L}$ is decidable in polynomial time. Verifying is the easier task of the two. The idea behind the definition is this.

We call y a certificate for x. It's a "piece of paper" telling that the problem x is in $\mathcal{L}$. What if the only thing we know is that $\mathcal{A}$ can be decided in polynomial time? Can we say something about $\mathcal{L}$? No. The search space for certificates is still exponentially huge, so it can be done, but still takes too much time. This is an algorithm, but not the only one to solve the problem (in some cases algorithms could work in polynomial time)

In conclusion, creating is different than verifying.

The Complexity Class $\mathbf{NP}$

N = Non-Deterministic.

The verifier works in polynomial time, while the certificate might not work in polynomial time. We're defining a class based on the verifier.

A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class $\mathbf{NP}$ iff there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ s.t. $\mathcal{L} = \{x \in \{0,1\}^* | \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$

- $\mathcal{M}$ is the verifier for $\mathcal{L}$ and is the only thing we want to check
- Any $y \in \{0,1\}^{p(|x|)}$ s.t. $\mathcal{M}(\llcorner (x, y) \lrcorner) = 1$ is called certificate for x.

Typical example of a class between $\mathbf{P}$ and $\mathbf{EXP}$, without any functional counterpart. It is only a class of languages.

Theorem

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

We cannot establish which one (if not both) equality is strict.

Example Problems in $\mathbf{NP}$

1. **Maximum Independent Set** (abstraction of the Wedding Problem)

   We work with an undirected graph. 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.

The certificate here (the property which makes us sure that the pair is in the language) is W itself. Check whether for every node in W the properties are respected (it is easy).

2. **Subset Sum**

   You have a sequence/list of natural numbers $n_1, ..., n_m$ and you want to check if there is a subset of these elements such that $\sum_{i \in I} n_i = k$. Choose elements s.t. the sum is k.

   This is a problem formulated in Natural Language but this is clear that the certificate here is simply I. Checking whether I is good means computing a sum and compare.

3. **Composite Numbers**

   Given a number $n \in \mathbb{N}$, we want to determine whether n is composite (i.e. not prime - can be obtained as a non-trivial product). The certificate is the factorization of n in the pair (m, l), non-trivial.

4. **Factoring**

   Checking that a number p is prime is not easy, but can be still done in polynomial time.

5. **Decisional Linear Programming**

   LP → Large class of Decision Problems which consists of a sequence of linear inequalities with rational coefficients. The task is to check whether it exists a rational assignments that makes all the inequalities true. Large search space, but the certificate is the assignment, and once you have it, it is easy to establish correctness

6. **Decisional 0/1 Linear Programming**

   Same as before but the assignment is of 0s and 1s.

Not all of these problems are NOT in $\mathbf{P}$. Some problems, like e.g. DLP, can be proved to be in $\mathbf{P}$ (algorithms look for a solution without enumerating all possible certificates, but using a smarter approach, e.g. Ellipsoid Algorithm uses Geometry).
All the others are not known to be in $\mathbf{P}$ (maybe they are, maybe they're not. And the answer's worth money).

Nondeterministic Turing Machines

Nondeterminism is "bad". But can use it as a theoretical tool to deal with situations in which your machine/algorithm is theoretically capable of choosing between more possible routes.

The class $\mathbf{NP}$ can also be defined using a variant of Turing Machines, called Nondeterministic TMs (NDTM). The difference between this and the original TM is that we have two transition functions δ_0 and δ_1 and at any point the machine can decide between picking δ_0 or δ_1 (the machine branches, tree representation); plus, we have a special state

q_{accept}. We will say that the NDTM accepts the input iff there exists at least one branch of the tree (possible evolution of the machine) whose state is q_{accept}. If there's no q_{accept} the input is rejected.

We say that a NDTM 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_{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 s.t. $M(x) = 1$ iff $x \in \mathcal{L}$

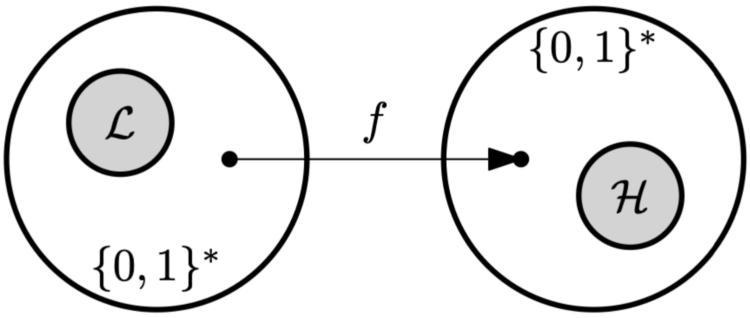
👩🏻‍💻 **Theorem:** (can also be used as a definition)
$\mathbf{NP} = \bigcup_{c \in \mathbb{N}} \mathbf{NDTIME}(n^c)$

It is similar to how we defined $\mathbf{P}$. Useful concept, but not meant to model any form of physically realizable machine.

Are all problems in $\mathbf{NP}$ equivalent?

