UAH

Computational Thinking: A Welcome Addition to the LOMLOE

Máster Universitario Formación del Profesorado de E.S.O., Bachillerato, F.P. y Enseñanza de Idiomas. Especialidad en Matemáticas.

Presentado por: D. Pablo Collado Soto

Dirigido por: Dr. Pedro Ramos Alonso

Alcalá de Henares, a 11 de septiembre de 2023

I only know that I know nothing.

Socrates

Contents

Abstract

If there is one thing we cannot deny it is how digital computing has revolutionised how we interact with the world around us. Everything from communicating with loved ones to operating on our bank accounts has been transformed in what appears to be a relentless drove towards an ever more connected world. However, the proportion of individuals who actually understand the ideas underpinning this revolution is minuscule.

In an effort to promote the understanding of these underlying concepts, several countries have included *Computational Thinking* (CT) in their curricula: Spain has followed suit with the latest and current teaching law; the LOMLOE. In this thesis we strive to explain how CT is much more than 'knowing how to use computers' whilst shining a light on the intricate relation between computational thinking and mathematics. Spain is not the first country to mention CT in its national education regulation: other countries have beaten us to this milestone. That is why we can look at how the different strategies panned out to learn from both mistakes and successful implementations. We devote part of the ensuing discussion to analysing different curricula in East Asia and the European Union to that effect.

One of CT's most appealing features is how transversal it is: the lessons derived from it can be applied to a myriad of fields. In the same fashion as with other areas of knowledge, the synergy between mathematics and computation is so evident and potentially beneficial that it is high time we explored it and included it in our national curriculum. In an effort to land the topic at hand we also provide examples of how CT can be leveraged from a mathematics class.

Keywords: Computational Thinking, Mathematics, Teaching.

Introduction and Background

> We may say most aptly that the Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.
>
> *Ada Lovelace (1843)*
> From (Wikiquote, 2023a)

It is no surprise computational thinking (CT) is deeply intertwined with computing and computer science as a discipline. What might come as a surprise however is the fact that computing as a blurry concept has been around much longer than one would possibly assume. In order to properly frame the concept of computational thinking it is compulsory to look back and analyse how computing as a field came into existence. This exercise will allow us to understand how CT was shaped through history until it became what we understand CT for today. This section is based on the first three chapters of (Denning & Tedre, 2019).

Ancient history

Precursors of computation date back to the Babylonians who, around 1800 BCE, wrote down several procedures for solving simple problems. This idea of having a 'recipe' describing how to generate a meaningful output from a given input can be regarded as a simplified version of what we now consider *algorithms* to be. It was Muhammad ibn Mūsā al-Khwārizmī who is considered the father of the algorithm as we know it today. Around 800 CE he offered some examples of these 'procedures', such as, for instance, computing the greatest common denominator of two given numbers.

Computational thinking's strong relation with computer science is in part due to the study of algorithms. However, we shall not forget CT is much broader than that. At any rate, a prime example of what can be achieved by studying algorithms has been bestowed upon as in the form of *Euclid's Algorithm*. It shows an efficient way of computing the greatest common divisor (GCD) of two given numbers by exploiting the liner relation between dividend, divisor, quotient and remainder. The algorithm exploits the fact that the GCD of two numbers is the same as the GCD of the smaller number and the remainder of dividing the larger by the smaller one. In other words:

$$\gcd(a,\ b) = \gcd\left(b, a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor\right);\ a,\ b \in \mathbb{N}$$

The idea of exploiting a fact like the one above to make a given procedure more efficient is a paradigmatic example of one of the key aspects of CT. On top of that, this example showcases the type of relation between mathematics and computing in which the latter *exploits*

the former so as to solve problems in a more efficient and, frankly, more elegant fashion.

Aside from freezing useful procedures in time, there was also a strong desire both for making generic computations (as in arithmetic operations) faster and error-free. Humans have a tendency of getting wrong results, specially when tasks are boring and repetitive: exactly the case with arithmetic. An initial solution to a subset of this speed and error problem came in the way of *analog computers*. Instead of harnessing the power of discrete states in a way current computers do so, analog computers rely on physical laws to offer results to rather specific problems. A prime example of these analog computers are Greek orreries, used to compute the position of planets around 100 BCE. These artifacts had several concentric rings which, when aligned in given patters, allowed the user to discover the desired positions. This specific application also showcases one of the underlying concepts sustaining much of CT: computing can be regarded as the process of taking some input data, transforming it and generating a desired output.

Calculus needs computing

Mathematical geniuses the size of Newton and Leibniz developed calculus around 1680. Even though we are brushing off a vast body of formalism with our affirmation, it can be regarded as a way of 'looking' at the infinitely small and local in such a way that all these local contributions add up to larger, more noticeable changes. Theory proves this intuition to be completely correct, but without efficient ways of both defining the procedures we need to carry out and a way of effectively doing so the theory has no real applications: it is limited as soon as symbolic calculus runs into, for instance, functions with no primitive.

Calculus soon gave way to the differential equations that model a myriad of physical phenomena. As the number of dimensions grew, new methods had to be devised so that these equations remained solvable and applicable to reality. A simple-yet-powerful strategy relies on overlaying a 'grid' on the domain of the functions (think $\mathbb{R}^2$) whose derivatives were members of the differential equations. If one were capable of computing the value of a function on the grid, it would be possible to solve these equations so that they would yield a result. Both the strategy of leveraging grids and studying local behaviours and values of functions as well as the need to accurately do so are deeply related to CT. The former is a prime example of the kind of strategies (divide and conquer) CT exploits whilst the other explains how the need of means of efficient computing grew ever more apparent.

Logic and algebra meet

As procedures became more complex, the need of 'choosing' what to do next became a necessity. However, if we want to minimise the errors arising from computations, how can we deal with choice? We need something akin to a language of thought that is unequivocal so that, even in the hand of laymen, algorithms allow their wielder to arrive at a correct result. This task was tackled by George Boole in the beginning of the nineteenth century in his book:

'An Investigation on the Laws of Thought' (Boole, 1854). In it, Boole describes a series of axioms on which he builds a way of leveraging algebra to formalise reasoning procedures. Put simply, Boole managed to express the operations of the mind through algebra, a feat whose implications have been (and continue to be) tremendously profound.

Boolean Algebra

In order to keep the discussion brief we can limit ourselves to the particularisation of Boolean algebra leveraged by computers. In general, Boolean algebra is a *6-tuple* containing:

- A set A.
- Two binary operations:
  1. **Conjunction**: *AND* ($\wedge$).
  2. **Disjunction**: *OR* ($\vee$).
- A unary operation, the **complement**: *NO* ($\neg$).
- Two elements: 0 ('floor') and 1 ('ceiling'). Note $0,\ 1\ \in A$.

Elements of A can be manipulated according to four axioms. Let $x,\ y,\ z\ \in\ A$, a Boolean Algebra verifies:

1. **Commutativity**: $x \wedge y = y \wedge x$ y $x \vee y = y \vee x$.
2. **Identity**: $x \wedge 1 = x$ y $x \vee 0 = x$.
3. **Distributivity**: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.
4. **Invertibility**: $x \vee \neg x = 1$ and $x \wedge \neg x = 0$.

These, in turn, let us prove several theorems such as the ones below. A more comprehensive collection can be found on ("Boole Algebra Theorems", n.d.).

1. **Idempotence**: $x \wedge x = x$ y $x \vee x = x$.
2. **Nullity**: $x \wedge 0 = 0$ y $x \vee 1 = 1$.
3. **De Morgan**: $\neg(x \wedge y) = \neg x \vee \neg y$ y $\neg(x \vee y) = \neg x \wedge \neg y$.

The above formalism explains how and why the 'operations of the mind' work the way they do. If we apply them to two elements $x,\ y\ \in\ A$ we can generate the *Truth Tables* 1 underpinning digital electronics.

It was Claude Shannon in his thesis (Shannon, 1940) who showed how Boolean Algebra could model telephone circuits and, more importantly, electric circuits in general. This comes

x	y	$x \wedge y$	$x \vee y$	$\neg x$	$x \oplus y$	$-(x \wedge y)$
0	0	0	0	1	0	1
0	1	0	1	1	1	1
1	0	0	1	0	1	1
1	1	1	1	0	0	0

Table 1
Truth Table of the elementary logic operations.

to show how the mathematical foundation on which computing as a filed is built on is quite dense. It is then no surprise that CT is deeply intertwined with mathematics in the sense that, just like math, it concerns itself with modeling and then working symbolically with whatever it is we are representing. This insight was one of the main driving forces behind the so called 'computer revolution' that has come to shape our daily lives.

The first computing machines

We have already stated how orreries are a full-fledged example of a computer. It is, however, not an analog computer by any means. In order to make arithmetic operations easier, faster and less error prone, several physical objects have been developed through the ages to aid human computers. A clear example would be the slide rule which, exploiting the property of logarithms that states that $\log(p \cdot q) = \log(p) + \log(q)$, turned multiplications into a series of additions whose results were obtained by physically manipulating the ruler. In a sense, the very exploitation of the aforementioned property is in itself a display of CT in its own right: we are representing arithmetic operations by a marked object and a series of displacements.

Even though the slide rule had a more general applicability than the Greek orrery, these two artifacts still called for some sort of human interaction. Tools such as the slide ruler were leveraged by teams of, literally, human computers to generate books crawling with tables containing results for well-known functions at different points. The idea went that whoever wanted to particularise an expression could look the values up so that no complicated and convoluted computations were necessary. However, these books were chock full of errors which, as a man named Charles Babbage put it, could wreck havoc when used for building civil infrastructure and charting out courses for ships. This led him to devise and then request the funding for what he named the *Difference Engine*. This machine would leverage mechanical components such as gears to automatically compute tables like the ones that were being generated by easily-bored and error-prone humans. A bit too revolutionary for his time, Babbage ran into the reality of the day's technology: mechanical pieces meeting the necessary standards could not be manufactured at the time. This, along with an unstable funding from Great Britain's government led him to redesign the original *Difference Engine*. With that, the *Analytical Engine*, capable of computing any function, was born. Just like today's computers would be of no use had nobody wrote software for them, Babbage knew the same was true for his new conscription. He

partnered up with a truly gifted mathematician who is also considered by many to be the first programmer in history: Ada Lovelace.

Instead of restricting the applicability of the *Analytical Engine* to computations over real quantities, Ada saw the true potential of such a design. In writing the programs she began to understand how the *Analytical Engine* could act on arbitrary symbols whose meaning could be altered at will. Given the machine could compute any symbol that also implied it could apply any transformation to its input. Thus, Ada regarded the *Analytical Engine* as an *information machine* instead of a glorified general purpose calculator. This point of view, quite revolutionary at the time, is the cornerstone of the ubiquity of today's computers. All in all, they concern themselves with transforming *symbols* which, in turn, can be modeled as elements of a set.

Even though the *Analytical Engine* would never be completed due to the rudimentary technology at the time (steam was the basis of technological developments), the ideas behind the design paved the way for the birth of the digital computer some 80 years later. These machines dated around 1820 and 1840 for the *Difference Engine* and the *Analytical Engine*, respectively, were the physical realizations of theoretical models of computing.

Theoretical general machines

If there is something computing as a field has had to fight for, it is being considered a science in its own right. This reality worked against the expansion of computer science departments and, implicitly, the spread of CT and its entry in school curricula. However, computing received a formal mathematical framework from the hand of both Alan Turing and Alonzo Church.

A-Machines

Alan Turing described what he called *a-machines* (i.e. *Automatic Machines*) in an effort to prove the *Decision Problem* (Hilbert & Ackermann, 1999) had no general solution. These machines were later named *Turing Machines* (TM) in a review by his advisor, Alonzo Church. These Turing Machines provide a mathematical model of computation that, at the same time, shines a light on the limitations of computers. This is a crucial point to make: Computational Thinking has been no stranger to hype. However, its limitations are not only rooted in the technology exploited by today's computers; it is an intrinsic part of computation.

Turing machines can be (and have been) built. Their input is an infinitely long tape partitioned into equal spaces with symbols belonging to an alphabet. At each step in the computation, the machine is associated with a given state and its head is on top of one of the tape's partitions so that it can query the inscribed symbol. Based on those two pieces of information it will transition to a new state, overwrite the current symbol by a new one and move the tape either to the left or to the right. At some point, the machine will reach a final state on which it will halt and cease the computation: the result can be read back from the current contents on the tape.

Turing Machines can be defined as a 7-tuple (Hopcroft & Ullman, 1979) M where $M = \langle Q, \Gamma, b, \Sigma, \delta, q_o, F \rangle$. Bear in mind the definition of the Cartesian Product (i.e. $\times$) for two sets A and B is given by $A \times B = \{(a, b) \, / a \in A \text{ and } b \in B\}$. In other words, $A \times B$ is a set containing all the ordered tuples one can generate from elements in both sets.

1. Γ: A finite and non-empty set of *symbols*. In other words, this is the machine's *alphabet*.
2. $b \in \Gamma$: The blank symbol which can appear in the tape infinitely.
3. $\Sigma \subseteq \Gamma \setminus \{b\}$: The set of *input symbols* (i.e. the ones initially on the tape.
4. Q: A finite and non-empty set of *states*.
5. $q_o \in Q$: The machine's *initial state*.
6. $F \subseteq Q$: The set of acceptable *final states*.
7. $\delta \ : \ (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$: The (possibly partial) *transition function*. This function maps all the ordered pairs of both the current state and the current symbol on the tape's head to all the ordered 3-tuples describing the possible next state, symbol to write and the direction in which to move the tape (either to the left (i.e. L) or to the right (i.e. R). The fact that the current state must not be one of the final states implies once the machine reaches a final state it halts and thus, there is no need to define transitions *from* those states. What is more, if the function is not defined for a given state, the machine halts.

This 7-tuple is usually succinctly represented either as a table defining δ or as a directed graph as seen on figure 1. The possibility of being able to model the concept of a Turing Machine in several fashions is also a prime example of Computational Thinking: different ways of representing data come with different advantages and weak aspects.

