Which arithmetic operations can be performed in constant time in the RAM model with addition?

Étienne Grandjean, Louis Jachiet

Which arithmetic operations can be performed in constant time in the RAM model with addition?

Étienne Grandjean and Louis Jachiet

June 29, 2022

Contents

Abstract: The Church-Turing thesis [86] states that the set of computable functions is the same for any "reasonable" computation model. When looking at complexity classes such as P, EXPTIME, PSPACE, etc., once again, a variation of this thesis [86] states that these classes do not depend on the specific choice of a computational model as long as it is somewhat "reasonable"[1]. In contrast to that, when talking of a function computable in linear time, it becomes crucial to specify the model of computation used and to define what is the time required for each operation in that model.

In the literature of algorithms, the specific model is often not explicit as it is assumed that the model of computation is the RAM (Random Access Machine) model. However, the RAM model itself is ill-founded in the literature, with disparate definitions and no unified results.

The ambition of this paper is to found the RAM model from scratch by exhibiting a RAM model that enjoys interesting algorithmic properties and the robustness of its complexity classes. With that goal in mind, we have tried to make this article as progressive, self-contained, and comprehensive as possible so that it can be read by graduate or even undergraduate students.

The computation model that we define is a RAM whose contents and addresses of registers (including input registers) are $O(N)$, where N is the size (number of registers) of the input, and where the time cost of each instruction is 1 (unit cost criterion).

The key to the foundation of our RAM model will be to prove that even if addition is the only primitive operation, such a RAM can still compute all the arithmetic operations taught at school, subtraction, multiplication and above all division, in constant time after a linear-time preprocessing.

Moreover, while handling only $O(N)$ integers in each register is a strict limit of the model, we will show that our RAM can handle $O(N^d)$ integers, for any fixed d, by storing them on $O(d)$ registers and we will have surprising algorithms that computes many operations acting on these "polynomial" integers – addition, subtraction, multiplication, division, exponential, integer logarithm, integer square root (or cth root, for any integer c), bitwise logical operations, and, more generally, any operation computable in linear time on a cellular automaton – in constant time after a linear-time preprocessing.

The most important consequence of our work is that, on the RAM model, the `LIN` class of linear-time computable problems, or the now well-known $\texttt{CONST-DELAY}_{\texttt{lin}}$ class of enumeration problems computable with constant delay after linear-time preprocessing, are invariant with respect to which subset of these primitive operations the RAM provides as long as it includes addition.

1 Introduction and discussion of the RAM model

While the Turing machine is the "standard" model in computability theory and polynomial time complexity theory, the standard character of this model is not essential due to the Church-Turing thesis and the "complexity-theoretic" Church-Turing thesis [86]: the polynomial time complexity class does not depend on the precise model of computation provided it is "reasonable".

Nevertheless, it is well-known that the Turing machine is too rudimentary to model the precise functioning of the vast majority of concrete algorithms. Instead, the complexity of algorithms is measured up to a constant factor using the RAM model (Random Access Machine) [2, 87].

However, there is a strange paradox about the RAM model, which has persisted since the model was introduced more than half a century ago [76, 32, 49, 22, 2]:

- on the one hand, the RAM is recognized as the appropriate machine model "par excellence" for the implementation of algorithms and the precise analysis of their time and space complexity [22, 2, 84, 66, 38, 39, 40, 71, 72, 42, 41, 35, 23, 87];
- on the other hand, unfortunately, the algorithmic literature does not agree on a standard definition of RAMs and the time costs of their instructions.

A confusing situation. The situation looks like a "Babel tower": to describe and analyze algorithms, while some manuals on algorithms [54, 67, 27, 73] only introduce a pidgin programming language – or a standard one (Pascal, C, Java, Python, etc.) – instead of the RAM model, many other books on algorithms [2, 81, 56, 25, 64, 65, 23, 77] or on computational complexity [50, 84, 66, 35] "define" and discuss "their" RAM models.

[1] We are talking about classical computers here, to the exclusion of quantum computers which can, for example, factorize any integer in polynomial time. The question of knowing if the factorization problem can be solved in polynomial time is a major open problem in the classical framework!

The problem is that the RAM model, whose objective is to model the algorithms in a "standard" way, presents a multitude of variants in the literature. While the representation of memory as a sequence of registers with indirect addressing ability is a common and characteristic feature of all versions of RAM – random access means indirect addressing –, there is no agreement on all other aspects of the definition of this model:

Contents of registers: the data stored in registers are either (possibly negative) integers [22, 2, 66], or natural numbers [84, 35]; in a few rare references [81], real numbers are also allowed.

Input/output: an input (or an output) is either a word (on a finite fixed alphabet) which is read (written) letter by letter [84, 25, 35], or a sequence of integers read (written) integer by integer [22, 2]; in several references [66, 71, 72, 42, 41], an input is a one-dimensional array of integers (a list of input registers) accessed by an index.

Operations: the set of authorized operations consists sometimes of the addition and the subtraction [22, 38, 39, 40, 71, 72, 42, 41], sometimes of the four operations $+, -, \times, /$, see [2]; in other cases, it is a subset of either of these two sets plus possibly some specific operations [81, 84, 66, 35, 77]: division by 2, bitwise Boolean operations, etc.

Sizes of registers: in most references, see e.g. [22, 2, 50, 84, 35, 66], the integer contents of registers are a priori unbounded; in other papers, they must be "polynomial", i.e. at most $O(N^d)$, see [5, 46, 25, 23], for a constant d, or "linear" $O(N)$, see [38, 39, 40, 71, 72, 57, 42, 41], where N is the "size" of the input.

Costs of instructions: the literature presents the following two measures defined in the pioneering paper by Cook and Reckhow [22] and the reference book by Aho, Hopcroft and Ullman [2]:

- **Unit cost criterion:** the execution time of each instruction is uniformly 1 (therefore, it is also called "uniform cost criterion", e.g. in [2]); this makes sense if the integers allowed are bounded, linear, or polynomial in the size of the input;
- **Logarithmic cost criterion:** the execution time of each instruction is the sum of the lengths of the binary representations of the integers handled by the instruction (addresses and contents of registers), or is 1 if the instruction uses no integer.

Note that in practice, the books and papers on algorithms, see e.g. [2, 67, 81, 56, 25, 64, 65, 35, 23, 77], apply almost *exclusively* the unit cost criterion, which is more intuitive and easier to use than the logarithmic one.

The confusing situation of the RAM model that we have just described is all the more damaging since the new algorithmic concepts that have appeared in recent years, see for example [29], require a finer complexity analysis than for traditional algorithms.

Renew the understanding of the RAM model for the analysis of "dynamic" algorithms Since the early 2000s, new concepts have been developed in artificial intelligence (AI), logic, database theory and combinatorics. Basically, they relate to "dynamic" problems whose resolution works in two or more phases.

As a significant example, a *knowledge compilation problem* in AI [19, 26, 58] is a query whose input breaks down into a "fixed part" F and a (much smaller) "variable part" V. An algorithm solving such a query operates in two successive phases: a (long) "preprocessing phase" acting on the "fixed part" F; a "response phase", which reads the "variable part" V of the input and answers (very quickly) the query on the (F, V) input.

A very active field of research for the past fifteen years concerns *enumeration algorithms* in logic, the theory of queries in databases, and combinatorics, see e.g. the surveys [74, 75, 85, 79, 11, 29]. A "fine-grained" and now widespread complexity class (which has some variants) has emerged for enumeration problems, see [6, 30, 8, 9, 24, 7, 51, 52, 31, 83, 3, 14, 53, 20]: the class of problems whose set of solutions can be enumerated with *constant delay* (between two consecutive solutions) after a *linear-time* preprocessing. Each of these papers is based on the RAM model and uses the "minimal" (or quasi-minimal) complexity classes of this model: mainly, linear time, for decision or counting problems, and, for enumeration problems, constant delay after linear (or quasi-linear, or polynomial) time preprocessing.

More recently, several authors in database theory have introduced what they call the "dynamic query evaluation algorithms" [12, 14]. The purpose of these algorithms is to modify the

answer of a query to a database, in *constant* (or *quasi-constant*) *time* when the database is updated by inserting or removing one tuple.

The "fine-grained" complexity classes associated with these new algorithmic concepts (dynamic algorithms) make a thorough understanding of the RAM computation model all the more urgent. The main objective of this paper is to elucidate and justify precisely what "the" RAM model "is" or "should be".

What should the RAM model be? Our requirements are as follows:

- the RAM model we "re-define" must be *simple* and *homogeneous*: in particular, the RAM input and the RAM memory must have the same structure; also, the set of allowed register contents must be equal to the set of allowed addresses;
- the RAM model must "stick" closely to the *functioning of algorithms on discrete structures* (integers, graphs, etc.);
- the "fine-grained" complexity classes [29] defined in the RAM model must be "robust": it must be proved that these complexity classes are largely *independent of the precise definition* of the RAM and, in particular, of the authorized *arithmetic operations*.

Remark 1 (What the RAM model does not model)**.** *In the 80s and 90s, a few authors, see e.g. [1, 68], have introduced several versions of the RAM model in order to model the memory hierarchy of real computers: fast memory (registers) and slower memory with a hierarchy of cache memory, etc.*

Nevertheless, we require that the RAM model be homogeneous, therefore without a memory hierarchy, as almost all the literature on algorithms does. As the reference book of Cormen et al. [23] argues, there are two reasons: first, the analysis of an algorithm is very complicated in memory hierarchy models, which, moreover, are far from being unified in a standard form; the second and main reason is that the analysis of algorithms based on the RAM model without a memory hierarchy "generally gives excellent predictions of the performances obtained on real computers" [23].

Our precise definition of the RAM model will be given in the preliminary section. Let us present below the main characteristics of our model with some elements of justification.

The RAM model we choose

1. **Contents of registers:** each register contains a *natural number*; a RAM with possibly negative integers can be easily simulated by representing an integer x by the pair of natural numbers $(s_x, |x|)$ with $s_x = 0$ if $x \geq 0$ and $s_x = 1$ if $x < 0$; real numbers are excluded for the sake of homogeneity.
2. **Input/output:** in order for the inputs/outputs to be of the same type as the RAM memory, a (standard) input is an *integer*[2] $N > 0$ or a *one-dimensional array of integers* $(N, I[0], \ldots, I[N-1])$, the positive integer N is called the "size" of the input; similarly, an output is an integer or a list of integers.
3. **Operations:** the set of primitive operations consists only of *addition*.
4. **Sizes of registers:** the contents of each register (including each input register $I[j]$) is $O(N)$, where N is the "size" of the input, or equivalently, its length in binary notation is at most $\log_2(N) + O(1)$.
5. **Costs of instructions:** the execution time of each instruction is 1 (*unit cost* criterion).

Our choice of a one-dimensional array of integers $(N, I[0], \ldots, I[N-1])$ of size N as standard input (item 2) is convenient for representing usual data structures such as trees, circuits, (valued) graphs, hypergraphs, etc., while respecting their size. For example, it is easy to represent a tree of n nodes as a standard input of size $\Theta(n)$, or a graph of n vertices and m edges as a standard input[3] of size $\Theta(m+n)$.

[2] From now on, we write "integer" in place of "natural number".

[3] For example, a graph without isolated vertex whose vertices are $1, \ldots, n$ and edges are $(a_1, b_1), \ldots, (a_m, b_m)$ is "naturally represented" by the standard input $\mathcal{I} = (N, I[0], \ldots, I[N-1])$ where $N := 2m + 2$, $I[0] := m$, $I[1] := n$, and finally $I[2j] := a_j$ and $I[2j+1] := b_j$ for $j = 1, \ldots, m$. Note that since the maximum number of vertices of a graph having m edges and no isolated vertices is $2m$ (case when the edges are pairwise disjoint), we have $n \leq 2m = N - 2$, so that each $I[j]$ is less than N.

We do not yet substantiate items (3-5) of our RAM model. Most of this paper is devoted to showing that limiting primitive operations to addition alone, which acts on integers in $O(N)$, is not a limitation of the computing power of RAMs. In fact, we will prove that each of the usual arithmetic operations, subtraction, multiplication, division, and many others, acting on integers in $O(N)$, even on "polynomial" integers in $O(N^d)$ represented in d registers (for a fixed integer d), can be simulated in *constant time* in our RAM model with only the addition operation. This will show the "robustness" and high computing power of our "simple" RAM model, which is the main focus of this paper.

Structure of the paper: We have tried to make this long paper as self-contained, educational and modular as possible. For this, we have given complete, detailed and elementary proofs[4] and structured our many sections so that they can be read, most of the time, independently of each other.

Sections 2 and 3 define our RAM model – with three equivalent instruction sets – and their complexity classes[5]. Those sections also introduce two kinds of RAM simulations with their transitivity properties: first, the *faithful simulation* which is very sensitive to the primitive arithmetic operations of RAMs as states Proposition 1; second, the *faithful simulation after linear-time initialization* (also called *linear-time preprocessing*) which we prove is the right notion to enrich the set of primitive operations of RAMs while preserving their complexity classes, see Corollary 3 (transitivity corollary).

Although Sections 2 and 3 present the precise concepts and their properties to formally ground the results of the paper, the reader can skip them on the first reading of the main results, which are given in Sections 4 to 8, while contenting herself with an intuitive notion of simulation. These sections show how to enrich the RAM model endowed with the only addition operation, by a multitude of operations – the most significant of which are *Euclidean division* (Section 5) and any operations *computable in linear time on cellular automata* (Subsection 8.3 and Appendix) – while preserving its complexity classes.

Third, Section 9 proves several results that argue for the "minimality" of the RAM model with addition.

Section 10 (concluding section) recapitulates and completes our results establishing the invariance/robustness of our "minimal" complexity classes, mainly $\texttt{LIN}$, $\texttt{CONST}_{\texttt{lin}}$ and $\texttt{CONST-DELAY}_{\texttt{lin}}$, according to the set of primitive operations (Corollaries 4 and 5). We also explain why and how our RAM model "faithfully" models the computations of algorithms on combinatorial structures (trees, graphs, etc.) and discuss the more subtle case of string/text algorithms. Finally, Section 10 gives a list of open problems, supplementing those presented throughout the previous sections.

Small digression: a comparison with Gurevich's Abstract State Machines. In addition to the sequential computation models that are the Turing machine, the "Storage Modification Machine" (or "pointer machine", see Section 10) and the RAM that we study here, we must mention the model of "Abstract State Machines" (ASM) introduced by Gurevich in the 80s [43] and studied since in many publications [18, 44, 16, 17, 45]. The ambition of Gurevich and his collaborators is to faithfully model, that is to say step by step, the functioning of all the algorithms (sequential algorithms, parallel algorithms, etc.).

Although the goals of the ASM Model are foundational, like ours, they are much broader in a sense:

- it is a question of refounding the theory of computation, algorithms and programming on a single model, the ASM;
- this unique model includes/simulates in a faithful and natural way the functioning of all algorithms and computation models: Turing machines, RAMs, etc. (see e.g. [16, 17]).

Our objectives in this paper are different:

- we limit our paper to a (systematic) study of the RAM model alone;
- if we are also interested in the fine functioning of algorithms, our main objective is to study the properties of fine-grained complexity classes, in particular the minimal classes;

[4]The only exception is the proof of Theorem 3 which requires implementing elaborate simulations of cellular automata acting in linear time.

[5]Moreover, Subsection 2.5 studies the properties of the RAM model when it is extended by allowing k-dimensional arrays (for a fixed k), that is, with $O(N^k)$ registers available.

- for this, our main tool is not the *faithful simulation*, similar to the *lock-step simulation* of the ASM (see e.g. [28]), because it is very sensitive to the primitive arithmetic operations of RAMs (by our Proposition 1), but it is the *faithful simulation after linear-time initialization*, which is not admitted within the framework of ASM.

2 Instruction sets for the RAM model

In this section, we define our computation model. This is the standard RAM with addition using only integers $O(N)$: here, N is the "size" integer, i.e. the size of (nonnegative number of registers occupied by) the input or, when an arithmetic operation is simulated, the upper bound of the operands. Note that this $O(N)$ limit on the contents of registers also places a limit on the memory since only cells whose addresses are $O(N)$ can be accessed.

To convince the reader of the flexibility of the model, we define a RAM model with a minimal instruction set, called $\mathcal{M}[\texttt{Op}]$, where `Op` denotes a set of allowed operations (e.g. addition, subtraction, etc.). We will also introduce two other minimal instruction sets and show that all these models are in fact equivalent.

For the sake of simplicity, since in what follows we are interested in "minimal" complexity classes, namely *linear time* complexity and *constant time* complexity after *linear time preprocessing*, our RAM model also uses *linear space*. However, it would be possible to define the RAM model for any time $T(N)$ and space $S(N)$ complexity class, provided that the functions $N \mapsto T(N)$ and $N \mapsto S(N)$ meet some "natural" conditions: constructibility (e.g., see [10], section 2.4), etc.

2.1 The RAM model with minimal instruction set

A *RAM* $\mathcal{M}[\texttt{Op}]$ has two special registers A (the *accumulator*) and B (the *buffer*), *work registers* $R[j]$ (the *memory*), *input registers* $I[j]$, for all $j \geq 0$, and an *input size register* N. (For the sake of simplicity, throughout this paper, we identify a RAM with its program.)

Syntax. A *program/RAM* (*AB-program*, *AB-RAM*) of $\mathcal{M}[\texttt{Op}]$ is a sequence $I_0, \ldots, I_{r-1}$ of r *instructions*, also called *AB-instructions*, where each of the I_i has one of the following forms:

Instruction	Meaning	Parameter
`CST` j	$A \leftarrow j$	for some constant integer $j \geq 0$
`Buffer`	$B \leftarrow A$	
`Store`	$R[A] \leftarrow B$	
`Load`	$A \leftarrow R[A]$	
`Jzero` ℓ_0 ℓ_1	`if` $A = 0$ `then goto` ℓ_0 `else goto` ℓ_1	for $0 \leq \ell_0, \ell_1 \leq r$
`op`	see semantics below	for `op` in `Op`
`getN`	$A \leftarrow N$	
`Input`	$A \leftarrow I[A]$	
`Output`	output A	

Table 1: Possible instructions for $\mathcal{M}[\texttt{Op}]$

Semantics. The semantics, called *computation*, of an *AB*-program $I_0, \ldots, I_{r-1}$ on a given input is intuitive. It is formally defined as the (possibly infinite) sequence of the *configurations* (*AB-configurations*) $\mathcal{C}_0, \mathcal{C}_1, \ldots$ where each $\mathcal{C}_i := (A_i, B_i, \lambda_i, D_i)$ consists of the contents A_i, B_i of the two registers A, B, the index λ_i of the current instruction as well as the description D_i of the memory (the contents of the $R[j]$) at instant i.

The *initial configuration* is $A_0 = B_0 = \lambda_0 = 0$ with the array D_0 initialized to 0. Then, from a configuration $\mathcal{C}_i = (A_i, B_i, \lambda_i, D_i)$, the next configuration – if it exists – is deduced from the instruction I_{λ_i} using the semantics given in Table 1. The instruction `Jzero` ℓ_0 ℓ_1 is a conditional jump for which we have $\lambda_{i+1} = \ell_0$ when $A_i = 0$, and $\lambda_{i+1} = \ell_1$ otherwise; for all other instructions, we have $\lambda_{i+1} = \lambda_i + 1$. Note that we allow $\lambda_{i+1} = r$: in that case, we say that $C_{i+1} = (A_{i+1}, B_{i+1}, \lambda_{i+1}, D_{i+1})$ where $A_{i+1} = A_i$, $B_{i+1} = B_i$, $D_{i+1} = D_i$ and $\lambda_{i+1} = r$ is a *final configuration*, it is the last of the computation. For an instruction `op`, its exact semantics depends on the particular operand but for all classical unary or binary operations (addition, multiplication, etc.) we assume that it reads A (and B when binary) and stores its result in A, i.e., its semantics is $A \leftarrow \texttt{op}(A)$ (resp. $A \leftarrow \texttt{op}(A, B)$) when `op` is unary (resp. binary).

The *output* of a program that stops is the list of values that it outputs in the order it produces them. The *running time* of a program is the number of configurations it goes through before ending in its final configuration.

Input In this paper, we will consider two forms of random access inputs:

- a list of integers $(N, I[0], \ldots, I[N-1])$, for $N > 0$ – contained in the registers having the same names – where each $I[j]$, $0 \leq j < N$, is such that $0 \leq I[j] \leq cN$, for some constant integer $c \geq 1$;
- a single integer $N > 0$, contained in the register N.

Note that these types of inputs impose a limit on what we can model. For instance, it does not allow to model streaming algorithms or online algorithms where the algorithm is required to read the data in a particular order whereas, in our model, the input is read in random order. We believe this design is adequate for our purpose (grounding the RAM model) but this limitation is not fundamental and the input/output model could easily be tweaked for many specific needs.

An illustrative program. We now demonstrate how to use $\mathcal{M}[\{+\}]$, the RAM equipped with addition (denoted + or `add`), with the repetitive but simple AB-program given by Table 2. We claim that the AB-program is equivalent to the following "high-level" program (in pseudo-code format), which reads the input size N, assumed to be greater than 1, sets the registers 1 to $N-1$ to the value $I[0]$ and outputs/returns the value $I[1]$.

For this, it uses a "while" construct, which does not exist in our RAM, and we also adopt the convention that the variable i is stored within the $R[0]$ register.

Algorithm 1 Example program in pseudo-code format

```
1: R[N] ← 0                                              ▷ lines 0 - 3
2: i ← 1          ▷ expressed by R[0] ← 1 since i is stored in R[0], see lines 4 - 7
3: while R[N] = 0 do                                     ▷ loop test at lines 8 - 10
4:     R[i] ← I[0]                                       ▷ lines 11 - 16
5:     R[i + 1] ← i + 1                                  ▷ lines 17 - 21
6:     i ← i + 1                                         ▷ lines 22 - 23
7: output I[1]                                           ▷ lines 25 - 27
```

For better readability, each instruction of Algorithm 1 is annotated with the lines it corresponds to in our AB-program; additionally, the AB-instructions of Table 2 are grouped according to each basic instruction of Algorithm 1 that they simulate together (bold lines beginning by $\triangleright$) and the column "Effect" also details what each AB-instruction does.

#	Instruction	Effect
▷	**R[N] ← 0**	
0	`CST` 0	$A \leftarrow 0$
1	`Buffer`	$B \leftarrow A = 0$
2	`getN`	$A \leftarrow N$
3	`Store`	$R[A] \leftarrow B$, which means $R[N] \leftarrow 0$
▷	**i ← 1**	
4	`CST` 1	$A \leftarrow 1$
5	`Buffer`	$B \leftarrow A = 1$
6	`CST` 0	$A \leftarrow 0$
7	`Store`	$R[A] \leftarrow B$, which means $R[0] \leftarrow 1$, or $i \leftarrow 1$
▷	**if R[N] = 0 then enter the loop**	
8	`getN`	$A \leftarrow N$
9	`Load`	$A \leftarrow R[A] = R[N]$
10	`Jzero` 11 25	Enter loop when $A = 0$, i.e. when $R[N] = 0$, else go to loop end
▷	**loop body start**	
▷	**R[i] ← I[0]**	
11	`CST` 0	$A \leftarrow 0$
12	`Input`	$A \leftarrow I[A] = I[0]$
13	`Buffer`	$B \leftarrow A = I[0]$
14	`CST` 0	$A \leftarrow 0$
15	`Load`	$A \leftarrow R[A]$, which means $A \leftarrow R[0] = i$
16	`Store`	$R[A] \leftarrow B$, which means $R[i] \leftarrow I[0]$
▷	**R[i + 1] ← i + 1**	
17	`Buffer`	$B \leftarrow A = i$
18	`CST` 1	$A \leftarrow 1$
19	`add`	$A \leftarrow A + B = 1 + i$
20	`Buffer`	$B \leftarrow A = i + 1$
21	`Store`	$R[A] \leftarrow B$, which means $R[i + 1] = i + 1$
▷	**i ← i + 1**	
22	`CST` 0	$A \leftarrow 0$
23	`Store`	$R[A] \leftarrow B$, which means $i = R[0] \leftarrow i + 1$
▷	**back to the loop test**	
24	`JZero` 10 10	unconditional jump to the conditional "go to"
▷	**loop end**	
▷	**output I[1]**	
25	`CST` 1	$A \leftarrow 1$
26	`Input`	$A \leftarrow I[A] = I[1]$
27	`Output`	`Output` A, which means `Output` $I[1]$

Table 2: Example of AB-program of $\mathcal{M}[\{+\}]$

Minimality of the instruction set. No AB-instruction of Table 1 can be removed without diminishing the expressive power of the model: `CST` j must be used to compute the constant function $N \mapsto j$ when `Op` is the empty set; if we remove `Load` (resp. `Buffer` or `Store`), then the memory registers $R[j]$ cannot be read (resp. modified); if we remove the jump instruction `Jzero` ℓ_0 ℓ_1 then any program performs a constant number of instructions, and finally, it is obvious that the input/output instructions `getN`, `Input`, and `Output` cannot be removed.

Note that the minimality we claim is that no instruction from our RAM can be removed without decreasing expressiveness. This does not mean that another RAM needs at least as many instructions to be as expressive.

2.2 Two more instruction sets

We have just observed that the RAM with the AB-instructions is minimal in the following sense: removing one of the AB-instructions listed in Table 1 strictly weakens the computational power of the model. Recall that our main goal in this paper is to exhibit a "minimal" RAM in the sense of a RAM endowed with the least number of operations on the registers: test at zero and addition, but neither multiplication, nor comparisons $=$, $<$, etc. The second (complementary) objective of this article is to establish the "robustness" of the RAM model: its computational power (its complexity classes) is (are) invariant for many changes of its definition. To this end, this subsection will present two alternative instruction sets and Subsection 2.3 will show that all programs can be transformed from one instruction set to another with time and space bounds multiplied by a constant.

RAM with assembly-like instruction set.

A RAM with `Op` operations and *assembly-like instruction set*, called R-RAM, has *work registers* $R[j]$, *input registers* $I[j]$, for all $j \geq 0$, and an *input size register* N. An *R-program* is a sequence $I_0, \ldots, I_{r-1}$ of labeled instructions, called *R-instructions*, of the following forms with given intuitive meaning, where i, j are explicit integers, ℓ_0, ℓ_1 are valid indexes of instructions and `op` is an operator from `Op` of arity k:

Instruction	Meaning
`CST` i j	$R[i] \leftarrow j$
`Move` i j	$R[i] \leftarrow R[j]$
`Store` i j	$R[R[i]] \leftarrow R[j]$
`Load` i j	$R[i] \leftarrow R[R[j]]$
`Jzero` i ℓ_0 ℓ_1	`if` $R[i] = 0$ `then goto` ℓ_0 `else goto` ℓ_1
`op`	$R[0] \leftarrow$ `op`$(R[0], \ldots, R[k-1])$
`getN` i	$R[i] \leftarrow N$
`Input` i j	$R[i] \leftarrow I[R[j]]$
`Output` i	output $R[i]$

Table 3: Set of R-instructions for $\mathcal{M}$[`Op`]

The semantics given in Table 3 is very similar to that of our original set of instructions. Formally, the *computation* of an *R-program* $I_0, \ldots, I_{r-1}$ on a given input is defined as the sequence of the *configurations* (*R-configurations*) $\mathcal{C}_0, \mathcal{C}_1, \ldots$ where each $\mathcal{C}_i := (\lambda_i, D_i)$ consists of the index λ_i of the current instruction and the description D_i of the RAM memory (contents of the $R[j]$) at instant i; in particular, $\mathcal{C}_0 = (\lambda_0, D_0)$ is the initial configuration with $\lambda_0 = 0$ and the RAM memory D_0 initialized to 0.

Noticing that we can store the value of A into $R[0]$ and the value of B into $R[1]$, it is relatively clear that any AB-program can be simulated with an R-program. The third definition of RAM data structures and instructions that follows is closer to those of a programming language.

Multi-memory RAM.

A *multi-memory RAM* M with `Op` operations uses a fixed number t ($t \geq 1$) of one-dimensional *arrays of integers* $T_0[0, \ldots], \ldots, T_{t-1}[0, \ldots]$, a fixed number of *integer variables* $C_0, \ldots, C_{v-1}$, an *input array* $I[0, \ldots]$ of integers and the *input size register* N. The *program* of M uses `Op`-*expressions* or, for short, *expressions*, which are defined recursively as follows.

Definition 1 (`Op`-*expressions*).

- *Any fixed integer i, the size integer N, and any integer variable C_j are* `Op`*-expressions.*
- *If α is an* `Op`*-expression, then an array element $T_j[\alpha]$ or $I[\alpha]$ is an* `Op`*-expression.*
- *If $\alpha_1, \ldots, \alpha_k$ are* `Op`*-expressions and* `op` *is a k-ary operation in* `Op`*, then* `op`$(\alpha_1, \ldots, \alpha_k)$ *is an* `Op`*-expression.*

Example: $T_1[T_3[2] + N] \times I[C_1]$ is a $\{+, \times\}$-expression.

The *program* of a multi-memory RAM with `Op` operations is a sequence $I_0, \ldots, I_{r-1}$ of instructions, called *array-instructions*, of the following forms with intuitive meaning, where α and β are `Op`-expressions and ℓ_0, ℓ_1 are valid indexes of instructions.

Instruction	Meaning
$T_j[\alpha] \leftarrow \beta$	
$C_j \leftarrow \alpha$	
`Jzero` $\alpha\ \ell_0\ \ell_1$	`if` $\alpha = 0$ `then goto` ℓ_0 `else goto` ℓ_1
`Output` α	

Table 4: Set of array-instructions for $\mathcal{M}$[Op]

The *computation* of a program with array-instructions (called *array-program*) $I_0, \ldots, I_{r-1}$ on a given input is defined as the sequence of the *configurations* (*array-configurations*) $\mathcal{C}_0, \mathcal{C}_1, \ldots$ where the configuration $\mathcal{C}_i := (\lambda_i, D_i)$ at instant i consists of the index λ_i of the current instruction ($\lambda_0 = 0$, initially) and the description D_i of the memory (the values of the integer variables and arrays, all initialized to 0).

2.3 Equivalence of our three instruction sets

The following lemma states that our three instructions sets (our three classes of RAMs) are "equivalent", within the meaning of Definition 3 of Subsection 3.1, when the addition is available. (Note that, for pedagogical reasons, the notions of "simulation" and "equivalence" involved in the statement of Lemma 1 are first presented here in an informal/intuitive way. Section 3 will give formal definitions and complete evidence.)

Lemma 1. *1. Any R-RAM can be simulated by an AB-RAM.*

2. Any AB-RAM can be simulated by a multi-memory RAM.

3. If addition belongs to the set Op *of allowed operations then any multi-memory RAM can be simulated by an R-RAM.*

Therefore, by transitivity, the three classes of AB-RAMs, R-RAMs, and multi-memory RAMs are equivalent when `add` $\in$ Op.

Proof of item 1. As representative examples, Tables 5, 6, 7, 8 and 9 give the simulation (by a sequence of *AB*-instructions) of each of the *R*-instructions `op`, for a binary `op`, `Output` i, `Store` $i\ j$, `Input` $i\ j$ and `Jzero` $i\ \ell_0\ \ell_1$.

Instruction	Effect
`CST` 1	$A \leftarrow 1$
`Load`	$A \leftarrow R[1]$
`Buffer`	$B \leftarrow R[1]$
`CST` 0	$A \leftarrow 0$
`Load`	$A \leftarrow R[0]$
`op`	$A \leftarrow \mathtt{op}(R[0], R[1])$
`Buffer`	$B \leftarrow \mathtt{op}(R[0], R[1])$
`CST` 0	$A \leftarrow 0$
`Store`	$R[0] \leftarrow \mathtt{op}(R[0], R[1])$

Table 5: Emulation of `op`

Instruction	Effect
`CST` i	$A \leftarrow i$
`Load`	$A \leftarrow R[i]$
`Output`	output $R[i]$

Table 6: Emulation of `Output` i

Instruction	Effect
`CST` j	$A \leftarrow j$
`Load`	$A \leftarrow R[j]$
`Buffer`	$B \leftarrow R[j]$
`CST` i	$A \leftarrow i$
`Load`	$A \leftarrow R[i]$
`Store`	$R[R[i]] \leftarrow R[j]$

Table 7: Emulation of `Store` $i\ j$

Instruction	Effect
`CST` j	$A \leftarrow j$
`Load`	$A \leftarrow R[j]$
`Input`	$A \leftarrow I[R[j]]$
`Buffer`	$B \leftarrow I[R[j]]$
`CST` i	$A \leftarrow i$
`Store`	$R[i] \leftarrow I[R[j]]$

Table 8: Emulation of `Input` $i\ j$

Instruction	Effect
`CST` i	$A \leftarrow i$
`Load`	$A \leftarrow R[i]$
`Jzero` $\ell_0\ \ell_1$	`if` $R[i] = 0$ `then goto` ℓ_0 `else goto` ℓ_1

Table 9: Emulation of `Jzero` $i\ \ell_0\ \ell_1$

□

Proof of item 2. This item is the easiest to prove: we can consider the special registers A, B as integer variables and the sequence of work registers $R[0], R[1], \ldots$ as the array of integers $R[0\ldots]$ and then any AB-program becomes a multi-memory program. □

Proof of item 3. Similarly to a compiler that works in two "passes" with an intermediate language, our simulation of array-instructions with R-instructions is done in two steps with an intermediate instruction set that we call "elementary array-instructions" depicted in Table 10 where $h, i, j, j_1, \ldots, j_k$ are explicit integers, ℓ_0, ℓ_1 are valid indexes of instructions and `op` is a k-ary operator from `Op`.

Instruction	**Meaning**
$C_i \leftarrow j$	
$C_i \leftarrow N$	
$C_i \leftarrow \texttt{op}(C_{j_1}, \ldots, C_{j_k})$	
$C_i \leftarrow I[C_j]$	
$C_h \leftarrow T_j[C_i]$	
$T_j[C_i] \leftarrow C_h$	
`Jzero` C_i ℓ_0 ℓ_1	`if` $C_i = 0$ `then goto` ℓ_0 `else goto` ℓ_1
`Output` C_i	

Table 10: Elementary array-instructions

From array-instructions to elementary array-instructions. To achieve this first compilation step, the idea is to create a new integer variable C_j for each sub-expression α' appearing in any of the `Op`-expression α that occurs in the initial multi-memory RAM program. So, any array-instruction will be simulated (replaced) by a sequence of elementary array-instructions.

As a representative example, the array-instruction $T_2[C_1 * T_1[4]] \leftarrow T_1[T_3[2] + N]$, which is of the form $T_j[\alpha] \leftarrow \beta$, is simulated by the following sequence of elementary array-instructions:

- Storing $C_1 * T_1[4]$ into D_0: $D_1 \leftarrow 4$; $D_2 \leftarrow T_1[D_1]$; $D_0 \leftarrow C_1 * D_2$;
- Storing $T_1[T_3[2] + N]$ into D_3:
  $D_4 \leftarrow 2$; $D_5 \leftarrow T_3[D_4]$; $D_6 \leftarrow N$; $D_7 \leftarrow D_5 + D_6$; $D_3 \leftarrow T_1[D_7]$;
- Storing $T_1[T_3[2] + N]$ into $T_2[C_1 * T_1[4]]$: $T_2[D_0] \leftarrow D_3$.

Note that the resulting instructions have the required form: to see it, rename each variable D_j to $C_{j'}$ for some j'. This process can be adapted for any array-instruction using `Op`-expressions α and β (see Table 4) by adding a number of instructions and variables roughly equal to the sum of the sizes of the expressions α and β.

From elementary array-instructions to R-instructions Let us suppose that we want to translate a program with elementary array-instructions into an R-program. Let $C_0, \ldots, C_{v-1}$ be the list of variables of the array-program and $T_0, \ldots, T_{t-1}$ be its list of arrays, and finally let k be an integer greater than the maximal arity of an `op` (which is at least 2 because we have `add`).

The idea for building an R-configuration from an array-configuration is to segment the memory in the following way: the k first registers $R[0], \ldots, R[k-1]$ will be used for intermediate computations, the v next registers $R[k], \ldots, R[k+v-1]$ will store the contents of the variables, and finally the rest of the memory will be dedicated to store the contents of the arrays. More precisely, the register $R[k+i]$ for $i < v$ will store the content of the variable C_i and $T_j[i]$ will be stored in the register $R[k+v+t*i+j]$. Clearly, there is no interference between the roles of registers $R[y]$.

