Turing Machines as Clocks, Rulers and Randomizers

José Félix Costa

Department of Mathematics, Instituto Superior Técnico, Technical University of Lisbon, Portugal
e-mail: `fgc@math.ist.utl.pt`
Centro de Matemática e Aplicações Fundamentais, University of Lisbon
Centro de Filosofia das Ciências da Universidade de Lisboa, University of Lisbon

Abstract In this paper we specify Turing machines to serve as clocks, rulers, and randomizers of the most basic complexity classes in such a way that it can be seen as a contribution to the understanding of computational complexity. The article is educational and first ideas about Turing machines, computation and classes are introduced from scratch. However, the expected examples of Turing machine computations are focused in the fundamental, nevertheless "semi-obscure" subject of the alarm clock and space bound ruler.

Keywords: Time; Space; Guess; Clock; Ruler; Time and space constructible functions; Complexity class; Enumeration.

1 Introduction

Generally, an introduction of a paper introduces the subject in a easier way than its body. However, our intention is to attract also the readers acquainted with the concept of a Turing machine and, therefore, we write the introduction to them and ask the reader who doesn't know anything about Turing machines to skip 2 or 3 paragraphs with the promise that the progression of concepts is slow and suitable for a first reading on the theory of computability and complexity.

Clocked Turing machines (or other clocked abstract devices) are a fundamental tool in complexity theory. For most of the applications of the theory it is enough to prove that in the worst case scenario a given algorithm runs in a number of basic steps bounded by some (suitable) function on the size of the data. But, once we start proving general results for a complexity class, the concepts of a clock and a ruler (to delimit space resources) are fundamental. It becomes obvious that the so-called hierarchy theorems of basic complexity theory hold true only for the so-called time and space constructible bounds — provided by alarm clocks and rulers. To preserve space complexity in some theorems of replacement of Turing machines by Turing machines that halt for all inputs, the concept of a space constructible bound turns again to be crucial. Finally, clocked

Turing machines become fundamental when one starts to enumerate the elements of a complexity class and to apply diagonalisation methods over the classes.

A clocked Turing machine is a Turing machine that "counts" its own steps of computation. It can be seen as a simultaneously simulation, in parallel, of a given machine $\mathcal{M}$ together with a given clock. The (alarm) clock itself is a Turing machine with some remarkable properties and the parallel of the two Turing machines is also a Turing machine.

Alarm clocks and rulers were introduced by Hartmanis, Lewis an Stearns in 1965 in a series of three papers (see [8, 10, 16]). Alarm clocks are not well developed in complexity theory literature, either in books or in research papers (may well be they are considered a boring subject), but their existence is taken for granted in the very beginning. Sufficient conditions for Turing machines to serve as clocks or rulers, proved by Kobayashi in [9],[1] are used to provide the grounds for complexity theory. Most of the students never saw a *clock* or a *ruler*, although in some courses they might have proven that they exist. The proof of Kobayashi's theorem does not guaranty that the clocks display exact English time up to a tick (!), but more or less precise time (as a few detailed constructions of the proof show).

Although to prove the theorems it is not needed to make clocks or rulers precise (!), it is obvious that the reader may wonder if such devices can be built to be exact. Students do! In fact such devices can be built as we prove along the paper and they are based on a technique that is recurrently used based on specific marks on the tapes of a Turing machine.

Which are the motivations to read this article? We sum them up as follows: (a) to get acquainted with the concept of a Turing machine, (b) to learn how to specify a multitape Turing machine through its transition diagram, (c) to learn how to specify non-trivial machines, namely the special machines — the clocks —, (d) to understand how to enumerate complexity classes and, finally, (e) to know some very basic results about Turing machines.

In what follows, we introduce Turing machines in Section 2 and we define in Section 3 time and space constructible functions in a similar (asymptotic equivalent but different) way to that found in standard books. In Section 4, we discuss the most well known models of computation: deterministic, non-deterministic and probabilistic. Examples of basic and composite clocks are given in Section 5. In Section 6, we prove that the class of constructible functions is closed under basic and useful operations. Then, finally, in Section 7, some theorems about the relation between clocks and rulers. We end up in Section 8 with some concluding remarks.

[1]Such a proof can be found, e.g., in the famous book by Balcázar, Diaz and Gabarró (see [2]).

2 The Turing machine

2.1 The idea of an abstract computing machine

The Turing machine was introduced by Alan Turing in 1936 in a paper entitled "On Computable Numbers, with an Application to the *Entscheidungsproblem*,[2] publish by the *Proceedings of the London Mathematical Society* (see [17]). Some corrections were done to this paper by Turing himself and published in 1937 in the same journal (see [18]). (An interesting, fully commented translation into French of the two most well known papers authored by Turing appeared in the book "La Machine de Turing" by Jean-Yves Girard (see [19]).)

Many of the readers to whom the concept of a Turing machine is familiar have certainly asked themselves: How did the idea of the Turing machine occurred to Turing? Turing was searching for a "proof" that mathematics is not reducible to algorithmic procedure. To do such a "proof", Turing needed to find (a) a reasonable concept of algorithm and (b) to prove that, once accepting such a concept as reasonable, some decision problems in mathematics are not algorithmic. And that would be a "proof" of undecidability of Hilbert's *Entscheidungsproblem*, a mathematical open problem that David Hilbert and Wilhelm Ackermann identified in 1928, and David Hilbert (in 1928) added to his "1900 research program" for mathematicians.[3]

How did Turing solved the problem of finding a suitable notion of algorithm? Well... He did it while abstracting from women computers of the thirties. Women computers[4] needed paper (unlimited amounts), pencil and rubber. They used to write notes on paper, to erase some of the notes, to continue writing, erasing and rewriting until the computation is finished. Writing on paper can be reduced to writing along a horizontal line in math paper, a unidimensional tape filled with cells. In each moment of a computation, the computer has access to a finite amount of information; based on her state of mind, the computer looks at the information available and makes a transition to a possible new state of mind, possibly erasing the information read and replacing it by new information, and moving to the next or the previous page. A more complete description of how the idea germinated can be found in the book by Martin Davis (see [3]). This process of human computing was abstracted by Turing in the way that follows. We will work out a definition of the Turing machine close to the one provided by Sipser in [15] (but there are dozens of different but equivalent definitions of a Turing machine).

In Figure 1 we see the pictorial representation of a Turing machine with two tapes,

[2]It means *decision problem* in German.

[3]Although the origin of such a problem goes back to the philosopher and scientist Gottfried Leibniz in the late XVII century, also inventor of some computing devices.

[4]In the thirties, in Engand, the word computer meant a person (typically female) whose job was doing computations. A person could apply to a job of computer (see the article of W. Barkley Fritz[5]).

the input tape and one working tape, together with the finite control. The finite control is then developed in transition diagrams depicted in the figures that follow along this paper. In the input tape we see a binary word written in the first n cells of the tape: we say that the input has size n ($= 5$ in the case), no matter the alphabet used to write the input. The complexity of a Turing machine is a function of this n. In the working tape we see a word over a larger alphabet occupying 6 cells. This larger alphabet is known as the working alphabet: the machine works with the input alphabet possibly enriched with further symbols considered necessary to better develop the algorithm. The heads can read one letter at a time. In fact we could define Turing machines in a way such that each head could read several symbols at the same time. But if it can improve algorithm specification language, it does not change the picture, it does not even change the complexity of the machine. The finite control is a finite state/transition device — called finite automaton — describing the algorithm. The memory of the machine is divided into two parts: one is external memory, the information that the machine has available in the tapes, and the other is the memory the machine has in the finite control that can not be changed during machine computations.[5] Three particular states are depicted: the current state q of the machine, the initial state q_0 and the halting states q_{halt}. Thus, we expect to find in the finite control — the transition diagram — what is required the machine to do, e.g., when in state q, the input head is reading 0 and the working tape head is reading 1. The Turing machine is completely specified if, for all states and for all possible symbols under the heads, it is specified what to do next. When the machine reaches an halting state, either the accepting or the rejecting states, q_a and q_r, respectively, it switches itself off. Thus, it is not that much relevant to specify what the machine has to do in a halting state.

2.2 Tapes and configuration

A tape is an infinite unidirectional unidimensional array holding symbols from an alphabet Γ (finite) containing the special blanc symbol $\sqcup$. In this way the tape abstracts blanc paper in an unbounded quantity to be used by a human computer to perform computations. The notes that the computer need to make are aligned in the tape followed by an unbounded number of blanc cells. We can even imagine that the cells are arranged in k tapes to which we add an input tape with the data to be processed and an output tape to print the result of the computation. A configuration of the machine corresponds to k sequences of symbols from Γ divided by a state from Q into two subsequences, the subsequence of symbols to the left and the subsequence of symbols to the right of the corresponding head; the second subsequence includes the symbol under the head.

[5] E.g., words can be hard-wired in the finite control in such a way that the Turing machine can write them on the tape whenever needed — this is the hard-wired memory.

Therefore, the configuration of the Turing machine represented in Figure 1, is the pair of sequences $1011q0$ and $1Xq1Y0$. These two sequences contain all the information we need to know about the machine at this step of computation.

