Narayana in Walnut

How to prove properties of Narayana's cow sequence in Walnut

A. A. Mendels

TU Delft

Narayana in Walnut

How to prove properties of Narayana's cow sequence in Walnut

by

A. A. Mendels

to obtain the degree of Bachelor of Science
at the Delft University of Technology,

Student number: 5835402
Project duration: May 1, 2025 – August 8, 2025
Thesis committee: Prof. dr. R. J. Fokkink, TU Delft, supervisor
Dr. ir. T. M. L. Janssen, TU Delft

TU Delft

Preface

This thesis explores Narayana's sequence, focusing on methods for proving some of its properties using the automated theorem prover, Walnut. The research presented here builds heavily upon the foundational work in this field by Jeffrey O. Shallit, whose contributions made this investigation possible. The material is intended for an audience with a background in mathematics and some experience with programming concepts.

This report has been submitted to obtain the degree of Bachelor of Science from the Department of Applied Mathematics at Delft University of Technology. The research and writing took place from May 2025 to August 2025, under the supervision of Robbert Fokkink. I wish to extend my sincere gratitude to Robbert Fokkink for his essential supervision and support.

A. A. Mendels
Delft, August 2025

Abstract

This thesis investigates properties of the Narayana sequence, a combinatorial sequence with recursive and morphic structure. We prove several known characteristics of the sequence. We then introduce the Walnut Prover, a tool for automating logical reasoning over automatic sequences. We describe its capabilities and limitations, and apply it to explore additional properties of the Narayana sequence.

Contents

1

Narayana's Cow Sequence

Yathā śikhā mayūrāṇā, nāgānā maṇayo yathā
tathā vedāṅgaśāstrāṇā, gaṇita mūrdhani sthitam.

Like the crest of a peacock, like the gem on the head of a snake, so is mathematics at the head of all knowledge.

— Vedanga Jyotisa

1.1. Introduction

This chapter provides a detailed introduction into Narayana's cow sequence. The sequence shares some traits with the Fibonacci numbers, where its third-order recurrence relation makes it a compelling subject for analysis.

We will begin by exploring the historical context of the sequence before deriving and proving several of its key characteristics. This includes its Golden Ratio, unique Narayana Representation and its generating function. The properties established here will be key to understand the sequence and will be highly beneficial for understanding chapter 3. There, Narayana's sequence will serve as our primary example for demonstrating the functionality of the Walnut theorem prover.

1.2. Narayana Pandita

While the Fibonacci sequence is practically a household name, the Narayana sequence and its creator often fly under the radar, despite their significant contributions to mathematics. The sequence is named after Narayana Pandita, a prominent Indian mathematician from the 14th century whose work represents a high point in medieval Indian mathematics.

Thanks to a note in his own writings, we know that he finished his most important work, the *Ganita Kaumudi*, in the year 1356 AD [1]. This masterwork, with a title that translates to the 'Moonlight of Mathematics,' was a huge and detailed book on arithmetic. It is in here that Narayana Pandita defines what he calls an "additive sequence".

Across his works, his most notable contributions include:

Narayana's Cows Sequence: This is the puzzle that made him famous in modern times. Posing a recreational problem about a growing herd of cows which was quite similar to Fibonacci's well-known rabbit problem, earned him the name to this sequence.

Combinatorics: Narayana Pandita's work in combinatorics was particularly advanced for his time. He came up with ways to generate permutations and solve problems involving combinations. His work on what we now call Pascal's Triangle (known in India as *Meru Prastaara*) was especially sophisticated [2].

1.3. A Richer History of Mathematics in India

It is worth remembering that India has a rich history as a frontrunner in mathematical discovery. Many findings that were later attributed to European mathematicians were, in fact, already known in other civilizations, including India, centuries earlier. The Fibonacci sequence is a perfect example of this.

In his historical analysis, Parmanand Singh points out that, "What are generally referred to as the Fibonacci numbers and the method for their formation were given by Virahanka (between A.D. 600 and 800), Gopala (prior to A.D. 1135) and Hemacandra (c. A.D. 1150), all prior to L. Fibonacci (c. A.D. 1202)" [1].

Another measure which shows how mathematically advanced India was in this area, is that the *Ganita Kaumudi* was the first work in which the ideas of the Fibonacci numbers were developed further.

1.4. Definition of Narayana's Cow Sequence

Numeral sequences are ordered lists where numbers follow one another according to some rule or pattern. *Recursive sequences* fall into this category, where each term in the sequence is determined by one or more preceding terms. Fibonacci's and Narayana's cow sequence are examples of this.

Narayana Pandit, in his *Ganita Kaumudi*, described the problem leading to his sequence as follows:

> A cow produces one calf every year. Beginning in its fourth year, each matured cow produces one calf at the beginning of each year. How many calves are there altogether after 20 years?

It assumes that in the n^{th} year there are N_n cows. Now, the definition for Narayana's sequence is:

Definition 1.1. *(Narayana's Cow Sequence). We define Narayana's cow sequence recursively by* $N_n = N_{n-1} + N_{n-3}$ *for all* $n \geq 3$*, starting from the initial conditions* $N_0 = 0, N_1 = 1$*, and* $N_2 = 1$.

This gives us the following sequence:

$$0, 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, \ldots$$

Narayana's sequence clearly shares numerous similarities with the Fibonacci sequence, yet it remains far less known. This leads to the question: what are the applications and properties of Narayana's sequence that make it worthy of study?

Interestingly, this sequence and its defining recurrence appear in various contexts, such as combinatorics and theoretical computer science. For example, as Donald Knuth discusses in his work, *The Art of Computer Programming*, such linear recurrence relations often govern the number of ways to construct certain objects—such as tiling a path with tiles of specific lengths, or decomposing a number into a sum of prescribed parts [3].

The analysis of certain recursive algorithms, whose structure is similar to these combinatorial problems, can therefore lead directly to sequences like Narayana's. A deeper understanding of the sequence's properties can thus provide insights into the efficiency and behavior of these algorithms.

Furthermore, its study helps understanding the broader family of linear recurrence sequences in general. This can help identify new applications or recognize the underlying structure of these sequences in places where they were not previously known to apply.

1.5. Binary Code Based on Narayana's Sequence

Transforming Narayana's cow sequence such that it forms the basis for a binary code system can facilitate many proofs and insights. Computers are after all way more efficient in processing binary data (sequences of 0's and 1's) compared to other numerical forms.

Binary code, in essence, implies that a sequence of 0's and 1's signifies a particular entity, which could be a number, an action, or a letter. All non-negative integers can, for example, be signified by the base-2 binary code, as shown for the first few integers in Table 1.1.

Table 1.1: Standard base-2 (binary) representation of the first few integers.

Decimal Number	Standard Binary String
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

The objective now is to establish a binary code system where non-negative integers can be represented using terms derived from Narayana's numbers with simple 0's and 1's. Such a system is only considered valid if every number possesses a unique binary code representation. For this uniqueness to hold, certain rules must be adhered to when selecting terms for the sum.

A representation system based on Narayana's numbers can be formed by following a clear set of steps. The goal is to find a unique way to represent any non-negative integer as a binary string, using the terms of the Narayana sequence as our building blocks.

Step 1: Understand the Basic Principle The idea is to represent a nonnegative number as a sum of unique terms from Narayana's sequence. In our binary code, a '1' at a certain position will signify that the basis number corresponding to that position is included in the sum. A '0' will signify that it isn't.

To build a proper numeral system, however, we first need a well-defined and strictly increasing set of these basis values. The original Narayana sequence begins $N_0 = 0, N_1 = 1, N_2 = 1, N_3 = 1, \ldots$. The repeated '1's sabotage uniqueness in binary code. Therefore, we define a Narayana Basis:

Definition 1.2. *(Narayana Basis). The Narayana Basis denoted $\{B_k\}_{k\geq 1}$, is the sequence of Narayana numbers starting from index 3:*

$$\{B_k\}_{k\geq 1} = \{N_3, N_4, N_5, N_6, \ldots\} = \{1, 2, 3, 4, 6, 9, \ldots\}$$

The k'th term in B_k has Narayana representation whith a '1' in the k-th position, and for the rest 0's in our binary code. The (Narayana) number zero is handled as a special case, represented by the string '0'.

Table 1.2 shows the resulting binary code for the first few Narayana (Basis) numbers themselves.

Table 1.2: Narayana binary representation for the first few unique Narayana numbers.

Number	Term	Narayana Binary String
0	*Special case*	`[0]N`
1	B_1	`[1]N`
2	B_2	`[10]N`
3	B_3	`[100]N`
4	B_4	`[1000]N`
6	B_5	`[10000]N`
9	B_6	`[100000]N`

```
The 'N' at the end of the code is there to clarify this is 'N'arayana code
```

Step 2: Find the Unique Sum (The Narayana Representation) Before we can write the binary code for any integer, we must first establish that a unique sum of our basis numbers exists for every integer. This process involves two key parts: first showing that a representation always exists, and second, showing that there is a only one way to do it.

Lemma 1.3 (Existence of Representation)**.** *Any non-negative integer can be written as a sum of terms from the Narayana Basis $\{B_k\}$, such that the indices of any two terms used in the sum differ by at least 3.*

Proof. Let n be any positive integer. Let B_k be the largest basis term such that $B_k \leq n$. This is the first term in our sum. Let the remainder be $n_1 = n - B_k$.

By the definition of B_k as the largest basis term less than or equal to n, we know that $n < B_{k+1}$. Using the recurrence relation for the basis, $B_{k+1} = B_k + B_{k-2}$, we can see that:

$$n < B_k + B_{k-2}$$

Subtracting B_k from both sides gives:

$$n - B_k < B_{k-2}$$

So, the remainder n_1 is strictly less than B_{k-2}.

This is the crucial step. When we repeat the process on the remainder n_1, the largest basis term we can choose, say B_j, must be less than or equal to n_1. Since $n_1 < B_{k-2}$, we must have $B_j < B_{k-2}$, which implies its index j must be less than $k-2$ (i.e., $j \leq k-3$). This guarantees that the index of the second term, j, is separated from the index of the first term, k, by at least 3. This condition holds at every step of the algorithm.

This process generates a sequence of remainders, $n > n_1 > n_2 > \ldots$, which is strictly decreasing. Since each term in the basis $\{B_k\}$ is a positive integer (with $B_1 = 1$ being the smallest), we are always subtracting

a positive value at each step. A strictly decreasing sequence of non-negative integers must be finite and is guaranteed to terminate at 0. Thus, the algorithm always succeeds in finding a complete representation.

□

Now we need to also check whether indeed every non-negative number can be defined uniquely, but for doing that we need to define a summation lemma first:

Lemma 1.4 (Maximal Sum Property)**.** *For* $j \in \{1,2,3\}$

$$\sum_{i=0}^{m+1} B_{3i+j} = B_{3m+j+1} - 1$$

Proof. The proof will be done by induction on the variable m.

Base Cases: First, we verify the formula for $m = 0$ for each case of $j \in \{1,2,3\}$.

- For $j = 1$: The LHS is $\sum_{i=0}^{0} B_{3i+1} = B_1 = 1$. The RHS is $B_{3(0)+1+1} - 1 = B_2 - 1 = 2 - 1 = 1$. The identity holds.
- For $j = 2$: The LHS is $\sum_{i=0}^{0} B_{3i+2} = B_2 = 2$. The RHS is $B_{3(0)+2+1} - 1 = B_3 - 1 = 3 - 1 = 2$. The identity holds.
- For $j = 3$: The LHS is $\sum_{i=0}^{0} B_{3i+3} = B_3 = 3$. The RHS is $B_{3(0)+3+1} - 1 = B_4 - 1 = 4 - 1 = 3$. The identity holds.

The base cases are verified.

Inductive Hypothesis: Assume the formula is true for some integer $k \geq 0$. We assume:

$$\sum_{i=0}^{k} B_{3i+j} = B_{3k+j+1} - 1$$

Inductive Step: Now, to show the formula holds for $k+1$, we start with the left-hand side of the sum up to $k+1$:

$$\begin{aligned}
\sum_{i=0}^{k+1} B_{3i+j} &= \left(\sum_{i=0}^{k} B_{3i+j}\right) + B_{3(k+1)+j} \\
&= (B_{3k+j+1} - 1) + B_{3k+j+3} && \text{(by the inductive hypothesis)} \\
&= (B_{3k+j+1} + B_{3k+j+3}) - 1
\end{aligned}$$

Using the original Narayana recurrence relation, we know that for our basis $B_n = N_{n+2}$, the equivalent recurrence is $B_n = B_{n-1} + B_{n-3}$. Letting $n = 3k + j + 4$, we get:

$$B_{3k+j+4} = B_{3k+j+3} + B_{3k+j+1}$$

Substituting this identity back into our expression:

$$\begin{aligned}
\sum_{i=0}^{k+1} B_{3i+j} &= (B_{3k+j+4}) - 1 \\
&= B_{3(k+1)+j+1} - 1
\end{aligned}$$

This is the right-hand side of the formula for the case $k+1$. Since the base cases and the inductive step hold, the lemma is proven.

□

With these lemmas in place, we can check whether indeed any non-negative integer can be uniquely defined using our sum. There are after all multiple possibilities for signifying any positive integer. For example, the number 10 could be $9+1$ or $6+3+1$. We use the following method:

Theorem 1.5 (Existence and Uniqueness of the Narayana Representation)**.** *Every non-negative integer can be represented **uniquely** as a sum of Narayana basis numbers whose indices differ by at least 3.*

Proof. To prove uniqueness, we use proof by contradiction. Assume there exists a positive integer n with two different valid Narayana representations, S_1 and S_2. Let these sums be:

$$S_1 = B_{x_1} + B_{x_2} + \cdots + B_{x_a}$$
$$S_2 = B_{y_1} + B_{y_2} + \cdots + B_{y_b}$$

We assume the indices in each sum are ordered ($x_1 > x_2 > \ldots$) and satisfy the separation condition ($x_i - x_{i+1} \geq 3$, and similarly for y_i). Since the representations are different, the set of indices $\{x_i\}$ is not equal to the set $\{y_i\}$.

Now, we remove any basis terms that are common to both sums. This is a valid step because if $S_1 = S_2$, then subtracting the same value from both sides maintains the equality. For example, if both sums contained the term B_c, we can subtract it to get the new equality $S_1 - B_c = S_2 - B_c$.

Let the largest term in the resulting S_1 be B_{k_1} and the largest term in S_2 be B_{j_1}. We must have $k_1 \neq j_1$, otherwise we could subtract that term as well.