Since each integer variable C_i is translated as $R[j]$ where j is the explicit value of $k+i$, each elementary array-instruction turns trivially into an R-instruction, with the exception of the three instructions $C_h \leftarrow T_j[C_i]$, $T_j[C_i] \leftarrow C_h$ and $C_i \leftarrow \texttt{op}(C_{j_1}, \ldots, C_{j_k})$.

The first two of these instructions can be translated as $R[k+h] \leftarrow R[k+v+t*R[k+i]+j]$ and $R[k+v+t*R[k+i]+j] \leftarrow R[k+h]$, respectively, which must be transformed. The two transformations are similar, so we will detail the first one. By using the buffers $R[0], R[1]$ and the addition operation (but not the multiplication!) we can transform the assignment $R[k+h] \leftarrow R[k+v+t*R[k+i]+j]$ into an equivalent sequence of R-instructions as shown in the program of Table 11 with $t+3$ instructions. (Note that h, i, j, k, v and t are all explicit integers and therefore $t+3$, $k+i$, $k+v+j$, $k+h$, etc., are also explicit integers whose value can be computed before the RAM is launched.)

#	Instruction	Effect
$\ell+0$	`Move` 1 $(k+i)$	$R[1] \leftarrow R[k+i]$
$\ell+1$	`CST` 0 $(k+v+j)$	$R[0] \leftarrow k+v+j$
$\ell+2$	`add`	$R[0] \leftarrow R[0]+R[1]$, meaning $R[0] \leftarrow k+v+j+R[k+i]$
$\ell+3$	`add`	$R[0] \leftarrow R[0]+R[1]$, meaning $R[0] \leftarrow k+v+j+2*R[k+i]$
$\dots$	$\dots$	$\dots$
$\ell+t+1$	`add`	$R[0] \leftarrow R[0]+R[1]$, meaning $R[0] \leftarrow k+v+j+t*R[k+i]$
$\ell+t+2$	`Load` $(k+h)$ 0	$R[k+h] \leftarrow R[R[0]]$, or $R[k+h] \leftarrow R[k+v+j+t*R[k+i]]$

Table 11: Emulation of $C_h \leftarrow T_j[C_i]$

For the `op` operation $C_i \leftarrow \texttt{op}(C_{j_1}, \dots, C_{j_k})$, the translation is simpler: we first store each C_{j_l} into $R[l-1]$ with `Move` $(l-1)$ $(k+j_l)$, then we apply the R-instruction `op`, which computes $R[0] \leftarrow \texttt{op}(R[0], \dots, R[k-1])$; finally, we store back the result into C_i with `Move` $(k+i)$ 0.

This completes the proof of item 3. Lemma 1 is proved. □

2.4 Richer instruction set

At the moment we saw three instructions sets and showed that they are equivalent. Since they are all equivalent, in the following sections, our procedures will be written in the third instruction set, which is the easiest to work with.

While this multi-memory model is the closest to a standard language among our three instruction sets, it still lacks a lot of the constructs available in most programming languages such as loops, functions, etc. This section is dedicated to show that we can easily compile a language with those features to our assembly.

Most of the techniques we use here to enrich the instruction set are very well known and very well studied for compilers. While they are interesting in themselves there are not the novelty of the paper, which is why we will only give the intuition of what is happening and point the reader to the existing literature.

From indexed programs to labeled programs. At this point in the paper, a program is still viewed as a sequence $I_0, \dots, I_{r-1}$ of instructions where the index of each instruction is important as a `Jzero` instruction will jump to a given index. We can replace this scheme with a label scheme and a program becomes a set of labeled instructions $I_a, I_b, \dots$ where each instruction has one or several labels and each instruction e which is not `Jzero` has a next label $next(e)$ (determining what instruction should be executed after). We also have a special label `end` for the label $next(e)$ where e is the final instruction and a special label `begin` to determine the first instruction. Such a labeled program can easily be translated into an indexed program.

Composing programs. Once the relabeling scheme is completed, it is easy to talk about the composition of two programs P_1 and P_2: we relabel the labels of P_1 and P_2 to make sure that they are unique in both, with the `end` of P_1 being the `begin` instruction of P_2; then we replace the output instructions of P_1 by write instructions in an array O and the input instructions of P_2 are replaced by read instructions of the said array O.

Equality conditions. Using labeled programs with array-instructions, we can allow the equality condition $\alpha = \beta$ for all expressions α, β, i.e. allow a new instruction `Jequal` α β ℓ_0 ℓ_1 (extending the `Jzero` instruction), which means "`if` $\alpha = \beta$ `then goto` ℓ_0 `else goto` ℓ_1". Actually, we claim that the instruction `Jequal` α β ℓ_0 ℓ_1 is equivalent to the following sequence of array-instructions using the new array `Eq` with suggestive name:

$$\texttt{Eq}[\alpha] \leftarrow 1 \ ; \ \texttt{Eq}[\beta] \leftarrow 0 \ ; \ \texttt{Jzero Eq}[\alpha] \ \ell_0 \ \ell_1$$

Proof. Clearly, if $\alpha = \beta$ then the assignment $\texttt{Eq}[\beta] \leftarrow 0$ gives $\texttt{Eq}[\alpha] = \texttt{Eq}[\beta] = 0$; otherwise, we still have $\texttt{Eq}[\alpha] = 1 \neq 0$. In other words, we have the equivalence $\alpha = \beta \iff \texttt{Eq}[\alpha] = 0$. □

Advanced Boolean manipulation. One of the features of a programming language is to be able to perform complex conditions such as $(a = b) \vee (b \neq c \wedge b \neq d)$. At the moment, the only condition allowed is the equality comparison but it is easy to convert such a comparison $\alpha = \beta$ into an integer c which is 0 when $\alpha \neq \beta$ and 1 when $\alpha = \beta$. This also allows manipulation of Boolean logic: the negation of cond_1 and the conjunction $\text{cond}_1 \wedge \text{cond}_2$ can be transformed into $c_1 = 0$ and $c_1 + c_2 = 2$, respectively, where c_i is the integer associated with cond_i. This allows us

to manipulate any Boolean combinations of conditions (the basic conditions only allow equalities for the moment, inequalities $<$ will be introduced in Subsection 4.2).

Basic constructions (if-then-else, loops, etc.). Our procedures and programs will freely use the basic constructs and syntax of a programming language. Once you have moved from an indexed program to a labeled program, it is easy to translate an "`if` $\alpha = \beta$ `then` body_0 `else` body_1" into a `Jequal` $\alpha\ \beta\ \ell_0\ \ell_1$ where ℓ_i points to the first instruction of body_i and body_0 finishes with a jump to the end of body_1.

The reader can also easily check that a loop "`while` $\alpha = \beta$ `do` body" can be translated with one conditional test at the beginning, then the translation of the body, then one unconditional test to go back to the conditional test. In the same manner, we can have the for-loop construct. We will use the following format "`for` *var* `from` α `to` β `do` body `done`" meaning that *var* will range from α up to β. At this stage, this construct is only defined for $\alpha \leq \beta$.

Dynamic arrays of bounded size. In our simulation of array-instructions with R-instructions we use a constant number of arrays with no bound on the size. As in most actual programming languages, it is often useful to be able to allocate arrays dynamically (i.e. during the execution).

In our RAM model, we can do this with the addition operation using the following memory allocator scheme. The idea to allocate dynamic arrays uses one static array called `DATA` storing the data of all the dynamic arrays and one variable `nbCellsUsed` storing the number of cells used for dynamic arrays in `DATA`. To allocate a dynamic array of size k we increase `nbCellsUsed` by k and return the value $p =$ `nbCellsUsed`$-k$. The value p identifies the array and is a sort of "pointer" towards the memory of the newly allocated array, very similar to what happens in languages such as C. Each read or write access to i-th cell of the array T is then replaced with an access to `DATA`$[T + i]$.

Note that this creates two kinds of arrays, the static ones (available in array instructions) and the dynamic ones but for the sake of simplicity we will denote by $T[i]$ the i-th cell of T no matter whether T is a value corresponding to a dynamically allocated array or a static one.

Two-dimensional arrays The allocation scheme presented above is very useful to allocate many arrays (i.e. more than a constant number) as long as the total allocated size is $O(N)$. For instance, using this we can allocate a dynamic array T of size R and store within each $T[i]$ an array of size C. This creates a two-dimensional array of dimension $R \times C$. Note that this is only allowed if the total allocated memory is $O(N)$ and thus we need to have $R \times C = O(N)$.

Note that we can also generalize this method to any number of dimensions as long as the total number of cells is $O(N)$.

Functions. Most programming languages are equipped with functions. It is well known that one can "de-functionalize" a program with a stack. Here, we will notice that the addition is all we need on a RAM to make this "de-functionalization" work.

For this, each active function call will have its memory stored in an array S (the stack) and we will have a variable F (the frame pointer) pointing to the point where is the memory for the current function call. The memory used by one function call will span $v + 2$ cells, where v is the maximal number of variables used in a function: the first cell (stored in $S[F]$) designates the point in the program where the function call originated from (so we can return to it), the cell $S[F + 1 + i]$ will store the memory of the i-th variable. When calling a function, we need to compute $F = F + v + 1$, then we set $S[F]$ to mark the current function call as the caller. Finally, we jump. When returning from a function call we determine the caller from $S[F]$ and then we set $F = F - v - 1$.

Since the memory of the function called is not overwritten until we make a new function call, we can simulate a "return" instruction by reading the value of the variables of the function we have just called. Note that we will often handle integers spanning a constant number of registers (that we later call "polynomial" integers) and will freely use **return** `x` for x an array of constant size.

2.5 RAM models with a different space usage

We have seen three different RAM models all equally expressive when the addition is present (equally expressive in the sense that they define the same complexity classes). These three models have three different instructions sets that all manipulate a memory that is zero-initialized, which leads to our first question:

Does the expressiveness of the model change if we suppose that the memory is not zero-initialized?

A second property of our model is that it depends on a parameter N and we suppose that all registers can only contain a value $O(N)$ which, in turn, limits the available memory to $O(N)$. This leads to our second natural question:

Can we change our model to take more memory into account? And if so, how does that impact expressiveness?

The initialization problem. As in the original RAM model [22, 2], we have assumed that all registers are initialized to zero before the start of the computation. However, as is the rule in the Python language, it is a principle of good programming that any memory variable (integer, array, etc.) be explicitly initialized as soon as it is declared/used.

This requirement can be satisfied for any program of the "linear time" complexity classes that interest us (see Subsection 3.2) provided that the addition is available. Indeed, such a program uses only a linear number of memory registers $R[0], R[1], \ldots, R[c \times N]$, for a constant integer c. Therefore, the program can be transformed into an equivalent program of the same complexity by adding at the start of the program the explicit initialization. For this, we can use a program very similar to the illustrative program given in Table 2 but where, instead of setting and reading $R[N]$, we set and read $R[N \times c]$ where $N \times c$ is N added c times to 0.

If we suppose that when reading uninitialized memory it might return any value, another way to deal with the initialization problem is the "lazy array evaluation technique" described in Moret and Shapiro's book [63]. interest of this technique is that it applies to general (multidimensional) arrays, we present this technique below just after presenting extended with more memory, $O(N^k)$, i.e. allowing k-dimensional arrays $N \times \cdots \times N$, for some fixed k.

RAM model with more memory. All the RAM models presented above suffer from the same limit on space usage. Since the integers stored in the registers are $O(N)$ and all native arrays are one-dimensional, it means that the memory accessed by any program is always $O(N)$.

One way to overcome this limit is to consider that we can have k-dimensional arrays available, for some fixed integer k, and in this case a k-dimensional array would allow $O(N^k)$ registers. Several recent papers about queries in logic and databases, see e.g. [31, 75, 12, 13], present algorithms inside such an extended RAM model. Significantly, Berkholz, Keppeler and Schweikardt write in [12]: " A further assumption that is unproblematic within the RAM-model, but unrealistic for real-world computers, is that for every fixed dimension $d \in \mathbb{N}_{>1}$ we have available an unbounded number of d-ary arrays $\mathbf{A}$ such that for given $(n_1, \ldots, n_d) \in \mathbb{N}^d$ the entry $\mathbf{A}[n_1, \ldots, n_d]$ at position $(n_1, \ldots, n_d)$ can be accessed in constant time."

We know that a machine with a memory of size $O(N^k)$ cannot be emulated (in general) by a machine with only $O(N)$ memory for $k > 1$. This means that the models with or without k-dimensional arrays are incomparable and, more generally, for all k the models differ. If we really want to emulate "true" k-dimensional arrays we need to set the parameter N' to N^k but this does not provide much hindsight in the problem at hands. An interesting case is when the program has access to k-dimensional arrays but actually only uses only $O(N)$ memory registers. Consider first the problem of initializing k-dimensional arrays.

The "lazy array evaluation technique" (according to Moret and Shapiro's book [63], Subsection 4.1, Program 4.2): Let P be a program using k-dimensional arrays *assumed to be initialized to zero for free*. We want to construct a program P' simulating P with k-dimensional arrays, for the same k, which are *not initialized*. For each k-dimensional array $\mathtt{A}[1..N]\ldots[1..N]$ of P, we create a variable `count` counting the number of initialized cells $(i_1, \ldots, i_k)$ of `A`, a k-dimensional array $\mathtt{rankInit}[1..N]\ldots[1..N]$ and its "inverse" one-dimensional array $\mathtt{rankInit}^{-1}$, called *lazy arrays*, such that $\mathtt{rankInit}[i_1]\ldots[i_k] = y$ and, conversely, $\mathtt{rankInit}^{-1}[y] = (i_1, \ldots, i_k)$ if $\mathtt{A}[i_1]\ldots[i_k]$ is the yth initialized cell of `A`. The important thing is to maintain the following equivalence, called equivalence (E), for each $(i_1, \ldots, i_k) \in [1, N]^k$:

$$\begin{array}{c}\text{the cell } \mathtt{A}[i_1]\ldots[i_k] \text{ has been given a value, i.e. has been initialized, } \textit{iff} \\ \mathtt{rankInit}^{-1}[\mathtt{rankInit}[i_1]\ldots[i_k]] = (i_1, \ldots, i_k), \text{ for } \mathtt{rankInit}[i_1]\ldots[i_k] \in [1, \mathtt{count}].\end{array}$$

This gives the following procedures (freely inspired by Program 4.2 in [63]), which we present, to simplify the notation, with $k = 2$. (Note that to simplify also, we do not mention `A` in the identifiers of the variable `count`, arrays `rankInit` and $\mathtt{rankInit}^{-1}$, and procedures INITCOUNT, INITCELL, STORE and FETCH corresponding to the array `A`.)

Algorithm 2 Lazy evaluation procedures for a 2-dimensional array `A`

```
1: procedure initCount()
2:     count ← 0

1: procedure initCell(i, j)
2:     ▷ answers the question: "Has cell A[i][j] been initialized?" ◁
3:     return (1 ≤ rankInit[i][j] ≤ count) and (rankInit^{-1}[rankInit[i][j]] = (i, j))

1: procedure store(i, j, x)
2:     ▷ runs A[i][j] ← x and validates that cell (i, j) is now initialized ◁
3:     A[i][j] ← x
4:     if not initCell(i, j) then
5:         count ← count + 1
6:         rankInit[i][j] ← count
7:         rankInit^{-1}[count] ← (i, j)        ▷ now, initCell(i, j) = true

1: procedure fetch(i, j)
2:     ▷ returns A[i][j] if cell (i, j) has been initialized, and 0 otherwise ◁
3:     if initCell(i, j) then
4:         return A[i][j]
5:     else
6:         return 0
```

Justification: Lines 3-6 of the procedure store, with the procedures initCell and initCount, justify that the function $(i,j) \mapsto \texttt{rankInit}[\texttt{i}][\texttt{j}]$ is always a bijection from the set of "initialized" pairs $(i,j) \in [1,N]^2$, i.e. for which $\textsc{initCell}(\texttt{i},\texttt{j})$ is true, to the integer interval $[1, \texttt{count}]$, and $y \mapsto \texttt{rankInit}^{-1}[y]$, for $y \in [1, \texttt{count}]$, is its inverse bijection. The fact that the set of images $\texttt{rankInit}^{-1}([1, \texttt{count}])$ is equal to the set of "initialized" pairs $(i,j) \in [1,N]^2$ justifies that the equivalence (E) seen above is true for all $(i,j) \in [1,N]^2$.

As a consequence, in the general case, a program P using k-dimensional arrays assumed to be initialized to zero for free, is simulated by a program P' constructed from P as follows:

- put the procedure call initCount() (i.e. the assignment $\texttt{count} \leftarrow 0$), for each k-dimensional array `A` of P, at the beginning of the program P';
- replace each *assignment* of the form $\texttt{A}[t_1]\dots[t_k] \leftarrow t$ (resp. each *argument* of the form $\texttt{A}[t_1]\dots[t_k]$) in P by the call instruction $\textsc{store}(t_1,\dots,t_k,t)$ (resp. by the term $\textsc{fetch}(t_1,\dots,t_k)$) corresponding to `A`.

Now, it is simple to verify that if the RAM program P using k-dimensional arrays $N \times \cdots \times N$, assumed to be initialized to zero for free, runs in time $O(T)$, then its simulating program P' using only *uninitialized* k-dimensional (and one-dimensional) arrays $N \times \cdots \times N$, including the additional *lazy arrays*, runs also in time $O(T)$. In conclusion, the lazy array evaluation technique (to simulate multidimensional arrays initialized to zero for free) does not lead to a real complexity overhead: time and space are only multiplied by a (small) constant.

Multidimensional arrays are equivalent to 2-dimensional arrays. We now give a (very) partial answer to our second question: how does the expressiveness of the RAM model depend on the dimensions of the allowed arrays? The following theorem states that the RAM model with arrays of arbitrary dimensions is equivalent to the RAM model using only 2-dimensional arrays, and even, more strongly, using arrays of dimensions $N \times N^{\epsilon}$, for any $\epsilon > 0$.

Theorem 1. *For any integers $k > 1$ and $d \geq 1$, a RAM model equipped with the basic arithmetic operations and k-dimensional arrays but using only $O(N)$ registers and time $O(t(N))$, can be simulated in time $O(t(N))$ and using $O(N)$ registers by a RAM model equipped with the same set of operations but using only 1-dimensional arrays and one 2-dimensional array $N \times \lfloor N^{1/d} \rfloor$.*

Proof. For the sake of simplicity, we will suppose that we want to emulate one k-dimensional array $N \times \cdots \times N$. Indeed, if we have multiple k-dimensional arrays $\texttt{T}_0, \dots, \texttt{T}_{\ell-1}$, each of dimensions $cN \times \cdots \times cN$, for a fixed integer c, we can replace each k-dimensional key $v_1, \dots, v_k$ ($v_j < cN$) of $\texttt{T}_i$ with one $(2k+1)$-dimensional key $i, u_1, u_2, \dots, u_{2k-1}, u_{2k}$, with $u_{2j-1} \coloneqq v_j \texttt{ div } c$ and

$u_{2j} \coloneqq v_j \bmod c$: the arrays $\mathtt{T}_0, \ldots, \mathtt{T}_{\ell-1}$ are emulated by one $(2k+1)$-dimensional array $\mathtt{T}$ of dimensions $N \times \cdots \times N$ such that $\mathtt{T}[i][u_1][u_2]\ldots[u_{2k-1}][u_{2k}] = \mathtt{T}_i[v_1]\ldots[v_k]$, assuming $N \geq \max(c, \ell)$; thus, the dimension k is increased to $2k+1$.

Now, the general idea of the emulation of a k-dimensional $N \times \cdots \times N$ array $\mathtt{T}$ with a 2-dimensional array $cN \times \lfloor N^{1/d} \rfloor$ (for some fixed integer c) called `node` and a one-dimensional called `A` consists in representing all the data in all the k-dimensional keys as paths in a tree where the leaves store the values of the original array $\mathtt{T}$. Each node in the tree will have an arity bounded by $\lfloor N^{1/d} \rfloor$ and all leaves will be at depth $\leq 2dk$, which is possible by the inequality[6] $\lfloor N^{1/d} \rfloor^{2dk} \geq N^k$. Since we only use $O(N)$ registers in $\mathtt{T}$, only $O(N)$ leaves and thus $O(N)$ nodes will be present in that tree.

Now, to find the leaf associated with a key $(v_1, \ldots, v_k) \in [0, N[^k$, we transform the sequence $(v_1, \ldots, v_k)$, seen as the sequence of digits of an integer in base N, into the sequence $(u_1, \ldots, u_{k'}) \in [0, \mathtt{B}[^{k'}$ representing the same integer in base $\mathtt{B} \coloneqq \lfloor N^{1/d} \rfloor$. (Note that $k' < 2dk$.) Now, we can access the value stored in the leaf corresponding to $(u_1, \ldots, u_{k'})$ by starting from the root $n_0 = 0$ and repetitively using $n_i = \mathtt{node}[n_{i-1}][u_i]$, for $1 \leq i \leq k'$. Thus, the time of this access is $O(k') = O(kd)$: it is a constant time. When the value/node returned by $\mathtt{node}[n_i][u_i]$ is undefined, we create a new node in the tree and set $\mathtt{node}[n_i][u_i]$ to correspond to this node. Finally, the value "stored" at the leaf $n_{k'}$ is $\mathtt{A}[n_{k'}] = \mathtt{T}[v_1]\ldots[v_k]$.

Note that the values (nodes) n_i are less than cN, for some fixed integer c, and therefore the arrays used by the simulation are $\mathtt{node}[0..cN-1][0..\mathtt{B}-1]$ and $\mathtt{A}[0..cN-1]$. Then, replace (encode) the $cN \times \mathtt{B}$ array `node` by the $N \times \lfloor N^{2/d} \rfloor$ array $\mathtt{E}$ defined by $\mathtt{E}[x \mathbin{\mathtt{div}} c][c \times y + x \bmod c] \coloneqq \mathtt{node}[x][y]$, for $x < cN$ and $y < \mathtt{B}$: this is possible because we have $\lfloor N^{1/d} \rfloor \geq c$, for $N \geq c^d$, which implies $c \times \mathtt{B} = c \times \lfloor N^{1/d} \rfloor \leq \lfloor N^{1/d} \rfloor^2 \leq \lfloor N^{2/d} \rfloor$. The theorem is obtained by replacing d by $2d$: this means that we set $\mathtt{B} \coloneqq \lfloor N^{1/(2d)} \rfloor$ instead of $\mathtt{B} \coloneqq \lfloor N^{1/d} \rfloor$. □

Remark 2 (Essential point)**.** *Note that it is assumed in the proof of Theorem 1 that the RAM has access to the basic arithmetic operations (essentially, computation of $\lfloor N^{1/d} \rfloor$, division by a constant c and base change) but since the RAM model with k-dimensional arrays is more powerful than the model we study in the rest of this paper (which allows to execute those operations in constant time with a linear-time preprocessing, as we will establish), we know that* a fortiori *those operations (for example, the conversion from base N to base $\mathtt{B}$, used in the above simulation) can be computed in constant time with a linear-time preprocessing in the more liberal model.*

From Theorem 1, we can deduce that two models using $O(N)$ registers with k-dimensional or k'-dimensional arrays are equivalent (for $k > k' > 1$) but remains the question of whether the RAM model with k-dimensional arrays but using only $O(N)$ cells is equivalent to the RAM model with only 1-dimensional arrays.

To our knowledge, even the following problem asked by Luc Segoufin (personal communication) is open.

Open problem 1. *Given a set S of N pairs of integers (a, b) with $a, b < N$, can we preprocess S in $O(N)$ time, using only 1-dimensional arrays, so that when given $(c, d) \in [0, N[^2$ we can answer in constant time whether $(c, d) \in S$?*

Note that this problem is easily solved with a Boolean 2-dimensional array $N \times N$. Conversely, if we can positively solve the open problem 1, this does not necessarily mean that we can have k-dimensional arrays: arrays are dynamic (with interleaving of reads and writes) and store values (and not only Boolean information).

3 Complexity classes induced by the RAM model

Given a RAM machine $\mathcal{M}[\mathtt{Op}]$, we can define the class $TIME(f(N))$ as the class of problems for which there exists a program running on $\mathcal{M}[\mathtt{Op}]$ and solving the problem in time $O(f(N))$ (where N is the parameter of the RAM, that is both the value or the size of the input and the number such that the values stored within registers must be less than cN, for a fixed c).

This definition of $TIME$ depends on the specific RAM we start with. For instance, if `Op` includes the division, it is pretty clear that the problem of dividing two integers of value $O(N)$ can be done in time $O(1)$ but if `Op` only includes the addition, it is not clear that division is in $TIME(1)$.

[6] Indeed, for $r = \lfloor N^{1/d} \rfloor$ and $N \geq 2^d$, we have $N < (r+1)^d < r^{2d}$ with $r \geq 2$, and therefore $\lfloor N^{1/d} \rfloor^{2d} > N$.

In Subsection 3.1, we define a notion of "faithful simulation" of a RAM by another RAM which preserves their "dynamic" complexity in a very precise sense, see Lemma 2. This allows to define a very strict notion of "equivalence" between classes of RAMs. Building on that, we will introduce in Subsection 3.2, Definition 5, a second, weaker, criterion for equivalence that states that two RAM models are equivalent when they compute the same operations in constant time *after a linear-time initialization*.

We believe that this second equivalence criterion is very interesting. It is strong enough to show that two equivalent RAM models induce the same class $TIME(N)$ (and similarly for many other complexity classes) but will also allow us to show that a model simply equipped with the addition is equivalent to a model with the subtraction, multiplication, division, etc.

3.1 Emulation, faithful simulation and equivalence of classes of RAMs

The notions that we used in the previous section to show that our three instruction sets are equivalent are intuitive notions but, for the sake of rigor, we introduce below the technical notion of "emulation" as well as the "faithful simulation" and "equivalence" relations and their properties (Definition 2, Lemma 2 and Definition 3). First, let us introduce some useful conventions and notation.

Convention (registers of a RAM, output instructions)**.** *We consider that the* registers *of an AB-RAM (resp. a multi-memory RAM) are A, B and the $R[j]$ (resp. the integer variables C_i and the array elements $T_j[i]$).*

The instructions `Output` *(meaning output A),* `Output` i *(meaning output $R[i]$) and* `Output` α *(for an* `Op`*-expression α) are called* output instructions*. Each one means "output ρ", for a register ρ. (Note that the "identity" of a register of the form $T_j[\beta]$, that is to say its "address" β, with j,depends on the value of the expression β in the current configuration.)*

Notation (value of a register in a configuration)**.** *Let $\mathcal{C}$ be any configuration (AB-configuration, R-configuration, or array-configuration). The content of a register ρ in $\mathcal{C}$ is denoted* $\texttt{value}(\rho, \mathcal{C})$.

Notation (transitions between configurations)**.** *We write $\mathcal{C} \vdash^{M}_{\mathfrak{I},O} \mathcal{C}'$ (resp. $\mathcal{C} \vdash^{M,j}_{\mathfrak{I},O} \mathcal{C}'$) to signify that for an input $\mathfrak{I}$, a configuration $\mathcal{C}$ is transformed into a configuration $\mathcal{C}'$ by one computation step (resp. j computation steps) of the RAM M, while running with the input $\mathfrak{I}$ and writing the output O.*

We are now ready to give the formal definition of a faithful simulation:

Definition 2 (emulation, faithful simulation)**.** *Let M_1 and M_2 be two RAMs (among AB-RAMs, R-RAMs, or multi-memory RAMs). An* emulation *E of M_1 by M_2 is a triplet of maps $(\mathcal{I} \mapsto \mathcal{I}^E,\ \rho \mapsto \rho^E,\ \mathcal{C} \mapsto \mathcal{C}^E)$ which satisfy the properties (1-4) given below:*

- *a map $\mathcal{I} \mapsto \mathcal{I}^E$ associating to each instruction $\mathcal{I}$ of M_1 an instruction or a sequence of at most k instructions $\mathcal{I}^E$ of M_2, for some constant k,*
- *a map $\rho \mapsto \rho^E$ associating to each register ρ of M_1 a register ρ^E of M_2,*
- *a map $\mathcal{C} \mapsto \mathcal{C}^E$ associating to each configuration $\mathcal{C}$ of M_1 a configuration $\mathcal{C}^E$ of M_2.*

Properties satisfied by E:

1. *for each configuration $\mathcal{C}$ and each register ρ of M_1, we have $\texttt{value}(\rho, \mathcal{C}) = \texttt{value}(\rho^E, \mathcal{C}^E)$;*
2. *if $\mathcal{C}'$ is the configuration obtained from any configuration $\mathcal{C}$ of M_1 by an instruction $\mathcal{I}$ (of M_1) for an input $\mathfrak{I}$, then $\mathcal{C}'^E$ is obtained from the configuration $\mathcal{C}^E$ by the instruction (or sequence of at most k instructions) $\mathcal{I}^E$ (of M_2) for the same input $\mathfrak{I}$;*
3. *moreover, if $\mathcal{I}$ is an output instruction, meaning "output ρ", which executes on a configuration $\mathcal{C}$ of M_1, then the sequence of instructions $\mathcal{I}^E$ outputs ρ^E from the configuration $\mathcal{C}^E$, which gives the same outputted value $\texttt{value}(\rho^E, \mathcal{C}^E) = \texttt{value}(\rho, \mathcal{C})$; if at the opposite $\mathcal{I}$ is not an output instruction, then the sequence of instructions $\mathcal{I}^E$ outputs no value;*
4. *if $\mathcal{C}$ is the initial (resp. final) configuration of M_1, then $\mathcal{C}^E$ is also the initial (resp. final) configuration of M_2.*

When properties (1-4) are satisfied we say that M_2 faithfully simulates *M_1 by the emulation E.*

Remark 3. *Our notion of faithful simulation is a very precise variant, for RAMs, of the different notions of "real-time simulation" (or real-time reducibility) in the literature for different machine models: see, e.g., [70], or [28] who calls it a "lock-step simulation".*

The following result which follows from Definition 2 establishes that an emulation of a RAM M_1 by another RAM M_2 transforms a computation of M_1 into a computation of M_2 which computes exactly the *same thing* (the same outputs for the same inputs), exactly the *same way*, that is to say with the same steps of the same durations to within a constant factor.

Lemma 2 (faithful simulation of a RAM)**.** *Let M_1, M_2 be two RAMs such that M_2 faithfully simulates M_1 by an emulation E.*

1. *Then, for any input $\mathfrak{I}$, if the computation of M_1 on input $\mathfrak{I}$ stops in time τ and outputs O (written $\mathcal{C}_0 \vdash_{\mathfrak{I},O}^{M_1,\tau} \mathcal{C}_\tau$, where $\mathcal{C}_0$ (resp. $\mathcal{C}_\tau$) is the initial (resp. a final) configuration of M_1), then on the same input $\mathfrak{I}$ the RAM M_2 stops in time $\tau' \leq k * \tau$ (where k is the constant integer mentioned in Definition 2), which is $O(\tau)$, and produces the same output O, which is written $\mathcal{C}_0^E \vdash_{\mathfrak{I},O}^{M_2,\tau'} \mathcal{C}_\tau^E$.*

2. *Moreover, if the computation of M_1 divides into intervals of durations $\tau_1, \tau_2, \ldots, \tau_\ell$, so that within the jth computation interval, M_1 reads $\mathfrak{I}_j$ and outputs O_j, which is written $\mathcal{C}_0 \vdash_{\mathfrak{I}_1,O_1}^{M_1,\tau_1} \mathcal{C}_1 \vdash_{\mathfrak{I}_2,O_2}^{M_1,\tau_2} \cdots \vdash_{\mathfrak{I}_\ell,O_\ell}^{M_1,\tau_\ell} \mathcal{C}_\ell$, where $\mathcal{C}_0$ (resp. $\mathcal{C}_\ell$) is the initial (resp. a final) configuration of M_1, then we get $\mathcal{C}_0^E \vdash_{\mathfrak{I}_1,O_1}^{M_2,\tau_1'} \mathcal{C}_1^E \vdash_{\mathfrak{I}_2,O_2}^{M_2,\tau_2'} \cdots \vdash_{\mathfrak{I}_\ell,O_\ell}^{M_2,\tau_\ell'} \mathcal{C}_\ell^E$, with $\tau_j' \leq k * \tau_j$.*

Proof of item 1. Let $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_\tau$ be the computation of M_1 on input $\mathfrak{I}$. That means that we have $\mathcal{C}_0 \vdash_{\mathfrak{I},O_1}^{M_1} \mathcal{C}_1 \vdash_{\mathfrak{I},O_2}^{M_1} \cdots \vdash_{\mathfrak{I},O_\tau}^{M_1} \mathcal{C}_\tau$, where $\mathcal{C}_0$ (resp. $\mathcal{C}_\tau$) is the initial (resp. a final) configuration of M_1 and $O_1, O_2, \ldots, O_\tau$ are the successive outputs produced (notice that a "non output" instruction produces an empty output). Then, by definition of the emulation E (Definition 2), we also have $\mathcal{C}_0^E \vdash_{\mathfrak{I},O_1}^{M_2,i_1} \mathcal{C}_1^E \vdash_{\mathfrak{I},O_2}^{M_2,i_2} \cdots \vdash_{\mathfrak{I},O_\tau}^{M_2,i_\tau} \mathcal{C}_\tau^E$, for some positive integers $i_1, i_2, \ldots, i_\tau \leq k$, where $\mathcal{C}_0^E$ (resp. $\mathcal{C}_\tau^E$) is the initial (resp. a final) configuration of M_2. Therefore, we have $\mathcal{C}_0^E \vdash_{\mathfrak{I},O}^{M_2,\tau'} \mathcal{C}_\tau^E$, with the output $O = O_1, O_2, \ldots, O_\tau$ of M_1, and the computation time $\tau' = i_1 + i_2 + \cdots + i_\tau \leq k * \tau = O(\tau)$. □

Proof of item 2. From $\mathcal{C}_{j-1} \vdash_{\mathfrak{I}_j,O_j}^{M_1,\tau_j} \mathcal{C}_j$, for $j = 1, \ldots, \ell$, we deduce (by a proof similar to that of item 1) that, for each j, we have $\mathcal{C}_{j-1}^E \vdash_{\mathfrak{I}_j,O_j}^{M_2,\tau_j'} \mathcal{C}_j^E$, for some $\tau_j' \leq k * \tau_j$. This is the expected result. □

Remark 4 (constant time after linear-time preprocessing)**.** *Lemma 2 has many applications. For example, let M_1 be a RAM whose computation divides into two phases:*

1. *M_1 reads an input $\mathfrak{I}_1$ and computes an output O_1 in time T_1;*

2. *after reading a second input $\mathfrak{I}_2$, the RAM M_1 computes in time T_2 a second output O_2 (depending on O_1 and $\mathfrak{I}_2$).*

Let M_2 be a RAM which faithfully simulates M_1. Then the computation of M_2 is exactly similar to that of M_1. This means that the computation of M_2 divides into the same two phases (1,2) with the same outputs O_1, O_2 for the same inputs $\mathfrak{I}_1, \mathfrak{I}_2$. There is only one difference: the durations of phases 1 and 2 of M_2 are respectively $O(T_1)$ and $O(T_2)$ (instead of T_1 and T_2).

*In this paper, we will be particularly interested in the complexity class called "*constant time after linear-time preprocessing*", defined in Subsection 3.2. This means that the time bound T_1 of the first phase of the RAM computation is* linear*, i.e. is $O(N)$, where N is the size of the first input $\mathfrak{I}_1$ (the first phase is called "linear-time preprocessing"), and the time bound T_2 of the second phase, which reads a second input $\mathfrak{I}_2$ of constant size, is* constant*.*

Let us define what it means that two classes of RAMs (for example, the class of AB-RAMs, or R-RAMs, or multi-memory RAMs) are equivalent:

Definition 3 (faithful simulation relation and equivalence of classes of RAMs)**.** *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two classes of RAMs. We say that $\mathcal{R}_2$* faithfully simulates *$\mathcal{R}_1$ if for each RAM $M_1 \in \mathcal{R}_1$ there are a RAM $M_2 \in \mathcal{R}_2$ and an emulation of M_1 by M_2. (Obviously, the faithful simulation relation is reflexive and transitive.) If moreover $\mathcal{R}_1$* faithfully simulates *$\mathcal{R}_2$, then we say that $\mathcal{R}_1$ and $\mathcal{R}_2$ are* equivalent*.*

3.2 Complexity classes in our RAM model

The *equivalence* we have defined using the notion the *faithful simulation*, is quite natural but it is very strong. Using this notion we can show that two equivalent RAMs define the same complexity classes but the converse is not necessarily true. For instance, as we will show in Proposition 1, the RAM equipped with addition *cannot* faithfully simulate the RAM equipped with the addition and the multiplication, but we will see in the next sections that they define the same complexity classes of "minimal time": linear time, etc.

