Linear Time Addition of Fibonacci Encodings

Maoyuan (Raymond) Song

CMU-CS-20-118

May 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Keywords: Theory, Algorithms, Compression, Fibonacci Codes, Zeckendorf’s Theorem

Abstract

Fibonacci Encoding is a binary coding theme with applications in cryptography and data transmission. However, fast addition of Fibonacci Encodings is non-trivial due to carrying being bi-directional. We present and prove correctness for an $O(n)$ algorithm that when given two Fibonacci encoded natural numbers of length n, returns a Fibonacci Encoding representing their sum, without decoding. The algorithm is implemented and tested against the naïve algorithm.

Acknowledgments

I would like to thank Dr. Carl Kingsford for advising me during my 5th year Master's program. Dr. Kingsford gave me countless invaluable advices not only on this thesis project, but also on conducting academic research in general. It is thanks to Dr. Kingsford that I can finish and polish this thesis, and refine myself to be a better researcher.

I would also like to thank Dr. Danny Sleator for accepting my invitation to my thesis committee, as well as offering advices and discussing potential methods to my thesis project. Dr. Frank Pfenning took time to implement and validate invariants and time guarantees proposed in this thesis, and I truly appreciate his insights and comments.

Ms. Veronica Peet, advisor to me in my freshman year in the Mellon College of Science, and a good friend to me ever since, supported me throughout this busy year mentally and emotionally. I am really grateful of her support and friendship.

Many other friends, families, and peers supported me throughout my academic career. I appreciate their company and understanding.

Finally I would like to pay appreciation to the Olympic Forest Park in Beijing. The most important breakthrough I had was made while hiking in the park, thanks to its quiet and serene environment.

Contents

Chapter 1

Introduction

Fibonacci Encoding is a binary coding scheme on positive natural numbers. The concept is based on Zeckendorf's Theorem [3] [8] [15], due to Belgian mathematician Edouard Zeckendorf, which states that every natural number can be represented uniquely as a sum of distinct, non-consecutive Fibonacci numbers. When encoded with 0-bits and 1-bits, Fibonacci encodings preserve the important property that there are no consecutive 1-bits in the representation.

This property is vital to Fibonacci Encoding's application as a variable length code in both cryptography and the field of coding and data transmission. By eliminating consecutive 1-bits, encoders can append a 1-bit to the most-significant side of every encoded integer to create an instance of two consecutive 1-bits, and the decoder can recognize this pattern to understand when to end decoding the current number and start decoding the next one.

Compared to other binary encodings, in particular base-2 encoding, Fibonacci Encoding harbors significant advantages and certain drawbacks. When applied to sequences of small integers with few but significantly larger outliers, Fibonacci Encoding can effectively accomodate these anomalies without padding every encoding to the bit-length of the largest integer. As a universal code, Fibonacci Encoding is also resistant to errors and erasures: a single error or erasure can only invalidate at most two transimssioned numbers, as the consecutive 1-bits as break marks stop the error from propagating infinitely. One drawback of Fibonacci Encoding is that the number of bits needed to encode an integer is greater than that of base-2 encoding. Luckily, n bits of Fibonacci Encoding can encode numbers up to approximately 1.618^n, which implies that it is only a constant factor longer than base-2 encoding. Another significant drawback is the lack of hardware-level optimization for Fibonacci Encoding, as compared to base-2 encoding. This can be resolved by optimizations at implementation level, and will not be the focus of this thesis.

Examples of real-world applications of Fibonacci Encodings include an approach to quantum key distribution [10], digital image scrambling [4], loseless compression via Burrows-Wheeler [2], image compression [11], and crosstalk avoidance in hardware design [9]. Specific applications of addition of Fibonacci Encodings remain mostly theoretical at the moment, as the naïve quadratic algorithm is efficient enough to operate on relatively short encodings, but the need for one is mentioned by Kautz in context of synchronization control [5]. It is reasonable to envision that as the amount of data increases, longer encodings would need to be transmitted, and addition on long encodings would give rise to more efficient algorithms on the subject.