Assume, without loss of generality, that $k_1 > j_1$. The sum S_1 is at least as large as its largest term:

$$S_1 \geq B_{k_1}$$

Now consider the sum S_2. Since B_{j_1} is its largest term and all terms in the sum must have indices separated by at least 3, the maximum possible value for S_2 is the sum $B_{j_1} + B_{j_1-3} + B_{j_1-6} + \ldots$. By applying our Maximal Sum Property (Lemma 1.4), we know this sum is equal to , $B_{j_1+1} - 1$. Thus we know that the sum of any valid representation is less or equal to the next basis term after its largest component subtracted by 1. Therefore:

$$S_2 = B_{j_1} + B_{j_2} + \cdots \leq B_{j_1+1} - 1 < B_{j_1+1}$$

Since the indices k_1 and j_1 are integers and we assumed $k_1 > j_1$, it must be that $k_1 \geq j_1 + 1$. Because the basis $\{B_k\}$ is strictly increasing, this implies:

$$B_{k_1} \geq B_{j_1+1}$$

Combining these inequalities, we get the following chain:

$$S_1 \geq B_{k_1} \geq B_{j_1+1} > S_2$$

This leads to the conclusion that $S_1 > S_2$. This result, $S_1 > S_2$, contradicts our initial premise that $S_1 = S_2$. Therefore, the assumption that two different valid representations exist for the same number must be false. The representation is unique.

□

We now have a proven method for finding the unique Narayana representation for any positive integer. The process is as follows:

For a given integer n, we first select the largest basis term B_k such that $B_k \leq n$. This becomes the first term in our sum. We then repeat this process on the remainder, $n_1 = n - B_k$, then on the next remainder, and so on, until the remainder is zero.

Let's test this with an example:

Example: Finding the unique sum for 18.

- The largest basis term ≤ 18 is $B_7 = 13$. Our first term is B_7.

- The remainder is $18 - 13 = 5$.
- We now need the largest Narayana number $B_j \leq 5$. This is $B_4 = 4$
- The remainder is $5 - 4 = 1$.
- The largest and only basis term so that basis term ≤ 1 is $B_1 = 1$.
- The unique sum is therefore $B_7 + B_4 + B_1 = 13 + 4 + 1$.

Step 3: Convert the Unique Sum into a Binary String Once you have the unique sum, creating the binary code is straightforward. If the k-th Basis number is present in the sum, we place a '1' in the k-th position in the code (reading from right to left), else a '0'. We know from the previous step that code snippets such as '[..11..]N' or '[..101..]N' won't exist.

Example: Writing the code for 18.

- Our unique sum for 18 was $B_7 + B_4 + B_1$.
- This means we need a '1' at the 7^{th}, a '1' at the 4^{th} position and a '1' at the 1^{st} position.
- The resulting binary string is:

$$[1001001]_N$$

By this binary code system, any nonnegative integer can be encoded in the Narayana representation system.

Step 4: Defining addition (adder) in binary code A system for writing down numbers is a good starting point, but for it to be a truly useful numeral system, we must be able to perform arithmetic. This requires an algorithm for operations like addition that works directly on the binary strings themselves.

Definition 1.6 (Adder Automaton)**.** *An adder is a computational machine that can verify addition. It is designed to read the binary strings for three numbers x, y and z, and give a definitive "yes" or "no" answer to the question: does $x + y = z$ hold true for certain x, y and z, according to the rules of a specific binary system?*

For a non-standard system like Narayana's, the adder must also enforce the grammatical rules of the representation. For example, it must ensure that the resulting sum, z, is a valid Narayana string (i.e., it contains no '11' or '101' substrings). This complexity begs the question: can such an adder automaton even exist for Narayana's sequence?

Fortunately, we do not have to guess. Researchers Frougny and Solomyak built a theorem, which can tell us whether a finite adder automaton exists.

Theorem 1.7. *(Adder Existence). Addition in a numeration system based on a linear recurrence is computable by a finite automaton if the dominant root of the sequence's characteristic polynomial is a Pisot number [4, 5].*

Proof. The proof is too large for this thesis, but can be found (partly)in the works of Frougny, cited above.

□

What this means, is that we can build an machine of some sort if the *Pisot property* is satisfied. The property relates to the sequence's characteristic polynomial. This polynomial is needed to define the growth rates. More about this, again, in the next section. There, we'll demonstrate that the dominant root of the characteristic equation indeed is a Pisot number.

It is thus known that the adder exists, but how to construct it? This would be done by comparing the sequence to other recursive sequences and their respective adders. Jeffrey Shallit found it earlier this year. Someone key, who will be mentioned in the next chapters as well.

1.6. Finding a formula for Narayana's sequence

1.6.1. The Characteristic Equation

Just as the behavior of the Fibonacci sequence is governed by the golden ratio, Narayana's sequence has its own constant that dictates its long-term growth. This constant, called the supergolden ratio by some [6], is one of the roots for the sequence's characteristic equation. The supergolden ratio signifies that eventually, the next number in the sequence is approximately equal to the product of the present number and this supergolden ratio.

Lemma 1.8. *(Characteristic equation) The characterisic equation for Narayana is given by:*

$$r^3 - r^2 - 1 = 0$$

Proof. We know we have

$$N(n) = N(n-1) + N(n-3)$$

thus

$$N(n+3) = N(n+2) + N(n)$$

thus

$$N(n+3) - N(n+2) - N(n) = 0$$

Now we know that for large n, we get $N(n+1) = N(n) * r$ for r being the supergolden ratio. Then for large enough n, we get $N(n) \approx r^n$:

$$r^{n+3} - r^{n+2} - r^n = 0$$
$$r^n(r^3 - r^2 - 1) = 0$$

We know $r = 0$ is for sure not true, as $N(n)$ keeps increasing for n increasing as well, thus we get

$$r^3 - r^2 - 1 = 0$$

□

Solving this equation will give use 3 roots, one of which will be the supergolden ratio. Cardano's method is mostly used in these types of equations, this time as well.

1.6.2. Cardano's Method and its History

Finding the roots of quadratic equations was known since antiquity, but a general solution for cubic equations remained elusive for centuries. The breakthrough came in 16th-century Italy, resulting in a method now famously, and controversially, known as Cardano's method.

The Method at a Glance

The core idea behind Cardano's method is to transform a complex problem into a simpler one through a series of substitutions. To solve a general cubic equation of the form $ax^3 + bx^2 + cx + d = 0$, the process is as follows:

1. **Depress the Cubic:** First, the equation is simplified into a "depressed cubic" that lacks an x^2 term. This is achieved with the substitution $x = y - \frac{b}{3a}$, which results in a much cleaner equation of the form:

$$y^3 + py + q = 0$$

2. **The Key Substitution:** Next, assume the solution for y can be written as the sum of two other variables, $y = u + v$. Substituting this into the depressed cubic and rearranging gives:

$$(u^3 + v^3) + (3uv + p)(u + v) + q = 0$$

3. **Reduce to a Quadratic Problem:** By imposing the condition that $3uv + p = 0$ (which simplifies to $uv = -p/3$), the equation collapses to $u^3 + v^3 = -q$. With a system of two equations for u^3 and v^3: their sum being $-q$ and their product being $u^3v^3 = (-p/3)^3$, the system can be solved with a quadratic equation. This allows us to find the values of u^3 and v^3.

4. **Find the Roots:** By taking the cube roots of the values found for u^3 and v^3, we can find our u and v. The key insight here is that every number has three cube roots in the complex plane. If u_0 is one cube root, the other two are $u_0\omega$ and $u_0\omega^2$, where ω has the property: $\omega^3 = 1$, being one of the complex roots of this same equation. The condition $uv = -p/3$ forces specific pairings of these roots, which gives the three solutions for y. Finally, by reversing the initial substitution, $x = y - b/3a$, we find all three roots of the original cubic equation.

A Story of Secrecy, Betrayal, and Genius

The story behind this formula is as dramatic as the mathematics is brilliant. The first person known to have solved a type of depressed cubic was *Scipione del Ferro*, an early 16th-century Italian mathematician. As was common practice at the time, he kept his solution a closely guarded secret, revealing it only to a few students on his deathbed [7].

Years later, another mathematician, *Niccolò Fontana Tartaglia* (whose surname, meaning "the stammerer," came from a childhood injury), independently rediscovered the method. His fame grew, attracting the attention of the brilliant and eccentric scholar *Gerolamo Cardano*.

Cardano, who was writing a comprehensive mathematical treatise, repeatedly begged Tartaglia to reveal the secret. Tartaglia eventually relented, but only after making Cardano swear a solemn oath never to publish it, as Tartaglia intended to publish it himself. However, after discovering that del Ferro had the solution before Tartaglia, Cardano felt that the oath was no longer binding. In 1545, he published the method in his monumental book, *Ars Magna* ("The Great Art"), giving full credit to both del Ferro and Tartaglia for their work. Despite this, Tartaglia was furious, leading to one of the most famous and bitter disputes in the history of mathematics [8].

Though Cardano did not claim to be the first to solve it, the method has been known as his ever since because his *Ars Magna* was the first publication to reveal it to the world.

Figure 1.1: Gerolamo Cardano (1501-1576), the Italian polymath who first published the general solution to the cubic equation in his 1545 work, *Ars Magna.*

1.6.3. The Characteristic Roots of Narayana's Sequence

We now apply Cardano's method to find an exact algebraic solution for the roots of the characteristic equation, $r^3 - r^2 - 1 = 0$. Such a solution, expressed using only arithmetic operations and roots (like square roots and cube roots), is known as a *solution in radicals*.

It is a famous result from algebra, the Abel-Ruffini theorem, that such general formulas using radicals only exist for polynomial equations of degree four or less [9]. Our characteristic equation is of the third degree, thus we are guaranteed that such a solution can be found.

Theorem 1.9 (The Characteristic Roots)**.** *The characteristic equation for Narayana's sequence, $r^3 - r^2 - 1 = 0$, has one real root and two complex conjugate roots. Their exact forms and approximate values are as follows:*

1. ***The Real Root,** r_1 **(The Supergolden Ratio):***

$$r_1 = \sqrt[3]{\frac{29+3\sqrt{93}}{54}} + \sqrt[3]{\frac{29-3\sqrt{93}}{54}} + \frac{1}{3} \approx 1.465571$$

2. ***The Complex Roots,** r_2 **and** r_3**:***

$$r_2 = \left(\frac{1}{3} - \frac{1}{2}\left(\sqrt[3]{\frac{29+3\sqrt{93}}{54}} + \sqrt[3]{\frac{29-3\sqrt{93}}{54}}\right)\right) + i\frac{\sqrt{3}}{2}\left(\sqrt[3]{\frac{29+3\sqrt{93}}{54}} - \sqrt[3]{\frac{29-3\sqrt{93}}{54}}\right)$$

$$r_2 \approx -0.232786 + 0.792552i$$

$$r_3 = \left(\frac{1}{3} - \frac{1}{2}\left(\sqrt[3]{\frac{29+3\sqrt{93}}{54}} + \sqrt[3]{\frac{29-3\sqrt{93}}{54}}\right)\right) - i\frac{\sqrt{3}}{2}\left(\sqrt[3]{\frac{29+3\sqrt{93}}{54}} - \sqrt[3]{\frac{29-3\sqrt{93}}{54}}\right)$$

$$r_3 \approx -0.232786 - 0.792552i$$

The magnitude of the complex roots is $|r_2| = |r_3| \approx 0.82603$.

Proof. We apply Cardano's method to solve the characteristic equation $r^3 - r^2 - 1 = 0$.

1. Depress the Cubic: First, we eliminate the r^2 term by substituting $r = y + \frac{1}{3}$. This transformation yields the depressed cubic equation:

$$y^3 - \frac{1}{3}y - \frac{29}{27} = 0$$

From this, we identify the coefficients $p = -\frac{1}{3}$ and $q = -\frac{29}{27}$.

2. Solve the Depressed Cubic: Cardano's method provides the solutions for y based on two components, which we will call u_0 and v_0 for this proof. These are defined as $u_0 = \sqrt[3]{-\frac{q}{2} + \sqrt{D}}$ and $v_0 = \sqrt[3]{-\frac{q}{2} - \sqrt{D}}$, where the discriminant is $D = (\frac{q}{2})^2 + (\frac{p}{3})^3$.

We calculate the necessary terms:

- $-\frac{q}{2} = \frac{29}{54}$
- $D = \left(-\frac{29}{54}\right)^2 + \left(-\frac{1}{9}\right)^3 = \frac{841}{2916} - \frac{1}{729} = \frac{841-4}{2916} = \frac{837}{2916} = \frac{31}{108}$
- $\sqrt{D} = \sqrt{\frac{31}{108}} = \frac{\sqrt{93}}{18}$

Substituting these values gives the exact forms for our building blocks, u_0 and v_0:

$$u_0 = \sqrt[3]{\frac{29}{54} + \frac{\sqrt{93}}{18}} = \sqrt[3]{\frac{29 + 3\sqrt{93}}{54}}$$
$$v_0 = \sqrt[3]{\frac{29}{54} - \frac{\sqrt{93}}{18}} = \sqrt[3]{\frac{29 - 3\sqrt{93}}{54}}$$

3. Construct the Final Roots: The three solutions for y are given by $y_1 = u_0 + v_0$, $y_2 = \omega u_0 + \omega^2 v_0$, and $y_3 = \omega^2 u_0 + \omega v_0$, where $\omega = -\frac{1}{2} + i\frac{\sqrt{3}}{2}$ is such that $\omega^3 = 1$ where $\omega^2 = \bar{\omega}$. To find the roots for r, we reverse the initial substitution, $r = y + \frac{1}{3}$. This gives:

- $r_1 = (u_0 + v_0) + \frac{1}{3}$
- $r_2 = (\omega u_0 + \omega^2 v_0) + \frac{1}{3}$
- $r_3 = (\omega^2 u_0 + \omega v_0) + \frac{1}{3}$

Expanding these expressions fully and separating the real and imaginary parts yields the explicit formulas stated in the theorem. The supergolden ratio, which describes the long-term ratio of consecutive Narayana numbers, must be a real number. Thus, the unique real root, r_1, is the supergolden ratio.

□

A key property of linear recurrence relations is that any linear combination of the fundamental solutions is also a solution. This is what will lead us to a specific formula to calculate numbers of Narayana's sequence in the next section.

Pisot property

Finally, we can examine the Pisot property. A number is classified as a Pisot number if it is an algebraic integer greater than 1, and all of its other conjugate roots have a magnitude less than 1.

The characteristic equation of Narayana's sequence has one real root, $\alpha \approx 1.46557$, and two complex conjugate roots with a magnitude of approximately 0.826. Since the dominant root's magnitude is greater than 1 while the magnitudes of all other roots are less than 1, the conditions are satisfied. This confirms that the supergolden ratio α is a Pisot number. The existence of a finite adder automaton for the Narayana numeral system is thus guaranteed.

1.7. Formula for Narayana's Sequence

Why Solving the System Gives the Formula

As we know that any linear combination of the fundamental solutions (the roots r_1, r_2, r_3) is also a solution, we can express the general solution to the recurrence relation as:

$$N_n = C_1 r_1^n + C_2 r_2^n + C_3 r_3^n \tag{1.1}$$

This form (1.1), satisfies the recurrence for any arbitrary constants C_1, C_2, C_3. If we then determine specific values for these constants by solving a system of equations using the initial conditions for Narayana's sequence ($N_0 = 0, N_1 = 1, N_2 = 1$), we can find a formula which holds for Narayana's recurrence and initial values. This value then allows for direct calculation of N_n.

The Explicit Formula for Narayana's Sequence

With the characteristic roots established, we can now state the formula for any term in the Narayana sequence.