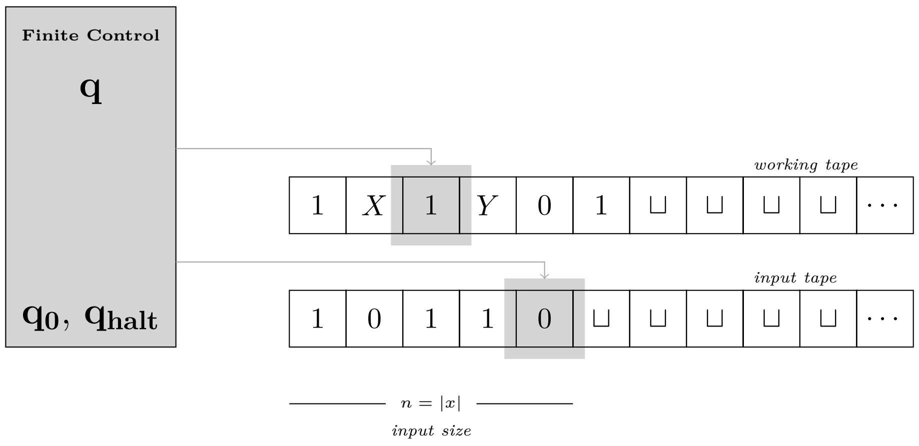

Figure 1: Illustration of the tapes and the reading/writing heads of a Turing machine. Some features: (a) the machine depicted has two tapes, (b) the input alphabet is binary $\{0, 1\}$, and (c) the working alphabet is larger $\{0, 1, X, Y, \sqcup\}$, namely it always contains the blanc symbol $\sqcup$. The input tape is read-only

2.3 Formal definition of a Turing machine

Formally, in mathematical terms, a Turing machine is an octuple:

$$\mathcal{M} = (k, \Sigma, \Gamma, Q, q_0, q_a, q_r, \delta)$$

where:

- k is the number of working tapes of $\mathcal{M}$ (in addition to the read-only input tape and the write-only output tape);
- Σ is the finite input alphabet not containing the special blank symbol $\sqcup$;
- Γ is the working alphabet, such that $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$;
- Q is the (finite) set of states;
- $q_0 \in Q$ is the initial state;
- $q_a \in Q$ is the accepting state;

- $q_r \in Q$ is the rejecting state (q_0, q_a and q_r are pairwise disjoint);
- $\delta : Q \times \Gamma^{k+1} \to Q \times \{L, R, N\}^{k+2} \times \Gamma^{k+1}$ is the transition function.

We may refer to any of q_a or q_r as q_{halt}.

To depict the *finite table* δ we use directed graphs. In this graphs the vertexes are the states and each edge is labeled with the transitions. Let us analise the complex transitions of the transition diagram of Figure 8: e.g., we have one transition between states p_0 and p_1, labeled by $0, 1; \sqcup; \sqcup \to R; \breve{0}, R; \breve{0}, R$.

> Once in state p_0, if the input head is reading 0 or 1, the second and the third heads are reading blanc (notice the punctuation in both sides of the arrow), then the machine makes the transition to state p_1, moving the input head one cell to the right, writing $\breve{0}$ in the blanc cell in the tape 2 and moving the head one cell to the right, and doing the same in the tape 3.

Consider now the transition between states p_k and q_1, labeled by $; 0; \to; \grave{0}, L; L$.

> Once in state p_k, no matter what is written in the tapes 1 and 3, if the head is reading 0 on the tape 2, then the machine makes a transition to the state q_1, moving the heads of tapes 2 and 3 one cell to the left and leaving the head of the input tape where it was (since, at this point of the computation, the input was already totally swiped, we conclude that the input tape is no more relevant).

To finish this section we mention that computer scientists essentially discuss machines in terms of the transition δ function. Different δ functions identify and characterize different machines. Thus, to be remembered, the dynamic map of $k+2$-tape Turing machine with one read-only input tape and one write-only output tape is:[6]

$$\boxed{\delta : Q \times \Gamma^{k+1} \to Q \times \{L, R, N\}^{k+2} \times \Gamma^{k+1}}$$

But, if more than one transition is permitted in some configuration of a state/transition device, then we say that the device is non-deterministic. Turing machines can be also non-deterministic. In this case the dynamic map becomes:

$$\boxed{\delta : Q \times \Gamma^{k+1} \to \wp(Q \times \{L, R, N\}^{k+2} \times \Gamma^{k+1})}$$

[6]Notice the exponents.

where $\wp$ denotes the powerset. In a particular state, while reading $k+1$ cells on the $k+2$ tapes (output tape excluded from the reading), the non-deterministic Turing machine can perform one within a finite number of possible different transitions; each of the possible transitions is like those of a deterministic Turing machine: a possible change of state, a possible change of content of the $k+1$ tapes (input tape excluded from the writing), a possible movement of the $k+2$ heads. It is known that two possible transitions in each state are enough to capture the full power of non-deterministic machines. (In Section 2.5 we will elucidate the reader about the structure of a non-deterministic Turing machine specification.) In this way, a change of perspective over the concept of computation corresponds to a change in the δ dynamics.

Sometimes, we make Turing machines simpler and we just specify one-tape Turing machines, where the input is provided in the leftmost cells of the tape, and the head can both read and write (and, consequently, rewrite over the input). Multitape Turing machines with an input and an output tape constitute a useful formal model when building Turing machines in a LEGO of simpler Turing machines.

2.4 The step of a computation

One step of a Turing machine $\mathcal{M}$ on a configuration c_i generates a configuration c_f. Starting from the initial configuration c_0, the machine $\mathcal{M}$ generates a (possibly infinite) sequence of configurations. Once providing an input to a Turing machine either (a) it never halts, running forever, jumping from configuration to configuration, not necessarily in a loop of repeating configurations, or (b) it halts in finite number of steps in a configuration containing the accepting state q_s or the rejecting state q_r.

The computation of the Turing machine is obviously the finite or infinite sequence of its configurations.

We will be concerned with Turing machines that halt for all inputs. Other papers in this volume discuss the non-halting cases.

Let Σ be the input alphabet (i.e., the alphabet used to write the input) as in Section 2.3. By $\Sigma^\star$ we denote the set of words written with the letters of Σ — i.e., the set of all possible inputs. We introduce now the concept of a set decided by a Turing machine:

Definition 1 *A set $A \subseteq \Sigma^\star$ is decided by a* deterministic Turing machine *$\mathcal{M}$ if the computation of $\mathcal{M}$ ends in an accepting state whenever $x \in A$ and the computation of $\mathcal{M}$ ends in an rejecting state whenever $x \notin A$.*

Definition 1 introduces the so-called *decision problem*, that is, *the problem of finding an algorithm to decide the question*

$$\boxed{x \overset{?}{\in} A}$$

While playing with the concept of a Turing machine, it became standard to measure the time complexity of a computation by the number of steps executed until the machine halts and the space complexity by the number of cells swiped by the heads of the machine during the computation. The time complexity of an algorithm became the maximum number of steps executed by the machine on inputs of a given size n. The space complexity of an algorithm became the maximum number of cells swiped for inputs of a given size n.

Thus, we read in books that the time complexity of a given problem is the function $t(n) = n^2$, meaning that, *asymptotically*, for all $n \in \mathbb{N}$, for inputs of size n, no more than $k \times n^2$ transitions are needed to reach an halting state, for some constant $k \in \mathbb{N}$. Complexity in this sense refers to the worse case scenario for each input size.

It does not mean that this is the best way to measure the difficulty of solving a problem: it is one model of such a measure. Models are what Mathematics provide. E.g., we could say that super-polynomial time complexity can be better than polynomial time complexity (see [12]), since the super-polynomial function $n^{log(log(n))}$ grows slower than the polynomial n^2 on any input of size n that can be physical written down. It is enough to notice that $n^{log(log(n))} > n^2$ only at $n = 10^{100}$ (the number of particles in the observable universe). In any case, for any k, there exists an order p such that, for $n \geq p$, $n^k < n^{log(log(n))}$.

2.5 Examples

Now follows an example of a deterministic Turing machine and an example a non-deterministic Turing machine.

In the specification of complex Turing machines it is sometimes necessary to make a copy of a sequence of symbols. In Figure 2, we depict the transition diagram of a machine for the purpose.

The Turing machine $\mathcal{M}$ starts its computation in the first cell of its unique tape, where, in the first $|x|$ cells we can find the input x made of As and Bs; $\mathcal{M}$ completes the copy in the leftmost symbol of the sequence xx, i.e., $\mathcal{M}$ halts with two copies of x, with no space between them, and the head in its original place, the first symbol of xx.

The Turing machine proceeds as follows: it reads an A, replaces it by an X and writes a $\dot{A}$ in the first blanc to the right; the machine reads B, replaces it by an Y and writes a $\dot{B}$ in the first blanc to the right; once the As and the Bs are gone, the machine rewrites each X and each $\dot{A}$ into an A and each Y and $\dot{B}$ into a B.

This example gives to the reader the flavour of a Turing machine computation. The reader may well exercise herself specifying Turing machines that compute the trivial operations of arithmetics, having their arguments in unary [7] and any two arguments separated by the symbol of the operation.[8]

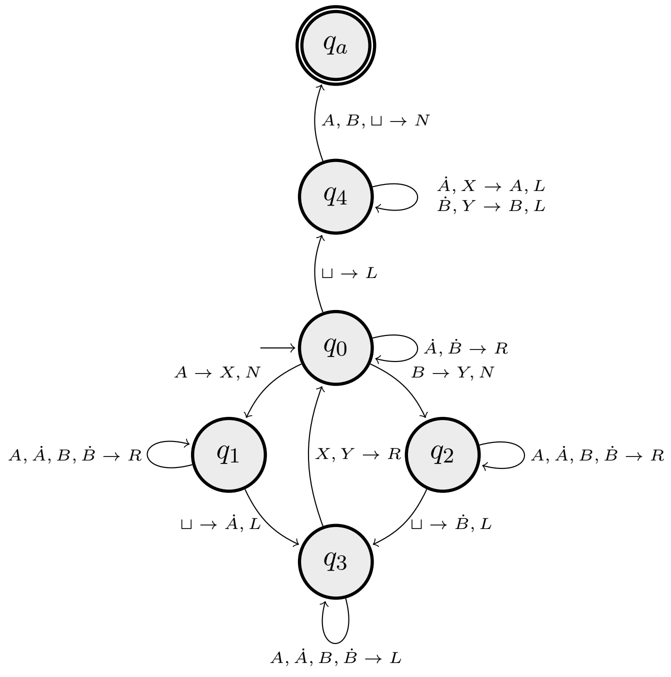

Figure 2: Example of Turing machine — the copier. The initial state is identified by an arrow with target but no source and the accepting state is identified by the double circle.

