Using Automata Theory to Solve Problems in Additive Number Theory

by

Aayush Rajasekaran

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Additive number theory is the study of the additive properties of integers. Perhaps the best-known theorem is Lagrange's result that every natural number is the sum of four squares. We study numbers whose base-k representations have certain interesting properties. In particular, we look at *palindromes*, which are numbers whose base-k representations read the same forward and backward, and *binary squares*, which are numbers whose binary representation is some block repeated twice (like $(36)_2 = 100100$).

We show that all natural numbers are the sum of four binary palindromes. We also show that all natural numbers are the sum of three base-3 palindromes, and are also the sum of three base-4 palindromes. We also show that every sufficiently large natural number is the sum of four binary squares.

We establish these results using virtually no number theory at all. Instead, we construct automated proofs using automata. The general proof technique is to build an appropriate machine, and then run decision algorithms to establish our theorems.

Acknowledgements

If the statement of authorship that opens this thesis gives the suggestion that this was a "one-man effort", let me hastily dispel that notion. There are many individuals and institutions that I have left unthanked for too long, and it is only proper that I attempt to right that wrong.

First and foremost, I would like to thank my supervisor, Dr. Jeffrey Shallit, for everything he has given me over the past 2 years. It is no exaggeration to say that this thesis would not have been possible without you. Your breadth of knowledge, passion for research, and incredible assiduity are all truly inspiring. I will always remember the day you sent me an email with the subject "cool problem" that included attachments describing nested-word automata. The past 2 years under your supervision have seen many weekly meetings, a few published papers, and nearly a dozen trivia nights. I will keep these memories with me, and will remain grateful to you for all the opportunities, experiences, and skills you have given me. Thank you, Jeff.

The work described in this thesis was performed in collaboration with some remarkable people. My co-author, Tim Smith, came up with many of the ideas that were critical in proving our results. I'm also grateful to the other mathematicians I got to collaborate with during my Master's study: Narad Rampersad, Dirk Nowotka, and Madhusudan Parthasarathy.

I would like to think Dr. Eric Blais and Dr. Jason Bell for agreeing to read this thesis. More generally, I would like to think the faculty and staff at the Cheriton School of Computer Science. I would also like to acknowledge the excellent psychological support I received during my course of therapy at the Centre for Mental Health Research. The University of Waterloo has been my home for 7 years now, and leaving is far more bitter than sweet.

There are two individuals whose friendships I savagely exploited for assistance at every stage of my graduate studies. Thank you, Ming-Ho Yee, for having been a big brother of sorts since 2011. How do most graduate students survive without your advice? I'm glad I never had to find out. Rein Otsason, you are in every page of this thesis. The story of my Master's study is the story of our relationship. Thank you.

And finally, I would like to thank the many people who make my daily life the joy that it is. Perhaps the most valuable service one human being can provide to another is to make them laugh. The many, many people I ungratefully demand that service from include Arjun, Ian, Christine, Shelby, Nick P., Nick S., Anubhav, and most recently, Jacky Jack. Thank you all.

For the constants.

Moo.

Dada and Anna.

Kira and Faith.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Contributions of the Thesis

The primary results presented in this thesis relate to expressing numbers as sums of binary palindromes and binary squares (defined in Sections 1.4 and 1.5). In Theorem 1, we show that all natural numbers are the sum of at most 4 binary palindromes. Theorem 4 establishes a similar result for binary squares. In Chapter 6, we use the automatic theorem-proving software `Walnut`[30] to establish a variety of results regarding the additive properties of automatic sequences.

The results regarding palindromes first appeared in the author's paper with Shallit and Smith [35]. The results regarding squares first appeared in the author's paper with Madhusudan, Nowotka, and Shallit [27]. Some sections of this thesis have been copied verbatim from these papers.

1.2 Definitions and Notation

We present some standard definitions in this section. An *alphabet*, usually denoted Σ, is a finite, non-empty set of symbols. We denote the alphabet of size k as Σ_k, with the letters implicitly assumed to be $0, 1 \ldots k-1$. We denote the set of all possible (finite) words on an alphabet Σ as Σ^*.

For integers $n, k \geq 0$, we denote the canonical base-k representation of n as $(n)_k$. The canonical base-k representation does not include any leading zeros, and starts with the

most significant digit. For instance, $(39)_2 = 100111$. The canonical representation of 0 in any base is the empty string, which we denote ϵ. Note that $(n)_k$ is a word on the alphabet Σ_k.

1.3 Overview of Additive Number Theory

One can easily see that every natural number is the sum of at most 2 odd integers. It is also self-evident that a similar statement cannot be made about the even integers. There are infinitely many natural numbers that cannot be written as the sum of any number of even integers. A non-trivial statement along these lines is Lagrange's famous theorem that every natural number is the sum of four squares [16, 29]. As an example, the natural number $983 = 21^2 + 22^2 + 7^2 + 3^2$. These are the kinds of results studied under the field of additive number theory.

Formally, additive number theory is the study of the additive properties of integers [31]. For a given $T \subseteq \mathbb{N}$, an *additive basis of order* h is a subset $S \subseteq \mathbb{N}$ such that every $n \in T$ is the sum of h members, not necessarily distinct, of S. The principal problem of additive number theory is as follows: given two subsets S and T, does S form an additive basis of some order h for T, and if so, what is the smallest value of h?

It is trivially evident that the natural numbers, $\mathbb{N}$, form an additive basis of order 1 for $\mathbb{N}$. Goldbach's famous conjecture [44] can be phrased in this language. If S is the set of even integers greater than 2, and T is the set of prime numbers, the conjecture asks whether T forms an additive basis for S of order 2. There has been much research in this area, and deep techniques, such as the Hardy-Littlewood circle method [42, 4], have been developed to solve problems.

We can rephrase Lagrange's theorem in our terminology. The theorem tells us that $S = \{0^2, 1^2, 2^2, 3^2, \ldots\}$ forms an additive basis for $\mathbb{N}$ of order 4. The celebrated problem of Waring (1770) (see, e.g., [15, 41, 43]) is to determine the corresponding least order $g(k)$ for k'th powers. Since Legendre's Theorem tells us that numbers of the form $4^a(8k + 7)$ cannot be expressed as the sum of three squares, it follows that $g(2) = 4$. It is known that $g(3) = 9$ and $g(4) = 19$.

In a variation on this concept, we say that for a given $T \subseteq \mathbb{N}$, $S \subseteq \mathbb{N}$ is an *asymptotic additive basis of order* h for T if every *sufficiently large* $n \in T$ is the sum of h members, not necessarily distinct, of S. The function $G(k)$ is defined to be the least asymptotic order for k'th powers. From above we have $G(2) = 4$. It is known that $G(14) = 16$, and $4 \leq G(3) \leq 7$. Despite much work, the exact value of $G(3)$ is currently unknown.

In this thesis, we study numbers with particular representations in base k. For example, numbers of the form $11\cdots1$ in base k are sometimes called *repunits* [45]. Special effort has been devoted to factoring such numbers, with the Mersenne numbers 2^n-1 being the most famous examples. The Nagell-Ljunggren problem asks for a characterization of those repunits that are integer powers (see, e.g., [38]).

We are primarily concerned with two particular classes of numbers: palindromes, and squares. In particular, we study whether these numbers can form additive bases for $\mathbb{N}$.

1.4 Palindromes

A word is said to be a *palindrome* if it reads the same forwards and backwards. Examples of palindromes include the English word `redder`, the French word `ressasser`, and the German word `reliefpfeiler`. Palindromes have been well-studied in the context of formal languages [13, 12].

We are interested in *palindromic numbers*: those numbers whose base-k representation forms a palindrome. Palindromic numbers have been studied for some time in number theory; see, for example, [6, 7], just to name two recent references. The first few binary (base-2) palindromes are

$$0, 1, 3, 5, 7, 9, 15, 17, 21, 27, 31, 33, 45, 51, 63, 65, 73, 85, 93, 99, 107\ldots;$$

they form sequence A006995 in the *On-Line Encyclopedia of Integer Sequences* (OEIS) [40].

Recently, Banks initiated the study of the additive properties of palindromes, proving that every natural number is the sum of at most 49 numbers whose decimal representation is a palindrome [5]. Banks' result was then improved by Cilleruelo, Luca, & Baxter [9, 10], who proved that for all bases $b \geq 5$, every natural number is the sum of at most 3 numbers whose base-k representation is a palindrome. The proofs of Banks and Cilleruelo, Luca, & Baxter are both rather lengthy and case-based. Up to now, there have been no results proven for bases $k = 2, 3, 4$.

We establish the following results for bases 2, 3, and 4.

Theorem 1. *Every natural number N is the sum of at most 4 binary palindromes.*

Theorem 2. *Every natural number N is the sum of at most three base-3 palindromes.*

Theorem 3. *Every natural number N is the sum of at most three base-4 palindromes.*

1.5 Squares

The second class of numbers that we study is that of "squares". Rather than the ordinary notion of the square of an integer, we study "squares" in the sense of formal language theory [23]. If the string x is the canonical binary representation of an integer N, we call N a *binary square* if $N = 0$, or if $x = yy$ for some nonempty string y that starts with a 1. For example, $N = 221$ is a binary square, since 221 in base 2 is $11011101 = (1101)(1101)$. The first few binary squares are

$$0, 3, 10, 15, 36, 45, 54, 63, 136, 153, 170, 187, 204, 221, 238, 255, \ldots;$$

they form sequence A020330 in the *On-Line Encyclopedia of Integer Sequences* (OEIS) [40].

We have the following results regarding squares.

Theorem 4. *Every natural number $N > 686$ is the sum of four binary squares.*

Theorem 5. *Every natural number $N > 7$ is the sum of 3 generalized squares.*

1.6 Our Approach

Perhaps the most striking thing about our work is not the results themselves, but the approach we use to establish them. The novelty in our approach is that we obtain these results in additive number theory *using very little number theory at all.* Instead, we use an approach based on formal languages and automata theory.

To prove our results on binary palindromes, we used the machine model of nested word automata. This is described in Chapters 2 and 3. This model did not work for our results on palindromes in larger bases, nor did it work for our results on squares. To establish those results, we switched to the more well-known model of finite automata. In order to make this work, we needed to "fold" the representation of the input to the machines. This is described in Chapters 4 and 5.

Chapter 2

Automata Theory

2.1 Finite Automata

In this section we introduce the concept of nondeterministic finite automata (NFAs), which is the machine model we use to prove all of our results on squares and some of our results on palindromes. We only present a brief introduction here.

For a more comprehensive overview of finite automata, and for all undefined notions, please consult, e.g., [23, 22, 39, 37].

Informally speaking, an NFA is a machine model that reads an input string from left to right, one symbol at a time. The machine has a set of *states* that can either be *accepting* or *rejecting*, and maintains a *current state*. Accepting states are also referred to as *final* states in this document. As an NFA reads an input symbol, it consults its *transition function* to see what states it can transition to on the symbol read. If it is possible for an NFA to end up in an accepting state after having read the entire input string, then the NFA is said to accept the input. If not, it rejects the input string.

More formally, an NFA is a quintuple $(Q, \Sigma, \delta, I, F)$, where Q is the set of states, Σ is the alphabet of the input string, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states.

As an example, consider the NFA M with

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{0, 1\}$,

- $\delta(q_0, 0) = \{q_0, q_1\}$, $\delta(q_0, 1) = \{q_0\}$, $\delta(q_1, 0) = \{q_2\}$,
- $I = \{q_0\}$,
- $F = \{q_2\}$.

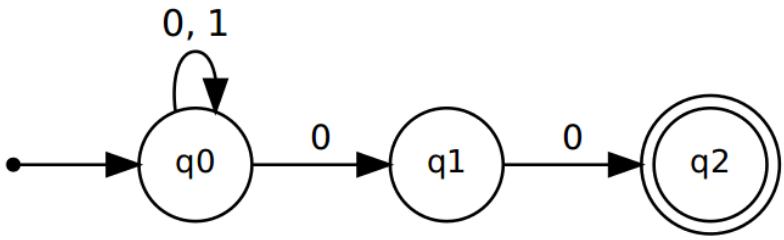

Figure 2.1: An illustration of M

This NFA only accepts input strings that end with two 0s. Note that some transitions, such as $\delta(q_1, 1)$, are implicitly defined to be the empty set, which means that M gets "stuck" if it reads a 1 when it is in the state q_1. An NFA rejects an input string if it gets stuck.

