SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Computational Complexity

Lecture notes integrated with the book "Computational Complexity: a modern approach", S. Arora, B. Barak

Author
Simone Bianco

October 5, 2024

Contents

Information and Contacts **1**

1 Introduction on computation **2**
1.1 Turing Machines 2
1.2 Simulation of Turing machines 5
1.3 Uncomputability and Gödel's theorems 9

2 Basics of complexity theory **11**
2.1 Decision problems and the Class P 11
2.2 Verifiability, Non-determinism and the Class NP 13
2.3 Reductions and NP-Completeness 16
2.4 Boolean circuits and the Class P/poly 20
2.4.1 The Cook-Levin theorem 24

Information and Contacts

Personal notes and summaries collected as part of the *Computational Complexity* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Basic knowledge of computability theory and algorithm complexity

Licence:

1

Introduction on computation

1.1 Turing Machines

Throughout history, humans have been solving problems through a wide variety of models capable of computing valid results, ranging from their intellect to mechanical devices capable of solving problems. In particular, a computation made by a model can be described as a list of sequential operations and initial conditions that will always yield the same result each time the computation is executed.

In modern mathematics, this idea is formalized through the concept of **algorithm**, a finite list of unambiguous instructions that, given some set of initial conditions, can be performed to compute the answer to a given problem. Even though this is a straightforward definition, it isn't as "mathematically stable" as it seems: each computational model could have access to a different set of possible operations, meaning that the same problem could be solved by different computational models in various ways. This innate nature of computational models makes life difficult for mathematicians, who want to prove results that are as general as possible. In 1950, Alan Turing defined the now-called **Turing machine**, an abstract machine capable of capturing the concept of computation itself through simple - but sufficient - operations.

A Turing machine is made of:

- A finite number of *tapes*, each divided into cells. Each cell contains a symbol from a finite set called *alphabet*, usually assumed to contain only 0 and 1, or a special symbol $\sqcup$, namely the *blank character*. The tape is finite on the left side but infinite on the right side. We assume that there always are a read-only *input tape* and a read-write *output tape*. The other tapes are called *work tapes*.
- A finite number of independent *read-write heads*, one for each tape, capable of reading and writing symbols on the tapes. The heads are always positioned on a single cell of their tape and can shift left and right only one cell per shift.

- A finite set of *states* that can be assumed by the machine. At all times the machine only knows its current state. The set contains at least one state that is capable of immediately halting the machine when reached (such states could be unreachable, making the machine go in an infinite loop).
- A finite set of *instructions* which, given the current state and the current cells read by the read-write heads, dictate how the machine behaves. Each instruction tells the machine to do three things: replace the symbols of the current cells (which can be replaced with themselves), move each head one cell to the left, one cell to the right or stay in place, while also moving from the current state to a new one (which can be the current state itself).

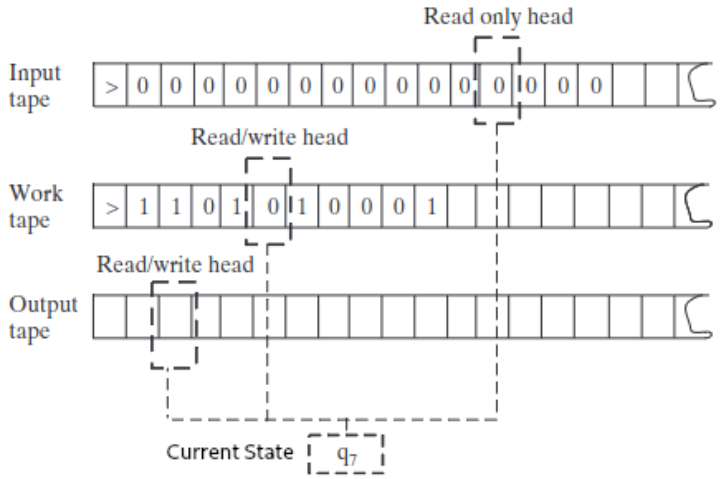


Figure 1.1: Representation of a Turing machine computation step

Definition 1

A k-tape Turing machine is a 7-uple $M = (Q, F, \Gamma, \Sigma, q_{\text{start}}, \delta)$ where:

- Q is a finite set of states, $F \subseteq Q$ is a finite set of halting states and $q_{\text{start}} \in Q$ is the initial state taken by the machine.
- Γ is a finite set of symbols, usually called the tape alphabet. The tape alphabet always contains the symbols $>$, i.e the start of tape symbol, and $\sqcup$, i.e. the blank tape cell symbol.
- Σ is a finite set of symbols, usually called the input alphabet, where $\Sigma \subseteq \Gamma - \{<, \sqcup\}$. The input string can be formed only of these characters.
- $\delta : (Q - F) \times \Gamma^k \to Q \times \Gamma^k \times \{\mathsf{L}, \mathsf{R}\}^k$ is a partial function, usually called the transition function, where L and R represent a left or right shift of the read-write head. Intuitively, if $\delta(q, a_1, \ldots, a_k) = (p, b_1, \ldots, b_k, X_1, \ldots, X_k)$, where $X_i \in \{\mathsf{L}, \mathsf{R}\}$, then, when the machine is in state q and it reads the symbols $a_1, \ldots, a_k$ on the current cells of the tapes, it transitions to the state p, replaces the symbols with $b_1, \ldots, b_k$ and moves the heads in the directions $X_1, \ldots, X_k$.

Given an input $x \in \Sigma^*$, we denote the output of the computation as $M(x)$. By definition, nothing prevents the computation made by a Turing machine to go into infinite loops, thus never halting. The *Church-Turing thesis* defines computation in terms of non-looping Turing machines: any problem that is computable can be solved by **a Turing machine that always halts**, i.e. returns a solution for every possible input.

Definition 2: Computable function

Given a function $f : \Sigma^* \to \Sigma^*$ and a Turing machine M that always halts, we say that M **computes** f if $\forall x \in \Sigma^*$ it holds that $M(x) = f(x)$.

Due to this definition, any Turing machine can be viewed as a function $M : \Sigma^* \to \Sigma^*$. For example, consider the following function:

$$\text{PALINDROME}(x) = \begin{cases} 1 & \text{if } x \text{ is a palindrome string} \\ 0 & \text{if } x \text{ is not a palindrome string} \end{cases}$$

To compute this function, we define the following 3-taped Turing machine M, made of an input tape, a work tape and an output tape:

M = "Given the input string x:

1. Copy the string x from the input tape to the work tape.
2. Move the head of the input tape to the first cell and the head of the work tape to the last written cell.
3. While moving the input tape forward and the work tape backwards, check if the current cell of both heads contains the same symbol. If one pair of different cells is found, write 0 on the output tape and halt the computation.
4. If the input tape head reached the last written cell and the work tape head reached the first cell, write 1 on the output tape and halt the computation"

Clearly, a lot of computational models, such as the modern computer, are more powerful than a Turing machine. First, we have to give a proper definition of *"power"*. In the context of computability theory, we are interested in studying the maximum amount of resources needed by a computational model to compute an answer to the problem they are designed for. In the case of Turing machines, we are interested in **running time** and **required space**.

Definition 3: Running Time and Required Space

Given a TM M that halts on all inputs, we define the **running time** and the **required space** respectively as the functions $T, S : \mathbb{N} \to \mathbb{R}^+$ such that $T(n)$ as the number of transitions of δ executed by M for an input of length n, while $S(n)$ is the number of tape cells visited by the heads during the computation for an input of length n (except the input tape's cells).

