# A Gentle Introduction to Theoretical Computer Science

**Salvador E. Venegas-Andraca**

Facultad de Ciencias, UNAM
svenegas@ciencias.unam.mx

## What is the ultimate objective of Theoretical Computer Science?

Theoretical Computer Science, traditionally thought of as a branch of mathematics, is divided into three fields linked by the question

*What are the fundamental capabilities and limits of computers?*

## Fields of the Theory of Computation

Automata theory. Definitions and properties of abstract models of computation.
Computability theory. What problems can(not) be solved by computers?
Complexity theory. What makes some problems computationally hard and others easy?

## Automata (1/3)

Automata are mathematical models of physical devices used to compute.

The definition of any automaton requires three components: states, language and transition rules.

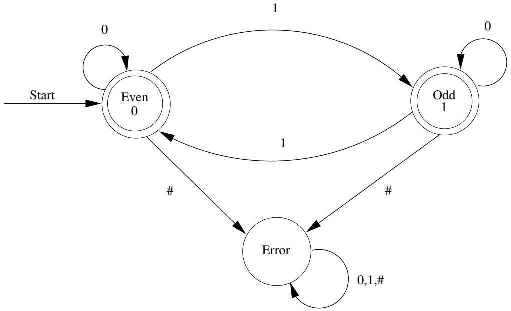
## Automata (2/3)

- States are abstractions of the different stages an automaton would go through while computing (e.g., *on* and *off* states of a switch).

- A language is a set of strings made by concatenating characters from a given alphabet. For example, one could define a language from alphabet $\{$a,b$\}$, consisting of all strings that begin with ‘a’ and end with ‘b’. This language would include ‘ab’, ‘aabb’, and ‘ababab’, but not ‘ba’ or ‘bbb’.

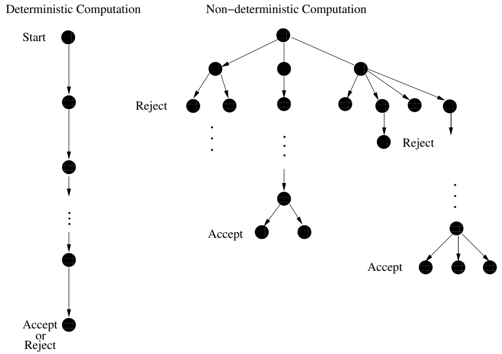
- Finally, transition rules describe the internal dynamics of an automaton (i.e., how and when an automaton changes its state).

## Automata (3/3) - An example of an automaton



A finite automaton for parity computation. Double-circled states are accept states and single-circled state is a reject state. Input strings for $\mathcal{R}$ are any combination of characters taken from $\Sigma = \{0, 1, \#\}$. Since an empty set of characters has no $1$s, we set the initial parity of a string as even (this is why the state 'Even' has an arrow labeled as 'Start').

## Deterministic & Nondeterministic Computation (1/3)



Deterministic automata follow a simple rule: given an input datum, every state of the computation is followed by only one state.

## Deterministic & Nondeterministic Computation (2/3)

Each and every automaton has deterministic and nondeterministic versions.

- Deterministic automata follow a simple rule:

  Given an input datum, every state of the computation is followed by only one state.

- In contrast, when computing on a nondeterministic machine:

  a state could be followed by *many* states

  depending on the input datum and the transition rules.

*Nondeterministic computation can be thought of as parallel computation with unlimited resources*.

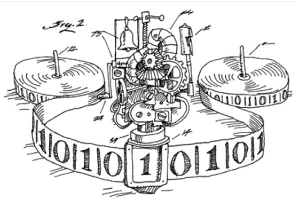
## Deterministic & Nondeterministic Computation (3/3)

Resources are always limited, hence nondeterministic computation may seem to be an unreasonable model.

However, there are two good reasons to use this model: i) to determine if a problem is computable even if limitless resources are available, and ii) to define a key set of computable problems: NP problems.

## Turing Machines (1/3)

So far, Turing machines constitute the most accurate model of general purpose computers.



Turing Machines (drawing by Tom Dunne, American Scientist, March-April 2002)

The 'hardware' elements of a Deterministic Turing Machine (DTM) are a limitless memory tape (the tape is divided into squares or cells), and a scanner which consists of a read-write head *plus* a finite state control system. The scanner has two purposes: to read and write information on the cells of the tape as well as to control the state of the DTM.

## Turing Machines (2/3)

There are Deterministic and Nondeterministic Turing machines.

DTMs are constrained to having access to only one computational state at a time, NTMs have access to as many states as needed.

Both Turing machines models are equally powerful in terms of computability.

## Turing Machines (3/3)

Three important properties of DTMs:

1) There is a Turing Machine, known as **The Universal Turing Machine** (UTM), that can simulate the behavior of any other Turing machine.