The set of strings accepted by an NFA is called the language accepted by the NFA. The class of languages that can be accepted by an NFA is the set of regular languages. The class of regular languages is closed under the operations of union, complement, and intersection. Testing emptiness, universality, and language inclusion are decidable problems for NFAs; there are well-known algorithms for these decision problems.

2.2 Nested Word Automata

Nested word automata (NWAs) were popularized by Alur and Madhusudan [2, 3], although essentially the same model was discussed previously by Mehlhorn [28], von Braunmühl & Verbeek [8], and Dymond [14]. They are closely related to a restricted variant of pushdown automata called visibly-pushdown automata (VPAs). Under linear encodings, NWAs recognize the same class of languages as VPAs, namely, the visibly-pushdown languages

[2, 3]. We only briefly describe their functionality here. For other theoretical aspects of nested-word and visibly-pushdown automata, see [26, 33, 17, 36, 32]. The definition of NWAs provided here differs from the standard definition by borrowing some aspects of VPAs. We use this definition because it matches the one used by the `ULTIMATE` program analysis framework [21, 20], which is the software tool we use to prove our results.

If the reader is familiar with pushdown automata (PDAs)[23], it is useful to note that NWAs are also stack-based machines. However, they have more restricted access to their stacks than PDAs do. An NWA's access to its stack is dictated by the input alphabet.

The input alphabet of an NWA is partitioned into three sets: a *call alphabet*, an *internal alphabet*, and a *return alphabet*. If the input symbol read is from the call alphabet, the NWA pushes its current state onto the stack, and then performs a transition, based only on the current state and input symbol read. If an input symbol is from the internal alphabet, the NWA cannot access the stack in any way. If the input symbol read is from the return alphabet, the NWA pops the state at the top of the stack, and then performs a transition based on three pieces of information: the current state, the popped state, and the input symbol read.

The rest of the behaviour of NWAs largely mimics the corresponding behaviour of an NFA. A nondeterministic NWA has a set of initial states and a set of final states. The transition function of an NWA is broken into three pieces for the call alphabet, internal alphabet, and return alphabet. An NWA accepts an input if it can terminate in an accepting state, and otherwise it rejects it.

As an example, Figure 2.2 illustrates a nested-word automaton accepting the language $\{0^n 1 2^n \ : \ n \geq 1\}$.

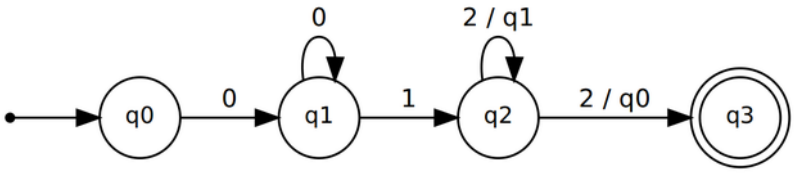

Figure 2.2: A nested-word automaton for the language $\{0^n 1 2^n \ : \ n \geq 1\}$

Here, the call alphabet is $\{0\}$, the internal alphabet is $\{1\}$, and the return alphabet is $\{2\}$.

The first 0 pushes q_0 onto the stack. Each of the $n-1$ subsequent 0s push q_1 onto the stack. When the machine reads the 1, it goes to state q_2 without accessing the stack. So long as the machine reads 2s and q_1 is on the stack, we stay in the non-accepting state q_2. Reading a 2 with q_0 on the top of the stack takes us to the only accepting state, q_3.

Nondeterministic NWAs are a good machine model for our problem, because nondeterminism allows "guessing" the palindromes that might sum to the input, and the stack allows us to "verify" that they are indeed palindromes. Deterministic NWAs are as expressive as nondeterministic NWAs, and the class of languages they accept is closed under the operations of union, complement and intersection. Finally, testing emptiness, universality, and language inclusion are all decidable problems for NWAs [2, 3].

Thus, given an NWA, we can run an algorithm to find out whether it accepts no words at all, or every word in the language. Given 2 NWAs, we can run an algorithm to decide whether the language accepted by the first NWA is a subset of the language accepted by the second NWA. We use the language inclusion algorithm to establish Theorem 1.

For a nondeterministic NWA of n states, the corresponding determinized machine has at most $2^{\Theta(n^2)}$ states, and there are examples for which this bound is attained. This very rapid explosion in state complexity potentially could make decision problems, such as language inclusion, infeasible in practice. Fortunately, we did not run into determinized machines with more than 40000 states in proving our results. Most of the algorithms invoked to prove our results run in under a minute.

Chapter 3

Results on Binary Palindromes

3.1 The Idea

We start by discussing the general construction of the NWAs that check whether inputs are sums of binary palindromes. We partition the input alphabet into the call alphabet $\{a, b\}$, the internal alphabet $\{c, d\}$, and the return alphabet $\{e, f\}$. The symbols a, c, and e correspond to 0, while b, d, and f correspond to 1. The input string is fed to the machine starting with the *least significant digit*. We provide the NWA with input strings whose first half is entirely made of call symbols, and second half is entirely made of return symbols. Internal symbols are used to create a divider between the halves (for the case of odd-length inputs).

As an example, consider the integer 197, whose binary representation is 11000101. In order to convert this to least-significant digit first representation, we reverse the string to obtain 10100011. We now draw the first four input symbols from the call alphabet, giving us the string *baba*. The second half of the input string comes from the return alphabet, and is $eeff$. Thus, the input string we would feed to our NWA to represent 197 is $babaeeff$. If we wanted to represent $117 = (1110101)_2$, the input string would be $babcfff$. Note that the middle symbol is drawn from the internal alphabet.

The idea behind the NWA is to nondeterministically guess all possible summands when reading the first half of the input string. The guessed summands are characterized by the states pushed onto the stack. The machine then checks if the guessed summands can produce the input bits in the second half of the string. The machine keeps track of any carries in the current state.

Following the above construction, we implemented our NWAs in the ULTIMATE program analysis framework[20, 21]. As an experimental spot check on the correctness of our implementations, we also built an NWA-simulator, and ran simulations of the machines on various types of inputs, which we then checked against experimental results.

For instance, we built the machine that accepts representations of integers that can be expressed as the sum of 2 binary palindromes. We then simulated this machine on every integer from 513 to 1024, and checked that it only accepts those integers that we experimentally confirmed as being the sums of 2 binary palindromes. This did not prove our results, but served as a sanity check.

The general procedure to prove our results is to build an NWA PalSum accepting only those inputs that it verifies as being appropriate sums of palindromes, as well as an NWA SyntaxChecker accepting all valid representations. We then run the decision algorithms for language inclusion, language emptiness, etc. on PalSum and SyntaxChecker as needed. To do this, we used the Automata Library toolchain of the ULTIMATE program analysis framework.

We have provided links to the proof scripts used to establish all of our results. To run these proof scripts, simply copy the contents of the script into https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automata_library and click "execute". Some of the larger proof scripts might time out on the web client, but can be run to completion by downloading the ULTIMATE Automata Library toolchain and executing them on a local machine.

3.2 The Result

In this section, we discuss how we established Theorem 1. The palindromic summands are guessed in parallel by our automaton. Consequently, if our summands are of differing lengths, then we must "remember" the most recent guesses of our summands. This leads to a blow-up in the number of states in our machines, even for a small difference in the lengths of our summands. As a result, we are restricted to guessing palindromic summands of roughly the same length. The first step in establishing our results was to find a good candidate for the lengths of the summands. To do this, we simply used trial-and-error, trying various different combinations of lengths. We considered a combination to be a strong candidate if we experimentally verified that it worked for a few million numbers.

The primary piece of our proof of Theorem 1 is the following auxiliary result.

Theorem 6. *For all $n \geq 8$, every n-bit odd integer is either a binary palindrome itself, or the sum of three binary palindromes*

(a) of lengths n, $n-2$, and $n-3$; or

(b) of lengths $n-1$, $n-2$, and $n-3$.

Proof. We build three separate automata. The first, `palChecker`, has 9 states, and simply checks whether the input number is a binary palindrome. The second, `palChecker2`, has 771 states, and checks whether an input number of length n can be expressed as the sum of three binary palindromes of lengths n, $n-2$, and $n-3$. The third machine, `palChecker3`, has 1539 states, and checks whether an input number of length n can be expressed as the sum of three binary palindromes of lengths $n-1$, $n-2$, and $n-3$. We then determinize these three machines, and take their union, to get a single deterministic NWA, `finalAut`, with 36194 states. We then run the command `finalAut = shrinkNwa(finalAut);` on `ULTIMATE`, which reduces this to a deterministic NWA of only 106 states.

The language of valid inputs to our automata is given by

$$L = \{\{a,b\}^n\{c,d\}^m\{e,f\}^n \ : \ 0 \leq m \leq 1, \ n \geq 4\}.$$

We also build an NWA `syntaxChecker` that accepts L. The final step in our proof is to simply run the following command in `ULTIMATE`.

`assert(isIncluded(syntaxChecker, finalAut));`

Since `ULTIMATE` tells us that this assertion holds, our theorem is proven. □

We are now in a position to prove our primary result.

Proof. (of Theorem 1)

It is a routine computation to verify the result for $N < 128$.

Now suppose $N \geq 128$. Let N be an n-bit integer; then $n \geq 8$. If N is odd, then Theorem 6 states that N is the sum of at most 3 binary palindromes. Otherwise, N is even.

If $N = 2^{n-1}$, then it is the sum of $2^{n-1} - 1$ and 1, both of which are palindromes.

Otherwise, $N - 1$ is also an n-bit odd integer. Use Theorem 6 to find a representation for $N - 1$ as the sum of at most 3 binary palindromes, and then add the palindrome 1 to get a representation for N. □

Corollary 7. *Given an integer N, we can find an expression for N as the sum of four binary palindromes in time linear in* $\log N$.

Proof. For $N < 128$, we can perform a brute-force search to find the required expression. For larger N, we build the machine `finalAut` as described in the proof of Lemma 6. We also build an NWA `B` that only accepts the representation of N. We can build a machine that accepts the intersection of the languages accepted by `finalAut` and `B`. This machine has at most $c \log N$ states. Performing a depth-first search of this machine and reading off the transitions gives us the required expression. See [2] for more information on how the intersections of two NWAs can be calculated. □

3.3 The Machines

We now go into more detail about how these NWAs work. We study the mechanism of the most complex machine, `palChecker3`, as an example. Recall that `palChecker3` checks whether an input number of length n can be expressed as the sum of three binary palindromes of lengths $n-1$, $n-2$, and $n-3$.

Let p, q and r be the binary palindromes representing the guessed $(n-1)$-length summand, $(n-2)$-length summand and $(n-3)$-length summand respectively. The states of `palChecker3` include 1536 t-states that are 10-tuples. We label these states

$$(g, x, y, z, k, l_1, l_2, m_1, m_2, m_3),$$

where $0 \leq g \leq 2$, while all other coordinates are either 0 or 1. The g-coordinate indicates the current carry, and can be as large as 2. The x, y and z coordinates indicate whether we are going to guess 0 or 1 for the next guesses of p, q and r respectively. The remaining coordinates serve as "memory" to help us manage the differences in lengths of the guessed summands. The k-coordinate records the most recent guess for p. We have l_1 and l_2 record the two most recent q guesses, with l_1 being the most recent one, and we have m_1, m_2 and m_3 record the three most recent r-guesses, with m_1 being the most recent one, then m_2, and m_3 being the guess we made three steps ago. We also have three s-states labeled s_0, s_1 and s_2, representing carries of 0, 1 and 2 respectively. These states process the second half of the input string.

The initial state of the machine is $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$ since we start with no carry, must guess 1 for our first guess of a valid binary palindrome, and all "previous" guesses are 0. A t-state has an outgoing transition on either a or b, but not both. If $g + x + y + z$

produces an output bit of 0, it takes a transition on a, else it takes a transition on b. The destination states are all eight states of the form $(g', x', y', z', x, y, l_1, z, m_1, m_2)$, where g' is the carry resulting from $g + x + y + z$, and x', y', z' can be either 0 or 1. Note that we "update" the remembered states by forgetting k, l_2 and m_3, and saving x, y and z.

The s-states only have transitions on the return symbols e and f. When we read these symbols, we pop a t-state off the stack. If state s_i pops the state $(g, x, y, z, k, l_1, l_2, m_1, m_2, m_3)$ off the stack, its transition depends on the value of $i + k + l_2 + m_3$. If this sum produces a carry of j, then s_i can take a transition to s_j on e if the output bit produced is 0, and on f otherwise. By reading the k, l_2 and m_3 values, we correctly realign p, q and r, correcting for their different lengths. This also ensures that we supply 0s for the last three guesses of r, the last two guesses of q and the last guess of p. The only accepting state is s_0.