The second example is based on the concept of a guess. Suppose that a given problem has one solution in a finite space of potential solutions, but we do not know an algorithm to construct the actual solution. We only know how to verify if a potential solution is the actual solution of the problem. In this case all potential solutions are guesses. By trial and error we are sure to find the actual solution: we guess the solution and we verify if it is the actual one. The verification procedure is in itself a deterministic procedure, but the guess part is a non-deterministic procedure. Note that NOT all non-deterministic Turing machines clocked in some class of bounds can be seen has a sequence of guess and verification! But the machines that have computations of size bounded by a polynomial can! (It is a theorem.)

[7] That is, the number n is denoted by a sequence of n 1s. Another way of doing it, to avoid the empty sequence denoting the number 0, is to represent n by a sequence of $n+1$ 1s.

[8] Such like $111 + 11 = 11111$ or $111 \times 11 = 1111111$.

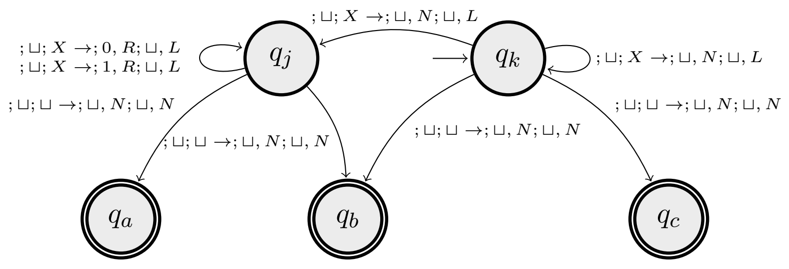
$; \sqcup; X \to; \sqcup, N; \sqcup, L$
$; \sqcup; X \to; 0, R; \sqcup, L$
$; \sqcup; X \to; 1, R; \sqcup, L$
q_j
q_k
$; \sqcup; X \to; \sqcup, N; \sqcup, L$
$; \sqcup; \sqcup \to; \sqcup, N; \sqcup, N$
$; \sqcup; \sqcup \to; \sqcup, N; \sqcup, N$
$; \sqcup; \sqcup \to; \sqcup, N; \sqcup, N$
$; \sqcup; \sqcup \to; \sqcup, N; \sqcup, N$
q_a
q_b
q_c

Figure 3: Transition diagram of a non-deterministic Turing machine that generates a guess of size at most $p(n)$, where p is polynomial and n the size of the input. The machine has 3 tapes: initially, using the constructibility of the polynomial p, the machine delimitates the space of $p(n)+1$ symbols X in tape 3, making one transition to the state q_k; in the state q_k, the machine can initiate the production of the guess in tape 2, making one transition to the state q_j, or consuming several Xs in this decision, with the production of smaller guesses; in the limit, the machine makes a transition to the state q_b, corresponding to the the empty guess (ε), or to the state q_c that does not correspond to any guess ($\emptyset$). In this guessing procedure the input tape is not in use.

Let us see the mechanism of generating guesses.

Given an input x of size n, the Turing machine $\mathcal{M}$ specified by the transition diagram in Figure 3 uses the time constructibility of polynomial p in order to deterministically delimit $p(n)+2$ cells with the symbol X, let us say $\sqcup X^{p(n)+2}$, where n is the size of the input, in a working tape, leaving a leftmost blanc cell and the head in the rightmost X. After this delimitation of space, the machine generates one guess of size at most $p(n)$.

For the guessing part, the Turing machine displays a tree of possible $2^{p(n)+2}$ computations of size $p(n)+2$; after generating the guess the Turing machine starts the verification part of the overall computation. For the pure non-deterministic reasoning it is not needed to generate the guesses in this way, in computations of fixed size. But for general applications, such like the use of probabilistic Turing machines in complexity analysis (see Section 4), it is, indeed, necessary to have all guesses generated by computations of the same size, even if the guess has size 1 within guesses of size n^2, where n is the size of the input. The reason for this is that all guesses are equiprobable outcomes of the randomizer, so that they should have the same weight implying the same computation depth.

For each transition of the Turing machine $\mathcal{M}$, the finite control erases one of the Xs and moves the corresponding writing head to the left. The procedure terminates when all of the Xs have been erased. Erasing an X in each transition, the Turing machine can delay the production of the guess in such a way that all the words of size less than $p(n)$ may be written.

Note that in Figure 3, there are 3 "accepting states"[9] that only denote the termination of the guess generating procedure in 3 different situations.

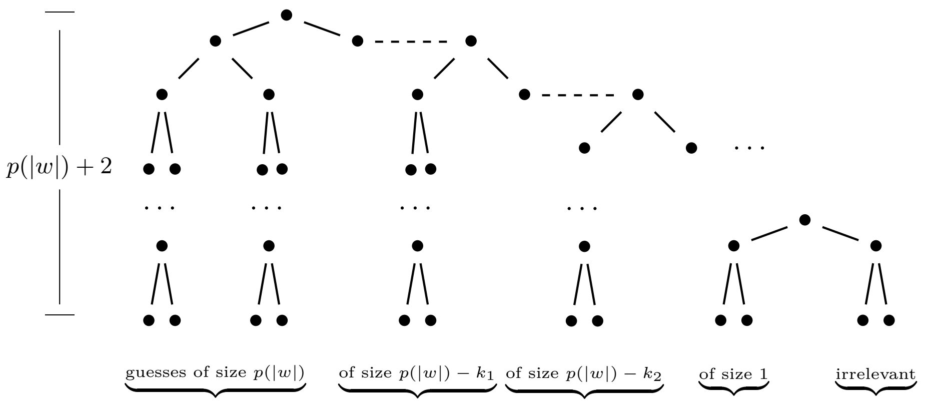

Figure 4: Tree of computations relative to the non-deterministic Turing machine of Figure 3. All the transitions, except those of rightmost branch, correspond to guesses of size $p(|w|)$, $p(|w|) - 1$, $p(|w|) - 2$, ..., 1. In each level in the rightmost branch, the transition to the left corresponds to a productive decision; the transition to the right corresponds to a delay in the production of a smaller guesses. The last two transitions in the bottom level are irrelevant in the non-deterministic computation, they just should lead to the rejecting state; however, in the case of probabilistic computation, they should lead to equal number of computations ending in the rejecting and the accepting states, in such a way that these branches do not count for the probabilistic decision; this is trivially done by extending the computations to the limit of time (see Section 4), accepting those to the right and rejecting those to the left.

The transition diagram of Figure 3 illustrates this procedure: it presupposes two working tapes; in one tape $\mathcal{M}$ counts the number of transitions made; in the other tape, it writes the guess. The first tape in Figure 3 is any reference tape, such as the input tape, where the symbol being read is irrelevant.

The tree of computations of $\mathcal{M}$ relative to the input x is depicted in Figure 4. Note that the tree has $2^{p(n)+2}$ branches of depth $p(n) + 2$, corresponding to different guesses; the two last branches are special. There is a convention about these branches: (a) in the case of non-deterministic machine, both branches produce rejecting computations; (b) in the case of the probabilistic machine, the penultimate branch produces accepting computations and the last branch produces rejecting computations. The reason for this

[9]It can only be just one accepting state and just one rejecting state according with our definition of a Turing machine.

convention is the necessity of counting the accepting branches and the rejecting branches: in this case, the number of accepting children of the penultimate branch is equal to the number of rejecting children of the last branch.

3 Proper functions of time and space

In this section we introduce the concepts of proper functions of time and space, used to define complexity classes of Turing machine computations. These concepts (a slight variation of them) were introduced by Hartmanis and Stearns, Stearns, Hartmanis and Lewis and Lewis, Stearns and Hartmanis in 1965 (see [8, 10, 16]). Since then, different authors, such like Hopcroft and Ullman (see [1]) have been referring to the concepts without illustration. We herein show such concepts implemented for the purpose of education, although, in Section 7, we state the two main theorems about such functions.

Definition 2 *A (total) function $t : \mathbb{N} \to \mathbb{N}$ is said to be a* proper function of time *if there is a deterministic Turing machine $\mathcal{M}$ and a number $p \in \mathbb{N}$ such that, for all inputs of size $n \geq p$, $\mathcal{M}$ halts (accepting) in exactly $t(n)$ steps.*

Proper functions of time are also called *time constructible*. A similar definition to Definition 2 can be given for space:

Definition 3 *A (total) function $s : \mathbb{N} \to \mathbb{N}$ is said to be a proper function of space if there exists a deterministic Turing machine $\mathcal{M}$ and a number $p \in \mathbb{N}$ such that, for all inputs of size $n \geq p$, $\mathcal{M}$ halts (accepting) in a configuration in which exactly $s(n)$ cells are non blank and, moreover, no more cells were used during the computation.*

Proper functions of space are also called *space constructible*. Figure 5 illustrates a Turing machine with bounded space resources $2s(n)$ for inputs of size n. We will pay more attention to clocks than to rulers.

Polynomials and exponentials are time constructible and related to very well known complexity classes. Some well known time bounds are:

$$|x|^k, \quad 2^{k|x|} \quad \textit{and} \quad 2^{|x|^k} . \tag{1}$$

where x is the input and $|x|$ is the size of the input (the number of symbols in the input word).