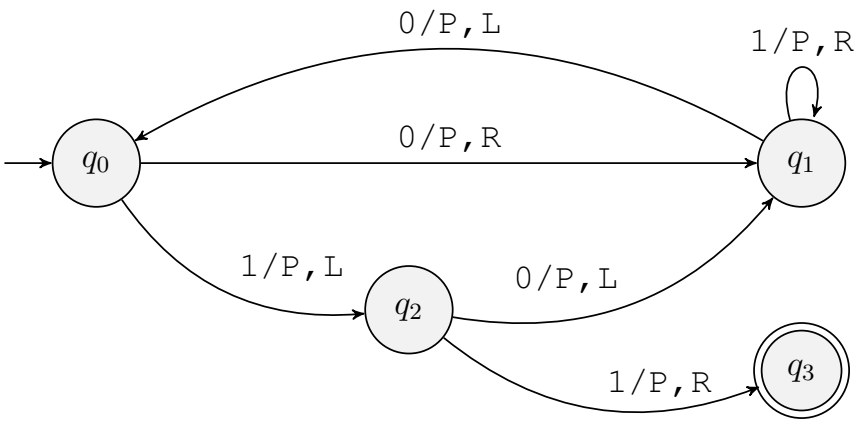


Figure 1
A graph representation of a Turing Machine implementing a Busy Beaver (Queiroz, 2010).

Now, where are these Turing Machines applicable? To begin with, they let us *reason* about computing. Given they are capable of executing any conceivable computing operation, being *Turing Complete* implies being able to replicate any behaviour a Turing Machine can exhibit. In other words, given an input a Turing Machine device can replicate the output generated by an arbitrary Turing Machine or, even more generally, a Turing Machine can emulate any other. Being Turing Complete is a feature of most (if not all) programming languages: even LaTeX, the system we are typesetting this document with, is Turing Complete. Some people even try to implement solutions with it... (Hicks, 2009) Demonstrating a system to be Turing Complete usually involves emulating a Turing Machine within said system as being able to do so implies being able to replicate whatever a Turing Machine is capable of.

The concept of replicating a Turing Machine within a Turing Complete system hints at the idea of *recursiveness* that permeates computing and some extremely interesting algorithms and data structures. The idea of recursiveness explicitly manifests itself in factorials for instance. We can define a factorial in terms of itself $n! = n \cdot (n-1)!;\ 0! = 1$. Another prime example is the very well known Fibonacci sequence which can be expressed as $F_n = F_{n-1} + F_{n-2};\ F_0 = 0;\ F_1 = 1$. These examples again showcase how the links between computation as a field and mathematics are much deeper than one would initially assume. In fact, another definition for meeting Turing Completeness relies on the system under test to be able to express any μ-recursive (which are all mapping from $\mathbb{N}^k \to \mathbb{N};\ k \in \mathbb{N}$) function (Dean, 2023).

At any rate, we can confidently state how the foundations provided by Turing Machines not only infuse a great deal of formality to the field of computing: it also provides a way to reason about computing as a discipline. With that we can establish boundaries separating what we can from what we cannot achieve with computers as ideal machines and, at the same time, learn how the very basis of computing shapes the way we think about the field from a more generalist point of view. In our precise scenario this translates to understanding how computational thinking can exploit the ideas and concepts derived from this fashion and truly groundbreaking way of analysing computers. However, the complexity behind all this can make Turing Machines too unwieldy for a direct transition to the classroom even if we are not interested in being extremely formal in our discussion. Given we run the risk of missing the forest for the trees, we believe we can still leverage the bulk of the essence behind this framework if we take a step back and study other simpler model of computation.

Finite State Machines

Turing Machines model the most general (and hence the most powerful) computers: they are capable of computing any function. If we somehow restrict what the Turing Machine is capable of, we can arrive at simpler models of computation that still offer a great deal of insight into computing as a discipline. Unlike Turing Machines, *Finite State Machines* (FSMs) do not produce any explicit output. They consume symbols from an input tape that moves in

a single direction through the machine's life-cycle. At each step, a symbol is read back from the input tape and, by also taking into account the current state, the transition function dictates what state to transition to. The fact that there is no explicit input does not imply this model of computation has nothing to offer: the final state the FSM arrives at is an indicator of whether the input is well-formed or not. This explains why FSMs find a myriad of uses in fields such as compilers, whose main task is to receive an arbitrary sequence of symbols (supposedly) adhering to a set of syntax rules. If the input is well-formed, the FSM will terminate in a valid state, but if it is ill-formed, the FSM will try to carry out an invalid transition. Then, depending on whether all the transitions are well formed or not we can decide whether the input is well formed or not.

Just like with TMs, we can succinctly define FSMs as a 5-tuple $F = \langle \Sigma, Q, q_o, \delta, F \rangle$ where:

1. Σ: The *input alphabet*, that is, the set of symbols presented to the FSM.
2. Q: A finite and non-empty set of *states*.
3. $q_o \in Q$: The machine's *initial state*.
4. $F \subseteq Q$: The set of *final states*.
5. $\delta : Q \times \Sigma \to Q$: The (possibly partial) transition function. A FSM machine transitions from a given state (i.e. q_t) to the next one (i.e. q_{t+1}) based on the current input symbol $s \in \Sigma$ and q_t. In some more complex models of computation δ can be non-deterministic. In this scenario we would be talking about transition probabilities, which can be rather useful in statistics. Actually, *Markov Chains* (Kuntz, 2022) can be modeled as FSMs where δ is completely probabilistic.

Just like with TMs, FSMs can be graphically represented by a directed graph where each vertex or node represents a state and each edge or link represents a transition. Figure 2 showcases the definition of a FSM capable of deciding whether a given input number (encoded in *binary)* is even or not. Even binary numbers must end with a 0, so that is the only input transitioning into the final acceptable state represented on the figure's right hand side.

It is undeniable how this model of computation while less complex still offers a myriad of options for goading the student's computational thinking through tasks such as designing FSMs that meet certain criteria. We will discuss an example of such an activity later in the document.

λ-*calculus*

When discussing TMs we mentioned that Alan Turing's advisor, Alonzo Church, was the one to actually name TMs when working as Turing's thesis advisor. Church is also responsible for developing a completely different model of computation in the 1930s: λ-calculus.

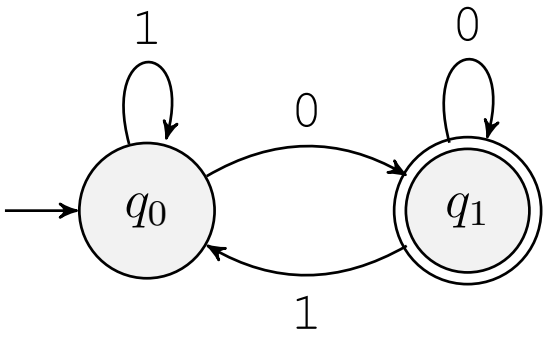


Figure 2
A graph representing a FSM capable of deciding whetheher a binary number is even or odd.

Instead of trying to model a 'real machine', λ-calculus takes a function-based approach making heavy use of abstraction. In its purest form λ-calculus concerns itself with the definition of λ-terms (i.e. *names* and *functions*) and their transformation. Not surprisingly, λ-calculus is *Turing Complete*: it can express any computable function. Its more abstract nature is usually a better fit for mathematicians who are not concerned with the 'real' implementations of computing machines in the least, something TMs capture and express much more clearly. The fact λ-calculus is Turing Complete implies it is, in itself, a full-fledged programming language that could theoretically be used to express any possible computation. The ideas and design behind λ-calculus are behind the paradigm of *functional programming* and it has also inspired the design of programming languages such as *Haskell* (Marlow et al., 2010). A great introduction on the topic is provided on Rojas (2015). In fact we will base the following (short) discussion on said paper.

The basic 'building block' of λ-calculus are functions. For example, the *identity function* (i.e. $f(x) = x$ would be defined as $\lambda x.x$. λ-calculus only relies on two symbols: λ signals a function definition and the dot (i.e. .) separates the arguments (on its left) from the function's *body*. These functions can be applied to arbitrary (yet valid) expressions which can themselves be functions. For instance, we can apply the identity function to y by writing $(\lambda x.x)\, y \rightarrow y$. The arrow is read as 'reduces to'. Of course, $\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$: the function's argument is just a 'placeholder'. Bear in mind that λ-calculus can only deal with functions accepting a single parameter. If one wants to define a function taking more than one parameter, we would have to resort to defining a function with a single argument returning another function taking a new parameter and so on and so forth. For instance, assuming $+$ represents the addition function and a^b represents the exponentiation function we can define a function $\lambda xy.x^2+y \equiv f(x,y) = x^2+y$. However, this would not be valid: our lambda function accepts two arguments. We can rework the expression into the following: $\lambda xy.x^2 + y \equiv \lambda x.\,(\lambda y.x^2 + y)$. The application of such a function would be carried out as:

$$f\,(1,2) \equiv \left(\lambda xy.x^2 + y\right)(1,2) \equiv \left(\lambda x.\left(\lambda y.x^2 + y\right)(1)\right)(2) \to \left(\lambda y.1^2 + y\right) 2 \to 1^2{+}2 \to 3$$

Bear in mind that, due to the simpler notation we will define functions accepting more than one argument: we can make the claim that for any function accepting n arguments we can define a set of n functions that, when curried together, are equivalent to the former. This means we can exploit the lighter notation without losing a trace of generality.

Now, what does λ-calculus offer us? Among its strongest points is capturing the idea of abstraction in a rather succinct manner by pushing the definition of 'function' to its limits. A prime example of that would be how $\mathbb{N}$ is defined within λ-calculus. Given we must work with the 'limitation' of **only** dealing with functions, we somehow need to construct $\mathbb{N}$ in terms of functions. We can exploit the idea of function composition (i.e. $f = g \circ h$). Then, 0 will be represented by a 0-composition, that is, the absence of application of a given function s. In the same fashion, 1 is represented by a single application of s. This strategy for representing natural numbers is known as *Church's Encoding*. Table 2 contains the definition for $n \in \{0, 1, 2, 3\}$

0	$\lambda sz.z$
1	$\lambda sz.s\,(z)$
2	$\lambda sz.s\,(s\,(z))$
3	$\lambda sz.s\,(s\,(s\,(z)))$

Table 2
Church Encoding of the first few natural numbers.

In order for this representation to be useful we must be able to manipulate it somehow. Just like we have done before, we must define a higher-order function capable of acting on the encoding of the natural numbers. A prime example of such kind of functions would be the *successor function* which, as its name implies, increments a given expression passed as an argument:

$$S \equiv \lambda nab.a\,(nab)\,;S\;0 \equiv (\lambda nab.a\,(nab))\,(\lambda sz.z) \to$$
$$\lambda ab.a\,((\lambda sz.z)\,ab) \to \lambda ab.ab \equiv \lambda ab.a\,(b) \equiv \lambda sz.s\,(z) \equiv 1$$

One might ask how can $\mathbb{R}$ be constructed. If we combine several natural numbers, we can define an algorithm for describing real numbers just like the `IEEE-754` ("IEEE Standard for Floating-Point Arithmetic", 2019) standard accomplishes it, the same goes for $\mathbb{C}$: we can define it in terms of tuples of elements in $\mathbb{R}$. We can then be convinced that, just like we stated before, λ-calculus can be leveraged to define any computable function without imposing limitations on what we can represent: we can limit ourselves to the purely symbolic realm knowing we will not run into any limitations should we want to particularise any definition.

Aside from arithmetic, it is also extremely useful to somehow represent logic or 'truthy'

values. Just like we introduced before when discussing Boolean Algebra, these values let us *reason* about propositions, allowing us to, for example, control the execution flow of an algorithm expressed as a function. Like before, we must define functions representing these truthy values as seen on table 3.

True $\equiv T$	$\lambda xy.x$
False $\equiv F$	$\lambda xy.y$

Table 3
Representation of truthy values in λ-calculus.

We believe it is extremely interesting to notice how $0 \equiv F$: this is a convention that is honoured in several programming languages such as the ever-pervasive `C` and whose reason of being is rooted in the very basis of λ-calculus. Just like in Boolean Algebra, we can define a series of logical operators as functions. For instance, the `AND` operator (i.e. $\wedge$) can be defined as $\wedge \equiv \lambda xy.xyF \equiv \lambda xy\,(\lambda xy.y)$. If we apply it to values TT and TF we can reduce the expression to find out how, as expected, the former application reduces to T and the latter to F:

$$\begin{aligned}
\wedge TT &\equiv (\lambda xy.xy\,(\lambda xy.y))\,(\lambda xy.x)\,(\lambda xy.x) \\
&\to (\lambda y.\,(\lambda xy.x)\,y\,(\lambda xy.y))\,(\lambda xy.x) \\
&\to (\lambda y.y)\,(\lambda xy.x) \to \lambda xy.x \equiv T
\end{aligned}$$

$$\begin{aligned}
\wedge TF &\equiv (\lambda xy.xy\,(\lambda xy.y))\,(\lambda xy.x)\,(\lambda xy.y) \\
&\to (\lambda y.\,(\lambda xy.y)\,y\,(\lambda xy.y))\,(\lambda xy.x) \\
&\to (\lambda y.\,(\lambda ab.b))\,(\lambda xy.x) \to \lambda ab.b \\
&\equiv \lambda xy.y \equiv F
\end{aligned}$$

Following a similar reasoning we could build the truth tables presented on table 1, thus showing how we can reason in a completely analogous manner solely by exploiting λ-calculus. Just like with Turing Machines, we believe the level of abstraction required to bring these contents to the class are well beyond the scope of the K-12 curriculum in Spain. However, the lessons learned from how λ-calculus is structured and built can be easily transferred to other areas of mathematics that are of paramount importance in a high school setting. A clear example would be the definition of functions. Where a substantial amount of pupils regard functions such as $f\,(x) = x$ as a mere 'equation', we can delve a bit deeper into what a function really means by stressing how it is 'nothing more' than a 'mapping' between two sets that *usually happen to be* $\mathbb{R}$. This would also allow discussions around the *domain* and *image* of functions to be much more grounded and meaningful whilst also breaking down the ever cryptic $f : \mathbb{R} \to \mathbb{R}$ notation into something not only understandable, but also wieldy.