It remains to describe how we transition from t-states to s-states. This transition happens when we are halfway done reading the input. If the length of the input is odd, we read a c or a d at the halfway point. We only allow certain t-states to have outgoing transitions on c and d. Specifically, we require the state's coordinates to satisfy $k = x$, $l_2 = y$, $m_1 = m_2$ and $m_3 = z$. These conditions are required for p, q and r to be palindromes. We transition to state s_i if the carry produced by adding $g + x + y + z$ is i, and we label the transition c if the output bit is 0, and d otherwise.

If the length of the input is even, then our t-states take a return transition on e or f. Once again, we restrict the t-states that can take return transitions. We require the state's coordinates to satisfy $l_1 = l_2$ and $m_1 = m_3$ to ensure our guessed summands are palindromes. Let the current state be $(g, x, y, z, k, l_1, l_1, m_1, m_2, m_1)$, and the state at the top of the stack be $(g', x', y', z', k', l_1', l_2', m_1', m_2', m_3')$. We can take a transition to s_i if the sum $g + k' + l_2' + m_3'$ produces a carry of i, and we label the transition e if the output bit is 0, and f otherwise.

The structure and behavior of `palChecker2` is very similar. One difference is that there is no need for a k-coordinate in the t-states since the longest summand guessed is of the same length as the input.

The complete script executing this proof is over 750000 lines long. Since these automata are very large, we wrote two C++ programs to generate them. Both the proof script, and the programs generating them can be found at https://cs.uwaterloo.ca/~shallit/papers.html. It is worth noting that a t-state labeled as $(g, x, y, z, k, l_1, l_2, m_1, m_2, m_3)$ in this report is labeled $q\_g\_xyz\_k\_l_1l_2\_m_1m_2m_3$ in the proof script. Also, `ULTIMATE` does not currently have a union operation for NWAs, so we work around this by using De Morgan's laws for complement and intersection.

3.4 An Example Input

In this section, we take a close look at how these NWAs might verify an example input. We use the symbol $*$ as a placeholder for either a 0 or a 1.

Let us consider the integer 187, whose binary representation, 10111011, is of length 8. This gives us an input string of $bbabffef$. Observe that $187 = 127+33+27$, where 127, 33, and 27 are binary palindromes of lengths 7, 6, and 5, respectively. Thus, the representation of 187 should be accepted by `palChecker3`.

Step 1:

Current state: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$

Input symbol: b

Stack: empty

7-bit palindrome guess: $* * * * * * *$

6-bit palindrome guess: $* * * * * *$

5-bit palindrome guess: $* * * * *$

Remaining input: $babffef$

The first coordinate indicates that we inherited a carry of 0. The next three coordinates indicate that we are guessing 1, 1, and 1 for the 7-bit, 6-bit and 5-bit summands respectively (which must be the case for the terminal bit of a valid binary palindrome). This produces an output bit of 1, which is good because we read a b. The resulting carry of adding three 1s is a 1. Following the transitions described in 3.3, we can transition to any state of the form $(1, *, *, *, 1, 1, 0, 1, 0, 0)$, where the choices we make for the wildcard $*$ symbols represent our next guesses for the palindromic summands. One such state is $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$, which is the one that works for this example.

Step 2:

Current state: $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$

Input symbol: b

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$

7-bit palindrome guess: $1 * * * * * 1$

6-bit palindrome guess: $1 * * * *1$

5-bit palindrome guess: $1 * * * 1$

Remaining input: $abffef$

The first coordinate indicates that we inherited a carry of 1. The next three coordinates indicate that we are guessing 1, 0, and 1 for the 7-bit, 6-bit and 5-bit summands respectively. The output produced is a 1, as required, and we have a new carry of 1. We can transition to any state of the form $(1, *, *, *, 1, 0, 1, 1, 1, 0)$. One such state is $(1, 1, 0, 0, 1, 0, 1, 1, 1, 0)$.

Step 3:

Current state: $(1, 1, 0, 0, 1, 0, 1, 1, 1, 0)$

Input symbol: a

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$, $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$

7-bit palindrome guess: $11 * * * 11$

6-bit palindrome guess: $10 * *01$

5-bit palindrome guess: $11 * 11$

Remaining input: $bffef$

We inherited a carry of 1, and we are guessing 1, 0, and 0 for the 7-bit, 6-bit and 5-bit summands respectively. The output produced is a 0, as required, and we have a new carry of 1. We can transition to any state of the form $(1, *, *, *, 1, 0, 0, 0, 1, 1)$. One such state is $(1, 1, 0, 1, 1, 0, 0, 0, 1, 1)$.

Step 4:

Current state: $(1, 1, 0, 1, 1, 0, 0, 0, 1, 1)$

Input symbol: b

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$, $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$, $(1, 1, 0, 0, 1, 0, 1, 1, 1, 0)$

7-bit palindrome guess: $111 * 111$

6-bit palindrome guess: 100001

5-bit palindrome guess: 11011

Remaining input: $ffef$

We inherited a carry of 1, and we are guessing 1, 0, and 1 for the 7-bit, 6-bit and 5-bit summands respectively. The output produced is a 1, as required, and we have a new carry of 1. We can transition to any state of the form $(1, *, *, *, 1, 0, 0, 1, 0, 1)$. It does not matter which one we consider for this example.

Step 5:

Current state: $(1, *, *, *, 1, 0, 0, 1, 0, 1)$

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$, $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$,
$(1, 1, 0, 0, 1, 0, 1, 1, 1, 0)$, $(1, 1, 0, 1, 1, 0, 0, 0, 1, 1)$

Input symbol: f

7-bit palindrome guess: $111 * 111$

6-bit palindrome guess: 100001

5-bit palindrome guess: 11011

Remaining input: fef

We inherited a carry of 1, and we read a symbol from the return alphabet. We thus need to transition from a t-state to an s-state. This is only allowed if $l_1 = l_2$ and $m_1 = m_3$. We observe that our current state satisfies that condition.

Next, we look at the state at the top of the stack, which is $(1, 1, 0, 1, 1, 0, 0, 0, 1, 1)$. We do this to find out what the relevant bits of the guessed summands are. This information is stored in the coordinates k, l_2, and m_3, which are 1, 0, and 1, respectively. This addition produces an output bit of 1, and a carry of 1. We can thus take a transition to s_1.

Step 6:

Current state: s_1

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$, $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$, $(1, 1, 0, 0, 1, 0, 1, 1, 1, 0)$

Input symbol: f

7-bit palindrome guess: 1111111

6-bit palindrome guess: 100001

5-bit palindrome guess: 11011

Remaining input: ef

We inherited a carry of 1, and the state at the top of the stack tells us to add 1, 1, and 0. This addition produces an output bit of 1, and a carry of 1. We can thus take a transition to s_1.

Step 7:

Current state: s_1

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$, $(1, 1, 0, 1, 1, 1, 0, 1, 0, 0)$

Input symbol: e

7-bit palindrome guess: 1111111

6-bit palindrome guess: 100001

5-bit palindrome guess: 11011

Remaining input: f

We inherited a carry of 1, and the state at the top of the stack tells us to add 1, 0, and 0. This addition produces an output bit of 0, and a carry of 1. We can thus take a transition to s_1.

Step 8:

Current state: s_1

Stack: $(0, 1, 1, 1, 0, 0, 0, 0, 0, 0)$

Input symbol: f

7-bit palindrome guess: 1111111

6-bit palindrome guess: 100001

5-bit palindrome guess: 11011

Remaining input: empty

We inherited a carry of 1, and the state at the top of the stack tells us to add 0, 0, and 0. This addition produces an output bit of 1, and a carry of 0. We can thus take a transition to s_0.

We are done! We terminate in s_0, which is an accepting state, and so we accept the input representation of 187. Further, our guessed binary palindromes are $(127)_2 = 1111111$, $(33)_2 = 100001$, and $(27)_2 = 11011$, as predicted.

3.5 Checking Correctness

As an experimental spot check on the correctness of our implementations, we built an NWA-simulator, and ran simulations of the machines on various types of inputs, which we then checked against experimental results.

For instance, we built the machine that accepts representations of integers that can be expressed as the sum of 2 binary palindromes. We then simulated this machine on

every integer from 513 to 1024, and checked that it only accepts those integers that we experimentally confirmed as being the sums of 2 binary palindromes. This served as a sanity check.

Chapter 4

Palindromes in Larger Bases

In this chapter, we study whether the base-3 and base-4 palindromes can serve as additive bases. We show that every sufficiently large natural number is the sum of at most three base-3 palindromes, and at most three base-4 palindromes.

4.1 The Folding Trick

It is possible for us to build NWAs along the same lines as we did for the binary case. Indeed, we did build these NWAs, and expected them to prove our results. Unfortunately, the machines were much larger than in the binary case, and so the decision algorithms timed out before they could give us an answer.

The larger machines were a direct result of the larger bases. In the machine `palChecker3`, for instance, we had states with 9 "variables", each of which tracked a guess that was either 0 or 1. This introduced a multiplicative factor of $2^9 = 512$. When we apply the same construction to base 3, however, we get a factor of $3^9 = 19683$. Thus, it seemed that we had pushed our luck as far as it would go with NWAs.

We hence used a modified approach using NFAs to prove these results. This approach was suggested to us by Dirk Nowotka and Parthasarathy Madhusudan, independently. We refer to this approach as the "folding trick". The challenge is because an NFA doesn't have a stack, it cannot remember the guesses it makes for the first half of the input integer. To get around this, we "fold" the input, providing both halves as input *at the same time*.

As a simple example, consider the problem of determining whether an input string is a palindrome. If we provide the string to an NFA without any modification, there is no

way for the NFA to test whether it is palindromic. Indeed, it is a simple exercise with the pumping lemma to show that the language of palindromes over the binary alphabet is not a regular language.

But consider what happens if we feed an NFA two letters at a time, one from the start, and one from the end. For instance, to test the word `hannah`, we might provide the NFA with the tuples $(\texttt{h}, \texttt{h})$, $(\texttt{a}, \texttt{a})$, $(\texttt{n}, \texttt{n})$. It is easy to see that an NFA could verify that this word is a palindrome. We can also easily reject a non-palindrome, like the word `time`, because the first input would be $(\texttt{t}, \texttt{e})$.

Note that we are making one non-trivial change here: if the string we are interested in studying is over the alphabet Σ, then we use an input alphabet of $\Sigma \times \Sigma$ for our NFA.

4.2 The Results

We prove that the base-3 palindromes and the base-4 palindromes both serve as asymptotic additive bases of order 3 for the natural numbers.

Our result for base 3, Theorem 2, is proven with the help of the following Lemma.

Lemma 8. *For all $n \geq 9$, every integer whose base-3 representation is of length n is the sum of*

(a) three base-3 palindromes of lengths n, $n-1$, and $n-2$; or

(b) three base-3 palindromes of lengths n, $n-2$, and $n-3$; or

(c) three base-3 palindromes of lengths $n-1$, $n-2$, and $n-3$; or

(d) two base-3 palindromes of lengths $n-1$ and $n-2$.

Proof. We wrote a C++ program generating a single NFA called `palBase3`. This machine has 4 parts, one for each case of the theorem. After minimizing, the machine has 378 states. We then built a second NFA that accepts folded representations of $(N)_3$ such that the unfolded length of $(N)_3$ is greater than 8. We then use `ULTIMATE` to assert that the language accepted by the second NFA is included in that accepted by the first. All these operations run in under a minute.

The relevant files can be found at https://github.com/arajasek/PalGenerators. The C++ generator is the file `PalBase3NFAUnioned.cpp`. The `ULTIMATE` file that contains the actual proof script is `PalBase3Conjecture.ats`.

We tested this machine by experimentally calculating which values of $243 \leq N \leq 1000$ could be written as the sum of palindromes satisfying one of our 4 conditions. We then asserted that for all the folded representations of $243 \leq N \leq 1000$, our machine accepts these values which we experimentally calculated, and rejects all others.

□

The proof of Theorem 2 follows from Lemma 8. An exhaustive search confirms the result for $n < 9$.

We also have the following lemma for base 4:

Lemma 9. *For all $n \geq 7$, every integer whose base-4 representation is of length n is the sum of*

(a) exactly one palindrome each of lengths $n-1$, $n-2$, and $n-3$; or

(b) exactly one palindrome each of lengths n, $n-2$, and $n-3$.

