A Concise Introduction to Computer Science

Computación Cuántica I
S. E. Venegas-Andraca
Facultad de Ciencias
Universidad Nacional Autónoma de México

Septiembre 2024

1 Introduction

The Theory of Computation is a scientific field devoted to understanding the fundamental capabilities and limitations of computers, and it is divided into three areas:

1. Automata Theory. The study of different models of computation.
2. Computability Theory. This focuses on determining which problems can be solved by computers and which cannot.
3. Complexity Theory. The objective in this area is to understand what makes some problems computationally hard and others easy.

The development and analysis of algorithms starts with defining a problem in proper mathematical jargon, followed by choosing a model of computation upon which we first determine whether such a problem is solvable, in principle, by implementing a series of steps in the model of computation we have adopted. If that were the case, then the last step would be to quantify the amount of resources needed to execute the algorithm.

This introduction to theoretical computer science is mainly based on [2], [3], and [1].

2 Decision problems, encoding schemes and languages

Formally speaking, computers are mathematical abstractions of devices built *to process symbols*. Since computers have been designed and built to solve problems, we now address a key concept in theoretical computer science: expressing the formal definition of a problem as well as the main characteristics of its solutions in an abstract manner.

Definition 2.1. Alphabet. 1) Any finite set of symbols Σ is an alphabet. 2) For any alphabet Σ, we denote by Σ^* the set of all finite strings of symbols from Σ.

Definition 2.2. Encoding scheme. An encoding scheme e for a problem Π provides a way for describing each instance of Π by an appropriate string of symbols over some fixed alphabet Σ.

Central to the use of sophisticated mathematical structures and physical theories in the creation of algorithms is the concept of *NP-completeness*, a theory designed to be applied only to decision problems. Let us briefly review what a decision problem is by using a *primus inter pares* example: The Traveling Salesman Problem.

Definition 2.3. The Traveling Salesman Problem (optimization version)
INSTANCE: A finite set $C = \{c_1, c_2, \ldots c_m\}$ of "cities" and a "distance" $d(c_i, c_j) \in \mathbb{N}$, the set of natural numbers.
QUESTION: Which is the shortest "tour" of all the cities in C, that is, an ordering $[c_{\Pi(1)}, c_{\Pi(2)}, \ldots, c_{\Pi(m)}]$ of C such that $[\sum_{i=1}^{m-1} d(c_{\Pi(i)}, c_{\Pi(i+1)})] + d(c_{\Pi(m)}, c_{\Pi(1)})$ is minimum?

Definition 2.4. The Traveling Salesman Problem (decision problem version)
INSTANCE: A finite set $C = \{c_1, c_2, \ldots c_m\}$ of "cities" a "distance" $d(c_i, c_j) \in \mathbb{N}$ and a bound $B \in \mathbb{N}$.
QUESTION: Is there a "tour" of all the cities in C having total length no more than B, that is, an ordering $[c_{\Pi(1)}, c_{\Pi(2)}, \ldots, c_{\Pi(m)}]$ of C such that $[\sum_{i=1}^{m-1} d(c_{\Pi(i)}, c_{\Pi(i+1)})] + d(c_{\Pi(m)}, c_{\Pi(1)}) \leq B$?

The correspondence between decision problems and languages is brought about by the encoding schemes we use for specifying problem instances whenever we intend to compute with them.

Thus the problem Π and the encoding scheme e for Π partition Σ^* into three classes of strings: those that are not encodings for instances of Π, those that encode instances of Π for which the answer is 'no', and those that encode instances of Π for which the answer is 'yes'. This third class of strings is the language we associate with Π and e:

Definition 2.5. Language for problem Π under encoding e.
$L(\Pi,e) = \{\ x \in \Sigma^* \mid \Sigma$ is the alphabet used by e, and x is the encoding under e of an instance $I \in Y_\Pi\ \}$

Our formal theory is applied to decision problems by saying that, **if a result holds for the language $L(\Pi,e) \Rightarrow$ it also holds for the problem Π under the encoding scheme e.**

3 Models of computation

So far we have formally defined what a language is, being that a preliminary step towards the definition of an algorithm. However, before delivering such a definition, let us make a quick reflection: when we think of an algorithm, i.e. a set of steps executed in a computer, we are assuming we have a computer, i.e. a physical machine or a mathematical model upon which we may be able to execute such steps. Consequently, prior to the definition of an algorithm, we must define a computer.

3.1 Deterministic vs Nondeterministic models of computation

For every single mathematical model of a computer machine, it is possible to define two variants: deterministic and nondeterministic versions.