Pseudocode

In the Spanish curriculum the study of formal logic is relayed to the subject of philosophy. However, we have already seen how formal and purely mathematical systems and frameworks such as λ-calculus can also offer a great tool set to delve into the mechanisms of the mind; the same is true for Boolean Algebra. Nonetheless, these formalisms can be a bit cumbersome to work with and, in their complexity, shine the light away from the essence of what we are trying to comprehend.

In order to avoid the pitfalls described above we can turn to other ways of expression such as *pseudocode*. Through pseudocode we can describe algorithms and processes of reasoning in an informal-yet-exact manner. A prime example of that would be the proof by contradiction (i.e. *ad absurdum*) showing how the so called *halting problem* (Strachey, 1965) is *undecidable*; that is, no general algorithm can provide a definitive and correct answer.

The halting problem poses the question of whether one can define an algorithm capable of stating whether another algorithm passed as an argument will terminate or not. In other words, the question can be stated as: does an algorithm capable of stating whether any algorithm will terminate or loop forever exist?

The (informal) proof that such an algorithm does not exist relies on the fact that one can write a program doing exactly the opposite of what the 'analysis algorithm' states. It is undeniable that this discussion with its abstract nature is rather difficult to express with the written word. However, what if we expressed our line of reasoning through a loosely structured set of words? Listing 1 sketches the algorithm proving the halting problem is undecidable.

Procedure 1 Proof of the *Halting Problem* being undecidable.

Require: Let procedure $T[R]$ return **true** if procedure R terminates and **false** otherwise.

```
procedure P(T)
    if T[P] ≡ true then
        loop
            Do nothing                    ▷ Infinite loop even though T[P] ≡ true !
        end loop
    end if
    return                                ▷ Procedure P finishes despite T[P] ≡ false !
end procedure
```

Now, what is the advantage of relying on loosely-yet-strict language such as pseudocode for tackling logic problems? Instead of offering a completely abstract language like, for example, λ-calculus we instead bring the process of reasoning a step closer to a reality pupils are more familiar with. Even though we would by no means consider the knowledge of computing concepts widespread, they are much more concrete in essence than philosophical constructs such as the *modus ponens*.

Contemporary Computing

After the introduction of Turing's Machines, the development of computing began being more concerned with the engineering aspect of it all. In the middle of the 1900s the rise of semiconductors and related technologies threw the field of computing into a breakneck growth spiral that culminated in the tremendously powerful computing machines we all carry in our pockets.

Even though this topic is extremely interesting, the truth is it does not offer us much more insight into where computing meets mathematics which is what this document focuses on. That is why this section ends with the (brief) exposition of the theoretical computing models underpinning computing in a formal fashion.

However, we must bear in mind that the rise in both importance and impact of computing as a field prompted its emancipation in the form of a new discipline in the 1950s and 1960s. This move was met with the clear opposition of other areas of knowledge such as mathematics and biology which believed computing would not qualify as a field given its lack of a clear 'subject of study'. Nonetheless, both fields (and many others) came to embrace computing with time.

Armed with the context provided in this section we devote ourselves to tackling the concept of Computational Thinking in the next section. Given the elusive nature of CT's definition we cannot dismiss the importance of knowing where its roots stem from: looking back is the only way of knowing where to move to.

Computational Thinking

> An algorithm is a set of rules for getting a specific output from a specific input. Each step must be so precisely defined that it can be translated into computer language and executed by machine.
>
> *Donald Knuth (1997)*
> From (Denning & Tedre, 2019)

The previous section has given us a general outlook of how computing came to be as well as the main frameworks and theories providing a rich tapestry on which to build. However, the question still remains: what is *Computational Thinking*? This section is devoted to providing out view by drawing from our background and influences whilst taking into account the myriad of definitions that exist even today.

Existing definitions

A plethora of definitions exist trying to capture the essence of CT. In this document we present four definitions representing milestones in the development of CT in a chronological fashion.

These four definitions are just a sample (albeit of the most influential) of the many definitions of CT. Given the novelty of CT some definitions do not agree with others whilst some definitions differ from others in slight subtleties.

In exploring all these options (Lodi & Martini, 2021) provides a superb collection of CT definitions as well as a comparison of the most influential. We cannot encourage the reader enough to go through said article should the definitions of CT presented here be deemed too shallow.

After poring through several definitions and, together with our experience as technical students, we have also decided to offer our own definition of CT based on what we believe are the most characteristic features of CT as identified by other authors.

Seymour Papert

The first appearance of the term CT on a printed work was by *Seymour Papert* (Papert, 1980). However, this appearance was no attempt at a definition: 'Their [of people using computers for providing mathematically rich activities] visions of how to integrate computational thinking into everyday life was insufficiently developed.' (Lodi & Martini, 2021).

The idea set forth by Papert hinged on leveraging computers as a tool for making the abstract concrete. He designed the `LOGO` programming language to make it easier for students to 'play' around with concepts in an effort to make them more understandable.

Papert worked hand in hand with renowned development psychologist *Jean Piaget*, the father of *constructivism* as a framework explaining how we learn. Piaget defended the idea of making pupils active in their learning in the sense that knowledge should not be presented and imported 'as-is'. Instead, the pupil must accommodate his or her mental structures to accept the new concepts and ideas so that these are incrementally enhanced in a logical and cohesive fashion. Papert took Piaget's framework a bit further, thus creating his own proposal in the process: *constructionism*. In it, Paper argued that learning is potentiated not only by altering and reordering mental constructs, but also by actively building meaningful objects aiding in the understanding of such concepts. For Papert, `LOGO` would be the tools with which to build these objects.

However, Papert's work has often been misquoted to support the claim that programming as a skill enhances the way students think. Later on, Papert made it clear that was not his idea at all (Lodi & Martini, 2021). Despite this reality, it is clear how Papert's CT, albeit not a definition 'per-se', states that programming in itself is not what shapes and enhances the way pupils think: programming is a malleable tool empowering pupils' understanding and learning in other areas (Lodi & Martini, 2021). By offering students a 'virtual playground' they are free to experiment, predict and build their own meaning and thus work towards a deeper understanding of the world that surrounds them.

Jeannette M. Wing

Despite an uptake in popularity in the late 1900s, CT's penetration into K-12 schools and beyond lost most of its momentum. Instead of exploring all the assets CT could provide, most curricula defaulted to focusing on *digital literacy*, neglecting deeper and more transferable concepts and strategies in the process.

In 2006 Jeannette M. Wing published a short and thought-provoking essay (Wing, 2006) which reignited the flame of CT and prompted teaching authorities to include it in curricula whilst revisiting what CT was in the first place. This essay offered a multidimensional definition of CT whose bottom line is that CT is a universal skill that should not be confined to the realm of computing and computer science: everyone can benefit from it.

Among other aspects, Wing emphasizes 'CT is not programming': she defends it is a much further reaching concept. In her argumentation she states 'CT is not thinking like a computer'; instead, it is 'thinking like a computer scientist'. Given the regular tasks and challenges a computer scientist faces, this ability can be regarded as a skillful management of abstraction. As Dijkstra (Dijkstra, 1974) put it, it is the capability of developing a 'mental zoom lens' letting one focus in and out of a subject so that complexity can be managed in a sustainable manner. This focus on abstraction is very characteristic of both Wing's definition of CT and of those definitions inspired by this essay.

Wing also focuses on how CT is a 'fundamental skill'. Instead of being something constrained and relegated to the area of computer science, it is something people in many other

fields can make use of and exploit. A prime example of that would be the tremendous growth in popularity of the so called *computational sciences* (Denning & Tedre, 2019). These are nothing more than the application of computing practices to traditional fields such as biology and physics.

In stating CT is no equivalent to 'thinking like a computer', Wing goes even further and defends CT is all about humans. The techniques and ideas underlying CT allow clever and inefficient humans to harness the power of 'dumb' and quick computers to solve problems we considered unsolvable beforehand.

In providing a clear argument for the contents of this document, Wing explicitly states there is a clear relation between both engineering and mathematics. The connection to the latter stems from the formal and comprehensive mathematical foundations of computing we visited in the previous section. This also comes to support Wing's claim that CT is by definition transversal in that it appears and cooperates with other fields not only of science, but also of human knowledge.

It is worth pointing out how we have not discussed either software of hardware when trying to define CT. That is because, as Wing defends, CT is not about the 'artifacts' we generate, it is about the ideas underlying their design. In being more general, these concepts can find applications in many other fields and thus, enjoy a more widespread impact than their 'fossilised' counterparts.

Despite discussing a plethora of advantages and assets of CT, Wing does not provide a clear-cut definition of CT, instead choosing to refer to it as a skill set comprising all the above. However, this lack of a explicit and succinct definition is by no means demeaning: Wing's contribution has been the spark that ignited the new push for bringing CT to K-12 schools.

Denning & Tedre

Denning and Tendre's book (Denning & Tedre, 2019) has been a treasure trove of information for the preparation of this document. Aside from offering a cohesive overview of the history of computing and how that shaped CT through the ages, they also propose their own definition. Instead of relying on an enumeration of characteristics of CT, the authors define CT as a two-dimensional mental skill set. The first dimension has to do with the *design* of computations so that we can harness the power of computers. The latter is devoted to *explanation* of the world as complex *information process*.

Given the clear differentiation among these two 'sides of the same coin', they go a step further and associate each dimension with a different field of knowledge. For them, the design of computations is closer to the area of engineering where one must be fully aware of the limitations and restrictions imposed by the reality we live in. On the other hand, the explanation of information processes is much more abstract in nature. The term *information process* reminds us of the ideas of Claude Shannon we already discussed in the previous section. They also state that, even though these two dimensions could be completely independent from

each other, in practice they are deeply intertwined. A reason behind this combination can be found in how the explanation of computations requires knowledge of how they are actually carried out. The argument for the other direction of the statement can be made by explaining how the design of any given machine is restricted and constrained by the purpose it must fulfil: we cannot design machines without knowing what they are for!

It is also worth mentioning how the authors, just like Wing, explicitly stress that CT 'is not programming': it is much more than that! Despite the fact that programming makes heavy use of CT concepts, we must convince ourselves that believing CT is only activated when programming is equivalent to limiting ourselves to a very small portion of what CT has to offer.

After providing a succinct and clear definition of CT, Denning and Tendre also break CT down into six clear aspects characteristic of CT:

1. **Methods**: This category devotes itself with procedures (i.e. algorithms) and how mathematicians and engineers developed them so that even non-experts could leverage them. This called for a clear and unambiguous definition of the steps to follow. This can be regarded as the development and subsequent definition of algorithms.

2. **Machines**: This aspect concerns itself with developing the physical computers (i.e. the hardware). Prime examples of individuals involved in this dimension are Babbage and Lovelace with their *Analytical Engine* as well as the massive teams of engineers that made the digital computer being used to write this document a reality. Given its concern with the fabrication and constraints of reality for making computers, this aspect of CT is not of much interest from a mathematical point of view.

3. **Computing Education**: The authors consider the codification of computing and its ways of thinking (i.e. CT) an integral part of CT as well. This aspect offers us a history-minded view of CT explaining how it came to be.

4. **Software engineering**: Errors in software have caused spectacular disasters. The fact that software has not been considered a 'real' engineering discipline prompted the appearance of a field of computing devoted exclusively to applying the ways of engineering (capable of making tremendously secure planes) to software. Like machines, this aspect is not of much interest from a mathematical point of view.

5. **Design**: Closely related to the previous aspect, Denning and Tendre argue that designing good computations which are usable by others involves a great deal of design-oriented expertise which is inseparable from CT. Given the current state of the software job market we can only agree with them.

6. **Computational science**: Despite not being taken seriously in their beginning by other established disciplines, computing has shown it merits its own area of knowledge. The

implications of the second dimension of Denning and Tendre's definition of CT are quite convoluted and deserve their own space for being pondered and developed. It is worth pointing out how Wing also mentioned how there were a myriad of intellectually challenging problems in the realm of computing. Also, it should be explicitly stated how the study of these problems involves a great deal of mathematical rigor and apparatus, making this aspect especially absorbing from a mathematical perspective.

The authors explain how each of these six aspects behaves like a 'window looking at CT'. As specified in the enumeration, some of them provide a clearer value from a mathematical perspective. However, it is undeniable that this open-minded approach at CT unites the engineering and mathematical dimensions of computing together with the social and moral concerns inherent to the design of computations. We believe this is extremely important given the age we live in, where huge multinational companies have been caught employing morally questionable practices in an attempt to increase revenue.

All in all, it is clear how Denning and Tendre take the essence of Wing's initial definition of CT and then push its boundaries to create a comprehensive and clearer definition of what CT is.

Belén Palop

Belén Palop is a researcher from the *University of Valladolid* and the leading expert in CT in Spain. She spoke at a conference in 2022 (Belén Palop, 2022a) where she offered her definition (together with Irene Díaz). The original definition is given in Spanish.

Palop and Díaz define CT to be 'a way of reasoning allowing us to tackle a problem over some input data so that a computer can solve it' (Belén Palop, 2022a).

This definition is clearly related with the previous two in the sense that there is an explicit reference to CT being a 'way of thinking', which, put simply, is practically equivalent to referring to a skill set. However, we consider this definition to be less ambitious than the one provided by both Denning and Tendre and by Wing. This definition restricts itself to a clear scenario on which we have to deal with a problem based on some data. Even though it is true they explicitly state the problem is to be solved in such a way that a computer can also reach a conclusion, they neglect the abstraction aspect of CT.

After delving into the idea of λ-calculus and experiencing how it deals with abstraction in a strict mathematical sense in such an elegant manner, we believe acknowledging this dimension of CT is nothing short of an asset, especially from a mathematical point of view such as the one this document adopts.

However, even though it is not explicitly present in the definition, Palop goes on to explain how abstraction is a clear aspect of CT. The fact that this is not taken into account in the definition must be considered, but she is quite clear that CT is a multi-dimensional skill set. Thus, we come full circle once more in that this definition of CT is clearly stated but then relies on a deeper explanation outlining its many subtleties.