Originally the concept of time constructible function was introduced in two varieties. A function $t(n)$ was considered to be time constructible if there exists a $t(n)$ bounded multitape Turing machine $\mathcal{M}$ such that for each n there exists some input on which $\mathcal{M}$

actually makes $t(n)$ transitions. We say that $t(n)$ is fully time constructible if there is a Turing machine that uses $t(n)$ time on all inputs of length n.

The use of clocks result from the fact that simply externally counting the steps of a Turing machine computation is inadequate. The Turing machine *itself* should count the number of steps and switch itself off in the last step. To make step counting and time explicit inside the machine we need to introduce a *system alarm clock* that must be defined as part of the Turing machine.

We can install alarm clocks in all our (let us say deterministic) Turing machines as follows. Let $\mathcal{M}$ be an arbitrary Turing machine. Let $\mathcal{M}_{clock}$ be a Turing machine that witnesses the time constructibility of some total function. We can construct a new Turing machine that is a parallel composition of machines $\mathcal{M}$ and $\mathcal{M}_{clock}$. We just have to add to $\mathcal{M}$ the tapes of $\mathcal{M}_{clock}$ and change the finite control of $\mathcal{M}$, enlarging it with the finite control of $\mathcal{M}_{clock}$. Now, if $\mathcal{M}$ reaches an halting state before the clock $\mathcal{M}_{clock}$ alarms, then we consider it a halting state, and the composite machine accepts or rejects according to the decision of the machine $\mathcal{M}$. Otherwise, if the clock $\mathcal{M}_{clock}$ of the composite machine reaches a halting (accepting) state first, before $\mathcal{M}$ halts, then the composite machine rejects; the computation is *timed out*.

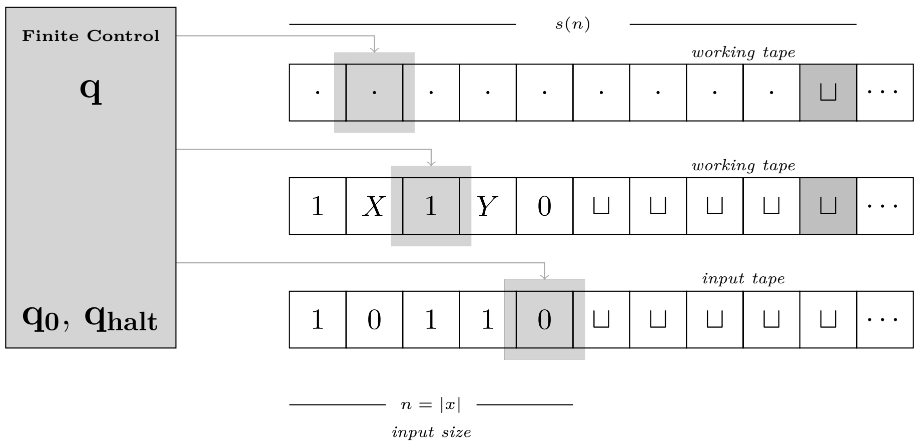

Figure 5: A deterministic Turing machine with limited resources in space: for every n, the working space is limited by $2s(n)$ cells in each working tape.

We note that this method allows us to make effective enumerations of clocked Turing machines, a mathematical tool to proceed with explorations in computational complexity. If $\mathcal{M}_0$, ..., $\mathcal{M}_n$, ... is an enumeration of all Turing machines, then the new machines resulting from the (algorithmically constructible) parallel combination of the previous machines with $\mathcal{M}_{clock}$, i.e., $\mathcal{M}_0||\mathcal{M}_{clock}$, ..., $\mathcal{M}_n||\mathcal{M}_{clock}$, ..., is an enumeration of all sets decided in the time provided by the given clock. If we want the class of sets decidable in polynomial time, then we just have to proceed to a careful enumeration. Let $\mathcal{M}_{n^i}$ denote the polynomial clock with exponent $i > 2$. Such an enumeration can be done as follows:

$$\mathcal{M}_0||\mathcal{M}_{n^2}, \mathcal{M}_0||\mathcal{M}_{n^3}, \mathcal{M}_1||\mathcal{M}_{n^2}, \mathcal{M}_2||\mathcal{M}_{n^2}, \mathcal{M}_1||\mathcal{M}_{n^3}, \mathcal{M}_0||\mathcal{M}_{n^4},$$
$$\mathcal{M}_0||\mathcal{M}_{n^5}, \mathcal{M}_1||\mathcal{M}_{n^4}, \mathcal{M}_2||\mathcal{M}_{n^3}, \mathcal{M}_3||\mathcal{M}_{n^2}, \mathcal{M}_4||\mathcal{M}_{n^2}, \mathcal{M}_3||\mathcal{M}_{n^3},$$
$$\ldots$$

Although this construction produces many (infinite to be precise) equivalent machines along the sequence (why?), and thus repeating the sets decided by the machines in the sequence, it is a major intellectual step in the theory of complexity, for it allows to perform a theoretical task known as *diagonalisation over a complexity class* (e.g. *NP* or *PSPACE*, see Section 4). And it can be done just because Turing machines can be specified to act as alarm clocks and rulers, so much for such a little thing as a Turing machine.

4 Models of computation

The deterministic and non-deterministic Turing machines are just *versions* of the same concept as we will elucidate in this section.

We first define the concept of a *canonical* Turing machine:

We impose the following conditions on a Turing machine: (a) every step of a computation can be made in exactly two possible ways that are considered different even if there is no difference in the corresponding transitions (this distinction corresponds to two different bit guesses of an irrelevant variable); (b) the machine is clocked by some time constructible function and the number of steps in each computation is exactly the number of steps allowed by the clock; (c) every computation ends in the *accepting* or the *rejecting* states.

Every specification of a clocked Turing machine, as discussed in Section 3, can be rewritten in order to satisfy the conditions of the previous paragraph.

The reader may find it odd to have a deterministic Turing machine that at each step of computation have two choices of transitions. These two transitions are "clones", but are they necessary? The reason of this more sophisticated definition is that it makes a clear separation between the Turing machine and the different accepting criteria.

Definition 4 *A set A is decided by a* deterministic Turing machine *in constructible time t if all computations (of length $t(|x|)$) end in an accepting state whenever $x \in A$ and all computations (with length $t(|x|)$) end in an rejecting state whenever $x \notin A$.*[10]

Definition 5 *[Rabin and Scott introduced the concept of non-determinism in 1959 (see [14])] A set A is decided by a* non-deterministic Turing machine *in constructible time t if there exists a computation (of length $t(|x|)$) ending in an accepting state whenever $x \in A$ and all computations (with length $t(|x|)$) end in an rejecting state whenever $x \notin A$.*[11]

$$\begin{tikzcd}
 & & & & \mathit{NPSPACE} & \\
 & & \mathit{NP} \arrow[rr, "\subseteq"] & & \mathit{PSPACE} \arrow[u, "="] \arrow[r, "\subset"] \arrow[d, "\subseteq"] & \mathit{EXPSPACE} \\
\mathit{LOG} \arrow[r, "\subseteq"] & \mathit{NLOG} \arrow[r, "\subseteq"] \arrow[uurrr, bend left, "\subset"] \arrow[d, dash, "="] & P \arrow[u, "\subseteq"] \arrow[r, "\subset"] & \mathit{DEXT} \arrow[r, "\subset"] & \mathit{EXPTIME} \arrow[r, "\subseteq"] & \mathit{NEXPTIME} \\
 & co - \mathit{NLOG} & & & &
\end{tikzcd}$$

Figure 6: Diagram representing the web of structural relations between the most well known complexity classes. Edges denote equality and directed edges denote inclusion. Information about the strict inclusions are provided by the labels.

If we restrict the time to the class of polynomials, then Definition 4 gives the class of sets P and Definition 5 the class NP. If we restrict the bounds to the class of exponentials of expression 2^{kn}, for $k \in \mathbb{N}$, then Definition 4 gives the class $DEXT$ and Definition 5 the class $NEXT$. And if we restrict the time to the class of exponentials of expression 2^{n^k}, for $k \in \mathbb{N}$, then Definition 4 gives the class $EXPTIME$, Definition 5 the class $NEXPTIME$. Similar definitions hold for space, namely the polynomial bound introduces $PSPACE$ and $NPSPACE$, for the deterministic and non-deterministic cases, respectively, and the first exponential bound introduces $EXPSPACE$.[12] The space complexity classes for the

[10]The number of computations is $2^{t(|x|)}$.

[11]The number of computations is the same, i.e., $2^{t(|x|)}$.