For example, consider the previous TM that computes the function PALINDROME. Let $|x| = n$. The first operation requires $2n$ steps of computation to copy the n symbols to the work tape. Then, n more steps are required to adjust the position of the two heads, followed by $2n$ steps to check all pairs of cells. Finally, the last operation requires a single computation step to write the single cell. We conclude that $T(n) = 5n$, while $S(n) = n + 1$.

1.2 Simulation of Turing machines

The definition of Turing machine given in the previous chapter is quite general. There are many variants of our Turing machine model, some which grant more freedom and some which restrict the model. We will show that any computational model can be reduced to a **1-tape minimal alphabet TM**, with a slight loss of power. First, we have to restrict our attention to **time-constructable functions**, a particular subset of functions from that can be computed in an amount of steps that is at most equal to themselves.

Definition 4: Time-constructable function

A function $T : \mathbb{N} \to \mathbb{R}^+$ is said to be **time-constructable** if there is a TM M that computes it in at most $T(n)$ steps for all $n \in \mathbb{N}$.

For example, all the common functions such as $n^2, \log_2 n$ and 2^n are time constructable. We'll start by reducing the tape alphabet. The idea here is pretty simple: for each symbol $a \in \Gamma$, we can choose a binary encoding that represents it. This is similar to how modern computers use standards such as ASCII and UTF to represent characters as strings of 0s and 1s. Given the number of symbols in Γ, we require $c \cdot \log_2 |\Gamma|$ bits to encode each symbol, where $c \geq 1$. When M' has to write a symbol $a \in \Gamma$ on a tape, M' writes the binary encoding $\langle a \rangle$ of such symbol. This means that each step of the computation has to be multiplied by a factor of $c \cdot \log_2 |\Gamma|$.

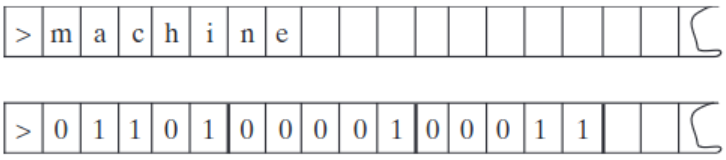


Figure 1.2: Simulation a general alphabet TM through a minimal alphabet TM

Proposition 1: Minimal alphabet Turing machine

Given a TM M with tape alphabet Γ and running time $T(n)$, where T is time-constructable, there is a TM M' with tape alphabet $\{>, \sqcup, 0, 1\}$ and running time $c \cdot \log_2 |\Gamma| \cdot T(n)$, for some constant $c \in \mathbb{R}$, such that $M(x) = M'(\langle x \rangle)$ for all $x \in \Sigma^*$, where $\langle x \rangle$ is the binary encoding of x.

Another change that could be made to the computational model involves the tapes of the machine: in our model, each tape is infinite on the right and finite on the left. This model is usually called **Bidirectional Turing Machine**. Again, we can easily reduce this variant to our model: we can simply double the number of tapes and treat each pair of tapes as two conjoined tapes. This idea is similar to how the number set $\mathbb{Z}$ can be viewed as the union of $\mathbb{N}$ and $-\mathbb{N}$. The running time of the new machine is clearly equal to the original one's.

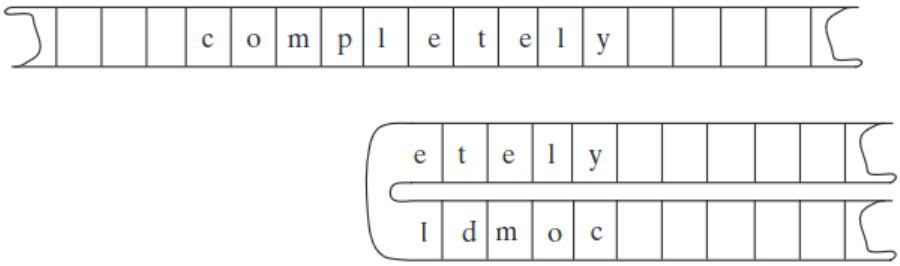


Figure 1.3: Simulation of a bidirectional TM through a monodirectional TM

Proposition 2: Monodirectional Turing machine

Given a bidirectional TM M with tape alphabet Γ and running time $T(n)$, where T is time-constructable, there is a monodirectional TM M' with tape alphabet Γ and running time $T(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma$.

Last but not least, we can reduce a k-tape TM to a single tape TM. Differently from the other two solutions, this idea is not so trivial: we can define the new tape alphabet as a tuple made of k symbols from the original alphabet. For example, given $a_1, \ldots, a_k$ such that a_i is the first cell of the i-th tape, the first cell of the single taped machine will contain the symbol $(a_1, \ldots, a_k)$. This means that the new alphabet requires $|\Gamma|^k$ symbols to represent all possible combinations of cells in the k tapes.

However, this is not sufficient: in the original machine, the k heads were independent from one another. We have to add a way to track where each independent head is in the simulation. To achieve this, we add new special symbols to the tape alphabet: if during a computation step in M the i-th head is on cell j, the i-th symbol in the j-th cell of M' will have a small dot above it to represent the position of the head. This means that the new final alphabet requires $2\,|\Gamma|^k$ symbols to represent all possible combinations of cells and head positions in the k tapes.

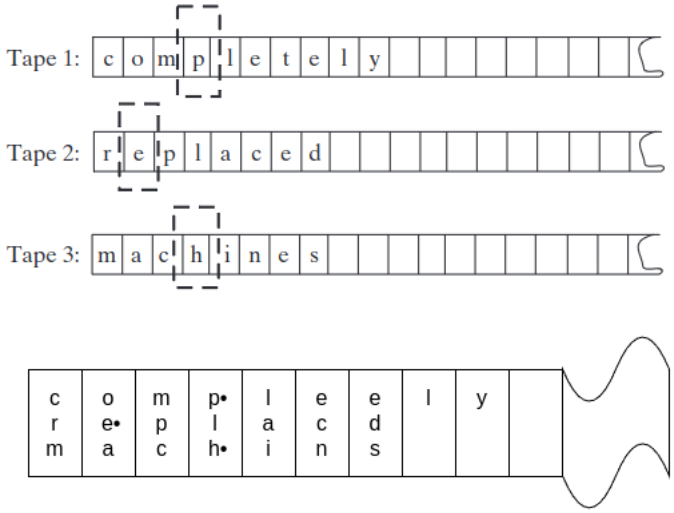


Figure 1.4: Simulation of a k-tape TM through a single tape TM

Before each step of the simulation, the new TM has to first find all the positions of the heads and only then move accordingly. This requires the machine to possibly go back and forth from the first written cell to the latest one.

Proposition 3: Single tape Turing machine

Given a k-tape TM M with running time $T(n)$ and required space $S(n)$, where T is time-constructable, there is a single tape TM M' with running time $n^2 + T(n) \cdot S(n)$ and required space $S(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma^*$.

We can even define an unconventional type of Turing machine, i.e. the **Oblivious Turing machine**. In an Oblivious TM, the movements of the heads are depend only on the length of the input string: for every input $x \in \Sigma^*$ and $i \in \mathbb{N}$, the location of each of the machine's heads at the i-th step of $M(x)$ is only a function of $|x|$ and i. In other words, instead of being defined through the transition function, in an Oblivious TM the movements done by the heads are "implicit". In fact, in this case the transition function is defined as $\delta : (Q - F) \times \Gamma^k \to Q \times \Gamma^k$.