Proof. The NFA we build is very similar to the machine described for the base-3 proof. Indeed, the generator used is the same as the one for the base-3 proof, except that its input base is 4, and the only machines it generates are for the two cases of this theorem. The minimized machine has 478 states. □

An exhaustive search for $n < 7$, combined with Lemma 9, proves Theorem 3. This, together with the results previously obtained by Cilleruelo, Luca, & Baxter, completes the additive theory of palindromes for all integer bases $b \geq 2$ [9, 10].

We can also find an expression for the palindromic summands for bases 3 and 4.

Corollary 10. *Given an integer N, we can find an expression for N as the sum of three base-3 palindromes in time linear in $\log N$.*

Proof. For $N < 256$, we can perform a brute-force search to find the required expression. For larger N, we build the machine `palBase3` as described in the proof of Lemma 8. We also build an NFA `B` that only accepts the folded representation of N. We can build a machine that accepts the intersection of the languages accepted by `palBase3` and `B`. This machine has at most $c \log N$ states. Performing a depth-first search of this machine and reading off the transitions gives us the required expression. □

The same idea works for base-4 palindromes, which gives us the following result.

Corollary 11. *Given an integer N, we can find an expression for N as the sum of three base-4 palindromes in time linear in* $\log N$.

4.3 The Machines

In this section, we take a close look at what the machine for the base-3 case looks like.

We represent the input in a folded manner over the input alphabet $\Sigma_3 \cup (\Sigma_3 \times \Sigma_3)$. We align the input along the length-$(n-2)$ summand by providing the first 2 letters of the input separately. Note that these are the 2 most significant digits of the input.

If $(N)_3 = a_{2i+1}a_{2i}\cdots a_0$, we represent N as the word

$$a_{2i+1}a_{2i}[a_{2i-1}, a_0][a_{2i-2}, a_1]\cdots[a_i, a_{i-1}].$$

As an example, $(5274)_3 = 21020100$ would be represented as $2, 1, [0, 0], [2, 0], [0, 1]$.

Odd-length inputs leave a trailing unfolded letter at the end of their input. If $(N)_3 = a_{2i+2}a_{2i+1}\cdots a_0$, we represent N as the word

$$a_{2i+2}a_{2i+1}[a_{2i}, a_0][a_{2i-1}, a_1]\cdots[a_{i+1}, a_{i-1}]a_i.$$

As an example, $(15823)_3 = 210201001$ would be represented as $2, 1, [0, 1], [2, 0], [0, 0], 1$.

We need to simultaneously carry out addition on both ends. In order to do this we need to keep track of two carries. On the lower end, we track the "incoming" carry at the start of an addition, as we did in the proofs using NWAs. On the higher end, however, we track the expected "outgoing" carry. What this means is that if the expected outgoing carry of a state is, say, 1, then we only allow transitions to leave that state if the corresponding addition produces a carry of 1.

To illustrate how our machines work, we consider an NFA accepting length-n inputs that are the sum of 4 base-3 palindromes, one each of lengths n, $n-1$, $n-2$ and $n-3$. Although this is not a case in Lemma 8, each of the four cases in our theorem can be obtained from this machine by striking out one or more of the guessed summands.

Recall that we aligned our input along the length-$(n-2)$ summand by providing the 2 most significant letters in an unfolded manner. This means that our guesses for the length-n summand will be "off-by-two": when we make a guess at the higher end of the length-n

palindromic summand, its appearance at the lower end is 2 steps away. We hence need to record the last 2 guesses at the higher end of the length-n summand in our state. Similarly, we need to record the most recent higher guess of the length-$(n-1)$ summand, since it is off by one. The length-$(n-2)$ summand is perfectly aligned, and hence nothing needs to be recorded. The length-$(n-3)$ summand has the opposite problem of the length-$(n-1)$ input: its lower guess only appears at the higher end one step later, and so we save the most recent guess at the lower end.

Thus, in this machine, we keep track of 6 pieces of information:

- c_1, the carry we are expected to produce on the higher end,
- c_2, the carry we have entering the lower end,
- x_1 and x_2, the most recent higher guesses of the length-n summand,
- y, the most recent higher guess of the length-$(n-1)$ summand, and
- z, the most recent lower guess of the length-$(n-3)$ summand,

Consider a state $(c_1, c_2, x_1, x_2, y, z)$. Let $i, j, k, l \in [0, 2]$ be our next guesses for the four summands of lengths n, $n-1$, $n-2$ and $n-3$ respectively. Also, let α be our guess for the next incoming carry on the higher end. Let the result of adding $i + j + k + z + \alpha$ be a value $0 \leq p_1 < 3$ and a carry of q_1. Let the result of adding $x_1 + y + k + l + c_2$ be a value $0 \leq p_2 < 3$ and a carry of q_2. We must have $q_1 = c_1$. If this condition is met, we add a transition from this state to $(\alpha, q_2, x_2, i, j, l)$, and label the transition $[p_1, p_2]$.

The initial state is $(0, 0, 0, 0, 0, 0)$. We expand the alphabet to include special variables for the first 3 symbols of the input string. This is to ensure that we always guess a 1 or a 2 for the first (and last) positions of our summands.

The acceptance conditions depend on whether $(N)_3$ is of even or odd length. If a state $(c_1, c_2, x_1, x_2, y, z)$ satisfies $c_1 = c_2$ and $x_1 = x_2$, we set it as an accepting states. A run can only terminate in one of these states if $(N)_3$ is of even length. We accept since we are confident that our guessed summands are palindromes (the condition $x_1 = x_2$ ensures our length-n summand is palindromic), and since the last outgoing carry on the lower end is the expected first incoming carry on the higher end (enforced by $c_1 = c_2$).

We also have a special symbol to indicate the trailing symbol of an input for which $(N)_3$ is of odd length. We add transitions from our states to a special accepting state, q_{acc}, if we read this special symbol. Consider a state $(c_1, c_2, x_1, x_2, y, z)$, and let $0 \leq k < 3$ be our middle guess for the $n-2$ summand. Let the result of adding $x_1 + y + k + z + c_2$ be a value $0 \leq p < 3$ and a carry of q. If $q = c_1$, we add a transition on p from our state to q_{acc}.

4.4 Example Inputs

In this section, we take a close look at how this NFA might verify an example input. We use the symbol $*$ as a placeholder for a 0, 1, or 2.

Let us consider the integer 15328, whose base-3 representation, 210000201, is of length 9. We flag the first input letter with subscript a, the second with the subscript b. So our input string begins $2_a 1_b$. The very first tuple is flagged with c, and all others have subscript d. The trailing unfolded letter is flagged with an e.

Thus, our full input string is $2_a 1_b [0,1]_c, [0,0]_d, [0,2]_d 0_e$.

Step 1:

Current state: $(0,0,0,0,0,0)$

Input symbol: 2_a

9-bit palindrome guess: $* * * * * * * * *$

8-bit palindrome guess: $* * * * * * * *$

7-bit palindrome guess: $* * * * * * *$

6-bit palindrome guess: $* * * * * *$

Remaining input: $1_b [0,1]_c, [0,0]_d, [0,2]_d 0_e$.

The first coordinate of our current state, c_1, indicates that we must produce a carry of 0 on the higher end. The subscript of a indicates that this is the first input of the string, and therefore we must guess a 1 or 2 for the 9-bit palindrome.

Guessing a 1 is okay if we inherit a carry of 1, which we can accommodate in the c_1 coordinate of the destination state. We can thus take a transition to $(1,0,0,1,0,0)$, which is the transition that works for this example.

Step 2:

Current state: $(1,0,0,0,1,0)$

Input symbol: 1_b

9-bit palindrome guess: $1 * * * * * * * 1$

8-bit palindrome guess: $* * * * * * * *$

7-bit palindrome guess: $* * * * * * *$

6-bit palindrome guess: $* * * * * *$

Remaining input: $[0,1]_c, [0,0]_d, [0,2]_d 0_e$.

We must produce a carry of 1 on the higher end. The subscript of b indicates that this is the second input of the string, and therefore we must guess a 1 or 2 for the 8-bit palindrome.

We can guess a 0 for the 9-bit palindrome, and a 2 for the 8-bit palindrome. This is okay if we inherit a carry of 2, because then we will produce an output bit of 1, and a carry of 1, as needed. We can thus take a transition to $(2,0,1,0,2,0)$, which is the transition that works for this example.

Step 3:

Current state: $(2,0,1,0,2,0)$

Input symbol: $[0,1]_c$

9-bit palindrome guess: $10*****01$

8-bit palindrome guess: $2******2$

7-bit palindrome guess: $*******$

6-bit palindrome guess: $*****$**

Remaining input: $[0,0]_d, [0,2]_d 0_e$.

The subscript of c indicates that this is the third input of the string, and therefore we must guess a 1 or 2 for the 7-bit palindrome.

Let us start with the higher end. We must produce an output bit of 0, and a carry of 2. If we guess 2 for the 9-bit summand, 0 for the 8-bit, and 2 for the 7-bit, this works.

On the lower end, we know we must guess 1 for the 9-bit summand, and 2 for the 8-bit summand. This is indicated by the x_1 and y coordinates of our current state. We guessed 2 for the 7-bit summand on the higher end, and so we must guess the same for the lower end, since the 7-bit summand is aligned with the input. The only way to produce the required output bit of 1 is to guess a 2 for the lowest digit of the 6-bit summand, so that is what we do.

What is our new state? The c_1 coordinate, which is the expected higher carry from the next step is 2. The c_2 coordinate, which is the lower carry going into the next addition is also 2. Our most recent guesses for the 9-bit summand are 0 and 2, which are saved in x_1 and x_2, with x_1 being the most recent. Our higher guess for the 8-bit summand is saved in y as 2, and our lower guess for the 6-bit summand is saved in z as 2. Our destination state is thus $(2,2,0,2,0,2)$.

Step 4:

Current state: $(2, 2, 0, 2, 0, 2)$

Input symbol: $[0, 0]_d$

9-bit palindrome guess: $102 * * * 201$

8-bit palindrome guess: $20 * * * *02$

7-bit palindrome guess: $2 * * * * * 2$

6-bit palindrome guess: $2 * * * *2$

Remaining input: $[0, 2]_d 0_e$.

We guess a 1 for the higher end of the 9-bit summand, a 0 for the higher end of the 8-bit summand, and a 1 for both ends of the 7-bit summand. The 6-bit summand contributes a 2, as tracked in the z-coordinate. This produces an output bit of 0 if we have an incoming carry of 2.

On the lower end, we have an incoming carry of 2, and must use 0, 0, and 1, for the 9-bit, 8-bit, and 7-bit summands respectively. The only way to produce a 0 is to guess 0 for the 6-bit summand.

This takes us to the state $(2, 1, 1, 2, 0, 0)$.

Step 5:

Current state: $(2, 1, 1, 2, 0, 0)$

Input symbol: $[0, 2]_d$

9-bit palindrome guess: $1021 * 1201$

8-bit palindrome guess: $200 * *002$

7-bit palindrome guess: $21 * * * 12$

6-bit palindrome guess: $20 * *02$

Remaining input: 0_e.

We guess a 2 for the higher end of the 9-bit summand, a 1 for the higher end of the 8-bit summand, and a 2 for both ends of the 7-bit summand. The 6-bit summand contributes a 0, as tracked in the z-coordinate. This produces an output bit of 0 if we have an incoming carry of 1.

On the lower end, we have an incoming carry of 1, and must use 2, 0, and 2, for the 9-bit, 8-bit, and 7-bit summands respectively. The only way to produce a 2 is to guess 0 for the 6-bit summand.

This takes us to the state $(1, 1, 2, 1, 1, 0)$.

Step 6:

Current state: $(1, 1, 2, 1, 1, 0)$

Input symbol: 0_e

9-bit palindrome guess: 102121201

8-bit palindrome guess: 20011002

7-bit palindrome guess: $212 * 212$

6-bit palindrome guess: 200002

Remaining input: 0_e.

The subscript e indicates that this is the end of the input. We know we inherit a carry of 1, and must guess 1, 1, and 0, for the 9-bit, 8-bit, and 6-bit summands. We also know we must produce an output bit of 0, and a carry of 1. The only thing we can manipulate is the guess for the 7-bit summand. If it is possible to satisfy these conditions, we accept, if not, we get stuck and reject.

In this case, guessing 0 for the 7-bit summand produces an output of 0, and a carry of 1, which is what we need. We can thus take a transition to our special accepting state, q_acc, and our guess for the 7-bit palindrome becomes 2120212.

