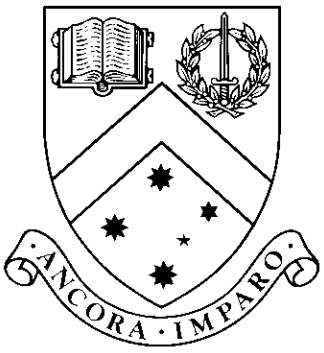
Computing with Arbitrary and Random Numbers

by

Michael Brand, M.Sc.



Thesis

Submitted by Michael Brand

for fulfilment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Prof. Graham Farr

Associate Supervisor: A/Prof. Ian M. Wanless

Clayton School of Information Technology
Monash University

October, 2013

To Orit, Maayan and Zoharel

The fairest order in the world is a heap of random sweepings.

Heraclitus[†]

[†]Collected in Charles H. Kahn, *The Art and Thought of Heraclitus*, 1979.

Contents

List of Tables

List of Figures

List of Algorithms

List of Notations

Operations

Computation models

ARAM	RAM with access to an arbitrary large number 41
ASRAM	arbitrary sequence RAM 88
stochastic RAM	A randomized RAM introduced in Simon (1981) 98
R*RAM	A RAM with access to RAND() 118
RRAM	same as R*RAM but with restricted RAND() 101
TM	Turing Machine 21
SLP	straight line program 42
CT	computation tree 43
Oblivious TM	Turing machine with predetermined head movements 62

Complexity classes

P-RAM	polynomial-time RAM 18
NP-RAM	RAM working in nondeterministic polynomial time 39
ER	a complexity class defined by Simon (1981) 57
r.e.	the recursively enumerable sets 40
PEL	polynomial time expansion limit 61
$\Sigma_i^0, \Pi_i^0, \Delta_i^0$	The i'th level of the arithmetical hierarchy 83
AH	the arithmetical hierarchy 91
BPP, RP, PP, ZPP, RL, ZPL	randomized complexity classes 96
PP-RAM	polynomial time stochastic RAM 98
RP-RAM	polynomial time RRAM 101

Data structures

(m, V, n)	encoded vector 31
(L, I, w)	encoded map 106
po-bit	potentially interesting bit 50
instantaneous description	instantaneous description of a TM 70

tableau	tableau of a TM's execution	70

Functions and pseudofunctions

$O^n_{a,m}$	$O^n_{a,m} \stackrel{\text{def}}{=} ((a \leftarrow nm) \dot{-} a)/((1 \leftarrow m) \dot{-} 1)$	34
U^T	Universal string, $U^T \stackrel{\text{def}}{=} \sum_{i=1}^{2^T-1} O^i_{1,T}$	81
EL	expansion limit	58
$stexp(n)$	defined in Simon (1981) as $stexp(n) \stackrel{\text{def}}{=} {}^n2$	59
ALN()	a pseudofunction returning an ALN	88
RAND(x)	a pseudofunction returning a u.r.v., y, in $0 \leq y < x$	98
TRUTH	a function returning whether a formula is true	92
ψ, ϕ, χ	logical formulae	92

Misc

$\Leftarrow$	assignment	32
RAM-constructable	an operation set over which EL can be calculated uniformly	58
ALN	arbitrary large number	41

Computing with Arbitrary and Random Numbers

Michael Brand, M.Sc.
[redacted]
Monash University, 2013

Supervisor: Prof. Graham Farr
[redacted]
Associate Supervisor: A/Prof. Ian M. Wanless
[redacted]

Abstract

We consider integer random access machines (RAMs) that receive one of two types of special integers as extra inputs: arbitrary numbers and random numbers.

Arbitrary numbers are numbers that have no special property, other than being large-valued. Informally, one can consider them as adversarially-chosen numbers.

Intuitively, it would not seem that arbitrary numbers offer any value. However, previous studies have shown for specific problems that such numbers contribute to the computational power of a RAM. The present work gives, for the first time, a broad characterisation of the scenarios in which arbitrary numbers do and those in which they do not increase computational power, rather than considering specific problems.

The second type of extra inputs, random numbers, is conceptually an intermediate ground between Oracle-given certificates and adversarially-chosen arbitrary numbers. The contribution of random inputs to Turing machines is the famous P = RP problem. In the context of RAMs, it was posed as an open question by Simon (1981). The present study closes this problem, by showing that for certain RAMs randomness does provide an advantage, whereas for others it does not.

In order to achieve the results presented, this work also reviews classical results regarding certificates and the use of indirect addressing (which is a pseudo-operation available to RAMs, but which can be conceptually disabled so as to have its contribution modelled). In both

cases, our results sharpen the state-of-the-art. In the case of certificates, we show how results by Simon (1981) on RAMs with certificates continue to hold also with a slightly reduced operation set (as well as with new operation sets). In the case of indirect addressing, we show how, for specific RAMs, this pseudo-operation can be simulated with no loss of complexity, whereas previously the state of the art was for a loss of complexity by a factor of the inverse Ackermann function.

The following are some of our main results. We define a new complexity class, PEL, which formalises the idea of describing the power of a RAM in terms of its capacity to generate large numbers, and show that for a wide class of RAMs (specifically those that support addition, Boolean operations, left-shift by a variable amount and division) computational power is, indeed, directly related to this capacity. In particular, the addition of an arbitrary large number to any RAM belonging to this class gives it the ability to recognise any recursively enumerable set in O(1) time. Indeed, functions computable in this model are on the second level of the arithmetical hierarchy. When adding to this the ability to generate additional arbitrary large numbers at will, we show that an O(1)-time computation has the same power as the entire arithmetical hierarchy, and an $\omega(1)$-time computation extends even beyond that.

For certain other RAMs, we are able to show that the addition of arbitrary large numbers does not increase their computational power at all.

For RAMs that have access to random numbers, we close an open problem raised by Simon (1981), who asked for a characterisation of the power of certain RAMs working in polynomial time on random numbers. Intriguingly, the power of these RAMs is PEL, too. However, it remains an open question whether they, too, enjoy the power-boost afforded by arbitrary large numbers.

Computing with Arbitrary and Random Numbers

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Michael Brand
October 14, 2013

Acknowledgments

I would like to begin by thanking Prof. Graham Farr, my supervisor, for his willingness to supervise a doctorate in this highly unusual topic. I will not repeat here all superlatives that I used when I nominated him for the Vice Chancellor's Award for Excellence in Postgraduate Supervision award, which he was ultimately awarded, but will merely reiterate that I feel privileged for having worked with him.

Both he and my associate supervisor, A/Prof. Ian Wanless, deserve the greatest of praise for their enormous support throughout the process, and in particular for allowing me my meandering and often roundabout ways of progress.

Among their many credits, Graham and Ian are the founders and managers of the cross-faculty Discrete Math Research Group. I would like to thank all members of this group for their encouragement, and particularly to Kerri Morgan, Daniel Horsley and Douglas Stones for their advice, insights and constant empathy.

Chronologically earlier, I would like to thank the many people who convinced me to begin a doctorate at this time in my life, most especially Dr. Orna Berry and Prof. Edwina Cornish. Just as importantly, from the moment I began seriously considering this path many good people steered me towards Graham and Ian. I will mention two in particular, whose help pushing me along was invaluable: Prof. Ingrid Zukerman and Prof. Kim Marriott.

Just as importantly, I would like to mention my boss when I began this endeavour, Aviad Maizels, for trusting that I am able to accomplish this feat while continuing to work for him, as well as my current boss, Annika Jimenez, for having taken up from him this torch of trust.

Moving further back in time, a special thank-you goes to my good friend Shachar Shemesh, who planted in me the initial seed of doubt regarding the validity of complexity calculations as they are usually presented.

Last but not least, I would like to thank my parents for teaching me to value curiosity and the independent search for knowledge, my children, Maayan and Zoharel, for instilling in me a sense of wonder and amazement, and my long-suffering wife, Orit, for imbuing in me a sense of urgency. I am told that less than one in four people who begin the road to a Ph.D. actually complete it. My personal recipe for success, unduplicable though it is, is in this combination: a love of independent learning, a sense of wonder and a dash of urgency.

Combine that with the proper, light-but-significant touch of guidance and let it simmer for a few years until ready.

Michael Brand

Monash University

October 2013

Preface, on a personal note

Some time ago, as I was contemplating whether or not to register for the "Three Minute Thesis" competition (which, unfortunately, I was ultimately unable to attend), I raised with the Discrete Math Research Group the question of how they think a topic as complex and background-requiring as that of the present work can be presented in only three minutes. Among the pieces of advice that I received was the following (by a person who shall remain anonymous): "Sometimes a lie is the quickest way to the truth".

This advice is applicable not only to three-minute thesis presentations. It is also commonly used in general pedagogical situations. My wife, who teaches grade school, once asked me what I learned in the first semester of my B.Sc., to which I replied that the first semester was devoted almost entirely to addition and multiplication. She was completely flabbergasted by this prospect, as — she told me — her grade school students were already quite adept at these tasks. Indeed, when she asked me if during my M.Sc. studies I also had to learn addition and multiplication, I was forced to reply that in my M.Sc. the material was not that advanced, and that, at the time (having taken a course in elementary set theory), I was still struggling with the concept of what a number is.

And yet, this description of my studies is accurate: the understanding of addition and multiplication gained in post-high-school studies is the basis for ring, group and field theory, as well as linear algebra, which, in turn, form the basic structures and concepts upon which many other mathematical disciplines are built. In grade school, though we learn the mechanical proficiency of addition and multiplication, we are "deceived" into believing that this mechanical proficiency is the same as a deep understanding.

This "lie to get to the truth" methodology is in common use, but for some reason where I find it most unacceptable is in how complexity is taught. My experience (and in my belief, this is the common experience) is that complexity is typically taught after an introduction to Turing machines, it is explained in the context of Turing machines, it is shown to hold beautiful properties for Turing machines, and then, from that point and throughout the rest of a person's professional training, complexity is used often and as a central tool, but *always* as it pertains to high-level code or pseudocode, which acts like a Random Access Machine. In all such cases, the computational model is never exactly specified, nor is there ever a discussion of how RAM complexities differ from Turing machine complexities, nor how they relate to each other.

Personally, I find the pedagogically-beneficial simplification of addition for grade-school students to be acceptable, because they are novices, not professionals seeking specialisation in the field. For university-level mathematics students, who do specialise in the field, a much deeper presentation of the topic is expected — and, indeed, given. On the other hand, for exactly the same reason, the simplification given in the treatment of complexity I find questionable. In this case, the simplified version is the version presented to students of IT and Computer Science, specialising in the development and analysis of algorithms. The very people who should gain an in-depth understanding of this topic are denied it.

This work grew from my personal frustration over this convenient simplification, the truth behind which I found to be virtually unknown and exceptionally fascinating. It is my hope that the present text will lead more people to take the high road of complexity theory, and to use it more judiciously.

Of course, now, if my wife asks me the follow-up question of what I learned during my Ph.D. studies, I will have to answer that this time I did not learn to add or multiply nor what a number is, but rather the essential meaning of what it is to compute.

Chapter 1

Introduction

1.1 Complexity and its underlying model

There is arguably no other theoretical tool in the arsenal of the algorithm developer that is as powerful as complexity, in its ability to compare the theoretical performance of algorithms. Complexity computation is a ubiquitous part of the development and presentation of any new algorithm, and much of the theory of algorithms deals with their classification into complexity classes. In fact, many basic texts (e.g. Brassard and Bratley, 1995) devote entire chapters specifically to it.

The idea of complexity is of some seniority, with many precursors (e.g. Rabin, 1960) that laid out the foundations to the notion that not all problems are equally complex, and that the complexity relations between problems define a hierarchy. This idea ultimately led to Hartmanis and Stearns (1964) where complexity in its modern form, relating the length of a problem's description to the time and resources necessary to solve it, was established.

Complexity is not, however, an intrinsic property of algorithms. Algorithms can only be attributed a complexity with respect to a particular computational model.[1] The computational model widely regarded as canonical is the Turing machine (Turing, 1936), and most

[1] Blum (1967) describes general axioms for deciding whether a function can reasonably be called a complexity measure, via a construction that does not require any underlying computational model to be explicitly defined. However, this is merely a general framework. Practical specific complexities are almost invariably tied to a specific model. On the contrary, some natural measures, such as the average run-time of a probabilistic Turing machine, have been shown (Gill, 1977) to not satisfy Blum's axioms. Complexity measures discussed in this thesis are all Blum complexity measures.

of the theoretical study of complexity has been devoted to Turing machine complexity. (A representative treatment of Turing machine complexity is Fortnow and Homer (2003), a 39-page review of the history of complexity that does not mention RAM complexity, the type of complexity that is the object of study in this work, even once.)

One of the appealing properties of the Turing machine computational model, with respect to complexity, is that many of the possibilities to vary over it (e.g. by the addition of extra tapes, by a change in alphabet size, etc.) only affect the complexity of algorithms by a polynomial function. As such, it is convenient to discuss algorithms and problems in terms of complexity classes (such as P, the class of problems that can be solved by polynomial-time algorithms) rather than specific complexities (such as $O(n)$ or $O(n^2)$). This is an extension of the abstraction level already present in basic complexity notation, which ignores, e.g., multiplication by a constant. A discussion of this equivalence up to a polynomial appears, for example, in Arora and Barak (2009), where an entire chapter is devoted to "why the computational model doesn't matter".

However, contrary to this (unfortunately common) presentation, the computational model does matter. General computational models do not share this good property of equivalence up to a polynomial exhibited by Turing machines. The complexities they attribute to algorithms are not polynomially-related to Turing-machine complexity, to each other, or even to variations over the same computational model.

As an example, consider sorting by comparisons[2]. Although algorithms for sorting by comparisons are often described as being, e.g., $O(n^2)$-time or $O(n \log n)$-time (Wikipedia, 2012b,a), in fact, they require $O(n^2)$ (or $O(n \log n)$) *comparison operations*. Nothing guarantees *a priori* that a single comparison can be performed in $O(1)$ time, or even time that is polynomial in the length of the input. Though this is true for RAMs, where the comparison of two integers, a and b is often assumed to be a basic, $O(1)$, operation, and is also true for Turing machines (which require $O(\log \min(a, b))$-time for the case of multi-tape Turing machines and $O(\log^2 \min(a, b))$-time for the case of single tape Turing machines), the same is not true for Minsky machines (Minsky, 1961), which require $O(\min(a, b))$-time to accomplish the

[2]Otherwise known as "comparative sorting" or "comparison sort".

same task. Comparison between two numbers is, for Minsky machines, an exponential time complexity problem. (Yet other computational models, such as Church's λ-calculus (Church, 1936), do not describe a hypothetical computing machine at all, and have no explicit concept comparable with time- or space-complexity.)

1.2 Random Access Machines

Among the wealth of possible computational models to choose from, the study of Random Access Machines (RAMs) is of particular importance. RAMs are a family of models of computation that allow the definition of basic operations that can be performed over an infinite set of registers, each of which can take values from an infinite set (typically, the non-negative integers). Basic operations can each be assigned any complexity, but typically they are regarded as operating in constant time. This is known as the "unit cost model". The reason for the importance of the RAM model is its ability to encapsulate high-level computer language.

In the three-quarters of a century that have passed since Turing introduced his seminal model, software-programmable computers have become a household product and a necessary ingredient of everyday life, having been integrated in mobile telephones, automobiles, television sets, gaming consoles and a host of other products that are integral to our daily routines. Almost without exception, the software driving these devices is written in some high-level programming language, abstracting over the computers' native assembly language. Not only does the high-level programming language function far more like a RAM than like a Turing machine, even the assembly language, with its ability to randomly access the device's internal memory, resembles a RAM far more than any other computational model.

Furthermore, the use of pseudocode, which is a facsimile of a high-level programming language, in the description (and subsequent analysis) of algorithms has become standard practice, and it, too, suggests a RAM by its use of high-level commands and its abstraction over data types.

As such, RAM complexity deserves a closer look and more focus than it currently receives. The field has been largely abandoned since the 1980s, after several breakthroughs led to the

remarkable conclusion that even fairly limited RAMs are able, in polynomial time, to solve every PSPACE problem. (That is, any problem that a Turing machine can solve using an amount of tape that is polynomial in the size of its input, regardless of the amount of runtime.) As evidence of this abandonment, consider the following comments by Hagerup (1998) regarding Paul and Simon's (1982) application of RAMs to sorting problems. Hagerup writes:

> The latter results did not bring about a revolution in the theory and practice of sorting because the corresponding algorithms were felt to cheat in a very real manner. In order to sort n integers of w bits each, they use intermediate results of nw or more bits, which makes them practically irrelevant and theoretically less interesting – the unit-cost assumption had never been intended to be stretched that far.

The new results essentially indicated that a RAM is far more powerful than any realistic computational device. Hagerup's description reflects the zeitgeist, stipulating that the drive for studying computational models should be fuelled by the industry's desire to create better computers and to equip these with optimal architectures. The conclusion that the RAM model is unrealistic shifted the focus of research to other, more promising avenues, such as the word-RAM (Kirkpatrick and Reisch, 1984).

We consider the present work to be mathematical research. In such a context, the criticisms of Hagerup and others are as persuasive as a call to abolish the study of infinite cardinals in set theory, because they are unreasonably large.[3] Certainly, large integers – the manipulation of which is the source of the RAMs' extraordinary power – emerge naturally in many mathematical disciplines. Indeed, much larger numbers than are used here can be found in research areas such as Ramsey Theory (Graham and Rothschild, 1971), analytic number theory (Skewes, 1933) and graph theory, e.g. when using the Robertson-Seymour machinery (see Lovász, 2006). In algorithm analysis outside the subdomain of RAMs, fast-growing functions that give rise to large numbers are just as prevalent (see, e.g., Tarjan, 1975; Brand, 2012).

[3]The reader is invited to seek out the history of this particular example. Cantor's work on cardinals did, indeed, meet with ferocious opposition of this kind exactly.

Some results regarding RAMs are about proving limits regarding what can be done with large numbers, demonstrating that in certain cases manipulation of large numbers can be simulated by actions on smaller numbers. In fields where the manipulation of large numbers is necessary, the machinery developed regarding RAMs can be used to provide mathematical workarounds (in the same sense as is often done with modular arithmetic).

On the flip side, the most interesting results regarding RAMs show that in some cases the ability to manipulate large numbers holds great powers. When seeking a technological motivation for investigations of this latter case, the situation is more complex, because the numbers used in RAMs are often so large that they cannot be written down, even when filling the entire observable universe with digits represented on the Plank scale.[4] However, whereas such an argument could have, in the past, been used to prove that numbers of so large a magnitude can never be used in practical computation, with the advent of quantum computation, which uses as its means of storage not the state of objects but rather their state distributions, we see that as our understanding of the physical universe improves it is not impossible for such hard limits to be exceeded. On the other hand, our engineering ability to manipulate computing objects seems to be forever constrained by what we can express, what Simon (1977) refers to as "feasible numbers". As such, knowing how to harness the power of simple operations on extremely large numbers may yet prove to be a much-needed technological key.

Furthermore, whereas conclusions reached on one computational model cannot be used directly for another computational model, the study of RAMs is interesting from a theoretical computer science point of view, where it can provide new insights into classical problems that have been at the forefront of research for years, in the context of Turing machines.

Ultimately, however, the prime motivation for studying RAMs is not the investigation of numbers at all, but rather the investigation of operations on numbers, either as standalone operations or in sets. The operations investigated here (and, for the most part, throughout RAM theory) are among the most natural operations, appearing everywhere in mathematics and computer science. Much as logic is the study of derivation, important because the validity

[4]This is not always the case. Paul and Simon (1982), mentioned earlier, do not require significantly more intermediate storage than the size of the input and output.

of derivations is the underpinning of the validity of all mathematics, so is the study of the power of operations important, because understanding this power is understanding the power of all mathematics.

The time is therefore perhaps ripe to re-visit the integer RAM model, hoping that it may shed light on some of the tantalising theoretical problems of computer science and beyond. As for the farther future: if one day we do find ways to manipulate large numbers, the implications of the tools of RAM theory on practical computations would be profound. This is theory waiting for technology to catch up, a situation not unfamiliar to the field of mathematics.

1.3 Nondeterminism

Consider, again, the class P of decision problems solvable by Turing machines in time that is polynomial in the length of their input. The introduction of nondeterminism as one possibility to vary over Turing machines was as early as Rabin and Scott (1959) (and possibly even as early as Turing, 1936). This was described via a process wherein a Turing machine's state was allowed to change simultaneously to several different new states. A computation was designated as "accepting" (returning a positive answer to a yes/no question) if at least one of the computation paths of the underlying state machine terminated in an accepting state. This idea introduced the new computational complexity class NP, those languages that can be recognised in polynomial time on a nondeterministic Turing machine.

An equivalent definition for NP (due to Scott, 1967), which is more relevant in the context of the present work, is that it is the set of problems that can be solved in polynomial time by a deterministic Turing machine, when this machine is also given as input a second, read-only tape containing a "hint" (sometimes referred to as a "certificate" or a "witness"). For a problem to be in NP, the machine should always terminate in polynomial time, and there should be some hint that will allow the machine to accept an input if that is the correct answer, whereas for an input that should not be accepted, the machine should never accept, regardless of the hint. It is clear that nondeterministic machines can be represented in this way, because the hint may simply indicate the accepting computation path.

In the early 1970s, Cook (1971) and, independently, Levin (1973) proved the existence of a specialised class of NP problems, which became known as NP-complete or NPC. The essence of NPC lies in the introduction of reducibility between different problems. Problem A is polynomial-time reducible to problem B if one is able to use an Oracle that solves A in O(1) time to solve B in polynomial time. Using this tool, Cook and Levin introduced problems that all NP problems can be reduced from. In other words, the complexity of any NPC problem is, up to a polynomial, an upper bound on the complexity of all NP problems. In particular, they showed that if any NPC problem was shown to be in P, then P=NP.

This gave, for the first time, an avenue to attack the problem of whether nondeterminism adds to the power of a Turing machine. Shortly after, Karp (1972) published a list of 21 natural problems in combinatorics and graph theory and showed these to be NPC, effectively pushing the P=NP question to the forefront of computer science research, where it has been ever since.

1.4 Randomness

Another problem, of a similar flavour, is the question of whether P=RP, that is, whether the availability of randomness increases computational power. Random computation was first introduced by Rabin (1963). Its power was demonstrated by Solovay and Strassen (1977) (and later by Rabin (1980)) who solved the question of random polynomial time primality testing, introducing algorithmic tools that have been utilised ever since, e.g. in implementations of the popular RSA cryptosystem (Rivest et al., 1978). However, this particular hallmark-problem of RP was shown by Agrawal et al. (2002) to also lie in P and, in general, the question of whether P=RP remains both open and of high interest.

Like in the case of NP, RP can be described both in terms of specialised (random) Turing machines, and in terms of regular Turing machines that are helped by special inputs. In the latter approach, RP is described as the set of decision problems that can be solved by a Turing machine that receives, in addition to its normal input, a second tape with random, uniformly and independently distributed bits, where by "solve" we mean that if the correct

answer is "no", this answer must be given always, whereas if the correct answer is "yes", it must be given with probability of at least 0.5.[5]

It is the insight into these and similar problems that may be the most important benefit from the study of RAMs. RAMs give us the opportunity to study, in a context similar to but distinct from Turing machines, the benefits afforded by the availability of nondeterminism and randomness. In RAMs, we model nondeterminism and randomness by the availability to the RAM of the certificate as an extra input number.

1.5 Arbitrariness

An even more extreme example for an extra input number for RAMs, and one that does not have any immediate equivalent in Turing machines, is an arbitrary number. This is a number, perhaps adversarially chosen, whose only distinction is that it is large. We define the class P-ARAM as the class of problems solvable in polynomial time by a RAM when this RAM is given as an additional input a number, such that there exists an integer N such that given any additional number $n > N$ the RAM is guaranteed to terminate with the correct answer. Because different RAMs are not polynomially reducible to each other, P-ARAM is a complexity class that must be parameterised by the underlying RAM.

Though the availability of an arbitrary number may seem unhelpful, previous studies (e.g. Bshouty et al., 1992), have shown for specific algorithms that such numbers contribute to the computational power of a RAM, but this power contribution has never yet been fully characterised.

1.6 Outline of the thesis

The present work answers, for several RAMs (differing in their choice of instruction sets), whether the availability of random or arbitrary numbers is of significant help to the run-time of algorithms.

[5] RP itself is just one of many setups for random computation. A more detailed overview of the topic can be found in Section 6.1.

The rest of this work is arranged as follows:

In **Chapter 2**, we give the formal definition of the RAM model, present the common variations of RAM models, survey the literature on RAMs and present known results. Additionally, we specify the Turing machine to be used as a comparison.

In **Chapter 3**, we present a somewhat restricted variation on the RAM model and prove that the restricted model is as powerful as the original to the extent that the restricted model and the original RAM model can simulate each other in real time, thus addressing an open question regarding the power of indirect addressing.

In **Chapter 4**, we define formally the notion of arbitrary large numbers in RAM programs. We determine the strength of arbitrary numbers in RAM computations that involve addition, Boolean operations, bit shifts and potentially also multiplication. We then go on to determine the strength of arbitrary numbers when division is also admitted as a basic operation.

In **Chapter 5**, we consider a generalisation of the model where an arbitrary large number is provided as input. In the generalisation, arbitrary large numbers can be generated by the model at will. We show that this capability increases computational power even under arithmetic complexity, and quantify this power exactly both with and without the availability of a division operation.

In **Chapter 6**, we review the existing models of randomised computation and how they have been applied to RAMs. We improve (and correct) existing results on the RAM equivalent of PP and finally close the longstanding open problem of characterising the power of the RP model in RAMs by introducing a new data type that facilitates parallelisation of universal computation. We also address the question of equivalence between variations of the stochastic RAM model.

Finally, in **Chapter 7**, we present our new findings in the context of known results from the field, speculate on possible implications for Turing machines, and describe potential questions for follow-up research.

The thesis is also accompanied by **Appendices**, as follows.

Appendix A presents a constant-time sorting algorithm, the existence of which is a direct corollary of Theorem 8 of Chapter 4, but whose implementation here is superior to that guaranteed by the theorem, in that it uses only polynomial-sized registers. Appendix B details how the techniques described are equally applicable to settle the open problem regarding the density of Friedman numbers. For example, use is made of $O^n_{a,m}$ vectors, defined in Chapter 3 and used throughout the thesis. The last appendix, Appendix C, contains full listings of implementations of algorithms discussed in Chapter 4.

Chapter 2

Background

2.1 The RAM model

The Random Access Machine (or RAM) is a generalisation of counter machines. Aho et al. (1975) provides a full and more formal description of RAMs. We summarise here:

Computations on RAMs are described by *programs*. RAM programs are sets of *commands*, each given a *label*. Without loss of generality, labels are taken to be consecutive integers. The bulk of RAM commands belong to one of two types. One type is an *assignment*. It is described by a triplet containing a k-ary operator, k operands and a target register. The other type is a *comparison*. It is given two operands and a comparison operator, and is equipped with labels to proceed to if the comparison is evaluated as either true or false. Other command-types include unconditional jumps and execution halt commands.

The execution model for RAM programs is as follows. The RAM is considered to have access to an infinite set of registers, each marked by a non-negative integer. The input to the program is given as the initial state of the first k_{in} registers, where k_{in} is algorithm-dependent. The rest of the registers are initialised to 0. Program execution begins with the command labelled 1 and proceeds sequentially, except in comparisons (where execution proceeds according to the result of the comparison) and in jumps. When executing assignments, the k-ary operator is evaluated based on the values of the k operands and the result is placed in the target register. The output of the program is the state of the first k_{out} registers at program

termination, where the choice of k_{out} is also algorithm-dependent. (Alternatively, and more resembling Turing machines, RAMs may be equipped with special halting instructions that either "accept" or do not.)

Types of RAM models differ in what the contents of their registers can be, what qualifies as operands, which operations are permitted, which comparisons are permitted and how complexity is measured. For the present purposes, we will use the most common choices among these. These are:

Register contents The contents of registers are non-negative integers. Other models include general integers, rational numbers, reals (Blum et al., 1989) and even elements of general abelian groups (Gaßner, 2001).

Operands Operands can be explicit integer constants, the contents of explicitly named registers or the contents of registers whose numbers are specified by other registers. This last mode is known as "indirect addressing".

We note that the target register of RAM assignments may likewise be specified either explicitly or by indirect addressing.

Operations The choice of operations for RAMs differs widely from paper to paper. There is no common consensus regarding it. We consider it at length in Section 2.2.

Comparisons We will mainly use the more restrictive set of comparison operators, including only equality "=". It is also possible to include the full complement of comparison operators. This is normally denoted simply as the inclusion of "$\leq$", from which all other comparisons can be calculated.

Complexity We will measure complexity in what is known as the "uniform cost" or "unit cost" model. This assigns the cost of a single time unit to each command executed. Other models assign higher costs for operations with larger operands.

Lürwer-Brüggemeier and Ziegler (2008) prefer this model because it is both convenient and largely realistic, whereas Stockmeyer (1976) provides deeper reasoning, based on the idea that the model is a good platform on which to examine the relative strengths

of various operations. Whichever the case, the uniform cost model is by far the most commonly used cost metric.

As can be seen, the RAM model closely resembles the intuitive notion of an idealised computer. Ben-Amram and Galil (1995) write

> "The random access machine (RAM) seems to be the computational model in widest use for algorithm design and analysis. The RAM is intended to model what we are used to in conventional programming, idealised in order to be better accessible for theoretical study. [It is] the standard platform for the study of algorithms."

Aho et al. (1975), a canonical text in the field of algorithm design, says about RAMs that they are

> "simple enough to establish analytical results but [...] at the same time accurately reflect the salient features of real machines".

This quote serves as an example of how the distinction between what RAM models are able to do and how "real machines" behave sometimes becomes blurred, even in a carefully written text. When Aho et al. later begin analysing specific algorithms, this distinction is omitted altogether. For example, regarding inserting an element into a linked list, Aho et al. write

> "Any reasonable translation to RAM code would result in an execution time for INSERT that is independent of the size of the list." (p. 46)

thereby glossing entirely over cost models for RAM operations and any hidden implications in the use of the loaded word "time" in this context.[1]

A key point, however, is that RAM complexities are typically not the same complexities as those afforded by Turing machines for the same tasks. This was exemplified in Section 1.1. Typically, the fact that RAMs are able to perform complex actions in constant time gives them

[1]The INSERT is performed in a constant number of operations. Stating that it is performed in constant *time* is only as "reasonable" as the assumptions of the unit-cost model. The author can attest, from personal experience, that inserting into a linked list that contains 2^{40} elements is far slower than doing so for a list that fits in memory.

considerable complexity advantages over Turing machines. However, the difference between the computational models is, perhaps surprisingly, not always in favour of the seemingly more powerful RAM. Stockmeyer (1976) proves that a RAM using arithmetic operations, working under the unit cost model, takes $\Omega(n)$ time to check if an n-bit integer is even - a procedure that can be accomplished in O(1) time by a suitable Turing machine.

The example above demonstrates also the non-triviality of the choice regarding which parameter to use for complexity calculations. Typically, in Computer Science, in saying that sorting by comparisons requires $\Omega(n \log n)$ comparisons one takes n to be the number of inputs. As is demonstrated in Chapter 3, this approach is not always viable: it is sometimes possible to compress any number of inputs into a bounded number of inputs. A stricter and more universal metric is the total bit-length of the input. This is the metric we will use here. It will make comparisons between RAM performance and TM performance an apples-to-apples comparison, because in TMs, the bit-length metric is the metric in use always.

2.2 RAM operations

There is no general consensus regarding which operations should be permitted implicitly in any RAM model. Schönhage (1979), in comparing between a large number of RAM variations, assumes only that RAMs must be able to increment a number. This seems to be a minimum requirement for all RAMs. Most other treatments also assume a basic decrement operation. Usually, addition and subtraction are also assumed (c.f. Regan, 1993). On the other hand, Trahan et al. (1994), in describing the closely-related model of the Parallel Random Access Machine (PRAM)[2] writes

> "The basic PRAM has unit-cost addition, subtraction, Boolean operations, comparisons and indirect addressing",

thus adding Boolean operations (which we denote here collectively as *Bool*) to the list of basic operations.

[2]PRAMs (see, e.g., Eppstein and Galil, 1988) are a family of computational models similar to RAMs but incorporating, additionally, explicit modelling of parallelisation. This work does not deal with PRAMs.

Notably, when registers can only be non-negative integers, these basic operations are augmented in such a way as to ensure a non-negative result: subtraction, for example, is replaced by *natural subtraction*, which is defined as $\max(a - b, 0)$ and is denoted by the symbol "$\dot{-}$".[3] Similarly, Boolean negation, "¬", is augmented so as to only affect binary digits up to (and including) the most significant 1-bit of the operand.

In addition to the basic operations presented ($\mathsf{inc}, \mathsf{dec}, +, \dot{-}, \mathit{Bool}$), there are other, more advanced operations. Listed here in approximate order of increasing computational power, these are:

Bit shift operations These include left shift, $a \leftarrow b \equiv a \times 2^b$, and right shift, $a \rightarrow b \equiv \lfloor a/2^b \rfloor$, which are sometimes introduced together and sometimes separately, sometimes without further restrictions and sometimes restricted to $b = 1$. Without restrictions, these are known as *variable shift* operations, whereas if b is required to be 1 (or, equivalently, to be bounded by any value that is independent of the input) then they are referred to as *bounded shift* operations. Variable shift is generally considered to be a stronger operation than bounded shift.

Multiplication Multiplication raises the computational power of a RAM model considerably. Trahan et al. (1992) and van Emde Boas (1990), for example, agree that having multiplication places RAM models outside the framework of sequential models and into the stronger class of parallel models. In the literature, RAMs with O(1) multiplication capability are often referred to as MRAMs.

Division Perhaps surprisingly, division was demonstrated by Simon (1981) to be much stronger than multiplication. The power boost afforded by division is not only super-polynomial, but is, in fact, larger than any elementary function, i.e. any function that can be described by a finite number of exponentiations and arithmetic operations. The availability of division can increase the power of some poly-time RAMs from PSPACE to ER.[4] (Compare this with Strassen (1973) and Granlund and Montgomery (1994),

[3] Natural subtraction will be taken here to share the same properties as normal subtraction in terms of operator precedence, etc..

[4] The definition of ER is given later on in this section.

where it is shown that in certain other RAM models division can be avoided altogether, and so entails no advantage.) Division comes in three flavors. Rational division is sometimes allowed in models where registers can store rational numbers. Where registers are integers, division is typically integer division, $\lfloor a/b \rfloor$, and denoted "$\div$". However, there is a weaker version known as "exact division" ("/"). It operates in the same way as integer division and as rational division. However, it is defined only when both other types of division yield the same answer. Simon (1981) conjectures that "$\div$" is stronger than "/", showing that #P $\subseteq$ PTIME-RAM$[+, \times, \div]$, whereas Schönhage (1979) proves PTIME-RAM$[+, \times, /] \subseteq FP$. Thus, this conjecture is a weaker form of the FP $\neq$ #P conjecture. (A definition of #P can be found in Valiant (1979). The class FP is defined in Rich (2008).)

In addition, some papers (e.g., Simon and Szegedy, 1992) introduce specialised operations, tailored for their needs. For example, one can introduce string manipulation operations that treat the RAM register contents as bit strings.

In this work we use the convention RAM$[op; comp]$ to denote a RAM with assignment operations from the set *op* and comparisons from the set *comp*. The operation "inc" and the comparison "=" are always assumed to be supported, and do not have to be listed. For clarity, all other operations are given explicitly. The syntax RAM$[op]$ indicates that $comp = \{=\}$. Operations appearing in brackets ("[ ]") within the operation list of a computational model are taken to be optional, in the sense that statements regarding the computational model hold both when the operation is part of *op* and when it is not. Indirect addressing is always assumed for RAMs. Where it is not, we use the syntax RAM$_0[op; comp]$, instead. Complexity classes are given their standard names, with the computational model appended to them as a suffix. When the computational model is not specified, it defaults to the Turing machine model. Where neither "SPACE" nor "TIME" is specified in the complexity class, "TIME" is the default.

So, for example, P-RAM$[+, [\dot{-}], \times]$ denotes the class of languages accepted in polynomial time by a RAM capable of unit-cost addition, multiplication, and optionally also natural subtraction, but not $\leq$ comparisons. (This is not to be confused with PRAM$[+, [\dot{-}], \times]$, which

is a class of parallel random access machines.) On the other hand, PSPACE denotes the class of languages accepted in polynomial space by a Turing machine. For Turing machines, space is measured by the number of tape cells used. For RAMs — by the highest-numbered register.[5]

2.3 Related work

The following are some of the known results regarding RAMs.

Schönhage (1979), Hartmanis and Simon (1974) and Bertoni et al. (1981) jointly prove the following equivalences:

$$\begin{aligned}\text{PSPACE} &= \text{P-RAM}[+, [\dot{-}], \times, \mathit{Bool}] \\ &= \text{P-RAM}[+, \dot{-}, \times, \div] \\ &= \text{P-RAM}[+, [\dot{-}], \times, \leftarrow].\end{aligned}$$

These extend results by Pratt et al. (1974) regarding Vector Machines (VMs), which are a model closely related to RAMs, where both Boolean and arithmetic operations are allowed, but on two different sets of registers. The original conclusions of Pratt et al. apply also to nondeterministic classes:

$$\begin{aligned}&\text{PSPACE}(= \text{NPSPACE}) \\ &\quad = \text{P-VM} = \text{NP-VM} \\ &\quad = \text{PSPACE-VM} = \text{NPSPACE-VM}.\end{aligned}$$

This is interesting, in that it shows that for vector machines P = NP. There is no benefit from nondeterminism. Pratt et al. conclude their discussion with the open question of whether unrestricted left shifting would make the VM model stronger, and whether allowing arithmetic and Boolean operations on the same registers would make the model stronger. They write:

[5]This method of measuring the space requirements of a RAM in registers rather than in bits may seem like a cheat. It is provided here to be used in the review of prior results. None of the results in this paper considers space-bounded RAMs.

> "Although the motivation in the previous section suggests that a vector machine should simply be a RAM with additional bit-wise Boolean and shift operations, we have not succeeded in finding any reasonable upper bound on the power of such a machine. The difficulty is that very rapidly growing functions may be computed simply by repeatedly shifting a vector a distance equal to its value. We have as yet found no use for the particular functions so computable; neither have we found a way to compute them in a small amount of space on a Turing machine."

This problem was ultimately solved by Simon (1977), who showed that

$$\text{P-RAM}[+, [\dot{-}], \leftarrow, \mathit{Bool}] = \text{PSPACE},$$

and by Simon and Szegedy (1992), who showed the same for P-RAM$[+, [\dot{-}], \times, \leftarrow, \mathit{Bool}]$.

Two general techniques are used to obtain many of these results. RAMs with finite-time execution can be described by "Computation Trees" (CTs). This is a model similar to RAM, where execution never repeats the same commands, but rather proceeds down one of the branches of an execution tree, chosen by the result of comparison operators. An even weaker model is the "Straight Line Program" (SLP), where comparisons are not supported and execution proceeds linearly from the start of the program to its termination. Schönhage (1979) developed a technique to assess the power of a RAM by considering SLPs of the same instruction set, and addressing the question of what resources (time and space) are required by a TM accepting two such SLPs as input, to determine which of the two SLPs calculates the larger number. (A more in-depth discussion of these notions follows in Section 4.2.)

Simon (1977) and Simon and Szegedy (1992) build on this and similar techniques by showing how such a TM can store the results of SLP computations, even those involving significant usage of the left-shift operation, in limited space. The general method of Simon (1977) is an "interesting bit" notation. Integers are denoted as runs of 1 bits and runs of 0 bits. An "interesting bit" is a place where one run ends and another begins. It is demonstrated that Boolean operations, addition, natural subtraction and bit shifts are all operations that do not expand the number of interesting bits in their operands very quickly. Note that the

"interesting bit" notation is a hierarchical notation, where the place in which a run ends and another run begins is, itself, denoted in interesting bit notation. In order to support multiplication, too, Simon and Szegedy (1992) extend the interesting bit notation from a notation for integers to a notation for polynomials.