When every step of a computation follows in a unique way from the preceding step we are doing **deterministic** *computation. In a* **nondeterministic** *machine, several choices may exist for the next state at any point. Non determinism is a generalization of determinism.*

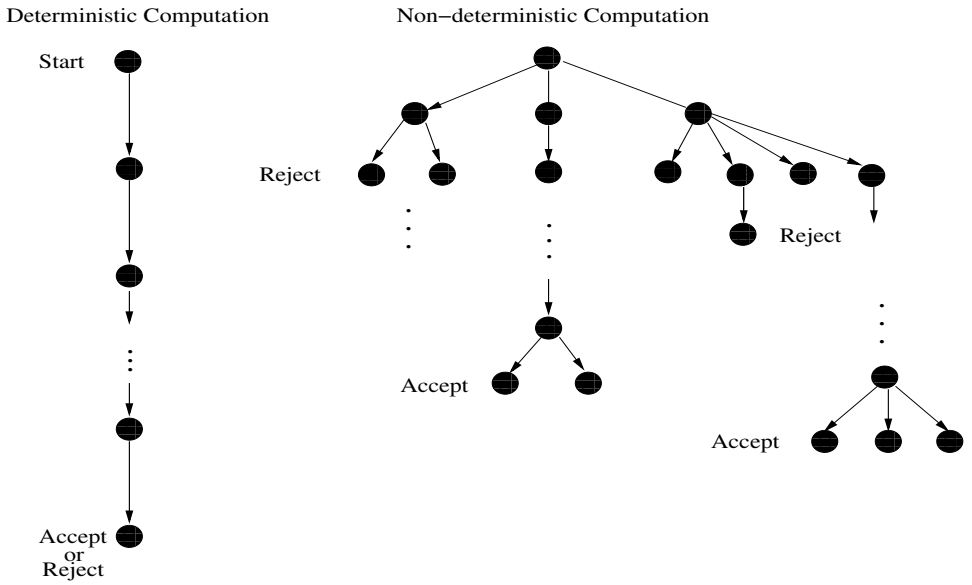

Figure 1: In deterministic computation, every single step is fully determined by the previous step. In nondeterministic computation, a step may be followed by n new steps or, equivalently, a nondeterministic computer machine makes n copies of itself, one for each possibility.

How does a nondeterministic machine (NM) compute? Suppose that we are running an NM on input symbol i and come to a q_i state with multiple states to proceed. After reading input symbol i, the machine splits into *multiple* copies of itself and follows all the possibilities in *parallel*. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol does not appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state. A graphical illustration of a nondeterministic computation is given in Fig. (1).

3.2 Deterministic Turing Machines

A Deterministic Turing Machine (DTM) is an accurate model of a general purpose computer. A DTM, pictured schematically in Fig. (2), can do everything a real computer can do (more on this in the following subsection.)

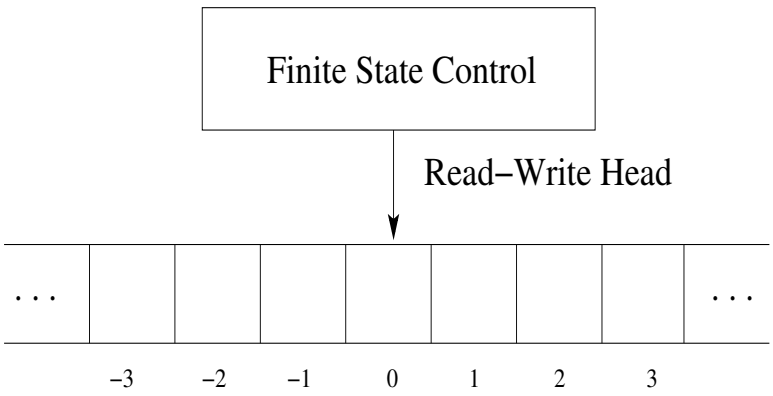

Figure 2: The 'hardware' elements of a Deterministic Turing Machine (DTM) are a scanner (read-write head plus a finite state control system) and a limitless memory-tape (the tape is divided into squares or cells). The scanner reads and writes information on the cells of the tape and controls the state of the DTM.

A DTM consists of a scanner and a limitless memory-tape that moves back and forth past the scanner. The tape is divided into squares. Each square may be blank (␣) or may bear a single symbol (0 or 1, for example). The scanner is able to examine only one square of tape at a time (the 'scanned square'). The scanner has mechanisms that enable it to erase the symbol on the scanned square, and to move the tape to the left or right, one square at a time. Also, the scanner is able to alter the state of the machine: a device within the scanner is capable of adopting a number of different states, and the scanner is able to alter the state of this device whenever necessary. The operations just described - erase, print, move, and change state - are the basic operations of a DTM. Complexity of operation is achieved by chaining together large numbers of these simple basic operations.