2) The UTM and the von Neumann computer (our silicon-based computers, like laptops) are equivalent in terms of computability power. In other words, any problem that can be solved using a von Neumann computer can also be solved using the UTM.

3) Although highly nontrivial, an arbitrary program written in any computer language can be translated into a series of elementary steps in a DTM (more on elementary steps shortly).

## Intuitive Intro to Computational Complexity (1/12)

In complexity theory, for a problem $\mathcal{P}$ that can be computed in a Turing machine, we are interested in answering the following question:

What is the minimum amount of time or energy / number of elementary steps that must be invested in order to solve $\mathcal{P}$?

## Intuitive Intro to Computational Complexity (2/12)

**Wait! What do we mean by elementary step?**

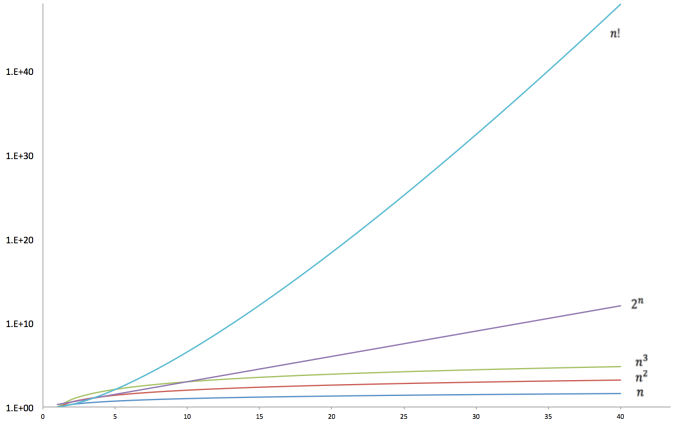
- Machine language
- Assembly language
- High level language
- Abstract algorithm (like computing the inner product between two vectors).

The key point: polynomial transformations from one level of algorithm description to another.

## Intuitive Intro to Computational Complexity (3/12)

Easy vs Difficult Algorithms

(1 step/ns $\Rightarrow 3.15 \times 10^{22}$ steps/one million years)



## Intuitive Intro to Computational Complexity (4/12)

Example of an easy problem: square matrix multiplication.
Assume addition and multiplication as elementary steps.

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} a_{1i}b_{i1} & \cdots & \sum_{i=1}^{n} a_{1i}b_{in} \\ \sum_{i=1}^{n} a_{2i}b_{i1} & \cdots & \sum_{i=1}^{n} a_{2i}b_{in} \\ \vdots & & \vdots \\ \sum_{i=1}^{n} a_{ni}b_{i1} & \cdots & \sum_{i=1}^{n} a_{ni}b_{in} \end{pmatrix}$$

The textbook multiplication of two $n \times n$ matrices requires $n$ multiplications and $n-1$ additions per entry. Since the resulting matrix has $n^2$ entries, then the total amount of elementary steps required to multiply two $n \times n$ matrices is

$$(n + (n-1))n^2 = 2n^3 - n^2$$

## Intuitive Intro to Computational Complexity (5/12)

For example, suppose that $n = 1000$ and that a computer running the above-mentioned algorithm computes an elementary step in $1\ \mu s$ (which is a rather slow computer). So,

$$2n^3 - n^2\Big|_{n=10^3} = 2 \times 10^9 - 10^6 = 1999 \times 10^6 \text{ elementary steps}$$

Now, if each elementary step takes $1\ \mu s$ then the total running time will be equal to

$$(1999 \times 10^6 \text{elementary steps})(1 \times 10^{-6} \frac{s}{\text{elementary steps}}) = 1999s = 33.31\ min$$

which is perfectly fine.

## Intuitive Intro to Computational Complexity (6/12)

In contrast, let us now analyze a key problem in theoretical computer science: the K-SAT problem.

## Intuitive Intro to Computational Complexity (7/12)

**K-SAT, a Fundamental NP-Complete Problem**

Let $B = \{x_1, x_2, \ldots, x_n, \bar{x_1}, \bar{x_2}, \ldots \bar{x_n}\}$ be a set of Boolean variables. Also, let $C_i$ be a disjunction of $k$ elements of $B$ and $F$ be a conjunction of $m$ clauses $C_i$.
Question: Is there an assignment of Boolean variables in F that satisfies all clauses simultaneously, i.e. F=1?
For example:

## Intuitive Intro to Computational Complexity (8/12)

Suppose $B = \{x_1, x_2, x_3, x_4, x_5, x_6, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6\}$ and

$$
\begin{aligned}
P \quad = \quad & (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_1 \vee x_2 \vee \bar{x}_5) \wedge (x_3 \vee x_4 \vee x_5) \wedge \\
& (x_4 \vee x_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_6) \wedge \\
& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6) \wedge (x_3 \vee \bar{x}_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \\
& (x_2 \vee x_5 \vee \bar{x}_6) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_2 \vee x_3 \vee x_6) \wedge \\
& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee x_6) \wedge \\
& (\bar{x}_2 \vee x_3 \vee \bar{x}_6) \wedge (x_2 \vee x_5 \vee x_6) \wedge (x_3 \vee x_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_6) \wedge \\
& (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_1 \vee x_2 \vee \bar{x}_3)
\end{aligned}
$$

Finding the solutions (if any) of even a modest SAT instance can become difficult quite easily. Can you tell whether $P$ has any solution at all and, if so, a bit string that could satisfy $P$?

## Intuitive Intro to Computational Complexity (9/12)

Suppose $B = \{x_1, x_2, x_3, x_4, x_5, x_6, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6\}$ and

$$\begin{aligned} P \;=\; & (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_1 \vee x_2 \vee \bar{x}_5) \wedge (x_3 \vee x_4 \vee x_5) \wedge \\ & (x_4 \vee x_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_6) \wedge \\ & (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6) \wedge (x_3 \vee \bar{x}_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \\ & (x_2 \vee x_5 \vee \bar{x}_6) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_2 \vee x_3 \vee x_6) \wedge \\ & (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee x_6) \wedge \\ & (\bar{x}_2 \vee x_3 \vee \bar{x}_6) \wedge (x_2 \vee x_5 \vee x_6) \wedge (x_3 \vee x_5 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_6) \wedge \\ & (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \end{aligned}$$

In fact, only $\{x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 0\}$ satisfies $P$.

## Intuitive Intro to Computational Complexity (10/12)

Given an arbitrary instance $P$, we can always determine whether there is an assignment of Boolean variables such that $P = 1$.

To do so, we just have to substitute each and every possible value of $x_i$, $i \in \{1, \dots n\}$ in $P$. However, this procedure would require us to produce $2^n$ different assignments, i.e. an *exponential* number of different assignments.

## Intuitive Intro to Computational Complexity (11/12)

Given an arbitrary instance $P$ of $K$ variables and $m$ clauses, we can always determine whether there is an assignment of Boolean variables such that $P = 1$.

We just have to substitute each and every possible value of $x_i$, $i \in \{1, \dots n\}$ in $P$. However, this procedure would require us to produce $2^n$ different assignments, i.e. an **exponential** number of different assignments.

| Row number | Value Assigned to $x_n x_{n-1} \dots x_2 x_1$ | Evaluation of instance $P$ $K$ variables/clause, $m$ clauses |
|---|---|---|
| $1$ | $00 \dots 00$ | P(00 . . . 00) |
| $2$ | $00 \dots 01$ | P(00 . . . 01) |
| $3$ | $00 \dots 10$ | P(00 . . . 10) |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $2^{n-1}$ | $11 \dots 10$ | P(11 . . . 10) |
| $2^n$ | $11 \dots 11$ | P(11 . . . 11) |

## Intuitive Intro to Computational Complexity (12/12)

For example, suppose that $n = 1000$ (remember, $n$ is the number of binary variables) and that a computer running the above-mentioned brute force algorithm produces an assignment in $1fs$, i.e. $10^{-15}s$ ( a fast computer, indeed!)

Since $2^{1000} \approx 1.0715086072{\times}10^{301}$ then the total amount of time to be invested in running our brute force algorithm would be about

$$1.0715086072 \times 10^{301} \times 10^{-15}s = 1.0715086072 \times 10^{286}s$$

i.e., about

$3.39 \times 10^{278}$ years!

## Algorithm Analysis

Algorithm analysis is divided into two parts:

- Algorithm Correctness.
- Algorithm resource consumption.

## Algorithm Correctness

An algorithm $A$ is correct if:

a) [Design step] $A$ successfully does what it is designed to do. This step usually involves using solid mathematical statements (e.g., theorems).

b) [Implementation step] Implementation of $A$ is robust and free of bugs.

**Of course, easier said than done!**

## Resource consumption

Algorithm analysis also requires thinking about the amount of resources that an algorithm will consume during its execution. More precisely,

We need to understand/estimate how resource requirements will **scale** as input size increases.

## Resource consumption

**By Complexity Measure we mean a definition of efficiency that is platform-independent, instance-independent and of predictive value with respect to increasing input sizes.**

## Resource consumption

A measure based on specific implementation details like computer hardware instruction sets (e.g., Intel or AMD processors) or the primitives of a particular computer language (e.g., Python or Lisp) would produce, for one and the same algorithm, different output values for different computer platforms.