In stark contrast to all results presented so far, Simon (1981) shows that adding the division operator makes a significant contribution to the computational power. Simon proves P-RAM$[+, \dot{-}, \leftarrow, \mathit{Bool}, /; \leq] =$ ER, where ER is the class of languages accepted by a Turing machine in time

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} n, \tag{2.1}$$

where n is the length of the input.

The technique used for this result, first developed by Yarbrough (1963), is the generation of "universal strings", these being numbers whose Boolean expansion includes every substring of length n, for some chosen n. It was later also used by Trahan et al. (1994).

2.4 The Turing machine model

Turing machines do not normally require an introduction. In much of the literature (and parts of the present work) they do not need an exact specification, either. This is due to the well known fact (see, e.g., Arora and Barak, 2009) that all Turing machines can simulate each other with at most a polynomial cost in time and space. This means that complexity classes such as P and PSPACE are well-defined without having to detail the exact model.

In parts of this work, however, the granularity of complexity classes we examine is different to what is defined by polynomial closure. We therefore specify our model exactly in this section.

The Turing machine we take throughout as a reference is a one-tape Turing machine working over a one-sided-infinite tape. The tape's alphabet is the binary digits, $\{0, 1\}$, where 0 doubles as the blank. The machine's head begins at the first position of the tape. Moving the head beyond the edge of the tape results in the machine halting and rejecting its input. The finite control has a special state for halting and accepting the input. The reading head

can go one element right, one element left or stay in the same place at any move. The input is the initial state of the tape, with the input's length being the position of the right-most "1". The machine has no output other than its accept/reject decision. As always, a non-halting Turing machine rejects its input.

We point out that this version of the Turing machine, despite being a standard description (see e.g. Radó, 1962, for an almost identical model), does not comply with the basic requirement of being Turing complete. Consider, for example, whether this machine can accept the language containing only the empty string (that is, the all-zeros tape) as its input. If there existed some machine that accepted the all-zeros tape, it would have had to do it in a certain finite number of steps, N. This means that the reading head could not have reached beyond point N in the tape. If the input was non-zero, but with the first "1" at position $N + 1$ or beyond, it would certainly have been accepted as well.

In order to make this description into a Turing complete one, one solution is to encode the input in a more machine-friendly manner. For example, one can write it using Manchester coding: each "0" is encoded as "01" on the tape and each "1" as a "10". This is already enough for Turing completeness.

The reasons we stress this issue are two-fold. First, we wish to make the point that encoding the input properly for each machine is standard practice. This fact serves as background to the discussion in Appendix A.3, where we examine the question of whether it is valid to re-encode algorithm inputs and outputs for the benefit of using a RAM_0 instead of a RAM. (For a more in-depth discussion of the need to properly encode a program's inputs, see Schroeppel (1973).)

Second, the need for input encoding, be it Manchester encoding or another method, changes the length of the input. In order to attain Turing completeness in the present model, some inputs will necessarily have to become longer. This demonstrates the fact that it is not particularly constructive (though it is well defined) to speak about an "input of length n". Instead, when we consider inputs of length n this should be taken to mean that the input is of length $\Theta(n)$, where the necessity for the use of big-O notation stems from the need to encode. We remark that other factors may cause the input length to shrink by a constant

multiplicative factor. For example, an increase in the tape's alphabet size (For a discussion, see Stearns et al., 1965). The need for the multiplicative ambiguity regarding the length of the input is therefore inherent.

The purpose of introducing a particular variant of a Turing machine here is to allow us, later on, to formally compare the computing power of Turing machines (and specifically, of *this* Turing machine) with the computing power of the RAM under study. For example, we may want to claim that the function computed by the RAM is in PSPACE. However, before we can do so we must first define in what sense a RAM can be said to compute at all. Turing machines accept their input in the form of a word on the input tape, and, typically, compute a Boolean function by halting in an accepting state or not. A RAM, on the other hand, receives any number of integers as inputs, and may terminate with any number of integers written in its registers (or may not terminate at all). In order to bring the RAM into the same playing field as Turing machines, we first restrict ourselves to considering only RAMs that accept a single input (in their first register), and whose output is deemed to be "accept" if at termination time the content of their first register is nonzero. In all other cases, including cases of non-termination, the output is deemed to be "reject". We call these *computing* RAMs.

Though the qualification "computing" precludes all but a select few RAMs, it is clear that in terms of sheer computing power nothing has been lost, as any function can be reduced to the form described. On the other hand, the model provides a straightforward analogy in the Turing machine world: by assuming the model of the Turing machine described here, the single input integer to the RAM can be represented by its binary digits on the Turing machine's tape at start-up, and the question of comparing RAM computational power with TM computational power becomes well-defined: a RAM computes a function that is in PSPACE if and only if there exists a Turing machine that, given the same input, returns the same output, and this Turing machine uses only an amount of tape that is polynomial in the length of the input.

The length of the input for a computing RAM is defined so as to be equivalent to the length of the corresponding input to a Turing machine. That is,

$$length(input) = \begin{cases} 0 & \text{if } input = 0, \\ \lfloor \log_2 input \rfloor + 1 & \text{else.} \end{cases}$$

So as not to confuse the "left" and "right" direction of binary representation with the "left" and "right" direction of tape motion, let us picture the tape as being finite to its right and infinite to its left.

2.5 Some redundant operations

This work concentrates on RAMs that are equipped with operation sets such as $\{+, \leftarrow, Bool\}$ and potentially also multiplication and division. In handling the computational tasks described, various other operations turn out to also be very useful. These include "$\dot{-}$", "$\rightarrow$", "mod" and the comparator "$\leq$". For the most part, these operations appear as optional in the operation sets used. In proofs of existence their presence is taken for granted, and in upper bound calculations they are treated as absent. The justification for this treatment lies in the fact that these operations can all be simulated in O(1) steps given the operations available in each proof where they are used.

Specifically:

- $\{+, Bool\}$ implies both "$\dot{-}$" and "$\leq$". (In fact, "$\dot{-}$" alone already implies "$\leq$", making all comparisons other than comparison for equality superfluous.)
- $\{\dot{-}, \times, \div\}$ (and therefore also $\{+, Bool, \times, \div\}$) imply both "$\rightarrow$" and "$\mathsf{mod}$".

To see how these reductions are implemented, consider first that had our RAMs been working on general integers (not necessarily non-negative), and numbers had been stored in their registers in standard two's complement notation (see Koren, 1993), then *Bool* would have included the standard bitwise negation operator $\neg$, and standard arithmetic subtraction ("$-$") would have been implementable as $a - b = a + \neg b + 1$. Because we deal with RAMs

working over nonnegative integers, we have a negation operator that only works up to (and including) the most-significant bit (the MSB) of its operand. This means, for example, that for any a, $a \vee \neg a$ is a number of the form $2^m - 1$, with the minimal m such that $2^m - 1 \geq a$. This is a number whose binary representation is a string of m "1"s. We therefore define a function

$$SET(a) = a \vee \neg a,$$

with which we can implement natural subtraction ("$\dot{-}$") as follows:

$$a \dot{-} b = \begin{cases} 0 & \text{if } SET(a) \vee SET(b) \neq SET(a), \\ 0 & \text{if } SET(a) = SET(b) \\ & \quad \text{and } (a + \neg b) \wedge (SET(a) + 1) = 0, \\ (a + \neg(b + SET(a))) \wedge SET(a) & \text{else.} \end{cases}$$

The rest of the implementations are straightforward:

$$a \leq b \Leftrightarrow a \dot{-} b = 0,$$

$$a \bmod b = a \dot{-} (a \div b) \times b,$$

$$a \rightarrow b = a \div (1 \leftarrow b).$$

Another operation that can be made redundant, but in a less universal fashion, is given by the following lemma.

Lemma 1. *For $op = \{\leftarrow, \rightarrow, [+], [\dot{-}], Bool\}$, if a $RAM[op]$ does not use indirect addressing and is restricted to shifts by bounded amounts, it can be simulated by a $RAM[op \backslash \{\rightarrow\}]$ without loss in time complexity. This result remains true also if the RAM can apply "$a \rightarrow b$" when b is the (unbounded) contents of a register, provided that the calculation of b does not involve use of the "$\rightarrow$" operator.*

Proof. We begin by considering the case of bounded shifts.

A RAM that does not use indirect addressing is inherently able to access only a finite set of registers. Without loss of generality, let us assume that these are $R[0], \ldots, R[k]$. The simulating RAM will have $R'[0], \ldots, R'[k+1]$ satisfying the invariant

$$\forall i : 0 \leq i \leq k, R[i] = R'[i]/R'[k+1].$$

To do this, we initialise $R'[k+1]$ to be 1, and proceed with the simulation by translating any action by the simulated RAM on $R[i]$, for any i, to the same action on $R'[i]$.[6] We do this for all actions except $R[i] \rightarrow X$, which is an operation that is unavailable to the simulating RAM.

To simulate "$R[j] \Leftarrow R[i] \rightarrow X$", we perform the following.

1. $R'[j] \Leftarrow R'[i]$.

2. $\forall x : x \neq j, R'[x] \Leftarrow R'[x] \leftarrow X$.

3. $R'[j] \Leftarrow R'[j]$ clr $\neg R'[k+1]$.

We note regarding the second step that this operation is performed also on $x = k+1$. The fact that k is bounded ensures that this step is performed in O(1) time.

Essentially, if "$R[j] \Leftarrow R[i] \rightarrow X$" is thought of as "$R[j] \Leftarrow \lfloor R[i]/2^X \rfloor$", Step 1 performs the assignment, Step 2 the division, and Step 3 the truncation.

The operator "clr" used above is the standard Boolean operator X clr $Y \stackrel{\text{def}}{=} X \wedge \neg Y$. However, in our case, the operation "$\neg$" has been tweaked so as to avoid negative outputs. That being the case, $X \wedge \neg Y$ in the operation set actually available to the RAM_0 may not clear the bits of X at exactly the bit positions of Y, as one may expect. Though "clr" is a Boolean operation and should, as such, be part of the full complement of Boolean operations at our disposal, if this operation is missing, for any reason, and our *Bool* includes only $\wedge$, $\vee$ and the modified $\neg$, it is still possible to simulate "X clr Y" by a slightly more complicated

[6] An action involving an explicit "1" (except for shifting by 1) will have the "1" replaced by $R'[k+1]$ in the simulation.

sequence of operations:

$$X \wedge \neg(Y \vee (\mathsf{inc}(X \vee Y \vee \neg(X \vee Y)))),$$

using only Boolean operations, and the increment operation which is implicitly assumed for all RAMs.

In order to support "$R[j] \Leftarrow R[i] \rightarrow X$" also when X is the product of a calculation, the simulating RAM also performs, in parallel to all of the above, a direct simulation that keeps track of the register's native values. In this alternate simulation, right shifts are merely ignored. Any calculation performed by the simulated RAM that does not involve right shifts will, however, be calculated correctly, so the value of X will always be correct. □

Chapter 3

Indirect addressing

3.1 The RAM_0 model

All RAMs have indirect addressing and an infinite number of registers. However, this feature of RAMs often introduces great difficulties for analysis, which is why some papers (e.g. Simon, 1977; Mansour et al., 1991a) only analyse RAMs that make no use of indirect addressing (and therefore necessarily utilise only a bounded number of registers). We denote the RAM model without indirect addressing as RAM_0. Regarding this choice, Mansour et al. (1991a) write

> "The power of indirect addressing has not been characterised, and it is not known whether it is a substantial advantage."

This statement is largely still true. Results with and without indirect addressing are typically handled in separate papers (e.g., Simon (1977) vs. Schönhage (1979)), with indirect addressing requiring more advanced mathematical techniques. In Ben-Amram and Galil (1992), sufficient conditions are given, under which indirect addressing is essential, in the sense that a machine without indirect addressing cannot simulate a machine with indirect addressing. Even in cases where indirect addressing can be simulated, it may entail a performance cost. For example, in Paul and Simon (1982) a computational model is described under which the symbol table problem is $\Omega(n \log n)$ time if and only if indirect addressing is not allowed. Furthermore, Paul and Simon (1982) proves that given indirect addressing, any program that achieves $\mathrm{o}(n \log n)$ time performance for this task must necessarily use space

unbounded by *any* function of n, the size of the input. The symbol table problem is the problem of finding, given $(x_1, \ldots, x_n, y_1, \ldots, y_n)$ as input, for each $i : 1 \leq i \leq n$, a value, t, such that $y_t = x_i$, or asserting than no such t exists.)

We show, however, that for powerful enough operation sets, indirect addressing entails no advantage. To clarify the term "powerful": in the case of RAMs in the unit cost model many innocuous-looking operation sets are surprisingly powerful. For example, a RAM$[+, \dot{-}, \times, \div]$ can simulate any PSPACE Turing machine in polynomial time (Schönhage, 1979). The operation set used by Ben-Amram and Galil (1995), being the starting point of our research, is $\{+, \dot{-}, \leftarrow, \rightarrow, \wedge\}$. It was shown by Simon (1977) to also span PSPACE in polynomial time. The operation sets we consider are $\{+, \dot{-}, \leftarrow, \rightarrow\}$ (not known to be as strong as the previous examples) for Theorem 2, and $\{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$ (shown in Simon (1981) to be even more powerful than the above) for Theorem 3.

One of the most general statements proven regarding the power of indirect addressing is given by Ben-Amram and Galil (1995):

Theorem 1 (Ben-Amram and Galil (1995))**.** *For the set of operations* $op = \{+, \dot{-}, \leftarrow, \rightarrow, \wedge\}$*, a RAM*$[op]$ *algorithm that takes* t *time and* s *space can be simulated by a RAM*$_0[op]$ *in* $t\alpha(s)$*, where* α *is the inverse Ackermann function.*[1]

It is conjectured in Ben-Amram and Galil (1995) that this is best possible. Note, however, that Ben-Amram and Galil's results pertain to a different model to that of the above-cited examples. It considers in the model of the *on-line RAM*, which is a model where the input is not given in advance, nor is the output read at the end of the process. Rather, the RAM is equipped with "READ" and "WRITE" instructions that can be accessed at any time. The initial state of the registers is assumed to be all zeroes. The on-line model is a stronger model in the sense that the on-line model is able to simulate off-line behaviour, while the converse is not necessarily true.

We complement the result of Ben-Amram and Galil (1995), by showing that for the standard (not on-line) RAM indirect addressing can be simulated at no cost to the run-time

[1] Here $\wedge$ denotes bitwise conjunction.

complexity of an algorithm, even using a smaller instruction set than the one used by Ben-Amram and Galil (1995). Using a slightly larger instruction set (adding the exact division operation, "/"), we demonstrate that real-time simulation is possible even in the on-line model.

Specifically, we claim

Theorem 2. *For the set of operations* $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$*, a RAM*$[op]$ *can be simulated by a RAM*$_0[op]$ *in linear time. (That is, a RAM*$[op]$ *requiring* n *steps can be simulated in* $O(n)$ *steps on a RAM*$_0[op]$*.)*

Theorem 3. *For the set of operations* $op = \{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$*, an on-line RAM*$[op]$ *can be simulated by an on-line RAM*$_0[op]$ *in real time. (That is, each operation of the on-line RAM*$[op]$ *can be simulated in* $O(1)$ *steps on an on-line RAM*$_0[op]$*.)*

3.2 Linear time simulation

In order to simulate a RAM on a RAM$_0$, we must first describe a method to encode the data from n registers (an unbounded number of registers) in a bounded number of registers. We encode the values $k_0, \ldots, k_{n-1}$ from n registers as follows:

- Let $Z = 2^m$ be a number greater than $\max(k_i)$.
- Let $V = \sum_{i=0}^{n-1} Z^i k_i$.

The triplet (m, V, n) contains all the information of the original values.

We use the following terminology.

Definition 1 (Vectors)**.** A triplet (m, V, n) of integers will be called an *encoded vector*. We refer to m as the *width* of the vector, V as the *contents* of the vector and n as the *length* of the vector. If $V = \sum_{i=0}^{n-1} 2^{mi} k_i$ with $\forall i : 0 \leq i < n \Rightarrow 0 \leq k_i < 2^m$, then $[k_0, \ldots, k_{n-1}]$ will be called the *vector* (or, the *decoded vector*), and the k_i will be termed the *vector elements*. Notably, vector elements belong to a finite set of size 2^m and are not general integers. It is well-defined to consider the most-significant bits (MSBs) of vector elements. Nevertheless, any n integers can be encoded as a vector, by choosing a large enough m.

Actions described as operating on the vector are mathematical operations on the encoded vector (typically, on the vector contents, V). However, many times we will be more interested in analysing these mathematical operations in terms of the effects they have on the vector elements. Where this is not ambiguous, we will name vectors by their contents. For example, we can talk about the "decoded V" to denote the decoded vector corresponding to some encoded vector whose contents are V.

We comment that the expression of input and output parameters as vectors allows the RAM_0 model surprising powers, compared to the ostensibly more powerful RAM. An example is given in Appendix A.

We begin by considering a simpler scenario than that of Theorem 2.

Lemma 2. *For the set of operations $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$, the first n execution steps of a RAM[op] can be simulated in $O(n)$ time by a RAM$_0$[op] program to which n is given as an additional input.*

Proof. Let $M = M_{op}(n, inp)$ be the largest number that can appear in any register of a RAM[op] initialised by inp in the course of its first n execution steps. We take inp, in this context, to be a K-tuple. We denote its i'th element by inp_i.

Our first claim is that there is a $\mathrm{RAM}_0[op]$ initialised by inp and n that calculates $M_{op}(n, inp)$ in $\mathrm{O}(n)$ execution steps. For our particular choice of op, an example of such a RAM_0 is one that first calculates

$$r_1 \Leftarrow \max\left(\{inp_i \colon 0 \le i < K\} \cup \{1\}\right), \tag{3.1}$$

and then goes through n rounds of

$$r_{i+1} \Leftarrow r_i \leftarrow r_i. \tag{3.2}$$

Here, "$\Leftarrow$" indicates assignment, as opposed to "$\leftarrow$" which indicates left shifting.

Let us temporarily assume that the RAM_0 supports multiplication. We want to store all registers of the original RAM as an encoded vector of width large enough to store M. One

way to do this is to choose $m = M$. It takes $O(n)$ steps to calculate this m value. Following this, loading the value stored in register i, for any i, can be simulated by

$$LOAD(i) = (V \to mi) \dot{-} ((V \to (mi + m)) \leftarrow m),$$

and storing a new value N in register i, $STORE(N, i)$, can be simulated by

$$V \Leftarrow V + (N \leftarrow mi) \dot{-} (LOAD(i) \leftarrow mi).$$

Hence, each instruction can be simulated in $O(1)$ time, following an $O(n)$-time setup, for a total of $O(n)$ operations.

We note that the only use made of multiplication is in calculating mi. Therefore, by switching to a vector of width $1 \leftarrow m$ rather than width m, this multiplication can be made into a left shift. □

As a side note, we mention that the LOAD and STORE procedures described above, though described here in the context of retrieving and altering an element in a properly-encoded vector, can also be considered natively in the natural number space, without resorting to vector terminology. If the number is thought of as being represented in base 2^m, these procedures allow access to a single digit in a number. The combination of digit access and integer arithmetic is, itself, quite powerful. An example is given in Appendix B.

We now move to the general case, where n is not available as part of the input.

Proof of Theorem 2. The proof of Lemma 2 allows us, in $O(n)$ time, to execute a $\mathrm{RAM}_0[op]$ program *simulate*(n) that simulates the first n execution steps of the target RAM. Without expanding the operation set, we can run on the same RAM_0 Algorithm 1.

If the simulated RAM does not halt, neither will the simulating RAM_0, but if it halts after n steps, the RAM_0 will terminate after having completed $O(n)$ simulation steps and $O(\log n)$ iterations of the loop, for a total of $O(n)$. □

Corollary 3.1. *For the set of operations* $op = \{\leftarrow, \to, Bool\}$*, a* $RAM[op]$ *can be simulated by a* $RAM_0[op]$ *in linear time.*

Algorithm 1 Linear-time simulation of indirect addressing

```
1: n ⇐ 1
2: loop
3:     simulate(n)
4:     if Simulation halts then
5:         return Halting state
6:     end if
7:     n ⇐ n ← 1
8: end loop
```

Proof. The only use of the operations "+" and "$\dot{-}$" in the proof of Theorem 2 is in the implementation of LOAD and STORE. However, they can equally be implemented using Boolean operations, instead:

$$LOAD(i) = (V \to mi)\ \mathsf{clr}\ (V \to mi \to m) \leftarrow m),$$

$$STORE(N, i)\colon\ V \Leftarrow V\ \mathsf{clr}\ (LOAD(i) \leftarrow mi) \vee (N \leftarrow mi).$$

□

3.3 Real time simulation

The proof of Theorem 3 is more involved, because it is no longer possible to have an O(n) preparatory step. For this proof, we require the vector

$$O^n_{a,m} \stackrel{\text{def}}{=} ((a \leftarrow nm) \dot{-} a)/((1 \leftarrow m) \dot{-} 1). \tag{3.3}$$

We note that this vector is equivalent to $a \times O^n_{1,m}$, but makes no use of multiplication, except for the multiplication "nm" which can be eliminated by the same technique as was used in the proof for Lemma 2, if m is a known power of 2. For $a < 2^m$, the resulting vector is $[a, \ldots, a]$.

Also, we require the following lemma:

Lemma 3. *It is possible to convert, in $O(1)$ operations from the set $\{\dot{-}, \div, \times, \leftarrow, \wedge\}$, a vector (m, V, n) to a vector (m', V', n) for any choice of m' so as to retain the original vector elements*

(given that m' is large enough to store the values that are in V). Furthermore, if m' is a large enough known power of 2, this can be done with the operation set $\{\dot{-}, /, \leftarrow, \wedge\}$ only.

Proof. We handle three cases:

Case 1: $m' \geq m(n+1)$ This is done by $V' = O^n_{V,m' \dot{-} m} \wedge O^n_{(1 \leftarrow m) \dot{-} 1, m'}$.

Case 2: $m'n < m$ This is done by $V' = V \bmod ((1 \leftarrow m) \dot{-} (1 \leftarrow m'))$, where $a \bmod b$ is calculated as $a - (a \div b) \times b$. To see this, recall that V is the value of the polynomial whose coefficients are the vector elements $k_0, \ldots, k_{n-1}$ evaluated at the point $Z = 2^m = 1 \leftarrow m$. In order to evaluate the polynomial at $Z' = 2^{m'}$, we take its value modulo $Z - Z'$. Because $Z \equiv Z' \pmod{Z - Z'}$, for any polynomial p, $p(Z) \equiv p(Z') \pmod{Z - Z'}$. The condition of Case 2 ensures that the desired V' is smaller than $Z - Z'$, so this residue can be evaluated without ambiguity.

Case 3: none of the above To handle this, first calculate (m'', V'', n) for a sufficiently large m'', using Case 1, then lower to the final m' using Case 2.

If m' is large enough, only the first case is needed, so no integer division is required (only exact division). Furthermore, if m' is known to be 2^k then the multiplication nm' in the calculation of $O^n_{(1 \leftarrow m) \dot{-} 1, m'}$ can be replaced by $n \leftarrow k$. For the calculation of $O^n_{V, m' \dot{-} m}$, n can be replaced by any value that is at least as large as n, without affecting the result after the $\wedge$ operation. As a result, the multiplication $n(m' \dot{-} m)$ can be replaced by $(m' \dot{-} m) \leftarrow n$, effectively replacing n by 2^n. □

We now continue to prove the main Theorem.

Proof of Theorem 3. To prove the theorem, one only needs to prove that the operations "LOAD" and "STORE" can each be simulated in O(1), after a one-time initialisation. (The initialisation receives no parameters, and therefore necessarily works in constant time.) This statement was made rigorously in Lemma 2 of Ben-Amram and Galil (1992). In essence, the RAM_0 is able to compute exactly as the RAM does. Its only difficulty is in reading the inputs to the computations from the registers and writing the computation results back to the

registers. Therefore, if it can simulate loading and storing properly, the rest of the problem is solved trivially.

Our simulation of the new "LOAD" and "STORE" works very much like the "LOAD" and "STORE" of Lemma 2. However, because we can no longer anticipate the size of integers the program will be required to handle (The RAM's first command may be a "READ" instruction, which will read from the user an integer of arbitrary size), we cannot use $m = M$ directly, as was done before. Instead, we begin by using $m = 1$, which is sufficient to store the initial all-zeroes state of the registers, then increase m whenever a "STORE" command requests storage of a value greater than or equal to 2^m.

In case a value r, $r \geq 2^m$, needs to be stored, the RAM_0 changes the width of the encoded vector, as per Lemma 3, from m to $m' = \max(m(n+1), r)$. The new width is necessarily large enough to ensure Lemma 3 can be implemented with a restricted set of operations. If the new width is not a known power of 2, we can use $1 \leftarrow m'$ as a width, instead of m'.

In this way, in $\mathrm{O}(1)$ operations we are able to widen the vector enough to allow the simulation of any assignment.

Correspondingly, we also update n to signify the greatest register address used thus-far in the program's execution. Updating n is a simple "max" operation, affecting neither m nor V. □

The proof of Theorem 2 is applicable not only to $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$, but also to any $op \supseteq \{+, \dot{-}, \leftarrow, \rightarrow\}$, given the restriction that there exists a $\mathrm{RAM}_0[op]$ that calculates any number at least as large as $M_{op}(n, inp)$ in $\mathrm{O}(n)$ steps. This condition holds for any reasonable set of operations. The proof of Corollary 3.1 applies similarly to any $op \supseteq \{\leftarrow, \rightarrow, Bool\}$, with the same restriction.

For the proof of Theorem 3, any $op \supseteq \{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$ will do, with no added restriction.

We have shown, therefore, that for RAMs with a sufficient set of operations, the use of indirect addressing entails no advantage.

Chapter 4

Arbitrary numbers

4.1 Introduction

4.1.1 RAMs with special numbers

Many papers deal with characterising broad complexity classes such as P and PSPACE for various RAMs, but there is a parallel effort to analyse the RAM complexity of specific problems, where specific complexities, rather than complexity classes, are required. Examples include Mansour et al. (1991a) (improving on Euclid's greatest common divisor algorithm), Mansour et al. (1989a) (calculating square root approximations), Mansour et al. (1991b, 1989b) (handling a class of functions including checking whether a number is a perfect square and calculating $\lfloor \log x \rfloor \bmod 2$), Bshouty et al. (1992) (calculating 2^{2^k}), Shamir (1979) (factoring a composite), Paul and Simon (1982) (sorting and related operations) and Lürwer-Brüggemeier and Ziegler (2008) (polynomial evaluation and related operations, building on tools from Bertoni et al. (1981)).

A repeating theme in these examples is the use of additional integers to speed up the computation. To understand how this can be done, consider first computation models over the reals, as in Blum et al. (1989). A reasonable computation model that includes reals must either forbid programs to hard-code a real number, or forbid any operation that allows O(1)

bit-access to a number[1]. A program that is able to store a real and access its bits can simply store in that single number the correct output associated with any possible input to it (e.g. by storing in the i'th bit a "1" if and only if the input corresponding to the integer i is in the language). Such a program will enjoy O(1) complexity regardless of the problem at hand.

The computation tree and the straight line program exhibit similar behaviours, but for different reasons. Unlike the RAM model, CTs and SLPs are *non-uniform* computational models. This means that for every problem-size, n, a completely different CT/SLP can be presented, and there does not need to be any way (reasonable or otherwise) of generating the CT/SLP for any given n. In a sense, this also allows pre-storing the answers to all possible questions, but instead of using a single real number capable of storing an infinite number of answers, the information is split into an infinite number of integers, each handling a finite number of possible inputs.

To see how additional integers can be used in the RAM context, consider the SLP presented in Lürwer-Brüggemeier and Ziegler (2008) to calculate the value of a polynomial p of degree at most d at x. Given a large enough Z,

$$p(x) = ((p(Z)(Z^{d+1} \div (Z - x))) \div Z^d) \bmod Z.$$

At face value, this seems like an O(1) time algorithm to evaluate polynomials (because Z^d and $p(Z)$ can be pre-calculated), but there is a catch: because Z must be "large enough" (with "large enough" being an attribute that is a function of the polynomial p and the input value x) Z must be determined only *after* p is given. However, now $p(Z)$ and Z^d must be calculated on the fly. If calculating $p(Z)$ is not faster than calculating $p(x)$, nothing has been gained.[2]

Incorporating "arbitrary" large values that need to satisfy additional constraints (such as one being a computed function over another) seems to give the RAM model power similar to that of a non-uniform model.

[1] See Mulmuley (1999) for a discussion of the power of bit operations in the context of the closely-related PRAM model.

[2] The proof of the formula used by Lürwer-Brüggemeier and Ziegler is somewhat involved. However, consider the simpler equality $p(x) \equiv p(Z) \pmod{Z - x}$, which follows immediately from $x \equiv Z \pmod{Z - x}$.

On the other hand, it appears that *unconstrained* arbitrary large numbers for which *no* function is precalculated can still make a difference in resulting complexities. Consider 2^{2^k}. Calculating this value can be accomplished trivially in $O(k)$ time by repeated squaring. For a RAM using no precalculated constants and for which repeated squaring is the fastest way to reach *any* large number, $O(k)$ time is also the best possible complexity. However, Bshouty et al. (1992) shows that by allowing the RAM access to a large enough but otherwise arbitrary integer, computation of 2^{2^k} can be reduced to $O(\sqrt{k})$ time.

The availability of arbitrary large numbers is therefore known to have a beneficial effect on the complexities of specific problems. In this chapter, we quantify this effect over general problems.

We begin by formally defining the question.

4.1.2 Formal definition of the model

The use of additional numbers as inputs to RAMs can be divided into several cases. The first case is the RAM analog of nondeterminism. Hartmanis and Simon (1974) prove that the following equalities among deterministic and nondeterministic RAMs:

$$\begin{aligned} \text{PSPACE} &= \text{P-RAM}[+, \dot{-}, \times, \mathit{Bool}] &&= \text{NP-RAM}[+, \dot{-}, \times, \mathit{Bool}] \\ &= \text{P-RAM}[+, \dot{-}, \times, \div, \mathit{Bool}] &&= \text{NP-RAM}[+, \dot{-}, \times, \div, \mathit{Bool}]. \end{aligned}$$

So, for RAMs with multiplication P = NP. Pratt et al. (1974) proves the same for vector machines, and Simon (1981) generalises this to all powerful RAM variations. Basically, the power to move nondeterministically from one command to the next in a RAM functioning in polynomial time allows one to obtain only a polynomial number of nondeterministic bits, or one of an exponential number of possibilities. Whereas for Turing machines this is (presumably) a significant leap, RAMs are able to generate and test in parallel an exponential number of options, e.g. by use of the universal strings of Simon (1981).

In order to allow RAMs a true benefit from nondeterminism, one must take a different view on what nondeterminism is. Although the classic method to describe nondeterminism in Turing machines is to allow the machine to move to more than one state simultaneously

(Rabin, 1963), an equivalent description is that a nondeterministic Turing machine is a deterministic Turing machine aided by a second input tape that includes a "certificate". In this view, a nondeterministic Turing machine is an "existential acceptor": it accepts an input (nondeterministically) if there exists a certificate for which the equivalent deterministic Turing machine will accept (deterministically). The class NP is the class of all languages that can be accepted in polynomial time by a deterministic Turing machine given a certificate. (A similar notion is that of the "universal acceptor", where an input is accepted only if it is accepted by a deterministic Turing machine given *any* certificate. This is an alternate definition of co-NP.)

This notion of nondeterminism can easily be generalised to RAMs, but, critically, in the context of RAMs the two approaches to nondeterminism are not equivalent. If we define the class P-NRAM[ ] as the class of languages accepted in polynomial time by a RAM given an additional integer as a certificate, then the power of the P-NRAM is greater than the power of the P-RAM. Simon (1979) shows that any recursively enumerable (r.e.) set can be recognised by an NRAM$[+, \times, \mathit{Bool}; \leq]$, with no need of division.

By contrast, from Hartmanis and Simon (1974),

$$\text{P-RAM}[+, \times, \mathit{Bool}; \leq] = \text{PSPACE},$$

so here P-RAM = NP-RAM $\subset$ P-NRAM.[3]

The idea of nondeterminism as existential acceptance allows us to study what can be done with an additional integer that is tailored specifically for our needs. The use of "arbitrary, large-enough" numbers is in many senses the opposite end of the scale: here, numbers can be chosen adversarially. This idea has never yet been properly formalised. We do so now, in the spirit of the "universal acceptor" leading to co-NP (but differing in details).

Definition 2 (ARAM)**.** The *ARAM* is the RAM model assisted by an arbitrary large number (or ALN). Formally, we say that a set S is computable by an ARAM$[op]$ in $f(n)$ time if there

[3]Recall that an NP-RAM is a RAM working in polynomial time with nondeterministic state transitions, whereas a P-NRAM is a polynomial time RAM using an integer certificate. An NP-RAM is equivalent to a P-NRAM using a certificate that is restricted to polynomial length.

exists a Boolean function $g(inp, x)$, computable in $f(n)$ time on a RAM[op], such that $inp \in S$ implies $g(inp, x) \neq 0$ for almost all x (all but a finite number of x) whereas $inp \notin S$ implies $g(inp, x) = 0$ for almost all x. Here, n conventionally denotes the bit length of the input, but other metrics are also applicable.

The parameter x is referred to as the program's *ALN*.

In this chapter we prove two main Theorems.

Theorem 4. *P-ARAM*$[+, [\bullet], [\times], \leftarrow, [\rightarrow], Bool] = PSPACE$

Theorem 5. *Any recursively enumerable (r.e.) set can be recognised by an ARAM*$[+, /, \leftarrow, Bool]$ *in* $O(1)$ *time.*

We see, therefore, that the availability of arbitrary numbers has no effect on the computational power of a RAM without division. However, for a RAM equipped with division, the boost in power is considerable, to the extent that any problem solvable by a Turing machine in any amount of time or space can be solved by an ARAM in O(1) time.

This result agrees with the observation that in all examples cited in Section 4.1.1 (and elsewhere in the literature) the use of arbitrary large numbers has so far been restricted to the context of the modulo operation, where it enables speeding up the calculation of polynomials beyond what is afforded by Horner's method, which is a method known to be optimal in the general case (Bürgisser et al., 1997, Theorem 6.5), when large numbers are not available.

Theorem 5 suggests, however, that the power of arbitrary large numbers in conjunction with division is far greater than the ability to compute polynomials quickly, and this power has thus far not been harnessed in the RAM literature.

4.2 P-ARAM$[+, \times, \leftarrow, \rightarrow, Bool]$

In order to prove Theorem 4, we begin by presenting a new proof for the following claim.

Theorem 6. *P-RAM*$[+, [\bullet], [\times], \leftarrow, [\rightarrow], Bool] = PSPACE$

This classic result first appeared in Simon and Szegedy (1992). In Section 4.2.1, we prove it using substantially different methods to those used by Simon and Szegedy (though the

derivation is heavily inspired by the original paper). Then, in Section 4.2.2, we show how the new methods can be adapted to handle arbitrary large numbers. This adaptation alone would have been reason enough to re-prove the original result. However, upon close inspection of the methods used by Simon and Szegedy, we discovered several errors in key points in Simon and Szegedy's derivation (as detailed in Section 4.2.1), which, put together, invalidate the original argument. As such, the proof appearing in Section 4.2.1 is, in fact, not a re-done proof, but rather the first appearance of a proof for this major claim.

4.2.1 Errata on Simon and Szegedy (1992)

We begin with a definition.

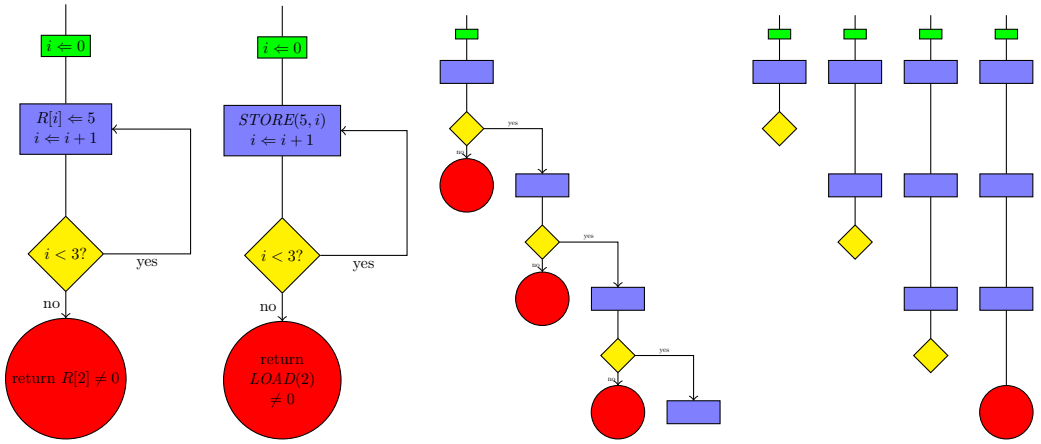
Definition 3 (Straight Line Program)**.** A *Straight Line Program*, or SLP[*op*], is a list of tuples, $s_2, \ldots, s_n$, where each s_i is composed of an operator, $s_i^{op} \in op$, and k integers, $s_i^1, \ldots, s_i^k$, all in the range $0 \leq s_i^j < i$, where k is the number of operands taken by s_i^{op}. This list is to be interpreted as a set of computations, whose targets are $v_0, \ldots, v_n$, which are calculated as follows: $v_0 = 0$, $v_1 = 1$, and for each $i > 1$, v_i is the result of evaluating the operator s_i^{op} on the inputs $v_{s_i^1}, \ldots, v_{s_i^k}$. The output of an SLP is the value of v_n.

A technique first formulated in a general form by Schönhage (1979) allows results on SLPs to be generalised to RAMs. Schönhage's theorem, as worded for the special case that interests us, is that if there exists a Turing machine, running on a polynomial-sized tape and in finite time, that takes an SLP[*op*] as input and halts on an accepting state if and only if v_n is nonzero, then there also exists a TM running on a polynomial-sized tape that simulates a RAM[*op*].

This technique is used both in Simon and Szegedy (1992) and in our new proof, and is a repeating theme in many similar proofs (e.g. Hartmanis and Simon, 1974). It is demonstrated visually in Figure 4.1.

The method in which this technique bounds the computing power of a RAM (in the sense introduced in Section 2.4) is by following a cascade of reductions.

The first reduction in our case is meant to avoid all uses of indirect addressing in the RAM. Though Schönhage's original method was different, in our case we can simply reduce from



(a) RAM (b) RAM$_0$ (c) CT (d) SLPs

Figure 4.1: Cascade of reductions

RAM to RAM$_0$ by using the methods described in Section 3.1. This is because the RAMs we need to consider all have operation sets that allow use of Corollary 3.1 (after application of Lemma 1, when right shifting is not in *op*).

Thus, a P-RAM can be simulated by a P-RAM$_0$. Figure 4.1(a) shows an example RAM program, and Figure 4.1(b) its transformation into a RAM$_0$ by elimination of indirect addressing. The next reduction, depicted in Figure 4.1(c), demonstrates how this RAM$_0$ can be reduced to a Computation Tree (CT). This is essentially a loop-unrolling procedure: a tree structure is built by following all possible execution paths through the RAM$_0$. In the figure, coloured boxes retain the meaning associated with their colours in the previous figures.