Our definition

After the previous discussion it is clear how CT is quite an elusive concept. Many authors have tried to succinctly capture the essence of what CT really is and, maybe more importantly, each of them has focused on different dimensions of what CT is understood to imply in our time. If there is a keyword identifying what CT comprises it would be no other than **dimension**. Just like we can think about abstract spaces in mathematics which in turn can be described through the linear combination of a series of dimensions, the same is somewhat applicable to CT. In a sense, each author scales each dimension in accordance with what they think best defines and captures the basics of CT. Just like in a regular space, this dimensions can be scaled by a null quantity, thus removing them from the overall 'mix'. A prime example of that would be Palop's definition in which CT is understood as a narrower and more tightly defined skill set than what Denning and Tendre offer.

Given the monstruous amount of effort put into the matter of defining CT by academics with a much broader and deeper understanding of the topic, it might even come across as 'disrespectful' to try and offer our own view on the matter. Even though this document pertains CT from a mathematical and pedagogical point of view, we are lucky enough to be able to draw on the expertise acquired by working in the techonological industry with a close relation to CT 'in practice'. Furthermore, we can consider ourselves 'digital natives' in the sense that computers have been an integral part of reality for as long as we have had a chance to work and interact with them. This implies CT has been a central-yet-unseen force exerted in our academic development from a young age. We believe all this offers a rich tapestry we can base our own definition on.

Aside from personal experience it is of the utmost importance to leverage the foundations laid before us by more knowledgeable people. After poring over literature we have found Denning and Tendre's defintion to be quite precise and extensible. More specifically, the idea that the entire world can be regarded as a terribly complex information process and that CT is devoted to understanding it all really resonated with us. This is the same line of thinking Ada Lovelace followed when realising how Babagge's Analytical Engine was a much more powerful construct than initially thought. It is also the driving force behind the recent 'explosion' in the application of computing through simulations in a myriad of fields so that predictions can be derived from well known systems. A prime example of that would be the simulation of mechanical systems by the mere application of Newton's laws which completely define the system's evolution through time. Like Denning and Tendre put it, this dimension of CT is very science-oriented in its method. This opens up several avenues for studying, formalising and applying CT in educational contexts which is why we believe it is much more applicable to K-12 classrooms. A clear instance of how this set of ideas can be leveraged would be the simulation of a given process by applying a set of rules laid out in the shape of temporal equations. This is completely equivalent to the evolution of a mechanical system we described from a conceptual

perspective: we can just drop the formalism (i.e. vector formulation and differential equations) so as not to lose sight of why we are bringing this kind of problems and situations to the classroom.

Even though the following example is well beyond the scope of the K-12 curriculum, the 'information process' dimension of CT is beautifully exemplified in the langragian formulation of classical mechanics. Where Newton set forth a set of rules in the form of equations governing how massive objects interact with one another, Lagrange decided to condense all that knowledge in a single expression. Given both formulations 'encode' the same information one might wonder why one is preferred over the other: after all, Newton's formulation can account for classical behaviours without a problem. The answer lies in how difficult Newton's equations are to handle under normal circumstances like changes in frames of reference. Newton's very well known first law showcases the relation between the force exerted on a body and the second derivative of its position with respect to time (i.e. its acceleration). On the other hand, Lagrange condenses the behaviour of the system in a integral that must be minimised. This expression can later be boiled down to a pair of partial derivatives allowing us to simplify expressions and describe a system in a much more wieldly and succinct manner. The following equation compares both approaches. Bear in mind $x = x(t)$ (i.e. $x : \mathbb{R} \to \mathbb{R}$ is a function of time) and that the lagrangian of the system is $L = L\left(x,\ \dot{x}\right)$ (i.e. a function of x and the time derivative of x which is denoted by $\dot{x} = \frac{dx}{dt}$):

$$F = m \cdot \ddot{x}$$
$$\frac{\partial L}{\partial x} = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right)$$

As we stated before, these expressions will never be brought to a K-12 classroom, but it is quite easy to see how different 'ways' of expressing the evolution of a system affect how easy it is to actually leverage the information for real applications. This is something that can easily be exploited in a classroom by, for instance, going over the summation of an arithmetic series and why expressing the addition in different fashions has value other than 'making things harder'.

The second dimension of CT proposed by Denning and Tendre also invites us to revisit well established concepts in a new light. An example of that would be regarding integrals as functions. Limiting the discussion to definite integrals, once students have come to terms with their definition it can be argued that $D : \mathbb{R} \times \mathbb{R} \times \{f \ / \ f : \mathbb{R} \to \mathbb{R}\} \to \mathbb{R}$ is a function representing a definite integral. Given the previous notation can be obscure at best, we can also define $D = D(a, b, f)$ (albeit in a physicist's style) where $a, b \in \mathbb{R}$ and $f : \mathbb{R} \to \mathbb{R}$. Of course, $D = \int_a^b f(x)dx$ too! This all comes to state how interpreting well known concepts as transformations of information offers new perspectives which, in turn, lead to more meaningful

insights. This is exactly one of the reasons why exploiting CT in a mathematical setting can be tremendously worthwhile.

Aside from Denning and Tendre's view on what CT really is, we cannot help but also mention Wing's own view. This second proposal gravitates around the foundation that CT is in itself a transversal skill set that can be applied to other areas such as mathematics. Her essay stresses the importance of abstraction whilst also providing a series of key aspects of what CT is. Even though it is true there is no clear cut definition, distilling the essay allows the reader to grasp that CT is a quite far reaching concept. What we find especially interesting in this essay is both the focus on abstraction as well as the clear portrait of CT as an interdisciplinary and 'foundational' skill set.

The management of abstraction Wing aludes to can be easily found in the math classroom. We can resort to basic arithmetic to understand how multiplication is nothing more than a generalization of repeated sums. Thus, in multiplying two numbers we are actually 'abstracting the repeated sum away'. Does that mean we should not be concerned with sums any more? No! Abstractions can come at the cost of a loss of generality: products can only express repeated additions but they do not generalise addition as an arithmetic operation. Understanding this caveat is at the core of managing abstraction: it is not a 'fire and forget' approach; we must always be sure higher levels of abstraction remain correct in any given context. What is more, given products 'encode' repeated additions we can resort to additions to prove the different properties of the product. An equivalent increase of abstraction takes place when defining powers. A more advanced example would be the definitions of functions as 'hard coded' expressions (i.e. $f(x) = x^2$) to then regard them as simple 'mappings' from one set to another. This comes to show how this increase in abstraction allows us to manage the tremendous complexities of mathematics without giving up any generality.

Given we are working on a rather limited application of CT just limiting ourselves to the mathematics classroom we would rather propose a smaller scope definition such as Palop's in an effort to avoid hyping CT up (more on this issue in the next subsection). Put together, we believe CT comprises a set of skills aimed at managing abstraction, representing information and regarding mathematical procedures as the definition of transformations of that information.

If there is something clear cut about CT is that it is extremely hard of pinning down and that, like many other 'fuzzy' concepts, it is easier to state what CT is not rather than what it actually is. This leads us to the final subsections of this section in which we will try to address some obstacles in the way of CT's uptake as well as some false promises it has to face.

Computing Literacy is not CT

The definitions of CT provided above do have something in common: none of them is directly tied to computers. That is, computers are not needed to exercise and leverage CT in a classroom. This follows from how CT is defined to be a 'skill set' with, as Wing put it, transversal applicability. Despite how 'silly' it may seem, one of CT's biggest problems is the

leading C making up its name which stands for Computing.

Nowadays computers and smartphones are part of our daily life. Given (pretty much) everybody knows how to use one we could say our society is digitally literate. That is, we all know how to use computers and smartphones to solve whatever tasks we might have to use them for. Now, does using a computer involve CT? Not really. If we try, for instance, to write a document such as this one we might have to grapple with fonts and typefaces after configuring the margins of the page, but what CT offers is not really activated. Maybe some users do regard the confection of a document as an information process converting a series of natural numbers representing letters into a full-fledged digital data structure we would usually call a PDF, but that will not really be the case in the vast majority of scenarios.

Given the time when IT lessons began appearing in Spanish schools, the initial focus on those lessons was on learning how to use the new tools offered by computers. Curriculum designers believed one would not be able to understand or appreciate the why of computers without knowing what they were for and how to actually use them. This explains why all references to computing were restricted to the technology and IT classroom: CT was sometimes erroneously taken to be what we now call Computing Literacy and, as this document argues, they have pretty much nothing in common. This is true even to the extent that we believe CT can be leveraged without any computing literacy.

Like many other skills, computing literacy can be attained at different levels. The shallowest levels devote themselves to using basic features to accomplish tasks, but as this literacy grows ever more deeply one can argue CT begins coming into focus. 'Under the hood' components in a computer such as its operating system are a marvel of modern engineering that make extensive use of CT in terms of how they manage complexity not only in their design, but also in their use by others. A very interesting exercise to try and grasp this management of abstraction would be to ask oneself what a 'file' really is. Put simply, it is a 'construct' of the operating system usually represented by an integer that indexes a table kept in a file system referencing blocks of raw data that are then assembled into the contents we stored through a series of instructions provided by the filesystem and the OS itself. However, the end user just sees a pretty icon they can click on! This management of abstraction is at the core of what CT offers, but it is true it sits on the engineering side of things instead of the mathematical side. Of course, this level of abstraction is seldom visited in K-12 schools.

Then, what is the advantage of bringing computing literacy to the classroom? It was intended to be a stepping stone in a path to CT and a deeper understanding of computing and what it has to offer. However, the reality is most of these courses focused on learning how to use text processors and image manipulation programs whilst missing out on a ton of opportunities for delving into other areas of computing where CT could have made quite a dent. Now, should they be removed? Given the current use patterns of technology, smartphones are getting the upper hand. The surprising outcome of all this is that newer generations are having a hard time whenever they have to use a 'real' computer for everyday tasks such as writing a document

in a word processor. Given the simplicity of the smartphone's computing model some of the literacy we had already attained is being lost to ever simpler applications. This supports the claim that digital literacy still needs to be a part of the curriculum, albeit in a more modern fashion.

The fact that computing literacy has little to do with CT does not imply we cannot leverage tools dealt with in computing literacy courses for CT-oriented purposes. An example of that would be data manipulation programs such as Microsoft Excel. Learning how to use Excel does not involve CT, but using Excel to do statistics analysis on data does, at least to some extent. A brilliant example of how one can make use of Excel to exercise CT is offered by Belén Palop (Belén Palop, 2022b). The exercise revolves around generating a set of random numbers based on the Excel function `rand()`. This function generates normally-distributed random numbers in the $[0, 1)$ interval. With it, one can try to compute random numbers belonging to other ranges by translating and scaling the interval through additions and products. In thinking how to accomplish that one would be exercising CT in a crystal clear fashion! In any case, this simple-yet-complete exercise would be characterised by nobody as an exercise in computing literacy.

Framing the discussion in terms of the mathematics classroom, computing literacy has little to offer given it never actually activates CT in a meaningful manner. It is however crucial to make this distinction explicit so that these two are never conflated.

Programming is not CT

Even though this reality has been tangentially described, it is necessary to explicitly state that CT is not programming. At least, they are in no way interchangeable. A clear example of that is offered by Papert: he devised programming as the way of expressing one's thoughts and ideas in an effort to understand them at a deeper and more conceptual level. It is true a competent programmer must draw from CT's aspects such as abstraction management to solve problems in a fast, efficient and correct fashion, but in no way is CT exclusively tied to computers and, therefore, to programming languages. For instance, Church developed the foundations of λ-calculus before computers were a reality!

Why is then programming so often conflated with CT? Our take is that programming is still deemed as an obscure ability reserved for a lucky and talented few. What is more, programming is touted as the 'ultimate ability' in our increasingly technocentric society: the paradox is a very slim percentage of the population actually knows what programming involves! The fact that the term Computing appears in CT prompts the layman to 'put everything in the same box' and therefore equate CT with programming. To make matters worse, programming finds application in areas such as robotics whose results are rather 'flashy'. This makes CT a perfect candidate for marketing-induced hyping: if one can program, one can control robots and therefore change the world, or so the line of reasoning goes.

The reality is thinking one's ability to program empowers her or him to do anything is

a naïve approach to knowledge. As Denning and Tendre put it in their book, a programmer will never be able to design and implement a fluid mechanics simulator if he or she has no knowledge about fluid mechanics and all the complex differential equations governing them. Just like being able to write does not make anyone a novelist, being able to program will not make anybody a physicist or mechanical engineer! However, this reality is not something of interest to a society in desperate need of an ever increasing number of programmers: it is much better to tout programming as the 'ultimate ability' in order to get more students and, in turn, more programmers in a huge job market with a demand that cannot be satisfied. This reality has other side effects that have little to do with math in K-12 schools such as the need for a barrier of entry that needs to be lowered to allow for more people to jump in. This is taking abstraction to such heights that people are forgetting about the reality of computers and, what is worse, spending much more energy on computations that could be carried out with other much more efficient algorithms. This is the direct consequence of loosing touch with the realities of the underlying hardware, which comes to show how crucial it is to never let an abstraction become our ground truth. Even though being aware of the electrons travelling through transistors is not necessary for performing daily tasks, knowing this is what underpins computers allows us to know what they can accomplish and what is beyond their reach. What is more, this also allows us to decide what is the best way to carry our a given task. In other words, abstraction is a welcome tool for managing complexity as long as it does not prevent us from knowing there is something 'underneath'.

All in all, just like with computing literacy a clear line has to be established separating programming from CT. It is true that the overlap between programming and CT is much larger than the one between computing literacy and CT. However, learning how to program is not going to make anybody an expert in the application of CT. This is why curricula focused on 'programming robots' as a means of developing CT are victims to an overinflated expectation of what programming is and what it can accomplish from a pedagogical point of view.

Technology's and CT's Hype

This document has been written in the summer of 2023. If one were to condense what this year has been about in a couple of words, the terms *extreme heat* and *AI* are compulsory. For the sake of the contents of this document the second one is of much more interest.