Theorem 1.10 (Formula for Narayana's Sequence)**.** *Let $\{N_n\}$ be Narayana's sequence with $N_0 = 0, N_1 = 1, N_2 = 1$. Let r_1, r_2, r_3 be the roots of the characteristic polynomial $P(r) = r^3 - r^2 - 1 = 0$. The n-th term of the sequence is given by the explicit formula:*

$$N_n = C_1 r_1^n + C_2 r_2^n + C_3 r_3^n \tag{1.2}$$

where the constants C_k are approximately given by

- $C_1 \approx 0.417233$
- $C_2 \approx -0.20862 - 0.18382i$
- $C_3 \approx -0.20862 + 0.18382i$

Proof. To find the specific constants for our sequence, we match the general form to the initial conditions $N_0 = 0, N_1 = 1$, and $N_2 = 1$. This creates the following system of three linear equations for the unknowns C_1, C_2, and C_3:

$$\begin{aligned} C_1 + C_2 + C_3 &= 0 \\ C_1 r_1 + C_2 r_2 + C_3 r_3 &= 1 \\ C_1 r_1^2 + C_2 r_2^2 + C_3 r_3^2 &= 1 \end{aligned}$$

This system can be expressed in matrix form as:

$$\begin{pmatrix} 1 & 1 & 1 \\ r_1 & r_2 & r_3 \\ r_1^2 & r_2^2 & r_3^2 \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

The matrix is a well-known type called a **Vandermonde matrix**. A standard method for solving such a system is Cramer's rule. This rule states that the solution for each unknown constant, C_k, is given by the ratio of two determinants:

$$C_k = \frac{\det(A_k)}{\det(A)}$$

A is the main Vandermonde matrix and A_k is the matrix formed by replacing the k-th column of A with the vector of initial conditions $(0,1,1)^T$.

The determinant of the Vandermonde matrix in the denominator is the same for all three constants:

$$\det(A) = \det\begin{pmatrix} 1 & 1 & 1 \\ r_1 & r_2 & r_3 \\ r_1^2 & r_2^2 & r_3^2 \end{pmatrix}$$

The numerators are then defined symmetrically for each constant:

- For C_1, we replace the first column:

$$C_1 = \frac{\det\begin{pmatrix} \mathbf{0} & 1 & 1 \\ \mathbf{1} & r_2 & r_3 \\ \mathbf{1} & r_2^2 & r_3^2 \end{pmatrix}}{\det(A)}$$

- For C_2, we replace the second column:

$$C_2 = \frac{\det\begin{pmatrix} 1 & \mathbf{0} & 1 \\ r_1 & \mathbf{1} & r_3 \\ r_1^2 & \mathbf{1} & r_3^2 \end{pmatrix}}{\det(A)}$$

- For C_3, we replace the third column:

$$C_3 = \frac{\det\begin{pmatrix} 1 & 1 & \mathbf{0} \\ r_1 & r_2 & \mathbf{1} \\ r_1^2 & r_2^2 & \mathbf{1} \end{pmatrix}}{\det(A)}$$

The determinant of the Vandermonde matrix in the denominator is a standard result: $(r_2-r_1)(r_3-r_1)(r_3-r_2)$. The determinant in the numerator can be expanded and simplified. After a fair amount of algebraic manipulation, and by using properties of polynomial roots, this ratio simplifies to the general form:

$$C_k = \frac{r_k}{P'(r_k)} \quad \text{for } k = 1,2,3$$

We know $P'(r) = 3r^2 - 2r$ is the derivative of the characteristic polynomial. Substituting these constants back into the general form gives the explicit formula stated in the theorem.

Numerically, these constants evaluate to approximately:

- $C_1 \approx 0.417233$
- $C_2 \approx -0.20862 - 0.18382i$
- $C_3 \approx -0.20862 + 0.18382i$

□

Dominant Term Approximation

The formula (1.2) is exact. For large n, the behavior of N_n is dominated by the term involving the root with the largest magnitude, r_1.

Remember, the magnitudes of the roots are:

- $|r_1| \approx 1.46557123 > 1$
- $|r_2| = |r_3| \approx 0.826029 < 1$

As $n \to \infty$:

- The term $|r_1|^n$ increases without bound.
- The terms $|r_2|^n$ and $|r_3|^n$ approach 0, since $0 < |r_k| < 1$ for $k = 2, 3$.

Convergence of Terms with Magnitudes Less Than 1:

Applying this to Narayana's sequence, as n becomes large, the contributions from $C_2 r_2^n$ and $C_3 r_3^n$ become negligible compared to $C_1 r_1^n$. Thus, for large n:

$$N_n \approx C_1 r_1^n$$

We now test the approximation $N_n \approx C_1 r_1^n$ using more precise constants:

- $C_1 \approx 0.417233400536$
- $r_1 \approx 1.465571231877$

Approximation for $n = 3$ (Expected $N_3 = 1$):

$$C_1 r_1^3 \approx (0.417233400536) \cdot (1.465571231877)^3 \approx 1.3134018$$

The approximation 1.3134018 differs from the actual value $N_3 = 1$ by 0.3134018. The relative error is approximately 31.3%. This indicates that for small n, the terms $C_2 r_2^n$ and $C_3 r_3^n$ are still significant.

Approximation for $n = 10$ (Expected $N_{10} = 19$):

$$C_1 r_1^{10} \approx (0.417233400536) \cdot (1.465571231877)^{10} \approx 18.98520$$

The approximation 18.98520 is very close to the actual value $N_{10} = 19$ (with relative error 0.078%). The small remaining error comes from $C_2 r_2^{10} + C_3 r_3^{10}$, but as seen, the term $C_1 r_1^{10}$ already gives a great approximation. In conclusion, as n increases, the term $C_1 r_1^n$ associated with the dominant real root r_1 increasingly governs the value of N_n. It results in a highly accurate approximation, since r_2 and r_3 diminish.

1.8. The Generating Function for Narayana's Sequence

A powerful algebraic tool for representing and analyzing an integer sequence is its generating function. This concept allows us to encode an entire infinite sequence into a single, compact function.

Definition 1.11 (Ordinary Generating Function)**.** *Let $\{a_n\}_{n \geq 0} = (a_0, a_1, a_2, \ldots)$ be a sequence of numbers. The ordinary generating function for this sequence is the formal power series:*

$$G(x) = \sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots$$

In this section, we derive the ordinary generating function for Narayana's sequence. Let $G(x)$ be the ordinary generating function for Narayana's sequence $\{N_n\}$. By definition:

$$G(x) = \sum_{n=0}^{\infty} N_n x^n = N_0 x^0 + N_1 x^1 + N_2 x^2 + N_3 x^3 + N_4 x^4 + \dots \tag{1.3}$$

Utility of Generating Functions

Generating functions are valuable for several reasons:

- **Compact Representation:** They provide a single algebraic expression that encodes an entire infinite sequence. This is often more manageable than listing terms or using only a recurrence relation.

- **Proving Identities and Properties of Sequences:** Algebraic manipulation of generating functions can often be the key to solving proofs of identities involving the terms of the sequence. Differentiating or integrating $G(x)$ can also yield such results.

- **Understanding the Long-Term Growth Rate:** Long-term growth be understood using a generating function. The key to this lies in analyzing the function's *singularities*—points where the function is not well-defined, often because a denominator becomes zero.

  For a large class of sequences, including those defined by linear recurrences like Narayana's, these singularities are well-behaved points known as *poles*. For the pole with the smallest magnitude: a point x_0, we know $r = \frac{1}{x_0}$.

- **Relationship with Other Sequences:** Comparing generating functions can reveal relationships between different sequences.

1.8.1. Narayana's Generating Function

Theorem 1.12 (Generating Function for Narayana's Sequence)**.** *The ordinary generating function for Narayana's sequence is given by:*

$$G(x) = \sum_{n=0}^{\infty} N_n x^n = \frac{x}{1 - x - x^3}$$

Proof. Let $G(x) = \sum_{n=0}^{\infty} N_n x^n$. We begin by writing out the first few terms and then applying the recurrence relation for $n \geq 3$:

$$\begin{aligned} G(x) &= N_0 + N_1 x + N_2 x^2 + \sum_{n=3}^{\infty} N_n x^n \\ &= 0 + 1x + 1x^2 + \sum_{n=3}^{\infty} (N_{n-1} + N_{n-3}) x^n \\ &= x + x^2 + \sum_{n=3}^{\infty} N_{n-1} x^n + \sum_{n=3}^{\infty} N_{n-3} x^n \end{aligned}$$

We now express the two summation terms in relation to $G(x)$.

For the first summation, we factor out x:

$$\begin{aligned}\sum_{n=3}^{\infty} N_{n-1}x^n &= x\sum_{n=3}^{\infty} N_{n-1}x^{n-1} \\ &= x\sum_{k=2}^{\infty} N_k x^k && \text{(letting } k = n-1\text{)} \\ &= x\left(\left(\sum_{k=0}^{\infty} N_k x^k\right) - N_0 - N_1 x\right) \\ &= x(G(x) - 0 - 1x) \\ &= xG(x) - x^2\end{aligned}$$

For the second summation, we factor out x^3:

$$\begin{aligned}\sum_{n=3}^{\infty} N_{n-3}x^n &= x^3\sum_{n=3}^{\infty} N_{n-3}x^{n-3} \\ &= x^3\sum_{j=0}^{\infty} N_j x^j && \text{(letting } j = n-3\text{)} \\ &= x^3 G(x)\end{aligned}$$

Substituting these two results back into the original equation for $G(x)$:

$$G(x) = x + x^2 + (xG(x) - x^2) + (x^3G(x))$$

The x^2 terms cancel, leaving:

$$G(x) = x + xG(x) + x^3G(x)$$

Finally, we solve for $G(x)$ by grouping terms:

$$\begin{aligned}G(x) - xG(x) - x^3G(x) &= x \\ G(x)(1 - x - x^3) &= x \\ G(x) &= \frac{x}{1 - x - x^3}\end{aligned}$$

□

1.9. A Summation Identity for Narayana's Sequence

One of the important properties of Narayana's sequence is a closed form for the sum of its terms. This identity will be proven later on using the principle of mathematical induction.

Lemma 1.13 (Sum of Narayana Numbers)**.** *For any integer $n \geq 0$, the sum of the first $n+1$ terms in Narayana's sequence is given by:*

$$\sum_{k=0}^{n} N_k = N_{n+3} - 1$$

Proof. The proof proceeds by induction on n. For $m = 0$:

$$\sum_{k=0}^{0} N_k = N_0 = 0$$
$$N_{0+3} - 1 = N_3 - 1 = 1 - 1 = 0$$

So it holds for $m = 0$

Assume the formula holds for some integer $m \geq 0$. That is, we assume:

$$\sum_{k=0}^{m} N_k = N_{m+3} - 1$$

Then for the left-hand side for $m+1$:

$$\begin{aligned}
\sum_{k=0}^{m+1} N_k &= \left(\sum_{k=0}^{m} N_k\right) + N_{m+1} \\
&= (N_{m+3} - 1) + N_{m+1} && \text{(by the inductive hypothesis)} \\
&= (N_{m+1} + N_{m+3}) - 1 \\
&= N_{m+4} - 1
\end{aligned}$$

This is precisely the right-hand side of the formula for the case $n = m+1$.

Since the base cases and the inductive step hold, we conclude by the principle of mathematical induction that the formula is valid for all integers $n \geq 0$.

□

This identity is a key property of Narayana's numbers. While proven here by induction, it is also a statement that can be formulated and verified automatically. In Chapter 3, we will demonstrate how a tool like Walnut can be used to prove the underlying identities that make such inductive proofs work. This provides an additional powerful approach to verifying these kinds of theorems.

2

Walnut

In the last chapter, we explored Narayana's sequence using classical mathematical tools like induction and algebra. Now, we shift our focus to a modern computational approach. This chapter introduces Walnut, an automated theorem prover designed specifically to reason about sequences, like Narayana's.

We will start with a look at the "gears" that make Walnut work. We will then get practical, covering how to write the logical statements that ask Walnut to prove something. This chapter will provide all the necessary groundwork, preparing us for Chapter 3, where we will use Walnut to verify some interesting properties of Narayana's sequence.

2.1. Defining Walnut

The exact definition and method of usage of Walnut is summarized as follows:

> "Walnut is free software, written in Java, for deciding first-order statements about the non-negative integers, phrased in an extension of Presburger arithmetic, called Büchi arithmetic. It can be used to provide rigorous proofs or disproofs of hundreds of assertions in combinatorics on words, number theory, and other areas of discrete mathematics." [10]

To understand this statement, some of these concepts need extra explanation: first-order logic, Presburger arithmetic and Büchi arithmetic. This will happen shortly. That being said, at its core, Walnut translates logical statements into computational objects known as finite automata. Via these automata, Walnut analyzes their properties so that it can reply with either **TRUE**, or **FALSE**, depending on the validity of the statement.

2.1.1. First-Order Logic in Walnut

Working in *First-Order Logic* means we can make statements about numbers, but not about properties of numbers. Its primary components are:

- **Variables:** Symbols like x, y, z that represent non-negative integers ($\mathbb{N}_0 = \{0, 1, 2, \ldots\}$).
- **Quantifiers:** 'A' for "for all" ($\forall$) and 'E' for "there exists" ($\exists$). These can only be used on number variables.
- **Logical Connectives:** '&' (AND), '|' (OR), '~' (NOT), '=>' (IMPLIES), '<=>' (IF AND ONLY IF).

A typical first-order statement is "for every natural number (we could call this number: 'x'), there exists a natural number (we could call this number: 'y'), that is larger." In Walnut, we would write this directly as:

```
eval test "Ax Ey y>x";
```

The key here is that the quantifiers ‘A‘ and ‘E‘ only apply to the number variables, x and y. A higher-order statement can quantify over properties themselves, for example, "there exists a property that is true for both the numbers 3 and 5" ($\exists P(P(3) \wedge P(5))$). This is not allowed in first-order logic.

2.1.2. Presburger and Büchi Arithmetic

First-order logic provides the way we can build up our statements, while the *arithmetic* provides the specific mathematical vocabulary to talk about numbers. These are the (limited) set of tools the Presburger arithmetic works with:

- The ability to add numbers (+).
- The ability to check for equality (=), from which we can also define concepts like "less than" (<) and "greater than" (>).

Multiplication between two variables (like $x \cdot y$) is forbidden. Thus, although this system is powerful enough to define some properties, like "evenness" (‘$Ey\ x = y + y$‘), it is too weak to describe more complex ideas like "is a perfect square" or "is a prime number". To handle more sophisticated properties of numbers, especially those related to their representation in different bases, we need a more powerful theory. This is where *Büchi arithmetic* comes in. It extends Presburger's system by adding one new, incredibly powerful tool.

Instead of adding full multiplication, Büchi arithmetic adds a special checker: $V_p(a,b)$, which is tied to a specific integer base $p \geq 2$. This checker answers a very precise question:

Is the number a the largest power of p that divides the number b?