Any standard TM can be transformed into an oblivious one. The idea behind this conversion is similar to how we transformed a k-tape TM to a single tape one. Let $T(n)$ be the running time of the initial TM M. After constructing the single tape, we mark the $T(n)$-th cell with a special symbol, marking a boundary on the right side of the tape. On each step of the computation, the new machine swipes from the leftmost cell to the rightmost one, reading the k head-marks and changing them as M would, to then return to the start of the tape. This small modification makes the new machine's movements oblivious.

The conversion process that we just explained is pretty simple, but quite inefficient since we get a runtime of at most $T(n)^2$. In a more clever way, in which we won't delve into, it's possible to reduce this overhead to a small logarithmic factor.

Proposition 4: Oblivious Turing machine

Given a TM M with running time $T(n)$, where T is time-constructable, there is an single taped Oblivious TM M' with running time $T(n) \cdot \log_2 T(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma^*$.

This capability of reciprocal simulation achievable by different types of Turing machines inspired Turing to show that there is a way to define a singular TM capable of simulating all the others, i.e. an **Universal Turing machine**. To prove this, Turing extended the concept of encoding to machines themselves: if anything can be encoded then TMs can also be encoded! In particular, every encoding $\alpha \in \{0,1\}^*$ corresponds to a single Turing machine M_α, while each TM M' can be obtained by infinitely many encodings. In fact, since we decide how the encoding works, we can encode a TM in multiple ways.

Again the idea here is extraordinarily simple. We assume that U has 5 tapes: an input tape, an output tape, a simulation tape, a tape containing the description of the machine's transition function and one containing the current state of the simulation.

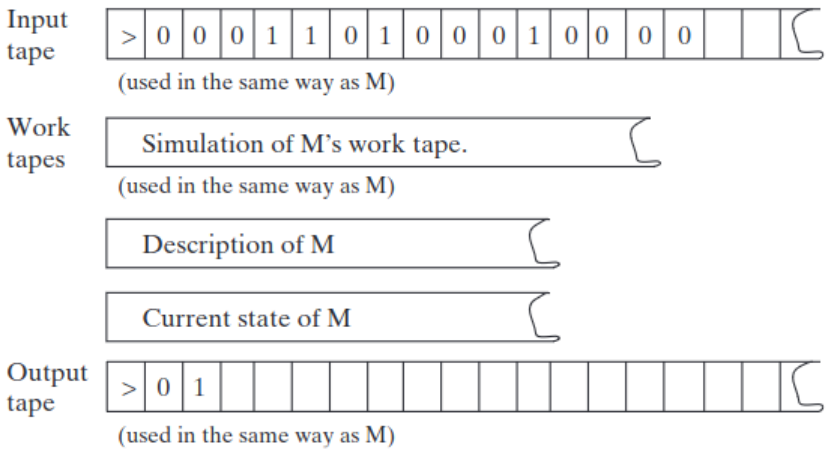


Figure 1.5: Tapes of the Universal Turing machine

Theorem 1: Universal Turing machine

There is a TM U called Universal TM such that for every $x, \alpha \in \{0,1\}^*$ it holds that $U(\langle x, \alpha \rangle) = M_\alpha(x)$. If M_α has running time $T(n)$, the simulation done by U has running time $c \cdot T(n) \cdot \log_2 T(n)$

The existence of such Universal TM shouldn't be a surprise: modern computers are nothing more than a UTMs that can execute any given algorithm, producing an output for a given input. The concept of Universal Turing machine also allows us to easily prove that many other computational models are capable of characterizing computation: if a model is capable of simulating an UTM then it is capable of making any possible computation. This idea is known as **Turing completeness**.

1.3 Uncomputability and Gödel's theorems

After achieving a mathematically stable definition of computation through Turing machines, Turing's focus shifted to understanding which problems are computable and which aren't. Consider the set $\{0,1\}^*$ and the set $\{M_{\alpha_1}, M_{\alpha_2}, \ldots\}$ containing all the possible Turing machines. Through a diagonalization argument, at some point, each machine M_{α_i} can take its own binary encoding as input:

	M_{α_1}	M_{α_1}	M_{α_1}	$\ldots$	M_{α_i}	$\ldots$
α_1	0	1	0	1	1	$\cdots$
α_2	1	0	0	1	0	$\cdots$
α_3	1	1	1	0	1	$\cdots$
$\vdots$	0	1	0	0	0	$\cdots$
α_i	1	0	0	1	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

From this idea, we can considered the following function $\mathrm{UC} : \{0,1\}^* \to \{0,1\}$:

$$\mathrm{UC}(\alpha) = \begin{cases} 0 & \text{if } M_\alpha(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases}$$

This functions takes in input an encoding α and returns 0 if the computation $M_\alpha(\alpha)$ returns 1, while it returns 1 if $M_\alpha(\alpha)$ returns 0 or goes into an infinite loop. By way of contradiction, suppose that there is a TM M that computes UC.

Let $\beta \in \{0,1\}^*$ such that $M = M_\beta$. Then, we get that:

$$\mathrm{UC}(\beta) = 0 \iff M_\beta(\beta) = 1 \iff \mathrm{UC}(\beta) = 1$$

which raises a contradiction. Thus, the only possibility is that this function is actually uncomputable. Through this function, we can show that many other functions are also uncomputable. For example, consider the function $\mathrm{HALT} : \{0,1\}^* \to \{0,1\}$:

$$\mathrm{HALT}(\langle \alpha, x \rangle) = \begin{cases} 1 & \text{if } M_\alpha \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

This functions takes in input an input x and an encoding α, runs $M_\alpha(x)$ and returns 1 if and only if the computation halts. By way of contradiction, suppose that HALT is computable. Then, we define can the following TM M:

M = "Given the encoding α:

1. Compute $h = \mathrm{HALT}(\langle \alpha, \alpha \rangle)$.
2. If $h = 0$, return 1.
3. Otherwise, return $1 - M_\alpha(x)$."

This machine actually computes UC, which we proved to be uncomputable, raising a contradiction. Thus, it must hold that HALT is also uncomputable.

Theorem 2

The functions UC and HALT are uncomputable.

The existence of uncomputable functions - and thus uncomputable problems - gives a negative answer to the *Entscheidungsproblem* (german for *decision problem*), a question posed by David Hilbert in 1928 which asks if there is an algorithm that for each input statement answers "yes" or "no" according to whether the statement is universally true. In addition to this question, Hilbert also posed the question «*is there a strong enough logical system based on recursive axioms and rules that is complete and consistent?*», where:

- A *strong enough* logical system is any proof system that captures basic arithmetic, i.e. any proof system that can describe the natural numbers.
- In a *complete* logical system, if a statement ϕ is true then it is provable
- In a *consistent* logical system, there is no statement ϕ such that both ϕ and $\neg\phi$ are true.