Knuth in his proof of associativity of ‘circle multiplication’ of Fibonacci Encodings [7] presented a procedure for adding two Fibonacci Encodings, applying two available carry rules repeatedly, as far to the more significant side as possible, but does not provide guarantee for linearity. Although it is hard to find a witness for the nonlinearity of Knuth’s procedure, proving linearity is also challenging, as operations executed at each iteration can differ drastically based on the input sequence.

In this thesis, we present a two-part linear time algorithm that adds two Fibonacci Encodings together using only three unidirectional scans of the input bitvector and runtime storage twice the length of the input, with rooms for improvement. We also explicitly present a proof of linearity for both parts of the algorithm. Experimental results on short inputs does not guarantee advantage against naïve algorithms due to hardware level optimizations on base-2 additions, but acceleration is expected on longer inputs.

Chapter 2

Addition of Fibonacci Encodings

2.1 Definitions

We start by introducing the Fibonacci Encoding Addition problem.

- **Input:** Two Fibonacci Encodings representing natural numbers a and b respectively.
- **Output:** A Fibonacci Encoding representing natural number $a + b$.

As Fibonacci Encodings are only defined on natural numbers, we concern ourselves exclusively with additions of natural numbers.

We also make the following definitions explicit:

- A *Fibonacci Encoding* of a natural number N is a sequence of 0's and 1's corresponding to the unique decomposition of N into a set of distinct and non-adjacent Fibonacci numbers that sum up to N. A Fibonacci Encoding consists of only 0's and 1's with no consecutive 1's.
- A *Fibonacci Sum* with a value of some natural number N is a sequence of 0's and 1's that corresponds to a set of distinct Fibonacci numbers that sums up to N. This set of Fibonacci numbers can contain consecutive Fibonacci numbers, and thus is not necessarily unique for N.
- A *Sum of two Fibonacci Encodings*, which we abbreviate as *Bitwise Sum*, is a sequence of 0's, 1's, and possibly 2's, which is the bitwise sum of two Fibonacci Encodings. Decoding a Bitwise Sum using Fibonacci Base arithmetics will yield the sum of the numbers encoded by the two Fibonacci Encodings.

Throughout this thesis, we adopt the tradition that the leftmost bit is the least-significant bit. The number 17, decomposed as $1 + 3 + 13$, is thus encoded as `101001`. Fibonacci Encodings forbid the use of the 0th Fibonacci number, 1, to avoid confusion with the 1st Fibonacci number, which is also 1.

The following notation will be used to represent an arbitrary subsequence of a sequence of bits:

$$\cdots \overset{i}{0}\,\overset{i+1}{1}\,\overset{i+2}{0}\,\overset{i+3}{1}\,\overset{i+4}{0} \cdots$$

where the index of each of the bits are appended on top of the bit. We abbreviate this notation by only appending their relative indices with respect to one of the bits under our examination, usually the first bit in the subsequence:

$$\cdots \overset{0}{0}\overset{1}{1}\overset{2}{0}\overset{3}{1}\overset{4}{0} \cdots$$

2.2 A naïve Algorithm

A naïve algorithm involves decoding the input, calculating in binary, and re-encoding the result. This algorithm has an asymptotic complexity of $O(n^2)$, where n is the length of the input bitvector, which is not ideal. There exists prior works on fast encoding and decoding algorithms for Fibonacci Encodings, but only a constant factor of acceleration is achieved, while the asymptotic complexity of encoding and decoding remains identical [1] [14] [13].