Definition 3.1. Deterministic Turing Machine. A Deterministic Turing Machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and
1. Q is the set of states
2. Σ is the input alphabet not containing the blank symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$.

Definition 3.2. Computation with a Deterministic Turing Machine.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a DTM.

Initially M receives its input $w \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is filled with blank symbols. The head starts on the leftmost square of the tape (Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input.) Once M has started, the computation proceeds according to the rules described by δ. The computation continues until it enters either the accept or reject states at which point it halts. If neither occurs, M just does not stop.

As a DTM computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the DTM. A configuration C_1 yields configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. In an accepting configuration the state of the configuration is q_{accept}. In a rejecting configuration the state of the configuration is q_{reject}. Accepting and rejecting configurations are *halting* configurations and do not yield further configurations.

A DTM M **accepts** input w if a sequence of configurations $C_1, C_2, \ldots C_k$ exists, where
1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

We say that M **accepts** language A if $A = \{w | M \text{ accepts } w\}$, i.e. A is the set of all strings accepted by M.

3.3 Example. Running a program in a Deterministic Turing Machine

The program specified by $\Gamma = \{0, 1, \sqcup\}$ ($\sqcup$ is the blank symbol), $Q = \{q_0, q_1, q_2, q_3, q_{\text{accept}}, q_{\text{reject}}\}$ and δ, provided in Table 1, is used in a deterministic Turing machine M to determine whether a number can be divided by 4 or, equivalently, whether the last two digits of a binary number, reading from left to right, are two consecutive zeros.

Table 1. Example of a deterministic Turing machine.

Q/Γ	0	1	$\sqcup$
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \sqcup, L)$
q_1	$(q_2, \sqcup, L)$	$(q_3, \sqcup, L)$	$(q_{\text{reject}}, \sqcup, L)$
q_2	$(q_{\text{accept}}, \sqcup, L)$	$(q_{\text{reject}}, \sqcup, L)$	$(q_{\text{reject}}, \sqcup, L)$
q_3	$(q_{\text{reject}}, \sqcup, L)$	$(q_{\text{reject}}, \sqcup, L)$	$(q_{\text{reject}}, \sqcup, L)$

The program works as follows. For a state $q_i \in \{q_0, q_1, q_2, q_3\}$ specified in the LHS column and a given alphabet symbol $s_j \in \{0, 1, \sqcup\}$ specified in the top row, the box that corresponds to row q_i and column s_j and contains three symbols: the first symbol is the new state of M, the second symbol is the new alphabet symbol that will be written in the current cell (substituting symbol s_i) and the third symbol specifies the motion direction of the read-write head. So, row q_i and column s_j are the current configuration of M and the symbols contained in the box corresponding to row q_i and column s_j are the next configuration of M.

For example, let $X = 10100$ be an input binary string (we shall read the input string from left to right). The initial state of M is q_0 and M's tape reads as

$\sqcup$	**1**	0	1	0	0	$\sqcup$

where our first input symbol (in bold face) is the leftmost **1**. So,the initial configuration of M is $q_0, \mathbf{1}$.

The transition function specifies that for a state q_0 and input symbol 1, M must take q_0 as new state, write the value 1 in the cell where its read-write head is now located (**1**) and take its read-write head one cell forward, i.e. to the right. So, M is now in the configuration $q_0, \mathbf{0}$ and its tape reads as

$\sqcup$	1	**0**	1	0	0	$\sqcup$

.

The full run of this program is given in the following sequence

$$
\begin{array}{lllll}
\text{Step 1:} & q_0, \sqcup\mathbf{1}0100\sqcup & \rightarrow & q_0, \sqcup 1\mathbf{0}100\sqcup \\
\text{Step 2:} & q_0, \sqcup 1\mathbf{0}100\sqcup & \rightarrow & q_0, \sqcup 10\mathbf{1}00\sqcup \\
\text{Step 3:} & q_0, \sqcup 10\mathbf{1}00\sqcup & \rightarrow & q_0, \sqcup 101\mathbf{0}0\sqcup \\
\text{Step 4:} & q_0, \sqcup 101\mathbf{0}0\sqcup & \rightarrow & q_0, \sqcup 1010\mathbf{0}\sqcup \\
\text{Step 5:} & q_0, \sqcup 1010\mathbf{0}\sqcup & \rightarrow & q_0, \sqcup 10100\sqcup \\
\text{Step 6:} & q_0, \sqcup 10100\sqcup & \rightarrow & q_1, \sqcup 1010\mathbf{0}\sqcup \\
\text{Step 7:} & q_1, \sqcup 1010\mathbf{0}\sqcup & \rightarrow & q_2, \sqcup 101\mathbf{0}\sqcup\sqcup \\
\text{Step 8:} & q_2, \sqcup 101\mathbf{0}\sqcup\sqcup & \rightarrow & q_y, \sqcup 10\mathbf{1}\sqcup\sqcup\sqcup
\end{array}
$$

We have said that DTMs can do everything a real computer can do, and real computers compute values of functions. Transducer Machines, a DTM variant, are capable of those computations.