Between completeness and consistency, the latter is clearly more important: if both ϕ and $\neg\phi$ are true, any statement can actually be proved to be both true and false at the same time! In his two acclaimed theorems, Kurt Gödel proved the the answer to Hilbert's additional question is also negative:

- If a strong enough logical system cannot be both complete and consistent
- A consistent logical system cannot prove his own consistency

These two theorems imply that we can do nothing but hope that mathematics is incomplete but at least consistent. Interestingly, Turing's answer can actually be used to give an alternative proof of Gödel's first theorem. First, we have to prove a weaker version of the theorem, where we substitute the concept of consistency with the concept of *soundness*: a logical system is said to be sound if there is no false statement that can be proved. Given $\alpha, x \in \{0,1\}^*$, let $\phi_{\langle\alpha,x\rangle}$ = "$M_\alpha(x)$ halts". Consider the following Turing machine:

M = "Given $\langle\alpha, x\rangle$ in input:

1. Build the formula $\phi_{\langle\alpha,x\rangle}$ and $\neg\phi_{\langle\alpha,x\rangle}$
2. Repeat for all $n \in \mathbb{N}$:
    3. Repeat for all $\pi \in \{0,1\}^n$:
        4. If π is a proof of $\phi_{\langle\alpha,x\rangle}$ then return 1
        5. If π is a proof of $\neg\phi_{\langle\alpha,x\rangle}$ then return 0"

By way of contradiction, suppose that our logical system is both complete and sound. Then, it holds that any statement is true if and only if it can be proven. This makes the machine M capable of computing HALT, which we know to be impossibile. Thus, we get that such logical system cannot be both complete and sound. From this result, is easy to replace soundness with consistency.

2

Basics of complexity theory

2.1 Decision problems and the Class P

Until now, we have discussed about problems described by functions of the general form $f : \Sigma^* \to \Sigma^*$. Furthermore, we have seen how the language Σ^* used by a Turing machine can be restricted to the minimal language comprised of 0 and 1.

Now, we will restrict our attention to **decision problems**, i.e. functions of the form $f : \{0,1\}^* \to \{0,1\}$. These problems can be described as simple questions with a «*yes*» or «*no*» answer, such as asking if some input object has some property or not. A «*yes*» answer is represented by a 1, while a «*no*» answer is represented by a 0. For example, given the language $\mathbb{N}$, the question «*is n a prime number?*» is modeled by the decision problem PRIMES $= \{\langle n \rangle \in \{0,1\}^* \mid n \text{ is prime}\}$.

Definition 5: Language of a Decision problem

The language of a decision problem $f : \{0,1\}^* \to \{0,1\}$ is a subset $L \subseteq \{0,1\}^*$ such that $L = \{x \in \{0,1\}^* \mid f(x) = 1\}$.

A decision problem is said to be *decidable* if there is a Turing machine answers the question posed by the problem with 0 or 1 for any input $x \in \{0,1\}^*$. This also implies that the machine has to halt for every input. The language of a problem decided by a Turing machine M is denoted as $L(M)$. Decidability theory plays a core role in math and computer science since most problems can be modeled through it. These problems can be grouped into different classes based on the minimal running time required for any known TM that solves them.

Definition 6: The DTIME class

Given a function $T : \mathbb{N} \to \mathbb{R}^+$, we define DTIME$(T(n))$ as the class of the languages of decision problems computable by a TM for which any input $x \in \{0,1\}^*$ of length $|x| \leq n$ is accepted or refused in at most $c \cdot T(n)$, for some constant $c \in \mathbb{R}$.

The most important subclass corresponds to the set of problems that can be **efficiently solved**. This class is referred to as P, i.e. the class of problems solvable by a Turing machine in polynomial time. In this context, we define an algorithm as "efficient" if it doesn't require an exponential amount of time.

Definition 7: The class P

We define the class of the languages decidable in polynomial time:

$$\mathsf{P} = \bigcup_{k \geq 0} \text{DTIME}(n^k)$$

For example, consider the *graph connectivity problem*:

$$\text{ST-CONN}(x) = \begin{cases} 1 & \text{if } x = \langle G, s, t \rangle \text{ and } G \text{ is a graph with a path } s \to t \\ 0 & \text{otherwise} \end{cases}$$

This problem can easily be solved in polynomial time through a Depth-first Search algorithm, thus ST-CONN $\in$ P. Not all decision problems are in the class P. For instance, consider the variant of the halting problem:

$$\text{k-HALT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y, k \rangle \text{ and } M_\alpha(y) \text{ halts in at most } k \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

This problem strictly requires exponential time to be solved: the input value k is encoded by $c \log k$ bits but the problem has to simulate $M_\alpha(y)$ for at least k steps, which is exponential in terms of the input size. These problems strictly lie in the class EXPTIME, a superset of the class P.

Definition 8: The class EXPTIME

We define the class of the languages decidable in exponential time:

$$\mathsf{EXPTIME} = \bigcup_{k \geq 0} \text{DTIME}(2^{n^k})$$

Note: by definition P $\subseteq$ EXPTIME

2.2 Verifiability, Non-determinism and the Class NP

Consider the following *formula satisfiability problem*, defined as:

$$\text{SAT}(x) = \begin{cases} 1 & \text{if } x = \langle\phi\rangle \text{ and } \phi \text{ is a satisfiable formula} \\ 0 & \text{otherwise} \end{cases}$$

Currently, we do not know if this problem is in P or not: the best known algorithms have a worst case running time complexity of 2^{cn}, for some $c \in \mathbb{R}$. This result seems strange: given an assignment $\alpha(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the variables of a formula ϕ, we can easily check in linear time if the assignment is valid or not. Shouldn't this problem be easy? The problem here lies in the amount of possible assignments for a given formula ϕ. If ϕ has n variables, then there are 2^n possible assignments to be checked for satisfiability. In other words, it's easy to verify if a given a assignment can satisfy ϕ, but it's very hard to find this assignment. This idea allows us to introduce the concept of **verification**.

Definition 9: Verifier

Given a computable function f, we say that a TM M is a **verifier** for f if $\forall x \in \{0,1\}^*$ it holds that $f(x) = 1$ if and only if there at least one additional string $w \in \{0,1\}^*$, called **witness** (or *certificate*), such that $M(x, w) = 1$

This definition can be rewritten in terms of *completeness* and *soundness* of the verifier, where the witness w acts as a proof for the statement $f(x) = 1$:

- Completeness: $f(x) = 1 \implies \exists w \in \{0,1\}^*$ such that $M(x, w) = 1$
- Soundness: $f(x) = 0 \implies \forall w \in \{0,1\}^*$ it holds that $M(x, w) = 0$

Mimicking the class P of efficiently solvable decision problems, we define the class of efficiently verifiable decision problems.

Definition 10: The class NP

We define the class of the languages decidable in polynomial time:

$$\text{NP} = \{L \subseteq \{0,1\}^* \mid L \text{ is verifiable in polynomial time}\}$$

By definition, it clearly holds that $\text{P} \subseteq \text{NP}$: if f is solvable in polynomial time then we can use any string as a witness and proceed to solve the problem. Moreover, in order for a verifier to have polynomial time complexity, the length of the witness w must always be at most $\text{poly}(n)$, since otherwise the machine wouldn't even be able to read all of it. Thus, we can always assume that $|w| \leq \text{poly}(|x|)$.

Lots of decision problems are known to be in the class NP, such as the 3-COL problem, which asks the question «*is this graph G 3-colorable?*», the CNF-SAT problem, which asks the question «*is this CNF formula ϕ satisfiable?*», the 3-SAT problem, which asks the question «*is this 3-CNF formula ϕ satisfiable?*» and the GRAPH-ISO problem, which

asks the question «*are these two graphs* G_1, G_2 *isomorphic to each other?*». We can easily notice that we can use efficient verifiers to solve decision problems. However, this process still requires exponential time.

Proposition 5: Decidability through verifiability

Given a language L, if L is verifiable in at most $T(n)$ time then it is decidable in at most $T(N) \cdot 2^{T(N)}$ time

Proof. Let M be a verifier for L that runs in at most $T(n)$ time. We define M' as follows:

M' = "Given the input string x:

1. Let $n = |x|$.
2. Repeat for $i = 0, \ldots, T(n)$:
    3. Repeat for each $w \in \{0,1\}^i$:
        4. Compute $b = M(x, w)$.
        5. If $b = 1$, return 1.
6. Return 0"

M' clearly decides the language L, but requires to generate all the possible witnesses of length at most $T(n)$ and then run $M(x, w)$, requiring at most $T(N) \cdot 2^{T(N)}$ steps. □

Corollary 1

NP $\subseteq$ EXPTIME.