In this subsection, we will introduce a somewhat weaker notion of equivalence that makes two classes of RAMs "equivalent" if one class can simulate the other (and vice versa) *after a linear time initialization*. While this notion might seem less natural, we can still show that the interesting complexity classes are the same for "equivalent" RAMs and we can prove our (surprising!) results making the RAM with addition equivalent to the RAM equipped with all the usual arithmetic primitives.

First, the following result shows that the notion of faithful simulation turns out to be too restrictive:

Proposition 1. $\mathcal{M}[\{+\}]$ cannot *faithfully simulate* $\mathcal{M}[\{+,\times\}]$.

Proof. To prove that $\mathcal{M}[\{+\}]$ cannot faithfully simulate $\mathcal{M}[\{+,\times\}]$, we only need to show that given two inputs $I[0]$ and $I[1]$, strictly less than N, a program in $\mathcal{M}[\{+\}]$ cannot output $I[0]\times I[1]$ in constant time.

For that, we will consider such a program P and suppose that the RAM starts initialized with zeros. We will consider M_k, the largest value strictly below N that the RAM has access to after the k-th step (i.e. any constant in the program, any value in the input or in any register).

First, M_0 is the maximum value among $I[0]$, $I[1]$, all the constants $c_1,\dots,c_r$ in the program and the maximum of registers (which are all set to 0). For M_{k+1} we see that the only way of creating a larger maximum value is by summing two values and in that case we still have $M_{k+1}\leq 2\times M_k\leq 2^{k+1}M_0$ (note that by summing N and anything else we get something bigger than N and therefore it won't change the value of M_k).

Overall, we have $M_k\leq 2^k\times M_0$ and thus by setting $I[0]=I[1]=\lfloor N^{1/3}\rfloor$ and taking N big enough, we have $I[0]\times I[1]=\lfloor N^{1/3}\rfloor^2>2^\ell\times M_0\geq M_\ell$, where the *constant* integer ℓ is the maximum number of steps executed by program P, because $M_0=\max(I_0,I_1,c_1,\dots,c_r)=O(N^{1/3})$. Therefore, the product $I[0]\times I[1]$ cannot be obtained at the ℓ-th (final) step or before. □

Remark 5. *Proposition 1 is not an isolated result : it has several variants, proved by adapting the above proof. For example, the same proof also establishes that* $\mathcal{M}[\{+,\dot{-}\}]$ cannot *faithfully simulate* $\mathcal{M}[\{+,\dot{-},\times\}]$ *where* $\dot{-}$ *is the "subtraction" defined by* $x\dot{-}y:=\max(0,x-y)$.

Similarly, $\mathcal{M}[\{+\}]$ cannot *faithfully simulate* $\mathcal{M}[\{+,\dot{-}\}]$ *because, for* $I[0],I[1]<N$, *a program in* $\mathcal{M}[\{+\}]$ *cannot output* $I[0]\dot{-}I[1]$ *in constant time; indeed, at each step of a constant-time program in* $\mathcal{M}[\{+\}]$, *each register content is equal to a linear combination of the form* $a_0I[0]+a_1I[1]+a_2N+b_1c_1+\cdots+b_rc_r$, *where* $c_1,\dots,c_r$ *are the constants of the program and the integer coefficients* a_i,b_j *depend on the current step and the register involved; the important point is that the number of possible tuples of coefficients* $(a_0,a_1,a_2,b_1,\dots,b_r)$ *is* bounded by a constant, *implying that, for* N *large enough, such a linear combination cannot be equal to* $I[0]\dot{-}I[1]$ *when we take* $I[0]=\lfloor N^{1/2}\rfloor$ *and* $I[1]=\lfloor N^{1/3}\rfloor$.

Similarly, we invite the reader to demonstrate that $\mathcal{M}[\{+,\times\}]$ *cannot faithfully simulate* $\mathcal{M}[\{+,\times,/\}]$ *where / denotes the Euclidean division.*

Operations computed in constant time after linear-time initialization: We say that a RAM $\mathcal{M}[\mathtt{Op}]$ computes an operation $\mathtt{op}$ *in constant time after a linear-time initialization* if it works in two successive phases:

1. *A pre-computation phase* where the RAM reads the integer $N\geq 1$ and computes an "index" $p(N)$ in time $O(N)$. The pre-computed "index" $p(N)$ will be a finite set of arrays (tables) and constants depending only on N, not on the rest of the input.

2. *A computation phase* where the RAM reads the input $\mathcal{X}:=(x_1,\dots,x_k)\in[0,cN]^k$, for fixed positive integers k,c and outputs in constant time the result $\mathtt{op}(x_1,\dots,x_k)\in[0,cN]$.

In this paper, we will mainly focus on the following problem: what operations can a RAM with addition (i.e., with $\mathtt{Op}:=\{+\}$) compute in constant time after linear-time initialization? We will show how this can be generalized to RAMs which compute operations acting on "polynomial"

operands with a "polynomial" result: a "polynomial" integer is an integer smaller than N^d, for a constant integer $d > 1$, and it is naturally represented by the sequence of its d "digits" in the "standard" base N.

Definition 4 (set of operations $\mathcal{M}[\mathtt{Op}]$)**.** *By abuse of notation, we call* $\mathcal{M}[\mathtt{Op}]$ *the set of operations* op *for which there exists a multi-memory RAM program using only the* Op *operations that computes* op *in constant time after linear initialization.*

Definition 5 (equivalence of sets of operations, equivalence of RAM models)**.** *Two sets of operations,* $\mathtt{Op}_1$ *and* $\mathtt{Op}_2$ *are* equivalent *when the sets of operations* $\mathcal{M}[\mathtt{Op}_1]$ *and* $\mathcal{M}[\mathtt{Op}_2]$ *are equal; we will also say that the RAM models* $\mathcal{M}[\mathtt{Op}_1]$ *and* $\mathcal{M}[\mathtt{Op}_2]$ *are* equivalent*.*

These definitions will be justified by Corollary 2 below.

In this paper, we will mainly consider the two following complexity classes: *Linear time* and *Constant time with linear-time initialization.* Recall that constant-time complexity is not a robust notion, as explained above, unless the constant-time computation is preceded by a linear-time initialization (also called linear-time preprocessing).

Linear time complexity: A RAM with an integer $N > 0$ as input (resp. with a list of integers $\mathfrak{I} \coloneqq (N, I[0], \ldots, I[N-1])$, $N > 0$, as input) *works in linear time* if it uses only integers $O(N)$ as addresses and contents of registers and stops after $O(N)$ steps.

Constant time after linear-time preprocessing: A RAM with a list of integers $\mathfrak{I} \coloneqq (N, I[0], \ldots, I[N-1])$, $N > 0$, as input and using only integers $O(N)$ (as addresses and contents of registers) *works in constant time after linear-time preprocessing* if it works in two successive phases:

1. *Pre-computation:* the RAM computes an "index" $p(\mathfrak{I})$ in time $O(N)$;
2. *Computation:* from $p(\mathfrak{I})$ and after reading a second input $\mathcal{X}$ of constant size, the RAM returns in constant time the corresponding result $\mathtt{output}(\mathfrak{I}, \mathcal{X})$.

Remark 6. *In our concluding section (Section 10), we will also study a third class of problems of "minimal" computational complexity: the now well-known class of problems* enumerable with constant delay after linear-time preprocessing.

The main result of this paper is the proof of the robustness of our three "minimal" complexity classes with respect to the set Op of arithmetic operations allowed, provided that Op contains addition. To begin with, the following proposition expresses that the two classes defined above are robust with respect to other variations of the instruction set.

Proposition 2. *Let* Op *be a finite set of operations* including addition. *Then the class of problems computable in linear time (resp. in constant time after linear-time preprocessing) on a RAM with* Op *operations does not change depending on whether:*

- *the program is given by a sequence of AB-instructions;*
- *the program is given by a sequence of R-instructions;*
- *the program is given by a sequence of array-instructions.*

Proof. It is a direct consequence of Lemma 1 and its proof. The only condition to check is that in the three simulations (1-3) of this lemma, each integer handled is $O(N)$. This is evident for simulations 1 and 2. In simulation 3 of an array-instruction by a sequence of R-instructions (the only one of the three simulations that explicitly uses addition), it suffices to notice that the address y of a register $R[y]$ which simulates an array element $T_j[x]$ is linear in its index x. □

As shown by the proofs of Lemma 1 and Proposition 2, the availability of addition seems essential to implement (one-dimensional) arrays in the RAM model. Most of this paper is devoted to showing that a RAM with only addition can implement the other usual arithmetic operations, subtraction, multiplication, division, etc., in constant time after linear-time preprocessing.

3.3 Reductions between operations in the RAM model

To show the robustness of the RAM model $\mathcal{M}[\mathtt{Op}]$ with respect to the set $\mathtt{Op}$ of allowed operations, it will be convenient to introduce a notion of *reduction* between operations or sets of operations.

By Definition 4, we have $\mathtt{op} \in \mathcal{M}[\mathtt{Op}]$, for an operation $\mathtt{op}$ of arity k, if there are two RAM procedures in $\mathcal{M}[\mathtt{Op}]$, a "preprocessing" procedure, suggestively called LinPreproc() and acting in time $O(N)$, and a "computation" procedure, called CstProc($\mathbf{x}$), for $\mathbf{x} = (x_1, \ldots, x_k)$, and acting in constant time, such that after executing LinPreproc(), executing CstProc($\mathbf{x}$) returns $\mathtt{op}(\mathbf{x})$, for each k-tuple of integers $\mathbf{x}$ in the definition domain of the $\mathtt{op}$ operation.

Example 1. *Let $c \geq 1$ be a fixed integer. (It will be convenient to choose c such that cN is an upper bound of the integers allowed as contents of the RAM registers.) To define the arithmetic operations by induction we will need the predecessor function* $\mathtt{pred} : x \mapsto x - 1$ *from* $[1, cN]$ *to* $[0, cN[$. *To demonstrate* $\mathtt{pred} \in \mathcal{M}[\{+\}]$, *we use the following procedures of* $\mathcal{M}[\{+\}]$, *where cN denotes the sum $N + \cdots + N$ (c times):*

Algorithm 3 Computation of PRED

```
1: procedure LinPreprocPred()
2:     y ← 0
3:     while y ≠ cN do
4:         PRED[y + 1] ← y
5:         y ← y + 1
```

```
1: procedure Pred(x)
2:     return PRED[x]
```

Clearly, running LinPreprocPred() *computes in $O(N)$ time the array* $\mathtt{PRED}[1..cN]$ *defined by* $\mathtt{PRED}[y] \coloneqq y - 1$, *for $y \in [1, cN]$, so that any (constant time) call* Pred(α), *for $\alpha \in [1, cN]$, returns* $\mathtt{pred}(\alpha)$. *This proves* $\mathtt{pred} \in \mathcal{M}[\{+\}]$.

Lemma 3. *1. Any program $P \in \mathcal{M}[\{+, \mathtt{pred}\}]$, i.e. using the operations $+$ and* pred *is faithfully simulated by a program $P' \in \mathcal{M}[\{+\}]$ after a linear-time initialization also using only the operation $+$.*

2. Therefore, we have the equality $\mathcal{M}[\{+, \mathtt{pred}\}] = \mathcal{M}[\{+\}]$. This means that the two sets of operations computed in constant time after linear-time initialization by a RAM using the operation $+$ and using *or* not using *the* pred *operation are equal.*

Proof of item 1. P' starts by calling the procedure LinPreprocPred (linear-time initialization) and then calls P but where each occurrence of the pred operation in P is replaced with a call to the Pred procedure. □

The following general lemma establishes a "transitivity" property.

Lemma 4 (Fundamental Lemma)**.** *Let* Op *be a set of operations including $+$ and* pred. *Let P be a program in $\mathcal{M}[\mathtt{Op} \cup \{\mathtt{op}\}]$, i.e. using the operations of $\mathtt{Op} \cup \{\mathtt{op}\}$, where* op *is an operation in $\mathcal{M}[\mathtt{Op}]$, $\mathtt{op} \notin \mathtt{Op}$.*

Then P can be faithfully simulated by a program P' in $\mathcal{M}[\mathtt{Op}]$, i.e. using only the operations in Op *after a linear-time initialization, also using only the operations of* Op.

Proof. Let P be a program in $\mathcal{M}[\mathtt{Op} \cup \{\mathtt{op}\}]$. Intuitively, the idea of the proof is to create a program P' that starts by calling LinPreproc, the initialization procedure for op, and then calls P but where each occurrence of the op operation in P is replaced with a call to CstProc, the computation procedure of op. This is the essence of our proof but we also need to take care of the following problem:

- The definition of $\mathtt{op} \in \mathcal{M}[\mathtt{Op}]$ does not guarantee that it is possible to make two (or more) calls of CstProc. For instance, we might imagine that CstProc is destructive and prevents us from executing a second call to CstProc or that the answer might be wrong.

To solve this problem, we can modify the procedure CstProc to ensure that several successive calls are possible. For that, we use a modified procedure CstProc$'$ where we store the value of each array cell or integer variable that is overwritten and at the end of the call to CstProc we restore the memory back to its state before the call. This is possible by maintaining a complete "log" (a history) of the successive modifications (assignments/writes) of the memory. At the end of the call to CstProc$'$, before its return instruction, the initial state of the memory is restored, step by step, by reading the "log" in the anti-chronological direction.

For that, we introduce three new arrays called OldVal, Index and Dest, as well as two new variables called NbWrite and ReturnValue. The modified procedure CstProc$'$ starts by setting

$\texttt{NbWrite} = 0$, and then, for each instruction in CstProc that writes something in an individual variable x_i or in an array cell $A_i[t]$, i.e. an instruction of the form $x_i \leftarrow u$ or $A_i[t] \leftarrow u$, we write into $\texttt{OldVal}[\texttt{NbWrite}]$ the current value v of x_i or $A_i[t]$ that will be overwritten; afterwards, we write into $\texttt{Dest}[\texttt{NbWrite}]$ the integer i that indicates[7] which variable x_i or array A_i is being written into, and, if the write was a write into an array A_i (i.e. if the integer i is greater than k), we note the value of the index t of the cell $A_i[t]$ overwritten into $\texttt{Index}[\texttt{NbWrite}]$; finally, we write the new value u into x_i or $A_i[t]$ and we increment $\texttt{NbWrite}$ by one.

At the end of the procedure, instead of returning directly the expected value, we store it in the $\texttt{ReturnValue}$ variable. Then, we restore the memory by replaying the "log" (write history) in reverse. To do this, we decrease the counter variable $\texttt{NbWrite}$ by one as long as it is strictly positive and, each time, we restore the value $v = \texttt{OldVal}[\texttt{NbWrite}]$ into the array cell (resp. variable) indicated by $i = \texttt{Dest}[\texttt{NbWrite}]$, at index $\texttt{index} = \texttt{Index}[\texttt{NbWrite}]$ for an array cell. This means that we execute the assignment $A_i[\texttt{index}] \leftarrow v$ if $i > k$ (resp. $x_i \leftarrow v$ if $i \leq k$). Once done, we return the value stored in the $\texttt{ReturnValue}$ variable.

Formally, the code of the CstProc′() procedure is the concatenation of the following calls using the procedures of Algorithm 4 below:

- InitCstProc'();
- InitCstProc() where each instruction of the form $\texttt{x}_\texttt{i} \leftarrow u$ (resp. $\texttt{A}_\texttt{i}[t] \leftarrow u$, $\texttt{Return}\ t$) is replaced by the call AssignVar($\texttt{i}, u$) (resp. AssignCell($\texttt{i}, t, u$), StoreReturn(t));
- FinalCstProc'().

It is important to note that the only operations used by those procedures are + and pred, which is used at line 3 of the FinalCstProc' procedure.

Finally, note that the time overhead of the procedure CstProc′ is proportional to the number of assignments performed by the original constant-time procedure CstProc it simulates, therefore the CstProc′ procedure is also constant-time. □

Algorithm 4 Procedures used to construct CstProc' from the code of CstProc whose list of variables is $\texttt{x}_0, \ldots, \texttt{x}_k$ and the list of arrays is $\texttt{A}_{k+1}, \ldots, \texttt{A}_{k'}$

```
1: procedure InitCstProc'()
2:     NbWrite ← 0

1: procedure AssignVar(i,u)                ▷ x_i ← u
2:     OldVal[NbWrite] ← x_i
3:     Dest[NbWrite] ← i
4:     x_i ← u
5:     NbWrite ← NbWrite + 1

1: procedure AssignCell(i,t,u)             ▷ A_i[t] ← u
2:     OldVal[NbWrite] ← A_i[t]
3:     Dest[NbWrite] ← i
4:     Index[NbWrite] ← t
5:     A_i[t] ← u
6:     NbWrite ← NbWrite + 1

1: procedure StoreReturn(t)
2:     ReturnValue ← t

1: procedure FinalCstProc'()
2:     while NbWrite > 0 do
3:         NbWrite ← NbWrite − 1
4:         i ← Dest[NbWrite]
5:         if i > k then
6:             ind ← Index[NbWrite]
7:             A_i[ind] ← OldVal[NbWrite]
8:         else
9:             x_i ← OldVal[NbWrite]
10:    return ReturnValue
```

The following result is a direct consequence of Lemma 4:

Corollary 1. *Let* Op *be a set of operations including* + *and* pred*, and let* op′ *be an operation in* $\mathcal{M}[\texttt{Op} \cup \{\texttt{op}\}]$ *where* op *is an operation in* $\mathcal{M}[\texttt{Op}]$*. Then we have* $\texttt{op}' \in \mathcal{M}[\texttt{Op}]$*.*

Proof. If op′ is in $\mathcal{M}[\texttt{Op} \cup \{\texttt{op}\}]$ and op is an operation in $\mathcal{M}[\texttt{Op}]$, then we can emulate the linear-time preprocessing and the constant-time computation of op in $\mathcal{M}[\texttt{Op}]$. □

Corollary 2. *Let* Op *be a set of operations including* + *and* pred*, and let* op *be an operation in* $\mathcal{M}[\texttt{Op}]$*. Then we have the equality of sets of operations* $\mathcal{M}[\texttt{Op} \cup \{\texttt{op}\}] = \mathcal{M}[\texttt{Op}]$*.*

[7] It is convenient to assume that the set of integers $i \leq k$ which identify the individual variables x_i is disjoint from the set of integers $i > k$ that number the arrays A_i. Note that the test $i \leq k$ does not actually use the order symbol $\leq$ since it is equivalent to the disjunction of $k+1$ equalities $\bigvee_{j=0}^{k} i = j$.

Starting from the equality $\mathcal{M}[\{+\}] = \mathcal{M}[\{+, \texttt{pred}\}$ of Lemma 3 and using iteratively Corollary 2, we deduce the following result.

Corollary 3 (Transitivity Corollary)**.** *Let* $\texttt{op}_1, \ldots, \texttt{op}_k$ *be a list of operations. Let us define the set of operations* $\texttt{Op}_0 \coloneqq \{+, \texttt{pred}\}$ *and* $\texttt{Op}_i \coloneqq \texttt{Op}_{i-1} \cup \{\texttt{op}_i\}$*, for* $i = 1, \ldots, k$*. Assume* $\texttt{op}_i \in \mathcal{M}[\texttt{Op}_{i-1}]$*, for* $i = 1, \ldots, k$*.*

Then we have the sequence of equalities $\mathcal{M}[\{+\}] = \mathcal{M}[\{+, \texttt{pred}\}] = \mathcal{M}[\texttt{Op}_1] = \cdots = \mathcal{M}[\texttt{Op}_k]$*. This means that each set of operations* $\texttt{Op}_i$ *thus defined recursively is equivalent to* $\{+\}$*.*

Proof. The first equality is given by Lemma 3. Each of the k following equalities are deduced from Corollary 2 by setting $\texttt{Op} \coloneqq \texttt{Op}_{i-1}$ and $\texttt{op} \coloneqq \texttt{op}_i$, for each $i = 1, \ldots k$. □

From now on, we will make repetitive and informal (intuitive) use of Lemma 4 (Fundamental Lemma) and Corollary 3 (Transitivity Corollary) without mentioning them explicitly.

Remark 7 (Cumulative property)**.** *The result of Corollary 3 is "cumulative". As we will use for the successive operations studied in the following sections, the preprocessing and computation of an* $\texttt{op}_i$ *operation can use any operation* $\texttt{op}_j$ *introduced before* $(j < i)$ *but also any element – array, constant integer – pre-computed previously, i.e. constructed to compute a previous* $\texttt{op}_j$*.*

4 Computing the sum, difference, product and base change in constant time

In this section, we show that the first three usual operations acting on "polynomial" integers can be performed in constant time on a RAM of $\mathcal{M}[\{+\}]$ after a linear-time preprocessing.

4.1 Moving from registers integers to a smaller base

A fundamental property for establishing the robustness of complexity classes in the RAM model is its ability to pre-compute *some tables. In particular, in this subsection, we show how a RAM with addition can* compute in linear time *several tables which allow to transform a register integer x into an integer in base B, with B large enough, so that x fits within a constant number of registers, but small enough to allow us, in the following subsections, to pre-compute in* constant time *the usual arithmetic operations on integers less than B: subtraction, multiplication, base change, division, etc. This in turn will allow us to compute in constant time operations over $O(N)$ integers.*

A general principle to implement binary arithmetic operations in constant time: To implement in constant time an arithmetic binary operation op, e.g. subtraction or multiplication, on operands less than N (or less than N^d, for a fixed integer d), it is convenient to convert these operands into a base B such that $N^{1/2} \leq B = O(N^{1/2})$. We will see below how it allows

- on the one hand, to represent any natural number $a < N$ (and therefore less than B^2) by its two digits a_1, a_0 in the base B: $a_1 = a \texttt{ div } B$ and $a_0 = a \texttt{ mod } B$,
- on the other hand, for any constant integer $c \geq 1$, to construct in time $O(B^2) = O(N)$ a "binary" table $T_{\texttt{op}}[0..B-1][0..B-1]$ of a binary operation op restricted to operands less than B, i.e., such that, for $x, y < B$, $T_{\texttt{op}}[x][y] \coloneqq x \texttt{ op } y$.

For this, we must pre-compute the constant $B \coloneqq \lceil \sqrt{N} \rceil$ and then the tables of the functions $x \mapsto x \texttt{ div } B$ and $x \mapsto x \texttt{ mod } B$, for $x < cN$, and of the function $x \mapsto Bx$, for $x < B$. This will be done below.

Computing square roots: we will construct the array $\texttt{CEIL\_SQRT}[1..N]$ and the constant B defined by $\texttt{CEIL\_SQRT}[y] \coloneqq \lceil \sqrt{y} \rceil$, for $1 \leq y \leq N$ and $B \coloneqq \lceil \sqrt{N} \rceil$. They are computed in time $O(N)$ by the following code using only equality tests and the addition operation:

Algorithm 5 Computation of SQRT, with $\text{SQRT}(x) := \lceil \sqrt{x} \rceil$ for $1 \leq x \leq N$, and $\texttt{B} := \lceil \sqrt{\texttt{N}} \rceil$

```
1: procedure LinPreprocSqrt()
2:     x ← 1
3:     xSq ← 1
4:     for y from 1 to N do
5:         CEIL_SQRT[y] ← x
6:         if y = xSq then
7:             xSq ← xSq + x + x + 1
8:             x ← x + 1
```

```
1: procedure Sqrt(x)
2:     return CEIL_SQRT[x]
```

```
1: procedure B()
2:     return CEIL_SQRT[N]
```

Proof of correctness. When initializing the variables $\texttt{x}, \texttt{xSq}$ (at lines 2-3) and after each update (lines 7-8), we always have $\texttt{xSq} = \texttt{x}^2$ due to the identity $(x+1)^2 = x^2 + x + x + 1$. Besides, the inequalities $(\texttt{x}-1)^2 < \texttt{y} \leq \texttt{x}^2$ (meaning $\texttt{x} = \lceil \sqrt{\texttt{y}} \rceil$) are maintained throughout the execution of the algorithm: first, they hold at the initialization, that is $\texttt{y} = 1 = \texttt{x}^2$; second, if $(\texttt{x}-1)^2 < \texttt{y} \leq \texttt{x}^2$, then either we have $(\texttt{x}-1)^2 < \texttt{y}+1 \leq \texttt{x}^2$ (case $\texttt{y} < \texttt{x}^2$), or we have $\texttt{x}^2 < \texttt{y}+1 \leq (\texttt{x}+1)^2$ (case $\texttt{y} = \texttt{x}^2$). This justifies the instruction $\texttt{CEIL\_SQRT}[\texttt{y}] \leftarrow \texttt{x}$. □

Choosing a new reference base: We have defined the base $B := \lceil \sqrt{N} \rceil$. Now that we are manipulating integers represented in base B, an integer $O(N)$ cannot be stored in a single register. For the sake of simplicity, we will represent integers as bounded arrays with a constant size d, hence a representation able to represent all the integers up to $N^{d/2}$.

Conversion between integers less than cN and integers in base B: An array $\texttt{a}[0..d-1]$ of size d *represents in base* B the integer $\sum_{0 \leq i < d} \texttt{a}[i] B^i$. In general we will assume that the representation is *normalized* which means that $\texttt{a}[i] < B$ for all $0 \leq i < d$. All the procedures that we are going to present return normalized integers but they also often accept non normalized integers as inputs. For functions that only accept normalized integers, we can normalize them using the NORMALIZE function. Let us now explain how to move from register integers less than cN (recall that $c \geq 1$ is defined as the integer such that cN is an upper bound of the register contents) into base B integers (i.e. arrays of d integers less than B).

Let us define the arrays $\texttt{DIV}B[0..cN]$ and $\texttt{MOD}B[0..cN]$ by $\texttt{DIV}B[x] := x \texttt{ div } B$ and $\texttt{MOD}B[x] := x \texttt{ mod } B$, for $x \leq cN$. Clearly, $\texttt{DIV}B$ and $\texttt{MOD}B$ are computed by the following algorithm running in time $O(N)$, and the procedure $\text{ToBaseB}(\texttt{x}, \texttt{Out})$, which computes for an integer $x \leq cN$ its base B representation[8] $\texttt{Out}[0..d-1]$, with $d \geq 3$, runs in constant time.

Algorithm 6 Computation of the base conversion

```
1: procedure LinPreprocBaseConv()
2:     DIVB[0] ← 0
3:     MODB[0] ← 0
4:     for x from 1 to cN do
5:         if MODB[x − 1] ≠ B − 1 then
6:             MODB[x] ← MODB[x − 1] + 1
7:             DIVB[x] ← DIVB[x − 1]
8:         else
9:             MODB[x] ← 0
10:            DIVB[x] ← DIVB[x − 1] + 1
11:            MULTB[DIVB[x]] ← x
```

```
1: procedure ToBaseB(x)
2:     Out ← array of size d
3:     for i from 0 to d − 1 do
4:         Out[i] ← MODB[x]
5:         x ← DIVB[x]
6:     return Out
```

```
1: procedure Normalize(In)
2:     carry ← 0
3:     Out ← array of size d
4:     for i from 0 to d − 1 do
5:         temp ← In[i] + carry
6:         carry ← DIVB[temp]
7:         Out[i] ← MODB[temp]
8:     return Out
```

Proof of correctness. The arrays DIVB and MODB are initialized such that $\texttt{DIVB}[x] = \lfloor x/B \rfloor$ and $\texttt{MODB}[x] = x \texttt{ mod } B$, for $x \leq cN$. The array MULTB satisfies the equality $\texttt{MULTB}[y] = y \times B$ for all y such that $y \times B \leq cN$. Obviously, the procedure NORMALIZE correctly normalizes any array In which represents (in base B) an integer less than B^d. □

[8] Note that for $N > c^2$ we have $c < \left\lceil \sqrt{N} \right\rceil$, which implies $cN < \left\lceil \sqrt{N} \right\rceil^3$, so that any integer $x \leq cN$ can be represented with 3 digits in base $B = \left\lceil \sqrt{N} \right\rceil$.

4.2 Computing sum, difference and product in constant time

In the previous subsection, we showed that our integers $O(N)$ could be moved into a base B such that $N^{1/2} \leq B = O(N^{1/2})$. This allows us to create d-dimensional arrays $B \times \cdots \times B$, for a fixed $d \geq 3$, storing the results for each "digit" of our base and by combining them we will get the result for integers $O(N)$. This technique will be first used for the comparison *(≤-inequality test) of integers and then we will show how to apply it to the* sum, difference *and* product *of integers $O(N)$, or even "polynomial" integers, i.e. integers less than $N^{d/2}$ (since $N^{d/2} \leq B^d$).*
The algorithms presented here are variants of the standard algorithms for large numbers manipulation but here they run in constant time and rely on the pre-computed tables (whereas standard algorithms generally suppose that each operation is available on registers).

Adding comparison of integers: In our RAM model we assumed that we cannot compare large integers by inequalities, $<$, $\leq$, etc., but we saw a way to move from large integers (up to cN) to relatively small integers (less than B). We now see how to pre-compute an array $\texttt{LowerEQ}[0..B-1][0..B-1]$ such that $\texttt{LowerEQ}[\texttt{x}][\texttt{y}] := 1$ when $x \leq y$ and 0 otherwise. Using this array and a simple for loop we will be able to compare integers lower than B^d by inequalities, all of that just using the addition and `pred`. Note that this comparison requires that both arguments are *normalized*!

Algorithm 7 Computation of $\leq$-comparison of integers lower than $\texttt{B}^d$

```
1: procedure LinPreprocComp()
2:     for x from 0 to B − 1 do
3:         for y from x to B − 1 do
4:             LowerEQ[x][y] ← 1
5:         for y from x−1 downto 0 do
6:             LowerEQ[x][y] ← 0
```

```
1: procedure LowerEqualBaseB(a, b)
2:     for i from d − 1 downto 0 do
3:         if a[i] ≠ b[i] then
4:             return LowerEQ[a[i]][b[i]]
5:     return 1                    ▷ Case of equality

1: procedure LowerEqual(a, b)
2:     return LowerEqualBaseB(
                          ToBaseB(a),
                          ToBaseB(b))
```

Proof of correctness. Once the array `LowerEQ` is initialized as expected, if we take two arrays of size d representing normalized integers lower than B^d then it is clear that we just have to look at the biggest index where the arrays differ and compare the values contained at this position. If the arrays do not differ, then the numbers are equal. □

Sum and difference: The sum and difference (of integers less than B^d) are pretty straightforward applications of the textbook algorithms, we just have to make sure to propagate the carries in the right way. For the difference, we pre-compute an array $\texttt{DIFF}[0..2B-1][0..2B-1]$ such that $\texttt{DIFF}[x][y] := x - y$ for $0 \leq y \leq x < 2B$.

Note that the sum is not checking for a potential overflow (i.e. when the result is greater than or equal to B^d) and the difference is not verifying that the number represented by a is actually greater than the number represented by b but in both cases we could read it from the `carry` variable. Those algorithms work as if the numbers are treated modulo B^d (just like many modern computers work modulo 2^{64}). Note also that the procedure Difference expects normalized arguments but the Sum procedure works even if the given arguments are not normalized and returns a normalized number.

Algorithm 8 Computation of sum and difference

```
1: procedure LinPreprocSumDiff()
2:     for x from 0 to 2 × B − 1 do
3:         DIFF[x][x] ← 0
4:         for y from x − 1 downto 0 do
5:             DIFF[x][y] ← DIFF[x][y + 1] + 1

1: procedure Sum(a, b)
2:     res ← array of size d
3:     for i from 0 to d − 1 do
4:         res[i] ← a[i] + b[i]
5:     return Normalize(res)

1: procedure Difference(a, b)
2:     carry ← 0
3:     res ← array of size d
4:     for i from 0 to d − 1 do
5:         c ← b[i] + carry
6:         if LowerEQ[c][a[i]] then
7:             res[i] ← DIFF[a[i]][c]
8:             carry ← 0
9:         else
10:            res[i] ← DIFF[a[i] + B][c]
11:            carry ← 1
12:    return res
```

Proof of correctness. The sum is done component by component. The result is not necessarily a normalized integer (even if the arguments a, b are normalized) but we normalize it afterward: finally, the integer represented by res is the sum of the integers val(a) and val(b) represented by a and b if it is less than B^d (more generally, it is $\texttt{val(a)} + \texttt{val(b)} \mod B^d$).

For the difference, the idea is to maintain a carry along the algorithm. When the carry is set to one, it means that we need to subtract it from the next "digit". Notice that storing a carry of 1 is enough as $\texttt{b}[i] < B$ and $\texttt{carry} \leq 1$ and therefore $\texttt{b}[i] + \texttt{carry} \leq B \leq B + \texttt{a[i]}$. Notice also that $\texttt{a[i]} < B$ and thus $\texttt{a[i]} + B < 2B$, which justifies that the array DIFF only needs to be initialized up to $2B - 1$. Finally, note that the Difference procedure uses not only the pre-computed array DIFF but also the array LowerEQ pre-computed in Algorithm 7 above. □

Multiplication: The recipe to get the multiplication is, unsurprisingly, the same as before: we pre-compute an array $\texttt{MULT}[0..B-1][0..B-1]$ and use it with the naive textbook multiplication algorithm.

Algorithm 9 Computation of multiplication

```
1: procedure LinPreprocMult()
2:     for x from 0 to B − 1 do
3:         MULT[x][0] ← 0
4:         for y from 0 to B − 2 do
5:             MULT[x][y + 1] ← MULT[x][y] + x

1: procedure Multiply(a, b)
2:     res ← array of size d
3:     for i from 0 to d − 1 do
4:         res[i] ← 0
5:     for i from 0 to d − 1 do
6:         for j from 0 to d − 1 − i do
7:             res[i + j] ← res[i + j]+
8:                          MULT[a[i]][b[j]]
9:     return Normalize(res)
```

Proof of correctness. Here MULT stores a result that might be larger than B, which means that in the multiplication res will not be normalized but it will get normalized later. (Each cell of res might contain a value less than or equal to $d \times (B-1)^2 < dN$. If that is considered too much we could also normalize after each loop but we don't do it here for the sake of simplicity.) Finally, the integer represented by res is the product of the integers val(a) and val(b) represented by a and b if it is less than B^d and, more generally, it is $(\texttt{val(a)} \times \texttt{val(b)}) \bmod B^d$. □

How does the complexity of the sum, the difference, and the product depend on the size of the operands? Clearly, all the algorithms presented above for addition, subtraction and product run in constant time. This is due to the fact that each instruction involves a fixed number of registers and that it is executed a number of times bounded by the constant d. Of course, more precisely, the execution time of our algorithms depends on the number of digits (= number of registers) in the base $B = \lceil N^{1/2} \rceil$ of their operands, which is at most twice their number of digits in the base N. It is a simple matter to observe that the above addition and subtraction algorithms are executed in time $O(d)$ and that the multiplication algorithm of two numbers of k and m digits, respectively, runs in time $O(k(k+m))$, which is $O(d^2)$, for $k+m=d$. Note that if d was large, one could fall back to any fast multiplication algorithm such as the Fast Fourier Transform.

Remark 8. *Note that our* Sum *(resp.* Diff*) algorithm on "polynomial' integers $a, b < N^d$ (or $a, b < B^d$, for $B = \lceil N^{1/2} \rceil$) working in time $O(d)$, can be seen as an addition (resp. subtraction) algorithm on integers $a, b < N^\ell$ working in time $O(\ell)$, for any non-constant integer ℓ.*

In particular, suppose that a, b are arbitrary integers written in binary with at most n bits, i.e. $a, b < 2^n$; then, splitting their binary representations into subwords of length $\lceil (\log_2 n)/2 \rceil$ (the last subword can be shorter), a, b can be represented in base $\beta := 2^{\lceil (\log_2 n)/2 \rceil}$ by the arrays $\mathtt{a}[0..N-1]$ and $\mathtt{b}[0..N-1]$ of N digits $\mathtt{a}[i], \mathtt{b}[i]$, where $N := \lceil n/\lceil (\log_2 n)/2 \rceil \rceil$, with $\mathtt{a}[i], \mathtt{b}[i] < \beta < N$ (for n large enough). Obviously, our Sum *(resp.* Diff*) algorithm applied to the arrays $\mathtt{a}[0..N-1]$ and $\mathtt{b}[0..N-1]$, with N instead of d, computes their sum array $\mathtt{s}[0..N]$ (resp. difference array $\mathtt{s}[0..N-1]$) in time $O(N) = O(n/\log n)$ under* unit cost criterion*. Note that this is $O(n)$ time (linear time) under* logarithmic cost criterion*.*

5 Running the division in constant time

- *This is the most significant and surprising result of this paper. Although the division is obviously the most difficult of the four standard operations, we will show in this section that it can also be computed in constant time – again after preprocessing in linear time.*
- *The importance of this result is underlined by its consequences:*
  - *the following sections will show that a number of other arithmetic operations on "polynomial" integers – square root, general cth root, exponential, logarithm – can also be computed in constant time, by using the four operations, including, essentially, division;*
  - *as the other arithmetic operations, the division can be used by some "dynamic" problems: as an example, the paper [9] uses a particular case of division to compute in constant time (with linear-time preprocessing) the jth solution of a first-order query on a bounded-degree structure.*

The great novelty of our division algorithm is a very subtle case separation which uses a form of approximation and recursive procedures whose number of calls is bounded by a constant.