For example, if we choose our base to be $p = 2$ (for the normal base-2 binary system), the checker $V_2(a,b)$ would answer "**TRUE**" for $V_2(8,24)$, because 8 is a power of 2 (2^3), it divides 24, and the next power of 2 (16) does not. It would answer "**FALSE**" for $V_2(4,24)$ because, even though 4 divides 24, it isn't the largest power of 2 that does.

This checker opens up a whole new world of provable properties. However, in Walnut you must commit to a single base for any proof. You can't mix a base-2 checker with a base-3 checker in the same formula. The checker could be used for the following: if we set our system to base-2, we could test if 16 is a power of 2 by asking if $V_2(16,16)$ is true. Walnut would evaluate this to be **TRUE**.

The fundamental insight by J. Richard Büchi was that any statement you can build using this extended language (addition plus the V_p checker) can be translated into an automaton [11]. This is the core mechanism that Walnut exploits. While Büchi's original theory was for integer bases, the same thing holds for non-standard systems with an adder, as we already saw in theorem 1.7.

2.2. Fundamentals of Automata Theory

Automata are the backbone of Walnut, as they are the computational objects it uses to verify whether a logical statement is true or false.

2.2.1. What is an Automaton?

At its heart, an automaton is a simple mathematical model of a machine. It performs a computation by reading an input string of symbols -like the bits ‘10011‘- one symbol at a time, and moves through a sequence of pre-defined *states* based on what it reads. A set of rules, called the transition function, dictates exactly where it goes next.

Let's start with a simple example to see this in action. Imagine we want to build a machine that checks if a given string consists **only** of 0s and 1s.

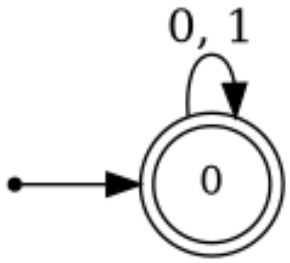

Figure 2.1: A simple automaton that accepts any string of 0s and 1s.

This automaton, shown in Figure 2.1, works as follows:

- It has only one state, which we'll call "State 0". The double circle means this is an "accepting" state.
- It starts in State 0.
- If it reads a '0' or a '1', the looping arrow tells it to stay in State 0.
- As long as it only reads 0s and 1s, it will remain in the accepting State 0. If it were to encounter any other symbol (like a '2'), there is no defined path, so the string would be immediately rejected.
- When the string ends, if the machine is in an accepting state, the input string is accepted.

2.2.2. Definition of an Automaton

Now that we have an intuitive idea, we can formalize it. A basic type of automaton, a **Finite Automaton** (FA), is formally defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of **states**. These represent the different configurations or memory states the machine can be in. In our simple example, $Q = \{0\}$, as this is the only state we can be in.
- Σ is a finite set of input symbols, called the **alphabet**. These are the symbols the automaton can read (and thus won't immediately reject). In our example, $\Sigma = \{0, 1\}$.
- δ (delta) is the **transition function**. It defines the rules for moving from one state to another. For a Deterministic Finite Automaton (DFA), this is a function $\delta : Q \times \Sigma \to Q$. For our example, the rules are $\delta(0, 0) \to 0$ and $\delta(0, 1) \to 0$. This means that if we are in state 0 and read input '0', we next turn we're (again) in state 0. Same thing when reading input '1'.
- $q_0 \in Q$ is the **initial state**. This is where the machine starts. In our example, $q_0 = 0$.
- $F \subseteq Q$ is the set of **final states** (or accepting states). If the automaton ends in one of these states (thus when the last bit has been read), the input string is accepted. In our example, $F = \{0\}$. An accepting state will be visualized here by putting 2 circles around the state and rejected if there's only 1 circle.

Although automata in general can process all kinds of strings -from patterns in text to sequences of events-, the automata used within Walnut are specialized. They are designed to work with strings that represent non-negative integers. When a user provides a number in a proof or definition, Walnut first translates that number into a string of digits according to a specific numeral system. The automaton then reads this string, bit by bit, to make its decision.

By default, Walnut assumes these strings are written in the *standard base-2* system. So, if it encounters the string '10011' and no other numeral system is specified, it interprets it as the number 19, because that is what the string signifies in the base-2 system. Of course, the same string '10011' could mean something completely different in another numeral system. You would have to specify that.

2.2.3. More Complex Examples

Automata can handle much more complex tasks by using more states as a form of memory.

The automaton in Figure 2.2 is designed to check if one number, m, is strictly less than another, n. To do this, it reads the binary strings for both numbers simultaneously, one bit-pair at a time, starting from the most significant digit (from left to right).

The logic of this comparison is based on finding the first position from the left where the bits of m and n differ. As soon as it reads a pair where the bit from m is ‘0‘ and the bit from n is ‘1‘, it knows for certain that $m < n$, regardless of any subsequent bits. Conversely, if it first encounters a pair where the bit from m is ‘1‘ and from n is ‘0‘, it knows $m > n$, and the condition $m < n$ has failed. As long as the bits are identical (‘(0,0)‘ or ‘(1,1)‘), the relationship is undecided, and it must continue reading.

The states of the automaton serve as its memory for this process. State 0 represents the "undecided" phase, while State 1 represents the "success" condition where $m < n$ has been confirmed. For the comparison to be true, the automaton must end in this "success" state.

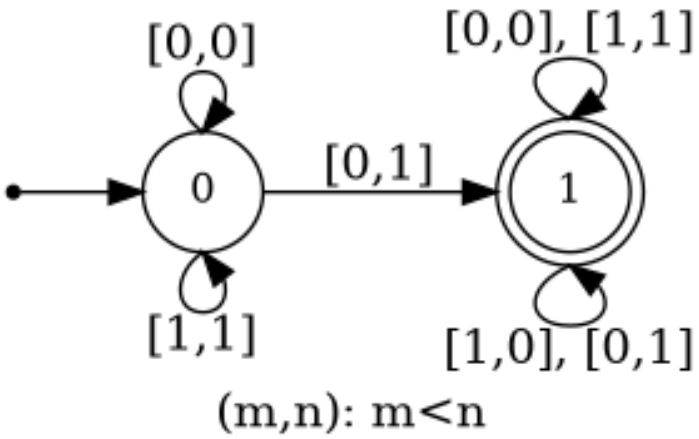

Figure 2.2: Automaton that determines whether a number m is strictly larger than a number n.

We can now formally describe this comparator using the 5-tuple notation $(Q, \Sigma, \delta, q_0, F)$:

- $Q = \{0, 1\}$: The set of states. State 0 represents the condition "m and n have been equal so far," while State 1 represents "$m < n$ has been established."
- $\Sigma = \{(0,0), (0,1), (1,0), (1,1)\}$: The alphabet consists of all four possible pairs of bits that can be read from the two numbers in a single column.
- δ: The transition function encodes the comparison logic. From State 0 ("equal so far"), reading a ‘(0,1)‘ pair confirms that $m < n$ and transitions to State 1. Reading equal pairs like ‘(0,0)‘ or ‘(1,1)‘ keeps the automaton in State 0. Any transition that would confirm $m > n$ (like reading ‘(1,0)‘ from State 0) is not shown, as it leads to an implicit, non-accepting "dead" state. Once in State 1, the outcome is decided, so all subsequent inputs loop back to State 1.
- $q_0 = 0$: The automaton starts in State 0, assuming the numbers are equal until evidence proves otherwise.
- $F = \{1\}$: The set of final states contains only State 1. The input is accepted only if the process ends with the condition $m < n$ having been definitively established. Ending in State 0 would mean the numbers were identical, so $m < n$ is false.

The automaton in Figure 2.3 is designed to verify binary addition, checking if the relationship $x + y = z$ holds true. It works by reading triplets of bits (x_i, y_i, z_i) from each number simultaneously, moving from right to left (from the least significant bit upwards), mimicking how we perform addition by hand.

The key to making this work is the concept of a "carry". When we add bits in a column, the result might be too large to fit in a single bit. For example, $1 + 1$ gives a result of 2, which in binary is ‘10‘. This means we write down ‘0‘ as the sum for that column and must "carry" the ‘1‘ over to be added to the next column on the left.

This is exactly what the states of the automaton are for: they serve as the machine's memory for this carry value. The state ‘0‘ represents a carry of 0, while state ‘1‘ represents a carry of 1. For a sum to be arithmetically correct, the entire calculation must finish with no leftover carry. This is why the automaton is designed to only accept if it ends in State 0.

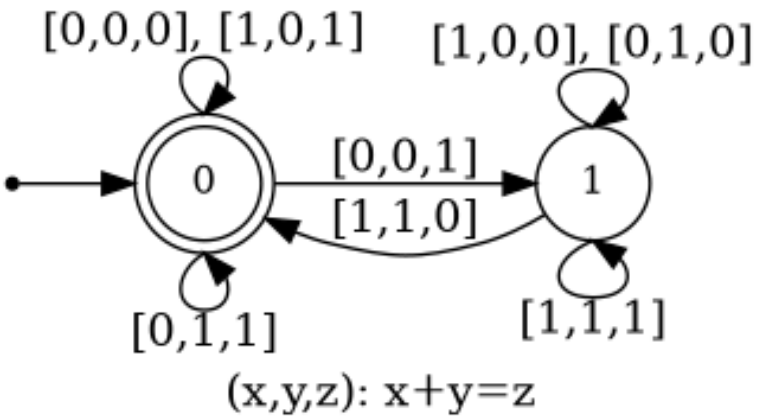

Figure 2.3: Automaton that checks whether three binary numbers satisfy $x + y = z$ bit-wise (LSB-first).

We can now formally describe this adder automaton using the 5-tuple notation $(Q, \Sigma, \delta, q_0, F)$:

- $Q = \{0, 1\}$: The set of states, where State 0 represents "carry 0" and State 1 represents "carry 1".
- $\Sigma = \{(a, b, c) \mid a, b, c \in \{0, 1\}\}$: The alphabet consists of all 8 possible triplets of bits, representing the bits (x_i, y_i, z_i) from a single column.
- δ: The transition function codifies the rules of binary addition. For a given state (the incoming carry) and an input triplet, it transitions to the new state (the outgoing carry). For example, if the machine is in State 0 (carry 0) and reads the triplet ‘(1,1,0)‘, the sum $1 + 1 + 0$ correctly results in a sum bit of ‘0‘ (matching the z_i bit) and a carry-out of 1. Therefore, the transition is $\delta(0, (1, 1, 0)) \rightarrow 1$. Any triplet that is arithmetically incorrect (e.g., ‘(1,1,1)‘ from State 0) has no defined transition and leads to rejection.
- $q_0 = 0$: The automaton starts in State 0, as there is no incoming carry before the first (rightmost) column.
- $F = \{0\}$: The set of final states contains only State 0. This enforces the condition that the addition must conclude with a final carry of 0 for the statement $x + y = z$ to be true.

2.2.4. Types and Extensions of Automata

There are several kinds of automata. Some of these will be shown below:

- **Deterministic Finite Automata (DFA):** This is the type we have primarily seen in our examples. Its path is completely predictable. For any given state, and for any given input symbol, there is *exactly one* transition to follow. The machine has no choices to make; its journey through the states is uniquely determined by the input string.

- **Nondeterministic Finite Automata (NFA):** NFAs relax the strict rules of DFAs, introducing a form of "choice." From a given state on a given input symbol, an NFA might have several possibilities:

  – It can transition to *multiple* different states at once, as if it's exploring several paths simultaneously.

  – It can have *zero* transitions, causing a particular path to "die."

  – It can even change state *without reading any input symbol at all* (an ϵ-transition).

  An NFA accepts an input string if *any one* of its many possible paths ends in an accepting state. It only needs to find a single "winning" path to succeed. "Projection", which we will see later, is a process that initially creates an NFA.

- **Automata with Output (Transducers):** The automata in our examples only give a (yes/no) verdict (accept/reject). A different class of automata, known as *transducers*, instead transforms an input string into an output string. There are two main types:

  – **Moore Machines:** The output is determined only by the current state (any input that gets you to state "x" gives you output "y").

  – **Mealy Machines:** The output is determined by both the current state and the current input symbol (the specific input "a" bringing you to state "x" gives you output "y").

  In Chapter 3, we will use a type of transducer called a Deterministic Finite Automaton with Output (DFAO). These machines will take (the binary representation of) an index n as input and produce the representation of the n-th term of a sequence as output.

- **Automata on Infinite Words (e.g., Büchi Automata):** Some processes are not expected to terminate. To model and verify properties of such non-terminating systems, we use automata that operate on infinite input words. A Büchi automaton, for instance, accepts an infinite word if its path visits at least one of the designated accepting states infinitely often. This concept is central to the theory behind Büchi arithmetic.

- **Turing Machines:** A much more powerful model of computation is the Turing machine. Unlike finite automata, it includes an infinite tape that it can both read from and write to. This infinite memory makes it a theoretical model for a general-purpose computer. According to the Church-Turing thesis, anything that is intuitively "computable" can be computed by a Turing machine [12].

2.2.5. Walnut evaluation

Walnut is typically obtained by downloading its source code from a public repository, such as GitHub. The user then builds the application, a one-time process which compiles the Java code and prepares the tool for use. Once built, Walnut is run from a command-line terminal, which opens up an interactive session with a dedicated `[Walnut]$` prompt.

This environment is the user's workspace. Here, one doesn't use a graphical interface, but instead types commands directly. These commands allow the user to carry out a range of tasks: from defining the fundamental rules of a number system (for example, specifying that numbers should be interpreted in our

Narayana binary representation) to constructing automata and, most importantly, posing complex logical questions as formal statements to be proven true or false. One such evaluative statement, which we will analyze shortly, is:

```
eval test "?msd_2 An Em n<m";
```

To ask Walnut to prove a statement, we use the ‘eval‘ command, which has a clear and consistent structure. Let’s break down the components of a typical evaluation command:

- **The ‘eval‘ Keyword:** Every proof command must begin with ‘eval‘. This tells Walnut that its task is to evaluate the truth of a logical formula, rather than defining a new automaton (‘def‘) or performing another action.
- **The Evaluation Name:** Following ‘eval‘, you must provide a name for the evaluation, like ‘test‘ or ‘cheesytoast‘. This name is primarily for labeling the output files that Walnut generates, but any name will do.
- **The Numeral System (Optional Prefix):** Next, you can specify the numeral system for the variables in your formula. This is done with a prefix like ‘`?msd_2`‘ for standard binary or ‘`?msd_fib`‘ for the Fibonacci representation. If this prefix is omitted, Walnut will use its default system, which is ‘`?msd_2`‘.
- **The Logical Statement:** Finally, the logical formula itself must be enclosed in double quotes. This is the mathematical statement that you want Walnut to prove or disprove.

Putting it all together, a complete command has the structure: ‘$eval < name >$ " $< optional\ prefix >< formula >$ ";‘.

The command tells Walnut to use the *standard base-2 representation*, where we look at: *Most Significant Digit first*. We can break down what that means with an example string, `10011`.

The "msd" (Most Significant Digit) part tells Walnut the direction to read. As the leftmost bit is the most significant, it processes the string `10011` by reading the bits in sequence from left to right: `1`, then `0`, then `0`, then `1`, `1`, just like we used with the "$m < n$ statement".