[12]For mathematical and historical reasons $EXPSPACE$ refers to the class of exponentials of expression 2^{kn}, for $k \in \mathbb{N}$, and not to the class of exponentials of expression 2^{n^k}, for $k \in \mathbb{N}$, being incoherent with the corresponding designation of time class $EXPTIME$.

logarithm bound (which can not exist for time (why?)) are designated by LOG and $NLOG$, for the deterministic and non-deterministic cases, respectively. All these classes are related through the web of inclusions of Figure 6.[13]

We can proceed now with further relevant models of polynomial time Turing machines:

Definition 6 *[Concept introduced by de Leeuw, Moore, Shannon, and Shapiro in 1956 and fully developed by Gill in 1972 in his PhD dissertation (see [4, 6, 7])] A set A is decided by a* probabilistic Turing machine *in constructible time t if at least* $2^{t(|x|)+1}$ *computations (of length* $t(|x|)$*) end in an accepting state whenever* $x \in A$ *and strictly less than* $2^{t(|x|)} + 1$ *computations end in an rejecting state whenever* $x \notin A$.[14]

Definition 7 *[Concept introduced by Gill in 1972 in his PhD dissertation (see [6, 7])] A set A is decided by a* bounded probabilistic Turing machine *in constructible time t if there exists a dyadic rational* $0 < \varepsilon < \frac{1}{2}$ *such that the number of computations (of length* $t(|x|)$*) ending in an accepting state is greater or equal to* $(\frac{1}{2} + \varepsilon)2^{t(|x|)}$ *whenever* $x \in A$ *and the number of computations (of length* $t(|x|)$*) ending in an rejecting state is greater or equal to* $(\frac{1}{2} + \varepsilon)2^{t(|x|)}$ *whenever* $x \notin A$.[15]

Definition 8 *[Concept introduced by Gill in 1972 in his PhD dissertation (see [6, 7])] A set A is decided by a* Rabin bounded probabilistic Turing machine *in constructible time t if there exists a dyadic rational* $0 < \varepsilon < \frac{1}{2}$ *such that the number of computations (of length* $t(|x|)$*) ending in an accepting state is greater or equal to* $(\frac{1}{2} + \varepsilon)2^{t(|x|)}$ *whenever* $x \in A$ *and all the computations (of length* $t(|x|)$*) end in an rejecting state whenever* $x \notin A$. [16]

Now, if we consider three possible final states in a Turing machine, *accept*, *reject*, and *don't know*, then we can still add a further class of probabilistic Turing machines:

Definition 9 *[Concept introduced by Gill in 1972 in his PhD dissertation (see [6, 7])] A set A is decided by a* Z bounded probabilistic Turing machine *in constructible time t if there exists a dyadic rational* $0 < \varepsilon < \frac{1}{2}$ *such that, whenever* $x \in A$*, the number of computations (of length* $t(|x|)$*) ending in an accepting state is greater or equal to* $(\frac{1}{2} + \varepsilon)2^{t(|x|)}$ *and all the other computations* ($(\frac{1}{2} - \varepsilon)2^{t(|x|)}$ *computations of length* $t(|x|)$)

[13]If $\mathcal{C}$ is a class, then $co - \mathcal{C}$ denotes the class of its complements. Each deterministic class $\mathcal{C}$ coincides with its dual or co-class: $\mathcal{C} = co - \mathcal{C}$. For the non-deterministic classes, it is an open problem to know if $\mathcal{C} \overset{?}{=} co - \mathcal{C}$.

[14]Ibidem.

[15]Ibidem.

[16]Ibidem.

end in the don't know *state and, whenever $x \notin A$, the number of computations (of length $t(|x|)$) ending in an rejecting state is greater or equal to $(\frac{1}{2}+\varepsilon)2^{t(|x|)}$ and all the other computations ($(\frac{1}{2}-\varepsilon)2^{t(|x|)}$ computations of length $t(|x|)$) end in the* don't know *state.*

Thus all these classes can be defined in the same way; proofs rely on the counting of accepting computations, rejecting computations, and computations ending in the state *don't know*. Each computation is generated by a random binary sequence of the same size. Thus, if the clock is adjusted to time t, we will end up with a tree of $2^{t(n)}$ computations for inputs of size n, corresponding to all binary sequences of size $t(n)$.

Definition 6 gives the class of sets *PP*, Definition 7 the class *BPP*, Definition 8 the class R (called *VPP* in Gill's PhD dissertation of 1972 [17] (see [6, 7])), and Definition 9 gives the class *ZPP*. Moreover, either *BPP* or R correspond to the so-called *Monte Carlo algorithms* and *ZPP* corresponds to the *Las Vegas algorithms*. These classes and their duals are related through the web of inclusions of Figure 7.[18]

5 Linear, polynomial and exponential time

Now, so far so good!, but we never saw a Turing machine specification of a time constructible or a space constructible function, maybe for historical reasons, but for other reasons as well that will be explained in section 7.[19]

5.1 Linear clock

We start with the linear clock, the Turing machine which on inputs of size n halts exactly in $k \times n$ steps, for $k \in \mathbb{N}$. We will consider first the general case of the function $t_k(n) = k \times n$, for $k \geq 4$, then we will refer to the special clocks $t_2(n) = 2n$ and $t_3(n) = 3n$. Note that the clock $t_1(n) = n$ does not exist, since this function amounts to less than the time needed for the machine to read its input, i.e., $n+1$ steps.[20] Moreover, for the general case $t_k(n) = k \times n$, with $k \geq 4$, we will consider first the case $n \geq 4$ and then the most general case using patching techniques.

The machine specified in the transition diagram of Figure 8 satisfies the requirements for $n \geq k$. Note that the cardinality of the input alphabet is irrelevant, only the size of the input really matters. The machine has three tapes, one input read-only tape and two working tapes, the first one designated the "dance" tape and the second the counting

[17]One of the reasons why this class is denoted by R of Rabin is the fact that Rabin was the first to develop a powerful polynomial time probabilistic algorithm, namely the algorithm to decide if a given number is a prime number (see [13]), based on the work of Miller (see [11]).

[18]See footnote 13.

[19]Surely, such specifications have been done all over the world.

[20]And so, the time function of expression $t'_1(n) = n+1$ is, indeed, time-constructible.

tape. Once given the value of k, the specification of the Turing machine of Figure 8 is complete. It can be retrieved for an arbitrary value of k. It works as follows:

$$\begin{tikzcd}
& & R \arrow[r] \arrow[dr] & NP \arrow[dr] & & \\
P \arrow[r] & ZPP \arrow[ur] \arrow[r] \arrow[dr] & NP \cap co-NP \arrow[ur] \arrow[dr] & BPP \arrow[r] & PP \arrow[r] & PSPACE \\
& & co-R \arrow[r] \arrow[ur] & co-NP \arrow[ur] & &
\end{tikzcd}$$

Figure 7: Diagram representing the web of structural relations between the most well known polynomial time complexity classes. Edges denote equality and directed edges denote inclusion.

1. The machine copies the input to the "dance" tape: in this task, the machine marks the first symbol being copied with a $\breve{0}$, the second with a $\acute{0}$, the next ones with 0s; once found the blank symbol at the end of the input, the machine copies it as a $\breve{0}$ and turns left revisiting the penultimate cell. In this task, the machine takes $n+1$ steps.

2. Turning back, the head in the "dance" tape, marks the penultimate symbol 0 by $\grave{0}$, and enters into the main "dancing cycle".

3. While in the copying procedure 1, the head on the counting tape is writing the sequence YX^{k-2} and stopping at the last X — one "dance" was just performed and $(k-1)$ "dances" are still to be made. Since we are assuming $n \geq k$, this routine can be done at the same time as procedure 1. Note that k, a fixed value for each such Turing machines, is a value registered in the finite control of the machine. This task ends with the head in the last X and possibly having the other head on the "dance" tape still performing its job (namely for $n \geq k$).

4. Every time a "dance" is completed, i.e., every time the "dance" head reaches one of the extremes of the word written in the tape, the counting head moves an X to the left. The finite control knows that, once having reached the Y, the "dance" that is now to be made is the last one. In fact, the machine will have visited the

$n+1$ written cells of the "dance" tape performing n (in step 1) + $n \times (k-1) = kn$ steps of computation.

5. However, in the last step, the machine can not visit the last written cell of the "dance" tape, or else would need a further step to halt, making the clock tick once more. Instead, it halts in a single transition, once reached the penultimate position in that direction, $\acute{0}$ if moving from the right to the left or $\grave{0}$ if moving from the left to the right.

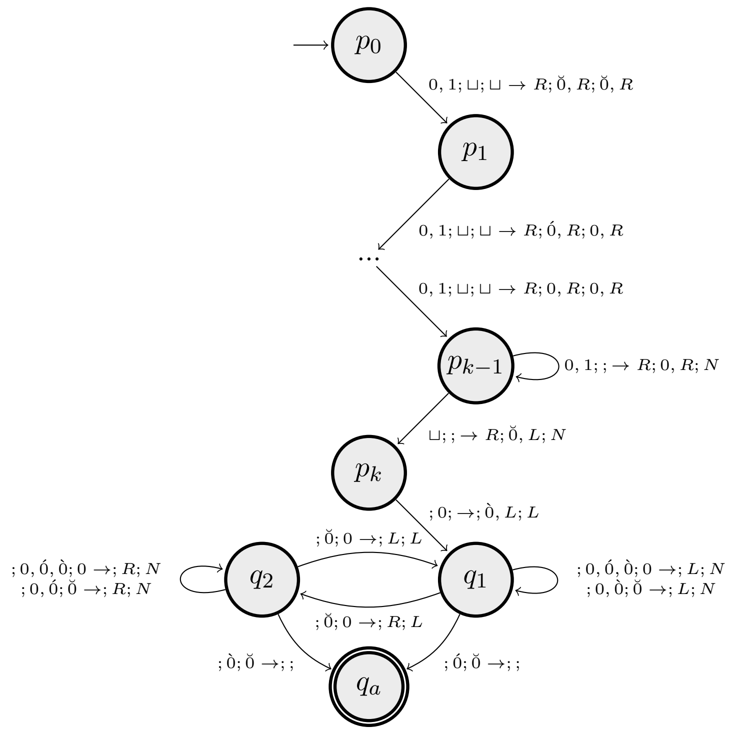
p_0
$0, 1; \sqcup; \sqcup \to R; \breve{0}, R; \breve{0}, R$
p_1
$0, 1; \sqcup; \sqcup \to R; \acute{0}, R; 0, R$
$\cdots$
$0, 1; \sqcup; \sqcup \to R; 0, R; 0, R$
p_{k-1}
$0, 1; ; \to R; 0, R; N$
$\sqcup; ; \to R; \breve{0}, L; N$
p_k
$; 0; \to; \grave{0}, L; L$
$; \breve{0}; 0 \to; L; L$
$; 0, \acute{0}, \grave{0}; 0 \to; R; N$
$; 0, \acute{0}; \breve{0} \to; R; N$
q_2
q_1
$; 0, \acute{0}, \grave{0}; 0 \to; L; N$
$; 0, \grave{0}; \breve{0} \to; L; N$
$; \breve{0}; 0 \to; R; L$
$; \grave{0}; \breve{0} \to; ;$
$; \acute{0}; \breve{0} \to; ;$
q_a

Figure 8: Transition diagram of a Turing machine that makes exactly $k \times n$ transitions, for $n \geq k$, $k \geq 4$ fixed. For $k = 1$ the function is not time constructible. For $k = 2, 3$, we can specify simplified machines witnessing the time constructibility of $2n$ and $3n$, that execute 2 and 3 dances, respectively. Note that in the transition diagram, the second transition between states p_1 and p_{k-1} is the transition from p_{k-2} to p_{k-1}, and not the transition from p_1 to p_2. The initial state is p_0 and the accepting state is q_a.