It will be convenient to represent the operands of a division sometimes in the base $K := \lceil N^{1/6} \rceil$, sometimes in the base $B := K^3$. (Note that the conversion of an integer from base K to base B, and vice versa, is immediate.) The bounds $N^{1/2} \leq B = O(N^{1/2})$ will be useful.

We say that an integer x is "small" if $x < B$, i.e., if it has only one digit when represented in the base B. First, let us check that this division by a "small" integer can be done in constant time by a variant of the standard division algorithm.

5.1 Dividing a "polynomial" integer by a "small" integer

Here, we want to divide a "polynomial" integer $a := a_{d-1}B^{d-1} + \cdots + a_1 B + a_0$, represented in the base B, by an integer $b < B$. For that we will use Horner's principle. Let us define the sequence $v_d, \ldots, v_1, v_0$ defined inductively by $v_d := 0$ and $v_i := B \times v_{i+1} + a_i$, for $0 \leq i < d$. Note that $v_0 = a$.

For the v_i, we can define q_i and r_i, the quotient and the remainder of dividing v_i by b, for $0 \leq i \leq d$. Note that $v_0 = a$, therefore q_0 is the number we are looking for. Since $v_i = B \times v_{i+1} + a_i$ and $v_{i+1} = q_{i+1} \times b + r_{i+1}$ we deduce

$$v_i = B \times q_{i+1} \times b + B \times r_{i+1} + a_i$$

Let us introduce the respective quotients q', q'' and remainders $r', r'' < b$ of the division of a_i and $B \times r_{i+1}$ by b: we have $a_i = q' \times b + r'$ and $B \times r_{i+1} = q'' \times b + r''$ and therefore

$$v_i = (q_{i+1} \times B + q' + q'') \times b + r' + r''$$

and thus, depending on whether $r' + r'' < b$, we have

- $q_i = q_{i+1} \times B + q' + q''$ and $r_i = r' + r''$ in case $r' + r'' < b$, which gives the equalities (1,2)
  1. $q_i = q_{i+1} \times B + (a_i \texttt{ div } b) + (r_{i+1} \times B) \texttt{ div } b$
  2. $r_i = (a_i \texttt{ mod } b) + (r_{i+1} \times B) \texttt{ mod } b$,

  and, in case $b \leq r' + r'' < 2b$,

- $q_i = q_{i+1} \times B + q'' + q' + 1$ and $r_i = r' + r'' - b$.

The following algorithm uses four arrays $\mathtt{D}[0..B][1..B]$, $\mathtt{R}[0..B][1..B]$, $\mathtt{DM}[0..B][1..B]$, and $\mathtt{RM}[0..B][1..B]$, defined by $\mathtt{D}[a][b] := \lfloor a/b \rfloor$, $\mathtt{R}[a][b] := a \bmod b$, $\mathtt{DM}[a][b] := \lfloor (a \times B)/b \rfloor$, and $\mathtt{RM}[a][b] := (a \times B) \bmod b$.

Algorithm 10 Computation of division $\lfloor \mathtt{a}/\mathtt{b} \rfloor$ by small integer ($\mathtt{a} < \mathtt{B}^d$ and $0 < \mathtt{b} < \mathtt{B}$)

```
1:  procedure LinPreprocDivBySmall()
2:      for b from 1 to B do
3:          D[0][b] ← 0
4:          R[0][b] ← 0
5:          for a from 1 to B do
6:              D[a][b] ← D[a − 1][b]
7:              R[a][b] ← R[a − 1][b] + 1
8:              if R[a][b] = b then
9:                  D[a][b] ← D[a][b] + 1
10:                 R[a][b] ← 0
11:         DM[0][b] ← 0
12:         RM[0][b] ← 0
13:         for a from 1 to B do
14:             DM[a][b] ← DM[a − 1][b] + D[B][b]
15:             RM[a][b] ← RM[a − 1][b] + R[B][b]
16:             if RM[a][b] ≥ b then
17:                 DM[a][b] ← DM[a][b] + 1
18:                 RM[a][b] ← RM[a][b] − b
```

```
1:  procedure DivBySmall(a, b)
2:      q ← 0
3:      r ← 0
4:      for i from d-1 downto 0 do
5:          q ← q×B+D[a[i]][b]+DM[r][b]
6:          r ← R[a[i]][b] + RM[r][b]
7:          if r ≥ b then
8:              q ← q + 1
9:              r ← r − b
10:     return q
```

Justification: Clearly, the LinPreproc procedure correctly computes the arrays D, R, DM and RM in time $O(B^2) = O(N)$. The DivBySmall procedure initializes the pair (q, r) to $(0, 0) = (q_d, r_d)$. Then, for each $i = d-1, \ldots, 1, 0$ of the for loop, it transforms $(q, r) = (q_{i+1}, r_{i+1})$ into $(q, r) = (q_i, r_i)$: indeed, lines (5) and (6) implement exactly the equalities of the above items (1) and (2), respectively, and lines (7-9) implement the case distinction according to whether r $(= r' + r'')$ is less than b or not. Therefore, the return value of q on line (10) is q_0, as expected. Obviously, DivBySmall runs in constant time.

5.2 Two fundamental "recursive" lemmas

Our division algorithm for the general case is based on the following two lemmas, which justify recursive processing.

Item 1 of Lemma 5 below expresses that if the divisor b is large enough then the quotient $\lfloor \lfloor a/K \rfloor / \lceil b/K \rceil \rfloor$ is an approximation within 1 of the desired quotient $\lfloor a/b \rfloor$ while item 2 justifies that this computation can be applied iteratively until the divisor is smaller than K^3, i.e. becomes a "small" integer.

Lemma 5 (first "recursive" lemma)**.** *Assume $N > 1$ so that $K := \lceil N^{1/6} \rceil$ is at least 2. Let a, b be integers such that $b \geq K^3$ and $b > a/K$. Let q be the quotient $\lfloor a'/b' \rfloor$ where $a' := \lfloor a/K \rfloor$ and $b' := \lceil b/K \rceil$.*

1. *We have the following lower and upper bounds: $q \leq a/b < q + 2$. In other words, we have either $\lfloor a/b \rfloor = q$ or $\lfloor a/b \rfloor = q + 1$.*
2. *Moreover, we still have $b' > a'/K$.*

Proof. We have $q = \lfloor \lfloor a/K \rfloor / \lceil b/K \rceil \rfloor \leq \lfloor a/K \rfloor / \lceil b/K \rceil \leq (a/K)/(b/K) = a/b$, hence $q \leq a/b$. There remains to show $a/b < q + 2$. We give a name to the difference between each rounded expression that appears in the expression of q and the value it approximates:

- $\epsilon_1 := a/K - \lfloor a/K \rfloor$; $\epsilon_2 := \lceil b/K \rceil - b/K$; $\epsilon_3 := \lfloor a/K \rfloor / \lceil b/K \rceil - \lfloor \lfloor a/K \rfloor / \lceil b/K \rceil \rfloor$.

Of course, we have $0 \leq \epsilon_i < 1$ for each ϵ_i and, by definition of q and the ϵ_i's,

$$q = (a/K - \epsilon_1)/(b/K + \epsilon_2) - \epsilon_3.$$

It comes $(q + \epsilon_3)(b/K + \epsilon_2) = a/K - \epsilon_1$, and then, by multiplying by K, $qb + \epsilon_3 b + q\epsilon_2 K + \epsilon_3\epsilon_2 K = a - \epsilon_1 K$. Hence, we deduce

$$a - qb = \epsilon_3 b + q\epsilon_2 K + \epsilon_3\epsilon_2 K + \epsilon_1 K.$$

Since each ϵ_i is smaller than 1, we get the inequality

$$a - qb < b + (q+2)K \tag{1}$$

From the hypothesis $b > a/K$ which can be written $a/b < K$ and from the inequality $q \leq a/b$ we deduce $q < K$, and then $q \leq K-1$, hence $(q+2)K \leq K^2 + K$ which is less than K^3 by the hypothesis $K \geq 2$. From the hypothesis $b \geq K^3$, one deduces $(q+2)K < b$ and then by (1) $a - qb < 2b$, which can be rewritten $a/b < q+2$. We have therefore proved item 1 of the lemma.

Let us now prove item 2. From the hypothesis $b > a/K$ we deduce $b' = \lceil b/K \rceil \geq b/K > (a/K)/K \geq \lfloor a/K \rfloor /K = a'/K$ and finally $b' > a'/K$. □

It remains to deal with the case where the condition $b > a/K$ of Lemma 5 is false. The idea is to return to the conditions of application of Lemma 5 and, more precisely, to the computation of a quotient $\lfloor a'/b \rfloor$ for an integer a' such that $b > a'/K$. This is expressed by item 2 of the following lemma.

Lemma 6 (second "recursive" lemma)**.** *Assume $0 < b \leq a/K$. Let $q \coloneqq K\lfloor a/(Kb) \rfloor$.*

1. *Obviously, we have $q \leq a/b$ and $\lfloor a/b \rfloor = q + \lfloor (a - qb)/b \rfloor$.*
2. *We have $a - qb < Kb$ and therefore the computation of $\lfloor a/b \rfloor$ amounts to the computation of $\lfloor a'/b \rfloor$ for $a' = a - qb < Kb$, that means $b > a'/K$.*

Proof. It suffices to prove the inequality $a - qb < Kb$ of item 2. We have $\lfloor a/(Kb) \rfloor > a/(Kb) - 1$, then $q = K\lfloor a/(Kb) \rfloor > a/b - K$, and finally $qb > a - Kb$, which gives the expected inequality. □

5.3 Recursive computation of division in constant time

The "recursive" lemmas justify recursive procedures for computing the quotient $\lfloor a/b \rfloor$, for "polynomial" integers $a, b < K^d$, for a constant integer d.

Here, we assume that the operands a, b are represented in base K. (Recall: $K \coloneqq \lceil N^{1/6} \rceil$ and $B \coloneqq K^3$.) For example, a is represented by the array $\mathtt{a}[0..\mathtt{d}-1]$ of its digits $\mathtt{a}[\mathtt{i}] \coloneqq a_i < K$ with $a = a_{d-1}K^{d-1} + \cdots + a_1 K + a_0$, which is also written $\overline{a_{d-1} \ldots a_1 a_0}$. Note that the quotients $\lfloor a/K \rfloor$ and $\lceil b/K \rceil$ involved in Lemma 5 are computed in a straightforward way: we have $\lfloor a/K \rfloor = a_{d-1}K^{d-2} + \cdots + a_2 K + a_1$, that means the d-digit representation of a/K is the "right shift" $\overline{0 a_{d-1} \ldots a_2 a_1}$; moreover, the following identity can be easily verified:

$$\lceil b/K \rceil = \lfloor (b-1)/K \rfloor + 1 \tag{2}$$

Dividing by an integer $b > a/K$. Lemma 5 completed by identity (2) proves that the following recursive procedure correctly computes $\lfloor a/b \rfloor$ when a and b are "close", which means $b > a/K$.

Algorithm 11 Computation of division for "close" operands

```
1: procedure DivClose(a, b)                          ▷ a/K < b < K^d and a < K^d
2:     if b < K^3 then                                ▷ recall B = K^3
3:         return DivBySmall(a, b)
4:     else                                           ▷ here, b ≥ K^3 and b > a/K : apply Lemma 5
5:         q ← DivClose(DivBySmall(a, K), DivBySmall(b − 1, K) + 1)
6:         if a < (q + 1) × b then
7:             return q
8:         else
9:             return q + 1
```

The general division algorithm. Lemma 6 justifies that the following recursive procedure which uses the previous procedure DivClose correctly computes $\lfloor a/b \rfloor$ for $b > 0$ in the general case.

Algorithm 12 Computation of division

```
1: procedure Divide(a, b)                            ▷ a, b < K^d and b > 0
2:     if b × K > a then
3:         return DivClose(a, b)
4:     else                                           ▷ here, 0 < b ≤ a/K : apply Lemma 6
5:         q ← K × Divide(a, K × b)
6:         return q + DivClose(a − q × b, b)
```

Lemma 7. *The time complexities of the procedures* DIVCLOSE *and* DIVIDE *are respectively* $O(d^3)$ *and* $O(d^4)$.

Proof. The complexity of those procedures mainly depends on the number of recursive calls they make.

Complexity of DIVCLOSE: If $K^3 \leq b < K^d$ and $b > a/K$ then the execution of DIVCLOSE(a, b) calls the same procedure DIVCLOSE with the second argument b replaced by $\lceil b/K \rceil \leq K^{d-1}$, etc. The number of nested recursive calls until the second argument of DIVCLOSE becomes less than K^3 is therefore at most $d - 2$.

The part of the procedure DIVCLOSE that takes the most time is the line (6) of its code which computes the product $(\mathtt{q} + 1) \times \mathtt{b}$ in time $O(d^2)$. Since DIVCLOSE(a, b) executes line (6) once for each recursive call and therefore less than d times, its total time complexity is $O(d^3)$.

Complexity of DIVIDE: The execution of DIVIDE(a, b), for $a, b < K^d$ and $b \leq a/K < K^{d-1}$, calls DIVIDE$(a, b \times K)$, which calls DIVIDE$(a, b \times K^2)$, etc. Due to the obvious inequality $b \times K^d > a$, the execution of DIVIDE(a, b) stops with the stack of recursive calls to the procedure DIVIDE of height at most d.

The most time consuming part of the procedure DIVIDE is the line (5) of its code which is executed as many times as the number of the nested recursive calls of the form DIVIDE$(\mathtt{a}, K \times \mathtt{b})$ that it makes. This number is at most d. It is also the number of calls of the form DIVCLOSE$(a - q \times b, b)$, line (6), each of which is executed in time $O(d^3)$. Thus, the time of the procedure DIVIDE is $O(d^4)$. □

As a consequence of the previous results including Lemma 7, the following perhaps surprising theorem is established.

Theorem 2. *Let $d \geq 1$ be a fixed integer. There is a RAM with addition, such that, for any input integer N:*

1. Pre-computation: *the RAM computes some tables in time $O(N)$;*
2. Division: *using these pre-computed tables and reading any integers $a, b < N^d$ with $b > 0$, the RAM computes in constant time $O(d^4)$ the quotient $\lfloor a/b \rfloor$ and the remainder a $\mathtt{mod}$ b.*

Using the fact that the usual four operations, addition, subtraction, product and, most importantly, division, are computable in constant time on a RAM with addition, we can now also design constant time algorithms for several other important operations on integers: square or cubic root (rounded), exponential function and rounded logarithm, etc.

Convention. *For simplicity, we will henceforth write x/y to denote the Euclidean quotient* DIVIDE$(x, y) = \lfloor x/y \rfloor$ *in the algorithms.*

6 Computing the exponential and the logarithm in constant time

Our next examples of arithmetic operations (on "polynomial" operands) we will prove to be computable in constant time are the logarithm function $(x, y) \mapsto \lfloor \log_x y \rfloor$ and the exponential function $(x, y) \mapsto x^y$ provided the result x^y is also "polynomial".

6.1 Computing exponential

We want to compute $z = x^y$, for all integers $x, y < N^d$ such that the result z is less than N^d, for a constant integer $d \geq 1$. In other words, we are going to compute the function $\mathtt{exp}_{N,d}$ from $[0, N^d[^2$ to $[0, N^d[\cup \{\mathtt{OVERFLOW}\}$ defined by:

$$\mathtt{exp}_{N,d} : (x, y) \mapsto \begin{cases} x^y & \mathtt{if}\ x^y < N^d \\ \mathtt{OVERFLOW} & \mathtt{otherwise} \end{cases}$$

Here, we will use the two integers $B := \lceil N^{1/2} \rceil$ and $L_d := \lceil \log_2(N^d) \rceil = \lceil d \log_2 N \rceil$ computable in time $O(N)$.

Pre-computation. We are going to compute two arrays:

- the array $\texttt{BOUND}[2..B-1]$ such that $\texttt{BOUND}[x] := \max\{y \mid x^y < N^d\}$ or, equivalently, $\texttt{BOUND}[x] := \lceil \log_x(N^d) \rceil - 1$ (note that $\texttt{BOUND}[x] < L_d$);
- the two-dimensional array $\texttt{EXP}[2..B-1][1..L_d-1]$ such that $\texttt{EXP}[x][y] := x^y$ for all integers x, y such that $2 \leq x < B = \lceil N^{1/2} \rceil$ and $1 \leq y \leq \texttt{BOUND}[x]$.

In other words, the arrays BOUND and EXP are defined so that the following equality is true:

$$\{(x,y,z) \in [2,B[\times[0,N^d[^2 \mid \ x^y = z\} = \{(x,y,z) \mid x \in [2,B[\wedge 1 \leq y \leq \texttt{BOUND}[x] \wedge \texttt{EXP}[x][y] = z\}.$$

The following code computes the arrays BOUND and EXP.

Algorithm 13 Pre-computation for EXPONENTIAL

```
1: for x from 2 to B − 1 do
2:     y ← 0
3:     z ← x
4:     while z < N^d do
5:         y ← y + 1
6:         EXP[x][y] ← z
7:         z ← z × x
8:     BOUND[x] ← y
```

Justification: One easily verifies inductively that each time the algorithm arrives at line 4, we have $x^y \times x = z$, so that the assignment at line 6 means $\texttt{EXP}[x][y] \leftarrow x^y$. This also explains why we have $x^y < N^d \leq x^{y+1}$ at line 8, which justifies the assignment $\texttt{BOUND}[x] \leftarrow y$.

Linear-time complexity of the pre-computation: Clearly, for each $x \in [2,B[$, the body of the while loop is repeated i times, where i is the greatest integer $i \leq \log_x(N^d)$, or, equivalently, $i = \lfloor d \log_x N \rfloor$, which is at most $\lfloor d \log_2 N \rfloor \leq L_d$. It follows that the pre-computation runs in time and space $O(B \times L_d) = O(N^{1/2} \times d \log_2 N) = O(N)$.

Main procedure: The following remark justifies that the EXPONENTIAL procedure given here below correctly computes the function $\texttt{exp}_{N,d}$.

Remark 9. *If we have $x^y < N^d$, $x \geq 2$ and $y \geq 2d$ then we also have $2 \leq x < N^{1/2}$ and $2d \leq y < d \log_2 N \leq L_d$.*

Proof. Note that $x^y < N^d$ and $y \geq 2d$ imply $x^{2d} \leq x^y < N^d$ and consequently $x < N^{1/2}$. Also, note that $x^y < N^d$ and $x \geq 2$ imply $2^y \leq x^y < N^d$ and therefore $y < \log_2(N^d)$. □

Algorithm 14 Computation of EXPONENTIAL

```
1: procedure EXPONENTIAL(x, y)
2:     if x = 1 or y = 0 then
3:         return 1
4:     if x = 0 then
5:         return 0
6:     if y < 2d then                                          ▷ x ≥ 2
7:         z ← 1
8:         for i from 1 to y do
9:             z ← x × z                                       ▷ z = x^i
10:            if z ≥ N^d then
11:                return OVERFLOW                             ▷ x^y ≥ x^i ≥ N^d
12:        return z                                            ▷ z = x^y
13:    if x < B and y ≤ BOUND[x] then
14:        return EXP[x][y]          ▷ the correct value by the definition of the array EXP
15:    else          ▷ either y ≥ 2d and x ≥ B, or y > BOUND[x] for x ≥ 2; therefore x^y ≥ N^d
16:        return OVERFLOW
```

Constant complexity of the exponential procedure: Clearly, the most time consuming part is the for loop, lines 8-11. It makes at most $2d$ multiplications of integers $x, z < N^d$, each in time $O(d^2)$. Therefore, the procedure runs in time $O(d^3)$.

6.2 Computing logarithm

We want to compute $\lfloor \log_x y \rfloor$, for all integers $x \in [2, N^d[$ and $y \in [1, N^d[$, where $d \geq 1$ is a constant integer.

Here again, we use the base $B \coloneqq \lceil N^{1/2} \rceil$.

Preliminary lemmas

Lemma 8. *For any integer $x \geq 2$ and any real $a \geq 2$, we have $\log_x(a) - 1 < \log_x \lfloor a \rfloor \leq \log_x(a)$.*

Proof. We have $a/x \leq a/2 \leq a - 1$ and therefore $\log_x(a) - 1 = \log_x(a/x) \leq \log_x(a-1) < \log_x \lfloor a \rfloor \leq \log_x(a)$. □

Here again, our main procedure will be recursive. It is based on the following Lemma.

Lemma 9. *For all integers $x \geq 2$, $\ell \geq 1$ and $y \geq 2x^\ell$, we have*

$$\lfloor \log_x y \rfloor = \begin{cases} \lfloor \log_x \lfloor y/x^\ell \rfloor \rfloor + \ell & (1) \\ \texttt{or} & \\ \lfloor \log_x \lfloor y/x^\ell \rfloor \rfloor + \ell + 1 & (2) \end{cases}$$

Proof. Lemma 8 applied to $a = y/x^\ell \geq 2$ gives $\log_x(y/x^\ell) - 1 < \log_x \lfloor y/x^\ell \rfloor \leq \log_x(y/x^\ell)$. Adding ℓ to each member of those inequalities yields

$$\log_x(y) - 1 < \log_x \lfloor y/x^\ell \rfloor + \ell \leq \log_x(y).$$

This can be rewritten: $\log_x \lfloor y/x^\ell \rfloor + \ell \leq \log_x(y) < \log_x \lfloor y/x^\ell \rfloor + \ell + 1$. Taking the integer parts, we obtain

$$\lfloor \log_x \lfloor y/x^\ell \rfloor \rfloor + \ell \leq \lfloor \log_x(y) \rfloor \leq \lfloor \log_x \lfloor y/x^\ell \rfloor \rfloor + \ell + 1.$$

□

How to determine which case (1) or (2) of Lemma 9 is true? The following lemma completes Lemma 9 by giving a simple criterion.

Lemma 10. *Let $s \coloneqq \lfloor \log_x \lfloor y/x^\ell \rfloor \rfloor + \ell$ for integers $x \geq 2$, $\ell \geq 1$ and $y \geq 2x^\ell$. Lemma 9 says (1) $\lfloor \log_x y \rfloor = s$ or (2) $\lfloor \log_x y \rfloor = s+1$. If $x^{s+1} > y$ then $\lfloor \log_x y \rfloor = s$. Otherwise, $\lfloor \log_x y \rfloor = s+1$.*

Proof. (1) means $x^s \leq y < x^{s+1}$ and (2) means $x^{s+1} \leq y < x^{s+2}$. So, $x^{s+1} > y$ implies (1), and $x^{s+1} \leq y$ implies (2). □

Principle of the algorithm and pre-computation in linear time. For each $x \in [2, B]$, let us define $\texttt{L}[x] \coloneqq \lceil \log_x B \rceil = \texttt{min}\{z \in \mathbb{N} \mid x^z \geq B\}$. $x \leq B$ implies $\texttt{L}[x] \geq 1$. Clearly, the array $\texttt{L}[2..B]$ can be pre-computed in time $O(B \log_2 B) = O(N)$ by the following algorithm where an invariant of the while loop is $y = x^z$.

Algorithm 15 Computation of the array L

```
1: for x from 2 to B do
2:     y ← x
3:     z ← 1
4:     while y < B do                          ▷ y = x^z < B
5:         y ← x × y
6:         z ← z + 1                           ▷ x^{z-1} < B ≤ y = x^z
7:     L[x] ← z
```

The algorithm that computes $\lfloor \log_x y \rfloor$, for integers $x \in [2, N^d[$ and $y \in [1, N^d[$, divides into the following four cases:

1. $x > y$: then $\lfloor \log_x y \rfloor = 0$;
2. $x \leq y < 2B$: then $\lfloor \log_x y \rfloor$ will be computed by a simple look up in a pre-computed array $\texttt{LOGAR}[2..2B-1][2..2B-1]$ defined by $\texttt{LOGAR}[x][y] \coloneqq \lfloor \log_x y \rfloor$, for $2 \leq x \leq y < 2B$;
3. $x \leq y$ and $2B \leq y < 2x^{L[x]}$: then we have $x \leq y < 2x^{L[x]} < 2x^{1+\log_x B} = 2xB$; this implies $1 \leq \lfloor y/x \rfloor < 2B$, and because of the identity[9] $\lfloor \log_x y \rfloor = \lfloor \log_x \lfloor y/x \rfloor \rfloor + 1$, the value of $\lfloor \log_x y \rfloor$ is obtained by computing $\lfloor \log_x \lfloor y/x \rfloor \rfloor$ according to cases 1 or 2;

[9] Indeed, $z = \lfloor \log_x y \rfloor$ means $x^z \leq y < x^{z+1}$, which implies $x^{z-1} \leq \lfloor y/x \rfloor < x^z$ and therefore $z - 1 \leq \lfloor \log_x \lfloor y/x \rfloor \rfloor < z$. That means $z = \lfloor \log_x \lfloor y/x \rfloor \rfloor + 1$.

4. $y \geq 2x^{L[x]}$: then apply Lemma 10 recursively with $\ell := L[x]$ (noting that the required assumptions $x \geq 2$, $\ell \geq 1$ and $y \geq 2x^{\ell}$ are satisfied).

We claim that the following code computes the array LOGAR:

Algorithm 16 Computation of the array LOGAR

```
1:  for x from 2 to 2 × B − 1 do
2:      y ← x
3:      z ← 1
4:      t ← x × x
5:      while y < 2 × B do
6:          while y < t do
7:              LOGAR[x][y] ← z
8:              y ← y + 1
9:          z ← z + 1
10:         t ← x × t
```

Indeed, it can be checked inductively that each time the execution of the algorithm reaches line 7 then we have $x^z \leq y < x^{z+1} = t$, which justifies the assignment $\texttt{LOGAR[x][y]} \leftarrow \texttt{z}$. It is also easy to verify that the algorithm runs in time $O(B^2) = O(N)$.

The main algorithm. We assert that the following procedure LOGARITHM correctly computes $\lfloor \log_x y \rfloor$, for all the integers $x \in [2, N^d[$ and $y \in [1, N^d[$.

Algorithm 17 Computation of LOGARITHM

```
1:  procedure LOGARITHM(x,y)
2:      if y < x then                                              ▷ case 1
3:          return 0
4:      if y < 2 × B then                                          ▷ case 2
5:          return LOGAR[x][y]
6:      if y < 2 × EXPONENTIAL(x, L[x]) then                       ▷ case 3: 1 ≤ y/x < 2B
7:          return LOGARITHM(x, y/x) + 1
8:      else                          ▷ case 4 (y ≥ 2x^L[x]): apply Lemma 10 with ℓ = L[x]
9:          s ← LOGARITHM(x, y/EXPONENTIAL(x, L[x])) + L[x]
10:         if EXPONENTIAL(x, s + 1) > y then
11:             return s
12:         else
13:             return s + 1
```

Justification: Obviously, the LOGARITHM procedure is correct for $y < x$ (case 1: lines 2-3) and for $x \leq y < 2B$ (case 2: lines 4-5). Lines 6-7 treats the case 3, $x \leq y < 2x^{L[x]}$ (recall $L[x] = \lceil \log_x B \rceil$), which implies $1 \leq y/x < 2B$: we therefore return to cases 1 or 2 by taking y/x instead of y. This justifies line 7. The case $y \geq 2x^{L[x]}$ (case 4: lines 8-13) is justified by Lemma 10 by taking $\ell := L[x]$ and noting that the required assumptions $x \geq 2$, $\ell \geq 1$ and $y \geq 2x^{\ell}$ are satisfied.

Constant time and space complexity of the logarithm procedure. Let us first note that all the numbers (operands, intermediate values, results) manipulated by the procedure are less than N^{2d} and that all the arithmetic operations it uses can be computed in constant time and space. In particular, this holds for the "exponential" expressions $2x^{\mathtt{L}[x]} < 2xB \leq N^{d+1}$ and $x^{s+1} \leq xy < N^{2d}$, which involve the function $\texttt{exp}_{N,2d}$ of the previous subsection, computed by the EXPONENTIAL procedure in constant time and space. Therefore, it suffices to prove that the LOGARITHM procedure performs at most a constant number of recursive calls. We have $x^{\mathtt{L}[x]} \geq B$, by definition of $\mathtt{L}[x]$. Therefore, for $y < N^d \leq B^{2d}$, the algorithm cannot divide y by $x^{\mathtt{L}[x]} \geq B$ more than $2d-1$ times. This implies that it makes at most $2d-1$ recursive calls (by case 4) before reaching a value of y less than x (basic case 1), or less than $2B$ (basic case 2), or less than $2x^{L[x]}$ (case 3, which performs one recursive call leading to case 1 or case 2). Thus, for a fixed d, the LOGARITHM procedure runs in constant time and space.

Open problem: computing the modular exponentiation. Obviously, the functions $(x, y, z) \mapsto x + y \mod z$, and $(x, y, z) \mapsto x \times y \mod z$, for $x, y < N^d$ and $1 \leq z \leq N^d$, are

computable in constant time. Thus, a natural candidate for being computable (or not) in constant time, is the following version of the exponential function, widely used in cryptography: the function ExpMod from $[0, N^d[^2 \times [1, N^d]$ to $[0, N^d[$ defined, for $x, y < N^d$ and $1 \leq z \leq N^d$, by

$$\textsc{ExpMod}(x, y, z) \coloneqq x^y \mod z.$$

Now, we cannot prove that $\textsc{ExpMod}(x, y, z)$ is computable in constant time, even when fixing $x = c$ or $z = N^d$, for constant integers $c > 1$, $d \geq 1$. Indeed, it seems likely that even the following functions are not computable in constant time after a linear preprocessing, for $x, y, z < N$:

$$(y, z) \mapsto 2^y \mod z \qquad (x, y) \mapsto x^y \mod N^2 \qquad \text{or even} \quad (x, y) \mapsto x^y \mod N.$$

Intuitively, the irregularity of the modulo function applied to a large number - an exponential - prevents us from designing an induction by approximation as we have done for division and logarithm and as we will for the square root and the other roots in the next section.

7 Computing the square root and other roots in constant time

- *It is easy to compute in linear time a table giving the square roots or the c-th roots $\lfloor x^{1/c} \rfloor$, for a fixed integer $c \geq 2$, of the integers $x < N$.*
- *In this section, we will show that, for a fixed integer $c \geq 2$, the c-th root of a "polynomial" integer can be computed in constant time.*
- *Our algorithm will use a discrete adaptation of Newton's approximation method.*

7.1 The cthRoot algorithm

Let $c \geq 2$ be a fixed integer. Let us explain how to compute the c-th root of an integer $x < N^d$, for a constant integer d, that is the function $x \mapsto \lfloor x^{1/c} \rfloor$ on the domain $[0, N^d[$.

Similarly to the division method, our algorithm will distinguish "small values" for which the c-th root will be precomputed and "large values" for which we will compute an approximation (using recursively the cthRoot function with smaller values) before improving this approximation.

The preprocessing here is similar to that of the division: we have a bound $\mathtt{M}'$ and we populate an array `cthRoot` such that $\mathtt{cthRoot}[\mathtt{x}] = \lfloor \mathtt{x}^{1/c} \rfloor$ for all $0 \leq \mathtt{x} < \mathtt{M}'$. To compute the c-th root for an $\mathtt{x}$ below this $\mathtt{M}'$ limit, we return its precomputed value. For an $\mathtt{x}$ over this $\mathtt{M}'$ limit, we first compute $\mathtt{t}$ which is the floored c-th root of the Euclidean quotient $\lfloor \mathtt{x}/\mathtt{K}^c \rfloor$. Note that the flooring happens twice: we floor the quotient $\mathtt{x}/\mathtt{K}^c$ and we floor the c-th root of $\mathtt{x}/\mathtt{K}^c$. Overall, we only know that the value $\mathtt{x}^{1/c}$ is in the range $[\mathtt{t} \times \mathtt{K}, (\mathtt{t}+1) \times \mathtt{K}[$. As we will show below, the following two points are essential:

- by a careful choice of $\mathtt{K}$ and $\mathtt{M}'$, this is a sufficiently good approximation and it suffices to update this approximation by using two steps of Newton's method and one step of checking whether the answer is $\mathtt{g}$ or $\mathtt{g}-1$ where $\mathtt{g}$ is our current approximation;
- the values for $\mathtt{K}$ and $\mathtt{M}'$ can also be chosen such that $\mathtt{M}' = O(N)$, which allows a linear preprocessing, and $\mathtt{K} = \Theta(N^{1/(2c)})$, which ensures that the procedure executes in constant time.

We know present the algorithm whose correctness is justified in the next paragraphs.

Algorithm 18 Computation of $x \mapsto \lfloor x^{1/c} \rfloor$ using the pre-computed constants $\mathtt{M}'$ and $\mathtt{K}$

```
1: procedure LinPreprocCthRoot()
2:     cthRoot ← an array of size M'
3:     cthRoot[0] ← 0
4:     for x from 1 to M' − 1 do
5:         s ← cthRoot[x − 1]
6:         if x < (s + 1)^c then
7:             cthRoot[x] ← s
8:         else
9:             cthRoot[x] ← s + 1

1: procedure improve(g, x)
2:     if g^c ≤ x < (g + 1)^c then
3:         return g
4:     else
5:         return (x+(c−1)×g^c)/(c×g^{c−1})

1: procedure cthRoot(x)
2:     if x < M' then
3:         return cthROOT[x]
4:     else
5:         s ← x/K^c
6:         t ← cthRoot(s)
7:         g_0 ← t × K
8:         g_1 ← improve(g_0, x)
9:         g_2 ← improve(g_1, x)
10:        if g_2^c ≤ x < (g_2 + 1)^c then
11:            return g_2
12:        else
13:            return g_2 − 1
```

Setting the integers $\mathtt{K}$ and $\mathtt{M}'$: Here, we will simply use $\mathtt{K} \coloneqq \lceil N^{1/(2c)} \rceil$, which is the integer $\mathtt{K}$ such that $(\mathtt{K}-1)^{2c} < N \leq \mathtt{K}^{2c}$, and let $\mathtt{M} \coloneqq \mathtt{K}^{2c}$. (Note that $\mathtt{K}$ and $\mathtt{M}$ can be easily computed in time $O(N)$.) The value of $\mathtt{M}'$ will be set as $\mathtt{M}' \coloneqq \mathtt{max}(\mathtt{M}, c_0^{2c} + c_0^c)$, for c_0 a constant specified in the next paragraph, which only depends on c. Note that, for any given c, we have $\mathtt{M} = \mathtt{M}'$ for any N large enough, which justifies the linearity of the preprocessing, and we have $\mathtt{K}^c \geq N^{1/2}$, which, considering lines 5-6 of the cthRoot procedure, will justify that the recursion depth is at most $2d$ for an integer $x < N^d$ (and thus the cthRoot procedure is constant time for any fixed d).

The mysterious constant c_0: We define the constant integers $c_0 \coloneqq \left\lceil \left\lfloor \sqrt{6c-12} \right\rfloor \times (c^2-1)/(6c) \right\rceil$ and $c_0^{2c} + c_0^c$. This allows us to define $M' \coloneqq \mathtt{max}(M, c_0^{2c} + c_0^c)$. The role of the strange constants c_0 and $c_0^{2c} + c_0^c$ will be elucidated at the end of the proof of the algorithm. Note that for $c = 2$, we have $c_0 = 0$ and therefore $M' = M$, and for $c = 3$, we have $c_0 = 2$ and $c_0^{2c} + c_0^c = 72$. More generally, for any fixed c, the values c_0 and $c_0^{2c} + c_0^c$ are explicit integers.