Generative Artificial Intelligence models such as the GPT-X family created by OpenAI have been all the rage. This document will not delve into the massive copyright infringement and outright theft of intellectual property providing their foundation and it will also overlook their ridiculously large impact on the climate due to their power consumption. It will instead look into what makes them 'tick'.

As Arthur C. Clark put it: 'Any sufficiently advanced technology is indistinguishable from magic' (Wikiquote, 2023b). This quote prompts the idea that relying on advanced technology implies some kind of dogmatic belief in it; even more so if the mechanisms behind said

technology are unknown to the user. To make matters worse, these models are intentionally opaque: the companies developing them want to retain an edge over the competition to make ever increasing amounts of money.

These 'magic tricks' that are generative models are very elaborate artefacts capable of 'fooling' anybody: even more so those who want to be fooled. As Bender et al. put it, these models behave like 'stochastic parrots' (Bender et al., 2021) capable of generating correlated textual data without any sense of meaning. In other words, these models do not generate correct answers, they generate the most likely one. What happens when a generative model generates a wrong answer? If you are the company behind it, you call it a 'hallucination' to anthropomorphise the model and avoid any responsibility for its consequences. If you are a user that is none the wiser, you assume the answer is correct of course. A prime example of this reality is how Google advised its own staff not to use code generated by its own generative AI (i.e. Bard) as well as not to disclose confidential information (Dastin & Tong, 2023). If the very makers of these models do not trust them for critical tasks how come the general population relies on them to such an extent? The only thing setting these two sets apart is that the former knows how these models work and what their pitfalls are; the latter are just subject to a series of advertisements and articles touting AI's superb capabilities.

CT can help us understand as a society the foundations of constructs such as generative models not to discard them, but to understand how and why they work (at least to an elementary level of complexity) to then judge whether we should rely on them and how we can use them. Knowing how they are trained it is much easier to protect ourselves against fake news for instance: we know that if more and more fake news are 'fed' to the algorithm, chances are these will be more likely to influence its answers for instance. We need to understand these models as information-transformation processes to be ready for what their use entails.

In knowing what is 'going on behind the scenes' we can also try to fight against the voices saying that teachers ought to be replaced by this kind of technology. Even though this line of reasoning conflates information with knowledge and wisdom whilst at the same time showing contempt for teachers, it is clear it is prey of a complete lack of understanding of what a generative AI really is. The way of changing this harsh reality is through the inclusing of CT in K-12 curricula in several subjects so that newer generations are ready to face the music of a society overly reliant on a technology it does not fully understand.

This 'hype' inherent to technology has had the adverse impact of a general belief that CT is somewhat like a 'superpower'. That has turned CT into a buzzword that is usually thrown around without much understanding. This, in turn, wraps CT in a halo of misconceptions, prompting incorrect definitions and outright lies. It is of the utmost importance to be aware of this reality so that one can steer clear of these 'false promises' when reading literature and reading information on CT in general media.

Computational Thinking in Other Countries

> The question of whether Machines Can Think... is about as relevant as the question of whether Submarines Can Swim.
>
> *Edsger W. Dijkstra*
> From (Dijkstra, 1984)

Computational thinking has already found its way into K-12 curricula around the world with varying degrees of success and ambition. This section goes over existing curricula in an effort to offer an overview of where CT stands in general education.

Aside from that, a brief discussion into some of the main challenges facing CT considering the Spanish educational system together with a small sample of interesting articles and papers is presented so as to make the extension of the presented information easier.

CT in the US

The main driving force behind CT in the US was Wing's essay published in 2006. Her defense of CT as a transversal skill set provided a huge amount of momentum to CT and it managed to rid CT of the computer literacy principles shackling it down. This explains how CT found its way into the curriculum of different states. This document has chosen to focus on the state of Virginia as an example.

The mathematics curriculum (Virginia Department of Education, 2016) explicitly mentions computing, albeit not in the 'regular mathematics' courses. Those courses devoted to general mathematics (i.e. the ones taught in Spain) devote themselves to topics and concepts equivalent to those visited in the Spanish curriculum. However, the overall curriculum goes on to define 'extended courses', two of which are quite interesting from the CT point of view.

On the one hand, the curriculum sets forth a *Computer Mathematics* course beginning on page 33. However, even though the initial description discusses solving problems that can be modelled mathematically most of the elements are concerned with the engineering side of computing: even the design of graphical interfaces is mentioned! Aside from explicit references to the design of algorithms for solving a problem as well as the mention of applying computation to the resolution of practical mathematical problems little else is directly related to mathematics. One could consider the implementation of predefined sorting algorithms such as bubble sort a mathematical exercise to some extent as long as the students do not limit themselves to only translating the algorithm without trying to understand the rationale behind it. A creative teacher might find ways to relate curriculum elements such as the description of a program in an abstract way with more mathematically-oriented concepts such as the definition of functions (i.e. by explicitly talking about types of data and linking it to the definition of a function as a

mapping of sets) as well as the thorough testing of a program so that an appropriate set of data is 'fed' to it. Nonetheless, it is clear how these types of activities cater more to an engineering and technical course rather than to one focused on the mathematics underpinning computer science.

This mathematics curriculum has been updated as of 2022 to include a *Data Science* course whose definition begins on page 40. According to the elements in the curriculum, this course provides a hands-on application of statistical concepts that are seldom visited in Spanish classrooms. Inviting students to 'select statistical models' and 'use goodness fits' as well as prompting them to 'prepare datasets for modelling and analysis' shows how this course tries to exercise the skills required by the ever more popular data science field whose demand for workers keep growing. This is, in our modest opinion, a great way of bringing statistics to the classroom in such a way that results are more 'tangible' than what is usually seen in a mathematics classroom. What is more, this type of tasks activate CT to a great extent given students are encouraged to represent and transform raw data in the most appropriate way whilst also crafting the most information-dense visualisations.

Even though the mathematics curriculum defines courses with a deep relation to computing and CT, it does not mention CT explicitly. CT is instead a part of the Computer Science curriculum (Virginia Department of Education, 2017), where a (gaseous) definition of CT is provided. The elements of the curriculum are much broader than those of mathematics, something that is not as surprising given the central role of mathematics whilst computer science is relegated to be a second-tier course. However, one can find explicit mentions to both the design and execution of algorithms along with mentions to abstract concepts such as recursion and the ideas of 'decoupling' a problem into smaller pieces.

Maybe paradoxically, even though CT is not explicitly mentioned in the mathematics curriculum the abilities it strives to work on do fall under the umbrella of CT. These skills are the subject of additional courses beyond the normative mathematical content, but they are treated in such a way that students can really benefit from how they are portrayed. At any rate the recent addition of courses such as Data Science show how the effects of Wing's essay are still a strong force shaping the curricula throughout the country.

CT in Singapore

Singapore is usually touted as a prime example of what a good mathematics curriculum can achieve in terms of education. During our Master's several subjects mentioned their way of teaching mathematics and some teachers even brought several of their strategies to the classroom. Simple ideas such as the 'bar model' show how an elegant solution can provide a powerful tool for teaching abstract concepts many students might find challenging. It is exactly this way of tackling the teaching of mathematics that makes Singapore's curriculum a compulsory read when it comes to researching how CT found its way into other teaching systems around the world.

Singapore's secondary education system can be a bit convoluted for newcomers. In order not to make this discussion overly complex, this document limits itself to the 'regular' stream for students with mid-tier rankings in the post-elementary school national tests. The syllabus for these courses can be found on (Singapore Ministry of Education, 2023). The *Mathematics* syllabus contains no references whatsoever to CT. Computational tools are mentioned when discussing e-learning where computers are portrayed as essential tools in the field of mathematics. The curriculum states that 'computational tools are essential for solving challenging problems' whilst also being essential for understanding concepts to a higher degree of complexity just like Papert defended in his original paper. Surprisingly enough, no clear examples are provided neither in the *Mathematics* nor in the *Advanced Mathematics* curriculum.

CT is explicitly mentioned in the *O-Level Computing* and the *Computer Application* syllabus. This document focuses on the former given the latter course is aimed at the engineering-side of computer science in which the focus is on how to apply computers to solve different tasks. This does indeed require the application of skills falling under the umbrella of CT, but it does not strive to explain and study computing from a more mathematically-inclined perspective.

The *O-level Computing* curriculum offers a definition of CT heavily inspired by Wing's 2006 essay. It argues that by reformulating a problem so that it is solvable by a computer, students are developing their CT. It also supports its claim that CT is fundamental by stating that other countries like the UK, the US, Australia and Hong Kong have included it in their respective curricula. In a fashion similar to Denning and Tendre's, the curriculum identifies three dimensions of CT where the one of more interest is the one treating computing as a science. Citing Wing's 2006 essay, the curriculum argues that CT is not about 'thinking like a computer'; it is about 'thinking how a computer can solve a problem'. What is more, the document offers a great and informative graph with the relation among all three relations in which CT is comprised of 'discrete mathematics', 'algorithmic thinking', 'abstraction', 'simulation' and some other keywords that have already appeared.

The curriculum advocates for the understanding of algorithms as a way of reflecting on CT as well as 'the application of formal reasoning and systems thinking in the analysis, design and implementation of computer solutions'. The preferred way of activating CT is through the development of simple programs for solving a set of challenges proposed by the teacher. A good example would be an exercise asking for the number of digits in any natural number, something that can be easily solved by noticing how the number of digits N_d of a natural number n is given by $N_d = \lfloor 1 + \log(n) \rfloor$.

All in all, Singapore does mention CT and does a great job of defining what CT is so as to strip CT of any hyped-up meaning. It does however relay CT to the computer science courses but it does not shy away from the fundamentals underpinning computer science in any way, even up to the point of mentioning discrete mathematics, something we have not seen in any other curriculum.

CT in the UK

This document limits itself to the curriculum for the UK's *Key Stages 3 and 4* which are the equivalent to Spain's secondary school years. The national curriculum (UK Department for Education, 2014) offers the contents of each of the courses taught throughout these years.

The contents of the mathematics curriculum are pretty much equivalent to those we find in its Spanish counterpart. The focus is on general mathematical knowledge and there are no explicit references to CT whatsoever. Like in the previous cases, CT is indeed part of the curriculum, albeit under the umbrella of *Computing*.

It is important to note that *Computing* is a compulsory course during *Key Stage 3* (i.e. 11 to 14 year olds) and a foundation course during *Key Stage 4* (i.e. 14 to 16 year olds). In other words, *Computing* is an important component of the UK's curriculum. This course's curriculum mentions CT in its introduction. However, it does fall into the trap of hyping CT up when it states 'CT and creativity equip pupils to understand and change the world'. Despite this claim, the elements set forth in the document are quite ambitious and on point. For instance, at *Key Stage 3* students are encouraged to 'design, use and evaluate computational abstractions that model the state and behaviour of real world problems and physical systems'. This is practically equivalent to the second dimension of CT as defined by Denning and Tendre in which CT strives to regard real situations as information systems. It also mentions the understanding of algorithms such as sorting procedures (a classic example) together with an incursion into Boolean Algebra thorough the study of the basic logical operations (i.e. $\wedge$, $\vee$, $\neg$). It also requires the use of not one but two programming languages to solve computational problems such as the ones posed as examples by Singapore's curriculum. This is a key idea that highlights how programming languages are a tool and not an end in themselves (when solving problems). It clearly portrays how CT is not programming but you do need programming languages to define and turn algorithms and procedures into executable digital constructs.

The curriculum also includes some more broadly defined elements whilst also emphasising the correct and moral use of technology. Aside from these aspects closer to the engineering side of CT, the elements of the *Computing* curriculum do fall very near the contents of mathematics if these were to be a bit more broadly defined. This is, to some extent, similar to what happened in the US curriculum. The difference is that, in the US' curriculum, some CT-oriented courses did fall under the umbrella of mathematics instead of being associated with computing on its own. However, one might argue that the fact that *Computing* is a course in itself in the British curriculum implies a great deal of importance is bestowed upon it (it is compulsory after all) so that it does not necessarily have to be associated with either technology or mathematics: it is a branch of knowledge in its own, albeit one deeply related to the previous two.

In any case, CT is a clear and important part of the UK's curriculum even if it does not directly belong to mathematics.

CT in Spain

When inspecting the Spanish National Curriculum contents (i.e. RD 217/2022) (Boletín Oficial del Estado, 2022) CT comes up in several subjects. Bear in mind this discussion restricts itself to compulsory education so that the different curriculums are comparable.

CT is briefly mentioned in *Biology* as a skill to be applied in the interpretation of results and the resolution of problems posed by biological processes. Even though it is true this idea could be understood as a realisation of the second dimension of CT as defined by Denning and Tendre so that biological processes are understood as information systems that does not appear to be the case.

CT is explicitly mentioned in the introduction of the contents of the course on *Mathematics* where a short definition is also provided. This definition is by no means as complete as the one offered in Singapore's curriculum, but it does include terms like 'data analysis', 'logic organisation of data' and 'sequential solutions that can be executed by a programmable tool'. It also goes on to explain that due to 'organizational reasons' CT is classified under the *Algebraic Sense* but that it should permeate the entire subject. This follows Wing's claim that CT is a transversal skill.

CT is the key component of *Specific Criterion 4* where CT is defined in a more strict and clear fashion. Terms such as 'abstraction', 'decomposition', 'patterns' and 'algorithms' permeate the criterion's description. Despite that, it comes across as a 'hollow' description of what CT really is and the lack of examples does not allow the reader to get a grasp on how to apply CT to the mathematics classroom in a clear-fashion.

CT is also mentioned in the different *Evaluation Criteria* throughout the document. Some of these criteria specify the 'computational interpretation' to be an assessable item whilst others discuss 'strategies for the interpretation and modification of algorithms' as well as the 'generalisation and transference of problem-solving processes' together with the 'computational interpretation of problems'. Again, these fall flat due to the lack of a down-to-earth specific definition despite them mentioning key aspects of CT. In other words, mentioning CT without providing examples of how it can be applied (like Singapore's curriculum) or a more specific context of how to activate it (should a programming language be leveraged?) makes the entry of CT in classrooms a hurdle-riddled affair.