The Maximum Independent Set is in $\mathbf{NP}$. The Palindrome Words problem is in $\mathbf{P}$ (thus in $\mathbf{NP}$). Intuitively, the inherent difficulty of solving these two problems should be different. We don't know in one case if $\mathbf{P}$ algorithm exists, while in the other case we have security. How to tell tasks apart? If a problem is in $\mathbf{NP}$ up to now we cannot say much. We cannot prove that a problem in $\mathbf{NP}$ is not in $\mathbf{P}$ (would be major breakthrough). We can use another approach, based on problem comparison (study the relative difficulty of deciding problems).

We can use the notion of reduction:



We write a coherent function between $\mathcal{L}$ and $\mathcal{H}$ and we say that $\mathcal{L}$ is polynomial-time reducible to $\mathcal{H}$ if there is a poly-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in \mathcal{L} \iff f(x) \in \mathcal{H}$.

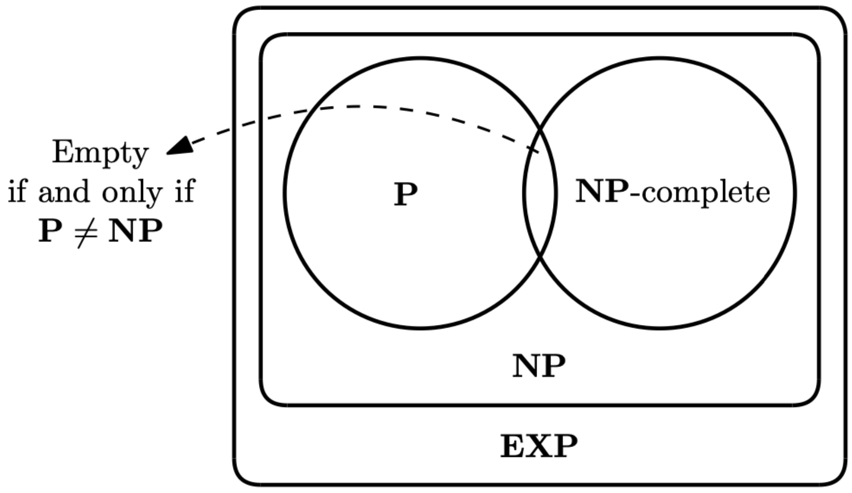
If this is the case $\mathcal{L} \leq_p \mathcal{H}$, which means that $\mathcal{L}$ is less or equal in terms of difficulty than $\mathcal{H}$. Reduction is polynomial-time computable. This holds for classes like $\mathbf{P}$ or above $\mathbf{P}$.

A language $\mathcal{H} \subseteq \{0,1\}^*$ is said to be:

- $\mathbf{NP}$-hard if it is at least as hard as any problem in $\mathbf{NP} \Rightarrow \mathcal{L} \leq_p \mathcal{H} \forall \mathcal{L} \in \mathbf{NP}$
- $\mathbf{NP}$-complete if $\mathcal{H}$ is $\mathbf{NP}$-hard and $\mathcal{H} \in \mathbf{NP}$ (it's the real problem at the "border")

Theorem:

1. The relation $\leq_p$ is a pre-order (i.e. it is reflexive and transitive - can form chains and go from first to last)
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 $\mathcal{L} \in \mathbf{P} \iff \mathbf{P} = \mathbf{NP}$



$\mathbf{NP}$-Complete Problems

@March 30, 2023

The problem we're gonna see has to do with simulation of the TMs. One of the hardest problem in NP is to simulate a TM. The proof of the fact that the problem is NP-Complete is based on the Hierarchy Theorem. The problem is called TMSAT (Turing Machine Satisfiability).

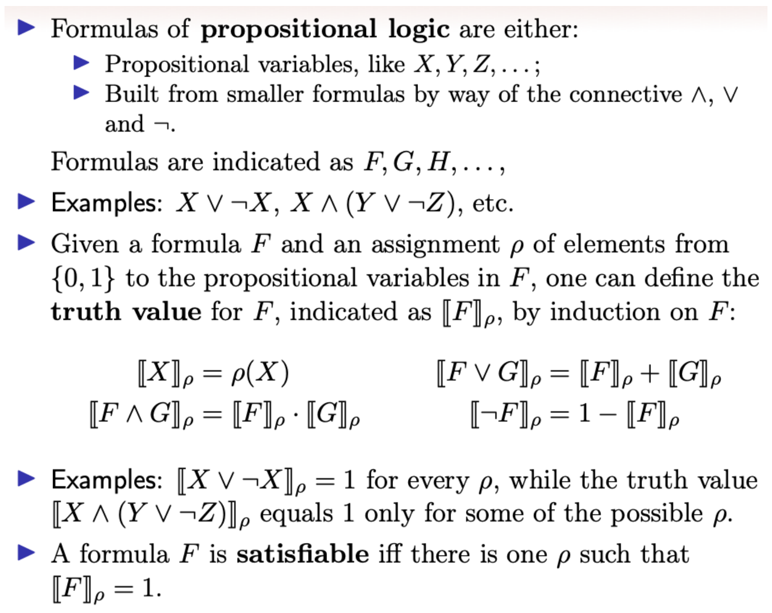
$TMSAT = \{(\alpha, x, 1^n, 1^t) | \exists u \in \{0,1\}^n . \mathcal{M}_\alpha$ outputs 1 on input (x, u) within t steps$\}$