7.2 Correctness and complexity of the cthRoot procedure

First, note the inequality, for all $a \geq 0$ and $c \geq 1$:

$$(a+1)^c + 1 \leq \left(a + \frac{c+1}{c}\right)^c \tag{3}$$

It follows from the (in)equalities $\left(a + \frac{c+1}{c}\right)^c = \left((a+1) + \frac{1}{c}\right)^c \geq (a+1)^c + c(a+1)^{c-1} \times \frac{1}{c} = (a+1)^c + (a+1)^{c-1} \geq (a+1)^c + 1$.

When $x < M'$, the cthRoot procedure returns $\mathtt{cthROOT}[x]$, which is the correct value. Otherwise, we have $x \geq K^{2c}$ and $x \geq c_0^{2c}$. The integers s and t will be such that $t^c \leq s < (t+1)^c$. Because of $t^c \leq s \leq x/K^c < s+1$ and $g_0 = tK$ and inequality (3) for $a = t$, we obtain: $g_0^c = t^c K^c \leq x < (s+1)K^c < ((t+1)^c + 1)K^c \leq \left(t + \frac{c+1}{c}\right)^c K^c = \left(tK + \frac{c+1}{c}K\right)^c$ and then $g_0^c \leq x < \left(g_0 + \frac{c+1}{c}K\right)^c$. Hence, $g_0 \leq x^{1/c} < g_0 + \frac{c+1}{c}K$. Let ϵ_0 denote the real number such that

$$x^{1/c} = g_0 \times (1 + \epsilon_0). \tag{4}$$

We have $0 \leq \epsilon_0 < \frac{K(c+1)}{cg_0} = \frac{c+1}{ct}$.

Starting from the approximate value g_0 of $x^{1/c}$, let us apply Newton's method: let g_1 be the abscissa of the point of ordinate 0 of the tangent to the point $(g_0, f(g_0))$ of the function $X \mapsto f(X) = X^c - x$. This means that g_1 is the number satisfying the equation:

$$f'(g_0) = \frac{0 - f(g_0)}{g_1 - g_0} \tag{5}$$

This gives $cg_0^{c-1} = \dfrac{x - g_0^c}{g_1 - g_0}$, and (multiplying by $g_1 - g_0$), $cg_1g_0^{c-1} - cg_0^c = x - g_0^c$, and therefore $cg_1g_0^{c-1} = (c-1)g_0^c + x$, and finally

$$g_1 = \frac{1}{c}\left((c-1)g_0 + \frac{x}{g_0^{c-1}}\right) \tag{6}$$

From (6) and (4) we get $g_1 = \dfrac{(c-1)g_0 + (1+\epsilon_0)^c \times g_0}{c} = g_0(1+\epsilon_0) \times \dfrac{c + c\epsilon_0 + (1+\epsilon_0)^c - 1 - c\epsilon_0}{c + c\epsilon_0} \leq x^{1/c} \times \left(1 + \dfrac{\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^i}{c}\right) = x^{1/c} \times \left(1 + \epsilon_0^2 \times \dfrac{\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^{i-2}}{c}\right)$. We deduce

$$x^{1/c} \leq g_1 \leq x^{1/c} \times \left(1 + \epsilon_0^2/c \times \left(\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^{i-2}\right)\right). \tag{7}$$

Let us compare two consecutive summands S_i, S_{i+1} of the sum $\sum_{i=2}^{c} S_i$ where $S_i := \binom{c}{i}\epsilon_0^{i-2}$. For $2 \leq i < c$, we have $S_{i+1} = S_i \times \dfrac{(c-i)\epsilon_0}{i+1} \leq S_i \times \dfrac{(c-2)\epsilon_0}{3}$, from which we deduce $S_{i+1}/S_i \leq \dfrac{(c-2)\epsilon_0}{3} \leq \dfrac{(c-2)(c+1)}{3ct}$ because of $\epsilon_0 \leq \dfrac{c+1}{ct}$. The quotient S_{i+1}/S_i is at most 1 if we assume

$$t \geq c_1 \text{ for the constant } c_1 := \frac{(c-2)(c+1)}{3c}.$$

Under this assumption, each of the $c-1$ summands $S_i = \binom{c}{i}\epsilon_0^{i-2}$, $2 \leq i \leq c$, is less than or equal to $S_2 = \binom{c}{2} = \frac{c(c-1)}{2}$, so that we obtain $\dfrac{\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^{i-2}}{c} \leq \dfrac{1}{c} \times (c-1) \times \dfrac{c(c-1)}{2} = \dfrac{(c-1)^2}{2}$ and, as another consequence of $\epsilon_0 \leq \dfrac{c+1}{ct}$,

$$\epsilon_0^2 \times \frac{\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^{i-2}}{c} \leq \frac{(c^2-1)^2}{2c^2t^2}. \tag{8}$$

Note that in the CTHROOT procedure, the variable $\mathtt{g}_1$ represents the integer part $\lfloor g_1 \rfloor$ of g_1. We know that $x^{1/c} \leq g_1$ but it is possible that $\lfloor g_1 \rfloor \leq x^{1/c}$. In this case, we have $\lfloor g_1 \rfloor \leq x^{1/c} \leq g_1$, which implies $\mathtt{g}_1 = \lfloor g_1 \rfloor = \lfloor x^{1/c} \rfloor$, that means $\mathtt{g}_1$ is the value we are looking for. Otherwise, we have $\lfloor g_1 \rfloor > x^{1/c}$ and we denote ϵ_1 the positive number such that $\lfloor g_1 \rfloor = x^{1/c} \times (1+\epsilon_1)$. From the inequalities $x^{1/c} \leq x^{1/c}(1+\epsilon_1) \leq g_1 \leq x^{1/c} \times \left(1 + \epsilon_0^2/c \times \left(\sum_{i=2}^{c}\binom{c}{i}\epsilon_0^{i-2}\right)\right)$ and (8), we deduce

$$0 \leq \epsilon_1 \leq \frac{(c^2-1)^2}{2c^2t^2} \tag{9}$$

When $\mathtt{g}_1 = \lfloor g_1 \rfloor > \lfloor x^{1/c} \rfloor$ we iterate Newton's method by using the approximate value $\lfloor g_1 \rfloor$ and, by replacing g_0 by $\lfloor g_1 \rfloor$ in expression (6), we obtain the integer part $\mathtt{g}_2 = \lfloor g_2 \rfloor$ of the number

$$g_2 := \frac{1}{c} \times \left((c-1)\lfloor g_1 \rfloor + \frac{x}{\lfloor g_1 \rfloor^{c-1}}\right).$$

Then, we have $g_2 = \dfrac{1}{c} \times \left((c-1)x^{1/c}(1+\epsilon_1) + \dfrac{x}{x^{(c-1)/c} \times (1+\epsilon_1)^{c-1}}\right) = \dfrac{x^{1/c}}{c} \times \dfrac{(c-1)(1+\epsilon_1)^c + 1}{(1+\epsilon_1)^{c-1}} = x^{1/c} \times \dfrac{(c-1)(1 + c\epsilon_1 + \sum_{i=2}^{c}\binom{c}{i}\epsilon_1^i) + 1}{c(1+\epsilon_1)^{c-1}} = x^{1/c} \times \dfrac{c + c(c-1)\epsilon_1 + (c-1)(\sum_{i=2}^{c}\binom{c}{i}\epsilon_1^i)}{c + c(c-1)\epsilon_1 + c(\sum_{i=2}^{c-1}\binom{c-1}{i}\epsilon_1^i)} \leq x^{1/c} \times \left(1 + \dfrac{c-1}{c} \times \left(\sum_{i=2}^{c}\binom{c}{i}\epsilon_1^{i-2}\right) \times \epsilon_1^2\right)$. Finally, we obtain

$$x^{1/c} \leq g_2 \leq x^{1/c} \times \left(1 + \frac{c-1}{c} \times \left(\sum_{i=2}^{c}\binom{c}{i}\epsilon_1^{i-2}\right) \times \epsilon_1^2\right) \tag{10}$$

With the same argument as above, the quotient between two consecutive summands of the sum $\sum_{i=2}^{c}\binom{c}{i}\epsilon_1^{i-2}$ is not greater than $\dfrac{(c-2)\epsilon_1}{3}$, which is at most $\dfrac{(c-2)(c^2-1)^2}{6c^2t^2}$ because of (9). This quotient is not greater than 1 if we assume $t^2 \geq \dfrac{(c-2)(c^2-1)^2}{6c^2}$, that means

$t \geq \dfrac{\sqrt{6c-12} \times (c^2-1)}{6c}$ or, if we assume, more strongly, $t \geq c_0$, by definition of c_0.

Therefore, if we assume $t \geq c_0$, then each of the $c-1$ summands $\binom{c}{i}\epsilon_1^{i-2}$, $2 \leq i \leq c$, is less than or equal to the first summand $\binom{c}{2} = \dfrac{c(c-1)}{2}$ so that one deduces the inequality

$$\frac{c-1}{c} \times \left(\sum_{i=2}^{c} \binom{c}{i} \epsilon_1^{i-2} \right) \leq \frac{c-1}{c} \times (c-1) \times \frac{c(c-1)}{2} = \frac{(c-1)^3}{2}$$

and, as another consequence of (9),

$$\frac{c-1}{c} \times \left(\sum_{i=2}^{c} \binom{c}{i} \epsilon_1^{i-2} \right) \times \epsilon_1^2 \leq \frac{(c-1)^3}{2} \times \frac{(c^2-1)^4}{4c^4t^4} = \frac{(c-1)^7(c+1)^4}{8c^4t^4}. \tag{11}$$

From the inequalities (10) and (11), we deduce

$$x^{1/c} \leq g_2 \leq x^{1/c} \times \left(1 + \frac{c_2}{t^4}\right) \text{ where } c_2 \coloneqq \frac{(c-1)^7(c+1)^4}{8c^4}.$$

Remark 10. *Recall that those inequalities hold* under the hypotheses $t \geq c_1 = (c-2)(c+1)/(3c)$ *and* $t \geq c_0$. *Also, note that the hypothesis* $t \geq c_0$ *is sufficient since we have* $c_0 > c_1$ *for* $c \geq 2$, *which is easy but tedious to prove.*

Proof of $x^{1/c} \leq g_2 < x^{1/c} + 1$ (under the hypothesis $t \geq c_0$). We have $x^{1/c} \leq g_2 \leq x^{1/c} + \epsilon_3$ for $\epsilon_3 \coloneqq \dfrac{c_2 x^{1/c}}{t^4}$. Besides, we have $t = \left\lfloor \left\lfloor \dfrac{x}{K^c} \right\rfloor^{1/c} \right\rfloor > \left(\dfrac{x}{K^c}\right)^{1/c} - 1 = \dfrac{x^{1/c} - K}{K}$ (because we have $\left\lfloor \lfloor a \rfloor^{1/c} \right\rfloor = \lfloor a^{1/c} \rfloor > a^{1/c} - 1$, for each positive real number a). We deduce $\epsilon_3 = \dfrac{c_2 x^{1/c}}{t^4} \leq \dfrac{c_2 K^4 x^{1/c}}{(x^{1/c}-K)^4} = c_2 \times \dfrac{K^4((x^{1/c}-K)+K)}{(x^{1/c}-K)^4} = c_2 \times \left(\dfrac{K^4}{(x^{1/c}-K)^3} + \dfrac{K^5}{(x^{1/c}-K)^4}\right)$. Recall the hypothesis $x \geq K^{2c}$ for $K = \lceil N^{1/(2c)} \rceil$. This implies $x^{1/c} - K \geq K^2 - K$ and $\epsilon_3 \leq c_2 \times \left(\dfrac{K^4}{(K^2-K)^3} + \dfrac{K^5}{(K^2-K)^4}\right) = c_2 \times \left(\dfrac{K}{(K-1)^3} + \dfrac{K}{(K-1)^4}\right) = c_2 \times \dfrac{K^2}{(K-1)^4}$. We obtain $\epsilon_3 \leq c_2 \times \dfrac{K^2}{(K-1)^4}$.

For $K \geq 1+2\sqrt{c_2}$ [10], we get $\dfrac{K^2}{(K-1)^4} = \dfrac{1}{(K-1)^2} + \dfrac{2}{(K-1)^3} + \dfrac{1}{(K-1)^4} \leq \dfrac{1}{4c_2} + \dfrac{2}{4c_2} + \dfrac{1}{4c_2} = \dfrac{1}{c_2}$, which implies $\epsilon_3 \leq c_2 \times \dfrac{K^2}{(K-1)^4} \leq 1$. Therefore, we obtain $x^{1/c} \leq g_2 \leq x^{1/c} + 1$, or, equivalently, $g_2 - 1 \leq x^{1/c} \leq g_2$.

This implies that we have either $\lfloor x^{1/c} \rfloor = \lfloor g_2 \rfloor = \mathtt{g}_2$ or $\lfloor x^{1/c} \rfloor = \lfloor g_2 \rfloor - 1 = \mathtt{g}_2 - 1$. This justifies the last line of the CTHROOT procedure and completes the proof of its correctness *if* we can prove that the hypothesis $t \geq c_0$ *always* holds when $x \geq M'$.

Why $x \geq M'$ implies $t \geq c_0$. Let us prove this implication by contradiction. Assume we have both $x \geq M'$ and $t < c_0$. We have $x \geq M = K^{2c}$ and $x \geq c_0^{2c} + c_0^c$.

Recall that $s = \lfloor x/K^c \rfloor$ and $t = \left\lfloor s^{1/c} \right\rfloor$ imply $x/K^c < s+1$ and $s^{1/c} < t+1$, which means $s < (t+1)^c$. With the hypothesis $x \geq M = K^{2c}$, this yields $K^c \leq x/K^c < s+1 < (t+1)^c + 1$ and therefore $K^c \leq (t+1)^c$. Finally, using the hypothesis $t < c_0$, we get $K \leq t+1 < c_0 + 1$ and $K \leq c_0$.

Besides, from the hypothesis $x \geq c_0^{2c} + c_0^c$ and from $K \leq c_0$, we deduce, for $s = \lfloor x/K^c \rfloor$, the inequalities: $s > x/K^c - 1 \geq (c_0^{2c} + c_0^c)/K^c - 1 \geq (c_0^{2c} + c_0^c)/c_0^c - 1 = c_0^c$, hence $s > c_0^c$ and $t = \lfloor s^{1/c} \rfloor \geq \lfloor (c_0^c)^{1/c} \rfloor = c_0$. Therefore, we get $t \geq c_0$, a contradiction.

The proof of the correctness of the CTHROOT procedure is now complete. □

[10] That means $M = K^{2c} \geq \left(1 + (c-1)^{7/2} \times (c+1)^2/(\sqrt{2} \times c^2)\right)^{2c}$. For instance, it means $M \geq 46$ for $c = 2$ and $M \geq 12\,405\,543$ for $c = 3$.

Proof of the constant time of the cthRoot procedure. For $x < N^d \leq K^{2cd}$, the algorithm (see lines 5-6) can divide x by K^c at most $2d-2$ times before reaching a value $x < M = K^{2c}$ for which the c-th root is given by $\texttt{cthRoot}[x]$. This means that the CTHROOT procedure makes at most $2d-2$ recursive calls and therefore, for a fixed d, executes in constant time. □

Finally, we have proved that any root of any "polynomial" integer can be computed in constant time. More precisely:

Proposition 3. *Let $c \geq 2$ and $d \geq 1$ be fixed integers. There is a RAM with addition such that, for any input integer N:*

1. Pre-computation: *the RAM computes some tables in time $O(N)$;*
2. Computation of the root: *using these pre-computed tables and reading any integer $x < N^d$, the RAM computes in constant time the c-th root $\lfloor x^{1/c} \rfloor$.*

7.3 Generalized root: a conjecture partially resolved positively

Although we are unable to prove it, we believe that Proposition 3 can be generalized to yth root, for an integer variable $y < N^d$.

Conjecture 1. *For any fixed integer $d \geq 1$, the function $(x, y) \mapsto \lfloor x^{1/y} \rfloor$, where $x < N^d$ and $2 \leq y < N^d$, is computable in constant time after preprocessing in $O(N)$ time.*

In this subsection, we give strong arguments in favor of this conjecture.

Remark 11. *The conjecture is true for the restriction of the function $(x, y) \mapsto \lfloor x^{1/y} \rfloor$ to the following subdomains:*

1. *$y \geq \log_2 N$: then $0 \leq \lfloor x^{1/y} \rfloor < 2^d$ (because for $z = \lfloor x^{1/y} \rfloor$ we have $z^{\log_2 N} \leq z^y \leq x < N^d = 2^{d \log_2 N}$, which implies $z < 2^d$) and therefore, $\lfloor x^{1/y} \rfloor$ can be computed in time $O(d)$ by dichotomic search;*
2. *$y \leq (\log_2 N)/(12 \log_2 \log_2 N)$: then the algorithm of subsection 7.1 computing the cth root $\lfloor x^{1/c} \rfloor$, for $x < N^d$ and any fixed integer c, and its justification (subsection 7.2) can be slightly adapted.*

Since the conjecture is also answered positively for $y < \log_2 N$ and $x < N^{1-\varepsilon}$, for any fixed $\varepsilon > 0$, by pre-computing in time $O(N)$ an array $\texttt{root}$ of dimensions $N^{1-\varepsilon} \times \log_2 N$, in fact the only open case is $\log_2 N/(12 \log_2 \log_2 N) < y < \log_2 N$ with $N^{1-\varepsilon} \leq x < N^d$.

This yields the following algorithm, which makes free use of the exponential and (rounded) logarithm functions proven to be computed in constant time in Section 6.

Algorithm 19 Computation of $(x, y) \mapsto \lfloor x^{1/y} \rfloor$, for $x < \mathtt{N}^d$ and $(2 \leq y < \mathtt{L}/(12\lceil \log_2 \mathtt{L} \rceil)$ or $\mathtt{L} < y < \mathtt{N}^d)$, where $\mathtt{L} \coloneqq \lfloor \log_2 \mathtt{N} \rfloor$

```
1: procedure LinPreprocRoot()
2:     B ← max(⌈N^{1/4}⌉^3, ⌈N^{1/(d+1)}⌉^d + 1)
3:     L ← ⌊log_2 N⌋
4:     RootN ← an array of dimension L
5:     root ← an array of dimensions B × L
6:     for y from 1 to L do
7:         r ← 0
8:         while r^{3×y} < N do
9:             r ← r + 1
10:        RootN[y] ← r        ▷ r = ⌈N^{1/(3×y)}⌉
11:    for y from 1 to L do
12:        root[0][y] ← 0
13:        for x from 1 to B − 1 do
14:            s ← root[x − 1][y]
15:            if x < (s + 1)^y then
16:                root[x][y] ← s
17:            else
18:                root[x][y] ← s + 1

1: procedure improve(g, x, y)
2:     if g^y ≤ x < (g + 1)^y then
3:         return g
4:     else
5:         return (x+(y−1)×g^y)/(y×g^{y−1})

1: procedure root(x, y)
2:     if y > L then        ▷ 0 ≤ ⌊x^{1/y}⌋ < 2^d
3:         r ← 0
4:         u ← 2^d
5:         while u − r > 1 do
6:             z ← (r + u)/2
7:             if x < z^y then
8:                 u ← z
9:             else
10:                r ← z
11:        return r
12:    if y < L/(12 × ⌈log_2 L⌉) then
13:        if x < B then
14:            return root[x][y]
15:        else
16:            s ← x/(RootN[y])^y
17:            t ← root(s, y)
18:            g_0 ← t × RootN[y]
19:            g_1 ← improve(g_0, x, y)
20:            g_2 ← improve(g_1, x, y)
21:            if g_2^y ≤ x < (g_2 + 1)^y then
22:                return g_2
23:            else
24:                return g_2 − 1
```

Justification of the algorithm. The case $y > \mathtt{L}$, for $\mathtt{L} = \lfloor \log_2 N \rfloor$, lines 2-11 of the ROOT procedure, is justified by the following invariant condition: $r^y \leq x < u^y$ (meaning $0 \leq x < 2^{dy}$ at the initialization): therefore, the return value r, obtained after at most d iterations of the body of the while loop, satisfies $r^y \leq x < (r+1)^y$ as required.

The case $y < \mathtt{L}/(12 \times \lceil \log_2 \mathtt{L} \rceil)$, lines 12-24 of the ROOT procedure, is a variant of the CTHROOT procedure, where the fixed integer c is replaced by the integer variable y. Moreover, the computations which justify the correctness of this case are the same as those of CTHROOT, see Subsection 7.2, with the following adaptations.

- Instead of setting $K \coloneqq \lceil N^{1/(2c)} \rceil$, we set $K \coloneqq \lceil N^{1/(3y)} \rceil$, which we pre-compute and store as an array element: $\mathtt{RootN}[y] \coloneqq \lceil N^{1/(3y)} \rceil$.
- The recursive case of of the ROOT procedure is $x \geq B$, for $B \coloneqq \mathtt{max}(\lceil N^{1/4} \rceil^3, \lceil N^{1/(d+1)} \rceil^d + 1)$, instead of $x \geq M'$, for $M' \coloneqq \mathtt{max}(M, c_0^{2c} + c_0^{2c})$ (with $M \coloneqq K^{2c}$ and $c_0 \coloneqq \lceil \lceil \sqrt{6c - 12}\, \rceil \times (c^2 - 1)/(6c) \rceil$).
- Our computations continue to use the numbers c_0, c_1, c_2 introduced in Subsection 7.2, now defined by replacing c by y. In particular, we now set $c_0 \coloneqq \lceil \lceil \sqrt{6y - 12}\, \rceil \times (y^2 - 1)/(6y) \rceil$ and $c_2 \coloneqq (y-1)^7 \times (y+1)^4/(8y^4)$.

For the reasoning and computations of Subsection 7.2 to still be valid, it suffices to check that the following three conditions hold under the assumption $y < L/(12 \times \lceil \log_2 L \rceil)$, for sufficiently large integers y and N:

1. $B \geq K^{2y}$;
2. $B \geq c_0^{2y} + c_0^y$;
3. $K \geq 1 + 2\sqrt{c_2}$.

It is important to note that, by items 1 and 2, the condition $x \geq B$ implies $x \geq \mathtt{max}(K^{2y}, c_0^{2y} + c_0^y)$. This corresponds to the condition $x \geq M' = \mathtt{max}(K^{2c}, c_0^{2c} + c_0^c)$ of the CTHROOT procedure. Recall also that, to conclude the reasoning of Subsection 7.2, condition 3 ($K \geq 1 + 2\sqrt{c_2}$) has been assumed.

Proof of item 1. Since we have $K = \lceil N^{1/(3y)} \rceil \geq 2$ for $N \geq 2$, we obtain $K/2 \leq K - 1 < N^{1/(3y)}$. This implies $K < 2N^{1/(3y)}$ and then $K^{2y} < 2^{2y} \times N^{2/3}$. From the assumption $y < L/(12 \times \lceil \log_2 L \rceil)$, we get $2^{2y} < 2^{L/(6\lceil \log_2 L \rceil)} \leq 2^{(\log_2 N)/(6\lceil \log_2 L \rceil)} \leq N^{1/(6\lceil \log_2 L \rceil)}$ and thus $2^{2y} < N^{1/(6\lceil \log_2 L \rceil)}$. Moreover, for $N \geq 16$, we have $L = \lfloor \log_2 N \rfloor \geq 4$ and then $\lceil \log_2 L \rceil \geq 2$. This gives $2^{2y} < N^{1/12}$. All in all, we get $K^{2y} < 2^{2y} \times N^{2/3} < N^{1/12} \times N^{2/3} = N^{3/4} \leq B$, and $B \geq K^{2y}$ as claimed. □

The following lemma will be useful to prove items 2 and 3.

Lemma 11. *For all numbers y, z and α such that $y \geq 2$, $z > 1$ and $0 < \alpha \leq 1$, we have the implication*

$$y < \alpha \times z / \log_2 z \Rightarrow y \log_2 y < \alpha \times z$$

Proof of the lemma. Assume $y \leq \alpha \times z / \log_2 z$. Using also the inequalities $\log_2 \log_2 y \geq 0$ and $\log_2 \alpha \leq 0$, we obtain
$\frac{y \log_2 y}{\log_2(y \log_2 y)} = \frac{y \log_2 y}{\log_2 y + \log_2 \log_2 y} \leq y \leq \frac{\alpha \times z}{\log_2 z} \leq \frac{\alpha \times z}{\log_2 z + \log_2 \alpha} = \frac{\alpha \times z}{\log_2(\alpha \times z)}$, and thus
$\frac{y \log_2 y}{\log_2(y \log_2 y)} \leq \frac{\alpha \times z}{\log_2(\alpha \times z)}$. Noticing that the function $x \mapsto \frac{x}{\log_2 x}$ is strictly increasing on its domain, we obtain $y \log_2 y \leq \alpha \times z$ as required. □

As a direct application of Lemma 11 where we take $z = L = \lfloor log_2 N \rfloor > 1$ and $\alpha = 1/12$, we obtain

Lemma 12. *If $y < L/(12 \times \lceil \log_2 L \rceil)$ then $y \log_2 y < L/12$.*

Proof of item 2 ($B \geq c_0^{2y} + c_0^y$). We have
$c_0 = \left\lceil \left\lceil \sqrt{6y - 12} \right\rceil \times (y^2 - 1)/(6y) \right\rceil < \frac{(\sqrt{6y - 12} + 1) \times (y^2 - 1)}{6y} + 1 < \frac{(\sqrt{y - 2} + 1/\sqrt{6}) \times y + \sqrt{6}}{\sqrt{6}}$.
If $y \geq 6$, which implies $\sqrt{6} \leq y/\sqrt{6}$ and $2/\sqrt{6} < 2 \leq \sqrt{y - 2}$, we obtain $c_0 < \frac{(\sqrt{y - 2} + 2/\sqrt{6}) \times y}{\sqrt{6}} < \frac{2\sqrt{y - 2} \times y}{\sqrt{6}} < y^{3/2}$. This implies $c_0^{2y} + c_0^y < 2 \times c_0^{2y} < 2 \times (y^{3/2})^{2y} = 2 \times y^{3y} = 2 \times 2^{3y \log_2 y} < 2^{4y \log_2 y}$ (note that $y \geq 2$ implies $y \log_2 y > 1$) and therefore $c_0^{2y} + c_0^y < 2^{4y \log_2 y}$. Because of $y \log_2 y < L/12$ according to Lemma 12, we get $\log_2 y < (1/12)L/y$ and then $4y \log_2 y < (1/3)L$. All in all, we deduce $c_0^{2y} + c_0^y < 2^{(1/3)L} \leq N^{1/3} < B$ and then $B \geq c_0^{2y} + c_0^y$. □

Proof of item 3 ($K \geq 1 + 2\sqrt{c_2}$). We have $1 + 2\sqrt{c_2} = 1 + \frac{(y - 1)^{7/2} \times (y + 1)^2}{\sqrt{2} \times y^2}$. Moreover, it is easy to verify that for $y \geq 6$, we have $\frac{(y + 1)^2}{\sqrt{2} \times y^2} < 1$. It comes $1 + 2\sqrt{c_2} < 1 + (y - 1)^{7/2} < y^{7/2}$, which implies $1 + 2\sqrt{c_2} < 2^{(7/2) \log_2 y}$. Because of $y \log_2 y < L/12$ according to Lemma 12, we get $\log_2 y < (1/12)L/y$ and then $1 + 2\sqrt{c_2} < 2^{(7/2) \times (1/12) \times L/y} < 2^{L/(3y)} \leq N^{1/(3y)} \leq \lceil N^{1/(3y)} \rceil = K$. We have proved $K \geq 1 + 2\sqrt{c_2}$. □

Justification of the complexity. In the LinPreprocRoot procedure, for each $y \in [1, L]$, the while loop, lines 8-9, does at most $N^{1/(3y)}$ iterations. Because of $N^{1/(3y)} \leq N^{1/3}$, the running time of the for loop, lines 6-10, is $O(L \times N^{1/3}) = O(N)$. Since the run time of lines 11-18 (of LinPreprocRoot) is $O(L \times B) = O(N)$, all preprocessing runs in time $O(N)$.

To analyze the time of the root procedure, it suffices to determine its recursive depth. We have $(\texttt{RootN}(y))^y = \lceil N^{1/(3y)} \rceil^y \geq N^{1/3}$. Since we cannot divide the integer $x < N^d$ by $(\texttt{RootN}(y))^y$ (which is an integer greater than $N^{1/3}$) more than $3 \times d$ times, the running time of the root procedure is $O(d)$, which is constant for fixed d.

Conveniently, the notations used must be recalled and completed:

Notation. *Let L, λ and B denote the integers $L \coloneqq \lfloor \log_2 N \rfloor$, $\lambda \coloneqq L/(12 \times \lceil \log_2 L \rceil)$, and, for a fixed integer $d \geq 1$, $B \coloneqq \texttt{max}(\lceil N^{1/4} \rceil^3, \lceil N^{1/(d+1)} \rceil^d + 1)$.*

In addition to the above algorithm, the root $\lfloor x^{1/y} \rfloor$ can also be computed in constant time for "most" integers y in the "remaining" interval $[\lambda, L]$. This is the consequence of the following two lemmas.

Lemma 13. *For all positive integers x, p, q, the identity $\lfloor x^{1/(p \times q)} \rfloor = \left\lfloor \lfloor x^{1/p} \rfloor^{1/q} \right\rfloor$ holds.*

Proof. Let u, v denote the successive roots $u \coloneqq \lfloor x^{1/p} \rfloor$ and $v \coloneqq \lfloor u^{1/q} \rfloor$. That means $u^p \leq x \leq (u+1)^p - 1$ and $v^q \leq u \leq (v+1)^q - 1$, from which we deduce $v^{p\times q} \leq u^p \leq x \leq (u+1)^p - 1 \leq ((v+1)^q - 1 + 1)^p - 1 = (v+1)^{p\times q} - 1$. That gives $v^{p\times q} \leq x \leq (v+1)^{p\times q} - 1$, which means $v = \lfloor x^{1/(p\times q)} \rfloor$ as expected. □

Lemma 14. *Let $d \geq 1$ be a fixed integer. Suppose N is large enough for the inequality $\lambda^2/d^2 > L$ to hold. Let y be an integer in $[\lambda, L]$, which is not of the form, called* forbidden form[11]*, $p \times q$ for a prime number p and an integer $q \leq d$. Then, y has a divisor y_1 such that $d < y_1 < \lambda$.*

Proof. Let us consider the decomposition of y into prime factors, $y = \prod_{i=1}^{k} p_i$, with $k \geq 2$ and $p_1 \geq p_2 \geq \cdots \geq p_k$. Let i_0 (resp. i_1) be the least index such that $\prod_{i=1}^{i_0} p_i > d$ (resp. $\prod_{i=1}^{i_1} p_i \geq \lambda$). For N large enough, we have $\lambda \geq d$, which implies $i_0 \leq i_1$. We have two cases:

1. $i_0 < i_1$: taking $y_1 = \prod_{i=1}^{i_0} p_i$, we obtain (by definition of i_0 and i_1) $d < y_1 < \lambda$;
2. $i_0 = i_1$: then, we get $\prod_{i=1}^{i_0} p_i = \prod_{i=1}^{i_1} p_i \geq \lambda$ and $\prod_{i=1}^{i_0-1} p_i \leq d$ (by definition of i_0 and i_1), which gives, by division, $p_{i_0} \geq \lambda/d$. It implies $i_0 = 1$, since otherwise, we would have $L \geq y \geq p_{i_0-1} \times p_{i_0} \geq p_{i_0}^2 \geq \lambda^2/d^2$, which contradicts the inequality $\lambda^2/d^2 > L$, for N large enough. Thus, we obtain the factorization $y = p_1 \times y_1$, where $p_1 \geq \lambda$ and $y_1 \coloneqq \prod_{i=2}^{k} p_i$. We get $y_1 < \lambda$, since otherwise, we would have $y = p_1 \times y_1 \geq \lambda^2 > L$, a contradiction. Moreover, the inequality $y_1 > d$ also holds, since otherwise, $y = p_1 \times y_1$ would be of the "forbidden form".

Thus, in both cases, we can exhibit a divisor y_1 of y such that $d < y_1 < \lambda$. □

Notice that we can compute in time $O(L \times \lambda) = O(N)$ an array `factor`$[\lambda..L]$ defined by `factor`$[y] \coloneqq y_1$, where y_1 is the least divisor of y in $[d+1, \lambda-1]$, if it exists (by Lemma 14), and is 0 otherwise, i.e. when y is of the forbidden form: it suffices to check, for all $y_1 \in [d+1, \lambda-1]$, if y `mod` $y_1 = 0$.

Now, using the `factor` array and Lemma 13, it is easy to compute $\lfloor x^{1/y} \rfloor$ in constant-time (with linear-time preprocessing), for every $x < N^d$ and every $y \in [\lambda, L]$ not in forbidden form: take $y_1 = $ `factor`$[y]$ and $y_2 = y/y_1$, which gives $\lfloor x^{1/y} \rfloor = \left\lfloor \lfloor x^{1/y_1} \rfloor^{1/y_2} \right\rfloor$; the roots $z \coloneqq \lfloor x^{1/y_1} \rfloor$ and then $\lfloor z^{1/y_2} \rfloor = \left\lfloor \lfloor x^{1/y_1} \rfloor^{1/y_2} \right\rfloor$ can be computed respectively by the above algorithm, lines 12-24, and by reading the element `root`$[z, y]$ of the array `root`$[0..B-1][1..L]$, since we have respectively $y_1 < \lambda$ and $y_1 \geq d+1$, from which $z = \lfloor x^{1/y_1} \rfloor \leq (N^d)^{1/(d+1)} \leq \lceil N^{1/(d+1)} \rceil^d < B$ is deduced.

Remark 12. *Using Lemmas 14 and 13, it is easy to verify that Conjecture 1 is equivalent to its following restriction: For any fixed integer $d \geq 1$, the function $(x, y) \mapsto \lfloor x^{1/p} \rfloor$, where $x < N^d$ and p is a prime number such that $\log_2 N/(12 \log_2 \log_2 N) < p < \log_2 N$, is computable in constant time after preprocessing in $O(N)$ time. Indeed, an integer in forbidden form being of the form $y = p \times q$, for a prime number p and an integer $q \leq d$, the computation of $\lfloor x^{1/y} \rfloor = \left\lfloor \lfloor x^{1/p} \rfloor^{1/q} \right\rfloor$ is reduced to the computation of $z \coloneqq \lfloor x^{1/p} \rfloor$ since, obviously, the final result $\lfloor z^{1/q} \rfloor$ can be computed in constant time.*

8 Computing many other operations in constant time

Real computers treat the contents and addresses of registers as binary words. In addition to arithmetic operations and for the sake of efficiency, assembly languages as well as programming languages (Python, etc.) use string operations and bitwise logical operations: `and`, `or`, `xor`, etc.

> *The objects processed by the RAM model are integers that can be assumed to be in binary notation, i.e. they can be seen as binary words. In this section, we show that the usual string operations and logical operations on binary words can be computed in constant time with linear-time preprocessing.*

[11] In particular, for $d = 1$ (resp. $d = 2$), an integer is of the forbidden form iff it is prime (resp. it is prime or twice a prime number). In general, by the repartition theorem of prime numbers, there are asymptotically about $d \times L/(\ln L)$ integers of forbidden form among the integers in $[\lambda, L]$, which is only $O((\log N)/(\log \log N))$.

8.1 String operations computed in constant time

We identify each integer $x \in \mathbb{N}$ with the string $x_{\ell-1}\dots x_1x_0 \in \{0,1\}^\ell$, of its binary notation with $x = \sum_{0\le i<\ell} 2^i \times x_i$. The length ℓ of the representation of x (without useless leading zeros: $x_{\ell-1} > 0$) is given by the following function:

$$\textsc{length}(x) = \begin{cases} 1 & \texttt{if } x = 0 \\ \lfloor \log_2 x \rfloor + 1 & \texttt{otherwise.} \end{cases}$$

Remark 13. *We have* $\textsc{length}(x) \le \log_2 N + O(1)$*, for each integer* $x = O(N)$ *contained in a RAM register, and* $\textsc{length}(x) = O(\log N)$ *for each "polynomial" integer* $x = O(N^d)$*, for a fixed* d*.*

Three operations are essential for manipulating binary strings:

- the concatenation of two strings X and Y is defined as $\textsc{Conc}(X,Y) \coloneqq X \times 2^{\textsc{length}(Y)} + Y$;
- the function that maps each pair (X,i) of a string X and an index i to the ith bit x_i of X is defined as $\textsc{Bit}(X,i) \coloneqq (X \texttt{ div } 2^i) \texttt{ mod } 2$;
- the substring function which associates to each string $X = x_{\ell-1}\dots x_1x_0$ and two indices i,j with $\ell \ge i > j \ge 0$ the (nonempty) substring $x_{i-1}x_{i-2}\dots x_j$, is defined as $\textsc{Substring}(X,i,j) \coloneqq (X \texttt{ mod } 2^i) \texttt{ div } 2^j$.