The tree resulting from the loop unrolling process may be infinite in size and height, but because the original RAM's execution is, by assumption, bounded by a polynomial, only a part of the tree of polynomial depth is ever really explored. The move from a RAM$_0$ to a CT eliminates all complex flow control. Whereas the RAM$_0$'s flow control is general, the CT describes a simple bifurcation process, where each split is the result of a comparison. As described in Section 2.5, given the operation "$\dot{-}$", all comparison operators can be described in terms of only "$a \neq 0$?".

The last step in the cascade of reductions, demonstrated in Figure 4.1(d), is the transformation of the Computation Tree into a sequence of Straight Line Programs (SLPs). Initially,

an SLP is built by following the CT's path up to the first bifurcation. This SLP is a sequence of operations from *op* only, without any comparisons. The return value of the SLP is the answer to whether the last term evaluated is nonzero. Given this bit of information, it is possible to tell which path the execution of the CT is to follow after the first comparison step. It is therefore possible at this point to create a new SLP that bypasses the first comparison (having already established its outcome) and continues down the correct path in the tree. This process continues, generating additional SLPs whenever a comparison is encountered, until the tree reaches a halting instruction, which is guaranteed to happen after at most a polynomial number of steps, at which point the last term computed is evaluated as either zero or nonzero (as in all previous steps), and the outcome of this comparison is the output of the final SLP. Necessarily, it will coincide with the output of the original RAM, so the simulation is correct.

This process is a formal guarantee that in order to prove that a polynomial time RAM can be simulated by a (for example) PSPACE Turing machine, it is enough to show that a PSPACE Turing machine can simulate the much simpler SLP model.

Formally, this is what is required from the Turing machine: it is to accept as input an SLP, should terminate in finite time, use up at most a polynomial amount of tape, and halt on an accepting state if and only if v_n is nonzero, where v_n is the final value calculated by the SLP.

As noted, the SLP in question has no "input", *per se*, but if the original RAM was initialised by $r[0] = x$, then x can be created as a v_i where i is $\mathrm{O}(\log x)$. Hence, adding the necessary commands $s_2, \ldots, s_i$ to the SLP is an operation that can also be simulated by a PSPACE TM.

Simon and Szegedy (1992) follow this scheme, and attempt to create such a Turing machine. In doing so, Simon and Szegedy's TM generates what the paper refers to as "monomials", but are, for our purposes, integers that are powers of two.

The main error in Simon and Szegedy (1992) begins with the definition of a relation, called "vicinity", between monomials, which is formulated as follows.

> We define an equivalence relation called *vicinity* between monomials. Let M_1 and M_2 be two monomials. Let B be a given parameter. If $M_1/M_2 < 2^{2^B}$ [...], then M_1 is in the vicinity of M_2. The symmetric and transitive closure of this relation gives us the full vicinity relation. As it is an equivalence relation, we can talk about two monomials being in the same vicinity (in the same equivalence class).

It is unclear from the text whether the authors' original intention was to define this relation in a universal sense, as it applies to the set of all monomials (essentially, the set of all powers of two), or whether it is only defined over the set of monomials actually used by any given program. If the former is correct, any two monomials are necessarily in the same vicinity, because one can bridge the gap between them by monomials that are only a single order of magnitude apart. If the latter is correct, it is less clear what the final result is. The paper does not argue any claim that would characterise the symmetric and transitive closure in this case.

However, the paper does implicitly assume throughout that the vicinity relation, as originally defined (in the $M_1/M_2 < 2^{2^B}$ sense) is *its own* symmetric and transitive closure. This is used in the analysis by assuming for any M_i and M_j which are in the same vicinity (in the equivalence relation sense) that they also satisfy $2^{-(2^B)} < M_i/M_j < 2^{2^B}$, i.e. they are in the same vicinity also in the restrictive sense.

Unfortunately, this claim is untrue. It is quite possible to construct an SLP that violates this assumption, and because the assumption is central to the entire algorithm, the proof does not hold.

We therefore provide here an alternate algorithm, significantly different from the original, that bypasses the entire "vicinity" issue.

Our proof adapts techniques from two previous papers. First, consider the technique used by Hartmanis and Simon (1974) to prove P-RAM[+, $\dot{-}$, ×, *Bool*] = PSPACE. This technique involves lazy evaluation of the operands one bit at a time, recursively where necessary, in order to avoid the need to store the operands whole: because multiplication is one of the operators considered, register contents may increase their bit-lengths at exponential rate, for which reason storing an entire register is impossible. An example of how this is done is given

in Algorithm 2. The example is of the (arguably) most complicated evaluation required, namely the evaluation of the product of two operands.

The code of Algorithm 2 is a C++ implementation. The class **Index** is a type designating a bit position. The code calculates the bit in the *index*'th position of the multiplication result of *arg1* and *arg2*.[4]

Algorithm 2 Lazy evaluation of a product

```
int Mult::eval(const Index& index) const
{
  int acc=0;
  for(Index i=begin();i<=index;++i) {
    acc>>=1;
    for(Index p1=begin();p1!=end();++p1) {
      Index p2=i−p1;
      acc+=arg1[p1]*arg2[p2];
    }
  }
  return (acc%2)!=0;
}
```

The procedure for calculating the product looks, on the face of it, like straightforward bit-by-bit multiplication. However, there are some subtleties.

First, note that when *arg1*[*p1*] and *arg2*[*p2*] are called, these are not table look-ups. Each of *arg1* and *arg2* is potentially exponentially long. Therefore, they are not stored. The bit in the required position in each is retrieved by recursive calls.

Second, note that only the bit in position *index* is actually retrieved by the function. Even though all previous bits were calculated, these intermediate results cannot be stored (because there are too many of them). Therefore, the next time any additional bit is required, the entire process must begin anew with the first bit.

Lastly, note that the algorithm does manage one integer, namely *acc*, which is used to store the carry. This can be done, because the carry can only be as large as the number of bits in the operands. If the operands are double exponential, their bit-length is exponential, and *acc* only needs to be of polynomial length. Furthermore, the depth of the recursion is

[4]This code fragment, and all other code fragments appearing in this section, are provided as part of an entire C++ implementation in Appendix C. Readers are encouraged to refer to the complete implementation for such details as how the parameters *arg1* and *arg2* are passed to the function.

at most n, the length of the SLP, so in total a polynomial amount of space is utilised by the simulating Turing machine.

This algorithm handles all operations that we need to simulate except for bit shifting. Bit shifting fails, because it allows operands to grow far more quickly than even double-exponentially, so, once it is admitted, neither the indexes nor the bit carry value can be stored any longer.

On the other hand, Simon (1977) describes a method that is able to handle bit shifts properly, but not multiplication. This is done by keeping track not of all the bits in the operands, but only of the "interesting" bits, where an interesting bit is a bit position where the bit-value of the operand is different from the bit-value of the preceding bit. Clearly, the list of interesting bits contains all the information regarding an integer. However, even though the number itself can grow rapidly with bit shifts, in the absence of multiplication the list of interesting bits remains only exponentially large. By what Simon himself, in his abstract, refers to succinctly as "a messy recursive method", his algorithm is able to perform calculations directly on the interesting bit data structure, while simultaneously generating this data structure on the fly because it is too large to store.

We note that the interesting bit notation is a recursive notation: the positions of the interesting bits must, themselves, be denoted in interesting bit notation, as they can be the result of many left shifts, and hence very large.

However, this method of keeping track of the interesting bits fails for multiplication, because in multiplication the number of interesting bits may grow double exponentially, rather than exponentially. Simon and Szegedy (1992) sum up the state of the art as follows.

> The only previous technique to deal with this situation uses the fact that there are only an exponential number of *interesting bits* in a register [and] cannot be used either, as multiplications can square the number of interesting bits. Thus if this approach is directly used, it would lead to a double exponential estimate of the number of interesting bits, placing the simulation into EXPSPACE. [...]

> As a result, any algorithm based on interesting bits must use space at least logarithmic in the number of interesting bits: the exponential tape simulation of the instruction set $\{+, \times, \leftarrow, Bool\}$ cannot be improved using these methods.

By contrast, at the end of the same paper Simon and Szegedy comment

> We note that it is a corollary to our results that the number of interesting bits remains only exponential even if multiplication is included in the operation set.

The method of the present proof is, fittingly enough, a hybrid of the two methods described, honing the advantages inherent in each. We begin by giving an explicit proof to Simon and Szegedy's (1992) commented corollary (following along the same reasoning as the original paper), and then follow this up by constructing a new algorithm around the observation. Specifically, we show that, even with multiplication, the list of interesting bits is always bounded by an exponential function, doing so by describing a set of positions of exponential size and proving that all interesting bit positions must necessarily lie within the set. A position in this set of exponential size can be described in a polynomial number of bits. We will show that it is possible to use this description as an index into the operands, to iterate over the interesting bits by use of this index and to perform calculations using it.

The proof is given as a sequence of lemmas.

Lemma 4. *In an SLP*$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$*, the number of interesting bits in the output* v_n *grows at most exponentially with* n*. There exists a Turing machine working in polynomial space that takes such an SLP as its input, and that outputs an exponential-sized set of descriptions of bit positions, where bit positions are described as functions of* $v_0, \ldots, v_{n-1}$*, such that the set is a superset of the interesting bit positions of* v_n*.*

Proof. Consider, for simplicity, the instruction set $op = \{+, \times, \leftarrow\}$. Suppose that we were to change the meaning of the operator "$\leftarrow$", so that, instead of calculating $a \leftarrow b = a \times 2^b$, its result would be $a \leftarrow b = aX$, where X is a formal parameter, and a new formal parameter is generated every time the "$\leftarrow$" operator is used. The end result of the calculation will now no longer be an integer but rather a polynomial in the formal parameters. The following are some observations regarding this polynomial.

1. The number of formal parameters is at most n, the length of the SLP.

2. The exponent of each formal parameter is at most 2^{n-k}, where k is the step number in which the parameter was defined. (This exponent is at most doubled at each step in the SLP. Doubling may happen, for example, if the parameter is multiplied by itself.)

3. The sum of all coefficients in the polynomial is at most $2^{2^{n-2}}$. (During multiplication, the sum of the product polynomial's coefficients is the product of the sums of the operands' coefficients. As such, this value can at most square itself at each operation. The maximal value it can attain at step 2 is 2.)

If we were to take each formal variable, X, that was created at an "$a \leftarrow b$" operation, and assign to it the value 2^b (Simon and Szegedy (1992) call this "the standard evaluation"), then the value of the polynomial will equal the value of the SLP's output. We claim that if p is an interesting bit position, then there is some product of formal variables appearing as a monomial in the result polynomial such that its standard evaluation is 2^x, and $p \geq x \geq p - 2^n$.

The claim is clearly true for $n = 0$ and $n = 1$. For $n > 1$, we will make the stronger claim $p \geq x \geq p - 2^{n-2} - 2$. To prove this, note that any monomial whose standard evaluation is greater than 2^p cannot influence the value of bit p and cannot make it "interesting". On the other hand, if all remaining monomials have degree smaller than $p - 2^{n-2} - 2$, the total value that they carry within the polynomial is smaller than $2^{p-2^{n-2}-2}$ times the sum of their coefficients, hence smaller than 2^{p-2}. Bits $p-1$ and p, however, are both zero. Therefore, p is not an interesting bit.

We proved the claim for the restricted operation set $\{+, \times, \leftarrow\}$. Adding logical AND ("$\wedge$") and logical OR ("$\vee$") can clearly not change the fact that bits $p-1$ and p are both zero, nor can it make the polynomial coefficients larger than $2^{2^{n-2}}$.

Incorporating "$\dot{-}$" and "$\neg$" into the operation set has a more interesting effect: the values of bit $p-1$ and p can both become "1". This will still not make bit p interesting, but it does require a small change in the argument. Instead of considering polynomials whose coefficients are between 0 and $2^{2^{n-2}}$, we can now consider polynomials whose coefficients are between $-2^{2^{n-2}}$ and $2^{2^{n-2}}$. This changes the original argument only slightly, in that we now need to

argue that in taking the product over two polynomials the sum of the absolute values of the coefficients of the product is no greater than the product of the sums of the absolute values of the coefficients of the operands.

Similarly, adding "$\rightarrow$" into consideration, we no longer consider only formal variables of the form $a \leftarrow b = aX$ but also $a \rightarrow b = \lfloor aY \rfloor$, where the standard evaluation of Y is 2^{-b} and $\lfloor \cdot \rfloor$ is treated as a bitwise Boolean operation (in the sense that, conceptually, it zeroes all bit positions that are "to the right of the decimal point" in the product).

We can therefore index the set of interesting bits by use of a tuple, as follows. If $i_1, \ldots, i_k$ are the set of steps for which $s_{i_j}^{op} \in \{\leftarrow, \rightarrow\}$, the tuple will contain one number between -2^{n-i-j} and 2^{n-i_j} for each $1 \leq j \leq k$, to indicate the exponent of the formal parameter added at step i_j, and an additional $k+1$'th element, between 0 and 2^n to indicate a bit offset from this bit position.

Though this tuple may contain many non-interesting bits, or may describe a single bit position by many names, it is a description of a super-set of the interesting bits in polynomial space. □

In Appendix C.1, such an enumeration is implemented by the method `Index::next`. We refer to the set of bit positions thus described as the *potentially-interesting* bits, or *po-bits*, of the SLP.

Lemma 5. *Let $\mathcal{O}$ be an Oracle that takes an $\mathcal{S} \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ as input and outputs the descriptions of all its po-bits in order, from least-significant to most-significant, without repetitions. There exists a TM working in polynomial space but with access to $\mathcal{O}$ that takes as inputs an $\mathcal{S}' \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ and the description of a po-bit position, i, of $\mathcal{S}'$, and that outputs the i'th bit of the output of $\mathcal{S}'$.*

Proof. Given a way to iterate over the po-bits in order, the standard algorithms for most operations required work as expected. For example, addition can be performed bit-by-bit if the bits of the operands are not stored, but are, rather, calculated recursively whenever they are needed. The depth of the recursion required in this case is at most n. Algorithm 3 describes such an implementation.

Algorithm 3 Lazy evaluation of a sum

```
int Add::eval(const Index& index) const
{
  int acc=0;
  for(Index i=begin();i<=index;++i) {
    acc>>=1;
    acc+=arg1[i]+arg2[i];
  }
  return acc%2;
}
```

The fact that iterating only over the po-bits, instead of over all bit positions, makes no difference to the results is exemplified in Figure 4.2.

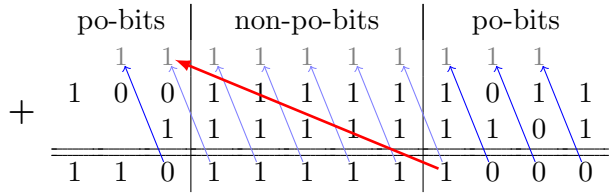


Figure 4.2: An example of summing two numbers

As can be seen, not only are the non-po-bits all equal to the last po-bit preceding them, in addition, the carry bit going over from the last po-bit to the first non-po-bit is the same as the carry bit carried over from the last non-po-bit to the first po-bit. Because of this, the sequential carry bits across non-po-bits (depicted in light blue in Figure 4.2) can be replaced by a single non-contiguous carry operation (the red arrow).

This logic works just as well for subtraction and Boolean operations. The only operation acting differently is multiplication. Implementing multiplication naïvely leads to incorrect results. Instead, we re-encode the operand bits in a way that reflects our original observation, that the operands can be taken to be polynomials with small coefficients in absolute value, though these coefficients may not necessarily be nonnegative.

The new encoding is as follows: going from least significant bit to most significant bit, a "0" bit is encoded as a 1 if preceded by a "1" and as 0, otherwise. A "1" bit is encoded as a 0 if preceded by a "1" and as -1, otherwise. This encoding procedure is known as Booth encoding. It is a special case of redundant binary encoding, and was introduced by Booth

(1951) specifically for the purpose of simplifying the handling of carry in multiplication (in computer hardware).[5]

It is easy to see that a number, A, encoded in regular binary notation but including a leading zero by a $\{0, 1\}$ sequence, $a_0, \ldots, a_k$, denoting coefficients of a power series $A = \sum_{i=0}^{k} a_i 2^i$, does not change its value if the a_i are switched for the b_i that are the result of the re-encoding procedure described. The main difference is that now the value of all non-po-bits is 0. An example of Booth encoding is given in Figure 4.3.

	po-bits				non po-bits					po-bits			
	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	0	1	1	1	1	1	1	0	1	1
=	1	-1	0	1	0	0	0	0	0	-1	1	0	-1
	5									-5			

Figure 4.3: Example of Booth encoding: $2555 = 5 \times 2^9 + (-5)$

After re-encoding, multiplication works as expected. The final algorithm is given in Algorithm 4.

To prove that the algorithm works correctly, consider the following.

1. When multiplying a single monomial with a single monomial, the answer is correct, because this case effectively requires only operations on contiguous bits.

2. When multiplying a general polynomial by a general polynomial, the answer remains correct, because all summations remain on contiguous bits, and the function calculated by the procedure is clearly bilinear.

3. No overflows can occur to non-contiguous bits, because coefficients are small. If a calculated coefficient is negative, this will result in its most significant bit being 1, with a carry of -1. For this carry, the algorithm is oblivious as to whether the carry is calculated over contiguous bits or not.

[5]The resulting multiplication procedure is known as Booth's multiplication. It is different to what is presented here, because Booth's multiplication does not use po-bits or lazy evaluation. However, much as was the case with addition and subtraction, what is shown here is an implementation of Booth's multiplication which, by some care, was made oblivious to the question of whether it iterates over all bits or just the po-bits and whether its operands are stored or evaluated on the fly.

Algorithm 4 Final code for evaluating a product

```
int Mult::make_pn(Command& command, const Index& index) const
{
  int curr_bit=command[index];
  if (index<begin()) {
    return 0;
  } else if (index==begin()) {
    return −curr_bit;
  } else {
    Index prev=index;
    −−prev;                                                          10
    int prev_bit=command[prev];
    return prev_bit−curr_bit;
  }
}

int Mult::eval(const Index& index) const
{
  int acc=0;
  for(Index i=begin();i<=index;++i) {
    acc>>=1;                                                         20
    for(Index p1=begin();p1!=end();++p1) {
      Index p2=i−p1;
      acc+=make_pn(arg1,p1)*make_pn(arg2,p2);
    }
  }
  return (acc%2)!=0;
}
```

4. The carry is limited in size by the number of contiguous bits, and so can always be stored directly.

Note that other than the ability to enumerate over po-bits in order, the only prerequisite for the multiplication procedure described is the ability to calculate "$p_2 \Leftarrow i - p_1$". However, as explained, po-bits correspond to offsets from the bit positions of monomials. For example, a bit position at offset c from the monomial X^2Y, where X, in the standard evaluation, is 2^α and Y is 2^β, is the bit position $c + 2\alpha + \beta$. Adding and subtracting indices can therefore be achieved by adding and subtracting the tuples (c, α, β), representing the coefficients. □

Lemma 6. *Let $\mathcal{Q}$ be an Oracle that takes an $\mathcal{S} \in SLP[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ and two po-bit positions of $\mathcal{S}$ and determines which position is the more significant. Given access to $\mathcal{Q}$, Oracle $\mathcal{O}$, described in Lemma 5, can be implemented as a polynomial space Turing machine.*

Proof. Given an Oracle able to compare between indices, the ability to enumerate over the indices in an arbitrary order allows creation of an ordered enumeration. The algorithm for doing so (with some error-handling removed) is given in Algorithm 5.

Algorithm 5 Finding the next index

```
Index& Index::operator++()
{
  Index rc=end();
  for(Index i=begin();i!=end();i.next()) {
    if ((i>*this)&&((rc==end())||(rc>i))) {
      rc=i;
    }
  }
  *this=rc;
  return *this;                                                    10
}
```

Essentially, the algorithm begins by choosing the smallest value, then continues sequentially by choosing, at each iteration, the smallest value that is still greater than the current value. This value is found by iterating over all index values in an arbitrary order and trying each in turn. □

Lemma 7. *Oracle $\mathcal{Q}$, described in Lemma 6, can be implemented as a polynomial space Turing machine.*

Proof. Recall that an index position is an affine function of the coefficients of the formal variables introduced, in their standard evaluations, as shown above, with the example $c + 2\alpha + \beta$. To determine which of two indices is the larger, we subtract these, again reaching an affine function of the same form. The coefficients themselves (in this case, c, 2 and 1) are small, and can be stored directly. Determining whether the subtraction result is negative or not is a problem of the same kind as was solved earlier: subtraction, multiplication and addition need to be calculated over variables, in this case α and β, instead of X and Y.

However, there is a distinct difference in working with α and β, as opposed to working with X and Y, in that X and Y are formal variables, but α and β are polynomials over formal variables. The calculation can, therefore, be transformed into addition, multiplication and subtraction, once again over X and Y.

Although it may seem as though this conclusion returns us to the original problem, it does not. Consider, among all formal variables, the one defined last. This variable cannot appear in the exponentiation coefficients of any of the new polynomials. Therefore, the new equation is of the same type as the old equation but with at least one formal parameter less. Repeating the process over at most n recursion steps (a polynomial number) allows us to compare any two indices for their sizes. □

In Appendix C.1, this algorithm is implemented by the function `cmp` and the method `Command::cmp`.

Proof of Theorem 6. We begin by recalling that P-RAM$[+, \leftarrow, Bool]$ = PSPACE was already shown by Simon (1977). Hence, we only need to prove P-RAM$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool] \subseteq$ PSPACE. This is done, as per Schönhage's (1979) method, by simulating a polynomial time SLP$[+, \dot{-}, \times, \leftarrow, \rightarrow, Bool]$ on a polynomial space Turing machine.

Lemmas 4–7, jointly, demonstrate that this can be done. □

We remark that Theorem 6 is a striking result, in that right shifting is part of the SLP being simulated, and right shifting is a special case of integer division. Compare this with the power of exact division, described in Theorem 8, which is also a special case of integer division.

4.2.2 Incorporating arbitrary numbers

The framework described in Section 4.2.1 can readily incorporate simulation of arbitrary large number computation. We use it now, to prove Theorem 4.

Proof of Theorem 4. Having proved Theorem 6, what remains to be shown is

$$\text{P-ARAM}[+, \dot{-}, \times, \leftarrow, \rightarrow, \mathit{Bool}] \subseteq \text{PSPACE}.$$

As demonstrated in the proof to Theorem 6, it is enough to show that an SLP that is able to handle all operations can be simulated in PSPACE.

We begin by noting that because the P-ARAM must work properly for all but a finite range of numbers as its ALN input, it is enough to show one infinite family of numbers that can be simulated properly. We choose $X = 2^\omega$, for any sufficiently large ω. In the simulation, we treat this X as a new formal variable, as was done with outputs of "$a \leftarrow b$" operations.

Lemmas 4–6 continue to hold in this new model. They rely on the ability to compare between two indices, which, in the previous model, was guaranteed by Lemma 7. The technique in which Lemma 7 was previously proved, was to show that comparison of two indices is tantamount to evaluating the sign of an affine combination of the exponents associated with a list of formal variables, when using their standard evaluation. This was performed recursively. The recursion was guaranteed to terminate, because at each step the new affine combination must omit at least one formal variable, namely the last one to be defined. Ultimately, the sign to be evaluated is of a scalar, and this can be performed directly.

When adding the new formal variable $X = 2^\omega$, the same recursion continues to hold, but the terminating condition must be changed. Instead of evaluating the sign of a scalar, we must evaluate the sign of a formal expression of the form $a\omega + b$. For a sufficiently large ω (which we assume ω to be), the sign is the result of lexicographic evaluation, as in Algorithm 6.

Lemma 7 also continues to be true for the SLP model incorporating an arbitrary large number. Hence, the model can be simulated in polynomial space, so any P-ARAM with the same set of operations can be simulated in polynomial space. □

Algorithm 6 Lexicographic evaluation of sign

```
1: function SIGN(aω + b)
2:     if a = 0 then
3:         return sign(b)
4:     else
5:         return sign(a)
6:     end if
7: end function
```

A full C++ implementation of the solution appears in Appendix C.2.

4.3 P-ARAM[+, /, ←, *Bool*]

In Section 4.2 we corrected and then built upon Simon and Szegedy (1992). In this section, we do the same for Simon (1981), in order to analyse the situation after adding division into the operation set (with or without multiplication) and to prove Theorem 5.

4.3.1 The new results

The main result from Simon (1981) is as follows.

Theorem 7 (Simon (1981))**.** *P-RAM*[+, $\dot{-}$, /, ←, *Bool*; ≤] = *ER.*

Here, ER is as defined in Equation (2.1):

Definition 4 (ER)**.** *ER* is the class of languages accepted by a Turing machine in time

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} n,$$

where n is the length of the input.

Despite being a classic result and one sometimes quoted verbatim (See, e.g., Trahan et al., 1992), the result is, in fact, erroneous.

We will show a number of logical fallacies in the proof, and will then correct these via a refinement of Simon's (1981) solution.

In a sense, the errata here are a case diametrically opposed to that of Section 4.2. On the one hand, despite the large number of errors and corrections, the revised proof of this section,

unlike that of Section 4.2, is merely a refinement over Simon's (1981) basic proof, using the techniques of the original paper, rather than a completely new algorithm. On the other hand, our *results* differ greatly from those reported by Simon.

To present them, we begin with two definitions, following from the definition of M in Section 3.2 and the definition of "reasonable" in Section 3.3, respectively.

Definition 5 (Expansion Limit)**.** Let $M_{op}(n, inp)$ be as defined in Section 3.2. We define $\mathrm{EL}_{op}(f(n))$ to be the maximum of $M_{op}(f(n), inp)$ over all inp for which $\mathrm{len}(inp) \leq n$. This is the maximum number that can appear in any register of a RAM[op] that was initialised by an input of length at most n, after $f(n)$ execution steps.

The subscript "op" may be omitted if understood from the context.

As a slight abuse of notation, we use $\mathrm{EL}(t)$ to be the maximum of $M_{op}(t, inp)$ over all inp of length at most n, when n is understood from the context and t is independent of n. (The following definition exemplifies this.)

Definition 6 (RAM-Constructability)**.** A set of operations op is *RAM-constructable* if the following two conditions are satisfied: (1) there exists a RAM program, el, that, given inp and t as its inputs, with n being the length of inp, returns in $\mathrm{O}(t)$ time a value no smaller than $\mathrm{EL}_{op}(t)$, [6] and (2) each operation in op is computable in $\mathrm{EL}(\mathrm{O}(l))$ space on a Turing machine, where l is the total length of all operands and of the result.

Any "reasonable" set of operations, and certainly any subset of the operations mentioned in this work, is RAM-constructable. The restriction on computability can, in fact, be made stricter (e.g. requiring that operations be computable in $\mathrm{EL}(l + \mathrm{O}(1))$ space, instead of in $\mathrm{EL}(\mathrm{O}(l))$ space) without this excluding any set of operations that would be deemed "reasonable". When choosing a stricter definition of RAM-constructability, the results developed here can be made correspondingly sharper. The exact details are left to the reader, but are straightforward from an examination of the constructive nature of the proofs.

Our results are as follows.

[6] This is not guaranteed from the definition of EL, for example because the definition of EL does not require the calculation of EL to be uniform over n.

Theorem 8. *For a constructable op* $\supseteq \{+, /, \leftarrow, Bool\}$ *and any function* $f(n)$*,*

$$\begin{aligned} O(f(n))\text{-}RAM[op] &= O(f(n))\text{-}RAM_0[op] \\ &= EL_{op}(O(f(n)))\text{-}TM \\ &= N\text{-}EL_{op}(O(f(n)))\text{-}TM \\ &= EL_{op}(O(f(n)))\text{-}SPACE\text{-}TM \\ &= N\text{-}EL_{op}(O(f(n)))\text{-}SPACE\text{-}TM. \end{aligned}$$

We remind that the notation $f(n)$-TM means "the set of all functions that can be calculated by a Turing machine (as described in Section 2.4) in $f(n)$ execution steps". The remaining notations refer to nondeterministic Turing machines, space-bounded Turing machines and nondeterministic space-bounded Turing machines.

To compare these results with those of Simon, let us first quantify the relation between EL and ER, for $op = \{+, \dot{-}, /, \leftarrow, Bool\}$, which is the operation set investigated by Simon.

Simon introduces a restricted tetration function, *stexp*, defined by $stexp(n) = {}^{n}2 \stackrel{\text{def}}{=} \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} n$, and defines ER as *stexp*(n)-TIME, where n is the input length. Thus, Simon's ER is equivalent to *stexp*(n)-TIME. For the chosen *op*, the EL function is as described in Equations (3.1) and (3.2). Thus, in a computing RAM with a nontrivial input, $\mathrm{EL}(0) = inp$. This places it at $stexp(0) \leq \mathrm{EL}(0) \leq stexp(\mathrm{len}(inp))$. In all further steps, $\mathrm{EL}(n+1) = \mathrm{EL}(n) \leftarrow \mathrm{EL}(n)$, so $2^{\mathrm{EL}(n)} \leq \mathrm{EL}(n+1) \leq 2^{2^{\mathrm{EL}(n)}}$. Thus, $\mathrm{EL}(f(n))$ is *stexp*$(\mathrm{O}(f(n))$, where n is the length of the input. In particular, we get that

$$\text{P-RAM}[+, \dot{-}, /, \leftarrow, Bool; \leq] = stexp(\text{Poly}(n))\text{-TIME},$$

as opposed to Simon's (1981)

$$\text{P-RAM}[+, \dot{-}, /, \leftarrow, Bool; \leq] = stexp(n)\text{-TIME}.$$

These results are substantially different. Whereas Simon equates the computing power of all polynomial RAM[*op*] machines, regardless of the polynomial degree, to a single class, ER,

we show that the equality is really to $\mathrm{EL}_{op}(\mathrm{O}(f(n)))$, which varies directly with the run-time complexity of the RAM. Taking *op* to be Simon's chosen instruction set, any RAM algorithm that requires $\omega(n)$ time is, in fact, *not* in ER. A RAM algorithm that requires $\Theta(n)$ time may or may not be in ER, because ER is equivalent to $\mathrm{EL}_{op}(n)$ but what is actually required is $\mathrm{EL}_{op}(\mathrm{O}(n))$. It is not difficult to find cases where ER does not suffice.

In short, the new result implies for polynomial-time RAMs that their computational power is far greater than what was previously believed.

For RAM algorithms that require $\mathrm{o}(n)$ time, Simon's theorem is correct, for a large enough n, but our bound is sharper. For example, taking $f(n)$ to be $\mathrm{O}(1)$, we get

$$\begin{aligned}\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE} \\ = \mathrm{NEXPSPACE} \subset \mathrm{O}(1)\text{-}\mathrm{RAM}_0[op] = \mathrm{O}(1)\text{-}\mathrm{RAM}[op]\end{aligned}$$

for any RAM-constructable $op \supseteq \{+, /, \leftarrow, \mathit{Bool}\}$.

The relationships noted above between TM complexity classes are well known. Equalities noted among space classes are a result of Savitch's Theorem (Savitch, 1970). Known results that are, in fact, sharper than what appears above are $\mathrm{P} \subset \mathrm{EXP}$ (Hartmanis and Stearns, 1965; Hennie and Stearns, 1966), $\mathrm{NP} \subset \mathrm{NEXP}$ (Cook, 1973; Seiferas et al., 1978; Žák, 1983) and $\mathrm{PSPACE} \subset \mathrm{EXPSPACE}$ (Stearns et al., 1965; Geffert, 2003; Seiferas, 1977). The sharpness of the comparison between TM complexity classes and RAM complexity classes is likewise derived by these known theorems: to prove

$$\mathrm{EXPSPACE} \subset \mathrm{O}(1)\text{-}\mathrm{RAM}[op]$$

we merely apply

$$\mathrm{EXPSPACE} \subset \mathrm{EXPEXPSPACE} \subseteq \mathrm{EL}(\mathrm{O}(1))\text{-}\mathrm{TM} = \mathrm{O}(1)\text{-}\mathrm{RAM}[op].$$

The result continues to be sharp given any bounded number of finite exponentiations, for the same reason.

A simple constant time sorting algorithm, being an example for

$$\mathrm{P} \subset \mathrm{O}(1)\text{-}\mathrm{RAM}_0[op]$$

is given in Appendix A.

Our results also differ from Simon's in that they are more general, being applicable to a large class of operation sets, whereas the original result is restricted to a single specified set. When more powerful operations are admitted into *op*, such as ones calculating the Ackermann function (Conway and Guy, 1996) or numbers in Knuth's up-arrow notation (Knuth, 1976), EL can become arbitrarily large.

We define the complexity class PEL[*op*] to be the union of $\mathrm{EL}_{op}(f(n))$-TM over all polynomial $f(n)$. A special case of Theorem 8 is that for a constructable $op \supseteq \{+, /, \leftarrow, Bool\}$, P-RAM[*op*] = PEL[*op*]. When PEL is used without qualification of an operation set, *op* is taken to be $\{\leftarrow\}$. It is trivial to show that PEL = PEL[*op*] for all $op \subseteq \{+, \dot{-}, \times, \div, /, \leftarrow, \rightarrow, Bool\}$ where left shifting is in *op*.

Note that by their very definition as complexity classes of languages that can be recognised by deterministic TMs working under certain defined time and space constraints, closure under union, intersection and complement is guaranteed for all EL-TIME and EL-SPACE classes, including PEL.

4.3.2 Errata on Simon (1981)

Before going into the errors in Simon (1981), it is worth noting that the wording of Simon's result is for a RAM that is clearly over-equipped. A common theme in the study of the power of operation sets is the attempt to distil the minimal set of operations required for each each proof. Simon (1981) is even given as an example (in Ben-Amram and Galil, 1995) for doing so. And yet, by the methods of Section 2.5 it is clear than anything that can be done using a P-RAM$[+, \dot{-}, /, \leftarrow, Bool; \leq]$ can equally be attained by a P-RAM$[+, /, \leftarrow, Bool]$. In fact, the operation "$\leq$" is never used in Simon's proof.

Furthermore, Simon is misrepresenting his own result by claiming P-RAM$[+, \dot{-}, /, \leftarrow, \mathit{Bool}; \leq] =$ ER, as his proof at no point attempts to demonstrate an inclusion of the form P-RAM$[+, \dot{-}, /, \leftarrow, \mathit{Bool}; \leq] \subseteq$ ER. Even if the proof had contained no errors, it would therefore have only sufficed to claim ER $\supseteq$ P-RAM$[+, \dot{-}, /, \leftarrow, \mathit{Bool}; \leq]$.

The serious errors in Simon (1981) begin, however, with the Turing machine model used by Simon. Simon writes:

> For technical reasons,... we shall consider a special model of Turing machine: it has a single tape, it is oblivious, and, if the length of the tape used is s, the position of the read/write head at time t is $t \bmod s$. For our purposes this is not a serious restriction: we are interested in super-polynomial speedups and it is not hard to verify that given a $T(n)$ time bounded ordinary multitape Turing machine, there is a one tape machine with the restrictions above that accepts the same language, using at most $T^2(n)$.

It is interesting to note that Simon uses the terminology "oblivious Turing machine", but not in the sense used by Pippenger and Fischer (1979), which is the classical source for the oblivious model. An oblivious Turing machine is one that does not contain right/left motion instructions for the reading head as part of its finite control. Rather, the reading head is moved by a predefined infinite sequence (possibly, one computed by a second Turing machine, with no access to the first machine or to its input). Pippenger and Fischer show that any Turing machine working on any number of tapes can be simulated by an oblivious Turing machine using only two tapes, so that an algorithm requiring n steps from the original machine will require $\mathrm{O}(n \log n)$ steps of the simulating machine. Pippenger and Fischer's solution requires rather complex motions of the reading head and constant usage of the second tape both as a stack, keeping track of the motions of the reading head, and as a buffer for copy operations. The algorithm uses a large number of copy operations to ensure that the relevant data does, eventually, meet the reading head. Without a second tape, the algorithm would have required a significantly larger run-time, beyond the $\mathrm{O}(n^2)$ predicted by Simon.

There are two problems with Simon's simplistic "$t \bmod s$" reading head motions, which Pippenger and Fischer's algorithm solves by far more complex means. The first problem is

that Simon is considering "a $T(n)$ time bounded ordinary multitape Turing machine", and yet simultaneously bounds the tape length used by s. These are two independent restrictions. The machine should therefore not be taken as a "time-bounded" machine, but rather as a "time-and-space bounded" machine. In fact, Simon's machine, restricted to work on a tape of predetermined length s, is no more than a finite automaton or a look-up table.

The difference between a space-bounded TM and a look-up table is in that the TM model is uniform. It is not bounded by s, a length of tape that is a constant, but rather by $s = s(n)$, where n is the length of the input. It must be able to handle any input, of any length, without change to its finite control. (For example, a PSPACE TM is one where s is determined as a polynomial function of n.) Consider, now, Simon's model. Although he omits the details of the model, leaving it at "it is not hard to verify", his claim that the machine uses "at most $T^2(n)$ steps" (note the lack of any big-O) suggests that what is being done is that the head moves one element to the left at each execution step. If the (non-oblivious) algorithm being simulated requires a motion to the left at this step, the simulating algorithm continues as usual. If, however, the simulated algorithm requires a motion to the right, the simulating machine continues $s - 1$ steps, noting that $s \leq T(n)$, and then writes the desired new element into the tape, having returned to the correct position after having gone full-circle. Unfortunately, this does not work, at least not in the trivial manner described. The reason for this is that this requires the finite control of the Turing machine to be able to calculate a time delay of $s - 1$ steps. Clearly, this requires at least $s - 1$ states. For s that is a function of n, not bounded by any number, this cannot be handled by a uniform model.

Whereas the delay problem can be fixed by appropriate handling, the fact that s is a function of n means that "$t \bmod s$"-type motions require the motions of the reading head to be a function of the input. Hence, this is no longer an oblivious model.

Lastly, Simon's main idea is in the use of "proof strings" (being, essentially, tableaux). The critical point in understanding the complexity of the RAM programs designed by Simon is in understanding the length of these proof strings. In the paper itself, this is not discussed. At one point, Simon suggests that the process takes $O(x)$ steps, where x is the input (note: not the *length* of the input). Clearly, however, that cannot be the case: Turing machines

are known for their ability to use tape and time disproportionate to their input, and any tableau-based solution would need to reckon with this fact.

Given the previous discussion regarding s, Simon's avoidance of an exact calculation is understandable: the initial assumption of non-uniformity would now render any attempt to analyse the complexity meaningless. Instead, Simon opts to assume, without reason, that the $\mathrm{O}(x)$ steps (being, essentially, the linear case in which Simon's original results coincide with our corrected results) can be generalised to a polynomial number of steps without this having any effect on the computational power.

We attempt, in Section 4.3.3, to re-introduce Simon's ideas while side-stepping the logical pitfalls.

4.3.3 A revised proof

We prove Theorem 8 by first establishing a sequence of lemmas.

Lemma 8 (Step by step simulation)**.** *A TM can be simulated by a $RAM_0[\leftarrow, \rightarrow, Bool]$ using only bounded shifts. A TM run requiring n execution steps can be simulated in this way by $O(n)$ RAM steps. In the simulation, advancing the TM by a single step is simulated by an $SLP[\leftarrow, \rightarrow, Bool]$.*

Proof. The simulation will store the state of the TM on three registers: *tape*, *head* and *state*. Register *tape* will store the current state of the tape. The format for this is the direct one described in Section 2.4, where the bits on the tape are translated to the binary digits of the number on *tape*. At start-up, this register is initialised by

$$tape \Leftarrow inp.$$

Register *head* will store the position of the reading head. The format for doing so is that if the reading head is at position i on the tape, the value of *head* will be 2^i, this being a "1" at binary digit position i and "0" everywhere else. At start-up, the register is initialised by

$$head \Leftarrow 1.$$

This places the head at the first position on the tape. (The count of position numbers begins at 0.)