We can clearly see that all our guesses are valid base-3 palindromes. Converting from base 3 to base 10, our guesses are 8470, 4484, 1886, and 488. They add to give 15328, as needed.

Chapter 5

Results on Squares

5.1 Introduction

The folding trick that was used in Chapter 4 to prove our results regarding base-3 and base-4 palindromes also works for binary squares. Recall that a binary square is a number whose binary representation is some block repeated twice, such as $(36)_2 = 100100$, which is the block 100 twice.

This suggests a method for how we could fold the input to our finite automata when we are trying to guess binary squares as summands. The lower track starts from the least significant bit, and the upper track starts at the halfway point. Both tracks move in the more significant direction.

As with the palindromes proofs, we establish an auxiliary lemma that spells out the exact combination of lengths of squares that we try to guess. We got to this lemma by experimentally trying different combinations of lengths, and seeing how they performed on the first few million numbers.

Lemma 12.

(a) Every length-n integer, n odd, $n \geq 13$, is the sum of binary squares as follows: either

- *one of length $n-1$ and one of length $n-3$, or*
- *two of length $n-1$ and one of length $n-3$, or*
- *one of length $n-1$ and two of length $n-3$, or*

- *one each of lengths* $n-1$, $n-3$, *and* $n-5$, *or*
- *two of length* $n-1$ *and two of length* $n-3$, *or*
- *two of length* $n-1$, *one of length* $n-3$, *and one of length* $n-5$.

(b) Every length-n integer, n *even,* $n \geq 18$, *is the sum of binary squares as follows: either*

- *two of length* $n-2$ *and two of length* $n-4$, *or*
- *three of length* $n-2$ *and one of length* $n-4$, *or*
- *one each of lengths* n, $n-4$, *and* $n-6$, *or*
- *two of length* $n-2$, *one of length* $n-4$, *and one of length* $n-6$.

Lemma 12 almost immediately proves Theorem 4:

Proof. If $N < 2^{17} = 131072$, the result can be proved by a straightforward computation. There are

- 256 binary squares $< 2^{17}$;
- 19542 numbers $< 2^{17}$ that are the sum of two binary squares;
- 95422 numbers $< 2^{17}$ that are the sum of three binary squares;
- 131016 numbers $< 2^{17}$ that are the sum of four binary squares.

Otherwise $N \geq 2^{17}$, so $(N)_2$ is a binary string of length $n \geq 18$. If n is odd, the result follows from Lemma 12 (a). If n is even, the result follows from Lemma 12 (b). □

It now remains to prove Lemma 12. We do this in the next section.

5.2 Proof of Lemma 12

Proof. We prove Lemma 12 by phrasing it as a language inclusion problem. For each of the two parts of the lemma, we can build an NFA A that only accepts such folded strings if they represent numbers that are the sum of any of the combination of squares as described in the lemma. We also create an NFA, B, that accepts all valid folded representations that are sufficiently long. We then check the assertion that the language recognized by B is a subset of that recognized by A.

5.2.1 Odd-length Inputs

In order to flag certain positions of the input tape, we use an extended alphabet. Define

$$\Gamma = \{1_f\} \cup \bigcup_{\alpha \in \{a,b,c,d,e\}} \{[0,0]_\alpha, [0,1]_\alpha, [1,0]_\alpha, [1,1]_\alpha\}.$$

Let N be an integer, and let $n = 2i + 1$ be the length of its binary representation. We write $(N)_2 = 1a_{2i-1} \cdots a_1 a_0$ and fold this to produce the input string

$$[a_i, a_0]_a [a_{i+1}, a_1]_a \cdots [a_{2i-5}, a_{i-5}]_a [a_{2i-4}, a_{i-4}]_b [a_{2i-3}, a_{i-3}]_c [a_{2i-2}, a_{i-2}]_d [a_{2i-1}, a_{i-1}]_e 1_f.$$

Let A_{odd} be the NFA that recognizes those odd-length integers, represented in this folded format, that are the sum of binary squares meeting any of the 6 conditions listed in Lemma 12 (a). We construct A_{odd} as the union of several automata $A(t_{n-1}, t_{n-3}, m_a)$ and $B(t_{n-1}, t_{n-3}, t_{n-5}, m_b)$. The parameters t_p represent the number of summands of length p we are guessing. The parameters m_a and m_b are the carries that we are guessing will be produced by the first half of the summed binary squares. A-type machines try summands of lengths $n-1$ and $n-3$ only, while B-type machines include at least one $(n-5)$-length summand. We note that for the purpose of summing, guessing t binary squares is equivalent to guessing a *single* square over the larger alphabet Σ_{t+1}.

We now consider the construction of a single automaton

$$A(t_{n-1}, t_{n-3}, m) = (Q \cup \{q_{\text{acc}}, q_0, s_1\}, \Gamma, \delta, q_0, \{q_{\text{acc}}\}).$$

The elements of Q have 4 non-negative parameters and are of the form $q(x_1, x_2, c_1, c_2)$. The parameter $x_1 \leq t_{n-3}$ is the last digit of the guessed summand of length $n-3$ and $x_2 \leq t_{n-3}$ is the previous higher guess of the length-$n-3$ summand; this must be the next lower guess of this summand. We use c_1 to track the higher carry and c_2 to track the lower carry. We must have $c_1, c_2 < t_{n-1} + t_{n-3}$.

We now discuss the transition function, δ, of our NFA. In this section, we say that the sum of natural numbers μ_1 and μ_2 "produces" an output bit of $\theta \in \Sigma_2$ with a "carry" of γ if $\mu_1 + \mu_2 \equiv \theta \pmod 2$ and $\gamma = \lfloor \mu_1 + \mu_2 \rfloor$.

We allow a transition from q_0 to $q(x_1, x_2, c_1, c_2)$ on the letter $[j,k]_a$ iff there exists $0 \leq r \leq t_{n-1}$ such that $x_2 + r + m$ produces an output of j with a carry of c_1 and $x_1 + r$ produces an output of k with a carry of c_2.

We allow a transition from $q(x_1, x_2, c_1, c_2)$ to $q(x_1', x_2', c_1', c_2')$ on the letters $[j,k]_a$ and $[j,k]_b$ iff there exists $0 \leq r \leq t_{n-1}$ such that $x_2' + r + c_1$ produces an output of j with a

carry of c_1' and $x_2 + r + c_2$ produces an output of k with a carry of c_2'. Elements of Q have identical transitions on inputs with subscripts a and b. The reason we have the letters with subscript b is for B-machines, which guess a summand of length $n-5$.

There is only one letter of the input with the subscript c, and it corresponds to the last higher guess of the summand of length $n-3$. We allow a transition from $q(x_1, x_2, c_1, c_2)$ to $q(x_1', t_{n-3}, c_1', c_2')$ on the letter $[j,k]_c$ iff there exists $0 \le r \le t_{n-1}$ such that $t_{n-3} + r + c_1$ produces an output of j with a carry of c_1' and $x_2 + r + c_2$ produces an output of k with a carry of c_2'.

There is only one letter of the input with the subscript d; it corresponds to the second-last lower guess of the summand of length $n-3$. We allow a transition from $q(x_1, t_{n-3}, c_1, c_2)$ to $q(x_1', 0, c_1', c_2')$ on the letter $[j,k]_d$ iff there exists $0 \le r \le t_{n-1}$ such that $r + c_1$ produces an output of j with a carry of c_1' and $t_{n-3} + r + c_2$ produces an output of k with a carry of c_2'.

There is only one letter of the input with the subscript e; it corresponds to the last lower guess of the summand of length $n-3$. We allow a transition from $q(x_1, 0, c_1, c_2)$ to s_1 on the letter $[j,k]_e$ iff $t_{n-1} + c_1$ produces an output of j with a carry of 1 and $x_1 + t_{n-1} + c_2$ produces an output of k with a carry of m.

Finally, we add a transition from s_1 to q_{acc} on the letter 1_f.

We now consider the construction of a single automaton

$$B(t_{n-1}, t_{n-3}, t_{n-5}, m) = (P \cup Q \cup \{q_{\text{acc}}, q_0, s_1\}, \Gamma, \delta, q_0, \{q_{\text{acc}}\}).$$

The elements of P have 6 non-negative parameters and are of the form

$$q(x_1, x_2, y_1, y_3, c_1, c_2).$$

The parameter $x_1 \le t_{n-3}$ is the last digit of the guessed summand of length $n-3$ and $x_2 \le t_{n-3}$ is the previous higher guess of the length-$n-3$ summand. The parameter $y_1 \le t_{n-5}$ is the last digit of the guessed summand of length $n-5$ and $y_3 \le t_{n-5}$ is the previous higher guess of the length-$n-5$ summand. We use c_1 to track the higher carry and c_2 to track the lower carry. We must have $c_1, c_2 < t_{n-1} + t_{n-3} + t_{n-5}$.

The elements of Q have 8 non-negative parameters and are of the form

$$q(x_1, x_2, y_1, y_2, y_3, y_4, c_1, c_2).$$

The parameter $x_1 \le t_{n-3}$ is the last digit of the guessed summand of length $n-3$ and $x_2 \le t_{n-3}$ is the previous higher guess of the length-$n-3$ summand. The parameters

$y_1, y_2 \leq t_{n-5}$ are the last digit and the second-last digit of the guessed summand of length $n-5$ respectively. The parameter $y_3, y_4 \leq t_{n-5}$ are the two most recent higher guess of the length-$n-5$ summand, with y_4 being the most recent one. We use c_1 to track the higher carry, and c_2 to track the lower carry. We must have $c_1, c_2 < t_{n-1} + t_{n-3} + t_{n-5}$.

We now discuss the transition function, δ, of our NFA. We allow a transition from q_0 to $p(x_1, x_2, y_1, y_3, c_1, c_2)$ on the letter $[j,k]_a$ iff there exists $0 \leq r \leq t_{n-1}$ such that x_2+y_3+r+m produces an output of j with a carry of c_1 and $x_1 + y_1 + r$ produces an output of k with a carry of c_2.

We use a transition from $p(x_1, x_2, y_1, y_3, c_1, c_2)$ to $q(x_1, x_2', y_1, y_2', y_3, y_4', c_1', c_2')$ on the letter $[j,k]_a$ iff there exists $0 \leq r \leq t_{n-1}$ such that $x_2' + y_4' + r + c_1$ produces an output of j with a carry of c_1 and $x_2 + y_2' + r + c_2$ produces an output of k with a carry of c_2.

We use a transition from $q(x_1, x_2, y_1, y_2, y_3, y_4, c_1, c_2)$ to $q(x_1, x_2', y_1, y_2, y_4, y_4', c_1', c_2')$ on the letter $[j,k]_a$ iff there exists $0 \leq r \leq t_{n-1}$ such that $x_2' + y_4' + r + c_1$ produces an output of j with a carry of c_1 and $x_2 + y_3 + r + c_2$ produces an output of k with a carry of c_2.

We use a transition from $q(x_1, x_2, y_1, y_2, y_3, t_{n-5}, c_1, c_2)$ to $q(x_1, x_2', y_1, y_2, t_{n-5}, t_{n-5}, c_1', c_2')$ on the letter $[j,k]_b$ iff there exists $0 \leq r \leq t_{n-1}$ such that $x_2' + r + c_1$ produces an output of j with a carry of c_1 and $x_2 + y_3 + r + c_2$ produces an output of k with a carry of c_2.

We use a transition from $q(x_1, x_2, y_1, y_2, t_{n-5}, t_{n-5}, c_1, c_2)$ to $q(x_1, t_{n-3}, y_1, y_2, t_{n-5}, t_{n-5}, c_1', c_2')$ on the letter $[j,k]_c$ iff there exists $0 \leq r \leq t_{n-1}$ such that $t_{n-3} + r + c_1$ produces an output of j with a carry of c_1 and $x_2 + y_3 + r + c_2$ produces an output of k with a carry of c_2.

We use a transition from $q(x_1, t_{n-3}, y_1, y_2, t_{n-5}, t_{n-5}, c_1, c_2)$ to $q(x_1, t_{n-3}, y_1, y_2, t_{n-5}, t_{n-5}, c_1', c_2')$ on the letter $[j,k]_d$ iff there exists $0 \leq r \leq t_{n-1}$ such that $r + c_1$ produces an output of j with a carry of c_1 and $t_{n-3} + y_1 + r + c_2$ produces an output of k with a carry of c_2.