With Substring, one can express that a string X is a suffix or a prefix of another string Y:

- X is a suffix of $Y \iff X = \textsc{Substring}(Y, \textsc{length}(X), 0)$;
- X is a prefix of $Y \iff \begin{cases} \textsc{length}(X) \le \textsc{length}(Y) \text{ and} \\ X = \textsc{Substring}(Y, \textsc{length}(Y), \textsc{length}(Y) - \textsc{length}(X)) \end{cases}$.

Remark 14. *Note that these procedures can be straightforwardly adapted if the integers are represented in a fixed base* $\gamma \ge 2$*, i.e. are identified with strings* $x_{\ell-1}\dots x_1x_0 \in \{0,1,\dots,\gamma-1\}^\ell$.

8.2 Bitwise logical operations can be computed in constant time

Processing is similar for each bitwise logical operation. As an example, let us study the operation `xor` applied to two integers X,Y which are less than N^d (for a fixed d) and can therefore be identified with the strings of their binary notations $x_{\ell-1}\dots x_0$ and $y_{\ell-1}\dots y_0$ of length $\ell = \textsc{length}(\texttt{max}(X,Y)) = O(\log N)$.

The natural idea is to decompose each operand $X,Y < 2^\ell$ into its ℓ bits copied in ℓ registers, to compute the bits $z_i = x_i \texttt{ xor } y_i$ in ℓ registers too, and to get the result $Z = X \texttt{ xor } Y$ by concatenating the ℓ bits z_i in one register: $Z = z_{\ell-1}\dots z_0$. Obviously, this process takes time $O(\ell)$, which is, generally, not a constant time.

To compute the integer $X \texttt{ xor } Y$ in constant time, for operands $X,Y < N^d$ where d is a fixed integer, our trick is to notice that if we set $X = X_1 \times p + X_0$ and $Y = Y_1 \times p + Y_0$, where p is a power of two such that $X_0, Y_0 < p$, then we get

$$X \texttt{ xor } Y = (X_1 \texttt{ xor } Y_1) \times p + (X_0 \texttt{ xor } Y_0) \tag{12}$$

Of course, the same "decomposition" property holds for the other bitwise operations `and`, `or`: $X \texttt{ and } Y = (X_1 \texttt{ and } Y_1) \times p + (X_0 \texttt{ and } Y_0)$ and $X \texttt{ or } Y = (X_1 \texttt{ or } Y_1) \times p + (X_0 \texttt{ or } Y_0)$. Also, note that those equalities hold for all pairs of integers $X,Y > 1$, i.e. represented with at least 2 bits.

Now, by using equation (12) iteratively for $p = 2$, we can pre-compute a 2-dimensional table, called `xorAr`, giving $Z = X \texttt{ xor } Y$ for all the pairs of small enough operands X,Y. For large operands, we use equation (12) for $p = 2^\ell$ and $\ell = \left\lfloor \frac{1}{2} \times \textsc{length}(\texttt{max}(X,Y)) \right\rfloor$. Thus, we obtain a recursive procedure called XOR: at each step the maximum length L of the arguments of XOR, $L \coloneqq \textsc{length}(\texttt{max}(X,Y))$, is divided by two, i.e. L becomes $\lfloor L/2 \rfloor$ or $\lceil L/2 \rceil$.

The previous arguments justify that the following algorithm correctly computes the `xor` operation.

Algorithm 20 Computation of the `xor` operation

```
1: procedure LinPreprocXOR()
2:     K ← sqrt(N)                                   ▷ K = ⌈√N⌉
3:     xorAr ← array of dimension K × K
4:     for i from 0 to 1 do
5:         for j from 0 to 1 do
6:             xorAr[i][j] ← (i + j) mod 2
7:     for i from 0 to K − 1 do
8:         for j from 0 to K − 1 do
9:             if i > 1 or j > 1 then
10:                xorAr[i][j] ← 2 × xorAr[i/2][j/2]
                              + xorAr[i mod 2][j mod 2]
```

```
1: procedure XOR(x, y)
2:     if x < K and y < K then
3:         return xorAr[x][y]
4:     else
5:         ℓ ← length(max(x, y))/2
6:         p ← 2^ℓ
7:         return p×XOR(x/p, y/p)
               + XOR(x mod p, y mod p)
```

Time complexity. We have the linearity of the preprocessing because $K = \left\lceil \sqrt{N} \right\rceil$ and thus the two nested loops, lines 7-10, take $O(N)$ time. The result of the preprocessing is an array such that $\texttt{xorAr}[\texttt{x}][\texttt{y}] = \texttt{x xor y}$ for all $\texttt{x}, \texttt{y} < K$. When calling $\textsc{xor}(\texttt{x}, \texttt{y})$, either both arguments $\texttt{x}, \texttt{y}$ are smaller than K, or we recurse with two calls, $\textsc{xor}(\texttt{x}', \texttt{y}')$ and $\textsc{xor}(\texttt{x}'', \texttt{y}'')$, where the length of the maximum of $\texttt{x}'$ and $\texttt{y}'$ (resp. $\texttt{x}''$ and $\texttt{y}''$) is, at most, half the length of the maximum of $\texttt{x}$ and $\texttt{y}$ plus one. Starting with $\texttt{x}, \texttt{y} < N^d$, which implies $\texttt{x}, \texttt{y} < K^{2d}$, we obtain, for the number L_2 defined as the smallest power of two greater than or equal to $\textsc{length}(\texttt{max}(\texttt{x}, \texttt{y}))$, the inequalities[12] $L_2 < 2 \times \textsc{length}(\texttt{max}(\texttt{x}, \texttt{y})) \leq 2 \times \textsc{length}(K^{2d}) \leq 4d \times \textsc{length}(K)$, from which we deduce $L_2/(2^{\lceil \log_2 d \rceil + 2}) \leq L_2/(4d) < \textsc{length}(K)$.

Therefore, the recursion depth is bounded by the number $\delta := \lceil \log_2 d \rceil + 2$ since, after repeatedly dividing L_2 by two, δ times, L_2 becomes smaller than $\textsc{length}(K)$ so that the operands of the XOR procedure themselves become less than K. This means that the XOR procedure executes in constant time, for any fixed integer $d \geq 1$.

Getting other bitwise logical operators. All the bitwise operations can be obtained in the same way. To get the operation `and` (resp. `or`), we only have to replace the `xorAr` array with an `andAr` (resp. `orAr`) array, and the assignment $\texttt{xorAr}[\texttt{i}][\texttt{j}] \leftarrow (\texttt{i} + \texttt{j}) \bmod 2$, which defines the Boolean `xor` operation (line 6 of the preprocessing procedure), is replaced by the assignment $\texttt{andAr}[\texttt{i}][\texttt{j}] \leftarrow \texttt{i} \times \texttt{j}$ (resp. $\texttt{orAr}[\texttt{i}][\texttt{j}] \leftarrow \texttt{max}(\texttt{i}, \texttt{j})$), defining the Boolean `and` (resp. `or`) operation.

8.3 The operations computed in linear time on Turing machines or on cellular automata are computable in constant time on RAMs

It is easy to verify that addition, subtraction and all the string or logical operations studied in the previous subsections are computable in linear time on multi-tape Turing machines. On the other hand, it is known that multiplication and modular multiplication are computable in linear time on cellular automata [36, 33, 62].

- *A natural question arises: is it true that each operation computable in linear time on a Turing machine or on a cellular automaton is computable in constant time on a RAM with addition, with linear-time preprocessing?*
- *We answer this question positively.*

Since it is known and easy to verify that any problem computable in linear time on a multi-tape Turing machine is computable in linear time on a cellular automaton [78, 82], it suffices to prove the expected result for cellular automata.

First, we need some definitions about cellular automata.

Definition 6 (cellular automaton, configuration, computation)**.** *A* cellular automaton *(CA)* $\mathcal{A} := (Q, \delta)$ *consists of a finite set of* states Q *and a* transition function $\delta : Q^3 \to Q$.

A configuration C *of the CA is a bi-infinite word* $\ldots c_{-2}c_{-1}c_0c_1c_2 \ldots \in Q^{\mathbb{Z}}$.

The successor configuration *of* C *for the CA is* $\delta(C) := \ldots c'_{-2}c'_{-1}c'_0c'_1c'_2 \ldots$ *where* $c'_i := \delta(c_{i-1}, c_i, c_{i+1})$*, for each* $i \in \mathbb{Z}$.

The computation *of the CA from any initial configuration* C *is the sequence of configurations* $(\delta^j(C))_{j \geq 0}$*, defined by* $\delta^0(C) := C$ *and* $\delta^{j+1}(C)) := \delta(\delta^j(C))$.

[12] We use the general inequality $\textsc{length}(u \times v) \leq \textsc{length}(u) + \textsc{length}(v)$, for all integers u, v.

We are interested in finite word problems.

Definition 7 (configuration representing a finite word)**.** *The configuration of a cellular automaton $\mathcal{A} = (Q, \delta)$ which represents a finite nonempty word $w \in \Sigma^+$ over a finite alphabet $\Sigma \subset Q$ is the bi-infinite word $\ldots\sharp\sharp w\sharp\sharp\ldots$, simply denoted $\sharp w\sharp$, where $\sharp \in Q \setminus \Sigma$ is a special symbol (comparable to the "blank" symbol of a Turing machine).*

Definition 8 (permanent state, permanent state on the right)**.** *The $\sharp$ state is* permanent *for the cellular automaton $\mathcal{A} = (Q, \delta)$ if we have $\delta(q, \sharp, q') = \sharp$, for all $q, q' \in Q$.*

The $\sharp$ state is permanent on the right *for $\mathcal{A} = (Q, \delta)$ if we have $\delta(q, \sharp, \sharp) = \sharp$, for each $q \in Q$.*

Remark 15. *If $\sharp$ is a permanent state on the right, the infinite sequence of $\sharp$ states to the right of a configuration of form $C := \ldots\sharp\sharp U\sharp\sharp\ldots$, simply denoted $\sharp U\sharp$, cannot be modified in $\delta(C)$: the interval of the "significant part" of the configuration (its states other than those of the infinite $\sharp$-sequences to its left or to its right) can only widen towards its left.*

We have to describe how a CA computes a function in linear time, a notion which is defined and studied in [37] and which we slightly adapt here below.

Definition 9 (function and operation computed by a CA in linear time)**.** *Let be two finite alphabets Σ, Σ' and a cellular automaton $\mathcal{A} = (Q, \delta)$ such that $\Sigma \subset Q$, with the state $\sharp \in Q \setminus \Sigma$ permanent on the right.*

$\mathcal{A}$ computes *a function $f : \Sigma^+ \to \Sigma'^+$* in linear time *if there are a constant integer $c \geq 1$ and a projection $\pi : Q \to \Sigma'$ such that, for each integer $n > 0$ and each word $w \in \Sigma^n$, called the* input *of $\mathcal{A}$, we have $\delta^{cn}(\sharp w\sharp) = \sharp v\sharp$ for a word $v = v_1 \ldots v_m \in (Q \setminus \{\sharp\})^+$ with $f(w) = \pi(v_1)\ldots\pi(v_m)$. The word $f(w) \in \Sigma'^+$ is called the* output *of $\mathcal{A}$.*

$\mathcal{A}$ computes *an operation $\mathtt{op} : \mathbb{N}^r \to \mathbb{N}$ of arity $r \geq 1$* in linear time *if it computes in linear time a function $X_1; \ldots; X_r \mapsto Y$ from $\{0, 1, ;\}^+$ to $\{0, 1\}^+$, i.e. for the integers $X_1, \ldots, X_r$ represented in binary and a binary representation Y of $\mathtt{op}(X_1, \ldots, X_r)$, possibly including leading zeros.*

Remark 16. *The condition of Definition 9 that the output configuration of a CA is $\delta^{cn}(\sharp w\sharp)$ for an input w of length n, which means that it is produced at exactly time cn, may seem difficult to satisfy. However, it can be controlled by the CA itself. Indeed, as we have known for many years [59, 60, 61], one can build a cellular automaton $\mathcal{A}_\varphi = (Q_\varphi, \delta_\varphi)$ – called a "firing squad" automaton – having two special states, the permanent state $\sharp$ and the "fire state" φ (synchronization state), such that, for each integer n and each input w of length n, we have $\delta^n_\varphi(\sharp w\sharp) = \sharp\varphi^n\sharp$, while, for each time $t < n$, we have $\delta^t_\varphi(\sharp w\sharp) = \sharp v(t)\sharp$ with a word $v(t) \in (Q \setminus \{\varphi\})^n$ (the "fire state" φ does not appear before time n).*

By "coupling" (by running in parallel) the original CA and the "firing squad" automaton $\mathcal{A}_\varphi$, it can be assumed that each active cell has a clock indicating instant n (the instant when φ is encountered), and by iterating, the instants $2n$, $3n$, and, more generally, cn, for any fixed integer $c \geq 1$.

It should be noted that the states of a CA can be composite without inconvenience (the important thing is that the number of states is finite); in particular, the output is finally "filtered" thanks to the π projection.

Here is the most general theorem of this paper.

Theorem 3. *Let $d \geq 1$ be a constant integer and let $\mathtt{op} : \mathbb{N}^r \to \mathbb{N}$ be an operation of arity $r \geq 1$ which is computable in linear time on a cellular automaton. There is a RAM with addition, such that, for any input integer $N > 0$:*

1. Pre-computation: *the RAM computes some tables in time $O(N)$;*
2. Operation: *using these pre-computed tables and reading any integers $X_1, \ldots, X_r < N^d$, the RAM computes in constant time the integer $\mathtt{op}(X_1, \ldots, X_r)$.*

The proof of this result is quite elaborate: it is given in the Appendix.

9 Minimality of the RAM with addition

We have proved that the RAM model using addition as the only operation can perform in constant time – with linear preprocessing – any other usual arithmetic operation and, above all, Euclidean division. The reader may ask the following natural questions which we explore in this section:

1. Is the addition a "minimal" operation for our complexity results? Can it be replaced by a finite set of unary operations (such as the successor and predecessor functions)?

2. Can addition be replaced, as a primitive operation, by any other usual arithmetic operation – subtraction, multiplication, division – or a combination of two of them?

9.1 Addition is not computable in constant time on a RAM with only unary operations

In this subsection, we will establish the negative result given by its title and formulated precisely and in a stronger form by Proposition 4.

Proposition 4. *Let* `UnaryOp` *be a finite set of* unary *operations. There is* no *RAM in* $\mathcal{M}$[`UnaryOp`] *such that, for any input integer* N*:*

1. Pre-computation: *the RAM computes some tables in* space $O(N)$*;*

2. Addition: *by using these pre-computed tables and by reading any two integers* $x, y < N$*, the RAM computes in* constant time *their sum* $x + y$*.*

Also, the result is still valid if the RAM is allowed to use any (in)equality test, with $=, \neq, <, \leq$*, in its branching instructions.*

Remark 17. *By its item 1, Proposition 4 states a strictly stronger result than what is required. Indeed, the space used by a RAM operating in linear time is by definition linear.*

Proof of Proposition 4 by contradiction. Suppose that there is a RAM M in $\mathcal{M}$[`UnaryOp`] which fulfills the conditions 1,2 of Proposition 4. Without loss of generality, the following conditions can be assumed.

- Step 1 of M (the *pre-computation*) ends with the internal memory registers $R(0), R(1), \ldots$ containing the successive integers $p(0), p(1), \ldots, p(cN), 0, 0, \ldots$, for a constant $c \geq 1$, with $p(i) \leq cN$ in the register $R(i)$, for each $i \leq cN$, and 0 in $R(i)$, for each $i > cN$;

- Step 2 of M (the *addition*) does not modify the internal memory registers $R(0), R(1), \ldots$ (which are read-only throughout Step 2) but uses a fixed number of read/write registers, denoted A (the accumulator) and $B_1, \ldots, B_k$ (the buffers), all initialized to 0, and two read-only registers X and Y, which contain the operands x and y, respectively. The program of Step 2 is a sequence $I_0, \ldots, I_{r-1}$ of labeled instructions of the following forms:

  - $A = 0$; $A = X$; $A = Y$;
  - $A = \texttt{op}(A)$, for some $\texttt{op} \in$ `UnaryOp`;
  - $A = R(A)$ (meaning $A = R(a)$, where a is the current content of A);
  - $B_i = A$, for some $i \in \{1, \ldots, k\}$; $A = B_i$, for some $i \in \{1, \ldots, k\}$;
  - `if` $A \prec B_1$ `then goto` ℓ_0 `else goto` ℓ_1, for $\prec \in \{=, \neq, <, \leq, >, \geq\}$ and some $\ell_0, \ell_1 \in \{0, \ldots, r-1\}$ (branching instruction);
  - `return` A (this is the last instruction I_{r-1}; recall that the instruction executed after any other instruction I_j, $j < r-1$, is I_{j+1}, except after the branching instruction).

Justification: The constraint that the internal memory be read-only throughout Step 2 is made possible by the hypothesis that the time of Step 2 is bounded by some constant. Thus, Step 2 can consult/modify only a constant number of registers $R(i)$. Instead of modifying the contents of a register $R(i)$, we copy it – at the first instant when it needs to be modified – in one of the buffer registers $B_1, \ldots, B_k$ so that its contents can be modified as that of the original register $R(i)$ that it simulates.

More precisely, if m is an upper bound of the constant time of the simulated (original) constant-time program so that this program modifies (at most) m distinct registers $R(i)$, then $k \coloneqq 2m + 2$ buffer registers $B_1, \ldots, B_k$ are sufficient for the simulation: B_1 is the original buffer; the contents of B_2 is the number of original registers $R(i)$ which have been currently modified (in the simulated/original constant-time program); the sequence of m buffers $B_3, \ldots, B_{m+2}$ (resp. $B_{m+3}, \ldots, B_{2m+2}$) contain the current sequence of addresses i (resp. sequence of contents) of

these registers $R(i)$ [13]. The reader can check that the time of Step 2 of M is still bounded by a constant $\mu = O(m^2)$.

For $0 \leq i \leq \mu$, let $(A(i), B_1(i), \ldots, B_k(i), L(i))$ respectively be the sequence of contents of the read/write registers $A, B_1, \ldots, B_k$ and the value $L(i)$ of the ordinal instruction counter at instant i of Step 2. At instant 0, we have $(A(0), B_1(0), \ldots, B_k(0), L(0)) = (0, 0, \ldots, 0, 0)$. For convenience, we adopt the following convention when $L(i) = r - 1$, for some $i < \mu$ (recall that the instruction I_{r-1} is `return` A):

$$(A(i+1), B_1(i+1), \ldots, B_k(i+1), L(i+1)) \coloneqq (A(i), B_1(i), \ldots, B_k(i), r-1) \tag{13}$$

Claim 1. *At each instant i ($0 \leq i \leq \mu$) of Step 2, an equation of the following form holds:*

$$(A(i), B_1(i), \ldots, B_k(i), L(i)) = \begin{cases} \texttt{if } t_1 \texttt{ then } (a_1, b_{1,1}, \ldots, b_{k,1}, \lambda_1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \texttt{if } t_q \texttt{ then } (a_q, b_{1,q}, \ldots, b_{k,q}, \lambda_q) \end{cases} \tag{14}$$

Here, each test t_j is a conjunction of (in)equalities $\bigwedge_h c_h \prec_h d_h$ where $\prec_h \in \{=, \neq, <, \leq, >, \geq\}$ and each term $a_h, b_{h,j}, c_h, d_h$ is of the (unary) form $(f_1 \circ f_2 \circ \cdots \circ f_s)(u)$ where u is either X, or Y, or the constant 0, and $s \geq 0$, and each f_j is either a unary operation `op` $\in$ `UnaryOp` *or the "internal memory function" $p : [0, cN] \to [0, cN]$ which sends each address α to the contents $p(\alpha)$ of the corresponding read-only register $R(\alpha)$ (note that the function p depends on N).*

Moreover, the family of tests $(t_j)_{1 \leq j \leq q}$ satisfies the partition property*: for each pair of integers (x, y),* exactly one *test t_j is true for $X \coloneqq x$ and $Y \coloneqq y$.*

Proof of the claim. It is a proof by induction on the instant i. Clearly, Claim 1 is true for $i = 0$. Assume that the equality 14 is true for some $i < \mu$. There is nothing to prove if $I_{L(i)}$ is the instruction `return` A because of equality 13. All other instructions are handled by a fairly similar process, with specific processing for the branching instruction. First, suppose that $I_{L(i)}$ is an assignment of the form $A = v$ where $v \in \{0, X, Y\}$. Then we have

$$(A(i+1), B_1(i+1), \ldots, B_k(i+1), L(i+1)) = \begin{cases} \texttt{if } t_1 \texttt{ then } (v, b_{1,1}, \ldots, b_{k,1}, \lambda_1 + 1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \texttt{if } t_q \texttt{ then } (v, b_{1,q}, \ldots, b_{k,q}, \lambda_q + 1) \end{cases}$$

Now, suppose that $I_{L(i)}$ is an assignment $A =$ `op`(A) where `op` $\in$ `UnaryOp`. Then we have

$$(A(i+1), B_1(i+1), \ldots, B_k(i+1), L(i+1)) = \begin{cases} \texttt{if } t_1 \texttt{ then } (\texttt{op}(a_1), b_{1,1}, \ldots, b_{k,1}, \lambda_1 + 1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \texttt{if } t_q \texttt{ then } (\texttt{op}(a_q), b_{1,q}, \ldots, b_{k,q}, \lambda_q + 1) \end{cases} \tag{15}$$

In the case where $I_{L(i)}$ is the assignment $A = R(A)$, replace the terms $\texttt{op}(a_1), \ldots, \texttt{op}(a_q)$ by the respective terms $p(a_1), \ldots, p(a_q)$ (involving the "internal memory function" p) in equality 15.

The cases where $I_{L(i)}$ is $A = B_i$ or $B_i = A$ are similar. For example, for $B_1 = A$, we have

$$(A(i+1), B_1(i+1), \ldots, B_k(i+1), L(i+1)) = \begin{cases} \texttt{if } t_1 \texttt{ then } (a_1, a_1, b_{2,1}, \ldots, b_{k,1}, \lambda_1 + 1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \texttt{if } t_q \texttt{ then } (a_q, a_q, b_{2,q}, \ldots, b_{k,q}, \lambda_q + 1) \end{cases}$$

Finally, suppose that $I_{L(i)}$ is `if` $A \prec B_1$ `then goto` ℓ_0 `else goto` ℓ_1. Clearly, we have

$$(A(i+1), B_1(i+1), \ldots, B_k(i+1), L(i+1)) = \begin{cases} \texttt{if } t_1 \wedge a_1 \prec b_{1,1} \texttt{ then } (a_1, b_{1,1}, \ldots, b_{k,1}, \ell_0) \\ \texttt{if } t_1 \wedge a_1 \not\prec b_{1,1} \texttt{ then } (a_1, b_{1,1}, \ldots, b_{k,1}, \ell_1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \texttt{if } t_q \wedge a_q \prec b_{1,q} \texttt{ then } (a_q, b_{1,q}, \ldots, b_{k,q}, \ell_0) \\ \texttt{if } t_q \wedge a_q \not\prec b_{1,q} \texttt{ then } (a_q, b_{1,q}, \ldots, b_{k,q}, \ell_1) \end{cases}$$

[13] When the original RAM tries to write a value v in a register $R(i)$, then the simulating RAM M consults the list of contents of the s "address" buffers $B_3, \ldots, B_{s+2}$, where s is the contents of the counter B_2. If the address i is found, that means, for some j ($3 \leq j \leq s+2$), B_j contains i, then the value v is written in B_{j+m} (the buffer that simulates $R(i)$). Otherwise, the (not found) address i is copied in the buffer B_{s+3} (the first buffer not yet used after B_{s+2}), the value v is written in the buffer B_{s+3+m} – this buffer now simulates $R(i)$ – and the counter buffer B_2 is incremented by 1: it now contains $s + 1$.

Note that each test has the required form since the negation $\not\prec$ of a relation $\prec$ in the set $\{=, \neq, <, \leq, >, \geq\}$ belongs also to this set; moreover, by construction, the $2q$ tests still satisfy the "partition property".

In conclusion, the reader can observe that for every possible instruction $I_{L(i)}$ of M, an equality of the required form 14 with i replaced by $i+1$ is true. This completes the proof by induction of Claim 1. □

End of the proof of Proposition 4. The contradiction will be proven as an application of the "pigeonhole" principle. By hypothesis, the Step 2 of M (addition step) computes/returns the sum $x + y$ at time μ, that means $A(\mu) = x + y$. Therefore, by Claim 1, we obtain, for all $(x, y) \in [0, N[^2$,

$$x + y = A(\mu) = \begin{cases} \texttt{if } t_1(x,y) \texttt{ then } a_1 \\ \ldots\ldots\ldots\ldots \\ \texttt{if } t_q(x,y) \texttt{ then } a_q \end{cases} \tag{16}$$

where each test t_j is a conjunction of (in)equalities $\bigwedge_h c_h \prec_h d_h$ where $\prec_h \in \{=, \neq, <, \leq, >, \geq\}$ and each term a_j, c_h, d_h is of the (unary) form $f(0)$, $f(x)$ or $f(y)$, in which f is a composition $f_1 \circ f_2 \circ \cdots \circ f_s$ of functions $f_i \in \texttt{UnaryOp} \cup \{p\}$.

Without loss of generality, assume $N > q$. Let us define the q parts D_j of $[0, N[^2$ defined by the tests t_j: $D_j := \{(x, y) \in [0, N[^2 \mid t_j(x, y)\}$. The partition property implies $\sum_{j=1}^{q} \texttt{card}(D_j) = N^2$ from which it comes – by the pigeonhole principle – that at least one of the q sets D_j has at least N^2/q elements. For example, assume $\texttt{card}(D_1) \geq N^2/q$.

Also, by 16, the implication $t_1(x, y) \Rightarrow x + y = a_1$ is valid. Therefore, we get $\texttt{card}\{(x, y) \in [0, N[^2 \mid x + y = a_1\} \geq \texttt{card}(D_1) \geq N^2/q$. Now, suppose that a_1 is of the form $f(x)$ (the case $f(y)$ is symmetrical and the case $f(0)$ is simpler). By a new application of the pigeonhole principle, we deduce that there is at least one integer $x_0 \in [0, N[$ such that

$$\texttt{card}\{y \in [0, N[\mid x_0 + y = f(x_0)\} \geq N/q$$

From the assumption $N > q$, we deduce that there are at least two distinct integers $y_1, y_2 \in [0, N[$ such that $x_0 + y_1 = f(x_0) = x_0 + y_2$, a contradiction. This concludes the proof of Proposition 4 by contradiction. □

Remark 18. *Using the pigeonhole principle again, it is also easily seen that not only addition but also each* binary *operation whose result really depends on the two operands cannot be computed in constant time on a RAM* using only unary operations*. Intuitively, a constant number of comparisons (branching instructions) generates a constant number of cases: to compute a binary operation, it is not enough to use a constant number of cases defined by (in)equalities over "unary" terms, each of which depends on only one operand.*

9.2 Can addition be replaced by another arithmetic operation?

Although addition is the simplest of the four arithmetic operations and perhaps for this reason, it is among them the only "fundamental" operation of the RAM model, as this subsection shows.

The question we attempt to answer in this subsection is: Can addition be replaced by one of the other three standard operations, subtraction, multiplication $\times$ and Euclidean division $\texttt{div}$? If not, can it be replaced by two of them?

Obviously, the addition can be replaced by the subtraction over $\mathbb{Z}$, i.e. relative integers, as $x+y = x-(0-y)$. Addition can also be replaced by subtraction *and* multiplication (or multiplication by 2) on natural numbers because of the following identity where $x \dot{-} y := \texttt{max}(0, x - y)$ denotes our subtraction on natural numbers:

$$x + y = \begin{cases} 2 * x \dot{-} (x \dot{-} y) \texttt{ if } y \dot{-} x = 0, \texttt{ i.e. } y \leq x \\ 2 * y \dot{-} (y \dot{-} x) \texttt{ otherwise} \end{cases}$$

Obviously also, subtraction and division are not enough to compute the successor function, and a fortiori the addition, because, for all integers x and $y > 0$, we have $x \dot{-} y \leq x$ and $x \texttt{ div } y \leq x$. More fully, we state the following negative results.

Proposition 5. *1. No RAM with subtraction* and *division over natural numbers computes the successor function.*

2. No RAM with multiplication computes the successor function.

Proof. As we have given the argument for item 1, it suffices to justify item 2. Suppose M is a RAM with multiplication which computes the function $N \mapsto N+1$. Let us fix an integer N greater than all the numbers $j_1, \ldots, j_k$ that appear in the program of M and such that $N+1$ is a prime number. By construction, each number that appears in the computation of M over the input N must be a product of integers belonging to the set $\{N, j_1, \ldots, j_k\}$ and therefore cannot be equal to $N+1$, because $N+1$ is prime and greater than all the factors, a contradiction. □

We therefore have $\mathcal{M}[\{\dot{-}, \times\}] = \mathcal{M}[\{+\}]$ but the RAM models $\mathcal{M}[\{\dot{-}, \mathtt{div}\}]$ and $\mathcal{M}[\{\times\}]$ are strictly weaker.

Also, we do not know if addition can be replaced by multiplication *and* division, i.e. if we have $\mathcal{M}[\{\times, \mathtt{div}\}] = \mathcal{M}[\{+\}]$, even if we think the answer is no.

The intuition comes from the ability to manage (dynamic or static) arrays using concatenation as we do with addition.

10 Final results, concluding remarks and open problems

We hope that the results established in the previous sections have convinced the reader that our RAM model with addition, also called *addition RAM*, or $\mathcal{M}[\{+\}]$, is the standard model for the design and analysis of algorithms. This means that the complexity classes defined in this model are robust and faithfully reproduce/model the intuitive complexity classes of concrete algorithms. This is already true for the "minimal" complexity classes defined as follows.

"Minimal" complexity classes of addition RAM

- $\mathtt{LIN}$ is the class of functions $\mathcal{I} \mapsto f(\mathcal{I})$ computed in *linear time* by an addition RAM.
- $\mathtt{CONST}_{\mathtt{lin}}$ is the class of "dynamic" problems $(\mathcal{I}, \mathcal{X}) \mapsto f(\mathcal{I}, \mathcal{X})$ computed in *constant time* by an addition RAM with *linear-time preprocessing*, which means:
  1. *Pre-computation:* from an input $\mathcal{I}$ of size N, the RAM computes an "index" $p(\mathcal{I})$ in time $O(N)$;
  2. *Computation:* from $p(\mathcal{I})$ and after reading a second input $\mathcal{X}$ of constant size, the RAM computes in *constant time* the output $f(\mathcal{I}, \mathcal{X})$.
- $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ (see [30, 8, 7, 29]) is the class of enumeration problems $\mathcal{I} \mapsto f(\mathcal{I})$ computed with *constant delay* and *constant-space* by an addition RAM with *linear-time preprocessing*, which means:
  1. *Pre-computation:* from an input $\mathcal{I}$ of size N, the RAM computes an "index" $p(\mathcal{I})$ in time $O(N)$;
  2. *Enumeration:* from $p(\mathcal{I})$ and using *constant space*, the RAM enumerates without repetition the elements of the set $f(\mathcal{I})$ with a *constant delay* between two successive elements and stops immediately after producing the last one.

Remark 19 (More liberal constant-delay complexity classes with linear-time preprocessing)**.** *In the literature on enumeration algorithms, the space complexity of the enumeration phase is often not accurately described or appears to be larger than our constant space bound, see e.g. [75, 3, 29]. We think that the most natural variants (extensions) of our* $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ *class are the following two ones:*

- *Let* $\mathtt{CONST\text{-}DELAY\text{-}LIN\text{-}SPACE}_{\mathtt{lin}}$ *denote the extension of the class* $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ *where the space bound of the enumeration phase is $O(N)$, instead of constant. Note that this complexity class can be defined in our RAM model which only uses $O(N)$ integers (contents and addresses of registers).*
- *One can also allow the constant-delay enumeration phase to use an unlimited memory, i.e. unlimited addresses, which is forbidden in our RAM model and seems unrealistic to implement.*

The operations we have studied. We have extended the RAM model with the following two kinds of operations:

- Let $\mathtt{Op}_{\mathtt{Arith}}$ be the set of arithmetic operations that follow: addition, subtraction, multiplication, division, exponential, logarithm, and c-th root (for any fixed integer $c \geq 2$), acting on "polynomial" integers, i.e. integers less than N^d, for a fixed d.
- Let $\mathtt{Op}_{\mathtt{CA}}^{\mathtt{lin}}$ be the set of operations on "polynomial" integers computable in linear time on cellular automata.

The following corollaries summarize our main results and establish the robustness and expressiveness of the complexity classes thus defined in our RAM model.

Corollary 4. *Let* $\mathtt{Op}$ *be a finite set of operations such that* $\{+\} \subseteq \mathtt{Op} \subseteq \mathtt{Op}_{\mathtt{Arith}} \cup \mathtt{Op}_{\mathtt{CA}}^{\mathtt{lin}}$*. Then we have* $\mathcal{M}[\mathtt{Op}] = \mathcal{M}[+]$.

Corollary 5. *The classes* $\mathtt{LIN}$, $\mathtt{CONST}_{\mathtt{lin}}$, $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ *and* $\mathtt{CONST\text{-}DELAY\text{-}LIN\text{-}SPACE}_{\mathtt{lin}}$ *do not change if the set of primitive operations of the RAMs is any finite set of operations containing addition and included in* $\mathtt{Op}_{\mathtt{Arith}} \cup \mathtt{Op}_{\mathtt{CA}}^{\mathtt{lin}}$.

Proof. Each of these four complexity classes allows linear-time preprocessing and we have proved that each operation in $\mathtt{Op}_{\mathtt{Arith}} \cup \mathtt{Op}_{\mathtt{CA}}^{\mathtt{lin}}$ can be computed by an addition RAM in constant time after linear-time preprocessing. □

Remark 20. *For simplicity, we have chosen to state Corollary 5 for the "minimal" complexity classes of the RAM model. However, it is easy to deduce the same result for complexity classes of greater time and/or greater space, e.g.* $\mathtt{Time}(N^2)$*, etc.*

Addition of randomness? Our RAM model is purely deterministic but we can easily add to our model a source of randomness, for instance by having an instruction `RANDOM` that fills the accumulator A (of an AB-RAM) with some random value (in $[0, cN[$, for some constant integer c).

Adding randomness to a RAM model does not change the worst case complexity of well-defined problems as, in the worst case, the random instruction might behave exactly like the instruction `CST` 0. For that reason, the addition of randomness is only useful when talking about complexity classes where the answer is allowed to be only *probably* correct (as opposed to *always* correct) or about programs terminating in *expected time* $T(N)$, for some function T.

In computational complexity theory, there are open conjectures about whether the addition of a source of randomness to Turing machines might make them exponentially faster. Since the classes of problems computable in polynomial or exponential time in our RAM or in a Turing machine are the same, there is little hope to solve whether a RAM machine with randomness is equivalent to a RAM without randomness.

Weakness of the RAM with successor and predecessor? Let $\mathtt{LIN}^{\mathtt{succ,pred}}$, $\mathtt{CONST}_{\mathtt{lin}}^{\mathtt{succ,pred}}$, and $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}^{\mathtt{succ,pred}}$ denote the complexity classes similar to the $\mathtt{LIN}$, $\mathtt{CONST}_{\mathtt{lin}}$, and $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ classes, respectively, when the successor and predecessor functions, $x \mapsto x+1$ and $x \mapsto x-1$, where $x < cN$, for some constant integer $c > 0$, are the only primitive operations of the RAM model instead of the addition operation.

In contrast with Corollary 5, the following "negative" result holds.

Corollary 6. *We have the strict inclusion* $\mathtt{CONST}_{\mathtt{lin}}^{\mathtt{succ,pred}} \subsetneq \mathtt{CONST}_{\mathtt{lin}}$.

Proof. The inclusion is a direct consequence of what we have proved before. So, it suffices to prove that it is strict. Let us consider the "dynamic" problem $(\mathcal{I}, \mathcal{X} = (x, y)) \mapsto x + y$, for $x, y \in [0, N[$, where N is the size of $\mathcal{I}$. This problem trivially belongs to $\mathtt{CONST}_{\mathtt{lin}}$ but it does not belongs to $\mathtt{CONST}_{\mathtt{lin}}^{\mathtt{succ,pred}}$ by Proposition 4. □

We ignore if Corollary 6 can be "extended" to the other "minimal" complexity classes.