Register *state* will signify the instantaneous state of the finite control. We number the states of the TM arbitrarily from 0 to $k-1$, where k is the total number of states. By convention, state 0 will be the initial state of the machine. The format for storing the state number on the *state* variable is that the variable equals the state number times *head*. In other words, the number is stored starting at the bit position of the reading head. At start-up,

$$state \Leftarrow 0.$$

Let $c = \lceil \log_2 k \rceil$, this constant being the number of bits required to store the state number. The transition function of the Turing machine is a function from the current state and the value of the tape currently under the reading head to a new state, a new value and a new position of the reading head (which is at most one position from its previous position).

Consider the function only for the new value under the reading head. This is a function from $c+1$ bits to 1 bit, and can therefore be described by a finite number of Boolean operations, e.g. by use of a Karnaugh map (Karnaugh, 1953). Applying this Boolean function requires, however, that all operands be available as bits. Consider, now, the numbers $state \rightarrow 0, \ldots, state \rightarrow (c-1)$. In each of these numbers a different bit of the original state description is aligned with the bit position of the reading head. Boolean algebra on these numbers along with the number *tape* will result in the output value having the correct new bit value for *tape* in the bit position that is under the reading head (though its other values may not signify anything meaningful). Let us refer to this numerical result as *output*.

To calculate the new value for *tape*, we note that the value under the reading head is the only one that requires updating. All other bit positions retain their original values. Hence,

$$tape \Leftarrow (output \wedge head) \vee (tape \ \mathsf{clr} \ head)$$

is the correct update.

Computing the value of the i'th bit of the new state number is attained by the same means as computing the new tape bit value. Let $outstate_i$ denote the result of applying Boolean operations on *tape* and $state \to 0$ through $state \to (c-1)$ to calculate this i'th bit. The update of *state* is

$$state \Leftarrow ((outstate_0 \wedge head) \leftarrow 0) \vee \cdots \vee ((outstate_{c-1} \wedge head) \leftarrow (c-1)).$$

Lastly, we need to update the variable *head*. To do so, we calculate three functions from *tape* and $state \to 0$ through $state \to (c-1)$, namely *IsRightMotion*, *IsStay* and *IsLeftMotion*, calculating whether the head is to move to the right, stay in place, or move to the left, respectively. The update required is

$$head \Leftarrow ((IsLeftMotion \wedge head) \leftarrow 1) \vee (IsStay \wedge head) \vee ((IsRightMotion \wedge head) \to 1).$$

This simulation, as described above, is already correct. However, we will change it slightly in order to make it more elegant. Specifically, we wish to avoid the scenario of the reading head dropping off the edge of the tape, which may happen if $head = 1$ and the next motion is a motion to the right, in which case the present calculation yields 0 for the new value of *head*.[7]

To avoid this, we introduce a new element into the simulation. This is a constant. Its name is *boundary* and its value is 1. Furthermore, we add one new state into the state machine, this being a rejecting halting state, signifying that the tape head has dropped off the tape. (Adding a new state may cause an increase in c, requiring a change in the update functions.) The boundary condition is a new bit in the input of the update functions. Its value is $boundary \wedge head \wedge IsRightMotion$, and it triggers a transition to the new rejecting state, and ultimately no head motion. □

Lemma 9 (Bounded tape step-by-step simulation)**.** *A TM working on a bounded tape of length s can be simulated by a $RAM_0[\leftarrow, \to, Bool]$ using only bounded shifts if the RAM is*

[7] The TM being simulated may legitimately move the reading head beyond the edge of the tape as a method of rejecting the input.

given $B = 1 \leftarrow s$ as an additional input. The simulation is required to be uniform in this input. Note that the parameter s is not given as an input to the RAM. A TM run requiring n execution steps can be simulated in this way by $O(n)$ RAM steps. In the simulation, advancing the TM by a single step is simulated by an SLP$[\leftarrow, \rightarrow, Bool]$.

Proof. We want to make the simulation of Lemma 8 into a tape-bounded one. In the new simulation, the value of *boundary* will be $1 \vee (B \leftarrow (c-1))$.

To the boundary condition discussed in Lemma 8 we now add a second, triggered by $head \wedge IsLeftMotion \wedge (boundary \rightarrow c)$, which will cause the state machine to transition to another new halting state, namely one signifying that the simulation was stopped due to the reading head exceeding its tape allocation. □

An explanation is due as to the question of why the second 1 bit of the boundary is at position $s+c-1$, and not at position s. The reason is that *boundary* does not signify the end of the tape, but rather the end of all bits used in the simulation process. When the reading head is at its left-most position (position $s-1$) the state of the TM is written in parameter *state* between its $s-1$'th and $s+c-2$'th position. Therefore, the $s+c-1$ bit is the first not to be part of the simulation. Some of the bits before the $s+c-1$ bit, such as the last $c-1$ bits of *tape* are not used in the simulation process.

This property is put to use in Lemma 10.

Lemma 10 (Simultaneous simulation)**.** *For any nonnegative k, and any naturals $s_1, \ldots, s_k$, $k+1$ copies of the same TM, working on inputs $inp_1, \ldots, inp_{k+1}$ on tapes of sizes $s_1, \ldots, s_k$ and infinity, respectively, can be simulated by a RAM$_0[\leftarrow, \rightarrow, Bool]$ using only bounded shifts, given that the RAM is given its input in the following format:*

$$B = \sum_{i=0}^{k} 2^{\sum_{j=1}^{i}(s_j+c-1)},$$

$$inp = \sum_{i=0}^{k} \left(inp_{i+1} \leftarrow \left(\sum_{j=1}^{i} (s_j + c - 1) \right) \right),$$

and that the output is described by

$$output = \sum_{i=0}^{k} \left(output_{i+1} \leftarrow \left(\sum_{j=1}^{i} (s_j + c - 1) \right) \right),$$

where $output_1, \ldots, output_{k+1}$ *are the halting states of the* $k+1$ *TMs at program termination. The simulation requires* $O(n)$ *RAM steps, where* n *is the number of steps required by the longest running of the executions. In the simulation, advancing all the TMs together by a single step is simulated by an* $SLP[\leftarrow, \rightarrow, Bool]$.

Proof. We simulate all TMs in parallel. In doing so, it is convenient to simulate an "advance one step" also on the TMs that have already halted. Let us therefore reconsider the term "halting state", in order for us to allow continued execution of the simulation even after a halt. To allow this, transition functions need to be defined also for halting states. We define these as follows: if x is a halting state, transitions from x are all to x, the tape contents are never changed, and the reading head will always move to the right, unless it is already at the end of the tape, in which case it will stop. Clearly, all this can be encapsulated by the Boolean functions already discussed.

We note that if a TM runs for n steps, its reading head is no more than n positions from the right end of the tape. Hence, in $O(n)$ steps, the TM's state, head-position and tape contents will have reached their final values.

The reason for the strange behaviour we require for the halting states is that it allows us to easily query the TM simulator, at the end of the simulation, regarding its halting condition. Let us assume, without loss of generality, that the only halting states are

- $\boldsymbol{state} = \mathbf{1}$, signifying an accepting calculation,
- $\boldsymbol{state} = \mathbf{2}$, signifying a rejecting calculation, and
- $\boldsymbol{state} = \mathbf{3}$, signifying that the calculation was aborted due to exceeded tape length.

Because the ultimate position of the reading head is known to be 0, querying the simulator for the final state is simply a test for equality.

To initialise the simulator we set

$$boundary \Leftarrow B,$$
$$head \Leftarrow B,$$
$$state \Leftarrow 0,$$
$$tape \Leftarrow inp.$$

Because the positions of the "1"s on *boundary* signify a separation of the bit positions into segments that cannot interact, simply using the SLP[←, →, *Bool*] described in Lemma 8 and then re-used in Lemma 9 results in all TMs advancing one step in parallel.

Lastly, we need to test whether all TMs have halted. This can be split into the following conditions. First, we verify that

$$head = B$$

and

$$state\ \mathsf{clr}\ (B \vee (B \leftarrow 1)) = 0.$$

This guarantees that all TMs are either in one of the three halting states or in the initial state, 0. To verify that none are in zero, we simply check that the set of TMs that have halted in one of the halting conditions covers all TMs. This is done by

$$(state \vee (state \rightarrow 1)) \wedge B = B.$$

When all of the above conditions are satisfied, execution of the RAM terminates, and the output can be read off *state*. □

Two side-notes should be made regarding the lemmas so far. First, note that in each of Lemmas 8, 9 and 10, Lemma 1 can be applied in order to omit the operation "→" from the operation set of the RAM.

Second, we note regarding Lemma 10 that even though parallelisation over $k+1$ machines is possible, it is not necessary. By zeroing-out all bits of *head* in any given segment, the TM

related to that segment ceases to advance. Indeed, that would have been a different method to approach the question of how to simulate the execution of machines that have already halted. (Although, taking that approach would have required different methods to check whether the simulated machine halted and what its halting state is.)

Before proceeding further, we introduce two definitions.

Definition 7 (Instantaneous Description)**.** Let $\mathcal{T}$ be a TM working on a bounded tape of size s and having a state space that can be described (including up to 3 additional halting states, as in Lemma 10) by c bits.

An *instantaneous description* of $\mathcal{T}$ at any point in its execution is the value of $tape + 2^{s+c-1}state + 2^{2(s+c-1)}head$ at that point in its execution, where *tape*, *state* and *head* are as in Lemma 9.

Definition 8 (Tableau)**.** Let $\mathcal{T}$, s and c be as in Definition 7. A *tableau* is a vector (in the sense of Definition 1) of width $3(s+c-1)$, whose i'th element is the instantaneous description of $\mathcal{T}$ after i execution steps.

The proofs below will also use the notation $O^n_{a,m}$ defined in Equation (3.3).

Lemma 11 (Fast nondeterministic simulation)**.** *There exists a constant, K, such that for every TM, $\mathcal{T}$, there exists a $RAM_0[\leftarrow, Bool]$, $\mathcal{R}$, that takes $K+1$ inputs, such that for every number inp, $\mathcal{T}$ accepts inp if and only if there exist witness numbers $w_1, \ldots, w_K$, for which $\mathcal{R}$ accepts on*

$$(inp, w_1, \ldots, w_K),$$

and $\mathcal{R}$ executes in $O(1)$ time on every set of inputs.

Proof. The TM $\mathcal{T}$ accepts if and only if there is an accepting computation path beginning at the initial instantaneous description at the start of execution with *inp* as its input. (The terminology "accepting computation path" is more commonly applied to nondeterministic computation. In deterministic computation, there is exactly one path. The question is only whether it is accepting or not.) If there exists an accepting computation path, it can be described by a tableau. (If it does not exist, no corresponding tableau exists.) Let (m, V, n)

be this tableau. We will choose witness integers so that the RAM is able to verify the existence and correctness of the tableau.

We use $K = 5$. The auxiliary inputs to the RAM will be the following five witness integers.

$$w_1 = m/3,$$
$$w_2 = V,$$
$$w_3 = (n-1)m,$$
$$w_4 = O^n_{2^{m/3}-1,m},$$
$$w_5 = O^n_{1,m}.$$

The triplet (w_1, w_2, w_3) defines the tableau, by a reversible transformation. The first step of the verification process is not to test the validity of this tableau, but rather to ascertain that w_4 and w_5 correspond properly to this tableau definition.

Let $a = 2^{m/3} - 1$, and note that it can be computed by $\neg(1 \leftarrow w_1)$. Ascertaining the validity of w_4 is done by verifying that

$$w_4 - a = (w_4 - (a \leftarrow w_3)) \leftarrow w_1 \leftarrow w_1 \leftarrow w_1.$$

This follows directly from the definition of $O^n_{2^{m/3}-1,m}$.

We do not actually have subtraction as part of our operation set, but $A - B$ can be simulated in this case by $A \mathsf{clr} B$ after having verified that $A \wedge B = B$.

Validating w_5 is done following exactly the same principles, only replacing a for 1 in the computation.

At first glance, it may seem that w_3 also needs to be verified in order to ascertain that it is a multiple of $3w_1$, but in fact this has already been covered by the previous checks: if $2^x - 1$ is a multiple of $2^y - 1$ then x is a multiple of y. That being the case, the fact that we have already established

$$w_5 = \frac{2^{w_3+3w_1} - 1}{2^{3w_1} - 1}$$

directly implies that $w_3 + 3w_1$ is a multiple of $3w_1$, and therefore also w_3 is a multiple of $3w_1$.

With these verified, the vector contents of the tableau, V, given in w_2 can be decomposed into its constituents, as demonstrated below. The equations below use right-shifting as well as left-shifting. This is justified by Lemma 1, which can be applied here, because all shifts are by w_1.

$$\begin{aligned} \mathit{tape} &= V \wedge w_4, \\ \mathit{state} &= (V \wedge (w_4 \leftarrow w_1)) \rightarrow w_1, \\ \mathit{head} &= (V \wedge (w_4 \leftarrow w_1 \leftarrow w_1)) \rightarrow w_1 \rightarrow w_1. \end{aligned}$$

Furthermore, a corresponding *boundary* parameter can be devised by

$$\mathit{boundary} = w_5 \vee (w_5 \leftarrow w_1).$$

On the assumption that the tableau is legitimate, if we were to use these new values as inputs for the RAM described in Lemma 10 (which can be used here, because, again, the right-shift operations can be circumvented, as per Lemma 1), then what these inputs describe is n copies of the same TM, each performing execution on the same input, but at a different stages in the execution. The first TM is at execution start, the next one is after one execution step, etc..

We can now use the RAM described in Lemma 10 to simulate, in O(1) total time, one execution step of each of these n TM steps. Let us denote the output variables after a single-step operation (tape', state', head'). The legitimacy of the tableau can now be verified. A true tableau will satisfy two conditions:

1. The state of the machine described in element 0 of the original tableau is the correct initial state of the TM to be simulated.

2. For each i, the instantaneous description of the machine described in element i of the tableau, after advancing by a single step, equals the original instantaneous description in element $i + 1$ of the tableau.

Verifying these conditions for *tape*, for example, is done by checking that

$$(inp \wedge a) \vee ((tape' \ \mathsf{clr} \ (a \leftarrow w_3)) \leftarrow w_1 \leftarrow w_1 \leftarrow w_1) = tape.$$

The idea is that $tape' \leftarrow m$ should equal *tape* in all but the first and last elements. These are handled separately. The same method works for verifying the validity of *state* and *head*.

Once the tableau is known to be correct, we finish by ascertaining that its final state is an accepting state:

$$(head \wedge (a \leftarrow w_3 \leftarrow w_1 \leftarrow w_1) = 1 \leftarrow w_3 \leftarrow w_1 \leftarrow w_1$$

and

$$(state \wedge (a \leftarrow w_3 \leftarrow w_1 \leftarrow w_1) = 1 \leftarrow w_3 \leftarrow w_1 \leftarrow w_1$$

□

Though not required for the main proofs that these lemmas have all been building up towards, it is still illuminating to see that Lemma 11 implies the following much stronger claim.

Theorem 9. *All recursively enumerable (r.e.) sets can be recognised in $O(1)$ time by an NRAM[op], where op is a superset of any of the following: $\{\leftarrow, Bool\}$, $\{\rightarrow, Bool\}$, $\{/, Bool\}$ and $\{\times, Bool\}$.*

We note that a lemma corresponding to Theorem 9 appears also as Lemma 1 of Simon (1981), but our results are somewhat sharper in that the instruction sets used by Simon are slightly larger and the operations stronger.

We prove separately for each of the instruction sets in the theorem.

Proof for op $\supseteq \{\leftarrow, Bool\}$. For this part of the proof, the missing piece from Lemma 11 is the ability to extract all 5 required integers, $w_1, \ldots, w_5$, from a single, Oracle-provided integer, α, in O(1) time.

A critical observation is that there are some degrees of freedom in the choice of witnesses. For example, w_1 merely needs to be large enough. It signifies the length of the tape available to the TM, so choosing a larger value does not invalidate the simulation. As such, we can restrict w_1 to be a power of two at no cost.

Next, we note that w_3 is not actually needed from the Oracle. The number w_3 provides the length, n, of the tableau, but choosing an overly long tableau is not a problem. All that needs to be ascertained is that the chosen length is not too small.

Consider how many simulation steps are needed to decide how a TM's execution terminates. A TM on a bounded-length tape is essentially a finite state machine (FSM). It only has a constant number of instantaneous descriptions that it can be in. In our case, each instantaneous description is stored by m bits. Hence, the size of the state space is trivially bounded by $S = 2^m$. There is no reason to simulate a deterministic FSM to more steps than the size of its state space, because if it has not terminated after S steps this indicates that it has revisited a state more than once, and is therefore in an infinite loop.

Assuming that w_1 is a power of two, multiplication by 3 can be performed by

$$m \Leftarrow w_1 \vee (w_1 \leftarrow 1).$$

Using $w_3 \Leftarrow m \leftarrow m$ is then equivalent to choosing $S+1$ for n, which is more than enough.

This leaves us with the necessity to extract w_1, w_2, w_4 and w_5 from α.

We do so by first considering $\mathsf{inc}(\alpha)$ clr α. If α's u lowest bits are ones, but the $u+1$ bit is 0, this operation yields the number 2^u. We will take this to be w_1, noting that by construction it is a power of two, as desired.

The next observation to make is that w_2, w_4 and w_5 all have at most $W = w_3 + m$ bits. As such, consider α as the contents of a vector of width W and length 4. The vector element in the 0'th position, of which $u + 1$ bits were already specified, we ignore. The remaining elements we take to be w_2, w_4 and w_5, respectively.

Let $M = 2^W$. This value can be calculated by

$$M \Leftarrow 1 \leftarrow w_3 \leftarrow w_1 \leftarrow w_1 \leftarrow w_1.$$

Given a right-shift operator, the three witness values would have been extractable by

$$w_2 = (\alpha \to w_3 \to w_1 \to w_1 \to w_1) \wedge \neg M,$$

and similarly for w_4 and w_5, from which point the rest of the construction could follow that of Lemma 11.

Because right-shifting is, in fact, not assumed to be part of *op*, we note, instead, that all shifts in both the construction above and that of Lemma 11 are by amounts that require no right-shifting to calculate. (Aside from shifts by a constant, they are by w_1, w_3 and m, which we show here how to calculate explicitly with no use of right-shifting.) That being the case, we can apply Lemma 1 to show that the right-shift operator is not necessary. □

Proof for op $\supseteq \{\to, Bool\}$. When right-shifting is the only shift available, we employ a slightly different tactic. Given a right-shift operator, unpacking in O(1) time K integers, $a_1, \ldots, a_K$ from a given integer, α, is fairly straightforward. We do this by storing in α the contents of the vector $[u-1, a_1, \ldots, a_K]$, where u is the width of the vector and is a power of two and greater than any of the a_i. (Regardless of what values we wish to store in the a_i, it is always possible to choose an appropriate width, u. We note that unlike in the first part of the proof, here u does not serve any role other than being the width of this vector. It does not participate in the actual simulation of the TM.)

Finding u is done by

$$u \Leftarrow \mathsf{inc}(\alpha)\ \mathsf{clr}\ \alpha$$

as in the first part of the proof. Extracting the elements is done by

$$a_i = (\alpha \overbrace{\to u \cdots, \to u}^{i \text{ times}}) \wedge \neg u.$$

In order to proceed from K extracted integers to a proof regarding a TM, we use the same algorithm as that of Lemma 11 (not using any of the short-cuts introduced for *op* $\supseteq$ $\{\leftarrow, Bool\}$). However, we switch every left shift for a right shift by the following technique.

First, note that Lemma 11 guarantees a procedure that terminates in O(1) time. As such, it can only use a bounded number of left-shift operations.

We can therefore transform each $X \Leftarrow Y \leftarrow Z$ operation in the original proof to a comparison step: $X = Y \leftarrow Z$. If all of X, Y, Z and $W = 2^Z - 1$ are given as part of α, this can be performed simply by the following verification steps.

First, we verify that $W + 1$ is a power of two by $\mathsf{inc}(W) \wedge W = 0$. Next, we verify that it is the correct power of two by $\mathsf{inc}(W) \rightarrow Z = 1$. Lastly, we verify the actual operation by $X \rightarrow Z = Y$ coupled with $X \wedge W = 0$. □

Proof for $op \supseteq \{/, Bool\}$. As before, we work from Lemma 11 and only need to complete two details.

1. How does one unpack K general integers from a single α?

2. How does one simulate left-shifting using op?

Extracting multiple integers from α is done in this case by storing in α the contents of the vector $[2^u - 1, 0, a_1, \ldots, a_K]$, where u is the width of the vector. Extracting $2^u - 1$ is done as before, and the rest of the elements can be retrieved by repeatedly dividing by 2^u, and then ultimately taking the bitwise intersection with $2^u - 1$.

Next, we wish to simulate $X \Leftarrow Y \leftarrow Z$, which, as before, is done by verifying $X = Y \leftarrow Z$. Specifically, note that Z is always either a constant or one of w_1 and w_3, and that this is the only context in which either w_1 or w_3 are ever used. Instead of storing these values directly, we therefore store $Z_1 = 2^{w_1} - 1$ and $Z_3 = 2^{w_3} - 1$.

Ascertaining that Z_1 and Z_3 are valid can be performed by $\mathsf{inc}(Z_i) \wedge Z_i = 0$. Following the transformation from w_i to Z_i, the left shifts become multiplications: $X = Y \times \mathsf{inc}(Z_i)$. We verify them by $X \wedge Z_i = 0$ and $Y = X / \mathsf{inc}(Z_i)$, where the first comparison ascertains that the second comparison involves an exact division (utilising the fact that $\mathsf{inc}(Z_i)$ is a power of two). □

Proof for $op \supseteq \{\times, Bool\}$. We use the same α as in the proof for $op \supseteq \{/, Bool\}$, noting that $X = Y \leftarrow w_i$ can be verified now simply by $X = Y \times Z_i$. This leaves us with the need to

extract multiple integers, $a_1, \ldots, a_K$ from a single α, which is done by

$$a_i \times 2^{u(i+1)} \Leftarrow ((2^u - 1)\overbrace{\times 2^u \times \cdots \times 2^u}^{i+1 \text{ times}}) \wedge \alpha.$$

Notably, the a_i themselves are not extracted, but rather a shifted version of them. The reason that this partial extraction suffices is because in all operations, $X \times Z$ will be offset by a known power of 2^u compared to the offset with which we calculate Y, and simply multiplying one side by 2^u the appropriate (and bounded) number of times will serve to prove equality. □

We now turn to prove the main theorem.

Proof of Theorem 8. The statement of the theorem involves equality among four Turing machine related complexities, one RAM complexity and one RAM_0 complexity. The equivalence between the deterministic time and deterministic space TM complexities stems from the well-known argument that

$$\mathrm{TIME}(n) \subseteq \mathrm{SPACE}(n) \subseteq \mathrm{TIME}(\mathrm{EXP}(n)).$$

In n execution steps, the reading head cannot reach more than n elements into the tape, proving $\mathrm{TIME}(n) \subseteq \mathrm{SPACE}(n)$, whereas an execution bounded by n tape elements cannot proceed more than $\mathrm{EXP}(n)$ execution steps without retracing its steps, proving $\mathrm{SPACE}(n) \subseteq \mathrm{TIME}(\mathrm{EXP}(n))$. In expansion limit terms, $2^{\mathrm{EL}_{op}(n)}$ is greater than $\mathrm{EL}_{op}(n)$ but is bounded from above by $\mathrm{EL}_{op}(n + \mathrm{O}(1))$. This argument shows equivalence between $\mathrm{EL}_{op}(n + \mathrm{O}(1))$ time-bounded TMs and $\mathrm{EL}_{op}(n + \mathrm{O}(1))$ space-bounded TMs, which is a stricter condition than the one actually needed for the theorem.

The addition of nondeterminism to a space-bounded TM requires only a limited amount of extra space to simulate on a deterministic TM, as is proved by Savitch's Theorem (Savitch, 1970), so here, too, an $\mathrm{EL}_{op}(n + \mathrm{O}(1))$ is all that is needed. Simulating an n time nondeterministic TM by a deterministic TM requires at most n nondeterministic bits, and can hence be simulated by $\mathrm{EXP}(n)$ runs of the deterministic algorithm, enumerating over the witness string. Exponentiation is, again, within $\mathrm{EL}_{op}(n + \mathrm{O}(1))$.

The equivalences among TM complexities presented are therefore trivial.

The equivalence between RAM and RAM_0 complexities is given by Theorem 3, even without requiring the operation set to be constructable. We remark that for computing RAMs, the transformation from RAM-style input to RAM_0-style input is trivial. An input *inp* can be represented as a vector of length 1 and width *inp* whose contents are *inp*. This transformation costs $\text{O}(1)$ time.

We will, therefore, only need to prove the following equality.

$$\text{O}(f(n))\text{-RAM}_0[op] = \text{EL}_{op}(\text{O}(f(n)))\text{-SPACE-TM}.$$

This, in turn, can be thought of as two statements. We begin by demonstrating the simpler

$$\text{O}(f(n))\text{-RAM}_0[op] \subseteq \text{EL}_{op}(\text{O}(f(n)))\text{-SPACE-TM}, \tag{4.1}$$

then continue to our main argument,

$$\text{O}(f(n))\text{-RAM}_0[op] \supseteq \text{EL}_{op}(\text{O}(f(n)))\text{-SPACE-TM}. \tag{4.2}$$

To prove Equation (4.1) we show that an $\text{EL}_{op}(\text{O}(f(n))$ space-bounded TM can simulate an $\text{O}(f(n))\text{-RAM}_0$. We allow this simulation to use a larger set of tape symbols than the original $\{0, 1\}$, knowing that this costs at most a linear increase in the amount of space required (Stearns et al., 1965), which cannot violate the $\text{EL}_{op}(\text{O}(f(n))$ bound.

Consider the following memory layout which may be used by the TM. Each of the (finite) number of registers of the RAM_0 is allocated a new tape-alphabet character. We refer to these as "control" characters. During the TM's execution, each control character will appear on the tape exactly once. All characters to the right of it and to the left of the next control character are deemed to be the contents of the relevant register. Additionally, a further control character, appearing last on the tape, marks the start of a "scratchpad". The TM is initialised by moving the input one element to the right in order to make room for the $R[0]$

control character, then writing the $R[0]$ control character before the input and the rest of the control characters after (signifying registers whose contents are zero).

At each execution step, the operands are copied to the scratchpad, the operation required by the RAM_0 is performed, the tape elements are moved enough to allocate enough space on the target register, and then the scratchpad result is copied to the target register and the scratchpad is cleared.

By the definition of EL, each register's contents requires at most $\text{EL}(f(n))$ elements to store. To store all registers we require $\text{O}(\text{EL}(f(n)))$. By definition of RAM-constructability, the scratchpad requires no more than $\text{EL}(\text{O}(f(n)))$ elements of memory.[8] Together, the amount of tape required is bounded by $\text{EL}(\text{O}(f(n)))$, as required by the theorem.

We now turn to the main problem of proving Equation (4.2). We do this by working from Lemma 11, noting that if there exists a tableau computation (m, V, n), then there also exists one with (m', V, n'), where m' can be any number at least as large as m, and n' is determined from m' so as to be at least as large as the number of possible instantaneous descriptions for the TM. (This point, regarding the maximum necessary length for the simulation, was made in the proof of Theorem 9: if the TM requires more steps, it will necessarily have retraced its steps and have entered an infinite loop.)

Given a chosen value for m', the computed value of n' allows us to determine the number of bits in V. It is $T = n'm'$. V is, therefore, one of finitely many possibilities. If we were to enumerate over all 2^T of these possibilities, we would be able to answer whether a tableau for an accepting computation exists for a particular value of m'.

Consider now several tableau candidates, $\mathit{tableau}_0, \ldots, \mathit{tableau}_{k-1}$, each of which is a variable in the range $0 \leq v < 2^T$. We wish to check simultaneously which of these (if any) is a correct tableau of bit-length T, describing the execution of a TM running on a tape of length s on input *inp*. To do this, we describe a new variable, $V = \sum_{i=0}^{k-1} 2^{Ti}\mathit{tableau}_i$.

Consider now Lemma 10. It allows the verification of a tableau by use of only bitwise operations and shifts by values that are functions only of s. Because all candidates share the

[8] This follows from $M_{op}(a + b, \mathit{inp}) = \max_x M_{op}(b, x))$, where the maximum is taken over all x that are computable from *inp* in a calculation steps.

same s, each of these operations is executed on all candidates simultaneously, by executing them on V. There are only two places where this strategy fails.

First, the verification requires the use of several constants, such as 1 and *inp*. In order to apply these simultaneously to all simulations, we need to replace the original constants by $O^k_{1,T}$ and $O^k_{inp,T}$, respectively.

Second, the final step in verifying whether a tableau candidate is correct involves several comparisons of the type "$a = b$". These must now be executed in parallel. This can be performed by the operation EQ, described in Appendix A.4. The operation EQ takes two vectors and outputs a new vector of the same length and width as its two operands, whose elements are 1 in positions where $a_i = b_i$ and 0 otherwise. A correct tableau is one that passes all equality checks outlined in the proof of Lemma 10. To simulate this in parallel for all candidates, we perform the necessary EQ per equality test, then take the bitwise AND over all results. The resulting integer, *res*, has a 1 in its Ti'th position if and only if the $tableau_i$ candidate is correct.

For the purposes of the present proof, we are not interested in finding which of the tableau candidates is correct, but rather whether *any* of them are correct. This can be checked by a simple "$res = 0$".

(Readers may wish to verify that the proof of Lemma 10 can be readily adapted to work on a tableau candidate that has been shifted left by an arbitrary amount; the tools for doing so are basically those of Lemma 1. Furthermore, the verification process is not disrupted by having any additional non-zero bits to the left or right of the tableau bits. The verification process does not use or disturb any bits in the integer, other than the T being checked.)

The above argument, showing that the condition "the tableau is correct" can be verified in constant time can be applied equally well with additional checks such as "the tableau is correct and the halting state is 1". Thus, if we were to check all 2^T tableau candidates, the program does not only work as a verifier, but also as a complete simulator: it takes an input, and can return (in constant time) what the final state of the TM finite control is. In our case, the interesting final states are 1, 2 and 3. The simulator should return which of these three the TM halted on, or return that none of the above occurred.

The only remaining question is how to create all $k = 2^T$ possible tableaux in a single integer, for them to be verified simultaneously. One way to do this is as follows. The function described produces the contents of a vector of width T and length 2^T, whose elements are all unique.

$$U^T = \sum_{i=1}^{2^T-1} O_{1,T}^i = \sum_{i=1}^{2^T-1} \frac{2^{Ti}-1}{2^T-1} = \frac{\frac{2^{T\times 2^T}-1}{2^T-1} - 2^T}{2^T-1} \tag{4.3}$$
$$= ((\neg(1 \leftarrow (T \leftarrow T))/\neg(1 \leftarrow T))\ \mathsf{clr}\ (1 \leftarrow T))/\neg(1 \leftarrow T)$$

It is computed solely by use of left-shifting, exact division and Boolean operations.

We see, therefore, that all tableaux can be created and verified for correctness in constant time. Let us refer to the RAM program that verifies simultaneously and in constant time all possible tableaux pertaining to the execution of a TM on a tape of size s as $simulate(s, inp)$.

By the assumption of constructability, there also exists a RAM program, $el(n)$ that calculates a value no smaller than $\mathrm{EL}_{op}(n)$ in $\mathrm{O}(n)$ time. In order to complete the proof of the theorem, we therefore present Algorithm 7 that enables a RAM working in $\mathrm{O}(n)$ time to simulate any TM working in $\mathrm{EL}(\mathrm{O}(n))$ time.

Algorithm 7 A complete TM simulator

```
1: n ⇐ 1
2: loop
3:     s ⇐ el(n)
4:     simulate(s, inp)
5:     if halting state is 1 then return Accept
6:     end if
7:     n ⇐ n ← 1
8: end loop
```

We remark that Algorithm 7 accepts if the original TM accepts, and continues indefinitely if the original TM does not, but this is merely because we check only for a halting state of 1. It is also possible to terminate the run and reject if the halting state is 2 (signifying that the TM halted and rejected) or if the halting state is not in 1-3 (signifying that the TM returned to a previous instantaneous description, and has therefore entered an infinite loop). However, if the TM continues indefinitely while consuming unbounded amounts of tape, the simulation

cannot detect this, and will continue forever. This is, of course, inherent, because had the simulator been able to determine in finite time for every TM whether it halts or not, this would have contradicted Turing's halting theorem (Turing, 1936). □

4.3.4 Adding arbitrary numbers

Despite the fact that the main ideas in the proof of Theorem 8 can all be traced back to Simon (1981), unlike Simon's original construction, our Theorem 8 strongly suggests Theorem 5: any recursively enumerable (r.e.) set can be recognised by an ARAM$[+, /, \leftarrow, \mathit{Bool}]$ in O(1) time. The reason for this, informally, is that once arbitrary numbers are admitted, the numbers that can be present in the RAM after even one execution step become arbitrarily large, making EL effectively infinite. If the power of the RAM is only bound by its ability to generate larger and larger numbers, as is implied by Theorem 8, adding ALNs should make the RAM arbitrarily powerful.

We now present this as a formal argument.

Proof of Theorem 5. If S is an r.e. set, then there is a TM that recognises it, indicating that $\mathit{inp} \in S$ if and only if there is an accepting computation of the TM with *inp* as the TM's input. This accepting computation by definition requires only a finite number of execution steps of the TM, and therefore only a finite number, s_{req}, of tape cells. Consider Algorithm 8, using the RAM-implementable function *simulate* discussed at Algorithm 7.

Algorithm 8 Simulating a TM in constant time

```
1: s ⇐ ALN
2: simulate(s, inp)
3: if halting state is 1 then
4:     return Accept
5: else
6:     return Reject
7: end if
```

By the arguments of the proof for Theorem 8, a sufficient condition for *simulate* to return the proper result is $s \geq s_{req}$. This is clearly satisfied by making s "large enough", which is what is guaranteed by "$s \Leftarrow$ ALN". Formally, almost all choices for s return the correct

result, as the number of s values that return an incorrect result is finite and bounded by s_{req}. □

We have shown, therefore, that while arbitrary numbers have no effect on computational power without division, with division they provide the strongest possible computational model: Turing completeness in O(1) computational resources.

4.4 Arbitrary numbers and the arithmetical hierarchy

In light of Theorem 5, which shows that ARAMs with division can perform in O(1)-time everything that a Turing machine working without time or space limitations can do, it is natural to ask whether these ARAMs can do even more, and if so, how to quantify their power.

One model which provides us with terminology for assessing computational power beyond what is available to Turing machines is the arithmetical hierarchy. A full exposition of the arithmetical hierarchy is available in Rogers (1987) and Moschovakis (2009). We introduce it here briefly.

The set Σ_0^0, also denoted Π_0^0 is the set of formulae logically equivalent to formulae including only bounded quantifiers. For any $i > 0$, the set Σ_i^0 is the set of formulae logically equivalent to $\exists a_1 \exists a_2 \cdots \exists a_k \phi$, where $\phi \in \Pi_{i-1}^0$, and the set Π_i^0 is the set of formulae logically equivalent to $\forall a_1 \forall a_2 \cdots \forall a_k \phi$, where $\phi \in \Sigma_{i-1}^0$. For any $i \geq 0$, the set Δ_i^0 is defined by $\Delta_i^0 = \Sigma_i^0 \cap \Pi_i^0$.

In our context, we take formulae as representing the languages composed of the values for which the formulae hold. The recursively enumerable sets are exactly Σ_1^0, whereas the decidable sets (those which can be decided by a TM that is guaranteed to eventually halt) are exactly Δ_1^0. The set Π_1^0 is exactly the co-r.e. languages.

Theorem 10. *Denote by $ARAM[+, /, \leftarrow, Bool]$ the set of languages that can be recognised by such an ARAM, regardless of time constraints. This set is on the second level of the arithmetical hierarchy, in the following sense:*

$$\Pi_1^0 \subset ARAM[+, /, \leftarrow, Bool] \subseteq \Pi_2^0,$$

Proof. Theorem 5 already established $\Sigma_1^0 \subseteq \text{ARAM}[+, /, \leftarrow, \mathit{Bool}]$. Note, however, that because the ARAM programs used in the proof of Theorem 5 are all guaranteed to terminate, it is possible to invert their ultimate result. This observation establishes $\Pi_1^0 \subseteq \text{ARAM}[+, /, \leftarrow, \mathit{Bool}]$, and, as a corollary, $\Sigma_1^0 \cup \Pi_1^0 \subseteq \text{ARAM}[+, /, \leftarrow, \mathit{Bool}]$.

Clearly, this union is larger than either base set. (The halting problem and its complement provide examples of languages that are in Σ_1^0 and, respectively Π_1^0, but not in Π_1^0 and, respectively, Σ_1^0.)

This observation is sufficient to establish the proper inclusion in the statement of the theorem.

We now turn to proving $\text{ARAM}[+, /, \leftarrow, \mathit{Bool}] \subseteq \Pi_2^0$.

By definition, an ARAM is the result of a RAM working with an extra input n, such that n is greater than some threshold value, N. The RAM itself is computationally equivalent to a TM, so is known to compute a formula $\phi \in \Sigma_1^0$. This indicates that ϕ can be written as

$$\phi = \exists a \chi, \tag{4.4}$$

where χ is a formula with only bounded quantifiers.

In logic formulation, an ARAM computes a formula ψ that satisfies two conditions

$$\psi(\mathit{inp}) = \exists N \forall n (n \leq N) \cup \phi(n, \mathit{inp}), \tag{4.5}$$

and

$$\psi(\mathit{inp}) = \forall N \exists n (n > N) \cap \phi(n, \mathit{inp}). \tag{4.6}$$

The first of these conditions stipulates that if $\psi(\mathit{inp})$ is true then the ARAM must accept *inp* for almost all values of the ALN extra input.[9] On the other hand, if $\psi(\mathit{inp})$ is false, then there must be an infinite number of ALN values that will make the ARAM reject *inp*.

[9] That is to say, all except a finite number.

The second condition stipulates that if $\psi(inp)$ is false then the ARAM must reject *inp* for almost all values of the ALN extra input. On the other hand, if $\psi(inp)$ is true, then there must be an infinite number of ALN values that will make the ARAM accept *inp*.

Together, these two conditions describe the requirement of an ARAM, namely that if the extra input is large enough, then the ARAM computes ψ.[10]

To prove that any function computed by an ARAM is in Π_2^0, we need only Equation (4.6). Substituting Equation (4.4) in, we get

$$\psi(inp) = \forall N \exists n \exists a (n > N) \cap \chi(a, n, inp),$$

where χ contains only bounded quantifiers. This places the formula in Π_2^0 by definition. □

Corollary 10.1. *For an ARAM program $\mathcal{A} \in ARAM[+, /, \leftarrow, Bool]$ that is guaranteed to halt,*

$$\mathcal{A} \in \Delta_2^0.$$

In particular,

$$P\text{-}ARAM[+, /, \leftarrow, Bool] \subseteq \Delta_2^0.$$

Proof. If an ARAM is known to terminate, the same is true for its underlying RAM (calculating what was previously denoted as ϕ). This RAM, therefore, calculates a decidable problem, so $\phi \in \Delta_1^0$. In particular, ϕ can be described as

$$\phi = \forall a \bar{\chi}. \tag{4.7}$$

Substituting this into Equation 4.5, we get

$$\psi(inp) = \exists N \forall n \forall a (n \leq N) \cup \bar{\chi}(a, n, inp).$$

[10] By definition, an ARAM computes $\psi(inp)$ if its underlying RAM computes $\phi(n, inp)$ and the two coincide for all but a finite number of n values. A succinct way of writing this is

$$\overset{\infty}{\forall} n \, \phi(n, inp) = \psi(inp).$$

This places ψ in Σ_2^0. Together with Theorem 10, we have $\psi \in \Delta_2^0$. $\square$

Chapter 5

Arbitrary number sequences

5.1 Introduction

Before departing completely from arbitrary numbers, we give in this chapter a short vignette into how arbitrary numbers interact with each other. For this, we introduce a new computational model, which we refer to as the ASRAM, for "arbitrary sequence RAM".