We use a transition from $q(x_1, t_{n-3}, y_1, y_2, t_{n-5}, t_{n-5}, c_1, c_2)$ to s_1 on the letter $[j,k]_e$ iff $t_{n-1} + c_1$ produces an output of j with a carry of 1 and $x_1 + y_2 + t_{n-1} + c_2$ produces an output of k with a carry of m.

Finally, we add a transition from s_1 to q_{acc} on the letter 1_f.

We now turn to verification of the inclusion assertion. We used the Automata Library toolchain of the `ULTIMATE` program analysis framework [21, 20] to establish our results. The `ULTIMATE` code proving our result can be found in the file `OddSquareConjecture.ats` at https://cs.uwaterloo.ca/~shallit/papers.html. Since the constructed machines get very large, we wrote a C++ program generating these machines, which can be found in the file `OddSquares.cpp` at https://cs.uwaterloo.ca/~shallit/papers.html.

The final machine, A_{odd}, has 2258 states. The syntax checker, B, has 8 states. We then asserted that the language recognized by B is a subset of that recognized by A. `ULTIMATE`

verified this assertion in under a minute. Since this test succeeded, the proof of Lemma 12 (a) is complete.

5.2.2 Even-length Inputs

In order to flag certain positions of the input tape, we use an extended alphabet. Define

$$\Gamma = \left(\bigcup_{\alpha \in \{a,b,c,d,e\}} \{[0,0]_\alpha, [0,1]_\alpha, [1,0]_\alpha, [1,1]_\alpha\} \right) \cup \left(\bigcup_{\beta \in \{f,g,h,i\}} \{0_\beta, 1_\beta\} \right).$$

Let N be an integer, and let $n = 2i + 4$ be the length of its binary representation. We write $(N)_2 = a_{2i+3}a_{2i+2} \cdots a_1 a_0$ and fold this to produce the input string

$$[a_i, a_0]_a [a_{i+1}, a_1]_b [a_{i+2}, a_2]_c [a_{i+3}, a_3]_c \cdots [a_{2i-3}, a_{i-3}]_c [a_{2i-2}, a_{i-2}]_d [a_{2i-1}, a_{i-1}]_e a_{2i_f} a_{2i+1_g} a_{2i+2_h} a_{2i+3_i}.$$

Let A_{even} be the NFA that recognizes the even-length integers, represented in this folded format, iff the integer is the sum of binary squares meeting any of the 4 conditions listed in Lemma 12 (b). We construct A_{even} as the union of several automata $A(t_n, t_{n-2}, t_{n-4}, t_{n-6}, m)$. The parameters t_p represent the number of summands of length p we are guessing. The parameter m is the carry that we are guessing will be produced by the first half of the summed binary squares.

We now consider the construction of a single automaton

$$A(t_n, t_{n-2}, t_{n-4}, t_{n-6}, m) = (Q \cup \{q_{\text{acc}}\}, \Gamma, \delta, q_0, \{q_{\text{acc}}\}).$$

The elements of Q have 8 non-negative parameters and are of the form

$$q(x_1, x_2, x_3, y_1, z_1, z_2, c_1, c_2).$$

The parameter x_1 is the second digit of the guessed summand of length n. The parameters x_2 and x_3 represent the previous 2 lower guesses of the length-n summand; these must be the next 2 higher guesses of this summand. The parameter y_1 represents the previous lower guess of the length-$(n-2)$ summand. We set z_1 as the last digit of the guessed summand of length $n-6$, while z_2 is the previous higher guess of this summand. Finally, c_1 tracks the lower carry, while c_2 tracks the higher carry. For any p, we must have $x_p \leq t_n$, $y_p \leq t_{n-2}$, $z_p \leq t_{n-6}$, and $c_p < t_n + t_{n-2} + t_{n-4} + t_{n-6}$. The initial state, q_0, is $q(0,0,0,0,0,0,0,0)$.

We now discuss the transition function, δ, of our NFA. Note that in our representation of even-length integers, the first letter of the input must have the subscript a, and it is the only letter to do so. We only allow the initial state to have outgoing transitions on such letters.

We allow a transition from q_0 to $q(x_1, 0, x_3, y_1, z_1, z_2, c_1, c_2)$ on the letter $[j, k]_a$ iff there exists $0 \leq r \leq t_{n-4}$ such that $x_1 + t_{n-2} + r + z_2 + m$ produces an output of j with a carry of c_2 and $x_3 + y_1 + r + z_1$ produces an output of k with a carry of c_1.

The second letter of the input must have the subscript b, and it is the only letter to do so. We allow a transition from $q(x_1, 0, x_3, y_1, z_1, z_2, c_1, c_2)$ to $q(x_1, x_3, x_3', y_1', z_1, z_2', c_1', c_2')$ on the letter $[j, k]_b$ iff there exists $0 \leq r \leq t_{n-4}$ such that $t_n + y_1 + r + z_2' + c_2$ produces an output of j with a carry of c_2' and $x_3' + y_1' + r + z_2 + c_1$ produces an output of k with a carry of c_1'.

We allow a transition from $q(x_1, x_2, x_3, y_1, z_1, z_2, c_1, c_2)$ to $q(x_1, x_3, x_3', y_1', z_1, z_2', c_1', c_2')$ on the letter $[j, k]_c$ iff there exists $0 \leq r \leq t_{n-4}$ such that $x_2 + y_1 + r + z_2' + c_2$ produces an output of j with a carry of c_2' and $x_3' + y_1' + r + z_2 + c_1$ produces an output of k with a carry of c_1'.

The letter of the input with the subscript d corresponds to the last guess of the lower half of the summand of length $n-6$, and it is the only letter to do so. We allow a transition from $q(x_1, x_2, x_3, y_1, z_1, t_{n-6}, c_1, c_2)$ to $q(x_1, x_3, x_3', y_1', z_1, 0, c_1', c_2')$ on the letter $[j, k]_d$ iff there exists $0 \leq r \leq t_{n-4}$ such that $x_2 + y_1 + r + c_2$ produces an output of j with a carry of c_2' and $x_3' + y_1' + r + t_{n-6} + c_1$ produces an output of k with a carry of c_1'.

The letter of the input with the subscript e corresponds to the last guess of both halves of the summand of length $n - 4$, and it is the only letter to do so. We allow a transition from $q(x_1, x_2, x_3, y_1, z_1, 0, c_1, c_2)$ to $q(x_1, x_3, x_3', y_1', 0, 0, 0, c_2')$ on the letter $[j, k]_e$ iff $x_2 + y_1 + t_{n-4} + c_2$ produces an output of j with a carry of c_2' and $x_3' + y_1' + t_{n-4} + z_1 + c_1$ produces an output of k with a carry of m.

We allow a transition from $q(x_1, x_2, x_3, y_1, 0, 0, 0, c_2)$ to $q(x_1, x_3, 0, 0, 0, 0, 0, c_2')$ on the letter j_f iff $x_2 + y_1 + c_2$ produces an output of j with a carry of c_2'.

We allow a transition from $q(x_1, x_2, 0, 0, 0, 0, c_2)$ to $q(x_1, 0, 0, 0, 0, 0, 0, c_2')$ on the letter j_g iff $x_2 + t_{n-2} + c_2$ produces an output of j with a carry of c_2'.

We allow a transition from $q(x_1, 0, 0, 0, 0, 0, c_2)$ to $q(0, 0, 0, 0, 0, 0, 0, c_2')$ on the letter j_h iff $x_1 + c_2$ produces an output of j with a carry of c_2'.

We allow a transition from $q(0, 0, 0, 0, 0, 0, 0, c_2)$ to q_{acc} on the letter 1_i iff $t_n + c_2$ produces an output of 1 with a carry of 0.

The final machine, A_{even} is constructed as the union of 15 automata:

- $A(0, 2, 2, 0, m)$, varying m from 0 to 3
- $A(0, 3, 1, 0, m)$, varying m from 0 to 3
- $A(1, 0, 1, 1, m)$, varying m from 0 to 2
- $A(0, 2, 1, 1, m)$, varying m from 0 to 3

We now turn to verification of the inclusion assertion. The `ULTIMATE` code proving our result can be found in the file `EvenSquareConjecture.ats` at https://cs.uwaterloo.ca/~shallit/papers.html. Since the constructed machines get very large, we wrote a C++ program generating these machines, which can be found in the file `EvenSquares.cpp` at https://cs.uwaterloo.ca/~shallit/papers.html.

The final machine, A_{even}, has 1343 states. The syntax checker, B, has 12 states. We then asserted that the language recognized by B is a subset of that recognized by A. `ULTIMATE` verified this assertion in under a minute. Since this test succeeded, the proof of Lemma 12 (b) is complete.

□

Corollary 13. *Given an integer $N > 686$, we can find an expression for N as the sum of four binary squares in time linear in $\log N$.*

Proof. For $N < 131072$, we do this with a brute-force search. Otherwise we construct the appropriate automaton A (depending on whether the binary representation of N has either even or odd length). Now carry out the usual direct product construction for intersection of languages on A and B, where B is the automaton accepting the folded binary representation of N. The resulting automaton has at most $c \log N$ states and transitions. Now use the usual depth-first search of the transition graph to find a path from the initial state to a final state. The labels of this path gives the desired representation. □

5.3 Checking Correctness

We tested our machine by calculating those integers of length 8 that can be expressed as the sum of up to 3 binary squares of length 4, and up to 4 binary squares of length 6. We then used the `ULTIMATE` framework to test that those length-8 integers are accepted

by our machine, but all others are rejected. The code running this test can be found as `Minus2Minus4SquareConjecture − Test1` at https://cs.uwaterloo.ca/~shallit/papers.html.

We also tested the machine by calculating those integers of length 10 that can be expressed as the sum of up to to 2 binary squares of length 6, and up to 4 binary squares of length 8. We then built the analogous machine and confirmed that these length-10 integers are accepted, but all others are rejected. We then repeated this test for those integers of length 10 that can be expressed as the sum of up to to 3 binary squares of length 6, and up to 3 binary squares of length 8. The code running these tests can be found as `Minus2Minus4SquareConjecture − Test2` and `Minus2Minus4SquareConjecture − Test3` at https://cs.uwaterloo.ca/~shallit/papers.html.

Further testing will involve calculating the analogous machine that accepts words that are the sum of fewer binary squares (say, up to 3 binary squares of length $n-4$ and 3 of length $n-2$) and asserting that this smaller machine accepts those input integers that are so expressible and rejects all others.

5.4 Other Results on Squares

Our technique can be used to obtain other results in additive number theory. The results presented in this section were obtained by making fairly minor changes to the proof scripts used to establish Theorem 4. This highlights one of the strengths of our method, because obtaining these results using classical number theory may have required a very different proof technique. In a sense, we get these similar results for "free" after we have proved Theorem 4.

Recently, Crocker [11] and Platt & Trudgian [34] studied the integers representable as the sum of two ordinary squares and two powers of 2. This led us to establish Theorem 5, which we do so with the help of the following lemma.

Lemma 14.

(a) Every length-n integer, n odd, $n \geq 7$, is the sum of at most two powers of 2 and either:

- *at most two squares of length $n-1$, or*
- *at most one square of length $n-1$ and one of length $n-3$.*

(b) Every length-n integer, n even, $n \geq 10$, is the sum of at most two powers of 2 and either:

- *at most one square of length n and one of length $n-4$, or*
- *at most one square of length $n-2$ and one of length $n-4$.*

Proof. We use a similar proof strategy as before. The `ULTIMATE` code proving our result can be found in the files `OddSquarePowerConjecture.ats` and `EvenSquarePowerConjecture.ats` at https://cs.uwaterloo.ca/~shallit/papers.html. The generators can be found as `OddSquarePower.cpp` and `EvenSquarePower.cpp` at https://cs.uwaterloo.ca/~shallit/papers.html.

The final machines for the odd-length and even-length cases have 806 and 2175 states respectively. The language inclusion assertions all hold. This concludes the proof.

□

With this lemma, we can prove our result.

Proof. (of Theorem 5) For $N < 512$, the result can be easily verified. Otherwise, we use Lemma 14 (a) if N is an odd-length binary number and Lemma 14 (b) if it is even. □

We also consider the notion of *generalized binary squares*. A number N is called a generalized binary square if one can concatenate zero or more leading 0s to its binary representation to produce a binary square. As an example, 9 is a generalized square, since 9 in base 2 is 1001, which can be written as $001001 = (001)(001)$. The first few generalized binary squares are

$$0, 3, 5, 9, 10, 15, 17, 18, 27, 33, 34, 36, 45, 51, 54, 63, \ldots;$$

they form sequence A175468 in the OEIS [40].