One might inquire about using carries to eliminate possible 2-bits after bitwise addition, similar to the technique used in base-2 addition. The following two carry rules apply to Fibonacci-base addition:

$$\cdots 1\,1\,0 \cdots \quad \rightarrow \quad \cdots 0\,0\,1 \cdots$$
$$\cdots 0\,0\,2\,0 \cdots \quad \rightarrow \quad \cdots 1\,0\,0\,1 \cdots$$

However, notice that the second rule passes the carry not only to the immediate more significant bit as in base-2 addition, but also to the less significant side. This double carry complicates the carrying process, and prevents a simple iterative process from resolving all carries. The result of resolving a single 2-bit via carrying can cascade in both directions and result in multiple additional 2-bits, if not handled properly. Thus, more intricate methods to resolve carrying are needed. We show that there exists a method, that given two Fibonacci Encodings, adds them together in linear, e.g. $O(n)$ time, without decoding.

Chapter 3

A Two-Part Linear Algorithm

Observe that we can partition this problem into two separate and independent parts:

- **2-Elimination:** Given two Fibonacci Encodings of a and b as inputs, simplify the corresponding Bitwise Sum and return a Fibonacci Sum with value $c = a + b$.
- **Canonicalization:** Given a Fibonacci Sum of some value c, canonicalize it and return the unique Fibonacci Encoding of c.

We present linear-time algorithms for both 2-Elimination and Canonicalization.

3.1 Canonicalization

We start by presenting an algorithm for Canonicalization, which is the easier part of the two.

The algorithm takes in a Fibonacci Sum, and scans the bitvector twice, first from the most-significant bit (right) to the least-significant bit (left), and then the other way around. During both scans, whenever the algorithm encounters consecutive 1's, it carries them over to the bit to their immediate right.

Theorem 1 (Correctness of Canonicalization)**.** *Algorithm 1 correctly returns the Fibonacci Encoding of the value of the input Fibonacci Sum.*

Proof. We start by noting that by our logic of operation, every pair of consecutive 1-bits removed will result in the addition of another more-significant bit equal to the pair in value to be added. Thus, the value represented by the bitvector throughout the execution is invariant. It remains to show that the output is indeed a Fibonacci Encoding.

First of all, we examine the first pass of the scans (lines 3-10).

Lemma 2. *No 2-bits will be generated at any point throughout the first scan in this algorithm.*

Proof. By induction on the timestamp throughout the execution. Initially the input is guaranteed

Algorithm 1 Canonicalization

```
 1: function CANONICALIZE(FibSum as F)
 2:     $i \leftarrow len(\mathrm{F}) - 1$
 3:     while $i > 0$ do                                  ▷ Scan from MSB to LSB
 4:         if $\mathrm{F}[i] \geq 1$ and $\mathrm{F}[i-1] \geq 1$ then
 5:             $\mathrm{F}[i-1] \leftarrow \mathrm{F}[i-1] - 1$
 6:             $\mathrm{F}[i] \leftarrow \mathrm{F}[i] - 1$
 7:             $\mathrm{F}[i+1] \leftarrow \mathrm{F}[i+1] + 1$
 8:         end if
 9:         $i \leftarrow i - 1$
10:     end while
11:     while $i < len(\mathrm{F}) - 1$ do                    ▷ Scan from LSB to MSB
12:         if $\mathrm{F}[i] \geq 1$ and $\mathrm{F}[i+1] \geq 1$ then
13:             $\mathrm{F}[i] \leftarrow \mathrm{F}[i] - 1$
14:             $\mathrm{F}[i+1] \leftarrow \mathrm{F}[i+1] - 1$
15:             $\mathrm{F}[i+2] \leftarrow \mathrm{F}[i+2] + 1$
16:         end if
17:         $i \leftarrow i + 1$
18:     end while
19:     return F
20: end function
```

to not contain 2-bits. In the first pass, scanning from MSB to LSB, bits are only altered in lines 5-7. By I.H. we know that bits at indices $i-1$ and i are both 1-bits. AFSOC that the bit at index $i+1$ was initially a 1-bit before the increment, but then notice that at the previous timestamp, when we're examining the index $i+1$, we would have carried the 1-bits at indices i and $i+1$ and set them to two 0-bits, which is a contradiction. Thus, the bit at index $i+1$ is guaranteed to be a 0-bit, and thus will not be generated as a 2-bit. □

To prove the same results for the second pass of the scans, we would need additional properties of the state of the bitvector after the first pass as invariants.