Aside from the previous two, CT is also mentioned extensively in the subject of *Technology*: even a whole 'block' is devoted in part to CT. Aside from mentioning CT's relation to algorithms and automation, little else information is provided on how to apply it in the classroom.

In light of the above we can conclude the Spanish curriculum falls prey to CT's hype: it permeates the curriculum (even *Biology*!) but no clear directives are offered when it comes to applying it in a real situation. Sadly, this is the case with most of the curriculum as it stands.

What is more, the curriculum requires the design of what it calls *Learning Situations*:

no clear example is given on what some of these could be... Luckily, Belén Palop's talk (Belén Palop, 2022b) offers a great deal of examples on how these could be designed according to her view of what the curriculum dictates: again, no clear directives are offered! An example of such an activity could be the randomness-oriented exercise discussed in the previous section as well as the explanation of *numbering systems* including the somewhat well-known binary system.

CT's hurdles in Spain

Including CT in the mathematics curriculum might seem like a questionable decision. Computer science entails quite a bit of mathematics. However, these concepts are not something every mathematician works with during their studies. What is more, the entire technological aspect of CT which is ultimately needed to fully leverage it in the classroom are by no means contents belonging to a degree in pure mathematics. Then, how should mathematics teachers grapple with CT?

The above is a real challenge CT runs into when reaching the schools. This reality calls for the continuous learning of teachers, something that requires both an effort on the side of the administration and the teacher him or herself. All in all we find a potential cause for friction in the smooth and seamless introduction into the classrooms. Should this not be addressed by the administration, CT would be left alone without any real support. Chances are teachers willing to give it a try and/or with the necessary expertise will actually dive into it whilst others just walk around it. This, in our opinion, calls for a large and clear effort on the side of the administration if CT is really a desired learning outcome and not a 'poster term' used to hollowly embellish the curriculum.

Aside from teacher's training CT faces yet another obstacle. Spanish public schools are not known, to put it mildly, for their superb infrastructure. Concerning CT, this means the availability of resources such as computer rooms are not something that can be taken for granted. This poses a set of management-oriented problems schools have to deal with if they do really want to make a bet on CT as a meaningful element in the curriculum.

Further reading

Despite the walk through samples of some country's curricula, there is still a vast amount of ground to cover when it comes to assessing CT's standing in a worldwide fashion. That is why this section offers some further references aiding the interested reader in continuing his or her quest for more information. Interesting options include (European Commission et al., 2016) which offers a great overview of CT in Europe whilst (Hsu, 2019) looks at the main challenges students, teachers and principals face when trying to introduce CT in schools. Even though this last article focuses on primary schools the lessons are broadly applicable in higher education. On the other hand (Kristensen, 2020) offers an overview of CT's standing in

Asia, and (Hickmott et al., 2018) distils a myriad of articles studying the links between CT and the learning of mathematics.

Bringing Computational Thinking to the Classroom

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> *John von Neumann (1951)*
> From (Wikiquote, 2023c)

Abstraction is an integral part of CT. However, this also works against its application in K-12 classrooms were even algebra poses a great obstacle for the younger generations. This section is devoted to sketching and justifying a few activity proposals striving to activate CT.

The current Spanish curriculum introduces the concept of *Learning Situations*. These large-scope activities should *activate* a series of *basic skills* so that learning takes place in a defined context. These activities can then be thought of as *Learning Situations*, at least to some extent.

The examples relying on code have been designed using the Go programming language (The Go Team, 2023). The choice is justified by how clear its syntax is together with the myriad of existing libraries making the implementation of even complex algorithms a completely attainable task. It might come as a surprise that the document steers clear of Python (Python Software Foundation, 2023). This decision is motivated exactly by how popular Python has become in a large portion of programming-related fields. This implies a large amount of freely accessible information is inaccurate or outright wrong. Working with a smaller language leveraged by a generally more technically-inclined community offers a better overall quality of resources and documentation. On top of that, Go is a *compiled* language meaning its efficiency, environmental impact and ease of deployment are all unmatched. It does call for some interaction with command-line interfaces, but it can be argued that is the case for every language (including Python) after an extremely rudimentary introduction period. In the end, Go offers a less cluttered and 'noisy' ecosystem in which to embrace programming as a tool for activating CT.

Finite State Machine Design

The idea for this activity is largely based on Belén Palop's talk (Belén Palop, 2022a). This section captures the essence of her idea whilst relating it to a deeper mathematical justification based on the concepts of the first section.

When discussing Turing Machines it was shown how they can be graphically represented by directed graphs. Aside from the graphical representation, graphs can also be symbolically defined as an ordered pair whose elements are two sets. A graph $G = (V, E)$ is composed by a set of *nodes* or *vertices* (V) and by a set of *edges* or *links* (E). Links belonging to E are themselves ordered pairs of members of V: they describe how the elements of V are

'connected'. Given this definition is specific for directed graphs, the order in which each element appears in each pair of E does matter. Let $a,\ b \in V$, a link $(a, b) \in E$ would represent an 'arrow' originating from a and terminating on b (i.e. its 'tip' would be pointing towards b). These edges can be extended with additional elements so that only the first two are elements of V and the latter ones are elements of arbitrary sets. This allows the modelling of parameters such as 'link costs' in telecommunication networks through scalars, for instance.

Given Turing Machines were a tad too complicated, the concept of Finite State Machines (FSMs) was also introduced. These can also be represented as directed graphs. For this particular application, elements of V represent the different FSM states (i.e. $V = Q$) and the edges represent the different state transitions (i.e. $E \sim \delta$). In order to 'fill the gap' of δ's definition we can add a third element to the edges which is in itself an element of Σ (i.e. the input alphabet) explicitly marking when this edge can be traversed based on the current input to the FSM. Thus, edges in this scenario are 3-tuples (a, b, c) where $a, b \in Q$ and $c \in \Sigma$.

These FSMs can encode and represent simple algorithms acting on an input alphabet Σ. Now, the exercise consists of proposing a series of input strings composed by elements $e \in \Sigma$ (i.e. the string can be regarded as a set s so that each element e of s belongs to Σ). The students should then design a FSM that **only accepts** these valid input strings but no others. In this scenario a FSM accepts an input string s if it makes it transition from the initial state q_o to the final state F. If for any $e \in s$ the FSM has no suitable transition, the presented input **is rejected**. Note we further restrict these FSMs so that F is an element of Q instead of a subset of Q (i.e. $F \in Q$ instead of $F \subseteq Q$) so as to make the exercise a bit simpler.

It goes without saying the input strings s and the target FSMs themselves should be kept simple so as not to be too challenging. The fact that this difficulty is very easy to control is a great feature of this activity: it can easily cater to a classroom's diversity.

Introducing binary numbers

Given the definition of the alphabet Σ must be simple, these FSMs lend themselves nicely to the introduction of the binary numbering system so that $\Sigma = \{0, 1\}$. Binary representations directly tie into how CT is concerned with the representation of information (i.e. $5_{\text{decimal}} = 101_{\text{binary}}$) and it allows some of the properties of natural numbers, such as being even or odd, to be spotted rather quickly. If a binary number b is even, its least significant bit is 0 and conversely, if it is odd, its least significant bit will be 1. This is exactly the idea underlying the graph presented on figure 2.

A good pedagogical sequence for introducing FSMs would be:

1. Propose the design of a FSM accepting vowels and solve it together with the class stressing the coherent use of symbols.

2. Propose the design of a FSM accepting consonants and let the class try to solve it in groups.

3. Propose the design of a FSM accepting two vowels back to back. Remember letters are presented individually to the FSM.

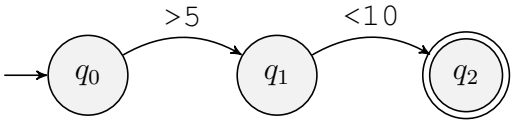
4. Propose the design of a FSM accepting two consecutive numbers where the first must be larger than 5 and the second lower then 10. A single number is presented to the FSM on each step.

5. Propose the design of a FSM accepting even binary numbers and solve it together with the class. Note this requires a prior introduction to the binary numbering system.

6. Propose the design of a FSM accepting odd binary numbers.

Figure 3 contains a sample solution for exercises 1 and 4, for instance.

(a) *Exercise 1.*



(b) *Exercise 4.*

Figure 3
Solutions to some of the proposed FSM exercises.

This activity not only activates CT's algorithmic-oriented side, but it also offers a great opportunity to delve into other topics such as binary numbers whilst also offering the possibility of stressing the importance of a correct formal notation in the representation of the different FSMs as graphs. Another very interesting aspect would be the verbal explanation of how the FSMs behave under a given input which can be both valid and invalid. This forces pupils to verbalise their ideas and allows the teacher to glimpse the pupil's thought and reasoning process, thus offering a chance to correct any misconceptions.

Applying Algorithms on Graphs

In the previous section graphs were formally defined and a few examples of areas of application were set forth. Modelling communication networks such as the Internet through graphs has allowed the design and application of complex algorithms accomplishing relevant tasks. A prime example of these types of algorithms would be *routing algorithms*. They allow for the discovery of *paths* 'telling us how to traverse the graph so as to get from node $a \in V$ to node $b \in V$ by traversing the available links E.

The Bridges of Königsberg

In 1736, Leonhard Euler showed how one could not go for a walk in the city of Königsberg and only cross each of its bridges exactly once (Euler, 1736). When he realised the actual outline of the city had no meaning in the context of the problem he abstracted it all away and represented each portion of land as a node in a graph and each bridge as an undirected edge. The city could be modelled with just 4 nodes and 7 edges. It turned out each node had 3 incident (i.e. connected) edges. Given one needs to 'leave' a node after 'entering' it, an even number of edges had to be incident on all the nodes except on two of them (i.e. the initial and starting ones). This condition was not fulfilled by the disposition of Königsberg bridges, and so the theorised walk was impracticable.

This example is the first recorded application of graphs for representing and then solving a problem. In fact, paths through graphs visiting each edge only once are still known as Eulerian paths!

This amusing tale clearly portrays one of CT's biggest advantages: it offers a straightforward connection with the real world so that students can feel what they are learning and practising has real value outside the classroom. This not only can potentially increase their motivation: it can also make the adoption of abstract concepts a tad easier!

When we think about the Internet we might regard it as a series of computers connected together. Without losing any generality we can model said network as a (possibly undirected) graph. If the graph were to be undirected that would simply mean that links $l \in E$ would become unordered pairs (i.e. let $a, b \in E \rightarrow (a, b) \equiv (b, a)$, which would not be true for directed graphs). This is usually represented by double-tipped (or untipped) arrows, meaning they can be traversed in any direction. Now, if we wanted to know how to reach a given computer to, say, send an email we would need to know the *path* from our machine to the other one. What is more, we will usually want to use the fastest possible link so that the communication progresses as fast as possible. We can model a link's quality through a scalar we call the *cost* ($c \in \mathbb{R}$). Usually, the higher the cost the slower the link is. This means our objective is to **minimise** this cost when we find out our paths.

Thus, the problem to solve implies finding a path from an origin node $O \in V$ to a destination node $D \in V$ so that the addition of the cost of the links we traverse is the minimum possible. The solution will be a subset P of V specifying the nodes one would have to 'jump through' to reach D from O. In order to make matters more interesting the problem asks for the paths from O to every other node in the graph. Figure 4 contains a sample graph extracted from (Kurose & Ross, 2018). Note the node names are the first few letters of the Greek alphabet: that introduces these letters to students in an effort to demystify them whilst also prompting them to learn their names.

After staring at the graph for a while and trying out several combinations, students will begin providing answers to the problem. They can be provided with a table whose format is

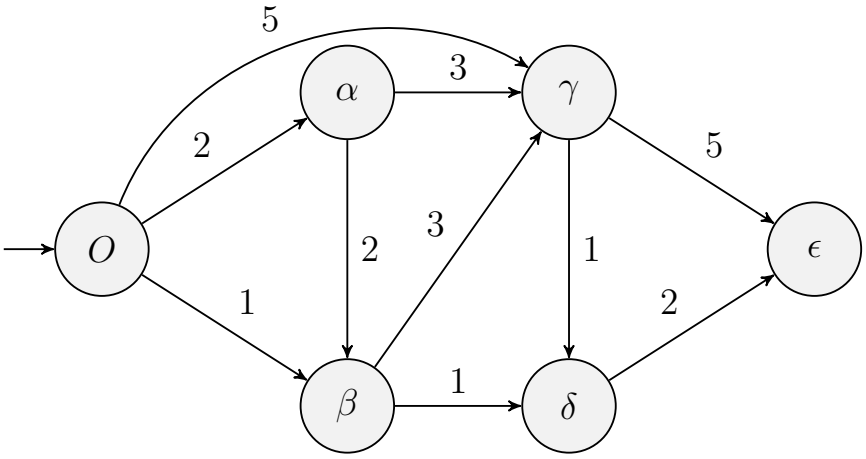


Figure 4
A graph modelling a network. Costs are represented by the numbers above the edges.

similar to that of table 4 so that they can organise their answers. At any rate, the pupils will be coming up with their own algorithms 'on the fly' to then apply them to get an answer.

The algorithms pupils will come up with will most likely be quite inefficient and they simply will not scale as the order (i.e. the number of nodes) of the graph grows. This offers a great opportunity for an informal introduction into algorithmic complexity and efficiency. This chance can also be leveraged to mention two of the most widely used routing algorithms. Maybe their most valuable quality is that they are guaranteed to *terminate*. Again, explaining the idea of termination as the guarantee that a solution will be found can be an extremely worthwhile time investment given the far reaching implications of the related concept of convergence in areas such as the study of series and integrals, for instance. In any case, the two most widely applied 'families' of routing algorithms are:

- **Dijkstra**: This algorithm devised in 1959 by Dutch Computer Scientist Edsger W. Dijkstra can find all the shortest paths from a given node to all the others on a graph by minimising the cost along the paths. It will always terminate as long as the costs of the different links are positive, which is usually the case. In other words, this algorithms solution would be exactly the same as the one set forth in table 4. The basic idea underlying the algorithm is finding the first node in the path by looking at all those that are a single 'hop' away to then choose the one whose cost is the lowest. After that, the algorithm tries to find the next node with the lowest cost out of those remaining and so on. After this process is repeated as many times as there are nodes in the graph, the paths to all the other nodes will be discovered. It is worth noting this algorithm requires a 'complete knowledge of the graph' (i.e. it needs to know all the edges and their costs). Enhanced versions of this algorithm exist such as A* (A-star) which find applications in fields such as the autonomous navigation of robots. Dijkstra's algorithm is at the heart of protocols key to the internet such as `OSPF`, which makes Dijkstra's contribution an invaluable one

Destination	α	β	γ	δ	ϵ
Path	α	β	$\beta \to \delta \to \gamma$	$\beta \to \delta$	$\beta \to \delta \to \epsilon$
Cost	2	1	3	2	4

Table 4
Least-cost paths from node O for the graph on figure 4.

on which we all depend on a daily basis. A more thorough discussion on this algorithm as well as on `OSPF` can be found on (Kurose & Ross, 2018).