For each value of k, the schematic machine of Figure 8 can be modified in a way such that all the values of n less than k are treated separately by patching in the finite control the finite automaton of the exceptions. However, it is worth emphasising that only the asymptotic behaviour of the machine is relevant, e.g., that there exists an order after which the machine behaves like a perfect clock.

This clock was verified in MATHEMATICA and works perfectly!

The clocks for the cases $k = 2, 3$ (for all $n \in \mathbb{N} - \{0\}$) are simple. We illustrate the second in Figure 9 and leave the specification of the first to the reader.

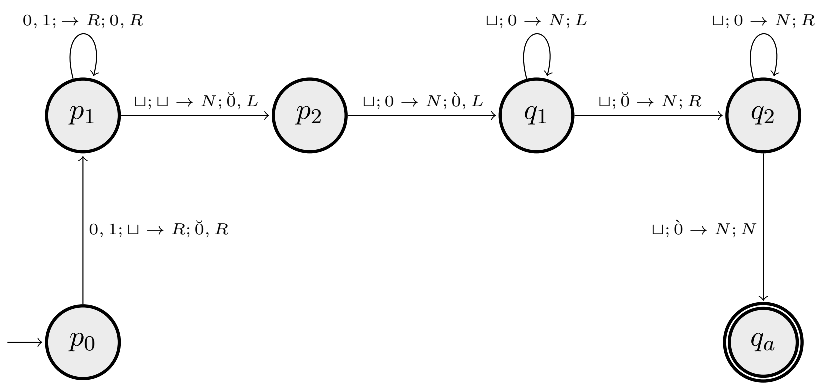
$0, 1; \to R; 0, R$
$\sqcup; 0 \to N; L$
$\sqcup; 0 \to N; R$
p_1
$\sqcup; \sqcup \to N; \breve{0}, L$
p_2
$\sqcup; 0 \to N; \grave{0}, L$
q_1
$\sqcup; \breve{0} \to N; R$
q_2
$0, 1; \sqcup \to R; \breve{0}, R$
$\sqcup; \grave{0} \to N; N$
p_0
q_a

Figure 9: Transition diagram of a Turing machine with two tapes that makes exactly $3n$ transitions before halting, for $n \geq 2$. The particular cases $n = 1, 2$ can be included by patching these two exceptions in the finite control. The initial state is p_0 and the accepting state is q_a.

5.2 Polynomial clock

Without loss of generality, we can assume that the input is written in unary since only its size really matters.[21]

Figure 10 displays the transition diagram of a 3-tape Turing machine that makes exactly n^2 transitions, for all $n \in \mathbb{N}$ greater than 3.[22]

As we saw, in computational complexity it is irrelevant what happens for a finite number of inputs, since only the asymptotic behaviour of Turing machines matters.[23] A

[21]Herein, we use input words of 0s. Thus either being in binary or written with a different alphabet what is copied to the working tapes is the same sequence of 0s (plus or minus one 0), possibly marked in some positions.

[22]I.e., for words of length at least 3 bits. For words of size less than 3 the problem can be solved by incorporating an automaton in the finite control of the Turing machine. This "delay" in the behaviour of these Turing machines that witness time constructibility is due to the necessity of preparing the tapes for the "dances" of the reading heads.

[23]This is the reason why the complexity classes are closed under finite variants, i.e., if A is in some complexity class $\mathcal{C}$, then if $A \Delta B = (A - B) \cup (B - A)$ is a finite set, then also B belongs to the same class $\mathcal{C}$.

finite set is always decidable by a finite automaton. Such an automaton can be added to the finite control of a Turing machine to solve the problem of finitely many exceptions.

Figure 12 displays the transition diagram of the exceptions of the Turing machine specified in Figure 10. Exception of exceptions is the value $n = 1$, since $1^2 \not> 1$ and, as we said above, every time constructible function t should be superlinear, i.e., redefined as $t(n) := \max\{t(n), n+1\}$. The function of expression $n+1$ is already time constructible.

The reader may notice that the Turing machine that witnesses the time constructibility of the function of expression n^2 is very similar to the one that witnesses the time constructibility of $k \times n$, as the comparison between Figures 8 and 12 show. One has only to replace k by n and follow the transition diagram, paying attention to a few further details.

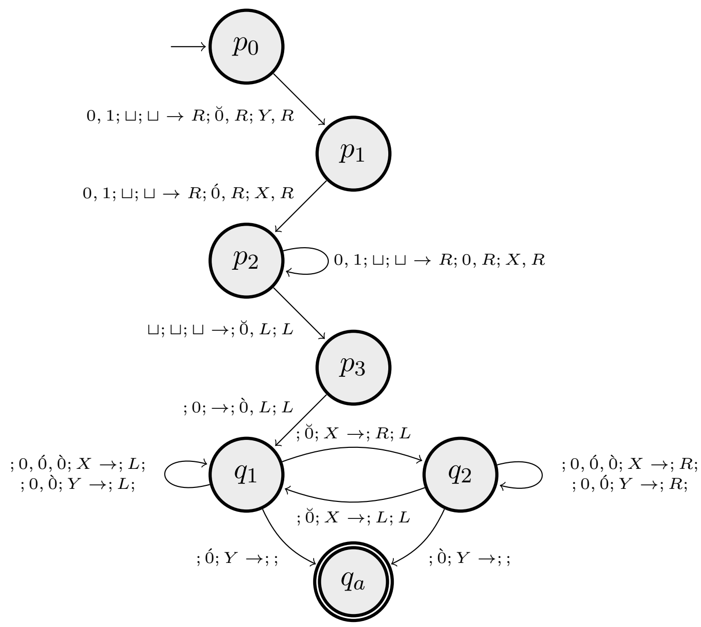
p_0
$0, 1; \sqcup; \sqcup \rightarrow R; \breve{0}, R; Y, R$
p_1
$0, 1; \sqcup; \sqcup \rightarrow R; \acute{0}, R; X, R$
p_2
$0, 1; \sqcup; \sqcup \rightarrow R; 0, R; X, R$
$\sqcup; \sqcup; \sqcup \rightarrow; \breve{0}, L; L$
p_3
$; 0; \rightarrow; \grave{0}, L; L$
$; \breve{0}; X \rightarrow; R; L$
$; 0, \acute{0}, \grave{0}; X \rightarrow; L;$
$; 0, \grave{0}; Y \rightarrow; L;$
q_1
q_2
$; 0, \acute{0}, \grave{0}; X \rightarrow; R;$
$; 0, \acute{0}; Y \rightarrow; R;$
$; \breve{0}; X \rightarrow; L; L$
$; \acute{0}; Y \rightarrow; ;$
$; \grave{0}; Y \rightarrow; ;$
q_a

Figure 10: Transition diagram of a Turing machine witnessing the constructibility of the function of expression n^2, for all input size $n \in \mathbb{N}$ such that $n \geq 4$. The initial state is p_0 and the accepting state is q_a.

We discuss now how to implement a polynomial alarm clock in general. The idea of one possible construction is as follows.

Notice the recurrences: $n^k = n \times (n^{k-1} - n^{k-2}) + n^{k-1} = n^{k-1}(n-1) + n^{k-1}$, with $k \geq 2$.

The input is copied to the working tape number 2 and marked (while doing the copy) in the first cell and in the first blanc cell to the right of the copy, in a way such that it can be reused during the computation. Also marked are the second and the penultimate positions (1 step after marking the first blanc cell). Another copy of the input, this time reduced in one symbol, is copied in the same way, to the tape number 3 (therefore, of size n for inputs of size n, implying that one "dance" in this tape corresponds to $n-1$ transitions), marked as the previous in the first, second, last and first blanc cell after it.

The input of size n is also copied to the working tape number 4 as a sequence $YX\dots X$ of size $n-1$: for each X, the copy of the input of size $n+1$ in tape 2, $\breve{0}\acute{0}0\dots0\grave{0}\breve{0}$, is swiped in a single "dance" (of n transitions). In the end of all these activities, n^2 transitions are counted in the tape number 5, through a sequence $YX\dots X$ of size n^2, that was initiated in the beginning of the computation.

The same activity is executed involving the tapes 3 and 5, counting the transitions in the working tape number 6 that, in this way, holds the sequence $YX\dots X$ of size $n^3-n^2+n^2=n^3$ that is being written since the beginning of the computation. Working with tapes 3 and 6, the machine makes $(n-1)\times n^3=n^4-n^3$ new transitions that added to n^3 precedent transitions, sum up to n^4 transitions, counted in the working tape number 7, in the sequence $YX\dots X$ of size $n^4-n^3+n^3=n^4$, that was initiated in the beginning of the computation.

Repeating this procedure until the last step of the cycle from n to n^k, the Turing machine makes n^k transitions, not being necessary to register this value in any tape.

Note that the initial symbols Y in the sequences are written in the first transition of the machine from the initial state. All the transitions are being counted in the tapes: until n^3 in tape 5, until n^4 in tape 6, ..., until n^{k-1} in tape $k+1$. $k+1$ working tapes are needed (with k fixed) for this computation of n^k steps.