Currently, it is not known whether P = NP or not. Researchers believe that the answer to this conjecture is false. For instance, the *Exponential time hypothesis* states that the 3-CNF version of the SAT problem, i.e. 3-SAT, cannot be solved in subexponential time, meaning that it requires at least 2^{cn} steps of computation, for some $c \in \mathbb{R}$, and thus that the currently known algorithms are the best we can achieve.

The answer to this question is considered to be one of the most important questions in mathematics. If P = NP were to be true, a lot of key problems in mathematics that are currently only efficiently verifiable could be solved in a reasonable amount of time by a modern computer. On the other hand, a large number of current technologies are based on the assumption that P $\neq$ NP. For example, cryptography assumes that it's easy to check that each encrypted string is the result of the encryption scheme being applied to the original message, which works as the certificate, and very hard to find this message only through the encrypted string. If P = NP were proven false, we would have to reconsider a large portion of the modern world, even digital currencies themselves.

We saw how a Turing machine computes a solution following a precise, step-by-step procedure dictated by a set of rules. This type of computation is said to be **deterministic**.

In a deterministic computation, the TM has only one possible action to take for each state it can assume. A **non-deterministic** computation, instead, can be thought of as having the ability to explore many different potential computation paths at once. It can "guess" the correct sequence of steps from the start to a halting state.

Definition 11

A k-tape Non-deterministic Turing machine (NDTM) is a TM provided with two distinct transition functions δ_1, δ_2. On each step, the computation made by the machine forks: one branch follows δ_1 and the other follows δ_2. The whole computation can be viewed as a binary tree. The machine accepts an input if and only if at least one of the branches of the computation tree accepts.

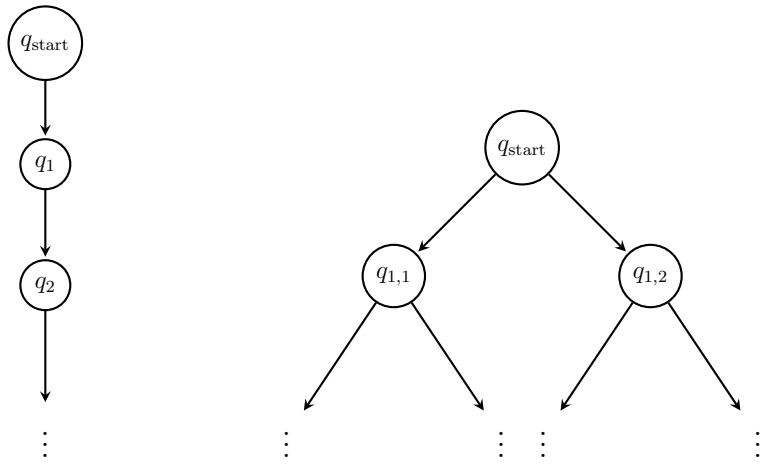


Figure 2.1: A deterministic computation (left) and a non-deterministic one (right)

Intuitively, non-deterministic Turing machines are more powerful than deterministic ones due to the possibility of exploring multiple choices without increasing the running time. We also notice that the number of transition functions doesn't matter: if we have a NDTM with h transition functions, we can simulate it with a NDTM with 2 transitions functions.

Definition 12: The NTIME class

Given a function $T : \mathbb{N} \to \mathbb{R}^+$, we define DTIME($T(n)$) as the class of the languages of decision problems computable by a NDTM for which any input $x \in \{0,1\}^*$ of length $|x| \leq n$ is accepted or refused in at most $c \cdot T(n)$, for some constant $c \in \mathbb{R}$.

Surprisingly, the concept of efficient non-deterministic Turing machine is equivalent to the concept of efficient verification, meaning that NP is also the class of all the languages decidable in polynomial time by a NDTM. In fact, this was the original definition of this class of decision problems, hence the name NP standing for Non-determinist P. However, since non-deterministic Turing machines are only a theoretical tool and not a real, concrete and realizable model of computation, it was quickly changed to the one based on verification.

Theorem 3: The class NP (2nd Definition)

$$\mathsf{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k)$$

Proof. Suppose L has a polynomial time verifier V in time at most n^k, for some $k \in \mathbb{N}$. We know that for each input $x \in L$ there is a witness $w \in \{0,1\}^{n^k}$. We define a NDTM N that takes in input the string x, non-deterministically generates all the possible strings $w \in \{0,1\}^{n^k}$ and then runs $V(x,w)$, accepting if and only if V accepts. Each branch takes at most $2n^k$ steps.

$$x \in L \iff \exists w \in \{0,1\}^* \;\; V(x,w) \iff \text{A branch of } N \text{ accepts}$$

Now suppose that L is decided by a NDTM N' in time at most n^k, for some $k \in \mathbb{N}$. Then, we know that $\forall x \in L$ there is at least a path P of length at most n^k that accepts x. We define a verifier V' for which the encoding of the accepting path of each input acts as the witness, concluding that:

$$x \in L \iff \text{A path } P \text{ of } N'(x) \text{ accepts} \iff \exists \langle P \rangle \in \{0,1\}^* \;\; V'(x,P) = 1$$

concluding that $L \in \mathsf{NP}$. □

2.3 Reductions and NP-Completeness

One of the most interesting aspects of computable (and uncomputable) problems is the ability to be transformed into another problem in order to achieve a solution. Suppose that we have an instance a of problem A and that we know an algorithm that transforms a into an instance b of a problem B such that a is a «*yes*» answer if and only if b is a «*yes*» answer. Then, by solving b we would get an answer to a.

In computer science, this concept is known as **reduction**: a problem A is said to be reducible into a problem B, written as $A \leq B$, if any instance a of A can be mapped into an instance b of B whose solution gives a solution to the former. For instance, the proof of the uncomputability of HALT given for Theorem 2. In that case, we *reduced* UC to HALT.

Many-to-one reductions are the simplest type of reduction possible, where a function f maps strings from a language A to strings of a language B.

Definition 13: Many-to-one reduction

Given two languages A, B, we say that A is **many-to-one reducible** to B, written as $A \leq_m B$, if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that:

$$x \in A \iff f(x) \in B$$

For example, consider the problems 3-SAT and SAT. The former is clearly reducible to the latter through the identity function $\mathrm{f}(x) = x$: a 3-CNF formula F is satisfiable if and only if F itself is satisfiable!.

For example, consider the problems CNF-SAT and 3-SAT. We can construct a many-one-reduction from the former problem to the latter. Consider a CNF formula $F = \bigwedge_{i=1}^{m} C_i$ where C_i is a logica clause, i.e. $C_i = \bigvee_{j=0}^{t} \ell_j$. We can construct a new 3-CNF formula $F' \bigwedge_{i=1}^{m} D_i$ by substituting each clause C_i with a new subformula D_i:

1. If $C_i = \ell_1$, we define D_i as:

$$D_i = (\ell_1 \vee y_1 \vee y_2)(\ell_1 \vee \overline{y_1} \vee y_2)(\ell_1 \vee y_1 \vee \overline{y_2})(\ell_1 \vee \overline{y_1} \vee \overline{y_2})$$

   where y_1, y_2 are two new variables.