Claim 1. *There are no runs of consecutive 1-bits with length longer than 2 after the first pass of Algorithm 1.*

Proof. Without loss of generality, AFSOC that there exists 3 consecutive 1-bits in the following configuration:

$$\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & \\ \cdots & 0 & 1 & 1 & 1 & 0 & \cdots \end{array}$$

with the leftmost 0-bit at relative index 0. Consider the two 1-bits at indices 2 and 3. By our construction we know that while scanning over index 3 in the first pass, index 2 must be a 0-bit, or else we will carry to index 4. The bit at index 2 must thus be the result of a carry that happened at indices 0 and 1, as in the following configuration.

$$\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & \\ \cdots & 1 & 2 & 0 & 1 & 0 & \cdots \end{array}$$

However, this implies that at one point there exists a 2-bit at either index 1 or some indices before it, which violates our invariant at Lemma 2 above and is thus a contradiction. □

Additionally,

Claim 2. *Two occurances of consecutive 1-bits must have an occurance of at least two consecutive 0-bits in between after the first pass of Algorithm 1.*

We will see that this property is vital to our proof of correctness.

Proof. Observe that any configuration that violates our assumption must be of the form:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 1\ 0\ 1\ \cdots\ 1\ 0\ 1\ 0\ 1\ \overset{n}{1}\ \cdots$$

with arbitrarily many 1-bits in between. Notice that when there are no in-between 1-bits, the formation will have four consecutive 1-bits, which by Claim 1 is impossible. Thus, without loss of generality, assume that there are at least 1 in-between 1-bits. Additionally let the leftmost bit be assigned index 0 and the rightmost bit be assigned index n.

By a logic similar to that in the proof of Claim 1, the 1-bit at index $n-1$ must be a result of a carry that happened to the left, as seen in the following configuration:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 1\ 0\ 1\ \cdots\ 1\ 0\ 2\ 1\ 0\ \overset{n}{1}\ \cdots$$

It seems like this is a violation of our invariant, but a possible scenario is that one of the 1-bits in the 2-bit at index $n-3$ is a result of a carry that happened to the left, and as the carry at indices $n-3$ and $n-2$ will happen before that, we can avoid generating the 2-bit as a byproduct. Thus, we can further backtrack and obtain an earlier formation:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 1\ 0\ 1\ \cdots\ 2\ 1\ 1\ 1\ 0\ \overset{n}{1}\ \cdots$$

We resolve the 2-bit at index $n-5$ in a recursive manner. We can do so infinitely until we encounter two consecutive 0-bits, and obtain the following formation:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 2\ 1\ 1\ 1\ 0\ \overset{n}{1}\ \cdots$$

which can be backtracked into:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ \overset{n}{1}\ \cdots$$

in which case no 2-bits are inferred to exist as byproducts anymore, and we're done. If we encounter the two consecutive 1-bits at the beginning instead, implying that no occurance of two consecutive 0-bits exists between two occurances of consecutive 1-bits, the following configuration will occur:

$$\cdots\ \overset{0}{1}\ 1\ 0\ 2\ 1\ 1\ \cdots\ 1\ 1\ 1\ 1\ 0\ \overset{n}{1}\ \cdots$$

$$\cdots\ \overset{0}{1}\ 2\ 1\ 1\ 1\ 1\ \cdots\ 1\ 1\ 1\ 1\ 0\ \overset{n}{1}\ \cdots$$

at which point the existense of the 2-bit can no longer be circumvented, and we have a contradiction to our invariant at Lemma 2. □

We now have sufficient information to set out to prove the correctness after the second pass of the scan (lines 11-18).

Lemma 3. *No 2-bits will be generated during both passes throughout the execution of this algorithm.*

Proof. We have shown that 2-bits will not be generated by the first pass. It remains to show that 2-bits will not be generated by the second pass.