5.3 Exponential clock

We prove last that the exponential function of expression 2^n is time constructible. We will consider the 3-tape deterministic Turing machine, with the transition diagram depicted in Figure 11, as witness of that fact. In the horizontal line, we find a sequence of states p_0, p_1, p_2 and q_1 that solve the particular cases for $n=0$, $n=1$ and $n=2$: for the non-admissible case of the empty input sequence, the machine accepts in one transition ($2^0=1$); for the case of an input of size 1, the machine accepts in two steps ($2^1=2$); for the case of an input word of size 2, the machine accepts in 4 transitions ($2^2=4$); for inputs of greater size, the machine undergoes a cycle of $n-2$ steps.

We consider the step of three transitions of the loop specified in Figure 11. Every number of the form 2^n, for $n\geq 2$, can be decomposed into a sum

$$2^n=2^0+2^1+\left(\sum_{i=1}^{n-1}2^i\right)+1\ , \tag{2}$$

where the two first terms count the transitions from p_0 to q_1, the last term 1 counts the final transition to the halting (accepting) state; for $n = 2$, it makes $1 + 2 + 1$ transitions ($2^2 = 4$). The composite term in the summation is computed in the cycle: in the tape 2 we have a number written in unary, starting with 1. The machine copies the tape 2 into the tape 3, always rewriting the tape 3, then copies the tape 3 into the right of the 1s in tape 2, duplicating the former value in each step of the cycle. The halting condition of this cycle is the detection of the first blanc cell in the input tape. Two symbols of the input tape have been read before the step of the cycle is executed for the first time. From an input of n symbols, we subtract 2 and add 1 (since the step of the cycle is advanced), what amounts to the counting of formula 2.

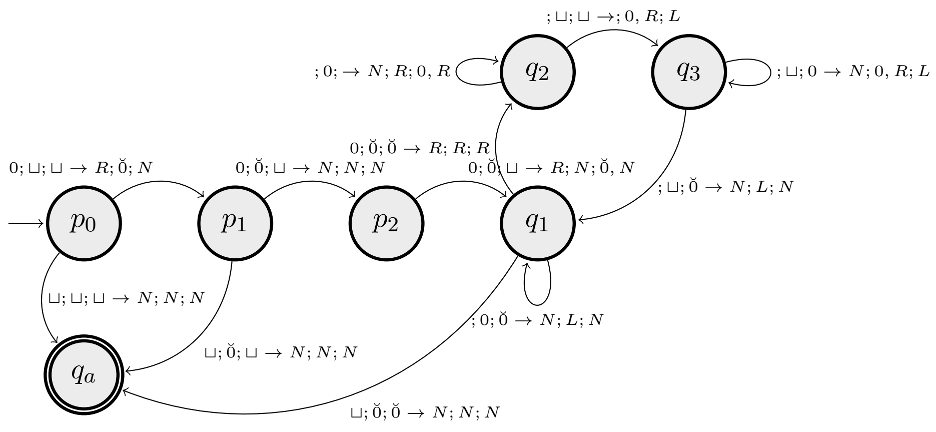
$; \sqcup; \sqcup \to; 0, R; L$
$; 0; \to N; R; 0, R$
q_2
q_3
$; \sqcup; 0 \to N; 0, R; L$
$0; \breve{0}; \breve{0} \to R; R; R$
$0; \sqcup; \sqcup \to R; \breve{0}; N$
$0; \breve{0}; \sqcup \to N; N; N$
$0; \breve{0}; \sqcup \to R; N; \breve{0}, N$
$; \sqcup; \breve{0} \to N; L; N$
p_0
p_1
p_2
q_1
$\sqcup; \sqcup; \sqcup \to N; N; N$
$; 0; \breve{0} \to N; L; N$
$\sqcup; \breve{0}; \sqcup \to N; N; N$
q_a
$\sqcup; \breve{0}; \breve{0} \to N; N; N$

Figure 11: Transition diagram of a Turing machine witnessing the constructibility of the function of expression 2^n, for all $n \in \mathbb{N}$. We specify a 3-tape Turing machine with 1 input tape and 2 working tapes that executes exactly 2^n transitions until the halting state is reached. The initial state is p_0 and the accepting state is q_a.

Therefore, the cycle works as follows: to count $2^3 = 8$, we need 4 transitions external to the cycle. The computation comes into the cycle with one single symbol in each working tape with the last input symbol left unread. The input symbol is read. One symbol of the tape 2 is copied to the working tape 3, overwriting its content; then 1 symbol from tape 3 is copied to the tape 2 to the right, making a total of 2 symbols in tape 2; and the head of tape 2 moves back 2 cells, summing up a total of 4 transitions; $4 + 4 = 8$, and we are done for this case. To compute $2^4 = 16$, we need 4 transitions external to the cycle; the computation comes into the cycle and, after one step of the cycle, the machine, as we saw in the above example, made more 4 transitions, a total of 8 transitions, 8 transitions less than the required amount of 16 transitions; there are 2 symbols in the tape 2 and a last unread symbol in the input tape. The last input symbol is read. Two symbols are copied from tape 2 to tape 3, then 2 symbols from tape 3 to the tape 2, making a total of 4 symbols in tape 2; in the end, the reading head of tape 2 counts the 4 symbols, making the total of 8 transitions; $8 + 8 = 16$, and we are done for this case. And so forth for $n > 4$.

5.4 Patching exceptions into the finite control of a Turing machine

Once given a Turing machine $\mathcal{M}$ that "misbehaves" at a finite number of inputs w_1, ..., w_k, the corrected machine $\tilde{\mathcal{M}}$, if it exists, can be obtained by patching the desired outputs in the finite control of $\mathcal{M}$. We emphasize *if it exists* since "misbehaving" can be a problem of time, i.e., the exceptions can not be handle in the required time. E.g., for the time bound $t(n) = n^2$, the machine would have to halt in one transition for inputs of size 1, which is not possible, since two transitions are required to detect the ending of the input.

The Turing machine in Figure 12 witnesses such a patching for the exceptions of the general Turing machine of Figure 10. Note again that such an exception of an input of size 1 has no solution.

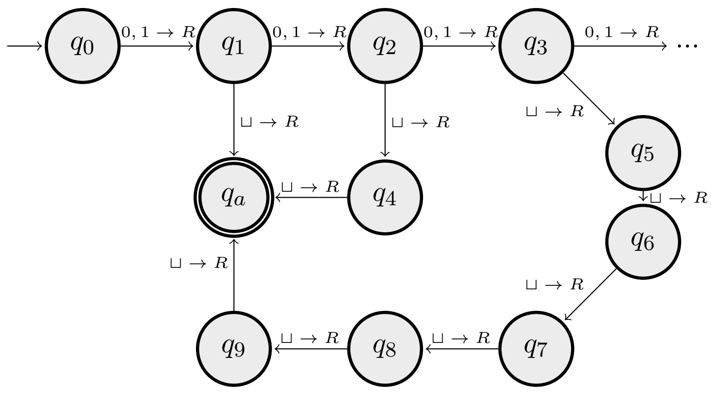

Figure 12: Transition diagram of a Turing machine that computes n^2 in n^2 transitions, for $n = 2, 3$. For the case $n > 3$ the computation progresses to the right of q_3. The labels refer only to the input tape. The initial state is q_0 and the accepting state is q_a.

6 The composition of proper functions of time

The composition of time constructible functions is still time constructible. Since we have proved that 2^n and n^k are time constructible bounds, this result implies that the class of bounds used to define the complexity class *EXPTIME*, 2^{n^k}, for $k > 1$, $k \in \mathbb{N}$, is a class of time constructible functions. It is also true that the closure of the class of constructible functions under product implies that the class of bounds used to define the complexity classes *DEXT* and *EXPSPACE*, 2^{kn}, for $k \geq 1$, $k \in \mathbb{N}$, are classes of time and space constructible functions, respectively.

Proposition 1 *The classes of proper functions of time and space are closed under addition, multiplication and composition.*

Proof: We will consider in this proof only the class of proper functions of time (time constructible functions), since we didn't pay to much attention to space constructibility. The proof is interesting for those readers that want to learn how to engineer Turing machines as a LEGO of diverse pieces — one of the ingredients of computational complexity.

Remember that Turing machines, when they reach the halting (accepting) state, they turn themselves off and no further transitions are permitted. Let $\mathcal{M}_f$ be the deterministic Turing machine witnessing the time constructibility of the function f and $\mathcal{M}_g$ the deterministic Turing machine witnessing the time constructibility of the function g. We have to specify a deterministic Turing machine witnessing the time constructibility of the function $f \circ g$.

We start by modifying $\mathcal{M}_g$ by including one more tape and enlarging the finite control in such a way that the new machine, let us say $\mathcal{M}$, writes in the new tape a symbol per transition of $\mathcal{M}_g$. It writes on the new tape the first symbol $\breve{0}$ and then, keeping the head moving to the right, a sequence of symbols 0, one per each new transition of $\mathcal{M}_g$. When the machine $\mathcal{M}_g$ makes the transition to the accepting state, the head of the new tape writes the last 0 and sill moves to the right; in a different set of tapes, in the same transition, initiates the computation of $\mathcal{M}_f$ which, instead of reading the input tape, starts reading the tape with the word $\breve{0}0\ldots0$ of size $g(n)$, where n is the size of the original input. This word is to be read from the right to the left and its end is detected with the reading of the symbol $\breve{0}$. The machine constructed in this way, incorporating the modified finite control of $\mathcal{M}_g$ and the finite control of $\mathcal{M}_f$, witnesses the time constructibility of $f \circ g$. The step to right that Turing machines make to detect the end of the input is a further step implemented in $\mathcal{M}$ by moving the head of the new tape to the left after the halting of $\mathcal{M}_g$. But there is still one problem: the machine $\mathcal{M}$ writes its $g(n)$ first steps simulating $\mathcal{M}_g$ and then $f(g(n))$ steps by simulating $\mathcal{M}_f$, i.e., in this way $\mathcal{M}$ makes more $g(n)$ steps than it was supposed to do. Then $\mathcal{M}$ has to make it differently. Since each such machine has to start by copying the input to a working tape, the machine $\mathcal{M}$ starts copying the input $0^{g(n)}$, with one transition delay to a another new tape, at the same time $\mathcal{M}_g$ is performing its computation writing the $g(n)$ 0s. Two transition after $\mathcal{M}_g$ has halted, the machine $\mathcal{M}_f$ detects the end of its input, the blanc space after the $g(n)$ 0s. This delay of two transitions can be avoided by propagating to the sequence of $g(n)$ 0s the markings on the "dancing" tape of $\mathcal{M}_g$.