The statement here checks whether for all non-negative integers n (as Walnut works with the natural numbers), there exists another non-negative integer m that is greater than n. This, of course, is true since for any number n, choosing $m = n+1$ always satisfies the condition $n < m$.

A final detail when entering commands into Walnut is the punctuation used to end the line. This character acts as a switch, allowing the user to choose the level of detail, or verbosity, in the output. The three options: ’a semicolon (‘;‘), a single colon (‘:‘), and a double colon (‘::‘)’, form a sequence of increasing detail.

- **Semicolon (‘;‘) for a Concise Answer:** If you only need the final verdict of a proof, ending the command with a semicolon tells Walnut to be as brief as possible. It will report only the ultimate TRUE/FALSE result.

  ```
  [Walnut]$ eval test0 "An Em m>n";
  ----
  TRUE
  ```

- **Colon (‘:‘) for a Summary of Steps:** If you want to see the main stages of the computation, using a single colon provides an intermediate level of detail. In addition to the final answer, Walnut outputs a log showing the key sub-formulas it evaluated, the number of states in the automata it constructed for each, and the time taken.

```
[Walnut]$ eval test0 "An Em n<m":
n<m: 2 states - 0ms
(E m n<m): 1 states - 13ms
(A n (E m n<m)): 1 states - 1ms
Total computation time: 16ms.
----
TRUE
```

- **Double Colon (‘::‘) for a Full Detailed Log:** For the most complete picture, a double colon provides the most verbose output. This will generate an extensive log detailing every minor step of the calculation, including information about intermediate automata. This "expert mode" is invaluable for in-depth debugging or a thorough analysis of Walnut’s procedure.

In short, the choice of terminator allows for an output that can be adjusted as needed: a semicolon for quick results, and a colon or double colon for a deeper look into the process.

Now, how is Walnut able to give a definite answer about the validity of this statement, considering it needs to check for every possible integer starting from 0? There are infinite numbers to check for, so intuition might suggest this would take infinite time. Yet, Walnut calculates these kinds of truth statements, and much more difficult ones, often in a matter of milliseconds!

As we already know, automata are the star players in this method. For this seemingly trivial statement, Walnut constructs and manipulates automata in a sequence of steps:

Step 1: Automaton for the ($n < m$) part

Input to this automaton: A pair of numbers, (m, n).

First, Walnut builds an automaton, we’ll call it $Automaton_{n<m}$. This automaton reads pairs of bits (m_i, n_i) –in Walnut always in alphabetical order, thus (m,n) no matter that in the statement they appear in order (n,m)– from the binary representations of m and n (typically MSD-first, with binary representations padded to the same length). It is designed to accept the pair of input words (w_m, w_n) if and only if the number represented by w_n is less than the number represented by w_m.

This automaton only requires a few states to track the comparison’s status as it reads the input from left to right. It uses one state for the condition "n=m so far," and another for when "n<m" has been definitively established. A third state for the "n>m" condition is implicit; any input pair not explicitly shown in the diagram is assumed to transition to this non-accepting "dead state."

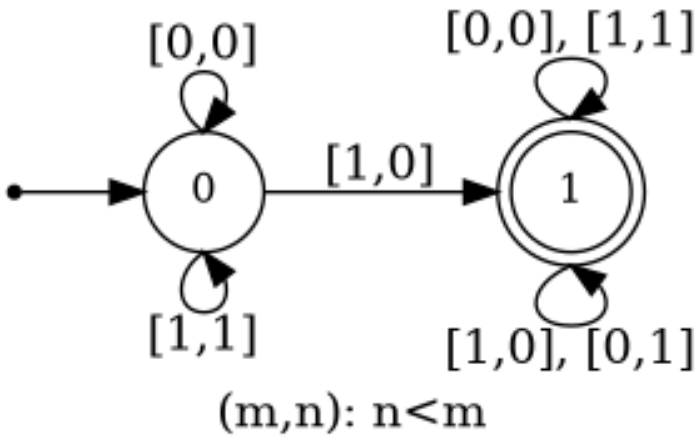

Figure 2.4: A DFA that accepts pairs of binary words (w_m, w_n) if and only if $n < m$, reading from Most Significant Digit first.

Step 2: Automaton for $\exists m(n < m)$ (Existential Quantification via Projection).

Input to this automaton: A single number, n.

Next, Walnut takes the automaton from Figure 2.9 and performs an operation called *projection* to handle the ‘Em‘ ("there exists an m"). It "forgets" about the m input and asks: "For a given input string w_n, is there any possible string w_m that could make the original automaton accept?"

This process involves creating a nondeterministic automaton (NFA) and then converting it to a deterministic one (DFA) using the subset construction. A first attempt at this process, based on a simple interpretation of comparing same-length strings, yields the DFA shown in Figure 2.5.

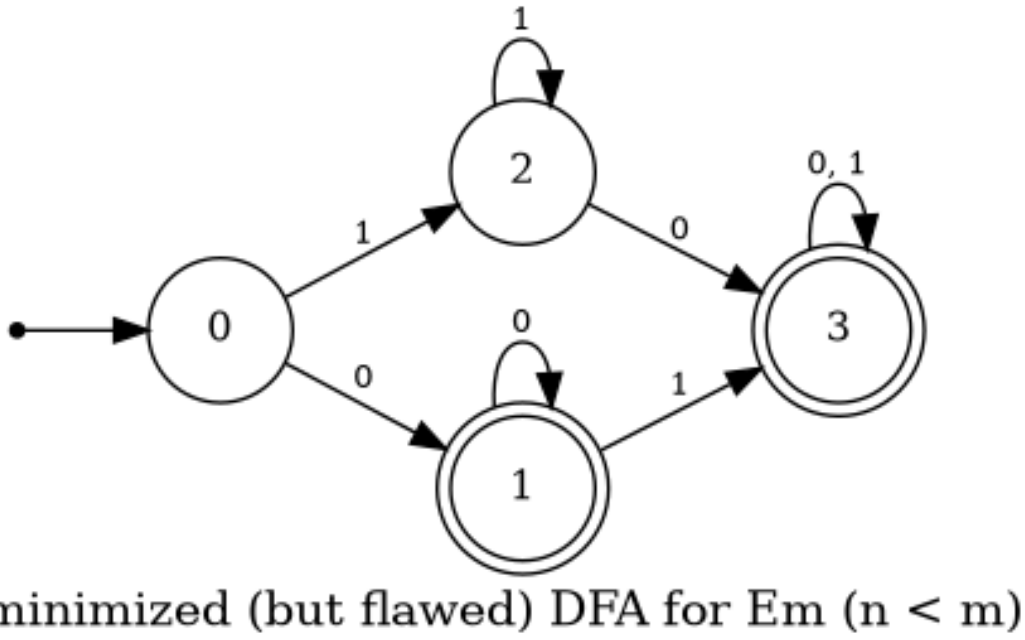

Figure 2.5: A preliminary DFA for $\exists m(n < m)$. While it correctly handles many cases, it contains a subtle flaw.

At first glance, this automaton seems plausible. For an input n starting with ‘0‘ (like the string ‘01‘), it correctly moves to the accepting state S_1, because we can choose an m that starts with ‘1‘ to guarantee $n < m$. However, this automaton fails for certain inputs. Consider an input string for n consisting of all ones, like $w_n = 111$. If we trace this input:

- Start in S_0. Read ‘1‘ → go to state S_2.
- In S_2. Read ‘1‘ → loop back to state S_2.
- In S_2. Read ‘1‘ (again) → loop back to state S_2.

The process ends in state S_2, which is non-accepting. The automaton incorrectly rejects "111".

The flaw in this automaton’s logic is that it doesn’t account for the fact that the "$\exists m$" allows us to choose an m that is represented by a string of a different length than n. For any number n represented by a string w_n of length L, we can always choose a number m that is larger. For example, we can choose m to be represented by the string ‘1‘ followed by L zeros.

To compare these, n is conceptually padded with a leading zero. For $n = 111$, we compare it to $m = 1000$ by looking at the padded strings $w_n = 0111$ and $w_m = 1000$. The very first bit-pair read by the original automaton would be ‘(0,1)‘, which immediately establishes $n < m$ and leads to an accepting state.

A correct automaton for $\exists m(n < m)$ must capture this possibility for any input n. Since we can always find a m that is larger, the correct automaton must accept all valid input strings for n. The minimization of this logic creates the simple one-state automaton shown in Figure 2.6.

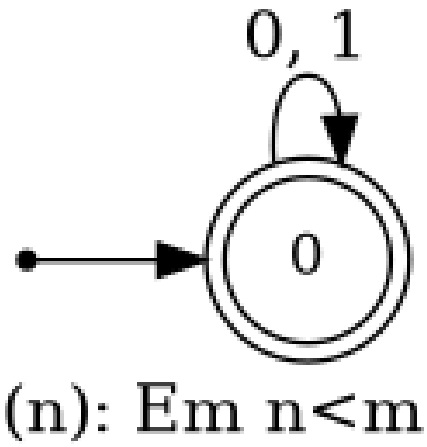

Figure 2.6: Final, minimal DFA for the statement $\exists m(n < m)$, (correctly) accepting all inputs.

Step 3: Automaton for $\forall n(\exists m(n < m))$ (Universal Quantification).

Input to this automaton: None. This is a closed formula, so the automaton represents a single TRUE/FALSE value.

Finally, Walnut takes the automaton from Step 2 (which accepts all w_n) and applies the 'An' (for all n). It checks if the language accepted by $Automaton_{\exists m(n<m)}$ is indeed the set of all possible strings w_n. Since the automaton from Step 2 already accepts everything for any n, this check passes. The final automaton for the entire closed statement $\forall n(\exists m(n < m))$ is again this very simple one-state automaton that accepts everything, signifying the original statement is TRUE.

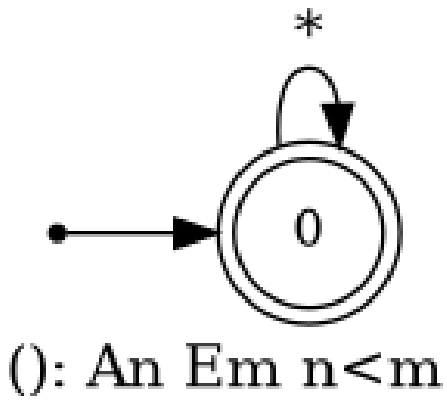

Figure 2.7: Since the statement has no free variables and is accepting, the automaton accepts all inputs, signified by the '*' loop on the start/accept state.

Walnut performs this same general process of building automata for parts of the total statement, and then applying operations corresponding to quantifiers and logical connectives (union for OR, intersection for AND) for any statement you give it. These operations on finite automata are algorithmic and always terminate, allowing it to decide the truth of complex statements over infinite domains.

2.2.6. Using Definitions to Build More Complex Proofs

While ‘eval‘ is the command for getting a final **TRUE/FALSE** answer, the ‘def‘ command is more powerful for building up a library of concepts. A ‘def‘ statement creates a reusable automaton that represents a specific property or relation. This new automaton can then be called in later proofs using a dollar sign, like ‘`$odd(n)`‘, making complex statements much cleaner to write and understand.

An example is defining what it means for a number to be "odd". We can create an automaton that checks this property and name it ‘odd‘:

```
def odd "Ek n=2*k+1;
```

This command tells Walnut: "Define a machine named ‘odd‘ that takes one input n, and returns TRUE if and only if there exists (‘Ek‘) an integer k such that $n = 2k+1$." From this point on, ‘`$odd(n)`‘ becomes a tool we can use in any other formula.

With this new tool in our arsenal, we can easily test other number-theoretic properties. For instance, we can investigate the classic rule that the sum of two odd numbers is even. That would look like this: "For all numbers, if m is odd AND n id odd, does that IMPLY their sum $m+n$ is also odd?"

This brings in three more logical operators:

- The ‘&‘ is the logical **AND** operator. It connects conditions that must all be true.
- The ‘=>‘ is the logical **IMPLIES** operator. A statement ‘P => Q‘ means "If P is true, then Q must also be true."
- The ’~’ is the logical **NOT** operator. It negates the truth value of what follows it. For example, ‘~$odd(k)‘ means "$k$ is not odd."

Here’s how we’d write that statement in Walnut:

```
eval test_odd_sum "Am,n ($odd(m) & $odd(n)) => ~$odd(m+n)";
```

When run, this command will return **TRUE**. Walnut arrives at this conclusion correctly. It creates an automata and will quickly see that only accepting states exist in it, thus the statement must be true. We could also ask the opposite, which is whether the sum of two odd numbers will always be odd. The statement for this:

```
eval test_odd_sum "Am,n ($odd(m) & $odd(n)) => $odd(m+n)";
```

This will give **FALSE**. Walnut works with automata and will quickly see that only rejecting states exist, but this can also been seen using for example the pair (1,3):

- The "IF" part, ‘`($odd(1) & $odd(3))`‘, is TRUE, since both 1 and 3 are odd.
- The "THEN" part, ‘`$odd(1+3)`‘, checks if ‘`$odd(4)`‘ is true. Since 4 is even, ‘`$odd(4)`‘ is FALSE.

Because we found a case where the premise is TRUE but the conclusion is FALSE, the entire implication ‘$TRUE$ => $FALSE$‘ is FALSE. This illustrates the core workflow in Walnut: you use ‘*def*‘ to create automata for fundamental properties (like "oddness"), and then you use ‘eval‘ to use these definitions with logical operators to ask complex questions about how these properties interact.

2.2.7. Regular Expressions

A powerful way to describe a set of strings -which in formal language theory is called a *language*-, is through the use of *regular expressions*. Before we see how this is used in in Walnut, a formal definition is given.

Definition 2.1 (Regular Expression)**.** *A regular expression is a sequence of characters constructed according to a few fundamental rules, which formally defines a set of strings (a regular language).*

In essence, a regular expression is a pattern built up from basic characters, using the operations for "union" and "repetition ".

Walnut allows us to build automata directly from these patterns using the '*reg*' command. We start with the keyword '*reg*', followed by a name for the automaton, the numeral system to be used (like 'msd_2'), and finally the regular expression itself, enclosed in double quotes.

We could define the property of being an odd number using a regular expression:

```
# Matches any string of 0s and 1s, followed by a final 1
reg odd msd_2 "(0|1)*1";
```

To understand this pattern, we need to look at the operators. The most important ones are:

- **Union (OR):** The operator '|' signifies a choice. The regex '$R_1|R_2$' matches any string that is matched by either R_1 or R_2. For example, '**0**|**1**' matches the string "0" or the string "1".
- **Kleene Star (*):** The Kleene star, written as $R*$, means "zero or more repetitions" of the expression R. For example, the regex '**10***' matches the string "1" followed by any number of 0s. This would match "1", "10", "100", "1000", and so on.
- **Parentheses '()'** Parentheses are used to group expressions together. Parentheses '()'For example, '**(10)***' means zero or more repetitions of the string between parentheses, here: "10", which would match " ", "10", "1010", etc.

With these rules, we can now decipher the pattern '**(0|1)*1**'. The part '**(0|1)***' matches any possible binary string of any length, and the final '**1**' ensures the number is odd, since any odd number in standard base-2 representation must end in a '1'.