Definition 3.3. Transducer Machines. A transducer machine T is used to compute functions. T has a string $w \in \Sigma^*$ as input and produces another string y as output. Output y is stored in a special tape called the output tape. Given a function f, a transducer T computes f if the computation $T(w)$ reaches a final state containing $f(w)$ as output whenever $f(w)$ is defined (if f is not defined, T never reaches a final state). A function f is computable if a Turing Machine T exists capable of computing it.

3.4 The Church-Turing Thesis

Alan Turing published a most influential paper in 1936 [5] in which he pionereed the theory of computation, introducing the famous abstract computing machines now known as *Turing Machines*. In [5], Turing:

1) Defined a systematic method, our modern definition of an **algorithm**.
2) Provided a rigorous definition of a Turing machine, i.e. a powerful model of computation.
3) Proved that it was possible to build a particularly powerful machine called **Universal Turing Machine (UTM)** that could simulate any other Turing machine in reasonable time.
4) Conjectured the **Church-Turing Thesis**, in which he established an equivalence correspondence between the existence of Turing machines and that of systematic methods. If the Church-Turing thesis is correct, then the existence or non-existence of systematic methods can be replaced *throughout mathematics* by the existence or non-existence of Turing machines.

5) Explained the fundamental principle of the modern computer, the idea of controlling the machine's operation by means of a program of coded instructions stored in the computer's memory, i.e. a Turing machine capable of simulating any other Turing machine (the Universal Turing Machine being the actual digital computer and the simulated Turing machine the program that has been encoded in the digital computer's memory).
6) Proved that not all precisely stated mathematical problems can be solved by computing machines (examples: the decision and the halting problems.)

No wonder why Alan Turing is such a big name! When Turing wrote [5], a computer was not a machine at all, but a human being. A computer was a mathematical assistant who calculated by rote, in accordance with a systematic method. The method was supplied by an overseer prior to the calculation. It is in that sense that Turing uses the word 'computer' in [5] and a Turing machine is an idealized version of this human computer. What Turing did in [5] was to propose a mathematical formalism for the idealization of a human computer as well as to study the calculation capabilites and limitations of that mathematical model.

3.1. The Church-Turing Thesis. *Three ways to express the thesis are:*
1. The UTM can perform any calculation that any human computer can carry out.
2. Any systematic method can be carried out by the UTM.
3. Every function which would be naturally regarded as computable can be computed by the Universal Turing Machine.

It is somewhat ironic that the branch of S&T that is perceived by society at large as a very precise and quantitative discipline, rests upon a conjecture! This is one of the reasons for the tremendous philosophical and scientific relevance of David Deutsch's formulation of the Church-Turing principle:

3.2. The Church-Turing principle *[6]. Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.*

In Deutsch's words, the rationale behind the Church-Turing priciple was "to reinterpret Turing's 'functions which would be naturally regarded as computable' as the functions which may in principle be computed by a real physical system. For it would surely be hard to regard a function 'naturally' as computable if it could not be computed in Nature, and conversely".

3.5 Algorithmic Complexity for DTMs

The performance of models of computation in the execution of an algorithm is a fundamental topic in the theory of computation. Since the quantification of resources (in our case, we focus on time) needed to find a solution to a problem is usually a complex process, we just estimate it. To do so, we use a form of estimation called **Asymptotic Analysis** in which we are interested in the maximum number of steps S_m that an algorithm must be run on large inputs. We do so by considering only the highest order term of the expression that quantifies S_m. For example, the function $F(n) = 18n^6 + 8n^5 - 3n^4 + 4n^2 - \pi$ has five terms, and the highest order term is $18n^6$. Since we disregard constant factors, we then say that f is asymptotically at most n^6. The following definition formalises this idea.

Definition 3.4. Big O Notation. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if $\exists\ \alpha, n_o \in \mathbb{N}$ such that $\forall\ n \geq n_o$

$$f(n) \leq \alpha g(n)$$

So, $g(n)$ is an asymptotic upper bound for $f(n)$ ("f is of the order of g"). Bounds of the form n^β, $\beta > 0$ are called **polynomial bounds**, and bounds of the form 2^{n^γ}, $\gamma \in \mathbb{R}^+$ are called **exponential bounds**. $f(n) = O(g(n))$ means informally that f grows as g or slower. Big O notation says that one function is asymptotically no more than another.