2. If $C_i = \ell_1 \vee \ell_2$, we define D_i as:

$$D_i = (\ell_1 \vee \ell_2 \vee y_1)(\ell_1 \vee \ell_2 \vee \overline{y_1})$$

   where y_1 is a new variable.

3. If $C_i = \ell_1 \vee \ell_2 \vee \ell_3$, we define $D_i = C_i$.

4. If $C_i = \ell_1 \vee \ldots \vee \ell_t$ with $t > 3$, we define D_i as:

$$D_i = (\ell_1 \vee \ell_2 \vee y_1)(\overline{y_1} \vee \ell_3 \vee y_2) \ldots (\overline{y_{t-3}} \vee \ell_j \vee y_{t-2})(\overline{y_{t-2}} \vee \ell_{t-1} \vee \ell_t)$$

For the first three cases, it's easy to see that given an assignment α it holds that $C_i(\alpha) = 1 \iff D_i(\alpha)$. The fourth case, instead, requires more attention. Suppose that there is an assignment α that satisfies C_i. This implies that at least one ℓ_j must be set to true in α. Then, in every subclause of D_i we can either set y_i or $\overline{y_{i-1}}$ to true in order to satisfy D_i. Vice versa, suppose that there is an assignment β that satisfies D_i. By way of contradiction, suppose that $\ell_1, \ldots, \ell_t = 0$ in β. Without loss of generality, we can assume that $y_1, \ldots, y_{t-2} = 1$ in β,. Then, for any assignment of y_t, the final subclause $(\overline{y_{t-2}} \vee \ell_{t-1} \vee \ell_t)$ cannot be satisfiable, raising a contradiction. Thus, there must be at least one ℓ_j set to true in β, implying that β also satisfies C_i.This concludes that F is satisfiable if and only if F' is satisfiable. Moreover, F' is a 3-CNF, hence CNF-SAT $\leq_m$ 3-SAT.

Many-to-one reductions between problems are transitive: starting from a problem A, we can reduce it to a problem B through a function f and then reduce it to a problem C through a function g. This implies that $h = g \circ f$ is a reduction from A to C. Hence, we have that CNF-SAT $\leq_m$ 3-SAT $\leq_m$ SAT.

Proposition 6: Transitive reduction

Given three languages A, B, C, if $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.

In particular, we notice that both of the previous reductions are easily computable in polynomial time by a Turing machine. When this happens, the reduction is said to be a **Karp reduction**.

Definition 14: Karp reduction

Given two languages A, B, we say that A is **Karp reducible** to B, written as $A \leq_p B$, if there is a many-to-one reduction from A to B computable in polynomial time

Clearly, transitivity between reductions also holds in this case: if two reductions f and g are both computable in polynomial time then $h = g \circ f$ also is.

Proposition 7: Transitive Karp reduction

Given three languages A, B, C, if $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.

When a problem A is Karp reducible to a problem B solvable (or verifiable) in polynomial time by a machine M, i.e. a problem that is in P (or NP), we can build a new Turing machine M' that first computes the reduction from A to B, then runs M and finally accepts if and only if M accepts the reduced instance. This new machine M' clearly also requires polynomial time, meaning that A is also solvable (or verifiable) in polynomial time.

Proposition 8

Let A, B be two languages such that $A \leq_p B$. Then:

- If $B \in \mathsf{P}$ then $A \in \mathsf{P}$.
- If $B \in \mathsf{NP}$ then $A \in \mathsf{NP}$.

Reductions between problems enable us to *classify* problems based on their "hardness": if a problem A is efficiently reducible to a problem B, then solving A cannot be harder than solving B, forming some sort of hierarchy between problems. In other words, if $A \leq_p B$ then A is at most hard as B and B is at least as easy as A.

Building on this idea of hardness, we can identify some privileged problems that are harder than a whole class of problems! For example, every problem from the class NP is efficiently reducible to a single problem B, then each NP problem is as hard as B. When this happens, we say that B is **NP-Hard**. When an NP-Hard problem also lies in NP, we say that such problem is **NP-Complete**.

Definition 15: NP-Hardness and NP-Completeness

A language B is said to be NP-Hard if $\forall A \in \mathsf{NP}$ it holds that $A \leq_p B$. If B is also in NP, we say that it is NP-Complete.

The simplest NP-Complete problem is the *TM satisfiability problem*, defined as:

$$\text{TM-SAT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y, 1^n, 1^t \rangle \text{ and } \exists w \in \{0,1\}^n \text{ s.t. } M_\alpha(y, w) = 1 \text{ within } t \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

First we show that this problem is in NP. Given an input $x = \langle \alpha, y, 1^n, 1^t \rangle$ and a witness $w \in \{0,1\}^*$, we start by asserting that $|w| = n$. Then, we simulate $M_\alpha(y, w)$ with a counter that immediately stops if the simulation executes more than t steps. Since the input has size $|x| \geq t$ and the simulation takes at most t steps, the verifier requires polynomial time, concluding that TM-SAT $\in$ NP. Finally, we show that this problem is also NP-Hard. Given a language $A \in \mathsf{NP}$, we know that it can be verified in at most $|x|^k$ for some $k \in \mathbb{N}$. Suppose that such verifier is encoded by β. The reduction is easily given by mapping any input $y \in A$ to $f(x) = \left\langle \beta, y, 1^{|w|}, 1^{|y|^k} \right\rangle$, where w is y's witness for A, concluding that $A \leq_p$ TM-SAT.

We notice that, by definition, NP-Hard problems can even be uncomputable! For instance, consider the *input acceptance problem*:

$$\text{ACCEPT}(x) = \begin{cases} 1 & \text{if } x = \langle \alpha, y \rangle \text{ and } M_\alpha(y) = 1 \\ 0 & \text{otherwise} \end{cases}$$

It's pretty easy to see that any NP problem (and more generally any computable problem) is Karp reducible to ACCEPT, meaning that ACCEPT is NP-Hard. However, this problem is clearly uncomputable since HALT $\leq_m$ ACCEPT.

If an NP-Hard problem could be solved in polynomial time, i.e. be inside P, then every single problem NP-Hard problem would also be solvable in polynomial time, meaning that P = NP. However, when an NP-Hard problem also lies in the class NP, this relation also becomes valid in the opposite direction: if P = NP then any NP-Hard problem that is in NP is also in P.

Proposition 9

Given a language B, it holds that:

- If B is NP-Hard then $B \in \mathsf{P}$ implies that $\mathsf{P} = \mathsf{NP}$
- If B is NP-Complete then $B \in \mathsf{P}$ if and only if $\mathsf{P} = \mathsf{NP}$

In other words, if an NP-Complete problem is proven to be solvable or unsolvable in polynomial time, we would get a proof for the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question. Moreover, by definition, each NP-Complete is Karp reducible to one another, meaning that they all of them

share the same hardness. These are the reasons why these problems are referred to as *complete*.

Even thought it is indeed NP-Complete, the problem TM-SAT(x) is quite artificial since, by definition, in order to efficiently solve it we would have to be capable of efficiently solving any efficiently verifiable problem, which is literally the same as asking to solve the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question. A more interesting NP-Complete problem is the *satisfiability problem* discussed in the previous sections, which actually is the first ever problem to be shown to be NP-Complete. This result is known as the Cook-Levin theorem.

Theorem 4: The Cook-Levin theorem

SAT and 3-SAT are NP-Complete

The standard proof of this theorem is quite long and tedious, requiring a very careful notation in order to make things clearer. Instead, we will achieve this theorem by proving that the *circuit satisfiability problem* is NP-Complete and then reducing it to both SAT and 3-SAT.