2.2.8. Combining Automata to Generate Sequences

The automata discussed so far have been (yes/no) automata, but Walnut has another powerful command: '*combine*'. This is used for building automata that, given an input, will output values instead of (TRUE/FALSE).

The '*combine*' command works by taking several pre-defined (yes/no) automata. These automata must have the property that every number is accepted by exactly one of them and assign a specific output value to each one, dependent on which automata answered 'TRUE' for that number.

An example would be to create a sequence that outputs '1' for odd numbers and '0' for even numbers. First, we need the automaton for '$odd' which we defined in subsection 2.2.6 and add the even automaton. Now, this is just the automaton that accepts is '$odd' rejects and rejects if '$odd' accepts.

```
# Gives opposite output of '$odd'
def even "~$odd(n)";
```

With these two complementary automata defined, we can use '*combine*' to create a new sequence-generating automaton:

```
# Outputs 1 for odd numbers, 0 for even numbers
combine oddeven odd=1 even=0;
```

This command builds a machine that, when given an input n, will output 1 if the number (n) is accepted by '$odd' and output 0 if the number is accepted by '$even'.

2.2.9. Walnut with the Zeckendorf Representation

Before asking Walnut to prove anything, we must tell it what kind of numbers we are working with. The prefix '`?msd_fib`' instructs Walnut to interpret all variables (in this case, m and n) using the *Zeckendorf representation*. This system is built on the rule that any positive integer has a unique representation as a sum of non-consecutive Fibonacci numbers.

When written as a binary string, this sum results in a word that cannot contain two adjacent '1's. This grammatical rule makes it fundamentally different from the standard base-2 system, which has no such constraints. In essence, declaring '`?msd_2`' adds the set of powers of 2 to the underlying Presburger arithmetic, while declaring '`?msd_fib`' adds the set of Fibonacci numbers as the recognizable basis.

A simple comparison like $n < m$ is a Presburger-type statement, meaning it can be decided in any well-defined number system. However, the complexity of the automaton required to do so depends on the system's rules.

For standard base-2, the comparison is straightforward. For the Fibonacci representation, however, Walnut constructs a notably more complex 6-state automaton. This increased complexity is due to the machine simultaneously enforcing the "no adjacent 1s" grammar while also performing the comparison. This approach is questionable, as the '`?msd_fib`' directive should already guarantee the inputs are valid, a point we will analyze in more detail later. Furthermore, since Fibonacci representations are generally longer than their base-2 counterparts, we can expect a corresponding increase in computation time for these kind of Presburger operations within this system.

Let's see what happens when we ask Walnut to evaluate our familiar statement, $\forall n\ \exists m\ (n < m)$, but this time in the Fibonacci world:

```
[Walnut]$ eval testf3 "?msd_fib An Em n<m";
n<m:6 states - 0ms
 (E m n<m):2 states - 1ms
  (A n (E m n<m)):1 states - 0ms
----
TRUE
```

The log shows the same logical process as before: building an automaton for '$m < n$', then one for the existential quantifier 'Em', and finally for the universal quantifier 'An'. However, the automata themselves are different, as they now operate within the constraints of the "no adjacent 1s" grammar.

Step 1: The Automaton for $n < m$

First, Walnut needs a machine that can look at two numbers in Zeckendorf form, let's say represented by words w_n and w_m, and decide if $n < m$. The log reports that this requires a 6-state automaton, which is shown in Figure 2.8.

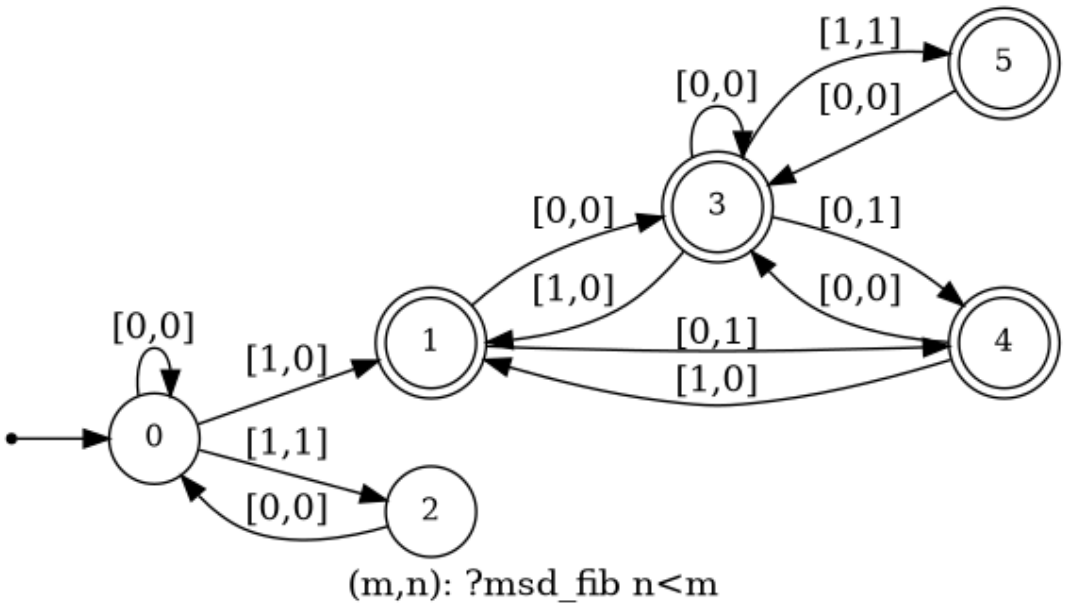

Figure 2.8: The 6-state DFA generated by Walnut for the predicate $n < m$ in the ‘msd_fib‘ system.

This machine is notably more complex than a standard binary comparator. This comes from the fact that it is performing two jobs at once: comparing the numbers from left to right (MSD-first) and simultaneously validating that the input strings themselves are valid Zeckendorf representations. The states must remember not just "are they equal so far?" but also "did I just see a 1?" in order to catch an illegal ‘11‘ pattern. For instance, from state 0, reading a ‘[1,1]‘ pair is a valid transition to state 2, but reading ‘[1,1]‘ again from state 2 is not a defined transition, which is how the grammar is enforced.

In truth, this is quite strange. The ‘`?msd_fib`‘ prefix already asserts that the variables m and n are validly formed Zeckendorf numbers. Therefore, an automaton that performs an additional, simultaneous check for the "no adjacent 1s" rule is, in principle, redundant. A simpler automaton, like the one for the standard base-2 case shown in Figure 2.9, would theoretically be sufficient for the comparison task for Zeckendorf as well.

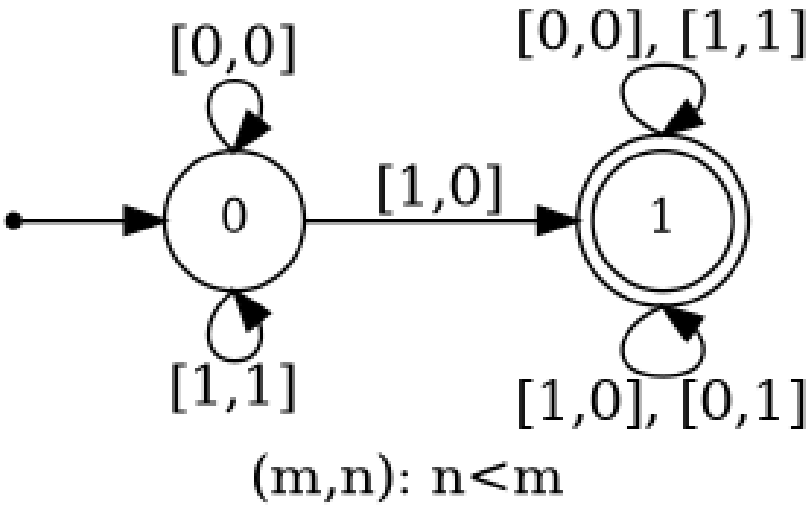

Figure 2.9: The simpler 3-state DFA for lexicographical comparison ($n < m$), without built-in grammar validation.

Step 2: The Automaton for $\exists m(n < m)$ after Projection
Next, Walnut handles the 'Em' ("there exists an m") by performing a *projection*. This operation asks: "For a given input n, could we have found any valid Zeckendorf number m that was bigger?" As we already know, the resulting automaton should accept all valid Zeckendorf inputs.

The log confirms that the automaton for this property has 2 states, as depicted in Figure 2.10.

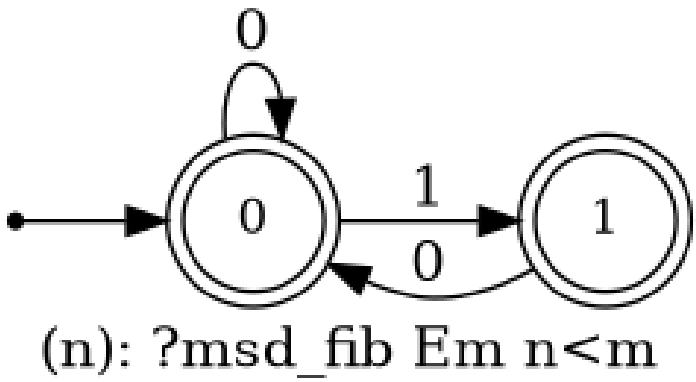

Figure 2.10: The 2-state minimal DFA for $\exists m(n < m)$ in the Fibonacci representation.

Here again, we see that Walnut produces an automaton whose entire function is to re-validate that the input string n adheres to the "no adjacent 1s" grammar. Again, this check is redundant.

Step 3: The Final Check with $\forall n$
Finally, Walnut takes the 2-state automaton from Figure 2.10 and checks if it accepts all numbers within the `?msd_fib` system. Since we've established that this machine is a perfect (if slightly redundant) validator for Zeckendorf strings, the answer is yes. The final result for the whole statement is TRUE. The automaton representing this final, absolute truth collapses to the minimal 1-state machine shown in Figure 2.11.

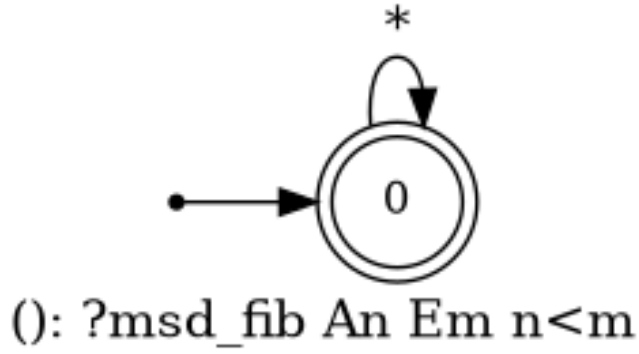

Figure 2.11: The final 1-state automaton for the closed statement $An\ Em\ m > n$, representing TRUE.

This automaton is perfectly logical. A closed statement that is universally true is represented by the simplest possible accepting machine: a single state that is both the start and final state, which accepts any input by looping back to itself. This correctly signifies that the statement holds for all numbers in the specified system.

3

Narayana in Walnut

In this chapter, Walnut's capabilities will be tested by conducting research on Narayana's numbers. Instead of manual proofs, we will formulate our questions in Walnut for him to solve. The work will begin by defining the essential automata for the Narayana system, most importantly the adder. This allows for arithmetic within this non-standard base. With these tools in place, we will then tackle a series of lemmas, but later also conjectures.

This chapter aims to provide a practical demonstration of how automated theorem proving can serve as a powerful ally in exploring the world of integer sequences.

3.1. Importing Narayana in Walnut

Walnut first needs to be taught the rules of the Narayana numeral system. Unlike standard base-2 or the Fibonacci representation, the Narayana system is not a built-in feature of the software. Before we can ask any questions about Narayana numbers, we must first provide Walnut with the fundamental automata that define the system.

Fortunately, these components have been developed by Jeffrey Shallit and are made available for download[1]. By loading these files into Walnut, we define two key automata:

- An automaton able to read and write in Narayana binary code.
- An automaton recognizing whether $x + y = z$ in Narayana arithmetic, thus making addition possible in Narayana code.

Once these are loaded, the Narayana system becomes a language that Walnut can speak, thus opening the door to proving Narayana theorems.

3.2. Proving Summation Identity with Walnut

Recalling Narayana's sequence and its binary system, several identities can be proven using walnut. One such identity was proven in chapter 1, lemma 1.13. Attempting to prove it directly in Walnut reveals some interesting aspects of how the software operates.

The method for proving it using Walnut turned out to be surprisingly tricky. The challenge lies in defining the left-hand side of the equation: the summation $\sum_{j=0}^{h} N_{i+3j}$. At first, I tried to define this using recursion, the natural way to define such a sum in any programming language. This however, is precisely what Walnut's 'def' command cannot do. A definition in Walnut can only return TRUE or FALSE, not a numerical value. For example, one cannot define a function that recursively builds upon itself like the following:

[1]The definitions for the Narayana numeral system, including the files 'msd_nara.txt' and 'msd_nara_addition.txt', can be found at `https://cs.uwaterloo.ca/~shallit/papers.html`.

```
# INVALID WALNUT COMMAND: Cannot return a value
def Sum(h) = Sum(h-1) + N_i+3h;
```

Instead of trying to define the sum recursively though, we can try to find a pattern in the Narayana binary representations. This, because if these sets of numbers can be described by *regular expressions*, we can use Walnut's 'reg' command to build automata for them.

The left-hand side (LHS) of the lemma, $S_h = \sum_{j=0}^{h} N_{i+3j}$, generates a sequence of partial sums. The right-hand side (RHS) of the corrected identities generates a sequence of single Narayana terms N_{i+3h+1} where '1' is being subtracted from these terms each time.

After observing the Narayana binary strings for both sides, it appears that the sides can be captured by surprisingly simple regular expressions. Remember that in Narayana code, if the "1's" are 3 or more spaces apart, the code is valid. Thus, adding terms like $N_i, N_{i+3}, N_{i+6}, \ldots$ (3 indices apart) can be captured surprisingly easily. The binary representation of the sum on the LHS is simply the result of setting the bits corresponding to each term to '1'. This creates a repeating pattern, as shown in the following tables for the LHS of the lemma.

Table 3.1: Binary patterns for the LHS for each case of i.

	Case: $i = 1$		**Case:** $i = 2$		**Case:** $i = 3$	
h	**Summation**	**Binary String**	**Summation**	**Binary String**	**Summation**	**Binary String**
0	N_1	1	N_2	10	N_3	100
1	$N_1 + N_4$	1001	$N_2 + N_5$	10010	$N_3 + N_6$	100100
2	$N_1 + N_4 + N_7$	1001001	$N_2 + N_5 + N_8$	10010010	$N_3 + N_6 + N_9$	100100100

Similarly, the RHS of the identities, which involves the terms N_{i+3h+1}, also produces numbers whose binary representations follow a clear pattern. The table below shows the representations for the main term of the RHS, ignoring the fact that '1' should still be subtracted from each Narayana term.

Table 3.2: Binary patterns for the main RHS term for each case of i.