## Resource consumption

Consequently, we will formulate abstract descriptions to outline algorithm behavior.

The key concept here is, as you may expect, the notion of **elementary step**.

## Resource consumption

We shall analyze algorithm performance in two different scenarios:

– Worst case complexity. We are interested in estimating an upper bound on the maximum number of elementary steps that an algorithm must carry out. This is particularly important for pathological inputs.

– Average case complexity. We are interested in estimating the expected number of elementary steps that an algorithm must typically carry out. This is useful to estimate how an algorithm should perform ‘in practice’.

In both cases, complexity if a function of input size $|x|$.

## Worst-case complexity

**Def. Worst-case complexity.** Let $D_n$ be the set of inputs of size $n$ for the problem under consideration and $I \in D_n$.

Also, let $t(I)$ be the number of elementary steps performed by the algorithm on input $I$. We define the function $W$ by

$$W = \max\{t(I) | I \in D_n\}$$

$W(n)$ is the maximum number of basic operations performed by the algorithm on *any* input of size $n$.

## Average-case complexity

**Def. Average-case complexity** Let $D_n$ be the set of inputs of size $n$ for the problem under consideration and $I \in D_n$.

Also, let $t(I)$ be the number of elementary steps performed by the algorithm on input $I$ and $p(I)$ be the probability that input $I$ occurs. We define the function $A$ by

$$A = \sum_{I \in D_n} p(I)t(I)$$

## Big $O$ notation

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 $S_m$.

## Big $O$ notation

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 formalizes this idea.

## Big $O$ notation

**Def. Big $O$ Notation.**
Let $f, g : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. We say that $f(n) = O(g(n))$ if $\exists\ \alpha, n_o \in \mathbb{N}$ such that $\forall\ n \geq n_o$

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

In other words, $f(n) = O(g(n)) \Leftrightarrow g$ is an upper bound of $f$ for large $n$.

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

## Big $O$ notation

**Example.** Consider an algorithm whose running time is, $\forall\ n > 1$ and $p, q, r \in \mathbb{R}^+$,

$$T(n) = pn^2 + qn + r$$

We can see that

$$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2 = O(n^2)$$

Exercise: what are the values of $n_0$ and $\alpha$?

## Big $O$ notation

**Exercises I.** Prove the following statements:

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

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

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

## Big $O$ notation

**Exercises II.**
$O(n)$ for matrix multiplication?
What about a brute-force algorithm for K-SAT (n variables, m clauses)?

## Big $O$ notation

**Theorem.**

a) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.
b) If $f(n) = O(h(n))$ and $g(n) = O(h(n))$ then $f(n) + g(n) = O(h(n))$
c) Let $k \in \mathbb{N}$ and $f_1, f_2, \ldots f_k, h$ be functions so that $\forall\ i \in \{1, 2, \ldots k\}$ $f_i(n) = O(h(n))$. Then $\sum_{i=1}^{k} f_i(n) = O(h(n))$.

Proof. Left as an exercise to the student ☺

## $\Omega$ notation

**Def. $\Omega$ Notation.**
Let $f, g : \mathbb{N} \rightarrow \mathbb{R}^{+} \cup \{0\}$. We say that $f(n) = \Omega(g(n))$ if $\exists\ \alpha, n_o \in \mathbb{N}$ such that $\forall\ n \geq n_o$

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

In other words, $f(n) = \Omega(g(n)) \Leftrightarrow g$ is a lower bound of $f$ for large $n$.

## $\Theta$ notation

**Def. $\Theta$ Notation.**
Let $f, g : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. We say that $f(n) = \Theta(g(n))$ if $\exists$ $\alpha_1, \alpha_2, n_o \in \mathbb{N}$ such that $\forall\ n \geq n_o$

$$\alpha_1 g(n) \leq f(n) \leq \alpha_2 g(n)$$

Note that this definition is equivalent to saying the following

a) $\alpha_1 g(n) \leq f(n) \Leftrightarrow f(n) \geq \alpha_1 g(n) \Leftrightarrow f(n) = \Omega(g(n))$
b) $b)\ f(n) \leq \alpha_2 g(n) \Leftrightarrow f(n) = O(g(n))$

In other words,

$$f(n) = \Theta(g(n)) \Leftrightarrow [f(n) = \Omega(g(n)) \text{ and } f(n) = O(g(n))]$$

## $\Theta$ notation

**Theorem.**

Let $a_i \in \mathbb{R}^+, i \in \{0, 1, 2, \ldots, k\}$ and $p(n) = \sum_{i=0}^{k} a_i n^i$, i.e. $p(n)$ is a polynomial of degree $k$ in $n$. Then

$$p(n) = \Theta(n^k)$$