- **Bellman-Ford**: This algorithm was first published by Richard Bellman and Lester Ford Jr., but it was Alfonso Shimbel who first proposed it in 1955. This algorithm solves the same problem as Dijkstra's but without the need of a complete knowledge of the network. It is indeed slower, but it is a more flexible alternative that can even work in the face of negative link costs. The algorithm itself relies on the so called *Bellman-Ford* equation. In order to succinctly define it, one needs to define some auxiliary functions. First of all, function $d_x(y)$ returns the least-cost path from node $x \in V$ to node $y \in V$. In a similar fashion, function $c(x, y)$ returns the cost between nodes x and y. Finally, note how the $\min_{v \in \text{neighs}(x)}\{\cdot\}$ operator simply states that the expression within the brackets is computed for each node $v \in V$ that is a neighbour of (i.e. directly connected to) node x. This allows the definition of the Bellman-Ford equation:

$$d_x(y) = \min_{v \in \text{neighs}(x)} \{c(x, v) + d_v(y)\}$$

The above expression simply states that the least cost path from node x to node y is the one traversing node v so that the addition of the cost from x to v and from v to y is the least possible one. This algorithm is behind some crucial internet protocols such as `BGP`, the culprit behind Facebook's outage in 2021. Again, (Kurose & Ross, 2018) contains a superb explanation on all these concepts.

The strategies devised by students will most likely have more in common with Dijkstra's algorithm, but if someone chooses to solve the graph 'backwards' (i.e. starting from each destination node and reaching the origin O) the ideas put in motion are more closely related with the Bellman-Ford approach.

All in all, this activity provides an engaging challenge with deep links to technologies used by students in a daily fashion. On top of that, this activity can be endlessly remixed and its difficulty can be easily tweaked to suit classrooms with very disparate levels.

This activity has been proposed by *Mª Emilia Llarandi Polo*, *Andrés Romero Rielo* and the author for the *Mates en la Calle* activity on the *Taller de Matemáticas* course. The choice to include it in this document has been made due to the great reception of the activity by both the pupils and teachers of the school it was put in practice.

The Mandelbrot Set

In Denning and Tendre's book, the construction of the *Mandelbrot Set* is pitched as an example of a common simulation technique. Given an n-dimensional grid a function is computed for each point. This strategy applied to the 2-dimensional *Mandelbrot function* shows how a seemingly simple expression can exhibit a hugely complex behaviour. The *Mandelbrot function* is:

$$f_c(z) = z^2 + c; \quad c, z \in \mathbb{C}$$

Then, the *Mandelbrot set* is defined on the complex plane as:

$$M_s = \{c \,/\, |f_c^n(0)| < M \text{ as } n \to \infty\}; \quad n, M \in \mathbb{R}$$

In other words, the Mandelbrot set is composed of the complex numbers c for which the absolute value of the function $f_c(z)$ is bounded when iterated at $z = 0$. Bear in mind iteration is represented by the exponentiation of the function f, that is, $f^3(0) = f \circ f \circ f(0) = f(f(f(0)))$.

The initial depiction of the Mandelbrot set was carried out by Robert W. Brooks and Peter Matelski in 1978 (Kra & Maskit, 1981) and it is depicted in figure 5.

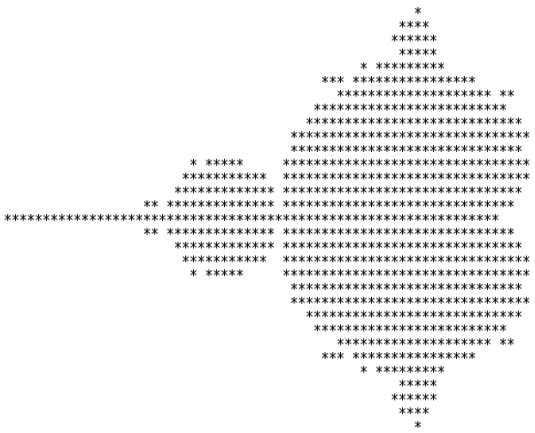


Figure 5
Command-line depiction of the Mandelbrot set (Elphaba, 2007).

Given its succinct definition, the Mandelbrot set lends itself really well to programming. It is rather easy to whip up a program capable of replicating figure 5. An example in `Go` (The Go Team, 2023) is provided on listing 1.

```
package main

import "fmt"
import "math/cmplx"
```

```
func mandelbrot(a complex128) (z complex128) {
    for i := 0; i < 50; i++ {
        z = z*z + a
    }
    return
}

func main() {
    for y := 1.0; y >= -1.0; y -= 0.05 {
        for x := -2.0; x <= 0.5; x += 0.0315 {
            if cmplx.Abs(mandelbrot(complex(x, y))) < 2 {
                fmt.Print("*")
            } else {
                fmt.Print(" ")
            }
        }
        fmt.Println("")
    }
}
```

Listing 1: Mandelbrot set implementation in Go with $n = 50$ and $M = 2$.

The same be accomplished with other languages as well. An example in `Haskell` (Marlow et al., 2010) can be seen on listing 2. It produces the exact same output as the code on listing 1. A quick (even if uninformed) visual inspection of both listings will show clear differences in how the languages are structured and how they strive to be written. It is interesting to note Haskell's design is heavily influenced by Church's λ-calculus to the extent Haskell is classified as a *functional* programming language instead of an *imperative* one, which is the category Go belongs to.

```
import Data.Bool ( bool )
import Data.Complex (Complex ((:+)), magnitude)

mandelbrot :: RealFloat a => Complex a -> Complex a
mandelbrot a = iterate ((a +) . (^ 2)) 0 !! 50

main :: IO ()
main =
  mapM_
    putStrLn
    [ [ bool ' ' '*' (2 > magnitude (mandelbrot (x :+ y)))
        | x <- [-2, -1.9685 .. 0.5]
      ]
      | y <- [1, 0.95 .. -1]
    ]
```

Listing 2: Mandelbrot set implementation in Haskell with $n = 50$ and $M = 2$.

These two pieces of code have been extracted from (Code, 2023) and then refined. Working examples have been published to `Replit` (Replit Inc., 2023) and can be found on (Collado Soto, Pablo, 2023a) and (Collado Soto, Pablo, 2023b), respectively.

One might wonder how this activity fits in the context of a K-12 classroom. The truth is no one would expect students to come up with the programs contained in the previous listings. However, this does not prevent the study of the implementations from having any values. On the one hand, the meaning of the definition of the set is clearly set forth in both listings. The fact that they can be played with and modified with guidance follows Papert's ideas that programming can behave as a 'laboratory' in which to explore and study far reaching and complex concepts such as the one this proposal revolves around. Questions such as what the increase of n would entail or what would happen if M changed pose difficult-yet-attainable challenges pushing the limits of the contents taught in classrooms. What is more, this activity perfectly portrays how programming can offer a platform on which to make abstract ideas concrete and therefore enhance the students' understanding along the way.

Computing π with a Monte Carlo simulation

A classic example of *computational algorithms* are the so called *Monte Carlo Methods*. Put simply, these involve generating large amounts of random samples so that predictions and numerical results can be derived from them. Everyone's introduction to these methods is usually related to obtaining an approximation of π's numerical value where $\pi = 3.141592653589793....$ The definition of π is that of the ratio between length of a circumference C and its diameter D and is given by $\pi = \frac{C}{D} = \frac{C}{2R}$ where R is the circumference's radius. The constant π is also present in the expression of the area A of a given circle with radius R so that $A = \pi R^2 \rightarrow \pi = \frac{S}{R^2}$. This last relation can be leveraged in a Monte Carlo experiment to numerically derive the value of π. As an interesting fact, the expression for A can be derived from the expression of π's definition through the following integral:

$$A = \int_0^R C\,(x)\,dx = \int_0^R 2\pi x dx = \pi \int_0^R 2x dx = \pi x^2 \Big|_0^R = \pi R^2$$

Figure 6 depicts the 'setup' of the Monte Carlo experiment. We begin by defining a square with side $l = 1$. This implies the square's area is $A_s = l^2 = 1^2 = 1$. Within it, we inscribe a circumference whose diameter $D = 1$ and, consequently, its radius $R = \frac{D}{2} = \frac{1}{2}$. This locates the circumferences centre at point $O = \left(\frac{1}{2}, \frac{1}{2}\right)$. Applying the well-known expression for the area of a circle (i.e. $A_c = \pi R^2$), once concludes the circle's area is $A_c = \pi \left(\frac{1}{2}\right)^2 = \frac{\pi}{4}$). This implies the ratio of both areas (R_t) is given by $R_t = \frac{A_c}{A_s} = \frac{\frac{\pi}{4}}{1} = \frac{\pi}{4}$. Then, if we multiply this ratio by 4 we can get an estimate of π's value, that is, $\pi \sim 4R_t$.

Knowing the above, we can define our experiment. We will begin generating random points $P_r = (x_r, y_r)$ where both x_r and y_r are random real numbers in the $[0, 1)$ interval. For each point we will decide whether it landed **within** the circumference or whether it landed

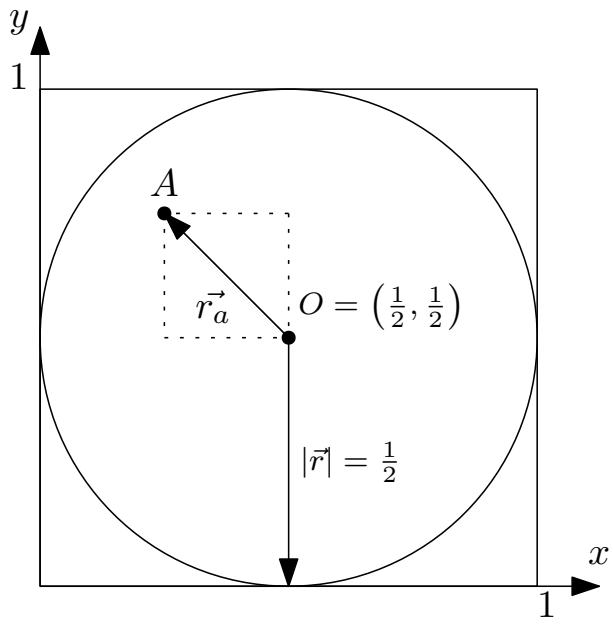


Figure 6
Monte Carlo experiment for finding an approximation of π.

outside it. Given both coordinates are constrained to the $[0, 1)$ interval they are guaranteed to land within the square. Now, given these points are randomly generated, the ratio between the points landing within the circumference and those landing beyond it **must** adhere to the ratio between the areas of the two figures. After all, the larger the area of a figure the more likely it is a random point will belong to it. One can be convinced about this line of reasoning by thinking about the limit cases. If π were to be 0, the area of a circle would vanish and no points would land on it. Thus, the Monte Carlo experiment would 'rightfully' conclude that $\pi = 0$. If on the other hand the area of the circle were the same than that of the square, all the points would hit. This would imply $R_t = 1 \rightarrow \frac{\pi}{4} = 1 \rightarrow \pi = 4$. This value would mean that the area of our circle becomes $A_c = 4 \cdot \left(\frac{1}{2}\right)^2 = 4 \cdot \frac{1}{4} = A_s$. In other words, the predicted value of π would be correct because both A_c and A_s would be the same. In an effort to be a tad more formal, we can state that if $n \in \mathbb{N}$ random points are generated (which entails generating $2n$ real numbers), the ratio between the hits n_h and the misses $n_m = n - n_h$ approaches π in the limit:

$$\pi = \lim_{n \to \infty} 4 \frac{n_h}{n - n_h}$$

The last 'piece of the puzzle' would be to define a function $h : \mathbb{R} \times \mathbb{R} \rightarrow \{0, 1\}$ so that we get a 1 when the point lies within the circumference's boundaries and a 0 when it is beyond said boundaries. Taking advantage of the expression for the distance between points $A = (x_a, y_a)$ and $B = (x_b, y_b)$, $d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$ (i.e. a direct application of Pythagora's Theorem) we can define h as:

$$h(x_r, y_r) = \begin{cases} 1 & \text{if } d\left(\left(x_r, y_r\right), \left(O_x, O_y\right)\right) < R \\ 0 & \text{otherwise} \end{cases}$$

Note that in this particular scenario $O_x = O_y = R = \frac{1}{2}$. This problem can be directly transposed into code. Listing 3 contains an example in Go in which increasing values of n are used. Listing 4 contains a sample output. Bear in mind this example can be found on (Collado Soto, Pablo, 2023c).

```
package main

import (
  "fmt"
  "math"
  "math/rand"
  "time"
)

const (
  MAX_EXPONENT int = 9

  R float64 = 0.5

  O_X float64 = 0.5
  O_Y float64 = 0.5
)

func distFromOrigin(x, y float64) float64 {
  return math.Sqrt((x - O_X) * (x - O_X) + (y - O_Y) * (y - O_Y))
}

func generateRandomPoint() (float64, float64) {
  return rand.Float64(), rand.Float64()
}

func main() {
  rand.Seed(time.Now().UnixNano())

  for i := 1; i < MAX_EXPONENT; i++ {
    nHits := 0

    nTries := int(math.Pow10(i))

    for j := 0; j < nTries; j++ {
      randX, randY := generateRandomPoint()
      if distFromOrigin(randX, randY) < R {
        nHits++
```

```
      }
    }

    fmt.Printf(
      "Estimated value of PI with %9d hits out of %9d tries: %g\n",
      nHits, nTries, float64(4) * (float64(nHits) / float64(nTries)))
  }
}
```

Listing 3: Monte Carlo experiment for computing π.

```
Estimated value of PI with         6 hits out of        10 tries: 2.4
Estimated value of PI with        84 hits out of       100 tries: 3.36
Estimated value of PI with       794 hits out of      1000 tries: 3.176
Estimated value of PI with      7870 hits out of     10000 tries: 3.148
Estimated value of PI with     78419 hits out of    100000 tries: 3.13676
Estimated value of PI with    785446 hits out of   1000000 tries: 3.141784
Estimated value of PI with   7852687 hits out of  10000000 tries: 3.1410748
Estimated value of PI with  78534238 hits out of 100000000 tries: 3.14136952
```

Listing 4: Results of the Montecarlo Experiment. The predicted value gets better as n grows.