2.4 Boolean circuits and the Class P/poly

Boolean circuits are defined as sets of logical AND, logical OR and logical NOT gates connected by cables. Boolean circuits have been proven to be Turing complete due to Turing machines and circuits being capable of simulating each other up to a polynomial factor. Again, none should be dumbfounded by this result: any modern computer is just a large amount of Boolean gates wired together.

Definition 16: Boolean circuit

A Boolean circuit is a directed acyclic graph whose nodes, called gates, each associated with either an input variable or a Boolean operator. Each input gate has in-degree 0 and unlimited out-degree. Each Boolean gate has an out-degree gate equal to 1 (except for the output gate which has out-degree 0) and in-degree equal to either 1 or 2. All the 1 in-degree gates compute the logical NOT and all 2 in-degree gates compute the logical AND or the logical OR of their given input variables or Boolean function.

Each gate v is associated with the Boolean function f_v computed by it. An assignment $x = x_1, \ldots, x_n$ defines the result of the computation for a gate. A function $f : \{0,1\}^n \to \{0,1\}$ is said to be computed by a circuit with output gate u if for all inputs $x \in \{0,1\}^n$ it holds that $f(x) = C(x)$, where $C(x)$ is the function computed by the output gate.

The complexity of Boolean circuits is measured in terms of their *size* and *depth*, i.e. the number of gates of the circuit and the length of the longest directed path from an input gate to the output gate. The **circuit complexity** of a function f is defined as the size of the smallest Boolean circuit that computes it.

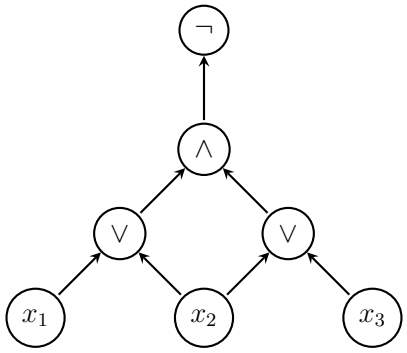


Figure 2.2: A Boolean circuit of size 7 and depth 3 computing $\overline{(x_1 \vee x_2)(x_1 \vee x_3)}$.

Boolean circuits can also be defined in a different way. Instead of using logical NOT gates, we can assume that the the negations $\overline{x_1}, \ldots, \overline{x_n}$ are also input variables. Any standard circuit of size S and depth D can be easily transformed into this different type of circuit by repeatedly applying the De Morgan rule on all the logical NOT gates, producing a circuit of size at most $2S$ and depth at most D due to the number of input gates being doubled and the logical NOT gates being removed. These circuits are usually called **De Morgan circuits**.

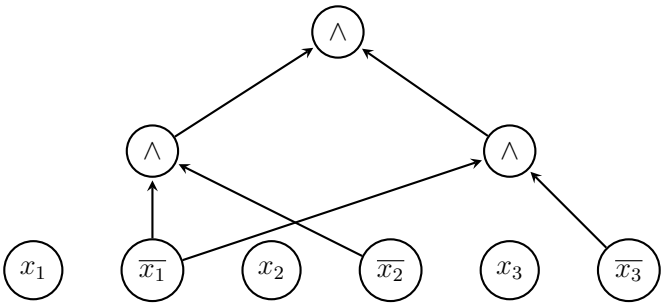


Figure 2.3: A De Morgan circuit of size 9 and depth 2 computing $\overline{(x_1 \vee x_2)(x_1 \vee x_3)}$.

An important thing to notice is that Boolean circuits do not allow computations to be reused. For example, consider the *n bit XOR function*:

$$\mathrm{XOR}(x) = \bigoplus_{i=1}^{n} x_i$$

Intuitively, we can compute this function through recursion:

$$(\neg\mathrm{XOR}(x_1, \ldots, x_{\frac{n}{2}}) \vee \mathrm{XOR}(x_{\frac{n}{2}+1}, \ldots, x_n)) \wedge (\mathrm{XOR}(x_1, \ldots, x_{\frac{n}{2}}) \vee \neg\mathrm{XOR}(x_{\frac{n}{2}+1}, \ldots, x_n))$$

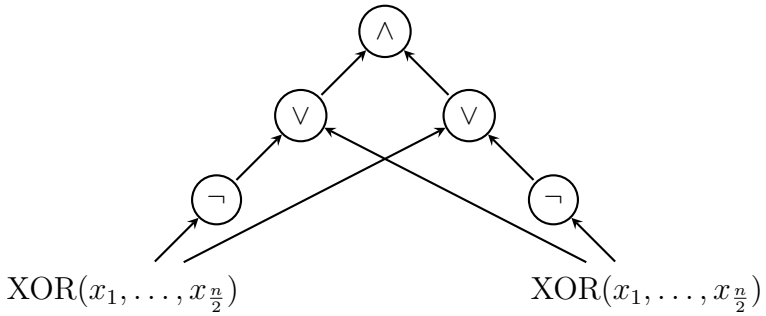


Figure 2.4: Recursive computation of XOR(x)

If we could model a circuit in this way, the size would be given by $S(n) = 5 + 2S\left(\frac{n}{2}\right)$, which is approximately $O(n)$. Without reusing computations, instead, we have to make two copies of each recursion.

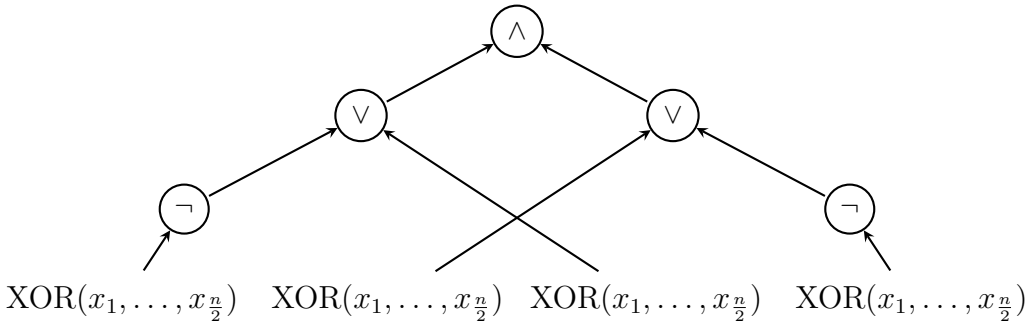


Figure 2.5: Recursive Boolean circuit that computes XOR(x)

Hence, the size of the circuit is actually given by $S(n) = 5 + 4S\left(\frac{n}{2}\right)$, that is $O(n^2)$. Differently from Turing machines, circuits are capable of computing only functions with a fixed amount of input bits. To compute a variable input size, we need a *family of circuits* $\{C_n\}_{n\in\mathbb{N}}$, where C_n computes all the inputs of length i.

Definition 17: Circuit family

A $T(n)$-size **circuit family** is a sequence $C_{n n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs, a single output and size at most $T(n)$. We say that a language L is decidable by a circuit family $C_{n n\in\mathbb{N}}$ if $\forall x \in \{0,1\}^n$ it holds that $x \in L$ if and only if $C_n(x) = 1$.

Through circuit decidability, we can define an equivalent of the class P, i.e. the class P/poly. The notation "/poly" comes from the equivalence between polynomial sized circuit families and Turing machines that take a polynomial amount of "advice bits" (this topic won't be discussed). Circuit families are even capable of simulating non-advice-taking Oblivious Turing machines. This result will enable us to get many strong results, including the proof of the Cook-Levin theorem that we omitted before.