Just as the ARAM model provides the RAM with the power boost of a single arbitrary number, the ASRAM allows us to study the case where arbitrary numbers are available in unlimited amounts. Specifically, in the ASRAM, *ALN* is provided as a pseudofunction: the program may call it at any step to generate a new arbitrary number.

This chapter quantifies the power of the ASRAM and, specifically, discusses the added contribution in power of the availability of multiple arbitrary numbers, as compared to a single ALN.

We note that the ASRAM model further provides us with a platform on which to investigate the intermediate scenario, in which only a predefined number (e.g. 2) of ALNs are available to the algorithm. This is simply done by limiting the number of times the *ALN* pseudofunction can be executed.

5.2 Formal definition of the model

Definition 9 (ASRAM)**.** An *ASRAM* is a computational model that provides the same functionality as the RAM, but also allows calls to a pseudofunction, "*ALN*()", that returns integers. We refer to the integer returned on the k'th call to this function as A_k.

A language $\mathcal{L}$ is said to be accepted by the ASRAM if for every input to the ASRAM there exists a subset, $\mathbf{S}$, of the set of (infinite) integer sequences, that satisfies the following conditions.

1. $\mathbf{S}$ is nonempty.

2. If the $\{A_k\}$ sequence is taken from $\mathbf{S}$, the underlying RAM computation accepts if and only if the input is in $\mathcal{L}$.

3. For any i and any sequence $\{A_k\} \in \mathbf{S}$ there exists a sequence $\{B_k\} \in \mathbf{S}$ such that $B_i = A_i + 1$, and if $j < i$, then $B_j = A_j$.

This definition of an ASRAM reflects a situation where every application of the function "*ALN*()" results in a number that is arbitrary and large enough with respect to everything that occurred in earlier steps of the ASRAM's execution. This generalises the notion of an ARAM, where the ALN is taken to be arbitrary and large enough with respect to the RAM's input.

The original ARAM is simply an ASRAM that is restricted to use only a single call to *ALN*(), and whose output therefore depends only on A_1. The definition, as it pertains to ARAMs, is that if S contains a sequence beginning with some A_1, indicating that A_1 is "large enough", then $B_1 = A_1 + 1$ is also "large enough", and a sequence beginning with this value should also exist in S. By induction, any number larger than A_1 is similarly "large enough", and a sequence beginning with it exists in S, and is therefore admissible as a sequence returned by the *ALN*() pseudofunction.

5.3 Arithmetic complexity

At face value, one may believe that multiple arbitrary numbers are no more powerful than a single arbitrary number. However, this is not so.

In this section we detour slightly from the main theme of this work, which focuses on RAMs that use shift left and mixed field operations as their basic operation set, to show that the extra power of arbitrary sequences is present already in the more conservative arithmetic complexity model, this being the computational model in which the basic operations used are the four arithmetic functions, $\{+, \dot{-}, \times, \div\}$.

We stress that the results presented rely heavily on the non-arithmetic nature of these so-called "arithmetic" operations. In the literature (Lürwer-Brüggemeier and Ziegler, 2008), the operation "$\div$" is, in fact, sometimes referred to as non-arithmetic division.

Formally stated, what we prove is the following theorem.

Theorem 11. *P-ARAM*$[+, \dot{-}, \times, \div] \subset$ *P-ASRAM*$[+, \dot{-}, \times, \div]$.

Proof. Consider Algorithm 9.

Algorithm 9 An arithmetic ASRAM calculating $2^{2^{2^x}}$ in $\mathrm{O}(x)$ time

1: **for** $i \in 1, \ldots, x$ **do**
2: $\quad A_i \Leftarrow ALN()$
3: **end for**
4: $P_x \Leftarrow A_x \times A_x$
5: **for** $i \in x-1, \ldots, 1$ **do**
6: $\quad temp \Leftarrow P_{i+1} \bmod (A_{i+1} - A_i)$
7: $\quad P_i \Leftarrow P_{i+1} \bmod (A_{i+1} - temp)$
8: **end for**
9: $temp \Leftarrow P_1 \bmod (A_1 - 2)$
10: $rc \Leftarrow P_1 \bmod (A_1 - temp)$
11: **return** rc

This algorithm utilises the fact that for an ALN, A, and a polynomial, P, the calculation $P(A) \bmod (A - x)$ is an algorithm for computing $P(x)$. Utilising this property, the algorithm calculates each P_i so that it equals $A_i^{2^{2^{(x-i)}}}$. After $\mathrm{O}(x)$ steps, it returns the value $2^{2^{2^x}}$.

From Bshouty et al. (1992), we know that an ARAM$[+, \dot{-}, \times, \div]$ can only calculate 2^{2^x} in $\Omega(\sqrt{x})$ time. The fact that Algorithm 9 calculates it in $\Theta(\log x)$ time places calculation of 2^{2^x} inside P-ASRAM but outside P-ARAM, proving that the inclusion is proper. □

Thus, ASRAM is a more powerful model than the ARAM even under arithmetic complexity. Full quantification of this extra power is still an open problem. In the remaining sections of this chapter we consider ASRAMs with the instruction sets discussed in all other chapters of this work, and for the powers of these we are able to provide more complete quantification.

5.4 Without division

Theorem 12. *P-ASRAM*$[+, [\dot{-}], [\times], \leftarrow, [\rightarrow], Bool] = PSPACE$

Proof. The proof is a direct extension of the proof to Theorem 4.

We begin by picking our ALNs, $A_1, A_2, \ldots$ so as to be powers of two: $2^{\omega_1}, 2^{\omega_2}, \ldots$. We then simulate the ASRAM in exactly the same way as was done before. Once again, we see that Lemmas 4–6 continue to hold in the new model. In order to complete the proof, we need a replacement for Lemma 7: a method to compare two po-bits to determine which is the most significant.

In the RAM model, proving Lemma 7 was done by recursively eliminating the last formal variable to be defined, until the comparison becomes a comparison between scalars. In the ARAM model, the same recursion yielded a formal expression of the form $a\omega + b$, which could then be evaluated lexicographically, based on the assumption that ω, being an ALN, is "large enough".

In the new model, the recursion yields a formal expression of the form $\sum_{i=0}^{n} a_i \omega_i$, where ω_0 is taken to be 1. Once again, because each ω_i is, per assumption, large enough compared to all ω_j with $j < i$, a simple lexicographic comparison is enough to determine which of two formal expressions has a larger value.

Thus, Lemma 7 continues to hold, and the entire simulation of the ASRAM can be performed in PSPACE. □

5.5 With division

In this section we prove two main theorems.

Theorem 13. $O(1)$-$ASRAM[+, [\dot{-}], [\times], /, [\div], \leftarrow, [\rightarrow], Bool] = AH$.

Theorem 14. $\omega(1)$-$ASRAM[+, [\dot{-}], [\times], /, [\div], \leftarrow, [\rightarrow], Bool] \supset AH$.

In the statements of Theorems 13 and 14, AH refers to the entire arithmetic hierarchy, $\bigcup_{i=0}^{\infty} \Sigma_i^0 \cup \Pi_i^0$.

Proof of Theorem 13. This proof builds on the proof of Theorem 10. We begin by recalling, as was shown there, that an ASRAM running k steps (and therefore utilising at most k ALNs) is logically equivalent to a formula with k "$\forall N_i \exists n_i (n_i > N_i) \cap$" clauses, one for each ALN used, followed by a formula, ϕ, that is logically equivalent to a RAM, and is, hence, in Σ_1^0. This shows that the ASRAM is in Π_{2k}^0.

We have thus established that the formula computed by the ASRAM is in AH. We will now show that it can be in an arbitrarily high level in the hierarchy.

To do so, we now show that any formula, "$\phi = \exists a_1 \forall a_2 \cdots \exists a_k \chi(inp, a_1, \ldots, a_k)$", with any k quantifiers, can be computed by an ASRAM with k ALNs. The way to do this is to bound every a_i by n_i. The formula ϕ is logically equivalent to "$\exists N_1 \forall n_1 (n_1 \leq N_1) \cup \cdots \exists N_k \forall n_k (n_k \leq N_k) \cup \exists a_1 < n_1 \forall a_2 < n_2 \cdots \exists a_k < n_k \chi(inp, a_1, \ldots, a_k)$". (If an a_i exists to satisfy some condition, its value can be bounded by some N_i, whereas if something is true for every a_i, it will also be true for every bounded a_i.)

This derivation shows that an ASRAM using k ALNs can simulate any formula with k quantifiers. In particular, it can simulate $\Sigma_k^0 \cup \Pi_k^0$, and is, therefore, strictly above the k'th level of the arithmetical hierarchy.

Following the techniques of Theorem 5, we know that the ASRAM that performs this calculation can be implemented as k ALN-generating steps, followed by $O(k)$ additional processing. (The additional processing includes $O(k)$ packing steps, in which all input is packed into a single integer, followed by an O(1)-time simulation step.) Thus, it is an $O(k)$-time ASRAM.

Any formula in AH is on some level of the hierarchy. To simulate it, we fix k to be that level. Thus, the simulating ASRAM runs in O(1)-time. □

Corollary 14.1. *An ASRAM restricted to use only k ALNs is equivalent to a formula on the $\Theta(k)$-th level of the arithmetical hierarchy.*

Proof. This result is a direct corollary of the proof for Theorem 13, which shows that an ASRAM utilising k ALNs can compute any formula in $\Sigma_k^0 \cup \Pi_k^0$, and can be computed by a formula in Π_{2k}^0. □

Consider, now, what happens when the ASRAM (not restricted to any fixed number of ALNs), is allowed to run in $\omega(1)$ time.

Proof of Theorem 14. A well-known example of a function that is not in AH is "TRUTH". This is a function that takes as input a formula, ψ, suitably encoded as an integer, and determines whether this formula is true or not.

The inability to describe TRUTH as a formula is known as Tarski's undefinability theorem (Tarski, 1939). It is a corollary of Gödel's incompleteness theorem (Gödel, 1931) and, in the formulation given above, is a direct result of Post's theorem, stating that the arithmetical hierarchy does not collapse (see Rogers, 1987).

Consider, first, an ASRAM working in $\Theta(k)$ time. As demonstrated in Theorem 13, such an ASRAM can compute, directly, any formula with $\Theta(k)$ quantifiers. Consider a formula, ψ, encoded in the straightforward manner as an integer with k bits. This formula will necessarily have $O(k)$ quantifiers, so a $\Theta(k)$-time ASRAM can be used to compute its truth value. Hence, TRUTH is in $O(k)$-ASRAM$[+, [\dot{-}], [\times], /, [\div], \leftarrow, [\rightarrow], \mathit{Bool}]$.

To extend this result from $\Theta(k)$-time execution to $\omega(1)$-time execution, we note simply that the straightforward formula encoding used above may be, perhaps, the most efficient encoding possible, but is certainly not the only one. For example, it is possible to encode the statement less efficiently by re-encoding the original input number, *inp* as $(2\mathit{inp} + 1) \times 2^T$. An arbitrary choice of T allows decoding of the original *inp*. By choosing a large enough T, the execution time allotted to an $\omega(1)$-time ASRAM becomes $\Theta(k)$, regardless of its actual complexity.

Tarski's undefinability theorem is independent of the exact choice of encoding used to make the input formula, ψ, into a number. The new, tweaked TRUTH function must, therefore, also lie outside of AH.

Thus, any $\omega(1)$-time ASRAM can compute functions that are outside of AH. □

Chapter 6

Random numbers

6.1 Introduction

In Turing machines, the idea of random computation was first introduced by Rabin (1963). The computational model that supported random computation was a Turing machine able to switch in each step to one of two new states depending on the results of a random bit. This gave rise to the complexity class RP composed of those sets that can be recognised by a probabilistic Turing machine in polynomial time, where if the answer is true the machine is required to reach this answer with probability at least half, whereas if the answer is false the machine is required to always reach this answer.

Since Rabin's original paper, randomisation has contributed to the definition of many new complexity classes. These include BPP (Adleman, 1978), PP and ZPP (Gill, 1977), and RL and ZPL (Borodin et al., 1989). These classes differ among themselves in two respects. The first is the respect used to differentiate conventional complexity classes as well, namely their time and tape requirements (taking into account that here these restrictions can be made more varied, because the requirements are now random variables, rather than deterministic functions of the input). The other respect in which the classes differ is in their acceptance criteria. Because a randomised Turing machine's final state is a random variable, such a machine has false-acceptance and false-rejection probabilities. Table 6.1 summarises the possibilities. For convenience of presentation, in the table non-termination is treated as termination in a

non-accepting final state. In practice, the two are, of course, distinct occurrences, and in all algorithms presented they are handled separately.

	$inp \in S$	$inp \notin S$
Final state is accepting	Correct acceptance (t_a)	False acceptance (f_a)
Final state is not accepting	False rejection (f_r)	Correct rejection (t_r)

Table 6.1: The four possibilities for termination in a randomised Turing machine.

The expressions appearing in parentheses in Table 6.1 are acceptance and rejection probabilities. Generally, these probabilities are taken to be the probabilities given a specific input, *inp*. That is, given some *inp*, if $inp \in S$, $t_a = t_a(inp)$ is the probability that it will be accepted and $f_r = f_r(inp)$ is the probability that it will be rejected, with $t_a + f_r = 1$. For $inp \notin S$, we have the corresponding $t_r + f_a = 1$.

The conditions for belonging to randomised complexity classes are generally worded as "$f_a \leq A$ and $f_r \leq B$" (with the "$\leq$" sometimes replaced by "$<$") where A and B are constants. This condition should be taken to mean "For all *inp*, if $inp \in S$, $f_a(inp) \leq A$, and if $inp \notin S$, $f_r(inp) \leq B$". That is, the bounds A and B relate to the worst-case inputs. For our purposes, choosing A (or B) to be 0 is shorthand for the stronger condition that a false acceptance (or rejection) never occurs. (The two conditions coincide for programs that terminate in finite time for every input and every random choice.)

Table 6.2 summarises the criteria used in each random complexity class mentioned. For example, a set, S, belongs to BPP if there exists a Turing machine, $\mathcal{T}$, running in polynomial time, such that $\mathcal{T}$ accepts an input with probability greater than $2/3$ if it belongs to S and rejects with probability greater than $2/3$ if it does not.

Class	f_a	f_r
RP, RL, ZPL	$= 0$	$\leq \frac{1}{2}$
BPP	$\leq \frac{1}{3}$	$\leq \frac{1}{3}$
PP, ZPP	$\leq \frac{1}{2}$	$< \frac{1}{2}$

Table 6.2: Acceptance criteria for various randomised computational classes.

In general, it not known whether randomisation helps Turing machines, in the sense that it is not known whether any of the complexity classes mentioned contains any problem that is not in P.

As mentioned previously, what used to be perhaps the canonical example for the power of randomness is primality testing (Solovay and Strassen, 1977; Rabin, 1980), however, in light of Agrawal et al. (2002) this example, now known to be in P, is no longer a candidate for RP $\setminus$ P, which may or may not be the empty set.

Having said this, several weak inclusion relations are evident directly from the definition of randomised complexity classes. Consider, for example, the classes P, RP, NP and PP, all defined by the acceptance criteria listed previously and the limitation to polynomial run-time. It is known that P $\subseteq$ RP $\subseteq$ NP $\subseteq$ PP, but it is not known regarding any of these whether they are proper inclusions.

The reasoning that leads to these inclusions is fairly straightforward.

- P $\subseteq$ RP because adding the ability to use randomness certainly cannot detract from computational powers. At worst, this ability can simply be ignored.

- RP $\subseteq$ NP because, if a random string of bits has at least 50% chance of leading to a correct accept and 100% chance of leading to a correct reject, then one can pick any one of the at least 50% of the random strings that lead to the correct result, and use it as an Oracle input. The NP success criterion merely requires that at least one Oracle string should lead to the correct result, and that none should lead to a false accept. Both of these are guaranteed by the RP success criterion. Therefore, any TM that recognises a language under the RP criterion would also recognise it under NP.

- Lastly, NP $\subseteq$ PP because one can allot an extra random bit, x, and then accept an input if either $x = 0$ or a randomly chosen string of bits happens to be a working Oracle witness. The probability that such a random string is a valid Oracle witness may be slim, but it is nonzero because at least one Oracle witness is known to exist. Its existence pushes the probability for true acceptance to be greater than $1/2$, whereas false acceptance rate is exactly $1/2$.

The idea of randomised computation has been extended to RAMs by Simon (1981). There, a RAM model is introduced that incorporates a special instruction *RAND*, such that

$$X \Leftarrow RAND(Y)$$

places in X a random integer, uniformly distributed in the interval $[0, Y)$. As in standard operations, Y can be a literal constant, the contents of an explicitly specified register or the contents of a register specified by indirect addressing. The results of any set of calls to *RAND* are assumed to be independent.

We use a slightly more restrictive model, in which Y is required to be 2^k for some known k. When Y is thus restricted, X contains, in its least significant k bits, k random, uniformly distributed and independent bit values. This model is interesting for theoretical study, because it is more directly comparable with the Turing machine model, where randomisation is given in the form of bits. It highlights the basic distinction between randomisation in RAMs and randomisation in TMs, in that the difference is not in the type of randomisation allowed, but rather in the RAM's ability to digest simultaneously large quantities of input. See Section 6.8 for a discussion of the equivalence of the restricted model and the unrestricted one.

Simon's results relate to the model of the RAM equipped with the *RAND* instruction, running an acceptance criterion equivalent to that of PP. In Simon (1981), this model is called a "P-time stochastic RAM". We refer to it as a PP-RAM[], so as to place it inside the larger framework of other choices of acceptance criteria. If execution time is not restricted to be polynomial, we call this a *stochastic RAM* as per Simon's terminology.

Simon claims regarding PP-RAM$[+, \dot{-}, \leftarrow, \mathit{Bool}; \leq]$ that it equals ER. This error is clearly the result of propagation from the paper's earlier mistakes, discussed in the previous chapter. The correct (and stronger) claim is

Theorem 15. *For a constructable $op \supseteq \{\leftarrow, \mathit{Bool}\}$ and any $f(n)$,*

$$\begin{aligned} O(f(n))\text{-}\mathit{stochastic\ RAM}[op] &= O(f(n))\text{-}\mathit{stochastic\ RAM}_0[op] \\ &= \mathit{EL}_{op}(O(f(n)))\text{-}\mathit{TM} \\ &= \mathit{R\text{-}EL}_{op}(O(f(n)))\text{-}\mathit{TM} \\ &= \mathit{N\text{-}EL}_{op}(O(f(n)))\text{-}\mathit{TM} \\ &= \mathit{EL}_{op}(O(f(n)))\text{-}\mathit{SPACE\text{-}TM} \\ &= \mathit{R\text{-}EL}_{op}(O(f(n)))\text{-}\mathit{SPACE\text{-}TM} \\ &= \mathit{N\text{-}EL}_{op}(O(f(n)))\text{-}\mathit{SPACE\text{-}TM}. \end{aligned}$$

Proof. The equivalence of most classes described is given in the proof of Theorem 8. Regarding the equivalence of the randomised TM classes, this follows from the same reasoning as P $\subseteq$ RP $\subseteq NP$, noting that at no point did this reasoning rely on the execution time being polynomial. The argument is equally valid when the execution time is $\mathrm{EL}_{op}(\mathrm{O}(f(n))$. In fact, it continues to be valid when considering space-bounded TMs, rather than time-bounded ones.

We need therefore only prove the following relations.

$$\mathrm{O}(f(n))\text{-stochastic RAM}[op] \subseteq \mathrm{R\text{-}EL}_{op}(\mathrm{O}(f(n)))\text{-SPACE-TM}. \tag{6.1}$$

$$\mathrm{O}(f(n))\text{-stochastic RAM}_0[op] \supseteq \mathrm{EL}_{op}(\mathrm{O}(f(n)))\text{-SPACE-TM}. \tag{6.2}$$

Equation (6.1) is proved along the same lines as Theorem 8, noting that the *RAND* operation can be simulated by an $\mathrm{R\text{-}EL}_{op}(\mathrm{O}(f(n)))$-SPACE-TM by allotting the random bits individually. Because the machine is not time restricted, this is possible to do. Where the

construction of Theorem 8 fails is in that it does not handle indirect addressing. We therefore provide a more powerful simulating algorithm for the TM, which can account for it.

In the new simulation, instead of storing the value of each of the RAM's registers in a portion of the TM's tape that has been specifically labelled for this use, we store the values of all non-zero-valued registers as (*address*, *value*) pairs. In each step of execution, we scan the available pairs for those with the appropriate addresses (creating a new pair for the target register if necessary), copy the values into a scratchpad area, calculate over the scratchpad and then copy the result out. Because at each step of the RAM at most one new tuple is created, the total space requirements are still within $\mathrm{EL}_{op}(\mathrm{O}(f(n)))$.

Equation (6.2), although more involved, follows from the proof of Theorem 9. The theorem itself states that all r.e. sets can be recognised in O(1) given $op = \{\leftarrow, Bool\}$ and an Oracle-given integer. The proof of Theorem 9 allows one to bound the size of the required integer. It is no more than what can be produced in $\mathrm{O}(f(n))$ exponentiation steps. Because these can be performed by the left-shift operation within the allotted time, it is possible to use the *RAND* operation in the PP-RAM model to simulate an NP-RAM execution, for which we know (from Theorem 9) that it can simulate the desired TM.

The actual simulation follows the argument for NP $\subseteq$ PP. If the NP-RAM is to accept the input, there is an Oracle number, o that will lead to acceptance. If o can be bounded by 2^k, the calculation $X \leftarrow RAND(2^k)$ yields a number that has a positive probability of being equal to o, and therefore having a positive probability of leading to acceptance. We therefore first compute $RAND(2)$ and reject the input if the answer is zero. If it is not zero, we compute 2^k (which we can do within the required time budget), and from it X, then use X as if it was provided by an Oracle. This leads to correct rejection having probability 0.5 and correct acceptance probability being (slightly) larger. Thus, the computation is in PP-RAM. □

Simon conveys dissatisfaction with his own result. He writes:

> The previous results rely, perhaps unfairly, on our definition of acceptance for stochastic RAMs. It would be more reasonable to require acceptance with bounded

> error probability. Ideally, the machine should not accept x if x is not in the language, and accept it with high probability if it is. We have been unable to prove [the theorem] for this strong notion of acceptance. (Simon, 1981)

Utilising modern terminology, we can say that the acceptance criterion wished for by Simon is that of RP, and one can therefore say that the open problem described is to characterise the power of RP-RAM[*op*]. When not restricting to polynomial time execution, we will refer to this computation model as an RRAM. The problem of characterising the power of RRAMs in general and RP-RAM in particular has remained open now for over 30 years. The present work closes this longstanding open problem by presenting a proof to the following theorem.

Theorem 16. *For any $f(n)$,*

$$O(f(n))\text{-}RRAM[+,[\dot{-}],[\times],[\div],\leftarrow,[\rightarrow],Bool] = O(f(n))\text{-}RAM[+,[\dot{-}],[\times],/,[\div],\leftarrow,[\rightarrow],Bool].$$

In particular,

$$RP\text{-}RAM[+,[\dot{-}],[\times],[\div],\leftarrow,[\rightarrow],Bool] = P\text{-}RAM[+,[\dot{-}],[\times],/,[\div],\leftarrow,[\rightarrow],Bool] = PEL.$$

The power of RAM$[+,[\dot{-}],[\times],/,[\div],\leftarrow,[\rightarrow],Bool]$ has been characterised by Theorem 8.

The result suggests, perhaps surprisingly, that a random number is more powerful than an arbitrary number for the operation set $\{+,\leftarrow,Bool\}$, but an arbitrary number is more powerful than a random number for the operation set $\{+,/,\leftarrow,Bool\}$. (The latter conclusion can be derived from Theorem 5 because an RRAM$[+,/,\leftarrow,Bool]$ working in O(1) time can only produce a bounded number of random bits, and so can be simulated in bounded time by a deterministic TM that enumerates over the random choices. Therefore, it cannot solve the halting problem, which the corresponding ARAM does.)

Note also that

$$\text{P-RAM}[+,/,\leftarrow,Bool] \supset \text{P-RAM}[+,\leftarrow,Bool],$$

(The former equals PEL by Theorem 8, the latter equals PSPACE by Theorem 6, and the two are known to be distinct (Stearns et al., 1965; Geffert, 2003; Seiferas, 1977)) so, by the theorem,

$$\text{RP-RAM}[+, \leftarrow, \mathit{Bool}] \supset \text{P-RAM}[+, \leftarrow, \mathit{Bool}].$$

Thus, Theorem 16 provides, for the first time, evidence that the RP model may be strictly more powerful than the P model, a question for which the answer was so far not known within the context of RAMs, and which, in the context of TMs, is one of the central open problems in computer science.

Before presenting the proof of Theorem 16, we begin with the following helper lemma.

Lemma 12. *For any $f(n)$,*

$$O(f(n))\text{-}RRAM_0[+, \leftarrow, \mathit{Bool}] \supseteq EL_{op}(O(f(n))\text{-}SPACE\text{-}TM.$$

The proof of this lemma spans Sections 6.2–6.7.

6.2 Outline of the proof

Similarly to what we have done in Theorems 8 and 9, one of the main tools we use here is Lemma 11, which guarantees an O(1)-time nondeterministic verification of any TM, given K Oracle-provided integers. In the original proof of Lemma 11, we used $K = 5$. The proof of Theorem 9 later showed that K can be reduced to 1. Here, we take a middle ground, with $K = 2$. Let w_1 through w_5 be as defined in the proof of Lemma 11. If w_1 is given by the Oracle, then for w_3 it is possible to use $(3w_1) \leftarrow (3w_1)$, which can be calculated from w_1 in O(1) time. The remaining values can be encoded into a single integer by packing them as the contents of a vector, X, of length 3 and width $w = w_3 + 3w_1$.

In Theorem 16 all computations must be performed without access to any Oracle. We will therefore compute w_1. The parameter w_1 signifies, in the proof of Lemma 11, the amount of tape utilised by the simulated TM. There is no need to compute this amount exactly, because

there is no penalty for overestimation. The number w_1 therefore only needs to be "large enough", and it can be computed by means of a function like *el*, described in Algorithm 10.

Algorithm 10 Calculating w_1

```
1: function EL(inp, N)
2:     rc ⇐ inc(inp)
3:     for i ∈ 0, ..., N do
4:         rc ⇐ rc ← rc
5:     end for
6:     return rc
7: end function
```

For all operation sets discussed in Theorem 16, this function is the *el* program guaranteed by definition to exist for any RAM-constructable operation set, which, basically speaking, generates numbers that are as large as possible, as quickly as possible.

Note that w_1 (and therefore also w, the width of X) is calculated with no need of the right-shift operator. We can therefore assume the availability of right-shifting, as per Lemma 1, in order to unpack X into its elements, w_2, w_4 and w_5.

Our general algorithm for the solution of the problem, without resorting to the use of Oracles, is to try out, simultaneously and in parallel, many candidates for X, much as was done in Theorem 8. A high-level overview is given in Algorithm 11.[1]

The loop of Step 2 of Algorithm 11, similar to the outside loop of Algorithms 1 and 7, is a means of simulating a TM of unknown space requirements, s, by use of a sequence of RAM programs that simulate the TM over a predetermined and increasing sequence of candidate s values. Each program in the sequence simulates a TM running on $s = el(maxstep) \geq \mathrm{EL}_{op}(maxstep)$ space in $\mathrm{O}(maxstep)$ time for the simulating RAM. The outside loop ensures that even without prior knowledge of the needed s, the total simulation time for the RAM will be $\mathrm{O}(\beta(s))$, where β is the functional inverse to the EL_{op} function.

In order to prove the theorem, we need to show that each iteration of the loop of Step 2 can be completed in a total of $\mathrm{O}(maxstep)$ time. Specifically, we will show that Step 8 can be performed in $\mathrm{O}(maxstep)$ time, and the entire loop of Step 9 can subsequently run in $\mathrm{O}(1)$ time, by parallelising over all candidates for X.

[1] As is, this algorithm is still not in RRAM$[+, \leftarrow, Bool]$. However, in Section 6.7 we show how it can be made into an RRAM.

Algorithm 11 Simulating a TM on an RRAM$[+, \leftarrow, \mathit{Bool}]$

1: $\mathit{maxstep} \Leftarrow 1$
2: **loop**
3: $s \Leftarrow \mathit{el}(\mathit{inp}, \mathit{maxstep})$
4: $w_1 \Leftarrow s$
5: $m \Leftarrow w_1 + w_1 + w_1$
6: $w_3 \Leftarrow m \leftarrow m$
7: $w \Leftarrow w_3 + m$
8: Generate candidates for X
9: **for all** Candidate X **do**
10: Extract $[w_2, w_4, w_5]$ as elements of the encoded vector $(m, X, 3)$
11: Check if $[w_1, \ldots, w_5]$ describes a consistent tableau, correct for the simulated TM.
12: Check if the tableau is correct for the given input.
13: Check if the final state described by the tableau is accepting.
14: **end for**
15: **if** all three checks above pass **then return** Accept
16: **end if**
17: $\mathit{maxstep} \Leftarrow \mathit{maxstep} + \mathit{maxstep}$
18: **end loop**

The rest of the proof is organised as follows. We begin by demonstrating how Steps 10, 11 and 13 can all be performed in parallel. We then go back to explain Step 8. Then, in the last part of the proof, we discuss Step 12.

6.3 Parallelised extraction and verification

Let us begin by considering the loop of Step 9 of Algorithm 11. Lemma 11 provides the means to perform one such verification in O(1) time. The parallelisation of all checks has so far been discussed only in Theorem 8, where extensive use was made of the division operator, which is not available in the present context. However, it is illuminating to note that most of the construction of Theorem 8 remains intact: any Boolean operation and any shift used in the proof is implicitly parallel, operating simultaneously on all X candidates (utilising the fact that the shifts used are all shifts by a function of s, never a function of X, and no use is made of the special properties of the tweaked negation operator).

In fact, examining the details of the proof of Theorem 8 reveals that division was used in exactly three contexts:

1. In generating U^T,

2. In generating $O^k_{1,T}$, and

3. In generating $O^k_{inp,T}$.

The first value was calculated as per Equation (4.3), and the latter two by use of Equation (3.3).

Note further that in our present algorithm the only part of the verification process that depends on *inp* is Step 12. We will discuss this step in Section 6.6. For all other steps, the value of $O^k_{inp,T}$ is not needed. The algorithm therefore merely requires the first two values to be calculated. Suppose, however, that the output of Step 8 is two integers, L and I, such that L can be used in place of the U^T value and I can be used in place of $O^k_{1,T}$. When that is the case, simply following the proof of Theorem 8 produces the required O(1)-time parallelisation over Steps 10, 11 and 13 of the loop in Step 9. The value pair (L, I) carries the information of the X candidates to be verified.

Sections 6.4–6.5 now deal with the generation of this pair, meant to implement Step 8 of Algorithm 11.

6.4 Producing X candidates as maps

Unfortunately, we cannot produce X candidates by directly following the same methods as in Theorem 8, because division is not available. In order to replace division by stochasticity in the generation of L and I, we will relax, somewhat, our requirements of L and I, instead of trying to produce $L = U^T$ and $I = O^k_{1,T}$ exactly. This can be done as long as we do not compromise the usability of the numbers produced for the type of parallelisation described in Section 6.3, above.

Specifically, the construction of Theorem 8 packs all X candidates side-by-side to each other, with no spacing between them in the bit-string representing L. We will now relax this condition, allowing arbitrary amounts of spacing between the candidates.

More formally, we introduce the following definition.

Definition 10 (Map)**.** A *map* (or, an *encoded map*) is a triplet, (L, I, w), of integers, where L is the *contents* of the map, I is the *domain* of the map and w is the *width* of the map. The domain, I, must satisfy the criterion that any two "1"s in its binary representation are at least w bit positions apart. The contents, L, must satisfy $L = L \wedge ((I \leftarrow w) - I)$ and for simplicity we assume $w > 1$.

Conceptually, a map represents a mapping from a list of *indices* $ind_1, \ldots$ to a list of *map elements* $k_1, \ldots$, in the following way: $I = \sum_i (1 \leftarrow ind_i)$ and $L = \sum_i (k_i \leftarrow ind_i)$. The k_i are required to be in the range $0 \leq k_i < 2^w$. The ind_i are required to satisfy that for every i, $w + ind_i \leq ind_{i+1}$. Under these restrictions, there is a one-to-one correspondence between mappings and encoded maps. Such mappings will be termed *decoded maps*. A map will often be referred to by the name of the encoded map contents. This is done, for example, in the following notation, which we introduce to signify the relation between an encoded map and its underlying mapping: $k_i = L[ind_i]$.

Note that the map elements belong to a finite set, the elements of which are representable by w-bit strings. As such, it is well-defined to consider their most significant bits (MSBs).

Operations on maps are RAM operations performed on the encoded maps. We will, however, mostly be interested in the effects these have on the map elements. Such operations may include multiple maps that share the same I and w, in which case, when we say that we apply a function f on two maps, V and U, we typically mean that we wish to produce a result, W, such that for all indices $W[ind_i] = f(V[ind_i], U[ind_i])$.

To demonstrate that maps provide all the functionality that was required in Section 4.3 from vectors, and generally by use of the same methods, we briefly develop here an O(1)-time function that calculates element-wise equality between two maps, this following the element-wise equality function between two vectors that is developed in Appendix A.4 and used in Section 4.3.

Let (V, I, w) and (U, I, w) be two maps.

The total set of bits that can be used by the map contents is

$$MASK(I, w) = (I \leftarrow w) \dot{-} I.$$

For convenience, we divide these into two subsets. We take the lowest $w-1$ bits of each map element to be its "data" bits, and the MSB to be the "flag" bit. The following functions extract these positions.

$$FLAGS(I,w) = I \leftarrow (w \dot{-} 1),$$

$$DATA(I,w) = FLAGS(I,w) \dot{-} I.$$

To actually extract the data from the positions we use

$$FLAGS(V,I,w) = V \wedge FLAGS(I,w),$$

$$DATA(V,I,w) = V \wedge DATA(I,w).$$

Summing the data of two maps can be performed by

$$ADD(V,U,I,w) = (DATA(V,I,w) + DATA(U,I,w)) \oplus FLAGS(V,I,w) \oplus FLAGS(U,I,w),$$

where "$\oplus$" signifies the exclusive or (XOR) bitwise operation. The ADD function, as implemented here, avoids overflows by calculating the sum modulo 2^w. In order to find out if an overflow occurred, we can calculate the carry bit.

$$CARRY(V,U,I,w) = (V + U \dot{-} ADD(V,U,I,w)) \rightarrow w.$$

The following function implements bitwise negation:

$$NEG(V,I,w) = MASK(I,w) \dot{-} V$$

Calculating a map, RC, such that for every i, $RC[ind_i]$ is 1 if $V[ind_i] > U[ind_i]$, and is 0, otherwise, this being an element-by-element "greater than" comparison:

$$GT(V,U,I,w) = CARRY(V, NEG(U,I,w), I, w).$$

With this build-up, we can finally implement element-by-element equality as follows:

$$EQ(V, U, I, w) = I \dot{-} GT(V, U, I, w) \dot{-} GT(U, V, I, w).$$

In order to satisfy the requirements of Section 6.3, what we need is a map, (L, I, w), in which with probability of at least 0.5 there exists an element equal to the desired X. Because we do not know in advance which X is the desired one, we simply extend this requirement to be true for any of the possible X values.

Once such a map is available, all operations required by Theorem 8 (Boolean operations, left shift, addition and, as a special case of addition, incrementation) can all be done as previously demonstrated, switching in L and I for U^T and $O^k_{1,T}$, respectively, wherever necessary.

6.5 Generating L and I

Summarising the conditions required of Step 8 of Algorithm 11:

1. In the binary representation of the output I, any two "1"s should be at least w bit positions apart.

2. Any bit string of length w should appear with probability of at least 0.5 as a substring of the bits of L beginning at some "1" position of I.

3. The generation of I and L, with $w \geq EL_{op}(maxstep)$, should be performed in O($maxstep$) time.

In this description of our requirements we omitted the criterion that L must be zero in all bit positions other than those describing map elements. The reason for this omission is that this last condition is unnecessary: simply perform

$$L \Leftarrow L \wedge MASK(I, w)$$

in order to make (L, I, w) a valid map.

Of the three conditions listed for L and I, the second is the most straightforward: if the Hamming weight of the bit-string of I is t, then there are t candidates for the tableau simultaneously being tested. Pick L to be a random integer taken from the range of numbers up to $I \leftarrow w$ or higher. Under this choice, the t candidates are uniformly distributed and independent. Each map element therefore has probability 2^{-w} to be the correct tableau. By choosing $t \geq 2^w$ we guarantee that the probability that none of the elements have the appropriate value is bounded by approximately 1/e, so better than the 0.5 that is required.

Generating I is more difficult. It must meet two criteria: first, its "1"s must be separated by at least w bit positions; second, with high probability it must have a Hamming weight of at least 2^w. For now, let us concentrate on the problem of satisfying the first criterion.

One thing that is important to note regarding these criteria is that the second condition is a probabilistic condition, but the first condition is a deterministic one. We cannot generate I by a random process that is simply biased towards "0" bits. Even if such a process has high probability of meeting the criterion, it still admits the possibility that the condition will not be satisfied and, as a result, that the calculation will be incorrect. Such an approach can create a positive false-accept probability, which is not acceptable in the RP model.

Let R_i, for $i = 0, \ldots$, be independent random values generated by calls to $RAND(2^k)$ for some chosen k. We note that for a fixed-sized w, a suitable I can be devised in O(1) time simply by applying Algorithm 12.

Algorithm 12 Creating I for a fixed-sized w

1: $I \Leftarrow R_0$
2: **for** $j = 1, \ldots, w - 1$ **do**
3: $\quad I \leftarrow I$ clr $(I \leftarrow j)$
4: **end for**
5: **return** I

This procedure will work for any w, but the time it takes is O(w). We, on the other hand, require O(*maxstep*) time, for values of w that are in EL_{op}(O(*maxstep*)). To solve this, we introduce an induction step. At step i, the input will be I_i and w_i, where I_i is a vector that satisfies the sparseness condition for w_i. The output will be I_{i+1}, satisfying the condition for $w_{i+1} = (w_i \leftarrow w_i) + 1$. By implementing each such step in O(1)-time, it is possible to reach

$w = el(\mathrm{O}(s))$ in $\mathrm{O}(s)$ steps and $\mathrm{O}(s)$ time. For the chosen operation sets, $\mathrm{EL}_{op}(s)$ is $el(\mathrm{O}(s))$, so the procedure meets the time bound.

The full code for implementing the induction step is given in Algorithm 13. The algorithm is described with use of "$\dot{-}$" and "$\rightarrow$", as we have already established that the operation set given suffices to simulate both.