Theorem

TMSAT is NP-Complete

This is interesting only from a merely theoretical perspective. The first practically interesting problem which is NP-Complete actually deals with Propositional Logic

Recap on PL (again, it’s like the third time in a year)



- 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 ρ of elements from $\{0, 1\}$ to the propositional variables in F, one can define the **truth value** for F, indicated as $[\![F]\!]_\rho$, by induction on F:

$$\begin{array}{cc} [\![X]\!]_\rho = \rho(X) & [\![F \vee G]\!]_\rho = [\![F]\!]_\rho + [\![G]\!]_\rho \\ [\![F \wedge G]\!]_\rho = [\![F]\!]_\rho \cdot [\![G]\!]_\rho & [\![\neg F]\!]_\rho = 1 - [\![F]\!]_\rho \end{array}$$

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

A propositional formula F is said to be in Conjunctive Normal Form (CNF) when it is a conjunction of disjunctions of literals (a literal being a variable or its negation). In general, we have three layers: a layer of conjunctions, a layer of disjunctions and a layer of literals. The disjunctions in a CNF are said to be clauses, and a kCNF is a CNF whose clauses contain at most $k \in \mathbb{N}$ literals.

The Cook-Levin Theorem

The following two languages are $\mathbf{NP}$-Complete:

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

They proved independently the same thing (Cook was working in the US, Levin was in the Soviet Union) and “luckily” they obtained the same results. The class of all satisfiable CNFs is $\mathbf{NP}$-complete (we can’t do better than brute force in the worst case). Even the restriction to CNFs of clauses with at most 3 literals is still $\mathbf{NP}$-Complete (and knowing that 3SAT is NP-Complete means necessarily that SAT is NP-Complete). 2SAT is polynomial.

We don't see the proof 😟. Though, as a rough structure, we consider any language $\mathcal{L}$ in NP and show that this language is at most as difficult as SAT $(\mathcal{L} \leq_p \texttt{SAT})$. To do this, wee have to take into account any possible polynomial p and every poly-time deterministic TM $\mathcal{M}$ (Intuitively, they exist because $\mathcal{L} \in \mathbf{NP}$). Then we define a transformation from string to CNFs, s.t. $\varphi_x \in \texttt{SAT} \iff \exists \texttt{y} \in \{0,1\}^{\texttt{p}(|\texttt{x}|)}.\mathcal{M}(\texttt{x},\texttt{y}) = \texttt{1}.$ It amounts to showing that computation in TM is inherently local. Computation is local means that at any step along the computation a TM only looks at the positions in the tape that are close to the hand. Most of our work works because locality holds.
Finally, we need to show that $\texttt{SAT} \leq_{\texttt{p}} \texttt{3SAT}$

Proving a Problem Hard = Proving a Problem $\mathbf{NP}$-Complete

Proving that $\mathcal{L} \in \mathbf{P}$ wouldn't work, as you'd rather prove $\mathcal{L}$ is easy. If I prove $\mathcal{L} \in \mathbf{EXP}$, I'm not proving anything, as the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ doesn't mean that no polynomial-time algorithm for $\mathcal{L}$ exist. Also proving $\mathcal{L} \in \mathbf{NP}$ doesn't amount to much.

If I prove the problem is $\mathbf{NP}$-Complete, I prove the problem does not belong to a class that is downward close, but it's a border class (with the tools we own in the module). We prove that we know how to do it, we need a certificate and a verifier. You have to prove also the $\mathbf{NP}$-Hardness. We can live happily thanks to the '73s and after.

Two statements to prove:

- The problem is in NP → not the crucial part, you need to write the language $\mathcal{L}$ in the form verifier works with certificate:

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

- The problem is $\mathbf{NP}$-Hard → $\mathcal{H} \leq_p \mathcal{L}$. Usually proven by going through another problem proven to be NP-Complete and work by transitive property.

$$\begin{tikzcd}
\vdots & & \\
\mathcal{H} \arrow[r, "\leq_p"] & \mathcal{J} \arrow[r, "\leq_p"] & \mathcal{L} \\
\cdot & &
\end{tikzcd}$$

Theorem
The Maximum Independent Set Problem $\texttt{INDSET}$ is $\mathbf{NP}$-Complete
The Subset Sum Problem $\texttt{SUBSETSUM}$ is $\mathbf{NP}$-Complete
The Decisional 0/1 Linear Programming Problem $\texttt{ILP}$ is $\mathbf{NP}$-Complete

This isn't the end of the story actually. We can think at $\mathbf{NP}$-Complete problems as forming a graph where edges are natural poly-time reductions.

What can we do once you know a problem is hard? It could well be that some instances can be solved in a reasonable time (not all, but some might take less than 2^{100} years). But we know that we have SAT solvers (and we can move our problems in SAT-Domain).