Open problem 2. *Is the inclusion* $\mathtt{LIN}^{\mathtt{succ,pred}} \subseteq \mathtt{LIN}$ *strict?*

Open problem 3. *Is the inclusion* $\mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}^{\mathtt{succ,pred}} \subseteq \mathtt{CONST\text{-}DELAY}_{\mathtt{lin}}$ *strict?*

The difficulty comes from the fact that RAM with successor, also known as successor RAM, is much more powerful than it seems at first glance. As a striking example, Schönhage [70] showed as early as 1980 that integer-multiplication, which is the problem of computing the product of two integers in binary notation, i.e., with one bit per input register [14], is computed in linear time on a successor RAM and therefore belongs to $\texttt{LIN}^{\texttt{succ,pred}}$. Moreover, Schönhage [70] proved that the successor RAM is equivalent to what he calls the "Storage Modification Machine", also called "Pointer Machine" by Knuth [54] and Tarjan [80], who proves that in this computational model, Union-Find operations can be performed in amortized quasi-constant time $\alpha(n)$, where $\alpha(n)$ is the inverse of Ackermann's function $A(n,n)$.

Why does our RAM model the algorithms in the literature? Our RAM model reproduces/models exactly the computations of the algorithms on the combinatorial structures. The reason is threefold:

1. Such a structure $\mathcal{S}$, e.g. a tree or a graph, is naturally encoded by a RAM input $\mathcal{I} = (N, I[0], \ldots, I[N-1])$, with $I[j] = O(N)$ and $N = \Theta(L)$, where L is the "natural length" of the structure $\mathcal{S}$, which is $L = n$ if $\mathcal{S}$ is a tree of n nodes, or $L = m + n$ if $\mathcal{S}$ is a graph of m edges and n vertices, etc.

2. The input array $\mathcal{I} = (N, I[0], \ldots, I[N-1])$ is of the same nature as the array $R[0], R[1], \ldots$ of RAM registers of contents also $O(N)$.

3. The RAM functioning is also homogeneous: a read/write instruction is a random access instruction like the other instructions.

Why RAM linear time is intuitive linear time under unit cost criterion? By our conventions, any *linear-time* (and linear-space[15]) algorithm from the literature on combinatorial structures (trees, graphs, etc.) is modeled/implemented by a RAM algorithm that uses $O(N)$ registers and executes $O(N)$ instructions, each involving $O(1)$ registers of bit length $\Theta(\log N)$. Consequently, its time complexity under unit cost criterion is $O(N)$, which is $O(L)$, where L is the "natural length" of the input structure, by item 1 of the above paragraph.

What about linear time under logarithmic cost criterion? Since a linear-time algorithm performs $O(N)$ instructions, each of cost $O(\log N)$, its time is $O(N \log N)$ under logarithmic cost criterion. Is it a linear time for this criterion?

Linear-time algorithms on combinatorial structures are also linear under logarithmic cost criterion. Let us justify this assertion for graph algorithms. (The proof is similar for algorithms on other structures: trees, formulas, etc.) A graph without isolated vertex whose vertices are $1, \ldots, n$ and edges are $(a_1, b_1), \ldots, (a_m, b_m)$ is "naturally represented" by the standard input $\mathcal{I} = (N, I[0], \ldots, I[N-1])$ where $N \coloneqq 2m + 2$, $I[0] \coloneqq m$, $I[1] \coloneqq n$, and finally $I[2j] \coloneqq a_j$ and $I[2j+1] \coloneqq b_j$ for $j = 1, \ldots, m$. Recall that we have $n \leq 2m = N - 2$.

If $\mathcal{I}$ is encoded by a binary word, its bit length, called $\textsc{length}(\mathcal{I})$, is $\Theta(\textsc{length}(N) + \sum_{j=1}^{m} \textsc{length}(\textsc{binarycode}(a_j, b_j)))$, which is $O(N \log N)$. On the other hand, since the m edges (a_j, b_j) are pairwise distinct, we have

$$\sum_{j=1}^{m} \textsc{length}(\textsc{binarycode}(a_j, b_j)) \geq \sum_{j=1}^{m} \textsc{length}(w_j) \tag{17}$$

where $w_1, \ldots, w_m$ is the list of the m shortest nonempty binary words numbered in increasing length, that means $\textsc{length}(w_j) \leq \textsc{length}(w_{j+1})$, for each $j < m$. The (in)equation (17) is justified by the following points:

- without loss of generality, assume that the m edges $(a_1, b_1), \ldots, (a_m, b_m)$ are numbered in increasing length of their (pairwise distinct) encodings $\textsc{binarycode}(a_j, b_j)$, $j = 1, \ldots, m$;

- consequently, we have $\textsc{length}(\textsc{binarycode}(a_j, b_j)) \geq \textsc{length}(w_j)$, for each $j \leq m$.

[14] Note that in our formalism, the input of this problem is of the form $\mathcal{I} \coloneqq (N, I[0] = a_0, \ldots, I[n-1] = a_{n-1}, I[n] = 2, I[n+1] = b_0, \ldots, I[2n] = b_{n-1})$, where $N = 2n + 1$ and the a_i, b_i belongs to $\{0, 1\}$, and its output is $(2n, p_0, \ldots, p_{2n-1})$, where $p = a \times b$, for $a = \overline{a_{n-1} \ldots a_1 a_0}$, $b = \overline{b_{n-1} \ldots b_1 b_0}$, and $p = \overline{p_{2n-1} \ldots p_1 p_0}$.

[15] As is the case with almost all linear-time algorithms in the literature and as required for RAMs, we require that a linear-time algorithm use linear space.

Moreover, the (in)equality $\sum_{j=1}^{m} \text{LENGTH}(w_j) = \Omega(m \log m)$ is intuitive and not hard to prove[16]. By (17), this implies $\text{LENGTH}(\mathcal{I}) = \Omega(m \log m) = \Omega(N \log N)$. Thus, we have proved[17]

$$\text{LENGTH}(\mathcal{I}) = \Theta(N \log N) \tag{18}$$

Consequently, a graph algorithm running in linear time under unit cost criterion, therefore in time $O(N \log N)$ under logarithmic cost criterion, runs in time $O(\text{LENGTH}(\mathcal{I}))$ under the same criterion, according to equation (18). This is the true linear time under the unit cost criterion!

Thus, algorithms on graphs or similar combinatorial structures (trees, formulas, hypergraphs, etc.) running in linear time under unit cost criterion are also running in linear time, that is to say in time $O(\text{LENGTH}(\mathcal{S}))$, under logarithmic cost criterion, where $\text{LENGTH}(\mathcal{S})$ is the bit length of the input structure $\mathcal{S}$.

However, our above argument for combinatorial structures does *not* apply to *word structures*.

What linear time complexity for algorithms on words? As seems most natural, it is the rule for a text algorithm (see e.g. the reference book [25] or Chapter 32 of [23]) to represent an input word (text, string) $w = w_0 \dots w_{n-1} \in \Sigma^n$ over a finite fixed alphabet Σ, identified with the set of integers $\{0, 1, \dots, b-1\}$ for $b = \texttt{card}(\Sigma)$, by an array $\texttt{W}[0..n-1]$ where $\texttt{W}[j] = w_j$. Thus, the text algorithm is faithfully simulated by a RAM whose input is $\mathcal{I} = (N, I[0], \dots, I[N-1])$ where $N \coloneqq n$ and $I[j] \coloneqq w_j$; this means $0 \leq I[j] < b = O(1)$.

Many word algorithms of the literature, e.g. the well-known string-matching algorithm of Knuth, Morris and Pratt [55, 25, 47], are executed in the RAM model in time $O(N)$ under unit cost criterion, where N is the length of the input word w (or the maximum of the lengths of the two input words). Therefore, such an algorithm runs in time $O(N \log N)$ under logarithmic cost criterion, for $N = \text{LENGTH}(w)$. A priori, it is not $O(\text{LENGTH}(w))$... This corresponds to the fact that the input elements $I[j]$ are in $O(1)$ while the contents of the memory registers are in $O(N)$: here, there is *heterogeneity between input and memory*.

To recover homogeneity, we can adopt a technical solution already introduced in [40, 71, 42]: partition any input word $w \in \Sigma^n$ (where $\Sigma = \{0, \dots, b-1\}$) into $N = \Theta(n/\log n)$ factors of length $\Theta(\log n) = \Theta(\log N)$. More precisely, using the parameters $\ell \coloneqq \lceil (\log_b n)/2 \rceil$ and $N \coloneqq \lceil n/\ell \rceil$, we partition w into the concatenation $w = f_0 \dots f_{N-1}$ of N factors f_j, with $\text{LENGTH}(f_j) = \ell$ for $j < N-1$ and $1 \leq \text{LENGTH}(f_{N-1}) \leq \ell$. The word $w \in \Sigma^n$ is represented by the input $\mathcal{I} = (N, I[0], \dots, I[N-1])$ defined by $I[j] \coloneqq f_j$ for each $j < N$. Many other variants of this representation can also be adopted, the important point being the orders of magnitude obtained:

$$N = \Theta(n/\log n) \ ; \ I[j] = O(b^{\ell}) = O(b^{(\log_b n)/2}) = O(n^{1/2}) = O(N), \text{ for each } j < N.$$

With such a representation of words, the time complexity of a RAM operating in time $O(N)$, which is in $O(n/\log n)$, under unit cost criterion, is $O(N \log N) = O(n)$, that is to say *really* linear, under logarithmic cost criterion. It would be a vast and challenging objective to determine which word problems known to be computable in "linear time" can be computed in time $O(n/\log n)$ under unit cost criterion using our word representation. For example, paper [40] proves that it is true for the following "sorting" problem from $(\Sigma \cup \{\sharp\})^+$ to $(\Sigma \cup \{\sharp\})^+$, for $\sharp \notin \Sigma$, defined as

$$x_1 \sharp x_2 \sharp \cdots \sharp x_q \mapsto x_{\pi(1)} \sharp x_{\pi(2)} \sharp \cdots \sharp x_{\pi(q)}$$

where $x_1, x_2, \dots, x_q$ are words in Σ^+, and π is a permutation of $\{1, 2, \dots, q\}$ such that $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(q)}$ for the lexicographic order $\leq$ of Σ^+.

Is multiplication more difficult than addition in the RAM model? We mentioned above that Schönhage [70] surprisingly proved that the successor RAM, equivalent to his "storage modification machine" (but seemingly weaker than our addition RAM!) can compute in time $O(n)$ the product of two integers of length n in binary notation, each contained in n input registers (one bit per input register).

---

[16] Note the sequence of equivalences $\text{LENGTH}(w_j) = \ell \iff \sum_{i=1}^{\ell-1} 2^i < j \leq \sum_{i=1}^{\ell} 2^i \iff 2^{\ell} - 2 < j \leq 2^{\ell+1} - 2 \iff 2^{\ell} < j + 2 \leq 2^{\ell+1} \iff \ell = \lceil \log_2(j+2) \rceil - 1$. Hence, $\text{LENGTH}(w_j) = \lceil \log_2(j+2) \rceil - 1$. Therefore, we get $\sum_{j=1}^{m} \text{LENGTH}(w_j) \geq \sum_{j=\lceil m/2 \rceil}^{m} \text{LENGTH}(w_j) \geq (m/2)\text{LENGTH}(w_{\lceil m/2 \rceil}) = (m/2) \lceil \log_2(\lceil m/2 \rceil + 2) \rceil - 1$, from which $\sum_{j=1}^{m} \text{LENGTH}(w_j) = \Omega(m \log m)$ is deduced.

[17] Note that the truth of equation (18) is largely independent of the representation of the graph provided it is "natural". E.g., if the vertices of the graph are $1, \dots, n$, some of which may be isolated, and its edges are $(a_1, b_1), \dots, (a_m, b_m)$, the graph is "naturally" represented by the standard input $\mathcal{I} = (N, I[0], \dots, I[N-1])$ where $N \coloneqq 2m + n + 1$, $I[0] \coloneqq m$, $I[j] \coloneqq j$ for $j = 1, \dots, n$, and finally $I[2j + n - 1] \coloneqq a_j$ and $I[2j + n] \coloneqq b_j$ for $j = 1, \dots, m$. We still have $\text{LENGTH}(\mathcal{I}) = O(N \log N)$. Moreover, by the same reasoning as above, we get $\text{LENGTH}(\mathcal{I}) = \Omega(n \log n + m \log m) = \Omega(N \log N)$ (because $N = \Theta(max(m, n))$) and therefore $\text{LENGTH}(\mathcal{I}) = \Theta(N \log N)$.

On this point, in his lecture on receiving the ACM Turing award [21] in 1982, Stephen Cook expressed his astonishment as follows: " Schönhage [70] recently showed..... that his storage modification machines can multiply in time $O(n)$ (linear time!). We are forced to conclude that either multiplication is easier than we thought or that Schönhage's machines cheat."

Although we agree that Schönhage's result is very beautiful and strong (it is all the more powerful that it uses a successor RAM instead of an addition RAM!), we believe that the paradox highlighted by Cook is only apparent. The product of two integers of binary length n is computed by a successor RAM in time $O(n)$ under unit cost criterion, and therefore in time $O(n \log n)$ under logarithmic cost criterion. This is not linear time!

At the opposite, we have noticed, see Remark 8, that our addition RAM can compute the sum (resp. difference) of two integers of binary length n in time $O(n/\log n)$ under unit cost criterion (by representing each input integer by the array of binary words of length $\lceil (\log_2 n)/2 \rceil$ whose concatenation is its binary notation, like the word representation we suggest in the previous paragraph). This is $O(n)$ time under logarithmic cost criterion, which is really linear time. This is not the case with the Schönhage multiplication algorithm or the Fast Fourier multiplication and we conjecture that our RAM model cannot compute the product of two integers of binary length n in time $O(n/\log n)$ under unit cost criterion as its does for their sum and difference.

Multiplication continues to seem harder than addition and subtraction!

Linear space or larger space? The RAM model we have chosen uses *linear space*. This means that the addresses and contents of registers $R[0], R[1], \ldots$ are integers in $O(N)$; in particular, no k-dimensional array $N_1 \times N_2 \times \cdots \times N_k$ is allowed unless it uses $N_1 \times N_2 \times \cdots \times N_k = O(N)$ registers so that it can be represented by a one-dimensional array of size $O(N)$. This is justified by two arguments:

- our RAM model models the vast majority of linear-time algorithms in the literature: they use a linear space;
- our RAM model is very "robust": its complexity classes are invariant for many variations, in particular with respect to the primitive operations of the RAM, as established at length in this paper.

Nevertheless, we have noticed that several recent papers about queries in logic and databases, see e.g. [31, 75, 12, 13], present algorithms which use k-dimensional arrays $\mathbf{A}$ with $O(N^k)$ available registers, such that for given $(n_1, \ldots, n_k) \in \mathbb{N}^k$, with each $n_i = O(N)$, the entry $\mathbf{A}[n_1, \ldots, n_k]$ at position $(n_1, \ldots, n_k)$ can be accessed in constant time. Although this "multidimensional" RAM model is "unrealistic for real-world computers", as [12] writes, it is robust in two respects:

- clearly, it inherits all the invariance properties of our (stricter) RAM model according to the primitive operations;
- it is invariant according to the dimensions of arrays: Theorem 1 states that the RAM model with arrays of arbitrary dimensions is equivalent to the RAM model using only 2-dimensional arrays, and even, arrays of dimensions $N \times N^\epsilon$, for any $\epsilon > 0$, hence with $O(N^{1+\epsilon})$ registers available.

There remains the open and difficult question whether the RAM model with k-dimensional arrays but only using $O(N)$ cells is equivalent to the RAM model with only 1-dimensional arrays of $O(N)$ size. Answering this question is crucial in determining whether the algorithms of [31, 75, 12, 13] can be implemented in linear space in "real-world computers".

What space complexity for enumeration algorithms? In accordance with and in the spirit of the definition of the complexity class $\texttt{CONST}_{\texttt{lin}}$, whose second phase uses constant time and therefore constant space, the "minimal" enumeration complexity class that we have chosen as the standard (when each solution to be enumerated is of constant size[18]) is the class $\texttt{CONST-DELAY}_{\texttt{lin}}$ whose second phase uses constant time *and also* constant space. This is justified by two points:

- some basic enumeration algorithms in combinatorics, logic and database theory [34, 8, 30, 7, 29] belong to $\texttt{CONST-DELAY}_{\texttt{lin}}$;

[18] Note that if the solutions are not of constant size, e.g. if they are subsets of vertices of an input graph, then the "minimal" (standard) enumeration complexity class is the class of problems, called $\texttt{LIN-DELAY}_{\texttt{lin}}$, whose preprocessing phase uses $O(N)$ time and space and enumeration phase uses $O(|S|)$ delay and space, for each solution S, see [6, 7, 24]. Of course, belonging to $\texttt{LIN-DELAY}_{\texttt{lin}}$ amounts to belonging to $\texttt{CONST-DELAY}_{\texttt{lin}}$ for an enumeration problem whose size of solutions is constant.

- this complexity class is as "robust" (and for the same reasons) as the classes LIN and $\texttt{CONST}_{\texttt{lin}}$.

The a priori larger complexity class $\texttt{CONST-DELAY-LIN-SPACE}_{\texttt{lin}}$, which allows the enumeration phase to use $O(N)$ space, instead of constant space, is also realistic (its two phases use a linear space) and as robust as $\texttt{CONST-DELAY}_{\texttt{lin}}$.

Again, the question of whether the inclusion $\texttt{CONST-DELAY}_{\texttt{lin}} \subseteq \texttt{CONST-DELAY-LIN-SPACE}_{\texttt{lin}}$ is strict remains an open and difficult problem.

Which operations are computed in constant time? We have the feeling that our addition RAM can compute in constant time (with linear-time preprocessing) any operation on "polynomial" integers defined from the usual arithmetic operations (whose versions on $\mathbb{R}$ are continuous), plus the rounding functions $x \mapsto \lfloor x \rfloor$ and $x \mapsto \lceil x \rceil$, and the compositions of such operations, provided that the intermediate results are "polynomial". We managed to prove this for all the operations we studied in Sections 4 to 7, except for the general root operation $(x, y) \mapsto \lfloor x^{1/y} \rfloor$ for which Subsection 7.3 gives only a partial proof. However, we have failed to prove any general result – except for the set of linear-time computable operations on cellular automata — and our proofs are "ad hoc" even though a discrete analog of Newton's approximation method used in Section 7 seems promising for establishing such a systematic result.

Final remarks on the RAM model. As observed and discussed in the introduction to this paper, on the one hand the RAM model is generally considered to be the standard model in the foundations of algorithm design and algorithm analysis, and on the other hand, surprisingly, the literature does not agree on a standard definition of the RAM model and its complexity classes.

Although in the past some papers [39, 38, 40, 42, 41, 30, 7] have defined and started to study the LIN class of our RAM model with addition and its subclass of decision problems called DLIN, so far no systematic work has investigated from scratch the robustness and extent of computational power of this model and its complexity classes.

Having such a systematic and unified study was all the more urgent as the many recent studies on the algorithms and the complexity of "dynamic" problems, and especially of enumeration problems [74, 75, 3, 14, 53, 79, 20, 11, 29], essentially refer to the reference book [2] (excellent but almost half a century old!) and to the scattered and incomplete papers cited above. This gives an impression of imprecision and approximation and can lead to ambiguities or even errors[19].

We hope that this self-contained article meets this need for precision. We have designed our paper as a progressive, detailed and systematic study.

We consider that the main arguments in favor of adopting the addition RAM as the basic algorithmic model are as follows:

- its simplicity/minimality and its homogeneity;
- its ability to represent usual data structures and in particular tables, essential for preprocessing;
- the robustness and computing power of its complexity classes (Corollary 5): the most important and surprising result that this paper establishes is that the addition RAM can compute the Euclidean division in constant time with linear-time preprocessing; as we also proved, this allows to compute many other arithmetic operations on "polynomial" integers in constant time with linear-time preprocessing.

Open problems. Even if we have positively solved in this paper the main (old) open question, which is the equivalence of the addition RAM with the RAM equipped with the four operations $+, -, \times$ and / acting on "polynomial" integers, for the complexity classes under the unit cost criterion, there remain many other interesting open questions regarding the RAM model, several of which have been presented throughout this paper.

We have tried to establish the most complete list of integer operations that an addition RAM can compute in constant time with linear-time preprocessing. In fact, many algorithms deal with combinatorial structures such as trees, circuits, graphs, etc. The next step would therefore be to undertake a similar systematic study of operations on combinatorial structures computable in constant time (or quasi-constant time) on addition RAMs with linear-time preprocessing. For example, although a lot of work has been done in this direction, it would be very useful to establish

[19] Most of the papers cite the book [2] which considers the four operations $+, -, \times$ and / as primitive operations of RAMs without limiting the values of their operands. However, Schönhage [69] proved that the class of languages recognized by such RAMs in polynomial time under the unit cost criterion contains the entire class NP!

the "largest" set of "atomic" tree operations computable in constant (or quasi-constant) time after a linear (or quasi-linear) time preprocessing[20]. This would constitute a toolbox to optimize algorithms for "dynamic" problems such as those studied in [3, 12, 15, 14, 4].

References

[1] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 204–216. IEEE Computer Society, 1987.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 111:1–111:15, 2017.

[4] Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. *CoRR*, abs/2102.07728, 2021.

[5] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979.

[6] Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[7] Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries)*. PhD thesis, University of Caen Normandy, France, 2009.

[8] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[9] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *ITA*, 42(1):147–164, 2008.

[10] José L. Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1990.

[11] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

[12] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. *CoRR*, abs/1702.06370, 2017.

[13] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD Queries Under Updates on Bounded Degree Databases. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory (ICDT 2017)*, volume 68 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018.

[20] For instance, as soon as 1984, Harel and Tarjan [48] proved that the closest common ancestor of any two nodes x, y of a tree T of N nodes can be computed in constant time after a preprocessing of time $O(N)$ and depending only on T.

[15] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, volume 98 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[16] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms I: axiomatization. *Log. Methods Comput. Sci.*, 3(4), 2007.

[17] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms II: abstract state machines and the characterization theorem. *Log. Methods Comput. Sci.*, 3(4), 2007.

[18] Egon Börger and James K. Huggins. Abstract state machines 1988-1998: Commented ASM bibliography. *Bull. EATCS*, 64, 1998.

[19] Marco Cadoli and Francesco M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1998.

[20] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 393–409. ACM, 2020.

[21] Stephen A. Cook. An overview of computational complexity. *Commun. ACM*, 26(6):400–408, 1983.

[22] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[24] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discret. Appl. Math.*, 157(12):2675–2700, 2009.

[25] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[26] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[27] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[28] Scott D. Dexter, Patrick Doyle, and Yuri Gurevich. Gurevich abstract state machines and schoenhage storage modification machines. *J. Univers. Comput. Sci.*, 3(4):279–303, 1997.

[29] Arnaud Durand. Fine-grained complexity analysis of queries: From decision to counting and enumeration. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 331–346. ACM, 2020.

[30] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.

[31] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 121–131. ACM, 2014.

[32] Calvin C. Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. *J. ACM*, 11(4):365–399, 1964.

[33] Shimon Even. Systolic modular multiplication. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 619–624. Springer, 1990.

[34] Tomás Feder. Network flow and 2-satisfiability. *Algorithmica*, 11(3):291–319, 1994.

[35] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[36] Lakshmi N. Goyal. A note on atrubin's real-time iterative multiplier. *IEEE Trans. Computers*, 25(5):546–548, 1976.

[37] Anaël Grandjean, Gaétan Richard, and Véronique Terrier. Linear functional classes over cellular automata. In Enrico Formenti, editor, *Proceedings 18th international workshop on Cellular Automata and Discrete Complex Systems and 3rd international symposium Journées Automates Cellulaires, AUTOMATA & JAC 2012, La Marana, Corsica, September 19-21, 2012*, volume 90 of *EPTCS*, pages 177–193, 2012.

[38] Etienne Grandjean. Invariance properties of rams and linear time. *Computational Complexity*, 4:62–106, 1994.

[39] Etienne Grandjean. Linear time algorithms and np-complete problems. *SIAM J. Comput.*, 23(3):573–597, 1994.

[40] Etienne Grandjean. Sorting, linear time and the satisfiability problem. *Ann. Math. Artif. Intell.*, 16:183–236, 1996.

[41] Etienne Grandjean and Frédéric Olive. Graph properties checkable in linear time in the number of vertices. *J. Comput. Syst. Sci.*, 68(3):546–597, 2004.

[42] Etienne Grandjean and Thomas Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. *SIAM J. Comput.*, 32(1):196–230, 2002.

[43] Yuri Gurevich. Evolving algebras: an attempt to discover semantics. In Grzegorz Rozenberg and Arto Salomaa, editors, *Current Trends in Theoretical Computer Science - Essays and Tutorials*, volume 40 of *World Scientific Series in Computer Science*, pages 266–292. World Scientific, 1993.

[44] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.*, 1(1):77–111, 2000.

[45] Yuri Gurevich. What is an algorithm? In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*, pages 31–42. Springer, 2012.

[46] Yuri Gurevich and Saharon Shelah. Nearly linear time. In Albert R. Meyer and Michael A. Taitslin, editors, *Logic at Botik '89, Proceedings*, volume 363 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 1989.

[47] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[48] David Harel and Robert Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[49] Juris Hartmanis. Computational complexity of random acess stored program machines. *Math. Syst. Theory*, 5(3):232–245, 1971.

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

[51] Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Log. Methods Comput. Sci.*, 7(2), 2011.

[52] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):25:1–25:12, 2013.

[53] Wojciech Kazana and Luc Segoufin. First-order queries on classes of structures with bounded expansion. *Log. Methods Comput. Sci.*, 16(1), 2020.

[54] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[55] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[56] Dexter C. Kozen. *Design and Analysis of Algorithms*. Texts and Monographs in Computer Science. Springer, 1992.

[57] Clemens Lautemann and Bernhard Weinzinger. Monadic nlin and quantifier-free reductions. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 322–337. Springer, 1999.

[58] Pierre Marquis. Compile! In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4112–4118. AAAI Press, 2015.

[59] Jacques Mazoyer. An overview of the firing squad synchronization problem. In Christian Choffrut, editor, *Automata Networks, LITP Spring School on Theoretical Computer Science, Angelès-Village, France, May 12-16, 1986, Proceedings*, volume 316 of *Lecture Notes in Computer Science*, pages 82–94. Springer, 1986.

[60] Jacques Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theor. Comput. Sci.*, 50:183–238, 1987.

[61] Jacques Mazoyer. On optimal solutions to the firing squad synchronization problem. *Theor. Comput. Sci.*, 168(2):367–404, 1996.

[62] Jacques Mazoyer and Jean-Baptiste Yunès. Computations on cellular automata. In Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 159–188. Springer, 2012.

[63] Bernard ME Moret and Henry D Shapiro. *Algorithms from P to NP (vol. 1) design and efficiency*. Benjamin-Cummings Publishing Co., Inc., 1991.

[64] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[65] Rolf Niedermeier. *Invitation to fixed-parameter algorithms*. Oxford Lecture Series in Mathematics and its Applications 31, 2006.

[66] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[67] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[68] Kenneth W. Regan. Linear time and memory-efficient computation. *SIAM J. Comput.*, 25(1):133–168, 1996.

[69] Arnold Schönhage. On the power of random access machines. In Hermann A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979.

[70] Arnold Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[71] Thomas Schwentick. Algebraic and logical characterizations of deterministic linear time classes. In Rüdiger Reischuk and Michel Morvan, editors, *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany, February 27 - March 1, 1997, Proceedings*, volume 1200 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 1997.

[72] Thomas Schwentick. Descriptive complexity, lower bounds and linear time. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1584 of *Lecture Notes in Computer Science*, pages 9–28. Springer, 1998.

[73] Robert Sedgewick and Kevin Wayne. *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016.

[74] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 10–20, 2013.

[75] Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, pages 13–27, 2014.

[76] John C. Shepherdson and Howard E. Sturgis. Computability of recursive functions. *J. ACM*, 10(2):217–255, 1963.

[77] Steven Skiena. *The Algorithm Design Manual, Third Edition*. Texts in Computer Science. Springer, 2020.

[78] Alvy Ray Smith. Simple computation-universal cellular spaces. *J. ACM*, 18(2):339–353, 1971.

[79] Yann Strozecki. Enumeration complexity. *Bull. EATCS*, 129, 2019.

[80] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[81] Robert Endre Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, 1983.

[82] Véronique Terrier. Language recognition by cellular automata. In Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 123–158. Springer, 2012.

[83] Takeaki Uno. Constant time enumeration by amortization. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 593–605. Springer, 2015.

[84] Klaus Wagner and Gerd Wechsung. *Computational Complexity*. Springer, 1986.

[85] Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016.

[86] Wikipedia. Church-turing thesis. `https://en.wikipedia.org/wiki/Church-Turing_thesis`.

[87] Wikipedia. Random access machine. `https://en.wikipedia.org/wiki/Random-access_machine`.

11 Appendix: Proof that an operation computed in linear time on a cellular automaton can be computed in constant time on a RAM

In this appendix, we prove Theorem 3, which we recall below.

Theorem 3. *Let $d \geq 1$ be a constant integer and $\mathtt{op} : \mathbb{N}^r \to \mathbb{N}$ an operation of arity $r \geq 1$, computable in linear time on a cellular automaton (CA). There is a RAM with addition, such that, for any input integer N:*

1. Pre-computation: *the RAM computes some tables in time $O(N)$;*
2. Operation: *using these pre-computed tables and reading any integers $X_1, \ldots, X_r < N^d$, the RAM computes in constant time the integer $\mathtt{op}(X_1, \ldots, X_r)$.*

Proof. Let $\mathcal{A} = (Q, \delta)$ be a CA which computes the operation $\mathtt{op}$, i.e. the function $f_{\mathtt{op}} : X_1; \ldots; X_r \mapsto \mathtt{op}(X_1, \ldots, X_r)$, in linear time. Before building a RAM that simulates the CA, let us slightly tweak the CA input configuration. In particular, we add the integer N to the input as a reference to the upper bound N^d of the operands $X_1, \ldots, X_r < N^d$.

Let us represent the $(r+1)$-tuple of integers $(N, X_1, \ldots, X_r)$, for $X_1, \ldots, X_r < N^d$, by the word, called $\mathtt{code}(N, X_1, \ldots, X_r) \coloneqq u_{L-1} \ldots u_1 u_0 \in \Gamma^L$, on the alphabet $\Gamma \coloneqq \{0, 1, \ldots, \gamma - 1\}$, where $\gamma \coloneqq 2^{r+1}$, which is defined as follows:

Length of the code: $L \coloneqq \texttt{length}(N^d)$;

Letters of the code: $u_j \coloneqq \sum_{i=0}^{r} x_{i,j} 2^i$, for $0 \leq j < L$, where $x_{i,L-1} \dots x_{i,1} x_{i,0}$ denotes the binary representation of X_i on L bits, for $i \in \{0, 1, \dots, r\}$ and $X_0 \coloneqq N$. In other words, u_j is the integer whose binary representation is column j of the 0/1 matrix $(x_{i,j})_{0 \leq i \leq r, 0 \leq j < L}$ when the columns are numbered from $L-1$ (left) downto 0 (right).

Example: For $r = 2$, $d = 1$, and the size integer $N = 13$, which has the binary representation 1101 of length $L = 4$, and for $X_1 = 7$ and $X_2 = 9$, which have the binary representations (of length $L = 4$) 0111 and 1001, respectively, we obtain $\gamma = 2^{r+1} = 8$, $\Gamma = \{0, 1, \dots, \gamma - 1 = 7\}$ and $\texttt{code}(N, X_1, \dots, X_r) = 5327 \in \Gamma^4$, see Table 12.

$X_2 = 9$	1	0	0	1
$X_1 = 7$	0	1	1	1
$X_0 = N = 13$	1	1	0	1
$\texttt{code}(N, X_1, X_2)$	$u_3 = 5$	$u_2 = 3$	$u_1 = 2$	$u_0 = 7$

Table 12: Encoding the input $(N, X_1, \dots, X_r)$

It is tedious but easy to build a CA which computes the function $g : \texttt{code}(N, X_1, \dots, X_r) \mapsto X_1; \dots; X_r$ in linear time. Since the class of functions computable in linear time on cellular automata is closed under composition (see Remark 10 in [37]), the composition of functions $f_{\texttt{op}} \circ g : \texttt{code}(N, X_1, \dots, X_r) \mapsto \texttt{op}(X_1, \dots, X_r)$ is computed in linear time on a cellular automaton, also called, for simplicity, $\mathcal{A} = (Q, \delta)$. By Definition 9, it means that there are a constant integer $c \geq 1$ and a projection $\pi : Q \to \{0, 1\}$ such that, for all integers $N, X_1, \dots, X_r$ with $X_j < N^d$, for $1 \leq j \leq r$, we have $\delta^{cL}(\sharp \texttt{Input} \sharp) = \sharp Y \sharp$,

- for $\texttt{Input} \coloneqq \texttt{code}(N, X_1, \dots, X_r)$ and $L \coloneqq \texttt{length}(\texttt{Input}) = \texttt{length}(N^d)$,
- and for a word $Y = y_{m-1} \dots y_0 \in (Q \setminus \{\sharp\})^+$ such that $\texttt{Output} \coloneqq \pi(y_{m-1}) \dots \pi(y_0) \in \{0, 1\}^+$, where $\texttt{Output}$ is the binary notation of the integer $\texttt{op}(X_1, \dots, X_r)$, written with m bits, possibly including leading zeros.

We can assume $Q \coloneqq \{0, 1, \dots s-1\}$ with $\Gamma \coloneqq \{0, 1, \dots, \gamma - 1\}$, $\gamma \coloneqq 2^{r+1}$, $\sharp \coloneqq \gamma$, and $\Gamma \cup \{\sharp\} \subseteq Q$, which implies $s \geq \gamma + 1$.

A cell of the CA is said to be *active* in the computation $\sharp \texttt{Input} \sharp, \delta(\sharp \texttt{Input} \sharp), \dots, \delta^{cL}(\sharp \texttt{Input} \sharp)$ if its state is different from $\sharp$ at at least one time $t \in \{0, 1, \dots, cL\}$ of the computation. It is clear that the set of active cells forms an interval, called the *active interval* of the computation, which, without loss of generality, can be assumed to be of length cL. For convenience, we number the active cells $cL - 1, \dots, 1, 0$, increasing from right to left (like the representation of integers in any base). Thus, at any time $t \in [0, cL]$ of the computation, we consider that the configuration has the form $\sharp q_{cL-1} \dots q_1 q_0 \sharp$, where $q_i \in Q$ is the state of cell i (possibly $\sharp$) at time t. We also assume that in the last configuration $\delta^{cL}(\sharp \texttt{Input} \sharp) = \sharp Y \sharp$, we have $Y \coloneqq y_{cL-1} \dots y_0$: this means that the output integer $\texttt{op}(X_1, \dots, X_r)$ is written $\pi(y_{cL-1}) \dots \pi(y_0) \in \{0, 1\}^{cL}$, with exactly cL bits.

We are now ready to present the simulation of the CA by a RAM. Before describing it with some details, let us first give the guiding ideas of this simulation.

A simplified view of the simulation. The trick is to divide the active interval of the computation, of length cL, into a constant number of consecutive blocks of "small" length ℓ, called ℓ-blocks, numbered $\dots 2, 1, 0$ (increasing from right to left!). We take $\ell = \epsilon \log_2 N$ for a "small" constant ϵ. (Precisely, we will define $\ell \coloneqq \lceil L/D \rceil$, for a large fixed integer D.) Similarly, we divide the computation, of time cL, into a constant number of computation intervals of duration ℓ, called "global ℓ-transitions".

By abuse of language, the string of states $B_j^t \in Q^\ell$ of an ℓ-block at some time t is called an ℓ-block at time t. Therefore, the configuration at any time t is the concatenation $\sharp B_{c_0}^t \dots B_1^t \sharp$, for the constant $c_0 \coloneqq \lceil cL/\ell \rceil$.