In the second pass, scanning from LSB to MSB, bits are only altered in lines 13-15 if they form the following formation:

$$\cdots\ \overset{0}{\mathtt{1}}\,\overset{1}{\mathtt{1}}\,\overset{2}{\mathtt{0}}\,\overset{3}{\mathtt{X}}\,\overset{4}{\mathtt{X}}\ \cdots$$

where the `X`-bits are unknown bits and not necessarily identical, and the indices appended on top are the relative indices with respect to the first bit at absolute index i.

If indices 0 and 1 are both 1-bits, by Claim 1 we know that index 2 must be a 0-bit, thus we can carry the 1-bits at indices 0 and 1 to a new 1-bit at index 2 without directly creating 2-bits.

We also claim that this will not violate invariants Claim 1 and Claim 2.

Observe that setting index 2 to a 1-bit cannot violate Claim 1, or else we have two consecutive 1-bits at indices 3 and 4, and the configuration prior to our carrying will be:

$$\cdots\ \overset{0}{\mathtt{1}}\,\overset{1}{\mathtt{1}}\,\overset{2}{\mathtt{0}}\,\overset{3}{\mathtt{1}}\,\overset{4}{\mathtt{1}}\ \cdots$$

which violates Claim 2 that two occurances of consecutive 1-bits must have an occurance of consecutive 0-bits in between.

Also observe that setting index 2 can either create a new occurance of consecutive 1-bits, or destroy an old occurance of consecutive 0-bits, depending on the bit at index 3. However, setting index 2 will also create a new occurance of consecutive 0-bits to satisfy our invariants.

In the case where setting index 2 creates new consecutive 1-bits, we have the following formation before and after the carry:

$$\cdots\ \mathtt{X}\,\overset{0}{\mathtt{1}}\,\overset{1}{\mathtt{1}}\,\overset{2}{\mathtt{0}}\,\overset{3}{\mathtt{1}}\,\overset{4}{\mathtt{0}}\,\mathtt{X}\ \cdots$$

$$\cdots\ \mathtt{X}\,\overset{0}{\mathtt{0}}\,\overset{1}{\mathtt{0}}\,\overset{2}{\mathtt{1}}\,\overset{3}{\mathtt{1}}\,\overset{4}{\mathtt{0}}\,\mathtt{X}\ \cdots$$

By our invariant, we know that there exists consecutive 0-bits both to the left of index 0 and to the right of index 4. These occurances of consecutive 0-bits are not altered by our carry, so the newly created consecutive 1-bits at indices 2 and 3 are still separated from other occurances of consecutive 1-bits by consecutive 0-bits, thus satisfying Claim 2.

On the other hand, when we destroy an old occurance of consecutive 0-bits, we have the following formation before and after the carry:

$$\cdots\ \mathtt{X}\,\overset{0}{\mathtt{1}}\,\overset{1}{\mathtt{1}}\,\overset{2}{\mathtt{0}}\,\overset{3}{\mathtt{0}}\,\overset{4}{\mathtt{1}}\,\mathtt{X}\ \cdots$$

```
      0 1 2 3 4
··· X 0 0 1 0 1 X ···
```

In this case, no new occurances of consecutive 1-bits are created, so the two occurances of consecutive 1-bits to the left of index 0 and to the right of index 4 are now adjacent to each other. However, the newly created occurance of consecutive 0-bits at indices 0 and 1 are now separating them apart, also keeping Claim 2 invariant.

Thus, by setting index 2 to a 1-bit we also cannot violate Claim 2.

We can thus conclude that every operation in the second pass of the scan will not generate 2-bits, and thus the algorithm will not generate 2-bits at any point of its execution, including the end state. □

This thus concludes the proof that Algorithm 1 correctly outputs a Fibonacci Sum. It remains to show that there are no consecutive 1-bits, and thus the output is actually a Fibonacci Encoding.