In what follows, when we refer to the length of a generalized square, we mean the length including the leading zeros. Thus, 9 is a generalized square of length 6 (and not 4).

Lemma 15.

(a) Every length-n integer, $n \geq 7$, n odd, is the sum of 3 generalized squares, of lengths $n+1$, $n-1$, and $n-3$.

(b) Every length-n integer, $n \geq 8$, n even, is the sum of 3 generalized squares, of lengths n, $n-2$, and $n-4$.

Proof. We use a very similar proof strategy as in the proof of Lemma 12. We drop the requirement that the most significant digit of our guessed squares be 1, thus allowing for generalized squares. Note that the square of length $n+1$ must in part (a) must start with a 0.

The `ULTIMATE` code proving our result can be found in the files `OddGenSquareConjecture.ats` and `EvenGenSquareConjecture.ats` at https://cs.uwaterloo.ca/~shallit/papers.html. The generators can be found as `OddGeneralizedSquares.cpp` and `EvenGeneralizedSquares.cpp` at https://cs.uwaterloo.ca/~shallit/papers.html. The final machines for the odd-length and even-length cases have 132 and 263 states respectively. □

We thus have the following theorem:

Theorem 16. *Every natural number $N > 7$ is the sum of 3 generalized squares.*

Proof. For $7 < N < 64$ the result can be easily verified. Otherwise, we use Lemma 15 (a) is an odd-length binary number and Lemma 15 (b) if it is even. □

Chapter 6

Some Results from Walnut

6.1 Automatic Sequences and Walnut

In this chapter, we present a series of small results on additive number theory. Some of the bases considered in this chapter relate to famous sequences like the Thue-Morse sequence and the Rudin-Shapiro sequence. Others relate to classes of numbers like the evil numbers and odious numbers.

These results are established by framing them as properties of automatic sequences. Recall that the states of a deterministic finite automaton (DFA) are partitioned into accepting states (which we can think of as outputting 1) and rejecting states (which output 0). We can extend this idea to use a larger output alphabet, Δ, and define a mapping function $\tau\colon Q \mapsto \Delta$, where Q is the set of states of our DFA. If a finite automaton reads an input string w and ends in the state q, then its output on the string w is $\tau(q)$.

A sequence $(a_n)_{n\geq 0}$ over the alphabet Δ is said to be a *k-automatic sequence* if we can build a DFA such that its output on $(i)_k$ is a_i for all $i \geq 0$. For more on automatic sequences, please consult, e.g. [1].

We express our results as properties of automatic sequences in *Presburger arithmetic*, which is a decidable theory of the natural numbers with addition, equality, and first-order logic. First-order logic includes the quantifiers "there exists" and "for all" and logical connectives such as "and" and "or". For more on Presburger arithmetic, please consult, e.g. [18].

All of our results in this section are established using the automatic theorem-proving software `Walnut` [30]. `Walnut` implements decision procedures to decide various combina-

torial properties of automatic words. In a recent paper [24], Jason Bell, Kathryn Hare, and Jeffrey Shallit characterize the k-automatic sequences that form additive bases for the natural numbers. They also provide an algorithm to determine the smallest possible order for a k-automatic additive basis. Their results are experimentally verifiable in `Walnut` in a manner similar to that described in this chapter.

6.2 The Approach

To see how the results in this chapter were established, let us take a close look at one of the theorems. This result concerns the Thue-Morse sequence and the Rudin-Shapiro sequence, which are well-known automatic sequences that can be written on the binary alphabet. Please see [1] for more on these sequences. We refer to the value of the Thue-Morse sequence at the kth position as T_k, and that of the Rudin-Shapiro sequence as R_k.

First, we state the result.

Theorem 17. *Every natural number $N > 9$ is the sum of*

- *a number l with $T_l = 1$, and*
- *a number m with $R_m = 1$.*

Proof. The first step in the proof is to build the finite automata that output the Thue-Morse and Rudin-Shapiro sequences.

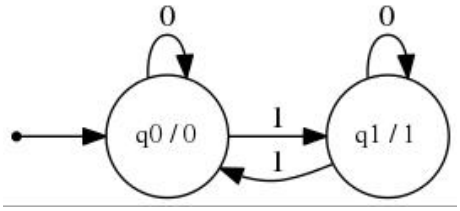

Figure 6.1: The automaton outputting the Thue-Morse word

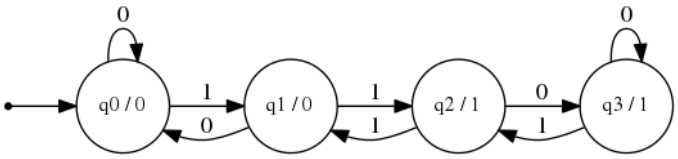

Figure 6.2: The automaton outputting the Rudin-Shapiro word

It is worth noting that these automata are prepackaged in an installation of `Walnut`.

The next step is to write out the `Walnut` command that accepts all integers in base-2 that can be written as the sum of integers l and m such that $T_l = 1 = R_m$. We indicate that we are working in base-2 by supplying `?msd_2` in our command. We wish to accept an input integer $i > 9$ if there exists integers l and m satisfying some conditions. We denote "there exists integers l and m" with the clause `E l, m`. We trivially accept if the input is less than 10, which we denote `i < 10`.

We need l and m to satisfy three conditions:

(a) They must sum to i, which is denoted by the clause `l + m = i`

(b) The Thue-Morse word must have a 1 at position l, denoted `T[l] = @1`

(c) The Rudin-Shapiro word must have a 1 at position m, denoted `RS[m] = @1`

We string this all together to get the following `Walnut` command:

```
eval testTMRS "?msd_2 (E l, m (l + m = i) & (T[l] = @1) & (RS[m] = @1))
| (i < 10)":
```

The final step in our proof is to run the command and see the automaton that `Walnut` produces. When we run this command, we get the following result automaton.

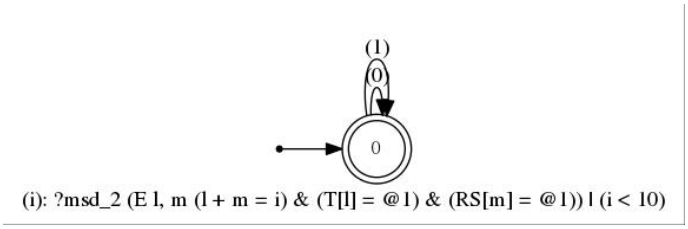

Figure 6.3: The result automaton

Since this automaton accepts all integers i, our result is proven.

□

6.3 Results on the Fibonacci word

The infinite Fibonacci word, **f**, is the fixed point of the morphism $0 \mapsto 01, 1 \mapsto 0$. It is found as sequence A003849 in the OEIS [40]. The first few digits of **f** are

$$0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, \ldots$$

The infinite Fibonacci word can be indexed as `F[i]` on `Walnut`. For more information about **f**, please see [1]. The positions of 1s in the infinite Fibonacci word are given by sequence A003622 in the OEIS. The first few positions of the 1s in **f** are

$$1, 4, 6, 9, 12, 14, 17, 19, 22, 25, 27, 30, 33, 35, 38, 40, \ldots$$

This should not be confused the better-known Fibonacci series, which is sequence A000045 on the OEIS. Zeckendorf's theorem [19] shows that every positive integer has a unique representation in a numeration system based on the Fibonacci numbers. To prove theorems in this numeration system on `Walnut`, the tag `?msd_fib` is supplied.

Just as powers of 2 have exactly one 1 in the binary numeration system, the Fibonacci numbers have exactly one 1 in their Fibonacci representation. We built a simple automaton that outputs 1 when the input is a Fibonacci number in the Fibonacci numeration system, and we index this automaton as `FIBS[i]`.

With this, we can prove the following theorems.

Theorem 18. *Every natural number is the sum of up to three terms from A003622 (positions where* **f** *has a 1).*

Proof. We run `Walnut` with the following command:

```
eval testThreeFibs "?msd_fib E x,y,z (x+y+z = i) & ( (x = 0) | (F[x] = @1) )
& ( (y=0) | (F[y] = @1)) & ((z=0) | (F[z] = @1))":
```

The resulting automaton accepts all inputs. Note that the inputs to the machine are the Fibonacci representation of integers. □

Theorem 19. *Every natural number is the sum of up to two terms from A003622 or is F_{n-2} (or both).*

Proof. We run `Walnut` with the following command:

```
eval testTwoFibs "?msd_fib (E x,y (x+y = i) & ( (x = 0) | (F[x] = @1) ) & (F[y]
= @1)) | (Et (t - 2 = i) & (FIBS[t] = @1))":
```

The resulting automaton accepts all inputs. Note that the inputs to the machine are the Fibonacci representation of integers. □

Theorem 20. *Every natural number $N > 3$ is the sum of up to three terms from A003622, at least 2 of which are unequal.*

Proof. We run `Walnut` with the following command:

```
eval testThreeFibsAlt2 "?msd_fib (i < 4) | (E x,y,z (x+y+z = i) & (x!=y) & (
(x = 0) | (F[x] = @1) ) & ( (y=0) | (F[y] = @1)) & ((z=0) | (F[z] = @1)))":
```

The resulting automaton accepts all inputs. Note that the inputs to the machine are the Fibonacci representation of integers. □

Theorem 21. *Every natural number $N > 12$ is the sum of either two or three distinct terms from A003622.*

Proof. We run `Walnut` with the following command:

```
eval testThreeFibsAlt2 "?msd_fib (i < 13) | (E x,y,z (x+y+z = i) & (x!=y) &
(y!=z) & (x!=z) & ( (x = 0) | (F[x] = @1) ) & ((F[y] = @1)) & ((F[z] = @1)))":
```

The resulting automaton accepts all inputs. Note that the inputs to the machine are the Fibonacci representation of integers. □

6.4 Evil and Odious Numbers

A natural number is called an *evil number* if it has an even number of 1s is in its binary representation. The position of 0 in the Thue-Morse word is precisely the set of evil numbers. Numbers that are not evil are called *odious numbers* and they correspond to 1s in the Thue-Morse word.

We can use `Walnut` to prove results on the additive properties of these numbers.

Theorem 22. *If a natural number is not the sum of two evil numbers, it is either 2, 4 or of the form* $2 \cdot 4^n - 1$ *for* $n \geq 0$.

Proof. We build a simple automaton that outputs 1 if a number is of the form $4^n - 1$, and we index this automaton as `X[i]`. We then run `Walnut` with the following command:

```
eval testTwoEvils "?msd_2  (E x,y (x+y = i) & (T[x] = @0) & (T[y] = @0)) =>
((i=2) | (i=4) | (X[i] = @1))":
```

The resulting automaton accepts all inputs. □

Theorem 23. *Every natural number* $N > 7$ *is the sum of 3 evil numbers, at least 2 of which are unequal.*

Proof. We run `Walnut` with the following command:

```
eval testThreeEvilsAlt "?msd_2 (i < 8) | (Ex, y, z (x != y) & (x+y+z = i) &
(T[x] = @0) & (T[y] = @0) & (T[z] = @0))":
```

The resulting automaton accepts all inputs. □

Theorem 24. *Every natural number* $N > 10$ *is the sum of 3 distinct evil numbers.*

Proof. We run `Walnut` with the following command:

```
eval testThreeEvilsAlt2 "?msd_2 (i < 11) | (Ex, y, z (x != y) & (y!=z) & (x!=z)
& (x+y+z = i) & (T[x] = @0) & (T[y] = @0) & (T[z] = @0))":
```

The resulting automaton accepts all inputs. □

Theorem 25. *Every natural number* $N > 15$ *is the sum of 3 distinct odious numbers.*

Proof. We run `Walnut` with the following command:

```
eval testThreeOdious "?msd_2 (i < 16) | (Ex, y, z (x != y) & (y!=z) & (x!=z)
& (x+y+z = i) & (T[x] = @1) & (T[y] = @1) & (T[z] = @1))":
```

The resulting automaton accepts all inputs. □

We can also consider numbers with an even number of 0s in their binary representations. We term such numbers *evening numbers* for the purpose of these results, and we build an automaton that outputs 1 to accept them. We index this automaton as `V[i]`.