Algorithm 13 Creating (I_{i+1}, w_{i+1}) from (I_i, w_i)

1: $R_i \Leftarrow RAND(2^k)$
2: $I_{begin} \Leftarrow I_i \ \mathsf{clr}\ (I_i \leftarrow w_i)$
3: $I_{end} \Leftarrow (I_i \leftarrow w_i) \ \mathsf{clr}\ I_i$
4: $I_{middle} \Leftarrow I_i \dot{-} I_{begin}$
5: $I_{goodbegin} \Leftarrow EQ(R_i \wedge MASK(I_{begin}, w_i), 0, I_{begin}, w_i)$
6: $I_{goodend} \Leftarrow EQ((R_i \leftarrow w_i) \wedge MASK(I_{end}, w_i), MASK(I_{end}, w_i), I_{end}, w_i)$
7: $I_{goodmiddle} \Leftarrow EQ(ADD((R_i \leftarrow w_i) \wedge MASK(I_{middle}, w_i), I_{middle}, I_{middle}, w_i), R_i \wedge MASK(I_{middle}, w_i), I_{middle}, w_i)$
8: $I_{i+1} \Leftarrow (MASK(I_{goodbegin} + I_{goodmiddle}, w_i) + I_{goodbegin}) \wedge I_{goodend}$
9: **return** $(I_{i+1}, (w_i \leftarrow w_i) + 1)$

The idea behind Algorithm 13 is to use the same tools we developed in order to verify a tableau, only this time to use them in order to verify the separation between "1" bits. Specifically, we begin by assuming that every "1" bit in I_i is followed by w_i zero bits. The entire integer can therefore be thought of as occurrences of the repeating pattern "$(0^{w_i-1}1)^+$", separated by zeroes. We want to measure the lengths of these repeating patterns. It is straightforward to find where such a pattern begins, where it ends, and which "1" bits are its middle bits. These are designated I_{begin}, I_{end} and I_{middle}, respectively. The question is only how to count the number of consecutive "middle" bits.

To do this, we treat R_i as an Oracle string. Specifically: a counter. We verify (in $I_{goodbegin}$, $I_{goodend}$ and $I_{goodmiddle}$, respectively) that the counter begins with a zero, ends with $2^w - 1$ and increments each time by one. In the last step, the addition by $I_{goodbegin}$ sends a carry bit through the entire verified part of the vector. If it reaches $I_{goodend}$, this is an indication that the counting was correct. We can therefore now take one bit ($I_{goodend}$) from the sequence, knowing that there must be at least $w_i \times 2^{w_i}$ zero bits preceding it.

We remark that because all we are interested in is to verify that the "1" bits are spaced far enough apart, it is not important to check, for example, that the counter did not go through

several entire revolutions, instead of just one, between the beginning and the ending of each repetition. The omitted checks are all for conditions which, if invalidated, merely extend the number of zero bits that separate the "1" bits.

6.6 Verifying the input

In the previous sections, we have shown how tableau candidates can be generated, tested for satisfying the criterion of properly advancing the simulated TM and checked for their terminating condition, all within the allotted time budget. To make the validation complete, we must also verify that the tableau begins with the appropriate value.

Specifically, the elements of the map L are candidate X values, which are vectors, the elements of which are $[w_2, w_4, w_5]$, of which the first element, w_2, is, itself, a vector (it is the contents of a tableau), of which the first element is the initial instantaneous description of the TM being described by the tableau, which is, by definition, $tape + 2^{s+c-1} state + 2^{2(s+c-1)} head$, as they are in the beginning of the execution (where s is the length of the TM's tape and c is the number of bits required to store the state of the finite control). The vector's width is $tableauwidth = 3(s + c - 1)$.

By construction, *tape* is initialised as *inp*, *state* is initialised as 0 and *head* is initialised as 1. The value of the tableau's first element should therefore be $init \stackrel{\text{def}}{=} inp + 2^{2(s+c-1)}$. The values of s and c are known, so this is a value that is both known and easy to calculate given the available operations. The problem is that in order to verify in parallel over all tableau candidates whether they begin with the correct elements, we need access to the value $init \times I$. If the operation set available to the simulating RAM had contained multiplication, this value would have been straightforward to calculate, however, in the present context we must assume multiplication not to be available, and must therefore provide alternate means to create the same effect.

To do this, we begin by making a small change in our TM simulation: instead of having consecutive elements of the tableau represent progressively advancing instantaneous descriptions of the simulated TM, we will have them represent progressively *less advanced* instantaneous descriptions. In other words, we reverse the element order within the (correct) tableau. The equations to verify a correct advancement of the TM must now compare an element with its preceding element, rather than with the element following it, and the test for the TM's halting state is now a test over the vector's first element, rather than over its last. Neither of these changes makes a material difference in the presented algorithms.

Similarly, X will no longer represent $[w_2, w_4, w_5]$. Rather, it will represent the reversed-order $[w_5, w_4, w_2]$.

Together, these two changes have the effect that the test that is still missing from the simulation, namely the test for the correct initial instantaneous description, is now no longer a test of the least significant bits of X, but rather a test of its most significant ones.

As the next step, consider that the tools which we have already developed suffice in order to simulate a TM working without input. In this case, $inp = 0$, $init = 2^{2(s+c-1)}$, and $init \times I$ can be calculated using the available operations as $I \leftarrow (s + s + c + c \dot{-} 1 \dot{-} 1)$.

In fact, the vector being verified does not need to correspond to a tableau, and the function relating consecutive elements need not be the advancement function of a TM: in Algorithm 13 it was demonstrated, for example, that the tools available are already enough in order to verify whether a sequence is the all-ones vector and whether a sequence is an arithmetic progression starting with 0 and advancing by one between consecutive elements (a "counter"). Let us therefore, for now, ignore the original TM that we intend to simulate and ignore its input. Instead, let us assume that we have at our disposal a pair (I_x, w_x), where w_x is not equal to w but rather larger than $(2^w) \leftarrow (2^w)$. (Such a pair can be found simply by running Algorithm 13 two additional iterations.) Consider, now, Algorithm 14, where I_x, w_x, w and *init* are as above, and where *tableauwidth* is the width of the tableau to be verified (the bit length of each of its elements).

Let us analyse this algorithm line by line.

Algorithm 14 Verifying the input

1: **function** INPUTVERIFY($I_x, w_x, w, init, tableauwidth$)
2: $elementwidth \Leftarrow 1 \leftarrow w$
3: $width \Leftarrow elementwidth \leftarrow elementwidth$
4: $L_{const} \Leftarrow RAND(2^k) \wedge MASK(I_x, width)$
5: $I_{goodbegin} \Leftarrow EQ(L_{const} \wedge MASK(I_x, elementwidth), I_x, I_x, elementwidth)$
6: $I_{goodtransition} \Leftarrow (MASK(I_{goodbegin}, width)\ \mathsf{clr}\ (L_{const} \oplus (L_{const} \leftarrow elementwidth))) \vee MASK(I_{goodbegin}, elementwidth)$
7: $I_{good} \Leftarrow I_{goodbegin} \wedge ((I_{goodtransition} + I_{goodbegin}) \rightarrow width)$
8: $L_{counter} \Leftarrow RAND(2^k) \wedge MASK(I_{good}, width)$
9: $I_{goodbegin} \Leftarrow I_{good} \wedge EQ(L_{counter} \wedge MASK(I_x, elementwidth), 0, I_x, elementwidth)$
10: $temp \Leftarrow ADD(L_{counter}, L_{const}, L_{const}, elementwidth) \leftarrow elementwidth$
11: $gbmask \Leftarrow MASK(I_{goodbegin}, elementwidth)$
12: $I_{goodtransition} \Leftarrow (MASK(I_{goodbegin}, width)\ \mathsf{clr}\ (L_{counter} \oplus temp)) \vee gbmask$
13: $I_{good} \Leftarrow I_{goodbegin} \wedge ((I_{goodtransition} + I_{goodbegin}) \rightarrow width)$
14: $M \Leftarrow MASK(I_{good} \leftarrow (init \leftarrow (w + w \dot{-} tableauwidth)), elementwidth \leftarrow (w \dot{-} tableauwidth))$
15: $L_{output} \Leftarrow L_{counter} \wedge M$
16: $I_{output} \Leftarrow L_{const} \wedge M$
17: **return** $(L_{output}, I_{output}, elementwidth)$
18: **end function**

Steps 4 through 7 generate a map $(L_{const}, I_{good}, width)$ that has the bit-string

$$\left(0^{elementwidth-1}1\right)^{2^{elementwidth}}$$

as each one of its elements. The program performs this by considering all elements in the map $(L_{const}, I_x, width)$ and filtering out first those indices whose elements do not begin with the substring

$$0^{(elementwidth-1)}1$$

(Step 5), and then those elements which are not composed entirely of repetitions of a constant string of length *elementwidth*. The latter is tested in Step 6 by verifying equality between each substring of length *elementwidth* of L_{const} and the substring of the same length following it immediately. As was done in Algorithm 13, an addition operation, carried out on Step 7, propagates a carry bit through every element. The good elements, remaining in the final index set, I_{good}, are those for which the carry propagated through the entire element, thereby verifying that all the element's bits are correct.

A similar technique, used in Steps 8 through 13, filters I_{good} even further, until it is known additionally that every element of the map $(L_{counter}, I_{good}, width)$ is a counter of width *elementwidth*. That is to say, its first *elementwidth* bits are all zero, its next *elementwidth* bits are the binary representation of the number 1, and so on, in arithmetic progression, until the last element, being $2^{elementwidth} - 1$. This second phase of filtering on I_{good} is, once again, performed by verifying first the lowest *elementwidth* bits (which, in this case, must equal 0) and then the relation between each element and the next (which is here incrementation). Simultaneous incrementation of all substrings of length *elementwidth* is done by adding L_{const} to $L_{counter}$.

The two maps generated, $(L_{const}, I_{good}, width)$ and $(L_{counter}, I_{good}, width)$, now have in any one of their remaining elements (assuming such elements still exist) an exact copy of the value $O_{1,elementwidth}^{(2^{elementwidth})}$ and the value $U^{elementwidth}$, respectively, much like those that are used in the proof of Theorem 8.

Now, however, we consider a new map: $(L_{counter}, L_{const}, elementwidth)$. If this map has any elements at all, then it has every possible element of bit-length *elementwidth*. In particular, it would have our desired tableau.

However, while this procedure has so far presented an alternate method for producing candidates for X, it still has not addressed the main problem of verifying that the candidate tableau begins with the correct bit-string, *init*.

The method by which this entire construction can now overcome the problem of verifying *init* is by noting that in the new structure the value of each tableau candidate is determined completely by its bit-position relative to the "1" bit of I_{good} immediately preceding it. Specifically, the mask M, built in Step 14, is able to mask out all candidates whose initial bits do not match the desired value.

We remark that M is constructed by use of left-shifting by a value that was calculated as a function of *inp*, and not just as a function of w_1. However, because the shift-by value is calculated, once again, with no use of right shifting, we are still justified, by Lemma 1, in continuing to assume the availability of right-shifting, even though it is not part of the RAM's native operation set.

6.7 Completing the proof

Proof of Lemma 12. The techniques developed above provide most of the proof of the lemma. Two more details are still missing. In order to claim that the algorithm developed is in RRAM, we must show that

1. it is guaranteed to terminate within the allotted time, and

2. if *inp* is in the language, its probability of being accepted is greater than 0.5.

Algorithm 11, as presented, does not meet the first of these conditions. This algorithm, regardless of its success probability, is not guaranteed to terminate at all, and much less within a time budget. A bad choice of random variables (e.g. consistently allotting the value 0) will cause the algorithm to continue indefinitely, regardless of how many steps are made by the simulated TM.

To fix this, we change the termination condition in Step 15 of Algorithm 11. As written, this termination condition merely checks whether the simulation succeeded *and* the simulated TM halted *and* the halting state was an accepting one. Instead, we can check each of these conditions separately, and choose as follows.

- If the simulation failed, halt and reject the input; else

- If the simulated TM halted on a rejecting state (or entered an infinite loop), halt and reject the input; else

- If the simulated TM halted and accepted the input, halt and accept the input; else

- In the remaining case (the simulated TM exhausted its tape) continue to the next iteration of the loop in Step 2.

We see that this change now fits within the time budget: the total number of iterations through Step 2 cannot exceed the number of iterations used in the deterministic simulation (of the same time budget) presented in Algorithm 7. Any bad choice of random numbers can only cause the program to halt prematurely (as per the first item listed above). Furthermore,

because all such halts reject the input, an input value that is not in the language is guaranteed to be rejected.

The remaining condition required for the simulation is that for a value of *inp* in the language, the probability for acceptance is at least 0.5. The only possibility for a false reject is, as before, a failure of the simulation due to a bad choice of random numbers. We need this probability to be bounded by 0.5 in total, regardless of the number of times the program iterates through the loop of Step 2. To do this, we will design the program so that a failure of the simulation will occur with probability less than 1/4 in the first iteration, 1/8 in the second iteration, 1/16 in the third iteration, and so on. The probability that *any* of the choices is bad is no higher than the sum of the probabilities across all choices, hence no more than 0.5.

Let us now bound from above the probability of a bad choice. A bad choice, specifically, is one that ultimately results in $I_{good} = 0$ in Step 13 of Algorithm 14. Consider, therefore, a single bit of I_{good} (of sufficiently large significance value), and let us calculate its probability of being "1".

A bit of I_{good} is "1" if

1. The *width* elements of L_{const} beginning at the chosen bit position all have exactly the required values.

2. The *width* elements of $L_{counter}$ beginning at the chosen bit position all have exactly the required values.

3. This bit position is "1" in I_x.

Of these, the first two conditions relate to bits that are output directly by *RAND*. As such, the probability for their occurrence is straightforward to calculate. It is $2^{-2width}$. The third condition, however, is regarding a value that is an output of Algorithm 13. The value I_x is really $I_{maxstep+2}$ of that algorithm. For it to be "1", approximately w_x different bits need to have exactly the correct values in both $R_{maxstep+1}$ and $I_{maxstep+1}$. As before, the former is directly an output of *RAND*, but the latter is calculated recursively. (Requiring the various random integers to have specific bit values in specific bit positions is, in fact, a stricter requirement than what is actually necessary. Several different bit masks may work

equally well. Nevertheless, in order to keep the calculation simple, wherever several different bit values can be used to reach the same result, we pick one of them that constrains the least number of bits. In this way, we bound from below the probability that a particular bit of I_x is "1". The bound itself may be very loose.)

Ultimately, a value of "1" in a specific position of I_{good} relies on $\mathrm{O}(w_x)$ randomly, independently and uniformly chosen bits being of pre-specified values in each of $\mathrm{O}(maxstep)$ integers. The probability is therefore $p \stackrel{\mathrm{def}}{=} 2^{-\mathrm{O}(w_x \times maxstep)}$. Let W (being of the same order of magnitude as w_x) be the length of the bit-window where all these affected bits lie. Now, consider I_{good} as a bit-vector split into windows of size W. In each of these windows, the probability of having a "1"-bit is bounded from below by p, and the probability that any t of them have no "1"-bits is bounded from above by $(1-p)^t$.

If we were to choose k, the bit-length of all our random numbers, to be $K \geq W/p$, which is $W \times 2^{\mathrm{O}(w_x \times maxstep)}$, we ensure that the probability of a non-zero I_{good} is at least $(1-p)^{1/p} \approx 1/\mathrm{e}$. If, however, we choose

$$k \Leftarrow K \leftarrow K,$$

then this probability drops much more quickly than the $1/4, 1/8, 1/16, \ldots$ geometric sequence described earlier, and the probabilities will certainly sum up to less than $1/2$.

Clearly, calculating a k value that is at least as large as this is possible in $\mathrm{O}(maxstep)$ time. □

We now turn to proving the main theorem.

Proof of Theorem 16. Let $fullop = \{+, \dot{-}, \times, /, \div, \leftarrow, rightarrow, Bool\}$ and let op be an operation set in $\{+, /, \leftarrow, Bool\} \subseteq op \subseteq fullop$.

The relation

$$\mathrm{O}(f(n))\text{-RRAM}[fullop] \subseteq \mathrm{O}(f(n))\text{-RAM}[op]$$

is evident from

$$\begin{aligned}\text{O}(f(n))\text{-RRAM}[\mathit{fullop}] &\subseteq \text{O}(f(n))\text{-stochastic-RAM}[\mathit{fullop}]\\ &= \text{EL}_{\mathit{fullop}}(\text{O}(f(n))\text{-SPACE-TM}\\ &= \text{EL}_{\mathit{op}}(\text{O}(f(n))\text{-SPACE-TM}\\ &= \text{O}(f(n))\text{-RAM}[\mathit{op}],\end{aligned}$$

where the relation between RRAMs and stochastic-RAMs stems from the fact that any RRAM is by definition also a stochastic-RAM, the first equality is simply a special case of Theorem 15, the second equality stems from the functions EL_{op} and EL_{fullop} being equal, and the third equality is a special case of Theorem 8.

The reverse direction,

$$\text{O}(f(n))\text{-RAM}[\mathit{fullop}] \subseteq \text{O}(f(n)\text{-RRAM}[\mathit{op}],$$

completing the statement of the theorem, is provided by Lemma 12. □

6.8 Generic *RAND*

The discussion above pertained to a RAM model that incorporates an $X \Leftarrow \mathit{RAND}(2^k)$ function. This still leaves the question of whether the more general $X \Leftarrow \mathit{RAND}(Y)$ is perhaps more powerful. Here we prove that this is not the case. Formally:

Corollary 16.1. *For any $f(n)$,*

$$f(n)\text{-}\mathit{RRAM}[+, [\dot{-}], [\times], [\div], \leftarrow, [\rightarrow], \mathit{Bool}] = f(n)\text{-}\mathit{R^*RAM}[+, [\dot{-}], [\times], [\div], \leftarrow, [\rightarrow], \mathit{Bool}],$$

where R^*RAM is the computational model that provides a generic RAND function.

Proof. First, due to Theorem 16, we know that we can assume the existence of "$\div$", "$\times$" and "$\dot{-}$", from which we can further assume the existence of a modulo operation.

Second, considering the technique that was showcased in Algorithm 1 and then reused in Algorithms 7 and 11, we know that it is enough if we are able to simulate the R*RAM by an RRAM (with appropriate success criteria) when we are given, as an additional input, the number of RAM steps, *maxstep*, that need to be simulated.

During *maxstep* steps, an R*RAM can invoke $RAND(Y_i)$ at most *maxstep* times, and the Y_i parameter used in each invocation is at most $\mathrm{EL}_{op}(maxstep)$.

The first step in the proof is to collate all calls to $RAND(Y_i)$ in the *maxstep*-step simulation into a single call. If we were able to know in advance the Y_i parameter used in each call, it would have been possible to make all calls into a single call whose parameter is the product of all Y parameters used. The individual random results can then be separated by applications of "$\div$" and "mod".

In fact, we do not know the parameters in advance, but can bound their product, $Y = \prod Y_i$, by $(\mathrm{EL}_{op}(maxstep))^{maxstep}$. Because this is clearly within $\mathrm{EL}_{op}(maxstep + \mathrm{O}(1))$, it is possible to generate a value, 2^k, that exceeds this limit.

Let us now continue with the result from the call to $RAND(2^k)$ as though it was a call to $RAND(Y)$. The extraction process of the individual X_i from X depends only on $X \bmod Y$. As such, the termination probabilities that it affords are the same as those of the R*RAM if we condition over $X < 2^k - (2^k \bmod Y)$. For higher X values, the RRAM will not accept the input if it is not in the language, and will accept the input with some probability if it is in the language.

It is not difficult to see that the worst-case for the total probability of acceptance for an input that is within the language is $p/(2-p)$ for the RRAM, if this probability is p for the R*RAM.[2] This worst-case is attained when 2^k is approximately $(2-p)Y$, and the accepting computations are those that work with $X \geq Y(1-p)$.

To meet with the acceptance criteria of the RRAM model, we simply run the simulation three times for each *maxstep* value. If in each of three independent runs the probability of acceptance is at least $p/(2-p)$, the probability of acceptance in at least one of the three is

[2]The total R*RAM algorithm must succeed with probability 0.5 or more, but this does not mean that the same is true for each one of the bounded-step simulations individually. This is the reason why a general parameter, p, is required.

at least

$$1 - \left(1 - \frac{p}{2-p}\right)^3.$$

A little calculus shows that this is never less than p for the entire range $0 \leq p \leq 1$, so the success rate of the simulating RRAM is always at least as good as that of the simulated R*RAM. □

Chapter 7

Conclusions

7.1 Summary

This work dealt with characterising the power of arbitrary and of random numbers when used as part of the RAM computational model. In the process, it uncovered hitherto unknown powers of the standard RAM model as well.

Table 7.1 presents a comparison of the power of the computational models investigated. For simplicity, it describes only the central models investigated, and measures these only in terms of their polynomial-time performance, as compared with the performance of Turing machines.

We also present similar information in graphical form in Figure 7.1.

op	P-RAM[op]	P-ARAM[op]	P-ASRAM[op]	RP-RAM[op]
$\{+, [\dot{-}], \leftarrow, [\rightarrow], Bool\}$	PSPACE	PSPACE	PSPACE	PEL
$\{+, [\dot{-}], \times, \leftarrow, [\rightarrow], Bool\}$	PSPACE	PSPACE	PSPACE	PEL
$\{+, [\dot{-}], [\times], /, \leftarrow, [\rightarrow], Bool\}$	PEL	$\Sigma_1^0 \cup \Pi_1^0 \subseteq \subseteq \Delta_2^0$	AH $\subset$	PEL

Table 7.1: Comparison summary of the power of RAM models

The table reveals the interesting fact that multiplication does not add computational power to any of the models, but division adds to both the standard RAM and to the ARAM and ASRAM models. (This lack of effect of multiplication is surprising in light of the power

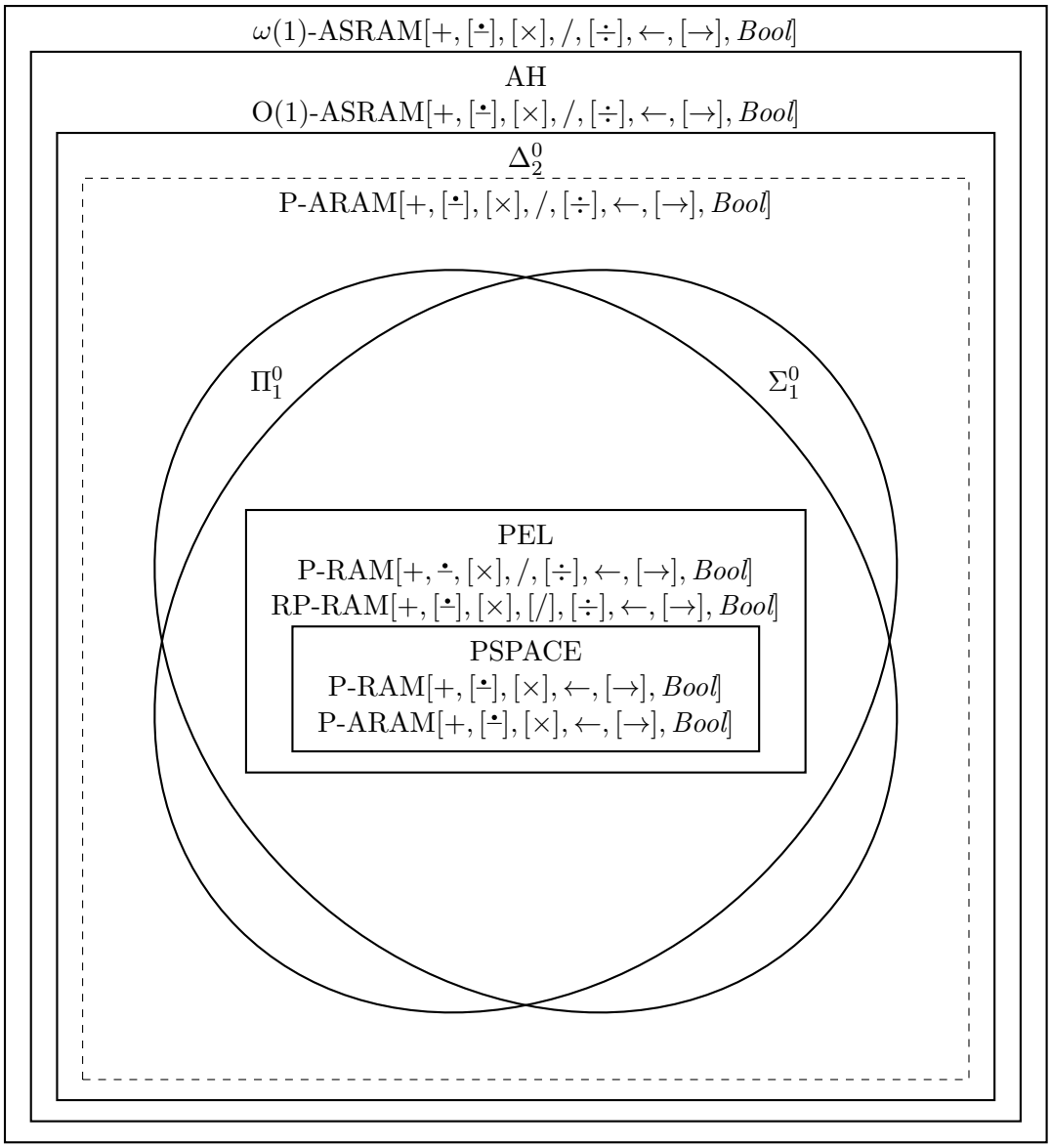


Figure 7.1: Graphical comparison of the power of RAM models

attributed to it by Trahan et al. (1992) and van Emde Boas (1990).) The benefit of stochasticity, available in RRAMs, provides exactly the same power as division. However, for ARAMs and ASRAMs the power boost afforded by division is far greater than for the other models. With division, an ARAM can recognise any recursively enumerable set in O(1) time, as well as at least some sets that are not, but are on the second level of the arithmetic hierarchy. This places the ARAM model as weaker than the RRAM model when division is not available, but stronger when it is. (While not quite this strong, the RAM and the RRAM can still attain much in O(1) time: they can accept any set that can be accepted by a TM in time/space that is *any elementary function* of the input length.)

The use of ALNs proves to not add any power when division is not present. However, not only is the ARAM with division far more powerful than the corresponding RAM, the ASRAM, allowing access to an unlimited number of ALNs, is even more powerful. Whereas the ARAM is only on the second level of the arithmetical hierarchy, the O(1)-time ASRAM conquers the entire arithmetical hierarchy, and the $\omega(1)$-time ASRAM is more powerful still.

Perhaps most interesting of all is the relationship between RAMs and RRAMs, and specifically the relationship between P-RAM and RP-RAM. One motivation for this work has been that investigating the $\mathrm{P} \stackrel{?}{=} \mathrm{RP}$ question in RAMs may give additional clues to the answer for this central question in the context of Turing machines. The table reveals that for models without division, P-RAM $\neq$ RP-RAM, but for models with division P-RAM = RP-RAM. Considering the general question of whether stochasticity adds computational power under RP acceptance criteria, not only does this result give a proven example for either direction, it demonstrates that the question is not one with a single, universal answer. Rather, the answer depends on the specifics of the computational model examined.

To the best of the author's knowledge, there is no other general-purpose computational model for which it is known that randomisation adds computational power in a P-vs.-RP-like comparison. (The results most comparable, in this respect, are those of Heller (1986), in the context of comparing relativised polynomial hierarchies with relativised RP hierarchies.)

For Turing machines, however, the P vs. RP question remains tantalisingly open. We remark regarding Turing machines only the following: all Turing machine complexity classes

described in Table 7.1 are ones for which randomisation is known *not* to add computational power under the TM model. The equivalence of PEL and R-PEL is given by Theorem 15. The equivalence of PSPACE and R-PSPACE is a corollary of PSPACE $\subseteq$ R-PSPACE $\subseteq$ N-PSPACE, in light of PSPACE = N-PSPACE from Savitch's theorem (Savitch, 1970).

To summarise, we reiterate the open problems solved in the present work and the new results innovating over or correcting older results.

1. Perhaps the most significant contribution of the present work is in quantifying the power of the RP-RAM model for a large number of operation sets, including all the operation sets studied in Simon (1979), where the question of the power of this model was left as an open question and has stayed open for 34 years.

2. The power of the RAM model over the same operation sets was also quantified. This corrected results that appeared in Simon (1979) and Simon (1981), as well as proofs that appeared in Simon and Szegedy (1992). These are oft-quoted papers that appeared in central venues and solved major problems, and yet the errors in them have not been spotted for several decades. (The newest of these results appeared in STOC 92.)

3. We improve on results by Ben-Amram and Galil (1995), where it was conjectured that their results are best possible in terms of simulating indirect addressing. We show that in the standard (not-online) RAM model, such simulation can be performed with no loss of time complexity. These new results are best possible.

4. In Appendix A, we improve over results of Paul and Simon (1982), which showed that sorting can be performed in O(n)-time in the RAM model. Our solution effectively reduces this to O(1)-time, making it, again, best possible.

5. In Appendix B, we settle, using the same tools as are used throughout the thesis, a 13-year-old open question regarding the density of Friedman numbers among the integers.

6. We introduce two new models of computation, the ARAM and the ASRAM, and quantify their powers. We show that without division these models are equivalent to a RAM, but with division they are far more powerful, with ASRAM being the more powerful

of the two. The O(1)-time ASRAM conquers the entire arithmetical hierarchy and the $\omega(1)$-time ASRAM goes even beyond that. This makes ASRAM complexities a useful tool in discussing computational power beyond the arithmetical hierarchy.

7. Furthermore, we show that ASRAMs are more powerful than ARAMs even in terms of the arithmetic complexities they afford to problems. This extends results by Bshouty et al. (1992), showing, effectively, that ARAMs are more powerful than RAMs in arithmetic complexity.

8. In total, the work examines four distinct computational models, each parameterised by its underlying operation set, and considers four distinct families of operation sets. (A typical result in this field handles a single computational model on a single family of operation sets.) This cross-model and cross-operation-set comparison allows us to make some across-the-board observations. The most salient of these is that, in stark contrast to division, multiplication never adds computational power when appended to an operation set that already includes left-shift. Surprisingly, right-shifting also never adds computational power to an operation set that already includes left-shift, regardless of whether division is present or not.

9. As a last achievement, we define a new computational complexity class, PEL, which seems to occur naturally in many of the models studied. This includes both P-RAMs with division and RP-RAMs. Why randomisation should provide exactly with the same power boost as division is an intriguing question.

7.2 Future work

There have been relatively few advances in the study of integer RAMs in recent years. It is our hope that the new results presented here will inspire follow-up work from others, by sending out the message that we have only begun to scratch the surface in our understanding of the workings of the RAM model.

In this section we present two follow-up directions, following directly from the results of the present work.

7.2.1 Arbitrariness and randomness together

The concept of the arbitrary number, presented here, is a new one and invites further research. Specifically, though we have studied it in isolation, it is unclear what its interactions are with other RAM features.

Consider the RP-ARAM. This is a RAM that has access to both arbitrary and random numbers. As shown in Table 7.1, the availability of random numbers allows, effectively, the simulation of a division function. We also know that arbitrary numbers with a division function allow any r.e. set to be recognised in O(1) time. However, we do not know whether in a model that contains both random and arbitrary numbers, but that does not contain division, the ability to simulate division with random numbers is enough to recognise any r.e. set in O(1) time. It may be that the two properties do not have this type of synergy, so the actual computational power of the model may be as low as PEL, with no power contribution from arbitrary numbers.

7.2.2 Arbitrariness and randomness in one input

In all models studied here, numbers are either purely arbitrary or purely random. Equivalently, they are chosen either by an adversary or by a neutral process. To this we can add Oracle numbers, which are chosen in a beneficial manner.

A different class of numbers would be ones which are, somehow, jointly picked by a mixture of these different types of processes. For example, one can consider a game-like setup, in which an Oracle, an adversary and a random process jointly formulate a single input integer.

The contribution of such integers would be interesting to study. These can be thought of, for example, as RAM analogues to standard studied complexity classes, such as those defined by interactive proofs (Goldwasser et al., 1989) (a combination of Oracle and adversarial inputs) or by Arthur-Merlin protocols (Babai, 1985) (a combination of Oracle and random inputs).

Appendix A

Constant time sorting

We present an algorithm for sorting an arbitrary number of arbitrary-length integers in constant time on a RAM_0. The algorithm forms a Straight Line Program. Unlike other parts of this work, the algorithm presented here requires registers to hold values whose bit-length is only polynomial in the bit-length of the input parameters. Without the restriction of polynomial bit-length, this result is a special case of Theorem 8.

A.1 Background

The task of sorting is a standard algorithmic benchmark. For sorting by comparisons, the lower bound of $\Omega(n \log n)$ time (Knuth, 1997; Cormen et al., 2001) is very well known, and is, in fact, one of the cornerstones of modern algorithm design. Integer sorting, on the other hand, is much less understood. It is studied predominantly in the context of models of computation where arithmetic operations on integers are assumed to be computable in O(1) time. Such models include, for example, the Straight Line Program (SLP), the Computation Tree (CT) and the Random Access Machine (RAM), but, notably, not the Turing machine.

Despite claims to the contrary (see e.g. Dittert and O'Donnell (1985), a claim that was retracted in Dittert and O'Donnell (1986)), the problem of giving a tight lower bound for the complexity of sorting under the operation set $\{+, \dot{-}, \times, \div\}$ remains open. However, certain related problems have been successfully tackled. For example, in Hong (1979), it is proved

that a RAM with these operations requires $\Omega(n \log n)$ time for sorting if it makes no use of indirect addressing, and in Paul and Simon (1982) it is shown that the $\Omega(n \log n)$ bound holds even with indirect addressing, but without division. (See Kirkpatrick and Reisch (1984) for a review of this topic, as well as Han (2004) and Hagerup (1998) for more modern approaches.)

On the other hand, using $\{+, \dot{-}, \times, \div, \leftarrow, \rightarrow, \mathit{Bool}\}$, Paul and Simon (1982) present a sorting algorithm in $\mathrm{O}(n)$ time.

We improve over this by presenting an algorithm running on a RAM without indirect addressing, over the operation set $\{+, \times, \div, \leftarrow, \mathit{Bool}\}$, that sorts n integers in $\mathrm{O}(1)$ time. Effectively, this is an SLP that sorts.

A.2 Input and output

One dilemma regarding the definition of the sorting problem under the RAM_0 model is that the customary methods for reading the input and writing the output of a RAM fail for the sorting problem when indirect addressing is unavailable. Typically, one would begin the program with, for example, $R[0]$ holding the value n, and $R[1], \ldots, R[n]$ storing the actual inputs. At program termination, the output could, similarly, be stored in positions $R[0], \ldots, R[n-1]$. However, without indirect addressing a RAM program necessarily uses only a fixed number of registers. It cannot read n as part of the input, and then read $R[n]$. The input and output must necessarily be encoded into a bounded number of registers.

To solve this problem, we encode inputs and outputs as vectors.

Definition 11 (Sorting)**.** In the context of RAM_0 machines, *sorting* is an operation that accepts an encoded vector (m, V, n) and returns an encoded vector with the same length and width parameters but with contents V', such that the elements of the returned vector are a permutation of the elements of the input vector, ordered in such a way that if the decoded V' is $[k_0, \ldots, k_{n-1}]$ then $\forall i, j : 0 \leq i < j < n \Rightarrow k_i \leq k_j$.

In the present context, we allow vectors to also be encoded as either (Z, V, n) or (m, Z, V, n), where $Z = 2^m$.

We claim:

Theorem 17. *Sorting* (m, V, n) *under* $RAM_0[+, \div, \times, \leftarrow, Bool]$ *can be performed in* $O(1)$ *time.*

In Chapter 3 it is shown that any RAM with the given set of operations can be stripped of its use of indirect addressing, given the vector input and output format, at no cost to the algorithm's time-complexity. This is done, essentially, by loading and storing register contents in a simulated way. In Section 3.2, where LOAD and STORE are described, they are implemented using, among others, the operators "$\rightarrow$" and "$\dot{-}$", which are not available to the RAM_0 machines in Theorem 17. However, in Section 2.5 we show that these can both be simulated given the available operations. Hence, each RAM instruction can be simulated in O(1) time.

A.3 Validity of the model

It is, of course, true that in *any* RAM model, regardless of its basic operation set, using the standard input and output formats, rather than vector notations, imposes an immediate $\Omega(n)$ lower bound on any sorting algorithm, simply due to the need to read n input register values and to write n output register values. As such, the new algorithm described here does not directly contradict the fact that, in the conventional sense, the algorithm described in Paul and Simon (1982) is best possible.

This raises the question of whether the switch to alternate input/output format is a valid one, or whether it should be taken as a "leg up" in the sorting process.

We claim that the $\Omega(n)$ is due *only* to the need to read inputs and write outputs, and not due to any part of the sorting itself.

To demonstrate this point, suppose that a RAM (maybe using indirect addressing) is programmed to accept n inputs and calculate n output values in the standard input/output format, but requires, for the calculation, to perform $f(n)$ sorts, each of n numbers (derived in some way from its inputs). Taking the algorithm of Paul and Simon (1982) as best possible would suggest that the algorithm must spend $\Omega(nf(n))$ on sorting, whereas the algorithm presented here gives a better (and best possible) $\mathrm{O}(n + f(n))$, which is attained as follows:

first, in O(n)-time, encode the input as a vector, then, in O($f(n)$)-time, perform $f(n)$ sorting computations, and lastly, in an additional O(n)-time, return the output from vector format to the standard format. The translation to and from vector format only needs to be performed once, because, as shown in Chapter 3, all operations can subsequently be performed within the vectorised storage format.

As such, this algorithm allows us to improve performance and bounds even in the unlimited RAM model using standard input/output formats. It is, therefore, not merely an artefact of the model used.

A.4 The algorithm

The algorithm is essentially divided into three parts. First, we "uniquify" the input, by which we mean that we create a new vector that has the same sorting order as the original vector but whose elements are unique. As a second step, we find the permutation that sorts the uniquified vector. Then, as a last step, we apply the permutation on the original vector.

Throughout, we do not work on vectors of the original width, m. Instead, we begin the algorithm by translating the original vector to a wider m_1, perform all actions on the wider vectors, then translate back to a width of m at the last step. The uniquification step requires $\lfloor \log n \rfloor$ bits to be added to the width and the rest of the algorithm requires one additional bit. Altogether, any $m_1 \geq m + \lfloor \log n \rfloor + 1$ would have sufficed. For convenience, we use $m_1 = m + n + 1$.

The top-level algorithm is described in Algorithm 15.

Algorithm 15 An O(1) sorting algorithm

1: $m_1 \Leftarrow m + n + 1$
2: $V_1 \Leftarrow \mathit{CHANGE\_WIDTH}(m, V, n, m_1)$
3: $V_2 \Leftarrow \mathit{UNIQUIFY}(V_1, n, m_1)$
4: $O \Leftarrow \mathit{FIND\_ORDER}(m_1, V_2, n)$
5: $V_3 \Leftarrow \mathit{MAKE\_ORDERED}(m_1, V_1, n, O)$
6: **return** $\mathit{CHANGE\_WIDTH}(m_1, V_3, n, m)$

Before detailing the subroutines used, we introduce some convenience functions.

First, recall that

$$\mathrm{RAM}_0[+, \div, \times, \leftarrow, \mathit{Bool}] = \mathrm{RAM}_0[+, \dot{-}, \div, \mathsf{mod}, \times, \leftarrow, \rightarrow, \mathit{Bool}, \leq].$$

This was demonstrated in Section 2.5.

Next, we define other useful functions that can be implemented in O(1) basic operations. Use of "/" instead of "÷" indicates that arithmetic division can be replaced by exact division because the numerator is known to be a multiple of the denominator.

- The integer composed of m 1s:

$$\mathit{MASK}(m) = (1 \leftarrow m) \dot{-} 1.$$

- The vector $[v, \ldots, v]$ of length n and width m:

$$\mathit{CONSTANT}(m, v, n) = v \times \mathit{MASK}(nm)/\mathit{MASK}(m).$$

- The counter $[0, 1, \ldots, n-1]$ of width m:

$$\begin{aligned}\mathit{COUNTER}(n, m) =&((n \dot{-} 1) \leftarrow (mn) \dot{-} (1 \leftarrow (mn) \dot{-} 1 \leftarrow (m))/\mathit{MASK}(m))\\ &/\mathit{MASK}(m).\end{aligned}$$

- "Greater than" comparison between the elements of two vectors (where, if an element in the first operand is greater than the corresponding element in the second operand this is indicated by a 1 in the result vector, and 0 otherwise):

$$\begin{aligned}\mathit{GT}(m, V_1, V_2, n) =&((V_1 + \mathit{CONSTANT}(m, \mathit{MASK}(m \dot{-} 1), n) \dot{-} V_2) \rightarrow (m \dot{-} 1))\\ &\wedge \mathit{CONSTANT}(m, 1, n).\end{aligned}$$

- Equality check between the elements of two vectors (using the same convention of 1 for "true" and 0 for "false" as above):

$$EQ(m, V_1, V_2, n) = CONSTANT(m, 1, n) \dot{-} GT(m, V_1, V_2, n) \dot{-} GT(m, V_2, V_1, n).$$

- A simple maximum between two integers:

$$MAX(a, b) = (a \dot{-} b) + b$$

We remark that the "if $a \diamond b$ then X else Y" constructs used both here and elsewhere in the construction can be implemented as

$$X \times (a \diamond b) + Y \times (1 \dot{-} (a \diamond b)),$$

where $(a \diamond b)$ is the assignment operator that calculates the indicator function for the comparator $a \diamond b$.