	Case: $i = 1$		**Case:** $i = 2$		**Case:** $i = 3$	
h	**Term**	**Binary String**	**Term**	**Binary String**	**Term**	**Binary String**
0	N_2	10	N_3	100	N_4	1000
1	N_5	10000	N_6	100000	N_7	1000000
2	N_8	10000000	N_9	100000000	N_{10}	1000000000

From these tables, the patterns are evident. The strings for the summations on the LHS appear to be formed by repeating blocks of '100' followed by a specific suffix, while the strings on the RHS (still ignoring the '-1') are a single '1' followed by a growing number of zeros. This suggests they can be captured by regular expressions.

The regular expressions for which the numbers on the *LHS* for $i = 1, 2, 3$ can be captured, are: '**0*(100)*1**', '**0*(100)*10**', and '**0*(100)*100**' respectively. Similarly, the *RHS* terms seem to follow patterns like '**0*10(000)***', '**0*100(000)***', and '**0*1000(000)***'. The leading '0*' is added to these. to handle arbitrary-length words in Walnut. Without such handling for leading zeros, the automaton would fail to match equivalent representations like '101' and '00101', which would break the proof.

3.2.1. The Proof for the Case $i = 1$

Since Walnut doesn't support a more dynamic way to use the reg function, we can't test for the general case for i=1,2,3 but only at separate cases. We take the case where the summation starts with $i = 1$. Based on the patterns observed in Tables 3.1 and 3.2, we can be sure that the sets of numbers on the left-hand side (LHS)

and right-hand side (RHS) of the lemma are able to be described by simple regular expressions. Defining automata for these sets in Walnut:

```
# Defines the set of numbers on the LHS for 'i'=1
reg nar1l msd_nara "0*(100)*1";

# Defines the set of numbers on the RHS (ignoring the '-1') for 'i'=1
reg nar1r msd_nara "0*10(000)*";
```

To prove the lemma (for $i = 1$), we need to show that for every number m in the set defined by '$nar1l$', the number $m+1$ is in the set defined by '$nar1r$' and vice versa for the number n in the set defined by '$nar1r$' and $n-1$ in the set defined by '$nar1l$'. This mapping should be a one-to-one correspondence. The reason for this will be explained at the end.

Part 1: Mapping from LHS to RHS

First, we ask Walnut: "For every number m that belongs to the set '$nar1l$', does there exist a corresponding number n in the set '$nar1r$' such that $n = m+1$?"

```
eval test "?msd_nara Am $nar1l(m) => En ($nar1r(n) & n=m+1)";
----
TRUE
```

Walnut's '**TRUE**' result confirms this. It tells us that for any number we can construct using the '$nar1l$' pattern, we are guaranteed to find a number in the '$nar1r$' pattern that is exactly one greater. This establishes that a mapping from the LHS set to the RHS set exists.

Part 2: Mapping from RHS to LHS

Next, we must ensure the mapping is not just one-way. We need to check if every number in the RHS set also has a partner in the LHS set. We ask the reverse question: "For every number n that belongs to the set '$nar1r$', does there exist a corresponding number m in the set '$nar1l$' such that $m = n-1$?"

```
eval test "?msd_nara An $nar1r(n) => Em ($nar1l(m) & m=n-1)";
----
TRUE
```

Walnut again returns '**TRUE**'. This confirms that the mapping goes both ways: not only does every LHS number have a partner in the RHS, but every RHS number also has a partner in the LHS.

Conclusion: The One-to-One Correspondence

Together, these two proofs establish that there is a one-to-one correspondence between the set of numbers on the LHS, defined by '`nar1l`', and the set on the RHS, defined by '`nar1r`', under the relation $n = m+1$.

This is a powerful result, but it doesn't immediately tell us that the k-th smallest number on the LHS maps to the k-th smallest number on the RHS. However, we also know that both of these sets of numbers are strictly increasing. Let's call the ordered set for '`nar1l`' $\{m_1, m_2, m_3, \ldots\}$ and for '`nar1r`' $\{n_1, n_2, n_3, \ldots\}$.

If the mapping were not order-preserving, it would create a contradiction. For example, suppose the second LHS number (m_2) mapped to the third RHS number (n_3), and the third LHS number (m_3) mapped to the second RHS number (n_2). This would mean $n_3 = m_2 + 1$ and $n_2 = m_3 + 1$. Since the sets are strictly increasing, we know $m_2 < m_3$, which implies $m_2 + 1 < m_3 + 1$, and therefore $n_3 < n_2$. This contradicts the fact that the '`nar1r`' set is also strictly increasing ($n_2 < n_3$).

Therefore, the only possible mapping is the one that preserves the order: the smallest number in ‘nar1l‘ must map to the smallest in ‘nar1r‘, the second-smallest to the second-smallest, and so on. This confirms that the h-th sum on the LHS corresponds to the h-th term on the RHS, proving the lemma for the case $i = 1$.

3.2.2. The Proof for the Cases $i = 2$ and $i = 3$

The same logic can be applied for the cases $i = 2$ and $i = 3$, simply by using the regular expressions that correspond to those starting indices.

```
# LHS numbers for 'i'=2
reg nar2l msd_nara "0*(100)*10";
# RHS numbers for 'i'=2
reg nar2r msd_nara "0*100(000)*";

# LHS numbers for 'i'=3
reg nar3l msd_nara "0*(100)*100";
# RHS numbers for 'i'=3
reg nar3r msd_nara "0*1000(000)*";
```

Running the same pair of ‘$eval$‘ commands for ‘$nar2l$‘/‘$nar2r$‘ and ‘$nar3l$‘/‘$nar3r$‘ also yields ‘**TRUE**‘ in all cases. Regular expressions can thus be used as a workaround to other methods other programming languages would use.

Unfortunately Walnut doesn't currently support a more dynamic way to define these patterns. For instance, by allowing a variable in the regular expression like ‘0*(100)*10{i-1}‘. Here, '0{i-1}' means that the 0 would be added 'i-1' amount of times. This would have allowed a single proof for all $i \in \{1,2,3\}$. Instead, we have to construct and verify the logic separately for each case.

3.3. Research on Narayana and Hofstadter

Building on the methods established by Shallit which can be seen in the appendix, my research explores a new sequence.

3.3.1. A Sequence Derived from Narayana's Representation

First, let's briefly revisit the key sequences from prior work that form the basis of this investigation. We have the sequence $h(i)$, defined as $h(i) = [m_1 \dots m_{t-1}]_N + m_t$, which was shown by Shallit to be the exact same as the Hofstadter H-sequence: $A005374$ in Appendix B. This sequence has the property: $H(i) = i - H(((i-1)))$ for $i \geq 1$, where $H(0) = 0$. This previously defined automata Appendix B, will be a useful tool.

My first investigation began with a simple question: what if we define a new sequence, let's call it $h'(i)$, based on the prefix of length $t-2$ and the sum of the last two bits.

$$h'(i) = [m_1 \dots m_{t-2}]_N + m_{t-1} + m_t$$

Visualizing this sequence using a table:

Table 3.3: Calculation of the first few terms of the sequence $h'(i) = [m_1 \dots m_{t-2}]_N + m_{t-1} + m_t$.

Index i	$[m_1 \dots m_t]_N$	**Prefix** $[m_1 \dots m_{t-2}]_N$	**Value of Prefix**	m_{t-1}	m_t	**Result** $h'(i)$
0	‘0* 00‘	’0*’	0	0	0	$0+0+0=$ **0**
1	‘0* 01‘	’0*’	0	0	1	$0+0+1=$ **1**
2	‘0* 10‘	’0*’	0	1	0	$0+1+0=$ **1**
3	‘0*1 00‘	‘0*1‘	1	0	0	$1+0+0=$ **1**
4	‘0*10 00‘	‘0*10‘	2	0	0	$2+0+0=$ **2**
5	‘0*10 01‘	‘0*10‘	2	0	1	$2+0+1=$ **3**
6	‘0*100 00‘	‘0*100‘	3	0	0	$3+0+0=$ **3**

This definition is based on the bit string of i. To implement this in Walnut, I created an automaton named ‘h_new‘.

```
# Accepts if one of the two bits equals 1
reg lastbit2 "?msd_nara (0|1)*01|(0|1)*10";

# Accepts if for ‘i’=[m_1..m_t]N we get ‘z’=[m_1...m_(t-2)]N+’m_(t-1)’+’m_t’
def h_new "?msd_nara Ex,y $rshift(i,x) & $rshift(x,y) &
    ((z=y+1 & $lastbit2(i)) | (z=y & ~$lastbit2(i)));
```

This definition works by first applying ‘$rshift‘ twice, which chops off the last two bits of i to get the number represented by the prefix, $y = [m_1 \dots m_{t-2}]_N$. Then, it uses ‘$lastbit2(i)‘ to check if the sum of the last two bits, $m_{t-1} + m_t$, is 1 or not (the sum can only be 0 or 1, since ‘11‘ is an illegal ending). If the sum is 1, it asserts that $z = y + 1$; otherwise, it asserts that $z = y$.

I then discovered what appeared to be a more abstract, but equivalent, way to define this same sequence using Shallit’s $h(i)$. I hypothesized that my sequence could be described by the formula $h(h(i) - m_t) + m_t$. Defining this sequence was done under the name: ‘*h_NEW*‘, with the help of *c*0.

```
# Accepts if ‘x’ binary representation ends in ..01, thus ’m_t=1’
def c0 "?msd_nara Ei $col0(i,x)";

# Accepts if ’z’ takes form: ‘h(h(i)-m_t) +m_t’ for an ‘i’
def h_NEW "?msd_nara Ea,b $h(i,a) &
    (($c0(i) & $h(a-1,b) & z=b+1) | (~$c0(i) & $h(a,b) & z=b));
```

The test here was to see whether the bit-wise definition (‘*h_new*‘) and my more abstract, sequence-based definition (‘*h_NEW*‘), were actually the same.

```
eval test "?msd_nara Ai,z $h_NEW(i,z) <=> $h_new(i,z)";
```

Walnut returned **TRUE**. This proves that indeed, the rule I devised was valid.

3.3.2. Comparison of Two Bit-Wise Defined Sequences

The second part of my research involved another custom sequence, $h''(i)$, defined as:

$$h''(i) = [m_1 \dots m_{t-2}]_N + [m_{t-1} m_t]_N$$

Here, we add the number represented by the final two bits as a pair. My hypothesis was that the difference between this sequence and the previous one: $d(i) = h''(i) - h'(i)$, would be a very simple sequence of 0s and 1s. For the value $d(i)$ in the sequence, it would be 1 if and only if the Narayana representation of i ended in ‘...10‘, and 0 otherwise.

To define this function, some helper functions are needed:

```
# Accepts if for input ‘(i,j)’, ’j’ has the same narayana representation as ’i’ except for a ‘0’
reg lshift {0,1} {0,1} "([0,0]|[0,1][1,1]*[1,0])*":

# Accepts if ‘x’ is the ‘i’th number in a sequence where each number ends in ..10
def col1 "?msd_nara Ex $col0(i,x) & $lshift(x,z)";

# Accepts if ‘x’ binary representation ends in ..10, thus ’m_(t-1)=1’
def c1 "?msd_nara  Ei $col1(i,x)";
```

I defined $c1$ here, which shows whether m_{t-1} equals 1. This was done using \$lshift on \$col0, which adds a ’0’ on the right for a $'col0'$ object, so instead of the ’1’ being on the utmost right, the object now ends in ..10. Then for the function $h''(i)$, I defined this in Walnut as the automaton ‘h_NEW2‘:

```
# Accepts if for ‘i’=[m_1..m_t]N we get ‘z’=[m_1...m_(t-2)]N+[m_(t-1)+(m_t)]N
def h_NEW2 "?msd_nara Es,t $rshift(i,s) & $rshift(s,t) &
    (($c1(i) & z=t+2) | ($c0(i) & z=t+1) | (~$c1(i) & ~$c0(i) & z=t));
```

- $[m_1 \dots m_{t-2}]_N + [m_{t-1} m_t]_N$ equals just $[m_1 \dots m_{t-2}]_N$ if $m_{t-1} m_t$ is 00.
- If $m_t = 1$ ($m_{t-1} = 0$ automatically), $m_{t-1} m_t$ is 01 corresponds with ’1’.
- If $m_{t-1} = 1$ ($m_t = 0$ automatically), $m_{t-1} m_t$ is 10 corresponds with ’2’.

With both sequences defined (‘h_NEW‘ for h' and ‘h_NEW2‘ for h''), I could define their difference sequence, ‘diff‘:

```
# Calculates sequence where h_NEW is subtracted from h_NEW2
def diff "?msd_nara Ex,y $h_NEW(i,x) & $h_NEW2(i,y) & z=y-x;
```

The final step was to define a sequence that is ‘1‘ for the set of numbers ending in ‘...10‘ and ‘0‘ otherwise. This can be done with the ‘*combine*‘ command in Walnut. In our case, we want to partition all numbers into two sets: those that are accepted by our automaton ‘c1‘ (the ones ending in ‘...10‘), and those that are not. First, we define an automaton for the second set using logical NOT (‘~‘):

```
# Accepts the numbers c1 rejects and rejects the numbers c1 accepts
def not_c1 "?msd_nara ~$c1(x)";

# For the numbers c1 accepts, T[i] gives 1, for the numbers c1 rejects, T[i] gives 0
combine T c1=1 not_c1=0;
```

The final proof is an evaluation that checks, for all i, if the value of the difference sequence ‘diff‘ matches the output of ‘T‘.

```
# Evaluates whether the numbers in the sequence: ‘diff’, equal 1
# if and only if the numbers’ binary narayana representation ends
#  in ...10 and the other numbers equal 0 if and only if the binary
#  representations don’t end like that
eval test "?msd_nara Ai ($diff(i,0)<=>T[i]=0) & ($diff(i,1)<=>T[i]=1)";
```

Walnut returns **TRUE**. This confirms that the difference between my two custom-defined sequences has the property of being ‘1‘ precisely at the indices whose Narayana representation ends in ‘...10‘, and ‘0‘ everywhere else.

Conclusion

In this thesis, we explored the world of Narayana's cow sequence from a historical perspective to its intricate mathematical properties. We illustrated how classical methods of mathematics, such as induction, and even Cardano's methods, could be applied to uncover the sequence's 'hidden' secrets. These include its formula and the supergolden ratio which governs its growth.

The main goal in this, however, was to prove how the automatic theorem prover, Walnut, could be of crucial help is discovering these 'secrets'. It was explained what Walnut was capable of and how to use its power to prove theorems with 100% certainty. It became clear that 'automata' were the backbone in all of this.

As shown in the last chapter, where Walnut was demonstrated to be integral in proving theorems about integer sequences -in specific Narayana's–, Walnut can greatly help in number theory. It provides the ability to analyze relationships and validate conjectures, which could be almost impossible to verify with hand computations. In many ways, Walnut feels like a preview of what the future of mathematical exploration might look like.

Bibliography

[1] Parmanand Singh. “The so-called Fibonacci numbers in ancient and medieval India”. In: *Historia Mathematica* 12.3 (1985), pp. 229–244. DOI: 10.1016/0315-0860(85)90021-7.