Definition 18: The class P/poly

We define the class of the languages decidable by a poly-sized circuit family:

$$\mathsf{P/poly} = \bigcup_{k \geq 0} \mathrm{SIZE}(n^k)$$

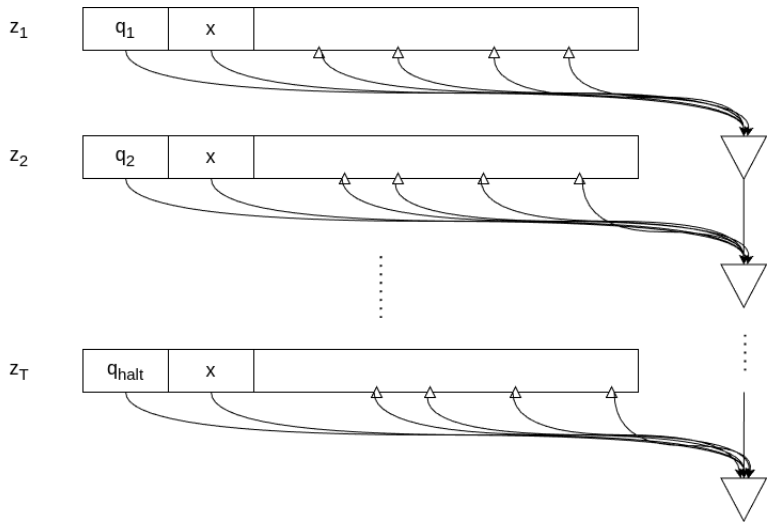
where $\mathrm{SIZE}(f(n))$ is the set of all languages decided by a $f(n)$-sized circuit family

Theorem 5: Simulating TMs through circuit families

Let L be a language decided by a TM in time $T(n)$, where T is time-constructable, and let $L_n = L \cap \{0,1\}^n$ for all $n \in \mathbb{N}$. There is a circuit C_n of size $cT(n)\log_2 T(n)$ such that $x \in L_n$ if and only if $C_n(x) = 1$,

Proof. Let M be the k-taped TM that decides L in time $T(n)$. We know that M can be converted into an Oblivious TM M' that decides L in time $T(n)\log_2 T(n)$. Let $T = T(n)\log_2 T(n)$ and le $z_1, \ldots, z_T$ be the sequence of *snapshots* of the computation of M', that is the sequence of the machine's state and symbols read by all the k-heads. We know that M' can use at most T cells, meaning that we can restrict our focus on the first T cells of every tape. Moreover, since M' is oblivious, we know that each i-th step of computation only depends on n and i, thus we can ignore the movements of the heads.

Hence, each snapshot z_i is encoded by a constant-sized binary string, meaning that we can compute the string z_i based on the input length n, the previous snapshot z_{i-1} and the snapshots $z_{i_1}, \ldots, z_{i_k}$, where z_{i_j} denotes the last step that the j-th head of M' was in the same position as it is in the i-th step. The composition of all the T constant-sized circuits gives rise to a circuit C_n of size $O(T)$ that computes from the input x the snapshot z_T of the last step of $M'(x)$, meaning that $C_n(x) = M'(x)$. Applying this process for all $n \in \mathbb{N}$, we get a circuit family that decides L. □



Encoding of an Oblivious TM as a circuit

The previous result concludes that from each TM that runs in time $T(n)$ we can build a circuit family $\{C_n\}_{n\in\mathbb{N}}$ that can decide the same language with size $cT(n)\log_2 T(n)$, giving us the following additional result.

Corollary 2

$\mathsf{P} \subseteq \mathsf{P/poly}$

2.4.1 The Cook-Levin theorem

We're finally ready to give a simple proof for Theorem 4. Consider the following *circuit satisfiability problem* as follows:

$$\text{CIRCUIT-SAT}(x) = \begin{cases} 1 & \text{if } x = \langle C\rangle \text{ and } C \text{ is a satisfiable circuit} \\ 0 & \text{otherwise} \end{cases}$$

A circuit is said to be satisfiable if there is at least one assignment of its variables such that $C(x) = 1$. Each circuit C_n built in the conversion from a TM to a circuit family can also be constructed in time $cT(n)\log_2 T(n)$. This allows us to prove that CIRCUIT-SAT is actually NP-Complete.

Lemma 1

CIRCUIT-SAT is NP-Complete

Proof. CIRCUIT-SAT is clearly in NP, since a satisfying assignment can act as a witness verifiable in polynomial time. We show that CIRCUIT-SAT is also NP-Hard. Let L be any language in NP and let V be verifier of L that runs in time at most n^k for some $k \in \mathbb{N}$. We define a TM M as follows:

M = "Given the input string $\langle x\rangle$:

1. Let $w \in \{0,1\}^m$ be a witness such that $V(x,w) = 1$
2. Compute $n = |x|$ and $m = |w|$.
3. Build the circuit C_V such that $C_V(x,w) = V(x_w)$ as described in Theorem 5
4. Build the circuit $C_x : \{0,1\}^m \to \{0,1\}$ such that $C_x(w) = C_V(x,w)$"

This machine computes a many-to-one reduction from L to CIRCUIT-SAT:

$$x \in L \iff \exists w \in \{0,1\}^* \;\; V(x,w) = 1 \iff$$

$$\exists w \in \{0,1\}^* \;\; C_x(w) = 1 \iff C_x \in \text{CIRCUIT-SAT}$$

Since both C_V and C_x can be built in $cn^k\log_2(n^k)$, we conclude that this reduction is actually a Karp reduction. □

Lemma 2

CIRCUIT-SAT $\leq_p$ CNF-SAT

Proof. Given a circuit C, let S be its size. We encode each gate g inside the circuit as a CNF ϕ_g:

- If g is a logical NOT gate with the input gate g_1 then we define ϕ_g as:

  $$\phi_g = (g \vee g_1)(\overline{g} \vee g_1)$$

  This subformula evaluates as true if and only if $g = \neg g_1$.

- If g is a logical AND gate with input gates g_1, g_2, we define ϕ_g as:

  $$\phi_g = (g \vee \overline{g_1} \vee \overline{g_2})(\overline{g} \vee g_1)(\overline{g} \vee g_2)$$

  This subformula evaluates as true if and only if $g = g_1 \wedge g_2$.

- If g is a logical OR gate with input gates g_1, g_2, we define ϕ_g as:

  $$\phi_g = (\overline{g} \vee g_1 \vee g_2)(g \vee \overline{g_1})(g \vee \overline{g_2})$$

  This subformula evaluates as true if and only if $g = g_1 \vee g_2$.

Let $\phi_C = \bigwedge_{i=1}^{S} \phi_{g_i}$ be the formula that encodes every gate in C. This encoding perfectly describes the computation made by C. In order to force this computation, we consider the formula $\phi = g_S \wedge \phi_C$, where g_S is the output gate. We get that C is satisfiable if and only if the ϕ is satisfiable. Moreover, since each subformula ϕ_g has at most 7 literals inside it, the final formula ϕ has at most $1 + 7S$ literals, implying that it can be constructed in polynomial time. □

This lemma gives us the following Karp reduction chain

$$\text{CIRCUIT-SAT} \leq_p \text{CNF-SAT} \leq_p 3 \leq \text{SAT}$$

Since CIRCUIT-SAT is NP-Complete, through transitivity of Karp reductions we get that all these problems are NP-Complete, finally concluding the proof of the Cook-Levin theorem!