These basic functions allow defining the highly-useful width-change function. (This function was already defined in Lemma 3, but we repeat here, briefly, in order to present all definitions in a consistent format.)

We begin by describing two "easy" cases: where the target width is known to be sufficiently large (i.e., $m_1 \geq m(n+1)$), and where the target width is known to be sufficiently small (i.e., $m_1 n < m$).

- Changing the width of a vector to a sufficiently large new width:

$$MAKE\_WIDE(m, V, n, m_1) = CONSTANT(m_1 \dot{-} m, V, n) \wedge CONSTANT(m_1, MASK(m), n).$$

- Changing the width of a vector to a sufficiently narrow new width:

$$MAKE\_NARROW(m, V, n, m_1) = V \bmod (MASK(m) \dot{-} MASK(m_1)).$$

Using *MAKE_WIDE* and *MAKE_NARROW*, the width change function can be implemented simply by case-based handling. The code for this function is given in Algorithm 16.

Algorithm 16 *CHANGE_WIDTH*(m, V, n, m_1)

```
1: if m_1 ≥ m × (n + 1) then
2:     return MAKE_WIDE(m, V, n, m_1)
3: else if m_1 × n < m then
4:     return MAKE_NARROW(m, V, n, m_1)
5: else
6:     l ⇐ MAX(m × (n + 1), m_1 × n + 1)
7:     return MAKE_NARROW(l, MAKE_WIDE(m, V, n, l), n, m_1)
8: end if
```

The following two operations exemplify the power of width change.

- Repeat the vector k times:

$$REPEAT(m, V, n, k) = CONSTANT(mn, V, k).$$

- Repeat each element of the vector k times:

$$\begin{aligned} STRETCH(m, V, n, k) =& CHANGE\_WIDTH(m, V, n, mk) \\ & \times CONSTANT(m, 1, k). \end{aligned}$$

REPEAT and *STRETCH* now allow us to define the two main procedures of the sorting algorithm. These are *FIND_ORDER*(m, V, n), a procedure for finding the sorting permutation for a vector of distinct elements and *MAKE_ORDERED*(m, V, n, O), a procedure for applying a permutation to a vector's elements. These are described in Algorithm 17 and Algorithm 18, respectively.

Algorithm 17 *FIND_ORDER*(m, V, n), find the sorting permutation

1: $V_r \Leftarrow REPEAT(m, V, n, n)$
2: $V_s \Leftarrow STRETCH(m, V, n, n)$
3: $C \Leftarrow GT(m, V_r, V_s, n \times n)$
4: **return** C mod $MASK(n \times m)$

Algorithm 18 *MAKE_ORDERED*(m, V, n, O), applying a permutation

1: $V_s \Leftarrow STRETCH(m, V, n, n)$
2: $O_s \Leftarrow STRETCH(m, O, n, n)$
3: $C_r \Leftarrow REPEAT(m, COUNTER(n, m), n, n)$
4: $E \Leftarrow EQ(m, O_s, C_r, n \times n)$
5: **return** $(V_s \wedge (E \times MASK(m)))$ mod $MASK(m \times n)$

The last remaining function is the uniquification function. Uniquification is performed by inserting a counter into the LSBs of the input, as follows.

$$UNIQUIFY(V_1, n, m_1) = V_2 \Leftarrow (V_1 \leftarrow n) + COUNTER(n, m_1).$$

Here, shifting left by n indicates that n bits have been allocated for the counter. In fact, fewer bits would have sufficed.

A.5 Additional results

Algorithm 15 provides a constructive proof for Theorem 17. For completion, this section provides an analysis of the complexity of sorting when left shifting, arguably the most powerful of the operators used above, is omitted from the model.

Corollary 17.1. *Sorting* (Z, V, n) *under* $RAM_0[+, \times, \div, Bool]$ *is a* $\Theta(\log n)$*-time problem.*

Proof. We begin by constructing an $O(\log n)$-time sorting algorithm.

The proof of Theorem 17 requires handling of vectors of two widths: m (the original width, $2^m = Z$) and m_1 (the width after the addition of an MSB and a counter, for uniquification). In order to calculate $Z' = 2^{m_1}$, we need to find a number, $Z'' = 2^k$, s.t. $Z'' \geq n$. The conventional way of doing this is by repeated squaring, which would take $O(\log \log n)$ time. However, as previously shown, $SET(n)$ provides an $O(1)$-time solution to the same problem.

After the width-change step, all shifts required are by m_1, $m_1 n$ or $m_1 n^2$. Shifting by m_1 is equivalent to multiplication by Z'. Shifting by $m_1 n$ is equivalent to multiplication by Z'^n and shifting by $m_1 n^2$ is equivalent to multiplication by Z'^{n^2}. Both Z'^n and Z'^{n^2} are numbers that can be calculated in $\mathrm{O}(\log n)$ time using repeated squaring and multiplications.

The total cost of the sort is $\mathrm{O}(\log n)$ time.

To see that this complexity is best possible, consider sorting the input

$$(Z, V, n) = (2^m, 1, 2^m).$$

The decoded vector is, in this case, $[1, 0, \ldots, 0]$, whereas the decoded output vector should be $[0, \ldots, 0, 1]$, leading to a V' value on the order of n^n.

The fastest growing sequence that can be computed in the given RAM model is the one computed by iterative squaring, beginning with n. Even so, reaching n^n, the desired order of magnitude for V', requires $\Omega(m) = \Omega(\log n)$ steps. □

Appendix B

Friedman numbers

A Friedman number is a positive integer which is the result of an expression combining all of its own digits by use of the four basic operations, exponentiation and digit concatenation. One of the fundamental questions regarding Friedman numbers, first raised by Erich Friedman in August 2000, is how common they are among the integers. In this appendix, we prove that Friedman numbers have density 1. We further prove that the density of Friedman numbers remains 1 regardless of the base of representation, and show that the density of "nice" Friedman numbers is also 1, when considered in binary, ternary or base four.

Despite the fact that RAMs and Friedman numbers seem to have little in common, the operation sets considered in the thesis are, in fact, very similar to those allowed in Friedman number computations: arithmetic functions are allowed in both, left shifting can be viewed as a combination of multiplication and restricted exponentiation, and bitwise Boolean operations in RAMs are replaced in Friedman numbers by digit operations. As such, it is not surprising that tools developed in this thesis and used throughout it, such as, for example, $O^n_{a,m}$ vectors, find their uses also in the study of Friedman numbers.

As such, the study of RAMs has inadvertently provided the tools needed to close here a number theoretical question that has been open for 13 years.

B.1 About Friedman numbers

Friedman numbers are numbers that can be computed from their own digits, each digit used exactly once, by use of the four basic arithmetic operations, exponentiation and digit concatenation (as long as digit concatenation is not the only operation used). Parentheses can be used at will. An example of a Friedman number is 25, which can be represented as 5^2. An example of a non-Friedman number is any power of 10, because no power of 10 can be expressed as the result of a computation using only arithmetic operations and exponentiation if the initial arguments in the computation are a smaller power of 10 and several zeros.

Friedman numbers are sequence A036057 of the Encyclopedia of Integer Sequences (Friedman, n.d.). They were first introduced by Erich Friedman in August 2000 (Friedman, 2000), and the first question to be asked about them was what their density is, inside the population of integers. That is, if $F(n)$ is the number of Friedman numbers in the range $[1, n]$, what is $\lim_{n\to\infty} F(n)/n$?

This question has remained open for the decade that has passed since then. However, much progress was made on other aspects of the problem. The following review summarises some of these advances, as detailed in Friedman (2000).

Mike Reid, Ulrich Schimke and Philippe Fondanaiche calculated the exhaustive list of all Friedman numbers up to 10,000 (there are 72 of them), and these were later supplemented by Erich Friedman to an exhaustive list that ranges until 100,000. In total, there are 842 Friedman numbers smaller than 100,000, or 0.842%. This is not much higher than the 0.72% among the first 10,000 numbers, and does not suggest the density 1 proved here.

Ron Kaminsky proved the existence of infinitely many prime Friedman numbers. Brendan Owen and Mike Reid showed (independently) that any string of digits can be the prefix to a Friedman number by appending to it a fixed suffix. (For example, 12588304 forms one such suffix. Any number N followed by the digits 12588304 is a Friedman number and can be calculated from its own digits as $N \times 10^8 + 3548^2$.) Erich Friedman augmented this result by showing that any string of digits can be the suffix to a Friedman number by preceding it with a prefix that is dependent only on the length of the suffix. (For example, if XY is a two-digit

number, then $2500XY = 500^2 + XY$ is a Friedman number. More generally, 25×10^k is a prefix that can precede any k-digit suffix to make a Friedman number.)

While these results, and many like them, were able to show that the density of Friedman numbers is greater than 0, only less than one percent of the integers were previously known to be Friedman numbers, whereas in this appendix we prove that the density of Friedman numbers within the integers is 1.

B.2 The basic techniques

The proof makes use of two basic techniques, which we now describe.

B.2.1 Encoding

The first building-block used in the proof is the ability to encode tuples of integers into a single integer in a way that later allows unambiguous decoding.

For the present proof, we do not require the tuple to be composed of arbitrary integers. A sufficiently rich set of integers is enough. However, it is important that the encoding be compact, in the sense that computing the encoded integer from the tuple being encoded should require minimal use of extraneous digits not originally from the tuple. The encoding chosen for the proof requires no extraneous digits in the computation. However, it does require that the tuple be composed only of radical-free integers, that is, of integers greater than one that cannot be represented as a nontrivial integer power (an integer taken to an integer power greater than one).

Specifically, the encoding used for $\{x_i\}_{i=1}^t$, where all x_i are radical-free integers (not necessarily distinct), is by the number $\mathrm{enc}\big(\{x_i\}_{i=1}^t\big) = x_1^{x_2^{x_3^{\cdots}}}$. It can easily be seen that $\mathrm{enc}\big(\{x_i\}_{i=1}^t\big)$ contains all information necessary to unambiguously reconstruct the entire tuple (including its length, t).

B.2.2 Self-description by repetition

Friedman numbers use their own digits to describe themselves. A second building block used in the proof is self-description by repetition. An example for a Friedman number that uses repetition is $10411041 = 1041 \times (10^4 + 1)$. One can think of one repetition ("1041") as containing data to be copied, and the other repetition ("$\times(10^4+1)$") as containing instructions regarding how to manipulate the data to form the repetitions of the original number.

In this appendix, we require a process that generates a large number of repetitions from the original data. If the data is s and the base of representation is 10, this can be done by

$$\frac{10^{L(s)r} - 1}{10^{L(s)} - 1} s,$$

where $L(s)$ is the digit length of s and r is the number of desired repetitions. More generally, in base b we can use

$$\frac{b^{L(s)r} - 1}{b^{L(s)} - 1} s.$$

B.3 Outline of the main proof

We wish to prove the following claim:

Theorem 18. *If $F(n)$ is the number of Friedman numbers in the range $[1, n]$, then*

$$\lim_{n\to\infty} F(n)/n = 1.$$

Let us first introduce some notation.

Let x and y be integers. We use the notation $L(x)$ to denote the digit length of x, $x.y$ to be the concatenation of the digits of x and of y, and $[x]^k$ to be the concatenation of k copies of x.

The proof itself can be divided into three steps.

In the first step, we combine the results of Owen and Reid with the results of Friedman. Owen and Reid showed that there exists a number m such that for all b, $b.m$ is a Friedman number. That is, they found a suffix that, when appended to any string of digits, creates

a Friedman number. Friedman, on the other hand, showed that there exists a number m and a number k such that the number $m \times 10^k + b$ is a Friedman number for any b satisfying $L(b) \leq k$. Here, m is not a suffix, but rather a prefix placed at a specific digit position. We say that m is placed here "at position k". Friedman furthermore showed that such (m, k)-pairs can be found with arbitrarily large k.

We extend on these results by showing pairs of (m, k) values where m is not necessarily a prefix or a suffix, but is rather an infix. Specifically, we claim the existence of (m, k)-pairs such that any number, n, is a Friedman number if the substring of n of length $L(m)$ starting at digit-position k of n is m. (That is, if m is a substring of n such that the least significant digit of m occupies the k'th least significant digit of n, then n is a Friedman number.)

In the second step, we show how for a given infix length $l = L(m)$ we can construct a large number of such (m, k)-pairs with distinct k values. In fact, for each value of l we utilise only a single m, and even so we demonstrate that the number of (m, k)-pairs increases exponentially with l.

In the third step, we show that the rate at which the number of (m, k)-pairs increases with l is sufficient to prove that $F(n)/n$ converges to 1, completing the proof of the theorem.

Once completing the main proof, that base-10 Friedman numbers have density 1, we augment the proof for use with the Friedman numbers of any other base of representation.

In the last section, we modify the main proof for use with a specific subset of the Friedman numbers known as the nice (or "orderly") Friedman numbers. The same proof shows that this subset is already of density 1. However, this extended proof is base-dependent, and works only for bases 2, 3 and 4.

B.4 Constructing a basic (m, k)-pair

Let the **span** of a nonnegative integer, x, be the set of nonnegative integers that can be produced from x's digits by use of the four basic arithmetic operations, exponentiation and digit concatenation. (It is tempting, at this point, to redefine Friedman numbers as the set of integers, x, for which $x \in \text{span}(x)$. However, such a redefinition would not be correct:

span(x) includes x for every x, because x can be produced from itself by concatenating all of its own digits in order. In the definition of Friedman numbers, such concatenation-only calculations are disallowed explicitly.)

Theorem 19. *There exist numbers m and k, such that for any a and any $b < 10^k$, $x = a.m.[0]^t.b$ is a Friedman number if $t = k - L(b)$. Moreover, for any $s > 2$ there exists a C, such that for any sufficiently large r', the number $m = [s]^{r's}$, in combination with any $k \in \text{span}\left([s]^{(s-2)r'-C}\right)$, forms an (m,k)-pair that satisfies this condition.*

We name such an (m,k)-pair an "infix", and $L(m)$ the "infix length". The reason that we are looking at infixes that are of the form $m = [s]^r$ is that, as discussed above, repetition provides us with a general tool to induce self description.

Consider the span of such repetitions:

Lemma 13. *For any positive integer s and any nonnegative i, there exists a number, c_i, such that $i \in \text{span}([s]^{c_i})$.*

Proof. If $i = 0$, $i = s - s$ provides a solution with $c_i = 2$. For positive values of i, one can choose $c_i = 2i$ by representing i as

$$\underbrace{\frac{s}{s} + \cdots + \frac{s}{s}}_{i \text{ times}}.$$

This is not necessarily the smallest possible c_i for any particular i. □

With this lemma, it is possible to prove Theorem 19 as follows:

Proof. Let s be some constant value, $s > 2$. The numbers $0, 1, 10$ and $L(s)$ are all constants, and therefore, by Lemma 13, there exist c_i values c_0, c_1, c_{10} and $c_{L(s)}$ for which $i \in \text{span}([s]^{c_i})$.

Let $r = r's$. For such a value of r, $r \in \text{span}\left([s]^{r'}\right)$ holds, because r can be represented as the summation of r' copies of s: $r = \underbrace{s + \cdots + s}_{r' \text{ times}}$.

Let c_k be such that $k \in \text{span}([s]^{c_k})$.

We can now use the numerical technique for self-description introduced in Section B.2.2, to create an infix containing r copies of s. This is done with the calculation

$$\left(a \times (0+10)^{L(s)r} + \frac{10^{L(s)r}-1}{10^{L(s)}-1}s\right) \times 10^k + \underbrace{0+\cdots+0}_{t \text{ times}} + b,$$

which can be computed by use of the digits of a, of b, of t zeros and by use of $c_0 + 2c_1 + 4c_{10} + 3c_{L(s)} + 1 + 2r' + c_k$ copies of s. If $c_0 + 2c_1 + 4c_{10} + 3c_{L(s)} + 1 + 2r' + c_k = r$, then this computation yields x, proving that x is a Friedman number.

The reason we require the form $(0+10)^{L(s)r}$ in the expression is in order to enable its replacement with $0 \times 10^{L(s)r}$ for the case $a = 0$.

Let $C = c_0 + 2c_1 + 4c_{10} + 3c_{L(s)} + 1$. To prove that x is a Friedman number, we want to choose c_k such that $C + 2r' + c_k = r$. Utilising the fact that $r = r's$, we can rearrange the equation to get $c_k = (s-2)r' - C$. For any value of r' greater than $\frac{C}{s-2}$, any choice of $k \in \text{span}\left([s]^{(s-2)r'-C}\right)$ yields a Friedman number. □

B.5 Finding many Friedman numbers with a given infix length

So far, we have considered the ability to generate a number from a number, but, technically, what we needed was to generate, for example, from $[s]^r$ three copies of $L(s)$, four copies of 10, etc.. It is therefore natural to also consider the ability to generate a tuple of numbers from a number. We say that $(L(s), L(s), L(s), 10, 10, \ldots) \in \text{tuplespan}([s]^r)$.

Formally, let the **tuplespan** of an integer, s, be the set of tuples $(x_1, \ldots, x_n)$ such that $\forall i : 1 \leq i \leq n, x_i \in \text{span}(s_i)$, where $\{s_i\}_{i=1}^n$ are numbers that result from a partitioning of the digits of s.

Let $N(s)$ be the size of the largest subset of tuplespan(s) in which each tuple is composed only of radical-free numbers and no two tuples are prefixes of each other, in the sense that if $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$ are both elements of the subset, with $n < m$, then $\exists i : 1 \leq i \leq n$, such that $x_i \neq y_i$. We say, borrowing terminology from coding theory, that the subset is a "prefix code".

Theorem 20. *If $m = [s]^r$, for some constant $s > 3$, then the number of k values that are divisible by $L(m)$ for which (m, k) is a pair that forms an infix for a Friedman number is in $\Omega(g^r)$, where $g = N(s)^{\frac{s-3}{s}}$.*

Proof. Theorem 19 shows that any $k \in \text{span}\Big([s]^{(s-2)r'-C}\Big)$ yields a Friedman number for $m = [s]^{r's}$. Here, we restrict ourselves to k values of the form $k'l$, where l, the infix length, is $l = L(m) = L(s)r's$. The value $L(s)$ can be computed by $c_{L(s)}$ copies of s and $r's$ can be computed using r' copies of s, so $l \in \text{span}\Big([s]^{r'+c_{L(s)}}\Big)$.

Accordingly, choosing $k' \in \text{span}\Big([s]^{(s-3)r'-C-c_{L(s)}}\Big)$ yields a Friedman number. For convenience, we define $C' = C + c_{L(s)}$. Our aim is to prove that

$$\lim_{r\to\infty} \left|\text{span}\Big([s]^{(s-3)r'-C'}\Big)\right| g^{-r} > 0.$$

To show this, consider once again the tuple-encoding procedure enc$(\cdot)$ introduced in Section B.2.1. It ensures that any tuple of radical-free numbers can be encoded uniquely. Specifically, tuplespan(s) includes $N(s)$ distinct tuples composed of radical-free integers that form a prefix code, and consequently tuplespan$\Big([s]^{(s-3)r'-C'}\Big)$ includes the concatenation of any $(s-3)r'-C'$ of these tuples, creating a total of $N(s)^{(s-3)r'-C'}$ tuples, all of which are distinct due to the prefix-code constraint. Each of these tuples can now be encoded by enc$(\cdot)$ to form a distinct value of k.

The total number of possible k values is therefore at least

$$N(s)^{(s-3)r'-C'} \propto N(s)^{(s-3)r'} = g^{r's} = g^r,$$

proving the claim.

This completes the proof for $m = [s]^{r's}$. If r is not a multiple of s, the number of possible k values is at least as many as there are for $m' = [s]^{\lfloor r/s \rfloor s}$, because m' is a substring of m. This bounds the number of possible k values at

$$N(s)^{(s-3)\lfloor r/s \rfloor - C'} > N(s)^{(s-3)\left(\frac{r}{s}-1\right)-C'},$$

which is still in $\Omega(g^r)$. □

B.6 Bounding the density of Friedman numbers

We return now to proving Theorem 18, for which we require an additional lemma.

Lemma 14. *There exists a value of s, for which $g = N(s)^{\frac{s-3}{s}} > 10^{L(s)}$.*

Proof. There are 5 radical-free integers that are 1 digit long, 82 radical-free integers that are 2 digits long and 872 radical-free integers that are 3 digits long. Let us define the constant s to be the digit concatenation of 13 copies of each of these radical-free integers. The digit length of s, $L(s)$, is $2785 \times 13 = 36205$.

The specific s we have chosen is composed of 13 copies of each of the 959 radical-free integers below 1000. If we partition s into these radical-free integers and form all tuples of length 959 that can be created by distinct permutations of these numbers, then we have $G = (959 \times 13)!/13!^{959}$ distinct tuples. This number is a lower bound for $N(s)$, because the tuples, being of constant length, necessarily form a prefix code.

Either direct calculation or use of Stirling's formula can be used to show that G is a value with 36258 digits, whereas $10^{L(s)\frac{s}{s-3}}$ has only $L(s) + 1 = 36206$ digits. Therefore, $G > 10^{L(s)\frac{s}{s-3}}$, and $g \geq G^{\frac{s-3}{s}} \Rightarrow g > 10^{L(s)}$. □

As a side note, for this particular s it is possible to choose $c_0 = c_1 = c_{10} = c_{L(s)} = 1$, and even more complicated expressions such as $10^{L(s)} - 1$ are still within the span of a single copy of s. This should come as no surprise, because s contains over 2000 copies of each of the ten digits. However, the present proof does not rely on the specific value of any c_i.

With Lemma 14 it is possible to prove Theorem 18 as follows:

Proof. In order to bound the density of Friedman numbers within the integers, consider all (m, k)-pairs constructable using a specific infix length $l = L(s)r$, where $k = k'l$, and let K be the maximum value of k among them. Furthermore, let M be a number greater or equal to $l + K$, and let x be an integer chosen randomly and uniformly in $[0, 10^M)$. We wish to bound

from above the probability that x is **not** any of the Friedman numbers corresponding to any of the (m, k)-pairs in the set.

For any given (m, k), this probability is $1 - 10^{-l}$, because exactly l digits are restricted to a specific value. However, the probabilities relating to any two (m, k)-pairs with the same l are independent, because the restricted digits do not overlap (hence our requirement that k be divisible by l). This means that the total probability is bounded by $(1 - 10^{-l})^{C_0 g^r}$, where C_0 is the multiplicative constant in the $\Omega(g^r)$ bound from Theorem 20.

Our goal is to prove that this probability drops to zero as r tends to infinity. Equivalently, we wish to prove that $10^{-l} C_0 g^r = C_0 \left(g/10^{L(s)}\right)^r$ tends to infinity with r. However, Lemma 14 already showed that $g > 10^{L(s)}$.

This proves that as n increases, the probability that an integer uniformly sampled in $[1, n]$ is a Friedman number approaches 1. However, we have only showed this for some sub-sequence of the integers, n, namely those describable as $n = 10^M - 1$. Within this sub-sequence, the density $F(n)/n$ tends to 1. Formally, this means that $\limsup_{n\to\infty} F(n)/n = 1$. To prove that $\lim F(n)/n = 1$ we need to show that for any density $\rho' < 1$ there is an N, such that $n \geq N \Rightarrow F(n)/n \geq \rho'$.

Consider, however, what has been demonstrated up to this point. The proof so far shows that for any density $\rho < 1$ (given above as $(1-10^{-l})^{C_0 g^r}$) there is a number m (given above as 10^{l+K}) such that there exists a set S, of size at least ρm, of numbers in the range $0 \leq x < m$, such that for any number, n, if $n \bmod m$ is in S, then n is a Friedman number.

Now, choose $\rho = \sqrt{\rho'}$, let k be a number greater than $\rho/(1-\rho)$ and choose $N = mk$. If $n \geq N$, $n = mk' + r$ with $0 \leq r < m$ and $k' \geq k$. By the claim proved, $F(n) \geq \rho m k'$, so

$$\frac{F(n)}{n} \geq \frac{\rho m k'}{n} \geq \frac{\rho m k'}{m(k'+1)} \geq \rho \frac{k'}{k'+1} \geq \rho \frac{k}{k+1} \geq \rho^2 = \rho'.$$

Therefore, the limit exists and equals 1. □

B.7 Other bases of representation

The operations defining Friedman numbers are operations on digits, and hence the definition of a Friedman number is dependent on the base of representation. What is a Friedman number in decimal notation may not be a Friedman number in binary, and vice versa. This may be one reason why Friedman numbers have not been studied much within the context of number theory.

In this section, we show that the proof given above can be augmented to work in any representation base.

Theorem 21. *For any b, if $F(n, b)$ is the number of base-b Friedman numbers in the range $[1, n]$, then $\lim_{n\to\infty} F(n, b)/n = 1$.*

The base 10 proof relies on a specific choice of s: s was chosen to be the concatenation of 13 copies of the radical-free integers up to 3 digits long. In base b we replace the maximum number of digits by a variable, t, and the number of copies by a variable, d. We show that for any b, suitable t and d can be found.

First, we note that the constants needed (0, 1, b and $L(s)$) can always be calculated using enough copies of s, and that the proof is not reliant on the exact number of any of these c_i. Any t and d will do here.

Choosing t and d arbitrarily allows, in fact, most of the base-10 proof to remain intact. The only difficulty is in finding a (t, d)-pair that satisfies the final inequality:

$$g > b^{L(s)}.$$

Let f_i be the number of radical-free integers with i digits, and let $f = \sum_{i=1}^{t} f_i$. As a reminder, the total number of radical-free integers in s is fd.

We begin with a lemma:

Lemma 15. *There exists a value of t for which $f \log_b f > \sum_{i=1}^{t} i f_i$.*

Proof. For $x > 1$, the number of integers in the range $(1, x]$ that are not radical-free is at most

$$\sum_{\substack{p \le \log_2 x \\ p \text{ prime}}} \lfloor \sqrt[p]{x} - 1 \rfloor.$$

The number of summands can be bounded by $\log_2 x$ and the largest one among them is smaller than $\sqrt{x}$. Therefore, the total number of integers in the range $(1, x]$ that are not radical-free is smaller than $\sqrt{x} \log_2 x$. The number of radical-free integers is at least $x - \sqrt{x} \log_2 x$.

If we take x to be b^t, we see that the number, f, of radical-free integers with up to t digits satisfies $f > b^t - b^{t/2} t \log_2 b$ (noting that b^t itself is not radical-free).

We claim:

$$\lim_{t\to\infty} \left(t - \frac{f \log_b f}{b^t} \right) = 0 < \frac{1}{b} \le \lim_{t\to\infty} \left(t - \frac{\sum_{i=1}^{t} i f_i}{b^t} \right),$$

A direct corollary of which is that for a sufficiently large t the inequality of Lemma 15 holds.

The first equality in the claim is given by

$$\lim_{t\to\infty} \left(t - \frac{f \log_b f}{b^t} \right) \le \lim_{t\to\infty} \left(t - \frac{(b^t - b^{\frac{t}{2}} t \log_2 b) \log_b (b^t - b^{\frac{t}{2}} t \log_2 b)}{b^t} \right)$$

$$= \lim_{t\to\infty} \left(t - (1 - b^{-\frac{t}{2}} t \log_2 b)(t + \log_b (1 - b^{-\frac{t}{2}} t \log_2 b)) \right)$$

$$= \lim_{t\to\infty} \left(b^{-\frac{t}{2}} t^2 \log_2 b + b^{-\frac{t}{2}} t \log_2 b \log_b (1 - b^{-\frac{t}{2}} t \log_2 b) - \log_b (1 - b^{-\frac{t}{2}} t \log_2 b) \right)$$

$$= \lim_{t\to\infty} \left(- \log_b (1 - b^{-\frac{t}{2}} t \log_2 b) \right) = 0,$$

whereas $t - \frac{f \log_b f}{b^t} \ge 0$ is guaranteed by $f \le b^t$, the total number of integers up to b^t.

On the other hand, f_t cannot be larger than $b^{t-1}(b-1)$ because this is the total number of t-digit numbers, whereas the rest of the f_i together cannot accumulate to more than b^{t-1} for the same reason. Therefore:

$$\lim_{t\to\infty} \left(t - \frac{\sum_{i=1}^{t} i f_i}{b^t} \right) \ge \lim_{t\to\infty} \left(t - \frac{(t-1) b^{t-1} + t b^{t-1} (b-1)}{b^t} \right)$$

$$= \lim_{t\to\infty} \left(t - \frac{tb^t - b^{t-1}}{b^t} \right) = \frac{1}{b}.$$

□

We claim that an $s = s(t, d)$ calculated with the value of t as provided by Lemma 15 and a sufficiently large d satisfies $\log_b g > L(s)$.

Specifically, we claim

Lemma 16. *For an $s = s(t, d)$ calculated with a value of t satisfying $f \log_b f > \sum_{i=1}^{t} i f_i$,*

$$\lim_{d\to\infty} \frac{\log_b g - L(s)}{d} > 0.$$

Proof. g can be bounded from below by $G^{\frac{s-3}{s}}$, where

$$G = \frac{(fd)!}{d!^f}.$$

Substituting Stirling's formula

$$\lim_{d\to\infty} \sqrt{2\pi d} \left(\frac{d}{e} \right)^d (d!)^{-1} = 1$$

we get

$$\lim_{d\to\infty} (2\pi d)^{-(f-1)/2} f^{df+1/2} G^{-1} = 1$$

and

$$\lim_{d\to\infty} \log_b G - (df + 1/2) \log_b f + \frac{f-1}{2} \log_b(2\pi d) = 0.$$

Recall that by definition $L(s) = d \sum_{i=1}^{t} i f_i$. Substituting this in, we get

$$\lim_{d\to\infty} \frac{\log_b g - L(s)}{d} \geq \lim_{d\to\infty} \frac{\frac{s-3}{s} \log_b G - L(s)}{d}$$

$$= \lim_{d\to\infty} \frac{s-3}{s} f \log_b f - \sum_{i=1}^{t} i f_i = f \log_b f - \sum_{i=1}^{t} i f_i > 0.$$

□

Using the values of d and t guaranteed by the two lemmas to exist, we are now able to construct a value of s that satisfies the inequality $\log_b g > L(s)$, completing the proof of Theorem 21, and demonstrating that the Friedman numbers of any base b have density 1.

B.8 Nice Friedman numbers

Several interesting subsets of Friedman numbers have been defined since the introduction of Friedman numbers. For example, there are several conflicting definitions in the literature (see e.g. Wilson, 2009; Weisstein, 2009) for the term "vampire numbers", initially introduced by Pickover (1995). By one definition, these are Friedman numbers that make no use of exponentiation.

Another interesting subset, introduced by Mike Reid, is the "nice" Friedman numbers (sometimes referred to as "orderly" or "good" Friedman numbers) (Rattner, 2003). These are the Friedman numbers that can be calculated from their own digits without changing the digit order.

It is not known, in general, what the density of either of these subsets is. Specifically, the method of the "infix" introduced here cannot be used in these other cases. For vampire numbers, the infix method fails because infixes explicitly make use of exponentiation. For nice Friedman numbers, infixes are constructable using the same method as shown above. However, not for every base of representation, b, is it true that there exists a value of s, for which Lemma 14 holds, once restricting $\mathrm{span}(s)$ to include only computations that do not change the order of the digits of s. Where such an s can be found, the previous proof remains intact for nice Friedman numbers, and shows that their density is 1.

Let us define $\mathrm{span}'(s)$ similarly to $\mathrm{span}(s)$, but for $\mathrm{span}'(s)$ to include only those elements of the original span that can be calculated without changing the digit order of s, and let us also define $\mathrm{tuplespan}'(\cdot)$ and $N'(\cdot)$ accordingly.

The limitations of the infix method can be expressed as follows:

Theorem 22. *In any base of representation $b \geq 28$, every s satisfies $b^{L(s)} > |\,\mathrm{span}'(s)\,|$.*

This means that even though infixes can be constructed for these bases, there are too few of them to establish a density of 1 by the same means as in the proof to Theorem 21.

Proof. span′(*s*) is the set of numbers that can be produced by a calculation from the digits of *s* while retaining the order of *s*'s digits. The size of span′(*s*) can be bounded from above by the size of the set of strings that describe these calculations. Each character in the strings is a base-*b* digit, one of 5 permitted operations or an opening/closing parenthesis.

In addition to the 5 binary operations we must also consider unary negation. This adds a sixth operation: x^{-y}. In all other operations, when used in conjunction with unary negation, we can eliminate the unary operation by pushing it forward in the string until it is voided or until it reaches the beginning of the string, via operations such as "$x \times -y$" $\mapsto$ "$-(x \times y)$". After applying these normalisation procedures all inter-digit operations are one of the six binary operations described, and if there remains a unary negation it must be at the very beginning of the string.

After the normalisation, any digit can have at most one operation preceding it, for a total of $6+1 = 7$ possibilities, and the number of possible arrangements to surround $l = L(s)$ digits with pairs of parentheses is the Catalan number $C(l-1) = \binom{2l-2}{l-1}/l$, which is smaller than 4^l. The total number of possibilities for such strings is therefore less than $(7 \times 4)^l$. □

On the other hand, for nice Friedman numbers it is still possible to use the infix method for small enough values of *b*.

Theorem 23. *In binary, ternary and base four the density of nice Friedman numbers is* 1.

Proof. Let us consider $b = 2$. Table B.1 lists the radical-free integers and the tuple-sizes of such that can be produced from a string of the form $[1_{BINARY}]^n$. The left-most column gives n, followed by the number $[1_{BINARY}]^n$. The next two columns gives the radical-free integers that can be produced from this string but not from any smaller n, as well as their total number. Lastly, we give $N'([1_{BINARY}]^n)$.

n	Number	radical-free	total	N'
1	1_{BIN}	$\{\}$	0	0
2	11_{BIN}	$\{2, 3\}$	2	2
3	111_{BIN}	$\{7\}$	1	3
4	1111_{BIN}	$\{5, 6, 15\}$	3	8
5	11111_{BIN}	$\{10, 12, 14, 21, 26, 28, 31\}$	7	18
6	111111_{BIN}	$\{\ldots\}$	23	55
7	1111111_{BIN}	$\{\ldots\}$	80	170

Table B.1: Radical-free numbers that can be formed from $[1_{BIN}]^n$

As can be seen, the maximal-sized prefix code composed of tuples of radical-free numbers that can be calculated from $[1_{BIN}]^7$ is already more than 2^7, giving an example of an s for which $N'(s) > b^{L(s)}$ and proving that the density of binary nice Friedman numbers is 1. The prefix code, in this case, is composed of all tuples that can be calculated from $[1_{BIN}]^7$ but not from $[1_{BIN}]^5$.

To prove for ternary and quaternary numbers, we require a finer tool. To explain it, we re-prove the claim regarding the density of binary nice Friedman numbers without making use of the last two rows of Table B.1.

If X is a set of integer tuples, let $\overline{X}$ be the subset of X containing only the tuples composed solely of radical-free integers.

Let $M(n)$ be $\overline{\text{tuplespan}'([1_{BIN}]^n)} \setminus \overline{\text{tuplespan}'([1_{BIN}]^{n-1})}$. Clearly, $N'([1]^n) \geq |M(n)|$, because $M(n)$ is a prefix code. Furthermore, if $x \in M(n)$ and $y \in M(m)$ then the tuple that is x appended to y is in $M(n+m)$.

Consider, again, the values of the first five rows of Table B.1. The number of radical-free numbers found for $n = 1, \ldots, 5$ is $0, 2, 1, 3, 7$, respectively. For any $n > 5$, any member of $\overline{\text{tuplespan}'([1]^{n-i})}$ with $i \leq 5$ can be extended by a suffix composed of any member of $\overline{\text{span}'([1]^i)}$ to make a unique member of $M(n)$. This indicates that for any $n > 5$,

$$N'([1_{BIN}]^n) \geq M(n) \geq 2M(n-2) + M(n-3) + 3M(n-4) + 7M(n-5).$$

The value of $N'([1]^n)$ must therefore be in $\Omega(z^n)$, where z is the greatest real solution to $P(x) = x^5 - 2x^3 - x^2 - 3x - 7$. However, substituting $x = 2$ we get that $P(x) = -1 < 0$, so the greatest real solution to $P(x)$ must be greater than $b = 2$. If so, then for a sufficiently large n, $N'([1_{BIN}]^n) > b^n$, and using $s = [1_{BIN}]^n$ we can prove that the density of binary nice Friedman numbers is 1.

This principle can also be applied for ternary and quaternary. In base three we use $[2_{THREE}]^n$ to get the following number of radical-free numbers in a partial count of $n = 1, \ldots, 6$: $1, 1, 4, 22, 98, 454$. The corresponding polynomial $P(x) = x^6 - x^5 - x^4 - 4x^3 - 22x^2 - 98x - 454$ has $P(3) = -175 < 0$ and for $[3_{FOUR}]^n$ the corresponding polynomial is $P(x) = x^6 - x^5 - 3x^4 - 13x^3 - 59x^2 - 369x - 2279$ with $P(4) = -740 < 0$. In both cases, the polynomials have a solution greater than b, the base of representation, proving the claim. □

There is a gap between Theorem 22 and Theorem 23, consisting of the bases between 5 and 27 inclusive, that still needs to be fully addressed, but note that even for bases over 27, the claims do not immediately imply that the density of nice Friedman numbers is not 1.

Appendix C

P-ARAM code listing

The following is C++ code giving full, functioning implementations for the simulation programs discussed in Section 4.2. These programs incorporate all code fragments given in Section 4.2.

C.1 Code to simulate P-RAM in PSPACE

This section contains a C++ program implementing the algorithm to simulate a polynomial time RAM in PSPACE that is described in Section 4.2.1.