Lemma 4. *There are no consecutive 1-bits in the bitvector outputted by Algorithm 1.*

Proof. AFSOC we have an occurance of consecutive 1-bits at indices i and $i+1$. We know that when we scan over index i, one of these two indices are 0-bits, or we would have carried them over to index $i+2$. As a result, one of the indices i and $i+1$ were set to a 1-bit while scanning over some index greater than i However, as our construction dictates, to set index i or $i+1$ to a 1-bit, we must be scanning either index $i-2$ or $i-1$, both less than i, which is a contradiction. □

We can thus conclude that the output contains only 0-bits and 1-bits with no consecutive 1-bits, and has the same value as the input. Thus, the output is the unique Fibonacci Encoding of the input value, and Algorithm 1 behaves correctly.

This thus concludes the proof of Theorem 1 □

It is trivial to see that Algorithm 1 runs in $O(n)$ time.

Notice that the two passes of Algorithm 1 must follow the given order. If we scan and carry from LSB to MSB first, it is very likely that we will encounter runs of consecutive 1-bits with length longer than 2, which will result in the generation of 2-bits. The first pass from MSB to LSB is meant to satisfy Claim 1 and Claim 2 to make sure that no 2-bits will be generated in the following pass from LSB to MSB.

Additionally, we can further simplify line 4 to 7 and 12 to 15 of Algorithm 1 by directly assigning values to bits instead of modifying them, in response to proven guarantees to their value during runtime execution.

3.2 2-Elimination

We next present an algorithm for 2-Elimination.

Algorithm 2 Simplified Canonicalization

```
1:  function SimpleCanonicalize(FibSum as F)
2:      $i \leftarrow len(\mathrm{F}) - 1$
3:      while $i > 0$ do                                  ▷ Scan from MSB to LSB
4:          if $\mathrm{F}[i] = 1$ & $\mathrm{F}[i-1] = 1$ then
5:              $\mathrm{F}[i-1] \leftarrow 0$
6:              $\mathrm{F}[i] \leftarrow 0$
7:              $\mathrm{F}[i+1] \leftarrow 1$
8:          end if
9:          $i \leftarrow i - 1$
10:     end while
11:     while $i < len(\mathrm{F}) - 1$ do                 ▷ Scan from LSB to MSB
12:         if $\mathrm{F}[i] = 1$ & $\mathrm{F}[i+1] = 1$ then
13:             $\mathrm{F}[i] \leftarrow 0$
14:             $\mathrm{F}[i+1] \leftarrow 0$
15:             $\mathrm{F}[i+2] \leftarrow 1$
16:         end if
17:         $i \leftarrow i + 1$
18:     end while
19:     return F
20: end function
```

Instead of two scans in opposite directions, the algorithm keeps certain invariants and uses only one scan from MSB to LSB, resolving every 2-bit it encounters locally, within the following window of length 5 around the 2-bit:

$$\cdots\ \overset{0}{\mathtt{X}}\,\overset{1}{\mathtt{X}}\,\overset{2}{\mathtt{0}}\,\overset{3}{\mathtt{2}}\,\overset{4}{\mathtt{0}}\ \cdots$$

where the `X`-bit at indices 0 and 1 are unknown. The bits at indices 2 and 4 are guaranteed by kept invariants to be 0-bits. Notice that by decomposing the 2-bit at index 3, we will increment index 4 by 1 which is a stable carry that does not create additional 2-bits, while the other increment to index 1 can be stablized together with the bit at index 0. More detailed descriptions of this procedure can be found in the proof of Lemma 7 and Lemma 8, as well as line 4 to 13 of the following pseudocode.