[2] George Gheverghese Joseph. *The Crest of the Peacock: Non-European Roots of Mathematics*. 3rd. Princeton, NJ: Princeton University Press, 2011. ISBN: 978-0691135267.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Upper Saddle River, NJ: Addison-Wesley Professional, 2011. ISBN: 978-0201038040.

[4] Christiane Frougny. “How to write a number in a rational base”. In: *LATIN ’92*. Ed. by Ireneo Lasota. Vol. 583. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 154–164. DOI: 10.1007/BFb0027506.

[5] Christiane Frougny. “Confluent linear numeration systems”. In: *Theoretical Computer Science* 178.1-2 (1997), pp. 233–253. DOI: 10.1016/S0304-3975(96)00159-9.

[6] Ian Stewart. “A Guide to Ugly Numbers”. In: *Scientific American* 275.5 (Nov. 1996), pp. 102–103. DOI: 10.1038/scientificamerican1196-102.

[7] Victor J. Katz. *A History of Mathematics: An Introduction*. 3rd. Boston: Addison-Wesley, 2009. ISBN: 978-0321387004.

[8] Dirk J. Struik. *A Concise History of Mathematics*. 4th Rev. Ed. New York: Dover Publications, 1987. ISBN: 978-0486602554.

[9] David S. Dummit and Richard M. Foote. *Abstract Algebra*. 3rd. Hoboken, NJ: John Wiley & Sons, 2004. ISBN: 978-0471433347.

[10] Jeffrey Shallit. *Walnut: Automatic Theorem Prover for Automatic Words*. Accessed on [Date you accessed it, e.g., May 16, 2024]. 2024. URL: https://cs.uwaterloo.ca/~shallit/walnut.html.

[11] J. Richard Büchi. “On a decision method in restricted second order arithmetic”. In: *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*. Ed. by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Stanford, CA: Stanford University Press, 1962, pp. 1–11.

[12] Michael Sipser. *Introduction to the Theory of Computation*. 3rd. Boston, MA: Cengage Learning, 2012. ISBN: 978-1133187790.

[13] Jeffrey Shallit. “The Narayana Morphism and Related Words”. In: *arXiv preprint arXiv:2503.01026* (Mar. 2025). arXiv: 2503.01026 [math.CO].

A

Appendix

A Goldbach-like Conjecture for Narayana Numbers

A famous unsolved problem in number theory, Goldbach's Conjecture, states that every even integer greater than 2 is the sum of two primes. Unfortunately, prime-numbers can only be defined using 2nd order logic, Walnut won't be able to prove it. That said, it did inspire me to investigate if a property of the same kind holds for Narayana numbers. Can every even Narayana number greater than 1 be expressed as a sum of other Narayana numbers, those numbers being odd, even, or a sum of both?

This is exactly the kind of question Walnut is built to answer. To get started, we first need a way to tell Walnut what an "odd" and "even" number is within our Narayana system.

Defining the necessary Automata

We already defined an odd function in subsection 2.2.6, but this was in the (default) base-2 representation. We should also define it in Narayana representation, as should we with an 'even' function:

```
def odd "?msd_nara Ek n=2*k+1";
def even "?msd_nara ~$odd(n)";
```

Secondly, we should define a function which accepts Narayana numbers. The k'th distinct Narayana number in Narayana representation is represented by a '1' in the k'th position with there being 0's for the rest. The exception, is the the Narayana number: (0), which has just representation of 0's all around:

```
reg isnara msd_nara "0*(0|1)0*";
```

Testing hypotheses

Hypothesis 1: Even Narayana Numbers as a Sum of Two Odd Terms

Conjecture A.1. *Every even Narayana number greater than 1 can be written as the sum of two odd Narayana numbers.*

Disproof. The conjecture is formulated as a first-order logical statement and passed to Walnut for evaluation.

For all k, if k is a valid, even Narayana number greater than 1, then there exist two odd Narayana numbers, m and n, that sum to k

In Walnut:

```
eval test "?msd_nara Ak (($isnara(k) & $even(k) & k>1) =>
    (Em,n (($isnara(m) & $odd(m)) & ($isnara(n) & $odd(n)) & k=m+n)))";
```

Walnut returns **FALSE** for this, which disproves our conjecture.

□

This result can be checked by making an automaton which accepts for numbers that defy our hypothesis:

```
# Accepts for input 'k' if it is the sum of 2 odd Narayana numbers
def sumodd "?msd_nara Em,n
    (($isnara(m) & $odd(m)) & ($isnara(n) & $odd(n)) & k=m+n)";

# Accepts for input 'k' if it is a number that defies our hypothesis
def ex "?msd_nara ~$sumodd(k)&$isnara(k)&$even(k)";
```

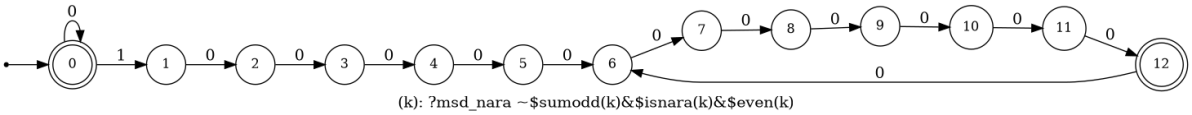

Figure A.1: Automaton 'ex'

As can be seen, there exists an accepted state (double circle on state 12), which means there exist numbers that defy our hypothesis. The number with Narayana representation beginning with a 1 followed by eleven 0's, which the 'ex' automaton accepts, is an example of this. This number is 88.

Hypothesis 2: Even Narayana Numbers as a Sum of Three Terms

Since the first idea didn't pan out, we relax the condition a bit.

Conjecture A.2. *Every even Narayana number greater than 1 is the sum of two odd Narayana numbers and one even Narayana number.*

Proof. Formulating the conjecture so Walnut can proof it:

For all k**, if** k **is a valid, even Narayana number greater than** 1**, then there exist three Narayana numbers** m,n **and** p **such that** m **and** n **are odd,** p **is even and "**$k = m+n+p$**"**

In Walnut:

```
eval test "?msd_nara Ak ($isnara(k) & $even(k) & k>1) =>
    Em,n,p (($isnara(m) & $odd(m)) & ($isnara(n) & $odd(n))
            & ($isnara(p) & $even(p)) & k=m+n+p)";
```

Walnut this time returns **TRUE**, which constitutes as a formal proof that the statement holds

□

This result is interesting because Narayana numbers themselves are built from sums (since $N_k = N_{k-1} + N_{k-3}$). The fact that we can still impose these odd/even constraints and find a representation for any even Narayana number greater than 1 is interesting. This, in turn, gives rise to a follow-up question: what about the odd Narayana numbers? Can they also be decomposed in a similar way?

Hypothesis 3: Odd Narayana Numbers as a Sum of Three Terms

A similar hypothesis to our previous one, would be that every odd Narayana number is the sum of one odd and two even Narayana numbers. Without an extra condition however, this would be trivially true, as we could always pick the two even numbers to be 0 and the odd number to be the original number itself. We fix this by requiring that at least one of the even numbers in the sum is greater than 0.

Conjecture A.3. *Every odd Narayana number greater than 1 is the sum of one odd, one even and one positive even Narayana number.*

Proof. Translating the conjecture into 1st order logic so that Walnut can determine the validity:

```
eval test "?msd_nara Ak ($isnara(k) & $odd(k)&k>1) =>
    Em,n,p (($isnara(m) & $odd(m)) & ($isnara(n) & $even(n))
        & ($isnara(p) & $even(p) & p>0) & k=m+n+p)";
```

Walnut returns **TRUE** again, thus the validity of the statement is proven.

□

To conclude, we now know any Narayana number greater than one can be decomposed into a sum of three other Narayana numbers with specific odd/even constraints. This pattern would be difficult to confirm without Walnut.

B

Appendix

The Sequence $h(i)$ and its Connection to Hofstadter

Recall Narayana representation. Any integer i has a unique Narayana binary representation, which is a string of bits $m_1 m_2 \dots m_t$. We can write this as $i = [m_1 m_2 \dots m_t]_N$, where t is the number of bits used to represent the number i. Shallit introduces a sequence $h(i)$ [13], defined as follows:

$$h(i) = [m_1 \dots m_{t-1}]_N + m_t$$

Table B.1: Calculation of the first few terms of Shallit's sequence $h(i) = [m_1 \dots m_{t-1}]_N + m_t$.

Index i	$[m_1 \dots m_t]_N$	**Prefix** $[m_1 \dots m_{t-1}]_N$	**Value of Prefix**	m_t	**Result** $h(i)$
0	‘0*0‘	‘0*‘	0	0	$0+0=\mathbf{0}$
1	‘0*1‘	‘0*‘	0	1	$0+1=\mathbf{1}$
2	‘0*10‘	‘0*1‘	1	0	$1+0=\mathbf{1}$
3	‘0*100‘	‘0*10‘	2	0	$2+0=\mathbf{2}$
4	‘0*1000‘	‘0*100‘	3	0	$3+0=\mathbf{3}$
5	‘0*1001‘	‘0*100‘	3	1	$3+1=\mathbf{4}$
6	‘0*10000‘	‘0*1000‘	4	0	$4+0=\mathbf{4}$

Note: The ‘0‘ notation used in the table signifies that in a binary representation, any number of leading zeros can precede the first ‘1‘ without changing the value of the number.*

Thus, you take the Narayana representation of i, chop off the last bit, find the number that the remaining prefix represents, and then simply add the value of that last bit (0 or 1) to it.

We can see that the sequence $h(i)$ starts: $0,1,1,2,3,4,4,\dots$. When looking this up in the On-Line Encyclopedia of Integer Sequences (OEIS), it appears to match sequence $A005374$, the famous Hofstadter H-sequence, defined by: $H(0) = 0$ and $H(i) = i - H(H(H(i-1)))$. Proving this connection is a perfect job for Walnut.

To do this, we first need to define the sequence $h(i)$ in Walnut using its bit-wise rule. This requires two helper automata: one to handle the "chopping off the last bit" part (‘$rshift$‘) and another to check what that last bit is (‘$lastbit1$‘).

```
# Accepts if for input ‘(i,j)’, ’j’ has the same narayana representation
# as ’i’ except for the uttermost right bit being deleted
reg rshift {0,1} {0,1} "([0,0]|[1,0][1,1]*[0,1])*(()|[1,0][1,1]*)":

# Accepts if last bit equals 1
reg lastbit1 msd_nara "(0|1)*1":
```

The automata for '$rshift$' was found by Shallit. While a formal proof of for why it works is omitted for brevity, it will be a tool in the definitions that follow.

With these regular expressions, we can define $h(i,z)$ (which return 'TRUE' if $z = h(i)$).

```
# For input ‘i’, accepts if ‘z’=[m_1 ... m_(t-1)]N + ’m_t’
def h "?msd_nara Ex $rshift(i,x) &
    ((z=x+1 & $lastbit1(i)) | (z=x & ~$lastbit1(i)));
```

This definition perfectly mirrors the rule: it finds the number x (that is i right-shifted), and then defines z as either $x+1$ (if the last bit of i was 1) or just x (if the last bit of i was 0). Now, we can check if this sequence is the same as Hofstadter's H-sequence using its recursive definition:

```
# Evaluates whether indeed ’h(i)=i-h(h(h(i-1)))’
eval hcheck "?msd_nara Ai,x,y,z,w (i>=1 & $h(i-1,x) & $h(x,y) &
 $h(y,z) & $h(i,w)) => i=w+z":
```

Walnut returns **TRUE**, confirming that these two very different-looking definitions produce the same sequence.

From Hofstadter to A202342

Shallit explores properties of this $h(i)$ sequence as well. For instance, does every positive integer appear as a value in the sequence? And does any value appear three or more times? Walnut can prove these statements easily:

```
# Evaluates whether ‘h’ can take on each positive number
eval every "?msd_nara An Ex $h(x,n)":

# Evaluates whether ‘h’ takes on each positive number less than 3 times.
eval nothree "?msd_nara ~En,x,y,z x<y & y<z & $h(n,x) & $h(n,y) & $h(n,z)":
```

Both commands return **TRUE**. This tells us that every positive integer appears as a value in the sequence $h(i)$, but never more than twice. A next question: 'how many times does each number appear?'. Sequence A202340 in the OEIS counts exactly this, from which we can define two sets of indices: those whose $h(i)$ value appears once, and those whose $h(i)$ value appears twice (this latter set of indices forms sequence A202342).

The relationship between these sequences is best understood with an example, as shown in Table B.2.

Table B.2: The relationship between $h(i)$ ($A005374$), its value frequencies ($A202340$), and the resulting index sequence $A202342$.

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12
A005374	0	1	1	2	3	4	4	5	5	6	7	7	8
A202340	1	2	2	1	1	2	2	2	2	1	2	2	1
A202342	1	4	5	7	10	13	14	17	18	20	23	24	26

To prove any properties about these sequences, an automaton for $A202342$ is needed. We know that Walnut can only verify, thus we need to guess for this sequence and let Walnut verify whether this automaton is correct.

Shallit provides a guessed automaton for $A202342$, which are the numbers whose values in h appear twice. He names this automaton ‘$a202342‘. It has 6 states and is shown in Figure B.1.

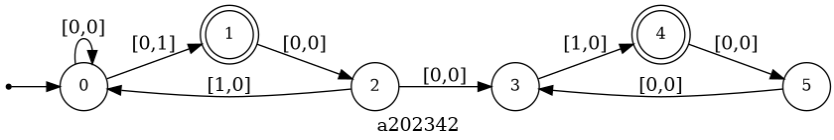

Figure B.1: The 6-state DFA ‘$a202342‘ that accepts an index i if its value $h(i)$ is a number that appears twice in the sequence h.

The proof (using Walnut), that this indeed forms the right automaton can be found in the paper [13].

The following connection is then stated: $A202342(n) + n = A020942(n)$. A number is in $'A020942'$ if and only if its Narayana representation ends with a ‘1‘. Walnut can verify this complex arithmetic identity. This is done by using our automaton ‘$a202342(i,z)‘ (true if z is the i-th number in the sequence $A202342$) to define the left hand side. We then assume this defines the sequence $'A020942'$.

```
# Defines function which allegedly accepts for ‘x’ if it is the ‘i’th number
# that has Narayana representation ending in 1
def col0 "?msd_nara $a202342(i,x-i);
```

This definition for ‘$col0(i,x)‘ is true if $x - i$ is the i-th term of $A202342$, which is just a rearrangement of $x = A202342(i) + i$.

Now we can prove the theorem. We check if the set of numbers accepted by ‘$col0$‘ is equivalent to the set of numbers whose Narayana representation ends in ‘1‘ (which we previously named ‘$lastbit1$‘).

```
# Evaluates whether indeed $col0 contains and only contains all of the
# numbers with Narayana representation ending in ...1
eval test1 "?msd_nara Ax (Ei $col0(i,x)) <=> $lastbit1(x)";
```

The statement is evaluated by seeing whether all numbers in each sequence, are also present in the other sequence. Walnut evaluates this statement to **TRUE**, verifying the theorem and linking these sequences together.