Theorem 26. *Every natural number* $N > 4$ *is the sum of 3 evening numbers.*

Proof. We run `Walnut` with the following command:

```
eval testThreeEvenings "?msd_2 (i < 5) | (Ex, y, z (x+y+z = i) & (V[x] = @1)
& (V[y] = @1) & (V[z] = @1))":
```

The resulting automaton accepts all inputs. □

Theorem 27. *Every natural number $N > 10$ is the sum of 3 distinct evening numbers.*

Proof. We run `Walnut` with the following command:

```
eval testThreeEveningsAlt "?msd_2 (i < 11) | (Ex, y, z (x+y+z = i) & (x!=y)
& (y!=z) & (x!=z) & (V[x] = @1) & (V[y] = @1) & (V[z] = @1))":
```

The resulting automaton accepts all inputs. □

Chapter 7

Discussion

7.1 Limitations of this Proof Technique

While we hope that the reader is excited about the potential of this kind of machine-based proof, there are some limitations to the approach. We ran into a few of these limitations during the work that produced this thesis.

7.1.1 Time and Space Complexity

Perhaps the most fundamental limitation that exists is the very high upper bound on the time complexity of the algorithms that determinise NWAs and NFAs. Recall that an NFA of n states can have up to 2^n states when determinised, and a nondeterministic NWA of n states can have up to 2^{n^2} states. In some sense, it is a matter of luck that these operations ran to completion to establish our results. However, there were operations that timed out. For example, we initially attempted to prove Theorems 2 and 3 with the use of NWAs, but these operations did not run to completion, even on high performance computers.

We also considered the matter of antipalindromes. The bitwise complement of a binary string x is the string obtained by flipping all 0s and 1s, and is denoted $\overline{x}$. A binary string is an *antipalindrome* if its reverse is equal to its bitwise complement. For example, $(52)_2 = 110100$ satisfies $(110100)^R = 001011 = \overline{110100}$.

We call an integer a *binary antipalindrome* if its canonical base-2 representation is an antipalindrome. The binary antipalindromes form sequence A035928 in the OEIS. We have the following conjecture about the additive properties of antipalindromes.

Conjecture 28. Every even integer of length n, for n odd and $n \geq 9$, is the sum of at most 10 antipalindromes of length $n-3$.

We built the NWA that would prove this conjecture, but it did not run to completion.

7.1.2 Limitations of the Machines

There are also some restrictions to this approach that arise due to the nature of automata themselves. A good example of this is that we were restricted to adding palindromes and squares of roughly the same length. We cannot think of a way to modify these machines to consider summands of arbitrary lengths. For example, we can see no way to try one summand of length n and another of length $\frac{n}{2}$.

There are also classes of numbers that we thought we might be able to study with this approach, but could not make it work. The *equinumerous numbers* are integers whose binary representation has an equal number of 0s and 1s. They form sequence A031443 on the OEIS. Experimental evidence suggests that these numbers form an asymptotic additive basis of order 3, but we cannot think of a way to prove it using either NFAs or NWAs. Any effort to track the number of 0s guessed in the current state fails because that number can get arbitrarily large, while the number of states in a finite automata must be finite.

7.2 Future Work on Palindromes

In personal communication with my supervisor, Dr. Jeffrey Shallit, in February 2018, Julien Cassaigne showed that there are infinitely many even numbers that are not the sum of two binary palindromes. Since all binary palindromes are odd, this means that there are infinitely many numbers for which the bound of 4 established in Theorem 1 is optimal.

We can use the same sorts of ideas to attack other problems in additive number theory. For example, we can obtain results about "generalized palindromes" (allowing an arbitrary number of leading zeroes), anti-palindromes, "generalized anti-palindromes", and so forth.

7.3 Future Work on Squares

We do not currently know whether the number 4 in Theorem 4 is optimal, although numerical evidence strongly suggests that it is.

Numerical evidence suggests the following two conjectures:

Conjecture 29. Let α_3 denote the lower asymptotic density of the set S_3 of natural numbers that are the sum of three binary squares. Then $\alpha_3 < 0.9$.

We could also focus on sums of *positive* binary squares. (For the analogous problem dealing with ordinary squares, see, e.g., [16, Chapter 6].) It seems likely that our method could be used to prove the following result.

Conjecture 30. Every natural number > 1772 is the sum of exactly four positive binary squares. There are 112 exceptions, given below:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 25, 27, 28, 29, 30, 32, 34, 35, 37,$$
$$39, 41, 42, 44, 46, 47, 49, 51, 53, 56, 58, 62, 65, 67, 74, 83, 88, 95, 100, 104, 107, 109, 113, 116, 122, 125,$$
$$131, 134, 140, 143, 148, 149, 155, 158, 160, 161, 167, 170, 173, 175, 182, 184, 368, 385, 402, 407, 419,$$
$$424, 436, 441, 458, 475, 492, 509, 526, 543, 552, 560, 569, 587, 599, 608, 613, 620, 625, 638, 647, 653, 671,$$
$$686, 698, 713, 1508, 1541, 1574, 1607, 1640, 1673, 1706, 1739, 1772.$$

Other interesting things to investigate include estimating the number of distinct representations of N as a sum of four binary squares, both in the case where order matters and where order does not matter. These are sequences A290335 and A298731 in the OEIS, respectively.

In recent work [25], my supervisor, Dr. Jeffrey Shallit and his co-authors have proven, using a combinatorial and number-theoretic approach, that the binary k'th powers form an asymptotic basis of finite order for the multiples of $\gcd(k, 2^k - 1)$. However, the constant obtained thereby is rather large.

References

[1] J-P Allouche and J. Shallit. *Automatic Sequences*. Cambridge, 2003.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. Thirty-Sixth Ann. ACM Symp. Theor. Comput. (STOC)*, pages 202–211. ACM, 2004.

[3] R. Alur and P. Madhusudan. Adding nested structure to words. *J. Assoc. Comput. Mach.*, 56:Art. 16, 2009.

[4] D. Baczkowski. Applications of the hardy-littlewood circle method. Available at https://pdfs.semanticscholar.org/2cb4/62f411c6c979a45e88dea8c6960dba9e1e3b.pdf, 2008.

[5] W. D. Banks. Every natural number is the sum of forty-nine palindromes. *INTEGERS — Electronic J. Combinat. Number Theory*, 16, 2016. #A3.

[6] W. D. Banks and I. E. Shparlinski. Prime divisors of palindromes. *Period. Math. Hung.*, 51:1–10, 2005.

[7] W. D. Banks and I. E. Shparlinski. Average value of the Euler function on binary palindromes. *Bull. Pol. Acad. Sci. Math.*, 54:95–101, 2006.

[8] B. von Braunmühl and R. Verbeek. Input-driven languages are recognized in $\log n$ space. In M. Karpinski, editor, *Foundations of Computation Theory, FCT 83*, volume 158 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, 1983.

[9] J. Cilleruelo and F. Luca. Every positive integer is a sum of three palindromes. Preprint available at https://arxiv.org/abs/1602.06208v1, 2016.

[10] J. Cilleruelo, F. Luca, and L. Baxter. Every positive integer is a sum of three palindromes. Preprint available at https://arxiv.org/abs/1602.06208v2, 2017.

[11] R. C. Crocker. On the sum of two squares and two powers of k. *Colloq. Math.*, 112:235–267, 2008.

[12] X. Droubay. Palindromes in the Fibonacci word. *Inform. Process. Lett.*, 55:217–221, 1995.

[13] X. Droubay and G. Pirillo. Palindromes and Sturmian words. *Theoret. Comput. Sci.*, 223:73–85, 1999.

[14] P. W. Dymond. Input-driven languages are in $\log n$ depth. *Inform. Process. Lett.*, 26:247–250, 1988.

[15] W. J. Ellison. Waring's problem. *Amer. Math. Monthly*, 78:10–36, 1971.

[16] E. Grosswald. *Representations of Integers as Sums of Squares*. Springer-Verlag, 1985.

[17] Y.-S. Han and K. Salomaa. Nondeterministic state complexity of nested word automata. *Theoret. Comput. Sci.*, 410:2961–2971, 2009.

[18] J. R. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge, 2009.

[19] M. Hazewinkel. Zeckendorf representation. *Encyclopedia of Mathematics*, 1994.

[20] M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski. Ultimate automizer with two-track proofs. In M. Chechik and J.-F. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems — 22nd International Conference, TACAS 2016*, volume 9636 of *Lecture Notes in Computer Science*, pages 950–953. Springer-Verlag, 2016.

[21] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In N. Sharygina and H. Veith, editors, *Computer Aided Verification — 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 2013.

[22] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

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

[24] J. Shallit J. Bell, K. Hare. When is an automatic set an additive basis? Available at https://arxiv.org/pdf/1710.08353.pdf, 2018.

[25] D. M. Kane, C. Sanna, and J. Shallit. Waring's theorem for binary powers. Preprint available at https://arxiv.org/abs/1801.04483, 2018.

[26] S. La Torre, M. Napoli, and M. Parente. On the membership problem for visibly pushdown languages. In S. Graf and W. Zhang, editors, *ATVA 2006*, volume 4218 of *Lecture Notes in Computer Science*, pages 96–109. Springer-Verlag, 2006.

[27] P. Madhusudan, D. Nowotka, A. Rajasekaran, and J. Shallit. Lagrange's theorem for binary squares. Preprint available at https://arxiv.org/abs/1710.04247, 2017.

[28] K. Mehlhorn. Pebbling mountain ranges and its application of DCFL-recognition. In *Proc. 7th Int'l Conf. on Automata, Languages, and Programming (ICALP)*, volume 85 of *Lecture Notes in Computer Science*, pages 422–435. Springer-Verlag, 1980.

[29] C. J. Moreno and S. S. Wagstaff, Jr. *Sums of Squares of Integers*. Chapman and Hall/CRC, 2005.

[30] H. Mousavi. Automatic theorem proving in `Walnut`. Preprint available at https://arxiv.org/abs/1603.06017, 2016.

[31] M. B. Nathanson. *Additive Number Theory: The Classical Bases*. Springer-Verlag, 1996.

[32] A. Okhotin and K. Salomaa. State complexity of operations on input-driven pushdown automata. *J. Comput. System Sci.*, 86:207–228, 2017.

[33] X. Piao and K. Salomaa. Operational state complexity of nested word automata. *Theoret. Comput. Sci.*, 410:3290–3302, 2009.

[34] D. Platt and T. Trudgian. On the sum of two squares and at most two powers of 2. Preprint, available at https://arxiv.org/abs/1610.01672, 2016.

[35] A. Rajasekaran, J. Shallit, and T. Smith. Sums of palindromes: an approach via nested-word automata. Preprint available at http://arxiv.org/abs/1706.10206, 2017.

[36] K. Salomaa. Limitations of lower bound methods for deterministic nested word automata. *Inform. Comput.*, 209:580–589, 2011.

[37] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge, 2008.

[38] T. N. Shorey. On the equation $z^q = (x^n - 1)/(x - 1)$. *Indag. Math.*, 48:345–351, 1986.

[39] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013.

[40] N. J. A. Sloane. The on-line encyclopedia of integer sequences. Available at https://oeis.org, 2016.

[41] C. Small. Waring's problem. *Math. Mag.*, 50:12–16, 1977.

[42] R. C. Vaughan. *The Hardy-Littlewood Method*. Cambridge, 1981.

[43] R. C. Vaughan and T. Wooley. Waring's problem: a survey. In M. A. Bennett, B. C. Berndt, N. Boston, H. G. Diamond, A. J. Hildebrand, and W. Philipp, editors, *Number Theory for the Millennium. III*, pages 301–340. A. K. Peters, 2002.

[44] Y. Wang. *Goldbach Conjecture*. World Scientific, 1984.

[45] S. Yates. The mystique of repunits. *Math. Mag.*, 51:22–28, 1978.

APPENDICES

Appendix A

Walnut automata files

We present the text of those automata used in Chapter 6 that do not come included in the standard installation of `Walnut`.

A.1 Fibonacci Numbers

This automaton only outputs 1 if the input is the Fibonacci representation of a Fibonacci number. It checks if the first digit is a 1, and no other digits are 1.

```
msd_fib

0 0
0 -> 0
1 -> 1

1 1
0 -> 1
```

A.2 Numbers of the Form $4^n - 1$

This automaton only outputs 1 if the input is the binary representation of an integer of the form $4^n - 1$. Such numbers have an even number of 1s and no 0s in their binary representation.

```
msd_2

0 0
1 -> 1

1 1
1 -> 0
```

A.3 Evening Numbers

This automaton only outputs 1 if the input is the binary representation of an evening number, that is, its binary representation has an even number of 0s.

```
msd_2

0 0
1 -> 1

1 1
0 -> 2
1 -> 1

2 0
0 -> 1
1 -> 2
```