Example 1. Prove the following statements:

1) $\sum_{i=1}^{n} i = O(n^2)$
2) $\sum_{i=1}^{n} i^2 = O(n^3)$
3) $\sum_{i=1}^{n} 3^i = O(3^n)$

Example 2.

1) Define Linear complexity
2) Define Quadratic complexity
3) Define exponential complexity
4) Compute the order (i.e. the computational complexity) of straightforward definition of matrix multiplication.

A DTM can be used to find a solution to a problem, so how efficiently such solution can be found? As stated previously, we shall be interested in finding the fastest algorithms. Let us now introduce a few concepts needed to quantify the efficiency of an algorithm.

The time complexity of an algorithm A expresses its time requirements by giving, for each input length, the largest amount of time needed by A to solve a problem instance of that size.

Definition 3.5. Time Complexity Function for a DTM. Let M be a DTM. We define $f : \mathbb{N} \to \mathbb{N}$ as the time complexity function of M, where $f(n)$ is **the maximum number of steps** that M uses on any input of length n.

Definition 3.6. Time Complexity Class for DTMs. Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. We define the time complexity class **TIME(t(n))**, as the collection of all languages that are decidable by an $O(t(n))$ time DTM.

Definition 3.7. Class P. The class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine is denoted by **P** and is defined as $\mathbf{P} = \bigcup_k \mathbf{TIME}(n^k)$ As an example, the language provided in subsection (3.3) is in **P**.

A most important question to reflect on:

What is the relationship between the algorithmic complexities introduced in example 2 and the formal definition of complexity in DTMs? In other words, what is the meaning of 'elementary step' in both a Turing machine and an algorithm?

The central idea here is: **for each algorithmic elementary step one has to perform a polynomial number of steps in a DTM.**

Example:
- Calculate the computational complexity of an algorithm that computes the multiplication of two matrices of order n by using the textbook definition. Executing this algorithm under the rationale of a DTM would be living hell!

A *polynomial time* or *tractable algorithm* is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function p, where n is used to denote the input length. Any algorithm whose time complexity function cannot be so bounded is called an *exponential time* or *intractable algorithm*. Tractable algorithms are considered as acceptable, as a sign that a satisfactory solution for a problem has been found. Finding a tractable algorithm is usually the result of obtaining a deep mathematical insight into a problem. In contrast, intractable algorithms are usually solutions obtained by exhaustion, the so called brute-force method, and are not considered satisfactory solutions.

Example of tractable and ingenious algorithms.
- Computational complexity of an algorithm that computes the multiplication of two matrices of order n by using the textbook definition. $O(n^3)$.
- Strassen's algorithm for matrix multiplication. $O(n^{log_2 7})$

Example of intractable algorithms, i.e. intractable problems.
- Computational complexity of 3SAT (we shall study this problem in detail on the second half of this course). If n binary variables $\Rightarrow 2^n$ different n-bit strings!
- Computational complexity for TSP. Factorial complexity.

So, no tractable algorithm for the Traveling Salesman Problem (2.3) is known so far. All solutions proposed so far are based on enumerating all possible solutions. **Why is the Traveling Salesman problem intractable?** *Nobody knows for sure.* It could be either that we need a deeper knowledge of the characteristics of this problem or, simply, that the Traveling Salesman problem is inherently intractable. However, no proof for neither of these two alternatives has been supplied so far.

References

[1] S.E. Venegas-Andraca, DPhil thesis: Discrete Quantum Walks and Quantum Image Processing, The University of Oxford (2006)

[2] M. Sipser, Introduction to the Theory of Computation, PWS (2005).

[3] M.R. Garey and D.S. Johnson, Computers and Intractability. A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., NY (1979).

[4] R. Motwani and P. Raghavan, Randomized Algorithms, CUP (1995).

[5] A.M. Turing, "On Computable Numbers, with an application to the Entscheidung problem", Proceedings of the London Mathematical Society, **42**, pp. 230-265 (1936-37).

[6] D. Deutsch, "Quantum theory, the church-turing principle and the universal quantum computer", Proc. Royal Soc. of London. Series A, **400(1818)**, pp. 97–117 (1985).

[7] S.A. Cook, The Complexity of Theorem-Proving Procedures, Proc. 3rd Ann. ACM Symposium on the Theory of Computing, pp. 151-158 (1971).