Algorithm 3 2-Elimination

```
function 2ELIM(BitwiseSum as B)
    $i \leftarrow len(\mathrm{B}) - 1$
    while $i > 0$ do                                          ▷ Scan from MSB to LSB
        if $\mathrm{B}[i] \geq 2$ then                                   ▷ Decompose a 2-bit
            $\mathrm{B}[i] \leftarrow \mathrm{B}[i] - 2$
            $\mathrm{B}[i+1] \leftarrow 1$
            $\mathrm{B}[i-2] \leftarrow \mathrm{B}[i-2] + 1$
            if $\mathrm{B}[i-2] \geq 1$ & $\mathrm{B}[i-3] \geq 1$ then      ▷ Stabilize the less significant carry
                $\mathrm{B}[i-3] \leftarrow \mathrm{B}[i-3] - 1$
                $\mathrm{B}[i-2] \leftarrow \mathrm{B}[i-2] - 1$
                $\mathrm{B}[i-1] \leftarrow 1$
            end if
        end if
        $i \leftarrow i - 1$
    end while
    return B
end function
```

Theorem 5 (Correctness of 2-Elimination). *Algorithm 3 correctly returns a Fibonacci Sum with the same value as the input Bitwise Sum.*

Proof. With similar reasoning as that in Theorem 1, we can observe that the value represented by the bitvector is invariant throughout the execution. It remains to show that the output is indeed a Fibonacci Sum.

We first observe an important property of a valid Bitwise Sum.

Lemma 6. *Indices adjacent to a 2-bit in a valid Bitwise Sum can only be 0-bits.*

This is evident from the fact that every 2-bit corresponds to two 1-bits at the same index in the two Fibonacci Encodings. As the properties of Fibonacci Encodings dictates it must be that the adjacent indices in both encodings are all 0-bits, and thus the adjacent indices in the Bitwise Sum must also be 0-bits.

Lemma 6 provides important guarantees to localize the potentially cascading effect of carrying in two directions. We would like to keep it as an invariant throughout Algorithm 3.

Lemma 7. *After every iteration of decomposing a 2-bit in Algorithm 3, indices adjacent to a 2-bit can only be 0-bits.*

Here, an iteration of decomposing a 2-bit is defined as an application of the carry rule on 2-bits, as well as actions taken to stablize the less significant carry, as implemented by line 4 to 13 of Algorithm 3.

We also keep the following invariant, which is vital to proving the correctness of Algorithm 3:

Lemma 8. *There are only 0-bits and 1-bits to the more-significant side of the index Algorithm 3 is currently scanning over.*

Confining the invariance of Lemma 7 within transition states between ‘steps’, defined as one iteration of the while loop on line 3, is necessary as there exists temporary violations of this invariant within each step that will be resolved by the end of the iteration.

Proof of Lemma 7 and Lemma 8. By induction on the number of iterations completed, if we prove that after each step our invariants hold given that they hold at the start of the step, we can conclude that Lemma 7 holds.

Each step starts with decomposing a 2-bit at index 0. The more-significant carry at index 1, implemented in line 6, is guaranteed to result in a 1-bit due to our invariants, which is what we want. The less-significant carry at index -2 is much more complicated, as we have no way to guarantee what bit index -2 originally houses. Luckily, we can use its adjacent bits to help stablize this carry. We case on the possible combinations of indices -3 and -2 before the decomposition of the 2-bit at index 0.

Case `00`**:**

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{2} & \mathtt{0} & \mathtt{X} & \cdots \end{array}$$

where the first `X`-bit corresponds to index -4, and the second `X`-bit corresponds to index 2. After decomposing, we have:

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{X} & \cdots \end{array}$$

The first `X`-bit at index -4 can potentially be a 2-bit, but our decomposition does not violate our invariants.

Case `10`**:**

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{2} & \mathtt{0} & \mathtt{X} & \cdots \end{array}$$

After decomposing, we have:

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{X} & \cdots \end{array}$$

The procedure on lines 8-12 of Algorithm 3 will further modify this to:

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{X} & \cdots \end{array}$$

which satisfies our invariants.