The closure of the class of time constructible functions under addition and multiplication requires another construction common to both addition and multiplication: the finite control of the machine $\mathcal{M}_f$ is modified in such a way that it writes in a new tape, in parallel with the computation of the original machine, one copy of the input in the form $\check{0}0\ldots0$ of size n, where n is the size of the input, leaving the head in the first blanc cell to the right. This task can be done without difficulty since the detection of the blanc symbol to the right of the original input allows to end the copy. In the case of addition, the accepting state of $\mathcal{M}_f$ corresponds to the initial state of $\mathcal{M}_g$. On the other side, the machine $\mathcal{M}_g$ is modified in a way such that it ignores the original input and reads as input the word written in the new tape, from right to left, until it finds the symbol $\check{0}$ corresponding to the blanc symbol ending the original input. The composite machine makes in this way $f(n)+g(n)$ transitions. Both machines, $\mathcal{M}_f$ and $\mathcal{M}_g$, with the required modifications, have been incorporated in a single machine $\mathcal{M}$

The closure of the class of time constructible functions under product can be shown by the technique introduced in the Section 5.2 for the function of expression n^2, noting that $f(n)\times g(n) = f(n)+g(n)+(f(n)-1)\times(g(n)-1)-1$. The machine computes first $f(n)$ in $f(n)$ transitions and then $g(n)$ in $g(n)$ transitions, in separate sets of tapes, counting, as for addition, $f(n)+g(n)$ transitions, concluding the stage with $f(n)$ symbols in one tape and $g(n)$ symbols in another. Then, the machine applies the given technique of Section 5.2 to compute, in unary, $(f(n)-1)\times(g(n)-1)$ in $(f(n)-1)\times(g(n)-1)$ transitions. In the last "dance" the machine halts one transition before the end, by means of suitable marking of the sequence $g(n)$. $\square$

7 From time to space constructible functions

In this section we prove some easy statements that help the reader to understand the relationship between the concepts introduced in this paper.

Definition 10 *A (total) function $f:\mathbb{N}\to\mathbb{N}$ is said to be* computable *by a deterministic Turing machine $\mathcal{M}$ if $\mathcal{M}$ never rejects and writes in the output tape the value $f(x)$ before accepting.*

Proposition 2 *Proper functions are computable.*

Proof: Let $\mathcal{M}$ be a witness of the time constructible function t. From $\mathcal{M}$ we get a Turing machine $\mathcal{M}'$ as follows: we add an output tape to the tapes of $\mathcal{M}$; then, we modify the finite control of $\mathcal{M}$ in such a way that, whenever $\mathcal{M}$ makes a transition, $\mathcal{M}'$ writes 1 in the output tape, moving at the same time the output write-only head one cell to the right. The new machine $\mathcal{M}'$ computes $t(n)$ in unary, where n is the size of the input. [A subcomputation, with one more tape, allows the Turing machine (now another machine $\mathcal{M}''$) to compute $t(n)$ in unary and to make the conversion to binary in the output tape.]

Identical construction can be done for the space: whenever $\mathcal{M}$ visits a new blanc cell, $\mathcal{M}'$ writes 1 in the output tape. $\square$

There are important characterization theorems of proper functions due to Kobayashi (see [9, 2]):

Proposition 3 *Let $t : \mathbb{N} \to \mathbb{N}$ be a (total) function such that there exist $\varepsilon > 0$ and an order $p \in \mathbb{N}$ such that, for $n > p$, $t(n) \geq (1+\varepsilon)\, n$.*[24] *Then function t is a proper function of time if and only if t can be computed in time $O(t)$.*[25]

Proposition 4 *The function s is a proper function of space if and only if s can be computed in space $O(s)$.*

Proposition 5 *If a function is proper of time, then it is proper of space.*

Proof: Let $t : \mathbb{N} \to \mathbb{N}$ be a proper function of time. Then, according to Proposition 3, $t(n)$ can be computed in $O(t(n))$ transitions in which only $O(t(n))$ cells can be visited. Therefore, t is a proper function of space according to Proposition 4. $\square$

A property of time constructible functions to keep in mind is the fact that they have no growing limit within the class of computable functions. Every computable function is bounded by a time constructible function.

Proposition 6 *For every (total) computable function f, there exists a time constructible function g such that g grows faster than f, i.e., for all $n \in \mathbb{N}$, $f(n) < g(n)$.*

Proof: Consider a Turing machine $\mathcal{M}_1$ with two tapes, the input tape and one working tape that computes f in unary. Let $\mathcal{M}_2$ be a Turing machine with four tapes, the input tape, two working tapes, and an output tape, such that, on input x, writes $1^{|x|}$ in the first working tape and then simulates $\mathcal{M}_1$ on input $1^{|x|}$ in the second working tape writing $1^{f(|x|)}$ in the output tape. The machine $\mathcal{M}_2$ consumes the same time for all inputs of size n, a proper time greater than $f(n) + n + 1$, that is the time necessary to read the input and detect its end $(n+1)$, plus the time needed to write the output $(f(n))$. The time of $\mathcal{M}_2$ is therefore constructible and greater than $f(n)$. $\square$

[24] I.e., t grows strictly faster than the identity.

[25] $O(t)$ is the class of functions $t'(n)$ such that, there exists $r > 0$, there exists $p \in \mathbb{N}$, such that, for $n \geq p$, $t'(n) \leq r \times t(n)$.

8 Conclusions

This paper is educational and has as goal to motivate the reader to learn more about Turing machines, computability and complexity.

The Turing machine is not only suitable as a definition of algorithm. It can perform diverse tasks, diverse computations not necessarily associated with the computation of a function or with a decision problem. In this paper we showed that Turing machines can behave like clocks, rulers, and randomizers. All clocks and rulers correspond to computable functions, but not all computable functions, even if they only depend on the size of the input, correspond to clocks or rulers. E.g., to be a clock, a computable function t has to be computable in time $O(t(n))$.

Turing machines with bounded resources specified by clocks and rulers are very important, since they can be used to enumerate the decision problems that have algorithmic solution in constructible times or spaces belonging to some class of bounds. The Turing machines need not be observed for their transitions and non-blanc cells to be counted. Once equipped with clocks or rulers they halt in the precise stipulated time or space, deciding the sets in some a priori given time or space bound.

Although most relevant topic, specification of Turing machines as alarm clocks and rulers are not found in books, namely because they are difficult to specify and, moreover, a general theorem guaranties their existence. In this paper we developed specification techniques to provide those devices in a practical sense and not only in high level abstract sense.

Acknowledgements. We thank to the student Daniel Sousa for having simulated the Turing machines in the Wolfram's MATHEMATICA, and for having done the transition diagram of Figure 11.

References

[1] John E. Hopcroft anf Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[2] José Luis Balcázar, Josep Días, and Joaquim Gabarró. *Structural Complexity I.* Springer-Verlag, 2nd edition, 1988, 1995.

[3] Martin Davis. *The Universal Computer, The Road from Leibniz to Turing.* W. W. Norton and Company, 2000.

[4] K. de Leeuw and N. Shapiro E. F. Moore, C. E. Shannon. Computability by probabilistic machines. In *Automata Studies (C. E. Shannon, ed.)*, Annals of Mathe-

matical Studies, 34, pages 183–198. American Mathematical Society, Rhode Island, 1956.

[5] W. Barkley Fritz. The women of eniac. *IEEE Annals of the History of Computing*, 18(3):13–28, 1996.

[6] J. Gill. *Probabilistic Turing Machines and Complexity of Computations.* PhD thesis, University of California at Berkeley, 1972.

[7] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6:675–695, 1977.

[8] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[9] K. Kobayashi. On proving time constructibility of functions. *Theoretical Computer Science*, 35:215–225, 1985.

[10] P. M. Lewis, R. E. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context sensitive languages. In *Proceedings of the 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 191–202, 1965.

[11] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and Systems Science*, 13:300–317, 1976.

[12] Piergiorgio Odifreddi. *Classical Recursion Theory II.* Studies in Logic and the Foundations of Mathematics. North Holland, 1999.

[13] Michael O. Rabin. Probabilistic algorithms. In John Traub, editor, *Algorithms and Complexity: New Directions and Results*, pages 21–39. Academic Press, London, 1976.

[14] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[15] Michael Sipser. *Introduction to the Theory of Computation.* Thomson, Course Technology, 1996, 2006.

[16] R. E. Stearns, J. Hartmanis, and P. M. Lewis. Hierarchies of memory-limited computations. In *Proceedings of the 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[17] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[18] Alan Turing. On computable numbers. *Proceedings of the London Mathematical Society*, 43:544–546, 1937.

[19] Alan Turing and Jean-Yves Girard. *La Machine de Turing.* Sources du Savoir, Seuil, 1991.