“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 2, 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 8

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 ⊔, 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).

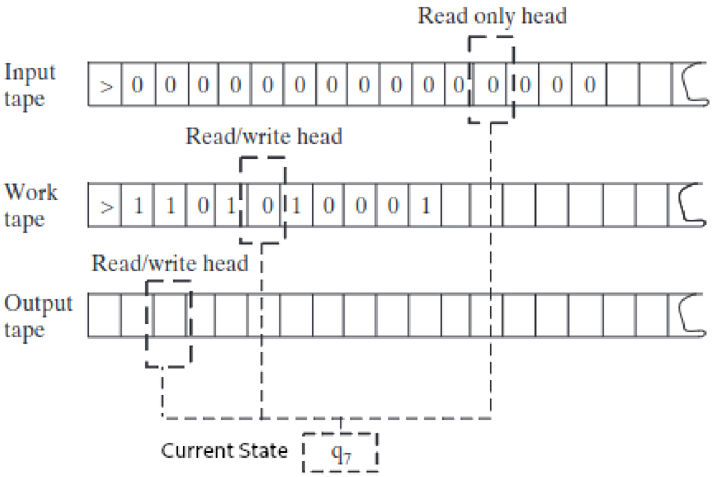

Figure 1.1: Representation of a Turing machine computation step

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 $\delta(q, a_1, \dots, a_k) = (p, b_1, \dots, b_k, X_1, \dots, X_k)$, where $X_i \in \{\mathsf{L}, \mathsf{R}\}$, then, when the machine is in state q and it reads the symbols $a_1, \dots, a_k$ on the current cells of the tapes, it transitions to the state p, replaces the symbols with $b_1, \dots, b_k$ and moves the heads in the directions $X_1, \dots, X_k$.

Given an input $x \in \Sigma^*$, we denote the output of the computation as $M(x)$. The *Church-Turing thesis* defines computation in terms of Turing machines: any problem that is computable can be solved by **a Turing machine that always halts**.

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:

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

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'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 $c \cdot \log_2 |\Gamma|$.

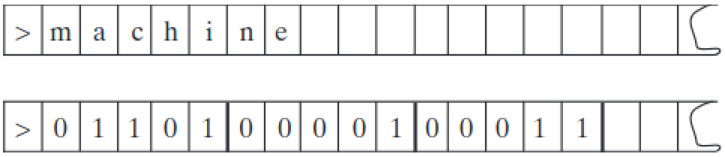

Figure 1.2: Simulation a general alphabet TM through a minimal alphabet TM

Proposition 1: Minimal alphabet Turing machine

Given a TM M with tape alphabet Γ and running time $T(n)$, 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.

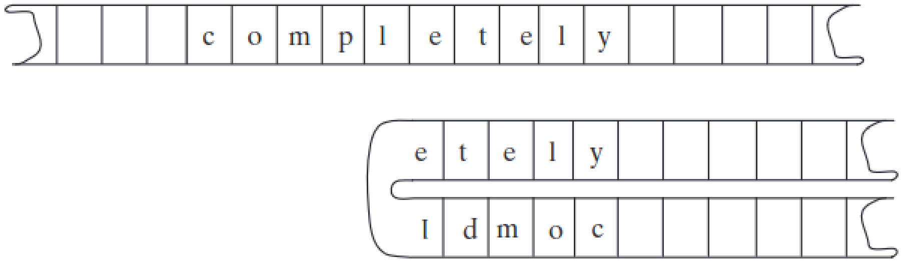

Figure 1.3: Simulation of a bidirectional TM through a monodirectional TM

Proposition 2: Monodirectional Turing machine

Given a bidirectional TM M with tape alphabet Γ and running time $T(n)$, 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.

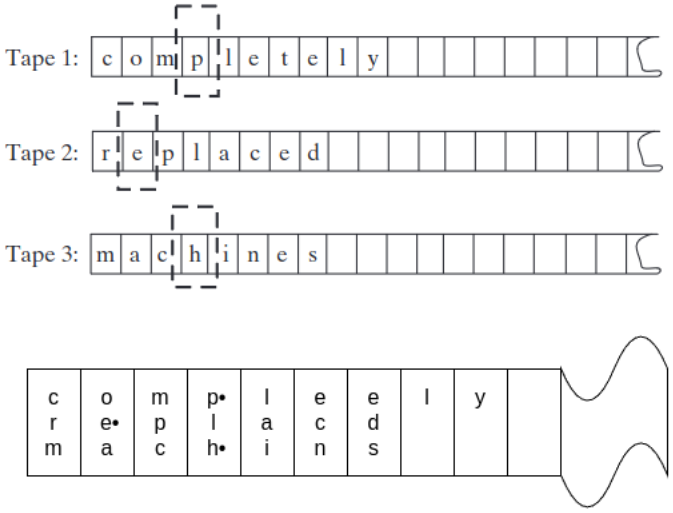

Figure 1.4: Simulation of a k-tape TM through a single tape TM

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 $S(n)$, 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. In other words, 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. Any standard TM can be transformed into an oblivious one. However, this process is not so easy to describe, so we'll omit how the conversion works.

Proposition 4: Oblivious Turing machine

Given a TM M with running time $T(n)$, there is an oblivious TM M' with running time $T(n) \cdot \log T(n)$ such that $M(x) = M'(x)$ for all $x \in \Sigma^*$.

Obliviousness comes in handy mostly in the fields of cryptography and zero-knowledge proofs. For now, we won't further discuss this model.

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.

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 $c \cdot T(n) \cdot \log T(n)$

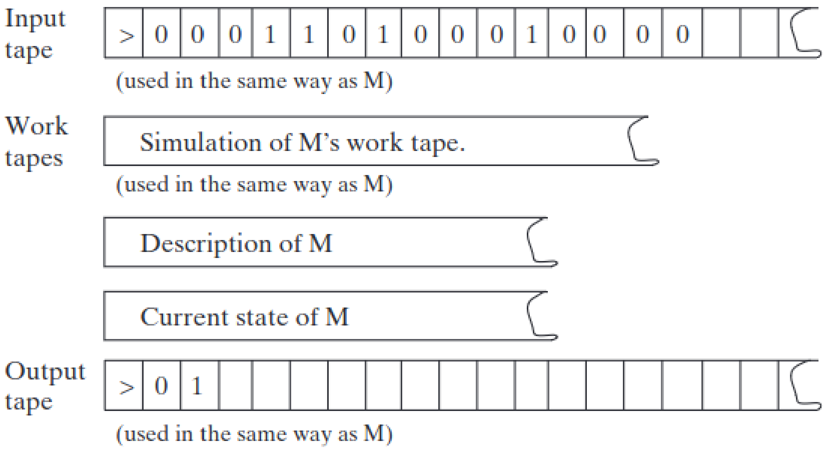
Input
tape
> 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0
(used in the same way as M)
Work
tapes
Simulation of M’s work tape.
(used in the same way as M)
Description of M
Current state of M
Output
tape
> 0 1
(used in the same way as M)

Figure 1.5: Tapes of the Universal Turing machine

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 $\mathrm{UC} : \{0,1\}^* \to \{0,1\}$ defined as:

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

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.

Theorem 2

The function UC is uncomputable.

Through this function, we can show that many other functions are also uncomputable. For example, consider the function HALT : $\{0,1\}^* \to \{0,1\}$ defined as:

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

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 $h = \mathrm{HALT}(\alpha, \alpha)$.
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 3

The function HALT is 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 $\pi$ is a proof of $\phi_{\langle \alpha, x \rangle}$ then return 1
            5. If $\pi$ 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.