Given the experiment is intrinsically random, rerunning the code will (practically) always yield different results. This behaviour can prompt very interesting and nuanced questions such as how random numbers are generated by a computer and whether they are completely random. What does it mean to be random? It is quite a subtle concept when one begins to ponder all these implications.

At any rate, this sample activity offers a myriad of areas where the teacher can modify and modulate its difficulty. Ranging from the 'hardest' option of asking pupils to write the code from scratch to simply executing it so that they can draw their own conclusion, students are bound to enhance their knowledge of skills directly related to CT.

Aside from clearly shinning the light on CT, this 'version' of this classic activity provides the advantage of actually generating good enough approximations of π so that students can convince themselves that there is at least some 'truth' behind the experiment's setup. Even though it is true the high degree of abstraction might render the proposal unsuitable for the lower levels, it does offer a plethora of possibilities and options for subsequent levels.

Still much more...

The above subsections provide a minuscule sample of the activities one can leverage in the context of a classroom to bring CT a bit closer to the realm of mathematics. Incursions into randomness, modelling and simulating simple systems, working with different coordinates and their representations, establishing connections between types in programming languages and the sets in which real numbers are classified and leveraging computers to verify the properties of limits are just another set of strategies one might follow to achieve results similar to the ones pursued before.

At any rate, as long as the teacher's aim is to increase his or her pupil's knowledge whilst exposing them to CT as yet another asset in a mathematical context, chances are these students will be better equipped to tackle the problems and situations they will surely face in the future.

Closing Thoughts

> What is a magician but a practising theorist?
>
> *Obi-Wan Kenobi*

In the previous sections this document has tried to lay the foundations for CT's appearance in the mathematics curriculum. The argument rests on the solid and comprehensive mathematical foundations of computing as a science. Everything from ancient computations and algorithms to extremely complex abstractions are an integral part of CT through the ages. Milestones in computing's history include Lovelace and Babbage's Analytical Engine, Boole's algebra and Turing's Machines along Church's λ-functions. They all were kind enough to bestow knowledge upon us that has fuelled the astonishing digital revolution in which we are all still immersed. This discussion did not shy away from a mathematically-heavy form whose primordial aim was showing how, as initially stated, mathematics play a much larger role in computing than most would initially assume.

Computing and CT are not interchangeable concepts. Actually, CT continues to be an elusive concept to define and many authors have tried their hand at providing a clear-cut definition. The first mention of CT was by Papert whose general viewpoint was that programming provided a virtual playground where pupils could stretch and bend their ideas in a quest to deepen their knowledge. CT's relevance was no stranger in technologically inclined contexts, but it was a long time before it began graining momentum in academic curricula. Wing's 2006 essay provided a tremendous stepping stone on which may authors helped bring CT into the classroom. Albeit a bit hyped up, Wing's definition enjoys still far reaching impact as it was the first to point out CT's transversal and universal applicability. Quite recently, Denning and Tendre wrote a book on CT in which they explored its chronological development together with other aspects more focused on the engineering aspect of CT. Not only that, but they also provided a clear and concise definition of CT in which they argued it was a two-fold concept. The first dimension concerned itself with computers as machines, whilst the second one (much closer to mathematics) looked into how behaviours can be modelled and understood as information-manipulation processes. This second stance really resonated with us. In the Spanish scene Belén Palop has been a relentless advocate for CT in the curriculum who has crafted and created a myriad of invaluable resources for preaching and understanding CT. Her definition is much more concrete than Denning and Tendre's as it focuses on CT's interpretation of problems over input data accompanied by a generated output. Taking all the above proposals into account, we provided our own definition of CT to then support our claim based on the document's contents up to that point. Aside from the theoretical aspect of CT the document also devoted some lines to CT's obstacles and overinflated hype as well as CT's limits. These two subtle and yet crucial aspects must be explicitly addressed so as to avoid any misconceptions

about what CT is and what it can offer. If there is one sentence to remember, is that ‘CT is not programming!’.

Even though it is a completely new addition to the Spanish curriculum, CT has been present in curricula around the world. The document takes a moment to analyse how it is included, defined and exercised so that common pitfalls can be avoided whilst at the same time leveraging the strong strategies found and discovered elsewhere. In an effort to get an international overview, Singapore’s, the US’, and the UK’s curricula are looked at to then compare their proposals with Spain’s. It also explains some of CT’s obstacles in Spain. Given this section’s breadth and depth, some further articles and valuable resources are also presented.

In order to provide a more concrete and down-to-Earth approach the document proposes four different activities exercising CT in the context of a regular mathematical classroom. Some of these examples require no computer whatsoever, thus reinforcing the idea that CT is fairly independent from computers themselves. Examples relying on small code snippets reference freely available and fully functional examples so as to provide the most useful material possible.

If something is clear, is that CT has come to stay. This loosely defined set of abilities are a key component in the skill set of modern citizens who will have to grapple with extremely vexing conundrums in the near future. This document tries to point out how CT is related to but not dependent on computers and how it is far from ‘just programming’. Even though some curriculum proposals fall prey to the public opinion’s view of CT, countries around the world are trying to include meaningful manifestations of CT in their teaching legislation already distancing themselves from computing literacy and the hollow poster-worthy applications involving robots and visual programming languages. Spain’s proposal is still wound and the initial adoption of CT across the country is still to be seen and gauged. However, an initial comparison lays bare the curriculum’s shortcomings: there is still work to do, specially in the area of teaching training. This might prove to be a gloomy outlook for CT in Spain, but nonetheless! The current scene of CT offers a myriad of opportunities to provide input and ideas so as to steer CT in the right direction.

Our most sincere hope is that this document proves to be a valuable asset for anybody tackling the sometimes elusive concept of CT so that this tremendously necessary and potent concept can be pushed to its limits.

References

Belén Palop. (2022a). Gathering for gardner. https://www.youtube.com/watch?v=Or6YnKpRHtQ [Online; accessed 2-August-2023]

Belén Palop. (2022b, November 29). Pensamiento computacional: Del nuevo currículo al aula de ESO. https://www.youtube.com/watch?v=bpynKYkt1wQ

Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? [event-place: Virtual Event, Canada]. *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623. https://doi.org/10.1145/3442188.3445922

Boletín Oficial del Estado. (2022, March). Real decreto 217/2022, de 29 de marzo, por el que se establece la ordenación y las enseñanzas mínimas de la educación secundaria obligatoria. https://www.boe.es/eli/es/rd/2022/03/29/217/con BOE-A-2022-4975

Boole, G. (1854). *An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities*. Cambridge University Press. https://doi.org/10.1017/CBO9780511693090

Boole algebra theorems. (n.d.). https://www.mi.mun.ca/users/cchaulk/misc/boolean.htm [Online; accessed 27-July-2023]

Code, R. (2023). *Mandelbrot set — rosetta code,* https://rosettacode.org/w/index.php?title=Mandelbrot_set&oldid=346921 [Online; accessed 27-July-2023]

Collado Soto, Pablo. (2023a, August 24). *Mandelbrot go implementation*. https://replit.com/@pcolladosoto/Mandelbrotgo

Collado Soto, Pablo. (2023b, August 24). *Mandelbrot haskell implementation*. https://replit.com/@pcolladosoto/Mandelbroths

Collado Soto, Pablo. (2023c, August 24). *Monte carlo experiment computing pi's value*. https://replit.com/@pcolladosoto/Monte-Carlo-for-Pi

Dastin, J., & Tong, A. (2023). Focus: Google, one of AI's biggest backers, warns own staff about chatbots. *Reuters*. https://www.reuters.com/technology/google-one-ais-biggest-backers-warns-own-staff-about-chatbots-2023-06-15/

Dean, W. (2023). Recursive functions. In E. N. Zalta & U. Nodelman (Eds.), *The stanford encyclopedia of philosophy* (Summer 2023). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2023/entries/recursive-functions/

Denning, P. J., & Tedre, M. (2019, May). *Computational thinking*. MIT Press.

Dijkstra, E. W. (1974). Programming as a discipline of mathematical nature. *American Mathematical Monthly*, *81*, 608–612. https://api.semanticscholar.org/CorpusID:122169249

Dijkstra, E. W. (1984, November). *The threats to computing science*. http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD898.PDF
[Online; accessed 24-August-2023]

Elphaba. (2007). *Command-line depiction of the mandelbrot set*. Retrieved August 23, 2023, from https://commons.wikimedia.org/wiki/File:Mandel.png

Euler, L. (1736). Solutio problematis ad geometriam situs pertinentis. https://archive.org/details/commentariiacade08impe/page/128/mode/2up
[Online; accessed 24-August-2023]

European Commission, Joint Research Centre, Engelhardt, K., Punie, Y., Chioccariello, A., Ferrari, A., Dettori, G., Kampylis, P., & Bocconi, S. (2016). *Developing computational thinking in compulsory education – implications for policy and practice* (Y. Punie & P. Kampylis, Eds.). Publications Office. https://doi.org/10.2791/792158

Hickmott, D., Prieto-Rodriguez, E., & Holmes, K. (2018). A scoping review of studies on computational thinking in k–12 mathematics classrooms. *Digital Experiences in Mathematics Education*, *4*(1), 48–69. https://doi.org/10.1007/s40751-017-0038-8

Hicks, S. (2009). Rapid prototyping in TeX. *The Monad Reader*, (13). https://wiki.haskell.org/wikiupload/8/85/TMR-Issue13.pdf

Hilbert, D., & Ackermann, W. (1999, September). *Principles of mathematical logic*. American Mathematical Society.

Hopcroft, J. E., & Ullman, J. D. (1979, January). *An introduction to automata theory, languages, and computation*. Pearson.

Hsu, T.-C. (2019). A study of the readiness of implementing computational thinking in compulsory education in taiwan. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 295–314). Springer Singapore. https://doi.org/10.1007/978-981-13-6528-7_17

IEEE standard for floating-point arithmetic. (2019). *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

Kra, I., & Maskit, B. (Eds.). (1981, May). *Riemann surfaces related topics (AM-97), volume 97*. Princeton University Press.

Kristensen, K. (2020). Implementation of computational thinking in school curriculums across asia. In C. Stephanidis & M. Antona (Eds.), *HCI international 2020 - posters* (pp. 269–276). Springer International Publishing.

Kuntz, J. (2022). Markov chains revisited [_eprint: 2001.02183].

Kurose, J., & Ross, K. (2018). *Computer networking: A top-down approach, global edition*. Pearson Education. https://books.google.es/books?id=IUh1DQAAQBAJ

Lodi, M., & Martini, S. (2021). Computational thinking, between papert and wing. *Science & Education*, *30*(4), 883–908. https://doi.org/10.1007/s11191-021-00202-5

Marlow, S. et al. (2010). Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books. https://books.google.es/books?id=1oPRDwAAQBAJ

Python Software Foundation. (2023, August 24). *The python programming language*. https://www.python.org

Queiroz, D. (2010). *State diagram of a 3 state busy beaver*. Retrieved July 30, 2023, from https://commons.wikimedia.org/wiki/File:State_diagram_3_state_busy_beaver_2B.svg

Replit Inc. (2023, August 24). *Replit*. https://replit.com

Rojas, R. (2015). A tutorial introduction to the lambda calculus. *CoRR*, *abs/1503.09060*. http://arxiv.org/abs/1503.09060

Shannon, C. E. (1940). *A symbolic analysis of relay and switching circuits* (Doctoral dissertation). Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/11173

Singapore Ministry of Education. (2023, August). Subjects for normal (academic) course. https://www.moe.gov.sg/secondary/courses/normal-academic/electives#subjects
[Online; accessed 23-August-2023]

Strachey, C. (1965). An impossible program. *The Computer Journal*, *7*(4), 313–313. https://doi.org/10.1093/comjnl/7.4.313

The Go Team. (2023, August 24). *The go programming language*. https://go.dev

UK Department for Education. (2014, December). National curriculum in england: Secondary curriculum. https://www.gov.uk/government/publications/national-curriculum-in-england-secondary-curriculum
[Online; accessed 23-August-2023]

Virginia Department of Education. (2016, September). Mathematics standards of learning. https://www.doe.virginia.gov/teaching-learning-assessment/instruction/mathematics/standards-of-learning-for-mathematics
[Online; accessed 23-August-2023]

Virginia Department of Education. (2017, September). Computer science standards of learning. https://www.doe.virginia.gov/teaching-learning-assessment/instruction/computer-science
[Online; accessed 23-August-2023]

Wikiquote. (2023a). Ada lovelace — wikiquote, https://en.wikiquote.org/w/index.php?title=Ada_Lovelace&oldid=3339889
[Online; accessed 3-August-2023]

Wikiquote. (2023b). Arthur c. clarke — wikiquote. https://en.wikiquote.org/w/index.php?title=Arthur_C._Clarke&oldid=3300288

Wikiquote. (2023c). John von neumann — wikiquote, %5Curl%7Bhttps://en.wikiquote.org/w/index.php?title=John_von_Neumann&oldid=3335657%7D
[Online; accessed 24-August-2023]

Wing, J. M. (2006). Computational thinking [Place: New York, NY, USA Publisher: Association for Computing Machinery]. *Commun. ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215