C.1.1 slp.h

```
/*
Interface file to the SLP and related classes.

This class simulates a P-SLP[+,-,*,<<,>>,Bool] in PSPACE on a Turing machine.
*/

#include <algorithm>
#include <vector>

class SLP;                                                    10
class Command;
```

```
class Index
{
  public:
    Command* command;
    std::vector<Command*> values;
    std::vector<int> counters;
    std::vector<int> lines;
    std::vector<int> maxima;
  public:
    Index begin() const;
    Index end() const;
    Index rbegin() const;
    Index zero() const;
    Index unity() const;
    Index& next();
  public:
    Index() {}
    Index(Command* _command, const Index& old);
    void make_index(int line, Command& value);
    Index& operator++();
    Index& operator--();
    Index normalize(const Index& ref) const;
};

int cmp(const Index& index1, const Index& index2);

bool operator<(const Index& index1, const Index& index2);
bool operator>(const Index& index1, const Index& index2);
bool operator==(const Index& index1, const Index& index2);
bool operator>=(const Index& index1, const Index& index2);
bool operator<=(const Index& index1, const Index& index2);
bool operator!=(const Index& index1, const Index& index2);

Index operator+(const Index& index1, const Index& index2);
Index operator-(const Index& index1, const Index& index2);
```

```
class Command
{                                                                    50
  private:
    int m_line;
    Index m_begin;
    Index m_end;
    Index m_rbegin;
  protected:
    virtual int eval(const Index& index) const=0;
    Index make_index(Command& exponent);
    SLP* slp;
  public:                                                            60
    const Index& begin() const { return m_begin; }
    const Index& end() const { return m_end; }
    const Index& rbegin() const { return m_rbegin; }
    virtual int cmp(const Index& index1, const Index& index2) const;
  public:
    Command(SLP& _slp);
    void set_slp(SLP* _slp, int _line);
    int operator[](const Index& index);
    int is_nonzero();
    int line() { return m_line; }                                    70
};

class Const : public Command
{
  private:
    int val;
  protected:
    virtual int eval(const Index& index) const;
  public:
    virtual int cmp(const Index& index1, const Index& index2) const;  80
    Const(SLP& _slp, int _val) : Command(_slp), val(_val!=0) {}
};
```

```
class And : public Command
{
  private:
    Command& arg1;
    Command& arg2;
  protected:
    virtual int eval(const Index& index) const;
  public:
    And(SLP& _slp, int _arg1, int _arg2);
};

class Or : public Command
{
  private:
    Command& arg1;
    Command& arg2;
  protected:
    virtual int eval(const Index& index) const;
  public:
    Or(SLP& _slp, int _arg1, int _arg2);
};

class Not : public Command
{
  private:
    Command& arg;
  protected:
    virtual int eval(const Index& index) const;
  public:
    Not(SLP& _slp, int _arg);
};

class Add : public Command
{
  private:
    Command& arg1;
```

```
    Command& arg2;
  protected:
    virtual int eval(const Index& index) const;
  public:
    Add(SLP& _slp, int _arg1, int _arg2);
};

class Sub : public Command
{
  private:
    Command& arg1;
    Command& arg2;
  protected:
    virtual int eval(const Index& index) const;
  public:
    Sub(SLP& _slp, int _arg1, int _arg2);
};

class Mult : public Command
{
  private:
    Command& arg1;
    Command& arg2;
    int make_pn(Command& command, const Index& index) const;
  protected:
    virtual int eval(const Index& index) const;
  public:
    Mult(SLP& _slp, int _arg1, int _arg2);
};

class LShift : public Command
{
  private:
    Command& mantissa;
    Index exponent;
  protected:
```

```
    virtual int eval(const Index& index) const;
  public:
    LShift(SLP& _slp, int _mantissa, int _exponent);
};

                                                                    160
class RShift : public Command
{
  private:
    Command& mantissa;
    Index exponent;
  protected:
    virtual int eval(const Index& index) const;
  public:
    RShift(SLP& _slp, int _mantissa, int _exponent);
};                                                                  170

class SLP
{
  private:
    std::vector<Command*> commands;
  public:
    SLP();
    void push_back(Command* command);
    Command& operator[](int line) { return *commands[line]; }
    int is_nonzero() { return (*commands.rbegin())->is_nonzero(); }  180
    ~SLP();
};
```

C.1.2 slp.cpp

```
/*
Implementation file of the SLP and related classes.


This class simulates a P-SLP[+,-,*,<<,>>,Bool] in PSPACE on a Turing machine.
```

```
*/

#include "slp.h"
#include <numeric>

using namespace std;

Index Index::begin() const
{
  Index rc=*this;
  for(int i=0;i<maxima.size();++i) {
    rc.counters[i]=-maxima[i];
  }
  return rc;
}

Index Index::end() const
{
  Index rc=*this;
  rc.counters.clear();
  rc.values.clear();
  rc.maxima.clear();
  return rc;
}

Index Index::rbegin() const
{
  Index rc=*this;
  for(int i=0;i<maxima.size();++i) {
    rc.counters[i]=maxima[i];
  }
  return rc;
}

Index Index::zero() const
{
```

```
  Index rc=*this;
  for(int i=0;i<maxima.size();++i) {
    rc.counters[i]=0;
  }
  return rc;
}

Index Index::unity() const
{
  Index rc=zero();
  rc.counters[counters.size()-1]=1;
  return rc;
}

Index& Index::next()
{
  int i;
  for (i=0;i<maxima.size();++i) {
    ++counters[i];
    if (counters[i]==maxima[i]+1) {
      counters[i]=-maxima[i];
    } else {
      return *this;
    }
  }
  if (i==maxima.size()) {
    *this=end();
  }
  return *this;
}

Index::Index(Command* _command, const Index& old)
  : command(_command), values(old.values), counters(old.counters),
    lines(old.lines), maxima(values.size())
{
  for(int i=0;i<values.size();++i) {
```

```
    maxima[i]=1<<(command->line()-lines[i]);
  }
}

void Index::make_index(int line, Command& value)
{
  values.push_back(&value);
  counters.push_back(0);
  lines.push_back(line);
  maxima.push_back(1<<(command->line()-line));
}

Index& Index::operator++()
{
  if (counters.size()==0) {
    return *this;
  }
  Index rc=end();
  for(Index i=begin();i!=end();i.next()) {
    if ((i>*this)&&((rc==end())||(rc>i))) {
      rc=i;
    }
  }
  *this=rc;
  return *this;
}

Index& Index::operator--()
{
  if (counters.size()==0) {
    return *this;
  }
  Index rc=end();
  for(Index i=begin();i!=end();i.next()) {
    if ((i<*this)&&((rc==end())||(rc<i))) {
      rc=i;
```

```
    }
  }
  *this=rc;
  return *this;
}

Index Index::normalize(const Index& ref) const
{
  if (counters.size()==0) {
    return *this;
  }
  Index rc=ref.end();
  for(Index i=ref.begin();i!=ref.end();i.next()) {
    if ((*this>=i)&&((rc==ref.end())||(rc<i))) {
      rc=i;
    }
  }
  return rc;
}

int cmp(const Index& index1, const Index& index2)
{
  if (index1.counters.size()==0) {
    if (index2.counters.size()==0) {
      return 0;
    } else {
      return 2;
    }
  } else if (index2.counters.size()==0) {
    return -2;
  }
  Index indexL,indexR;
  if (index1.counters.size()>=index2.counters.size()) {
    indexL=indexR=index1;
  } else {
    indexL=indexR=index2;
```

```
  }
  for(int i=0;i<indexL.counters.size();++i) {
    indexL.counters[i]=indexR.counters[i]=0;
  }
  for(int i=0;i<index1.counters.size();++i) {
    indexL.counters[i]=max(index1.counters[i],0);
    indexR.counters[i]=max(−index1.counters[i],0);
  }
  for(int i=0;i<index2.counters.size();++i) {
    indexL.counters[i]+=max(−index2.counters[i],0);
    indexR.counters[i]+=max(index2.counters[i],0);
  }
  // comparing indexL to indexR is equivalent to comparing index1 to index2,
  // but none of the coefficients are negative.
  return indexL.command−>cmp(indexL,indexR);
}

Index Command::make_index(Command& exponent)
{
  m_begin.make_index(line(),exponent);
  m_begin=m_begin.zero();
  m_end=m_begin.end();
  m_rbegin=m_begin.rbegin();
  return m_begin.unity();
}

int Command::cmp(const Index& index1, const Index& index2) const
{
  Command* command=(*index1.values.rbegin());
  Index i=command−>begin();
  int maximum=max(accumulate(index1.counters.begin(),index1.counters.end(),0),
                  accumulate(index2.counters.begin(),index2.counters.end(),0));
  int logmax=0;
  while (1<<logmax<maximum) ++logmax;
  i.maxima[0]+=logmax;
  int acc1=0;
```

```
  int acc2=0;
  int sign=0;
  for(;i!=command->end();++i) {
    acc1>>=1;
    acc2>>=1;
    for(int j=0;j<index1.counters.size();++j) {
      int bit=(*index1.values[j])[i];
      acc1+=index1.counters[j]*bit;
      acc2+=index2.counters[j]*bit;
    }
    if (acc1%2>acc2%2) {
      sign=1;
    }
    if (acc1%2<acc2%2) {
      sign=-1;
    }
  }
  return sign;
}

Command::Command(SLP& _slp)
{
  _slp.push_back(this);
}

void Command::set_slp(SLP* _slp, int _line)
{
  slp=_slp;
  m_line=_line;
  if (m_line>0) {
    m_begin=Index(this,(*slp)[m_line-1].begin());
    m_begin=m_begin.zero();
    m_end=m_begin.end();
    m_rbegin=m_begin.rbegin();
  }
  if (m_line==1) {
```

```
    make_index(*this);
  }
}

bool operator<(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)<0;
}

bool operator>(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)>0;
}

bool operator==(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)==0;
}

bool operator>=(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)>=0;
}

bool operator<=(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)<=0;
}

bool operator!=(const Index& index1, const Index& index2)
{
  return cmp(index1,index2)!=0;
}

Index operator+(const Index& index1, const Index& index2)
{
```

```
  Index rc=index1;
  for(int i=0;i<rc.counters.size();++i) {
    rc.counters[i]=index1.counters[i]+index2.counters[i];
  }
  rc.normalize(index1);
  return rc;
}

Index operator−(const Index& index1, const Index& index2)
{
  Index rc=index1;
  for(int i=0;i<rc.counters.size();++i) {
    rc.counters[i]=index1.counters[i]−index2.counters[i];
  }
  rc.normalize(index1);
  return rc;
}

int Command::operator[](const Index& index)
{
  if (index<begin()) {
    return 0;
  }
  Index normalized=index.normalize(begin());
  return eval(normalized);
}

int Command::is_nonzero()
{
  for(Index index=begin();index!=end();++index) {
    if (eval(index)==1) {
       return 1;
    }
  }
  return 0;
}
```

```
int Const::eval(const Index& index) const
{
  return val&&(index==begin());
}

int Const::cmp(const Index& index1, const Index& index2) const
{
  return index1.counters[0]−index2.counters[0];
}

int And::eval(const Index& index) const
{
  return arg1[index]&&arg2[index];
}

And::And(SLP& _slp, int _arg1, int _arg2)
  : Command(_slp), arg1(_slp[_arg1]), arg2(_slp[_arg2])
{
}

int Or::eval(const Index& index) const
{
  return arg1[index]||arg2[index];
}

Or::Or(SLP& _slp, int _arg1, int _arg2)
  : Command(_slp), arg1(_slp[_arg1]), arg2(_slp[_arg2])
{
}

int Not::eval(const Index& index) const
{
  int rc=1;
  for(Index i=rbegin();(i>=index)&&(rc==1);−−i) {
    rc=1−arg[i];
```

```
  }
  if (rc==1) {
    return 0; // The tweaked NOT operator
  }
  return 1-arg[index];
}

Not::Not(SLP& _slp, int _arg) : Command(_slp), arg(_slp[_arg])
{
}

int Add::eval(const Index& index) const
{
  int acc=0;
  for(Index i=begin();i<=index;++i) {
    acc>>=1;
    acc+=arg1[i]+arg2[i];
  }
  return acc%2;
}

Add::Add(SLP& _slp, int _arg1, int _arg2)
  : Command(_slp), arg1(_slp[_arg1]), arg2(_slp[_arg2])
{
}

int Sub::eval(const Index& index) const
{
  int acc=0;
  Index i;
  for(i=begin();i<=index;++i) {
    acc>>=1;
    acc+=arg1[i]-arg2[i];
  }
  int rc=(acc%2)!=0;
  for(;i!=end();++i) {
```

```
    acc>>=1;
    acc+=arg1[i]-arg2[i];
  }
  if (acc==0) {
    return rc;
  } else {
    return 0; // arg2>arg1
  }
}

Sub::Sub(SLP& _slp, int _arg1, int _arg2)
  : Command(_slp), arg1(_slp[_arg1]), arg2(_slp[_arg2])
{
}

int Mult::make_pn(Command& command, const Index& index) const
{
  int curr_bit=command[index];
  if (index<begin()) {
    return 0;
  } else if (index==begin()) {
    return -curr_bit;
  } else {
    Index prev=index;
    --prev;
    int prev_bit=command[prev];
    return prev_bit-curr_bit;
  }
}

int Mult::eval(const Index& index) const
{
  int acc=0;
  for(Index i=begin();i<=index;++i) {
    acc>>=1;
    for(Index p1=begin();p1!=end();++p1) {
```

```
      Index p2=i−p1;
      acc+=make_pn(arg1,p1)*make_pn(arg2,p2);
    }
  }
  return (acc%2)!=0;
}

Mult::Mult(SLP& _slp, int _arg1, int _arg2)
  : Command(_slp), arg1(_slp[_arg1]), arg2(_slp[_arg2])
{                                                                    410
}

int LShift::eval(const Index& index) const
{
  return mantissa[index−exponent];
}

LShift::LShift(SLP& _slp, int _mantissa, int _exponent)
  : Command(_slp),
    mantissa(_slp[_mantissa]), exponent(make_index(_slp[_exponent]))  420
{
}

int RShift::eval(const Index& index) const
{
  return mantissa[index+exponent];
}

RShift::RShift(SLP& _slp, int _mantissa, int _exponent)
  : Command(_slp),                                                   430
    mantissa(_slp[_mantissa]), exponent(make_index(_slp[_exponent]))
{
}

SLP::SLP()
{
```

```
  new Const(*this,0);
  new Const(*this,1);
}

                                                                    440
void SLP::push_back(Command* command)
{
  commands.push_back(command);
  (*commands.rbegin())->set_slp(this,commands.size()-1);
}

SLP::~SLP()
{
  for(vector<Command*>::iterator it=commands.begin();
      it!=commands.end();++it) {                                    450
    delete *it;
  }
}
```

C.1.3 main.cpp

```
/*
Example program, using the class SLP.
*/

#include <iostream>

#include "slp.h"

using namespace std;

                                                                    10
int main(void)
{
  SLP sample_slp;     // The SLP being simulated is: s[0] := 0; s[1] := 1;
  new LShift(sample_slp,1,1);                  // s[2] := s[1] << s[1];
  new Sub(sample_slp,2,1);                     // s[3] := s[2] - s[1];
```

```
  new Sub(sample_slp,3,1);                    // s[4] := s[3] - s[1];
  cout << sample_slp.is_nonzero() << endl; // (s[4] != 0) ?
  return 0;
}
```

C.2 Code to simulate P-ARAM in PSPACE

This section contains a C++ program implementing the algorithm to simulate a polynomial time ARAM in PSPACE that is described in Section 4.2.2. The program requires both the new files described here and those of Appendix C.1.

C.2.1 aln.h

```
/*
Interface file to the ALN class.

This class extends the capabilities of class SLP by adding support of
arbitrary large numbers.

Together, the two enable simulation of an SLP[+,-,*,<<,>>,Bool] that has access to an ALN
in PSPACE on a Turing machine.
*/

#include "slp.h"

class ALN : public Command
{
  private:
    Index exponent;
  protected:
    virtual int eval(const Index& index) const;
  public:
    virtual int cmp(const Index& index1, const Index& index2) const;
    ALN(SLP& _slp) : Command(_slp), exponent(make_index(*this)) {}
```

```
};
```

C.2.2 aln.cpp

```
/*
Implementation file of the ALN class.

This class extends the capabilities of class SLP by adding support of
arbitrary large numbers.

Together, the two enable simulation of an SLP[+,-,*,<<,>>,Bool] that has access to an ALN
in PSPACE on a Turing machine.
*/

#include "ALN.h"

int ALN::eval(const Index& index) const
{
  return index==exponent;
}

int ALN::cmp(const Index& index1, const Index& index2) const
{
  if (*index1.counters.rbegin()==*index2.counters.rbegin()) {
    Index indexL=index1;
    Index indexR=index2;
    indexL.counters.pop_back();
    indexL.values.pop_back();
    indexL.maxima.pop_back();
    indexR.counters.pop_back();
    indexR.values.pop_back();
    indexR.maxima.pop_back();
    return Command::cmp(indexL,indexR);
  } else {
```

```
    return (*index1.counters.rbegin())−(*index2.counters.rbegin());
  }
}
```

C.2.3 main_w_ALN.cpp

```
/*
Example program, using the class ALN.
*/

#include <iostream>

#include "aln.h"

using namespace std;

                                                                    10
int main(void)
{
  SLP sample_slp;     // The SLP being simulated is: s[0] := 0; s[1] := 1;
  new ALN(sample_slp);                          // s[2] := 2^\omega;
  new Add(sample_slp,1,2);                      // s[3] := s[1] + s[2];
  cout << sample_slp.is_nonzero() << endl; // (s[3] != 0) ?
  return 0;
}
```

Vita

Publications arising from this thesis include:

Brand, M., (2012). Does indirect addressing matter?, *Acta Informatica*, **49**(7): 485–491.

Brand, M., (2013). Computing with and without arbitrary large numbers. In T.-H. Hubert Chan, Lap Chi Lau, and Luca Trevisan, editors, *Theory and Applications of Models of Computation, 10th International Conference, TAMC 2013, Hong Kong, China, May 20–22, 2013. Proceedings*, volume 7876 of *Lecture Notes in Computer Science*, pages 181–192. Springer.

Brand, M., Friedman numbers have density 1, *Discrete Applied Mathematics. In press.*

Brand, M., The RAM equivalent of P vs. RP. *Submitted.*

Brand, M., Constant time sorting. *Submitted.*

Brand, M., Arbitrary sequence RAMs and the arithmetic hierarchy. *In preparation.*

Brand, M., On the density of nice Friedmans. *In preparation.*

See also:

Brand, M. (2012). Tightening the bounds on the Baron's omni-sequence. *Discrete Mathematics*, **312**(7): 1326–1335.

Brand, M., (2012). Münchhausen matrices. *Electronic Journal of Combinatorics*, **19**(4): P40.

Brand, M., Lower bounds on the Münchhausen problem. *Submitted.*

Brand, M., Small polyomino packing. *Submitted.*

Brand, M. and Stones, D. S., The trouble with network motifs: an analytical perspective. *Submitted.*

Brand, M., No easy puzzles: A hardness result for jigsaw-like puzzles and polyomino packing. *In preparation.*

Permanent Address: Clayton School of Information Technology

Monash University

Australia

This thesis was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

References

Adleman, L. (1978). Two theorems on random polynomial time, *19th Annual Symposium on Foundations of Computer Science (Ann Arbor, Mich., 1978)*, IEEE, Long Beach, Calif., pp. 75–83.

Agrawal, M., Kayal, N. and Saxena, N. (2002). PRIMES is in P, *Ann. of Math* **2**: 781–793.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1975). *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam. Second printing, Addison-Wesley Series in Computer Science and Information Processing.

Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*, Cambridge University Press, Cambridge.

Babai, L. (1985). Trading group theory for randomness, *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, ACM, New York, NY, USA, pp. 421–429.

Ben-Amram, A. M. and Galil, Z. (1992). On pointers versus addresses, *J. Assoc. Comput. Mach.* **39**(3): 617–648.

Ben-Amram, A. M. and Galil, Z. (1995). On the power of the shift instruction, *Inf. Comput.* **117**: 19–36.

Bertoni, A., Mauri, G. and Sabadini, N. (1981). A characterization of the class of functions computable in polynomial time on random access machines, *Proceedings of the Thirteenth*

Annual ACM Symposium on Theory of Computing, STOC '81, ACM, New York, NY, USA, pp. 168–176.

Blum, L., Shub, M. and Smale, S. (1989). On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines, *Bull. Amer. Math. Soc. (N.S.)* **21**(1): 1–46.

Blum, M. (1967). A machine-independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* **14**(2): 322–336.

Booth, A. D. (1951). A signed binary multiplication technique, *Q. J. Mech. Appl. Math.* **4**(2): 236–240.

Borodin, A., Cook, S. A., Dymond, P. W., Ruzzo, W. L. and Tompa, M. (1989). Two applications of inductive counting for complementation problems, *SIAM J. Comput.* **18**(3): 559–578.

Brand, M. (2012). Münchhausen matrices, *Electron. J. Comb.* **19**(4): P40.

Brassard, G. and Bratley, P. (1995). *Fundamentals of Algorithmics*, Prentice Hall.

Bshouty, N. H., Mansour, Y., Schieber, B. and Tiwari, P. (1992). Fast exponentiation using the truncation operation, *Comput. Complexity* **2**(3): 244–255.

Bürgisser, P., Clausen, M. and Shokrollahi, M. A. (1997). *Algebraic Complexity Theory*, Vol. 315 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*, Springer-Verlag, Berlin. With the collaboration of Thomas Lickteig.

Church, A. (1936). An unsolvable problem of elementary number theory, *Amer. J. Math.* **58**(2): 345–363.

Conway, J. H. and Guy, R. K. (1996). *The Book of Numbers*, Copernicus, New York.

Cook, S. A. (1971). The complexity of theorem-proving procedures, *in* M. A. Harrison, R. B. Banerji and J. D. Ullman (eds), *3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA. Proceedings*, ACM, pp. 151–158.

Cook, S. A. (1973). A hierarchy for nondeterministic time complexity, *J. Comput. System Sci.* **7**: 343–353. Fourth Annual ACM Symposium on the Theory of Computing (Denver, Colo., 1972).

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2001). Section 8.1: Lower bounds for sorting, *Introduction to Algorithms*, 2nd edn, MIT Press and McGrow-Hill, pp. 165–168.

Dittert, E. and O'Donnell, M. J. (1985). Lower bounds for sorting with realistic instruction sets, *IEEE Trans. Comput.* **34**(4): 311–317.

Dittert, E. and O'Donnell, M. J. (1986). Correction to: "Lower bounds for sorting with realistic instruction sets" [IEEE Trans. Comput. **34** (1985), no. 4, 311–317], *IEEE Trans. Comput.* **35**(10): 932.

Eppstein, D. and Galil, Z. (1988). Parallel algorithmic techniques for combinatorial computation, *Ann. Rev. Comput.* **3**: 233–283.

Fellows, M. R. (1993). Computer science and mathematics in the elementary schools, *CBMS Issues in Mathematics Education* **3**: 143–163.

Fortnow, L. and Homer, S. (2003). A short history of computational complexity, *Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS)* **80**: 95–133.

Fredman, M. L. and Willard, D. E. (1993). Surpassing the information-theoretic bound with fusion trees, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (Baltimore, MD, 1990)*, Vol. 47, pp. 424–436.

Friedman, E. (2000). Problem of the month, `http://www2.stetson.edu/~efriedma/mathmagic/0800.html`. [Online; accessed 23 August 2010].

Friedman, E. (n.d.). A036057, `http://www.research.att.com/~njas/sequences/A036057`. [Online; accessed 23 August 2010].

Gaßner, C. (2001). The P-DNP problem for infinite abelian groups, *J. Complexity* **17**(3): 574–583.

Geffert, V. (2003). Space hierarchy theorem revised, *Theor. Comp. Sci.* **295**(1-3): 171–187.

Gill, J. (1977). Computational complexity of probabilistic Turing machines, *SIAM J. Comput.* **6**(4): 675–695.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I., *Monatshefte für Mathematik und Physik* **38**: 173–198.

Goldwasser, S., Micali, S. and Rackoff, C. (1989). The knowledge complexity of interactive proof systems, *SIAM J. Comput.* **18**(1): 186–208.

Graham, R. L. and Rothschild, B. L. (1971). Ramsey's theorem for n-parameter sets, *Trans. Amer. Math. Soc.* **159**: 257–292.

Granlund, T. and Montgomery, P. L. (1994). Division by invariant integers using multiplication, *SIGPLAN Not.* **29**: 61–72.

Hagerup, T. (1998). Sorting and searching on the word RAM, *in* M. Morvan, C. Meinel and D. Krob (eds), *STACS*, Vol. 1373 of *Lecture Notes in Computer Science*, Springer, pp. 366–398.

Han, Y. (2004). Deterministic sorting in $O(n \log\log n)$ time and linear space, *J. Algorithms* **50**(1): 96–105.

Hartmanis, J. and Simon, J. (1974). On the power of multiplication in random access machines, *15th Annual Symposium on Switching and Automata Theory (1974)*, IEEE Comput. Soc., Long Beach, Calif., pp. 13–23.

Hartmanis, J. and Stearns, R. E. (1964). Computational complexity of recursive sequences, *SWCT (FOCS)*, IEEE Computer Society, pp. 82–90.

Hartmanis, J. and Stearns, R. E. (1965). On the computational complexity of algorithms, *Trans. Amer. Math. Soc.* **117**: 285–306.

Heller, H. (1986). On relativized exponential and probabilistic complexity classes, *Inform. and Control* **71**(3): 231–243.

Hennie, F. C. and Stearns, R. E. (1966). Two-tape simulation of multitape Turing machines, *J. Assoc. Comput. Mach.* **13**: 533–546.

Hong, J. W. (1979). On lower bounds of time complexity of some algorithms, *Sci. Sinica* **22**(8): 890–900.

Karnaugh, M. (1953). The map method for synthesis of combinational logic circuits, *Commun. and Electronics* **1953**: 593–599.

Karp, R. M. (1972). Reducibility among combinatorial problems, *in* R. E. Miller and J. W. Thatcher (eds), *Complexity of Computer Computations*, The IBM Research Symposia Series, Plenum Press, New York, pp. 85–103.

Kirkpatrick, D. and Reisch, S. (1984). Upper bounds for sorting integers on random access machines, *Theor. Comp. Sci.* **28**(3): 263–276.

Knuth, D. (1997). Section 5.3.1: Minimum-comparison sorting, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd edn, Addison-Wesley, pp. 180–197.

Knuth, D. E. (1976). Mathematics and computer science: coping with finiteness, *Science* **194**(4271): 1235–1242.

Koren, I. (1993). *Computer Arithmetic Algorithms*, Prentice Hall Inc., Englewood Cliffs, NJ.

Levin, L. A. (1973). Universal search problems, *Problems of Information Transmission* **9**: 265–266.

Lockhart, P. (2002). A mathematician's lament. [Online; accessed 5 June 2013].
URL: *http://www.maa.org/devlin/LockhartsLament.pdf*

Lovász, L. (2006). Graph minor theory, *Bull. Amer. Math. Soc.* pp. 75–86.

Lürwer-Brüggemeier, K. and Ziegler, M. (2008). On faster integer calculations using non-arithmetic primitives, *Unconventional Computation*, Vol. 5204 of *Lecture Notes in Comput. Sci.*, Springer, Berlin, pp. 111–128.

Mansour, Y., Schieber, B. and Tiwari, P. (1989a). The complexity of approximating the square root, *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, pp. 325–330.

Mansour, Y., Schieber, B. and Tiwari, P. (1989b). Lower bounds for computations with the floor operation, *Automata, Languages and Programming (Stresa, 1989)*, Vol. 372 of *Lecture Notes in Comput. Sci.*, Springer, Berlin, pp. 559–573.

Mansour, Y., Schieber, B. and Tiwari, P. (1991a). A lower bound for integer greatest common divisor computations, *J. Assoc. Comput. Mach.* **38**(2): 453–471.

Mansour, Y., Schieber, B. and Tiwari, P. (1991b). Lower bounds for computations with the floor operation, *SIAM J. Comput.* **20**(2): 315–327.

Minsky, M. L. (1961). Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines, *Ann. Math.* **74**(3): 437–455.

Moschovakis, Y. N. (2009). *Descriptive Set Theory*, Vol. 155 of *Mathematical Surveys and Monographs*, second edn, American Mathematical Society, Providence, RI.

Mulmuley, K. (1999). Lower bounds in a parallel model without bit operations, *SIAM J. Comput.* **28**(4): 1460–1509.

Paul, W. and Simon, J. (1982). Decision trees and random access machines, *Logic and Algorithmic (Zurich, 1980)*, Vol. 30 of *Monograph. Enseign. Math.*, Univ. Genève, Geneva, pp. 331–340.

Pickover, C. (1995). Vampire numbers, *Keys to Infinity*, Wiley, New York, pp. 227–232.

Pippenger, N. and Fischer, M. J. (1979). Relations among complexity measures, *J. Assoc. Comput. Mach.* **26**(2): 361–381.

Pratt, V. R., Rabin, M. O. and Stockmeyer, L. J. (1974). A characterization of the power of vector machines, *Sixth Annual ACM Symposium on Theory of Computing (Seattle, Wash., 1974)*, Assoc. Comput. Mach., New York, pp. 122–134.

Rabin, M. (1960). Degree of difficulty of computing a function and hierarchy of recursive sets, *Tech. Rep. 2*, Hebrew University, Jerusalem.

Rabin, M. O. (1963). Probabilistic automata, *Information and Control* **6**(3): 230–245.

Rabin, M. O. (1980). Probabilistic algorithms in finite fields, *SIAM J. Comput.* **9**(2): 273–280.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems, *IBM J. Res. Dev.* **3**(2): 114–125.

Radó, T. (1962). On non-computable functions, *Bell System Tech. J.* **41**: 877–884.

Rattner, D. (2003). A080035, `http://www.research.att.com/~njas/sequences/A080035`. [Online; accessed 23 August 2010].

Regan, K. W. (1993). Machine models and linear time complexity, *SIGACT News* **24**: 5–15.

Rich, E. (2008). The problem classes FP and FNP, *Automata, Computability and Complexity: Theory and Applications*, Prentice Hall, pp. 689–694.

Rivest, R. L., Shamir, A. and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems, *Commun. ACM* **21**(2): 120–126.

Rogers, Jr., H. (1987). *Theory of Recursive Functions and Effective Computability*, second edn, MIT Press, Cambridge, MA.

Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities, *J. Comput. System. Sci.* **4**: 177–192.

Schönhage, A. (1979). On the power of random access machines, *Automata, Languages and Programming (Sixth Colloq., Graz, 1979)*, Vol. 71 of *Lecture Notes in Comput. Sci.*, Springer, Berlin, pp. 520–529.

Schroeppel, R. (1973). *A Two Counter Machine Cannot Calculate* 2^N, Artificial Intelligence Memo, Massachusetts Institute of Technology, A.I. Laboratory.

Scott, D. (1967). Some definitional suggestions for automata theory, *J. Comput. Syst. Sci.* **1**(2): 187–212.

Seiferas, J. I. (1977). Techniques for separating space complexity classes, *J. Comput. System Sci.* **14**(1): 73–99.

Seiferas, J. I., Fischer, M. J. and Meyer, A. R. (1978). Separating nondeterministic time complexity classes, *J. Assoc. Comput. Mach.* **25**(1): 146–167.

Shamir, A. (1979). Factoring numbers in $O(\log n)$ arithmetic steps, *Inform. Process. Lett.* **8**(1): 28–31.

Simon, J. (1977). On feasible numbers (preliminary version), *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (Boulder, Colo., 1977)*, Assoc. Comput. Mach., New York, pp. 195–207.

Simon, J. (1979). Division is good, *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, pp. 411–420.

Simon, J. (1981). Division in idealized unit cost RAMs, *J. Comput. System Sci.* **22**(3): 421–441. Special issue dedicated to Michael Machtey.

Simon, J. and Szegedy, M. (1992). On the complexity of RAM with various operation sets, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, ACM, New York, NY, USA, pp. 624–631.

Skewes, S. (1933). On the difference $\pi(x) - \mathrm{Li}(x)$, *J. London Math. Soc.* **8**: 277–283.

Solovay, R. and Strassen, V. (1977). A fast Monte-Carlo test for primality, *SIAM J. Comput.* **6**(1): 84–85.

Stearns, R. E., Hartmanis, J. and Lewis, P. M. (1965). Hierarchies of memory limited computations, *Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*, FOCS '65, pp. 179–190.

Stockmeyer, L. J. (1976). Arithmetic versus Boolean operations in idealized register machines, *Tech. Rep. RC 5954*, IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

Strassen, V. (1973). Vermeidung von Divisionen, *J. Reine Angew. Math.* **264**: 184–202.

Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22**: 215–225.

Tarski, A. (1939). On undecidable statements in enlarged systems of logic and the concept of truth, *J. Symbolic Logic* **4**: 105–112.

Trahan, J. L., Loui, M. C. and Ramachandran, V. (1992). Multiplication, division, and shift instructions in parallel random-access machines, *Theor. Comp. Sci.* **100**(1): 1–44.

Trahan, J. L., Ramachandran, V. and Loui, M. C. (1994). Parallel random access machines with both multiplication and shifts, *Inform. and Comput.* **110**(1): 96–118.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.* **42**: 230–265.

Valiant, L. G. (1979). The complexity of computing the permanent, *Theor. Comput. Sci.* **8**: 189–201.

van Emde Boas, P. (1990). Machine models and simulations, *Handbook of Theoretical Computer Science (Vol. A)*, MIT Press, Cambridge, MA, USA, pp. 1–66.

Weisstein, E. (2009). A014575, `http://www.research.att.com/~njas/sequences/A014575`. [Online; accessed 23 August 2010].

Wikipedia (2012a). Merge sort, `http://en.wikipedia.org/wiki/Merge_sort`. [Online; accessed 9 July 2012].

Wikipedia (2012b). Time complexity, `http://en.wikipedia.org/wiki/Time_complexity`. [Online; accessed 9 July 2012].

Wilson, D. (2009). A020342, `http://www.research.att.com/~njas/sequences/A020342`. [Online; accessed 23 August 2010].

Yarbrough, L. D. (1963). Decimal-to-binary conversion of short fields, *Commun. ACM* **6**: 63–64.

Žák, S. (1983). A Turing machine time hierarchy, *Theor. Comp. Sci.* **26**(3): 327–333.

Afterword, on a personal note

Whew, what a ride this has been. What a journey! I am grateful to have been able to undertake, over the past few years, this safari into a rarely-visited mathematical no-man's-land, and to come back with pictures of some exotic, and in some cases never-before-seen creatures.

Like most trips away from the beaten path, this excursion did not come without its share of forewarnings. Many a concerned friend advised me, before and during, that perhaps I should rethink this idea of studying the unit cost integer RAM. Their reasons were compelling: the RAM, they said, is notorious for being too powerful to be a realistic model. It is unreasonable, outmoded and unfashionable. Publishing about the integer RAM will be difficult, and any results about it are bound to lack any real-world import.

Not without some foreboding, I have chosen to embark on my quest despite all this advice, but this does not mean that I ignored it. In the preface, I mention that this thesis is an exploration into the essence of computing. The "Conclusions" section is written in this spirit, giving hard, mathematical truths about the nature of computing, RAMs, ALNs, etc.. However, in parallel to this study, I have also, over the same time span, been busy with a different sort of exploration, this one, following the calls against the practicality of what I was doing, into understanding the nature of practicality itself. In this afterword, allow me to share some of the softer truths that I uncovered in this other quest.

In the thesis itself, I compare the study of RAMs to the study of cardinals. A comparison closer to home would perhaps be to number theory, where it is quite common to use "unreasonably large" integers to prove a point regarding all integers. If we were bound, in our discussions, to numbers that can easily be stored and manipulated by computers, there would

not be any Graham's number or Skewes' number to shed light on important combinatorial problems. In this limited range of integers, even the density of Friedman numbers, discussed in Appendix B, would be close to 0.01, rather than 1. And yet, it is the 1, not the 0.01, that is the interesting number-theoretical result.

Are the rules different in computer science? Do we restrict ourselves there only to what is practically computable? If so, then one has to wonder about the centrality of NP, or, in fact, any Oracle-based computation, and yet we use these all the time, we build hierarchies around them, and we pose P vs. NP as, perhaps, *the* central question of computer science.

Realistic integer sorting is done in $O(n)$ time using radix sorting. And yet, the most celebrated result in integer sorting, Han's (2004) $O(n \log\log n)$-time sorting algorithm, is only applicable when using the transdichotomous computational model (Fredman and Willard, 1993), which is not only unrealistic, but also on philosophically shaky grounds because it violates the basic dichotomy between machine model and problem size.

One can argue the reverse direction equally well. Is P realistic? Under the auspices of P, academic papers time and again present algorithms that are unrealistic, either because the exponent is too large or because the multiplicative factor is too large. The author of these lines has many times — more often than not, in fact — used in practical industry settings exponential time algorithms, because in practice these turn out to be faster than the polynomial-time ones. In fact, I challenge the reader to find a case in which either $O(1)$-time array initialisation or linear-time median finding — both, cornerstone algorithms in data structure design — have any practical use. In twenty years of algorithmics, I have not encountered a first case of this.

Realism — if, by realism, we mean "what can be implemented by a computer" — seems to be an irrelevant metric for theoretical study. To paraphrase a well-known quote: computer science is no more about computers than astronomy is about telescopes. (Fellows, 1993)

So, why *do* we study Oracles and transdichotomous models? For the same reason that we study the unit-cost RAM: because it is beautiful. To paraphrase from Lockhart's (2002) moving plea, teaching math because it is useful is akin to teaching people to read in order for them to fill out vehicle registration forms.

The reason why we study the complexity class P is because its closure properties and invariance to Turing machine flavour make it an *elegant* object of study. In the same way, the unit cost RAM is an elegant device.

Moreover, the objects of study in this thesis are, in my mind, not the RAMs themselves, but the nonnegative integers, the arithmetic and nonarithmetic operations allowed, and, of course, delicate and little-understood properties such as randomness. Like many mathematical objects, these are simple constructs that engender fascinatingly complex behaviours, and it is this tendency that makes them worthy of study, and it is this tendency that makes them appear again and again in different contexts, making their study in one context sometimes pay off in another.

Indeed, the two new mathematical objects introduced by this thesis exemplify well this idea of mathematical beauty. First, ALNs were introduced. This essentially simple concept was shown to exhibit fantastically complex behaviours and, moreover, to generate unexpected amounts of computational power. The study disproves our strong intuition that such arbitrary numbers cannot be useful. Even by sheer force of the surprise factor involved, this exploration of ALNs proved itself to be worthwhile.

Next, we introduced the complexity class PEL. This demonstrated its innate simplicity and elegance by cropping up in the context of studying two completely different machines, one a deterministic RAM, the other an RRAM, both being able to recognise exactly the languages in this set in polynomial time, without there seeming to be any reason for this convergence. (And, to make the point even further, we have no idea whether this strange equality continues to hold if, say, both machines are now given an ALN to work with.)

The beauty or elegance of our objects of study is therefore not in question. However, having thought much about this idea of "practicality", my present belief is that skepticism about RAMs is actually grounded elsewhere. I believe it is, in fact, skepticism about the *study* of RAMs, on the grounds that such study has little potential to impact on classically "interesting" problems.

Here is how I see the impact of the present thesis on non-RAM-related problems:

First, consider the arithmetical hierarchy and what lies beyond it. By considering the ASRAM time-complexity of problems, we have gained an unexpected new tool with which to stratify the difficulty of problems that are outside this hierarchy.

Second: stochasticity. One of the fundamental questions in computer science is the P vs. RP question. Going into this research, I did not believe any RAM result could possibly have any implication for this central problem. And yet, the results arrived at uncover an interesting truth: P vs. RP is a *relative* question. Its answer depends on the computational model within which it is asked. While, perhaps, not shedding any light on the *answer* to this question for Turing machines, this sheds light on the question itself, in that now it is not whether stochasticity adds computational power, but whether it does so *for Turing machines*. At least in my mind, the importance of this find is that it makes the original P vs. RP question *less important*. It is now no longer a fundamental, universal question about the nature of computation, but merely one about a particular model of computation, invented at a particular moment in history.

Third: the thesis pushed through the apparently best-possible bound of $O(n)$-time sorting. Once again, the achievement is not in the algorithm itself, but rather in forcing us to reconsider what we mean by "sorting" (or any other specific algorithm), and therefore what we take as its absolute, best-possible time-bounds (or memory bounds, or any other resource). The answers to these questions are still far from given.

And last, as a cherry on top, the same techniques developed here were put to use in solving a (minor) question from number theory, again demonstrating a cross between disciplines.

So, with the hindsight of having all of these results, I think the unit-cost integer RAM can no longer be so readily dismissed. It is a powerful tool, whose true powers we have only just now begun to discover. It holds many other secrets that it still has not told us, both about itself and about many other, completely unrelated problems. I feel lucky for having studied RAMs, and hope to continue studying them, and letting them uncover new secrets, for many years to come.

I can only hope you will join me in this quest.