The essential point is the "locality" of the cellular automaton: the state of any cell x at any time $t+1$ (resp. $t+\ell$) is entirely determined by the states of the three cells $x-1, x, x+1$ (resp. the state string of the cell interval $[x-\ell, x+\ell]$) at time t. As a consequence, an ℓ-block $B_j^{t+\ell}$, i.e. the ℓ-block j at time $t+\ell$, is completely determined by the three ℓ-blocks, B_{j+1}^t, B_j^t, B_{j-1}^t, which are the ℓ-block itself and its left and right neighboring ℓ-blocks at time t. In other words, the ℓ-block $B_j^{t+\ell} \in Q^\ell$ is determined by the concatenation of ℓ-blocks $B_{j+1}^t B_j^t B_{j-1}^t \in Q^{3\ell}$. Such

a concatenation is called a "local configuration" and the transformation $B^t_{j+1}B^t_jB^t_{j-1} \mapsto B^{t+\ell}_j$ from $Q^{3\ell}$ to Q^ℓ is called a "local ℓ-transition". There are two crucial points:

What preprocessing in linear time? The number of possible "local configurations", therefore the number of possible "local ℓ-transitions" corresponding to the parameter N, called $\texttt{NbLocalConf}(N)$, is the cardinality of the set $Q^{3\ell}$, which is "small", precisely $s^{3\ell}$; this will make it possible to pre-compute in $O(N)$ time a table of all possible "local ℓ-transitions" for the parameter N;

How to compute op in constant time? Since the time (resp. length of the active interval) of the computation is cL, then the computation (resp. the active interval) is the concatenation of $cL/\ell = O(1)$ global ℓ-transitions (resp. ℓ-blocks), and all the computation on $\texttt{Input} \coloneqq \texttt{code}(N, X_1, \ldots, X_r)$ can be done by performing $(cL/\ell)^2 = O(1)$ local ℓ-transitions; it is a constant time since each local ℓ-transition is performed by consulting a single element in the table of local ℓ-transitions.

Remark 21 (processing the input and output in constant time)**.** *In this simplified view, we have not mentioned the reading of the operands $X_1, \ldots, X_r < N^d$, with the encoding process giving the initial configuration $\sharp\texttt{Input}\sharp$, for $\texttt{Input} \coloneqq \texttt{code}(N, X_1, \ldots, X_r)$, nor the final output process building $\texttt{op}(X_1, \ldots, X_r)$ by the projection of the final configuration. We will explain below how these input and output processes can also be performed in constant time.*

Remark 22. *Our simplified view also overlooked the fact that in general, the time cL of the computation is not a multiple of ℓ. In general, we have $cL = (c_0 - 1)\ell + \rho$, for the "constant" $c_0 \coloneqq \lceil cL/\ell \rceil$ and an integer $\rho \in [1, \ell]$. Therefore, the computation breaks down into $c_0 - 1$ global ℓ-transitions, each composed of c_0 local ℓ-transitions, and one global ρ-transition composed of c_0 local ρ-transitions of time ρ. As a local ℓ-transition, a local ρ-transition produces, from a local configuration $(B_2, B_1, B_0) \in (Q^\ell)^3$ of adjacent ℓ-blocks at some time t, the value $B'_1 \in Q^\ell$ of the middle ℓ-block at time $t + \rho$ (instead of time $t + \ell$ for a local ℓ-transition).*

The number of local configurations: Let us now determine the value of the "cut" constant D which determines the block length $\ell(N) \coloneqq \lceil L(N)/D \rceil$, for $L(N) \coloneqq \texttt{length}(N^d)$, so that the number $\texttt{NbLocalConf}(N)$ of the possible local configurations for an input $(N, X_1, \ldots, X_r)$ of the cellular automaton $\mathcal{A} = (Q, \delta)$ is $O(N^\sigma)$, for some $\sigma < 1$. By definition of local configurations, we have

$$\texttt{NbLocalConf}(N) = s^{3\ell} = s^{3\lceil L/D \rceil} < s^3 s^{3L/D}$$

because of $\lceil L/D \rceil < L/D + 1$. We also have $L = \texttt{length}(N^d) \leq 1 + \log_2 N^d = 1 + d \log_2 N$. It comes $s^{3L/D} \leq s^{3/D} s^{(3d/D)\log_2 N} = s^{3/D} N^{(3d/D)\log_2 s}$. from which we deduce

$$\texttt{NbLocalConf}(N) < s^{3+3/D} \times N^{(3d/D)\log_2 s}$$

This implies $\texttt{NbLocalConf}(N) < c_1 N^\sigma$, for the constant factor $c_1 \coloneqq s^{3+3/D}$ and the constant exponent $\sigma \coloneqq (3d/D)\log_2 s$, which is less than 1 if we fix the value of our constant integer D to more than $3d \log_2 s$. Therefore, setting $D \coloneqq 1 + \lfloor 3d \log_2 s \rfloor$ is suitable.

It is time to present in detail how a RAM can simulate in constant time the cellular automaton $\mathcal{A} = (Q, \delta)$ thanks to tables pre-computed in linear time. It is convenient to confuse the active part (resp. any ℓ-block) of a computation of $\mathcal{A}$, which is a nonempty word over the alphabet $Q = \{0, 1, \ldots, s-1\}$, with the integer represented by this word in the base s. As we have already seen, we can easily pass from a string of s-digits to the integer represented by this string in the base s, by Horner's method, and vice versa.

Computation of local transition tables. First, to represent the transition function δ of the CA, let us define the 3-dimensional array $\texttt{DELTA}[0..s-1][0..s-1][0..s-1]$ by

$$\texttt{DELTA}[x_2][x_1][x_0] \coloneqq \delta(x_2, x_1, x_0)$$

(Note that x_0, x_1, x_2 and $\delta(x_2, x_1, x_0)$ are explicit integers.) Of course, the following code computes the array DELTA in constant time:

Algorithm 21 Computation of the DELTA array

```
for x0 from 0 to s − 1 do
    for x1 from 0 to s − 1 do
        for x2 from 0 to s − 1 do
            DELTA[x2][x1][x0] ← δ(x2, x1, x0)
```

Let us now represent the table of local ρ-transitions $B^t_{j+1}B^t_jB^t_{j-1} \mapsto B^{t+\ell}_j$ from $Q^{3\ell}$ to Q^ℓ, for all the $\rho \in [1,\ell]$, by the 2-dimensional array $\texttt{LT}[1..\ell][0..s^{3\ell}-1]$ defined, for $1 \le \rho \le \ell$, $B_0, B_1, B_2 \in Q^\ell$ and $B = B_2s^{2\ell} + B_1s^\ell + B_0$, by $\texttt{LT}[\rho][B] := R$, which must be interpreted as follows: if, at any time t, three consecutive ℓ-blocks numbered $j+1$, j, $j-1$ are respectively B_2, B_1, B_0, then at time $t+\rho$, the middle ℓ-block (numbered j) is R.

The following code computes the LT array thanks to the DELTA array:

Algorithm 22 Computation of the LT array

```
1:  for B from 0 to s^(3*ℓ) − 1 do
2:      for ρ from 1 to ℓ do
3:          R ← B
4:          for x from 0 to 3 * ℓ − 1 do
5:              C[x][0] ← R mod s
6:              R ← R div s
7:          for t from 1 to ρ do
8:              for x from ℓ − ρ + t to 2 * ℓ − 1 + ρ − t do
9:                  C[x][t] ← DELTA[C[x + 1][t − 1]][C[x][t − 1]][C[x − 1][t − 1]]
10:         R ← 0
11:         for x from 2 * ℓ − 1 downto ℓ do
12:             R ← R * s + C[x][ℓ]
13:         LT[ρ][B] ← R
```

Justification: First, note that the main loop (1) goes through all the integers $B < s^{3\ell}$ which encode the local configurations (groups of three adjacent ℓ-blocks). Of course, $C[x,t]$ represents the state of the cell $x \in [0, 3\ell-1]$ of the local configuration at the instant numbered $t \in [0,\rho]$ of a time interval of duration ρ, and the main assignment (9) is a paraphrase of the transition function

$$C[x][t] = \delta(C[x+1][t-1], C[x][t-1], C[x-1][t-1]) \tag{19}$$

The loop condition $x \in [\ell-\rho+t, 2\ell-1+\rho-t]$ of line (8) for the application of (19) is justified inductively by the locality of the transition function: obviously, the states at time t of the cell interval $[\ell-\rho+t, 2\ell-1+\rho-t]$ (which is equal to the middle ℓ-block interval $[\ell, 2\ell-1]$ for $t=\rho$) are determined by the states at time $t-1$ of the cell interval $[\ell-\rho+(t-1), 2\ell-1+\rho-(t-1)]$, which has one more cell, numbered $\ell-\rho+t-1$ (resp. $2\ell-1+\rho-t+1$), at each end.

Besides, we use Horner's method for passing from an integer $B < s^{3\ell}$ (representing any local configuration) to the array of its 3ℓ s-digits (the initial string $C[x][0]$, for $0 \le x \le 3\ell-1$), and for passing from an array of ℓ s-digits (the final string $C[x][\rho]$, for $\ell \le x \le 2\ell-1$) to the integer $R < s^\ell$ represented by this array (the resulting middle ℓ-block of the local ρ-transition).

Representing the computation by a two-dimensional array of ℓ-blocks. We represent the computation of the CA by the sequence of its c_0+1 global configurations $\mathcal{C}_{t\ell} := \sharp B^{t\ell}_{c_0} \dots B^{t\ell}_1 \sharp$ at time $t\ell$, for $0 \le t \le c_0$, where $B^{t\ell}_i$ is the ith ℓ-block, $1 \le i \le c_0$, of $\mathcal{C}_{t\ell}$ from the right. Since we have $cL = (c_0-1)\ell + \rho$, for the constant $c_0 := \lceil cL/\ell \rceil$ and an integer ρ such that $1 \le \rho \le \ell$, note that, in case $\rho < \ell$, we will have to add $\ell - \rho$ symbols $\sharp$ at the left of the ρ-block numbered c_0 to make it an ℓ-block.

It will be convenient to add to each global configuration $\mathcal{C}_{t\ell}$ two ℓ-blocks, $B^{t\ell}_0 := \sharp^\ell$ at its right, and $B^{t\ell}_{c_0+1} := \sharp^\ell$ at its left. Thus, the global configuration $\mathcal{C}_{t\ell}$ is now written $B^{t\ell}_{c_0+1}B^{t\ell}_{c_0} \dots B^{t\ell}_1 B^{t\ell}_0$. Note that the string $\sharp^\ell$ is assimilated to the integer represented in base s by ℓ digits γ, $\overline{\gamma\cdots\gamma}$, which, by our convention $\sharp := \gamma := 2^{r+1}$, is equal to $2^{r+1}(s^{\ell-1}+\dots+s+1) = 2^{r+1}(s^\ell-1)/(s-1)$.

Computing the initial configuration. Recall $cL = (c_0-1)\ell + \rho$, for the "constant" $c_0 := \lceil cL/\ell \rceil$ and the integer ρ, which satisfies $1 \le \rho \le \ell$. Similarly, $L = (c_1-1)\ell + \lambda$, for the "constant" $c_1 := \lceil L/\ell \rceil$ and the integer λ, which satisfy $1 \le \lambda \le \ell$ and $c_1 \le c_0$. (We write that c_0 and c_1 are "constant" between quotes since, even if they depend on N, the main thing is that they are bounded by constants: precisely, we have $c_1 = \lceil L/\ell \rceil = \lceil L/\lceil L/D \rceil \rceil \le \lceil L/(L/D) \rceil = D$ and, likewise, $c_0 \le cD$.) Also, recall the definition $\texttt{code}(X_0, X_1, \dots, X_r) := u_{L-1}\dots u_1u_0 \in \Gamma^L$ where $u_j := \sum_{i=0}^r x_{i,j}2^i$, for $0 \le j < L$, and $x_{i,L-1}\dots x_{i,1}x_{i,0}$ denotes the binary representation of X_i on L bits. In other words, the jth γ-digit of $\texttt{code}(X_0, X_1, \dots, X_r)$, where $\gamma := 2^{r+1}$, is

$$\sum_{i=0}^{r} (j\texttt{th bit of } X_i) \times 2^i$$

We obtain the initial configuration $\mathcal{C}_0 = B^0_{c_0+1}B^0_{c_0}\dots B^0_1B^0_0$ by the four following steps.

1. Decompose each of the binary strings $X_i \in \{0,1\}^L$, $0 \le j \le r$, as a concatenation $X_i = X_{c_1,i}X_{c_1-1,i}\dots X_{1,i}$ of, from right to left, $c_1 - 1$ ℓ-blocks $X_{1,i},\dots,X_{c_1-1,i} \in \{0,1\}^\ell$ and a λ-block $X_{c_1,i} \in \{0,1\}^\lambda$;

2. Compute the decomposition of $\texttt{Input} \coloneqq \texttt{code}(X_0, X_1, \dots, X_r) \in \Gamma^L$ as a concatenation $U = U_{c_1}U_{c_1-1}\dots U_1$ of $c_1 - 1$ ℓ-blocks $U_1,\dots,U_{c_1-1} \in \Gamma^\ell$ and a λ-block $U_{c_1} \in \Gamma^\lambda$; here again, for each $h \in [1, c_1]$, we write $U_h = \texttt{code}(X_{h,0}, X_{h,1}, \dots, X_{h,r})$, to mean that for each rank i, the ith γ-digit of U_h is $\sum_{j=0}^{r}(i\texttt{th bit of } X_{h,j}) * 2^j$;

3. Compute the ℓ-blocks B^0_j, for $j \in [1, c_1-1]$, of the global configuration $\mathcal{C}_0 = B^0_{c_0+1}B^0_{c_0}\dots B^0_1B^0_0$ at time 0; note that, for each $j \in [1, c_1 - 1]$, the representation of U_j in base γ, denoted $\overline{a_{\ell-1}\dots a_1a_0}^\gamma$, "is" the representation of B^0_j in base s, i.e. the same string of digits: $B^0_j = \overline{a_{\ell-1}\dots a_1a_0}^s$; in terms of integers, this means that if we have $U_j = \sum_{i=0}^{\ell-1} a_i\gamma^i$ $(0 \le a_i \le \gamma - 1)$, then we obtain $B^0_j = \sum_{i=0}^{\ell-1} a_i s^i$, which we write $B^0_j = \texttt{convert}_{\gamma\to s}(U_j)$;

4. Compute the other ℓ-blocks of $\mathcal{C}_0$: as seen above, we have $B^0_j = \sharp^\ell = 2^{r+1}(s^\ell - 1)/(s-1)$, for $j \in \{0\} \cup [c_1 + 1, c_0 + 1]$ (recall $\sharp \coloneqq \gamma \coloneqq 2^{r+1}$); finally, the representation of the integer $B^0_{c_1}$ in the base s is the concatenation of strings $\gamma^{\ell-\lambda}U_{c_1}$, that means

$$B^0_{c_1} = 2^{r+1}(s^\ell - s^\lambda)/(s-1) + U'_{c_1}$$

with $U'_{c_1} \coloneqq \texttt{convert}_{\gamma\to s}(U_{c_1})$, which means that $U_{c_1} = \sum_{i=0}^{\lambda-1} a_i\gamma^i$ $(0 \le a_i \le \gamma - 1)$ implies $U'_{c_1} = \sum_{i=0}^{\lambda-1} a_i s^i$.

Computing the tables CODE and CONVERT. To compute the initial configuration in constant time by the previous algorithm (1-4), we need to pre-compute two arrays in linear time. Let $\texttt{CODE}[1..\ell][0..2^\ell - 1][0..2^\ell - 1]\dots[0..2^\ell - 1]$ be the array defined, for each $\lambda \in [1, \ell]$ and all $X_0, X_1, \dots, X_r \in [0, 2^\lambda - 1]$, by

$$\texttt{CODE}[\lambda][X_0][X_1]\dots[X_r] \coloneqq \texttt{code}(X_0, X_1, \dots, X_r)$$

where $\texttt{code}(X_0, X_1, \dots, X_r)$ is the integer in $[0, \gamma^\lambda - 1]$ $(\gamma = 2^{r+1})$ defined by item 2 above. The CODE array is computed by the following algorithm:

Algorithm 23 Computation of the CODE array

```
1:  for λ from 1 to ℓ do
2:      for (X_0, X_1, ..., X_r) ∈ [0, 2^λ − 1]^{r+1} do
3:          for j from 0 to r do
4:              for i from 0 to λ − 1 do
5:                  BIT_j[i] ← X_j mod 2
6:                  X_j ← X_j div 2
7:          for i from 0 to λ − 1 do
8:              DIGIT[i] ← 0
9:              for j from r downto 0 do
10:                 DIGIT[i] ← 2 * DIGIT[i] + BIT_j[i]
11:         V ← 0
12:         for i from λ − 1 downto 0 do
13:             V ← γ * V + DIGIT[i]
14:         CODE[λ][X_0][X_1]...[X_r] ← V
```

Similarly, let $\texttt{CONVERT}[1..\ell][0..\gamma^\ell - 1]$ be the array defined, for all lengths $\lambda \in [1, \ell]$ and all integers $B \in [0, \gamma^\lambda - 1]$, by

$$\texttt{CONVERT}[\lambda][B] \coloneqq \texttt{convert}_{\gamma\to s}(B)$$

where $\texttt{convert}_{\gamma\to s}(B)$ is the integer in $[0, \gamma^\lambda - 1]$ defined by items 3 and 4 above. Clearly, the CONVERT array is computed by the following algorithm:

Algorithm 24 Computation of the `CONVERT` array

```
1: for λ from 1 to ℓ do
2:     for B from 0 to γ^λ − 1 do
3:         U ← B
4:         for i from 0 to λ − 1 do
5:             DIGIT[i] ← U mod γ
6:             U ← U div γ
7:         U ← 0
8:         for i from λ − 1 downto 0 do
9:             U ← s ∗ U + DIGIT[i]
10:        CONVERT[λ][B] ← U
```

Computing the output by projecting the final configuration. This step is similar but much simpler than the computation of the initial configuration from the input. Here again, by its locality, the projection is done for each symbol $x \in Q$ of each ℓ-block of the final configuration. It uses the array $\texttt{PROJECT}[0..s^\ell - 1]$ defined by

$$\texttt{PROJECT}[B] \coloneqq \sum_{i=0}^{\ell-1} \pi(x_i) \times 2^i$$

where B is any ℓ-block $x_{\ell-1} \dots x_1 x_0 \in Q^\ell$, or equivalently, $B = \sum_{i=0}^{\ell-1} x_i \times s^i$, for $x_i \in [0, s-1]$. Clearly, this array is defined by the following algorithm:

Algorithm 25 Computation of the `PROJECT` array

```
1: for B from 0 to s^ℓ − 1 do
2:     R ← B
3:     for i from 0 to ℓ − 1 do
4:         DIGIT[i] ← PI[R mod s]
5:         R ← R div s
6:     R ← 0
7:     for i from ℓ − 1 downto 0 do
8:         R ← 2 ∗ R + DIGIT[i]
9:     PROJECT[B] ← R
```

The code above uses the array $\texttt{PI}[0..s-1]$ defined by $\texttt{PI}[x] \coloneqq \pi(x)$ and computed by the following code, where $\pi(0), \pi(1), \dots, \pi(s-1)$ are explicit elements of Q:

```
PI[0] = pi(0) ; PI[1] = pi(1) ; ... ; PI[s-1] = pi(s-1)
```

Simulation of the complete computation of the cellular automaton. Recall that, for the "constants" $c_0 \coloneqq \lceil cL/\ell \rceil$ and $\rho \coloneqq cL - (c_0 - 1)\ell$ (so that $1 \le \rho \le \ell$), the computation decomposes into $c_0 - 1$ global ℓ-transitions, each composed of c_0 local ℓ-transitions, and one global ρ-transition composed of c_0 local ρ-transitions of time ρ. (As a local ℓ-transition, a local ρ-transition produces, from a local configuration $(B_2, B_1, B_0) \in (Q^\ell)^3$ of adjacent ℓ-blocks at some time t, the value $B_1' \in Q^\ell$ of the middle ℓ-block at time $t + \rho$.)

Finally, the operation `op` is computed by the following procedure, also called OP, which simulates the cellular automaton $\mathcal{A} = (Q, \delta)$ with its projection $\pi : Q \to \{0, 1\}$. The main data structure of the OP procedure is a two-dimensional array, called $\texttt{BK}[0..c_0+1][0..c_0]$ and defined by $\texttt{BK}[x][t] \coloneqq B_x^{t\ell}$, where $B_x^{t\ell}$ is the ℓ-block numbered x, $0 \le x \le c_0 + 1$, of the global configuration $\mathcal{C}_{t\ell}$ at time $t\ell$.

Algorithm 26 Constant-time computation of the op operation computed by the cellular automaton $\mathcal{A}$ in linear time

```
1:  procedure OP(X_1, ..., X_r)
2:      X_0 ← N
3:      for j from 0 to r do
4:          for x from 1 to c_1 do
5:              X[j][x] ← X_j mod 2^ℓ
6:              X_j ← X_j div 2^ℓ
7:      for x from 1 to c_1 − 1 do ▷ 2: get the c_1 − 1 ℓ-blocks (and the λ-block) of
        code(X_0, X_1, ..., X_r)
8:          U[x] ← CODE[ℓ][X[0][x]]...[X[r][x]]
9:      U[c_1] ← CODE[λ][X[0][c_1]]...[X[r][c_1]]
10:     for x from 1 to c_1 − 1 do
11:         BK[x][0] ← CONVERT[ℓ][U[x]]
12:     BK[0][0] ← 2^{r+1}(s^ℓ − 1)/(s − 1)
13:     BK[c_1][0] ← CONVERT[λ][U[c_1]] + 2^{r+1}(s^ℓ − s^λ)/(s − 1)
14:     for x from c_1 + 1 to c_0 + 1 do
15:         BK[x][0] ← 2^{r+1}(s^ℓ − 1)/(s − 1)
16:     for t from 1 to c_0 − 1 do
17:         for x from 1 to c_0 do
18:             BK[x][t] ← LT[ℓ][CONC(BK[x + 1][t − 1], BK[x][t − 1], BK[x − 1][t − 1])]
19:     for x from 1 to c_0 do
20:         BK[x][c_0] ← LT[ρ][CONC(BK[x + 1][c_0 − 1], BK[x][c_0 − 1], BK[x − 1][c_0 − 1])]
21:     R ← 0
22:     for x from c_0 downto 1 do
23:         R ← s^ℓ ∗ R + PI[BK[x][c_0]]
24:     return R
```

It is immediate to check that the body of each "for" loop of the OP procedure is repeated a number of times bounded by r, c_0 or c_1, therefore by a constant, and therefore the OP procedure works in constant time as claimed.

Why is the preprocessing time linear? Our constant-time procedure OP uses five precomputed tables: PI, computed in constant time, and LT, CODE, CONVERT and PROJECT. By examining the structure of the loops of the above algorithms, it is easy to verify that

- LT is computed in time $O(\texttt{NbLocalConf}(N) \times \ell^3) = O(N^\sigma \times (\log N)^3) = O(N)$ (recall $\texttt{NbLocalConf}(N) = s^{3\ell} = O(N^\sigma)$, for a fixed $\sigma < 1$),
- CODE and CONVERT are computed in time $O((2^\ell)^{r+1} \times \ell^2) = O(\gamma^\ell \times \ell^2)$, and $O(\gamma^\ell \times \ell^2)$, respectively, and therefore in time $O(s^\ell \times \ell^2)$, and PROJECT is computed in time $O(s^\ell \times \ell)$, and therefore, a fortiori, all these tables are computed in time $O(N)$.

This completes the proof of Theorem 3.

□

12 Appendix: Another constant-time algorithm for division

While multiplication can be computed in linear time on a cellular automaton, so Theorem 3 applies to it, it is unknown if division is. However, the intuitive principle of the usual division algorithm is also "local": to compute the most significant (leftmost) digit of the quotient of integers $\lfloor A/B \rfloor$, the first step is to "try" to divide the integer a consisting of the 1, 2, 3, or 4 leftmost digits of A by the integer b consisting of the 1, 2, or 3 leftmost digits of B.

We are going to design a new constant-time algorithm for division by a careful adaptation of the usual division principle combined with a judicious choice of the base K in which the integers are represented. To avoid any vicious circle here, we will only use the addition, subtraction, multiplication and base change of "polynomial" integers, and the procedure DIVBYSMALL(A, B) of Subsection 5.1, which returns $\lfloor A/B \rfloor$ for $A < \beta^d$, $0 < B < \beta$, for a fixed integer $d \geq 1$ and $\beta \coloneqq \lceil N^{1/6} \rceil^3$.

Analyzing and optimizing the usual division algorithm. Assume the dividend A and the divisor B are represented in any fixed base $K \geq 2$ and let a (resp. b) be the integer represented by the 1, 2, 3, or 4 (resp. 1, 2, or 3) leftmost digits of A (resp. B). Each positive integer is assimilated to the string $\overline{x_{p-1} \dots x_1 x_0}$ of digits x_i, $0 \leq x_i < K$, which represents it in base K, with leading digit $x_{p-1} > 0$. Clearly, we get the inequalities $aK^m \leq A < (a+1)K^m$ and $bK^n \leq B < (b+1)K^n$, for some integers $a, b \geq 1$ and m, n. Because of $A \geq B$, we can assume $m \geq n$. It implies $\frac{a}{b+1}K^{m-n} < A/B < \frac{a+1}{b}K^{m-n}$. To do *one* test instead of several tests to get the *most significant* digit of the quotient A/B in base K, we want the difference between the bounds to be less than K^{m-n}, it means $\frac{a+1}{b} - \frac{a}{b+1} \leq 1$. Noticing the identity $\frac{a+1}{b} - \frac{a}{b+1} = \frac{a}{b(b+1)} + \frac{1}{b}$, this can be rephrased as

$$\frac{a}{b(b+1)} + \frac{1}{b} \leq 1. \tag{20}$$

Lemma 15. *Assume $aK^m \leq A < (a+1)K^m$ and $bK^n \leq B < (b+1)K^n$, for some integers $K \geq 2$, and A, B such that $A > B \geq 1$, and m, n, a, b such that $m \geq n \geq 0$, and $a > b \geq 1$, and $\frac{a}{b(b+1)} + \frac{1}{b} \leq 1$. We then get*

$$qK^{m-n} \leq A/B < (q+1)K^{m-n} \text{ for } q = \left\lfloor \tfrac{a}{b+1} \right\rfloor \text{ or } q = \left\lfloor \tfrac{a}{b+1} \right\rfloor + 1,$$

which implies $A = BqK^{m-n} + R$ for some integer $R < BK^{m-n}$, and therefore

$$\lfloor A/B \rfloor = qK^{m-n} + \lfloor R/B \rfloor \tag{21}$$

with $q \geq 1$ and $\lfloor R/B \rfloor < K^{m-n}$.

Proof. From the inequalities $\frac{a}{b+1}K^{m-n} < A/B < \frac{a+1}{b}K^{m-n}$ we deduce $\left\lfloor \frac{a}{b+1} \right\rfloor K^{m-n} < A/B < \left\lceil \frac{a+1}{b} \right\rceil K^{m-n}$. Besides, the inequality $\frac{a+1}{b} - \frac{a}{b+1} \leq 1$ implies $\left\lceil \frac{a+1}{b} \right\rceil \leq \left\lfloor \frac{a}{b+1} \right\rfloor + 2$. This yields $qK^{m-n} < A/B < (q+2)K^{m-n}$ for $q = \left\lfloor \frac{a}{b+1} \right\rfloor \geq 1$, which proves the lemma. □

Remark 23. *Lemma 15 gives and justifies the principle of a recursive division algorithm in any fixed base $K \geq 2$: to compute $\lfloor A/B \rfloor$, compute the integers q and $\lfloor R/B \rfloor$ in base K; the representation of the integer $\lfloor A/B \rfloor$ in base K is the concatenation of the string representing the integer $q \geq 1$ and the string of $m - n$ digits representing the integer $\lfloor R/B \rfloor < K^{m-n}$ (possibly with leading zeros).*

How to use Lemma 15? How many leftmost (most significant) digits of A and B, written in base K, should we take to obtain integers a and b, with $a \geq b \geq 1$, that satisfy the condition $\frac{a}{b(b+1)} + \frac{1}{b} \leq 1$ of Lemma 15? This is expressed precisely by the following question: for which integers i, j is the implication $K^i \leq a < K^{i+1} \wedge K^j \leq b < K^{j+1} \Rightarrow \frac{a}{b(b+1)} + \frac{1}{b} \leq 1$ true for all a, b?

Lemma 16. *Let i, j be two positive integers. Then the condition $i < 2j$ is equivalent to the condition*

$$\forall (a,b) \in [K^i, K^{i+1}[\times [K^j, K^{j+1}[\ \left(\frac{a}{b(b+1)} + \frac{1}{b} \leq 1 \right)$$

Proof. The maximum of the numbers $\frac{a}{b(b+1)} + \frac{1}{b}$ for all pairs of integers $(a, b) \in [K^i, K^{i+1}[\times [K^j, K^{j+1}[$ is $\frac{K^{i+1}-1}{K^j(K^j+1)} + \frac{1}{K^j} = \frac{K^{i+1}+K^j}{K^{2j}+K^j}$, which is at most 1 if and only if $i + 1 \leq 2j$. This proves the lemma. □

We are now ready to give our new division algorithm using the principle given by Lemma 15. Here again, we must choose carefully the base K.

The choice of the base. We take $K \coloneqq \lceil N^{1/7} \rceil$ and we assume that each "polynomial" operand $X < N^d$, for a fixed integer d, is written $\overline{x_{m-1} \dots x_1 x_0}$, $0 \leq x_i \leq K - 1$, in base K, with one digit x_i per register, and is therefore stored in $m \leq 7d$ registers, since $X < N^d \leq K^{7d}$. In particular, the following functions can be computed in constant time for their argument X so represented: $\texttt{length}_K(X) \coloneqq m$; $\texttt{Substring}_K(X, i, j) \coloneqq \overline{x_{i-1} \dots x_j}$, for $m \geq i > j \geq 0$.

Notation. *For an integer X written $\overline{x_{m-1} \dots x_1 x_0}$ in base K, $0 \leq x_i \leq K - 1$, and a positive integer $p \leq m$, denote by $X_{(p)}$ the prefix of the p most significant digits of X: $X_{(p)} \coloneqq \overline{x_{m-1} \dots x_{m-p}} = X \texttt{ div } K^{m-p}$.*

The division algorithm. Let A, B be two "polynomial" integers such that $0 < B \leq A < K^d$, for a fixed integer d. To divide A by B, we consider four cases:

1. $B < K^2$;
2. $K^p \leq B \leq A < K^{p+1}$, for an integer $p \geq 2$, and $A_{(3)} = B_{(3)}$;
3. $K^2 \leq B \leq A$ and $A_{(3)} > B_{(3)}$;
4. $K^p \leq B < K^{p+1} \leq A$, for an integer $p \geq 2$, and $A_{(3)} \leq B_{(3)}$.

Obviously, these four cases are mutually exclusive. To ensure that there is no other case than these four cases, just note that if condition 2 is false when $K^2 \leq B \leq A < K^d$, then either we have $A_{(3)} > B_{(3)}$ (case 3), or we have $A_{(3)} \leq B_{(3)}$ *and* A has *more* digits than B in base K (case 4).

- Case 1 is handled by the procedure DivBySmall(A, B) of Subsection 5.1 (note that $K^2 = \lceil N^{1/7} \rceil^2 \leq \lceil N^{1/6} \rceil^3 = \beta$).
- In case 2, we clearly get $A - B < K^{p-2} < B$, which implies $A < 2B$ and therefore $\lfloor A/B \rfloor = 1$.
- Cases 3 and 4 are handled by Lemma 15. In case 3 (resp. case 4), take $a = A_{(3)}$ and $b = B_{(3)}$ (resp. $a = A_{(4)}$ and $b = B_{(3)}$), which implies $K^2 \leq b < a < K^3$ (resp. $K^2 \leq b < K^3 \leq a < K^4$).

The integers $i = 2$ and $j = 2$ (resp. $i = 3$ and $j = 2$) satisfy the condition $i < 2j$ of Lemma 16. Thus, in cases 3 and 4, the condition $\frac{a}{b(b+1)} + \frac{1}{b} \leq 1$ of Lemma 15 is satisfied. Therefore, Lemma 15 proves the correctness of the following Division procedure which implements cases 1-4 for $A \geq B$. More precisely, the lines 17-21 of the algorithm compute $\lfloor A/B \rfloor$ in cases 3 and 4 by using the pre-computed array $\texttt{DIV}[0..K^4-1][2..K^3]$ defined by $\texttt{DIV}[x][y] := \lfloor x/y \rfloor$, for $0 \leq x < K^4$ and $2 \leq y \leq K^3$ (see the pre-computation code of the DIV array presented after the Division procedure).

Algorithm 27 Constant-time computation of the `division` operation

```
1:  procedure Division(A, B)
2:      if A < B then
3:          return 0
4:      if B < K^2 then                                          ▷ case 1
5:          return DivBySmall(A, B)
6:      ℓ_A ← length_K(A)
7:      a ← Substring_K(A, ℓ_A, ℓ_A − 3)                         ▷ a ← A_(3)
8:      m ← ℓ_A − 3
9:      ℓ_B ← length_K(B)
10:     b ← Substring_K(B, ℓ_B, ℓ_B − 3)                         ▷ b ← B_(3)
11:     n ← ℓ_B − 3
12:     if ℓ_A = ℓ_B and a = b then
13:         return 1          ▷ length_K(A) = length_K(B) and A_(3) = B_(3): case 2
14:     if a ≤ b then                                            ▷ A_(3) ≤ B_(3): case 4
15:         a ← Substring_K(A, ℓ_A, ℓ_A − 4) ▷ a ← A_(4), so b = B_(3) < a < (b + 1)K
16:         m ← ℓ_A − 4
17:     q ← DIV[a][b + 1]          ▷ b + 1 ≤ a < (b + 1)K and therefore 1 ≤ q < K
                ▷ common treatment of cases 3 and 4 justified by Lemma 15
18:     if A ≥ B ∗ (q + 1) ∗ K^(m−n) then
19:         q ← q + 1
20:     R ← A − B ∗ q ∗ K^(m−n)
21:     return q ∗ K^(m−n) + Division(R, B)                      ▷ justified by equation 21
```

Comments: In case 4 ($a = A_{(3)} \leq B_{(3)} = b$), after the assignment $a \leftarrow A_{(4)}$ of line 15, we have $a = A_{(4)} < (A_{(3)} + 1)K$. Now, by the hypothesis $A_{(3)} \leq B_{(3)} = b$ (line 14), we get $b = B_{(3)} < a = A_{(4)} < (A_{(3)} + 1)K \leq (b + 1)K$, and thus $b < a < (b + 1)K$, as the comment of line 15 claims.

In case 3, i.e. for $a = A_{(3)} > B_{(3)} = b$, we also have $a = A_{(3)} < B_{(4)} < (B_{(3)}+1)K = (b+1)K$. This justifies, for cases 3 and 4, the assertion given as a comment of line 17: $b+1 \leq a < (b+1)K$.

It remains to describe the preprocessing and to prove that the claimed complexity is achieved.

Pre-computation of the `DIV` array. Note that in the DIVISION procedure, we compute $q = \lfloor a/(b+1) \rfloor$ when $2 \leq b+1 \leq K^3$ and $1 \leq q \leq K-1$. Therefore, the following algorithm constructs the `DIV` array.

Algorithm 28 Pre-computation of the `DIV` array

```
1: for y from 2 to K^3 do
2:     for q from 1 to K − 1 do
3:         for r from 0 to y − 1 do
4:             x ← q ∗ y + r
5:             DIV[x][y] ← q
```

Complexity of the preprocessing and division algorithm. We immediately see that the time of the above algorithm which constructs the array `DIV`$[0..K^4-1][2..K^3]$ is $O(K^3 \times K \times K^3) = O(K^7) = O(N)$.

The complexity of the DIVISION procedure is determined by its recursive depth (number of recursive calls). By equation 21 (implemented by line 21), which is $\lfloor A/B \rfloor = qK^{m-n} + \lfloor R/B \rfloor$ with $q \geq 1$ and $\lfloor R/B \rfloor < K^{m-n}$, the quotient $\lfloor R/B \rfloor$ has at least one digit less than $\lfloor A/B \rfloor$ when represented in base K (the digit q, multiplied by K^{m-n}). Therefore, the recursive depth is bounded by the number of digits of the representation of $\lfloor A/B \rfloor$ in base K, which is less than $7d$ because of $\lfloor A/B \rfloor < K^{7d}$. Since the most time consuming instruction is the multiplication, executed $O(d)$ times (lines 18, 20, 21) and whose time cost is $O(d^2)$, the time of the DIVISION procedure is $O(d^3)$, at compare to the time $O(d^4)$ of the DIVISION procedure of Section 5, see Theorem 2.