Case `20`**:**

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{2} & \mathtt{0} & \mathtt{0} & \mathtt{2} & \mathtt{0} & \mathtt{X} & \cdots \end{array}$$

After decomposing, we have:

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{2} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{X} & \cdots \end{array}$$

The procedure on lines 8-12 will modify this to:

$$\begin{array}{ccccccccc} & & {\scriptstyle -3} & {\scriptstyle -2} & {\scriptstyle -1} & {\scriptstyle 0} & {\scriptstyle 1} & & \\ \cdots & \mathtt{X} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{X} & \cdots \end{array}$$

Observe that the first `X`-bit at index -4 cannot be a 2-bit, as the bit adjacent to it at index -3 was initially a 2-bit as opposed to a 0-bit before decomposing the 2-bit at index 0. Thus, our invariants are kept.

Case `01`:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{1}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{2}}\ \overset{1}{\mathtt{0}}\ \mathtt{X}\ \cdots$$

After decomposing, we have:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{2}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{0}}\ \overset{1}{\mathtt{1}}\ \mathtt{X}\ \cdots$$

which will not be further modified.

Observe that although a new 2-bit has been created, it still satisfies our invariants as its adjacent indices are all 0-bits. Algorithm 3 will then proceed to decompose this new 2-bit two iterations later.

Case `11`:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{1}}\ \overset{-2}{\mathtt{1}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{2}}\ \overset{1}{\mathtt{0}}\ \mathtt{X}\ \cdots$$

After decomposing, we have:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{1}}\ \overset{-2}{\mathtt{2}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{0}}\ \overset{1}{\mathtt{1}}\ \mathtt{X}\ \cdots$$

The procedure on lines 8-12 will modify this to:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{1}}\ \overset{-1}{\mathtt{1}}\ \overset{0}{\mathtt{0}}\ \overset{1}{\mathtt{1}}\ \mathtt{X}\ \cdots$$

which satisfies our invariants.

Case `02`:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{2}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{2}}\ \overset{1}{\mathtt{0}}\ \mathtt{X}\ \cdots$$

After decomposing, we have:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{3}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{0}}\ \overset{1}{\mathtt{1}}\ \mathtt{X}\ \cdots$$

At this point we allow 3-bits to appear, and extend Lemma 7 to apply to 3-bits as well. Here, Algorithm 3 will simply finish the current iteration and move on without processing the new 3-bit.

Although this seems like a worrying event, observe that a 3-bit is essentially a 2-bit and a 1-bit overlapped together, with the following carrying rule applicable:

$$\cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{0}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{3}}\ \overset{1}{\mathtt{0}}\ \mathtt{X}\ \cdots \quad\rightarrow\quad \cdots\ \mathtt{X}\ \overset{-3}{\mathtt{0}}\ \overset{-2}{\mathtt{1}}\ \overset{-1}{\mathtt{0}}\ \overset{0}{\mathtt{1}}\ \overset{1}{\mathtt{1}}\ \mathtt{X}\ \cdots$$

where an 1-bit remains at index 0 after the decomposition.

In all the six cases above, index 0 becomes a 0-bit after the iteration. Thus, we can safely decompose 3-bits in an indentical method to decomposing 2-bits, only replacing the bit at index 0 with an 1-bit instead of a 0-bit, which is implemented in line 5 of Algorithm 3. This extra 1-bit will not be adjacent to any 2-bit or 3-bit, thus our invariants are kept.

As we have shown our invariants to be kept for every possible combination of indices -3 and -2, we can conclude by induction that Lemma 7 and Lemma 8 hold. □

Given that Lemma 8 holds, we can conclude that at the end of the while loop, the entire bitvector will be composed of 0-bits and 1-bits. As the value our bitvector represents are invariant, the output is indeed a Fibonacci Sum with the same value as the input Bitwise Sum, and Algorithm 3 behaves correctly.

This thus concludes the proof of Theorem 5. □

It is also trivial to see that Algorithm 3 runs in $O(n)$ time.

It is worth noting that the rule for 2-decompositions change when decomposing 2-bits at indices 0 and 1, but it is trivial to case them prove correctness independently.

3.3 Conclusion

As correctness is proven, the concatenation of Algorithm 3 and Algorithm 2 is a linear-time algorithm for the Fibonacci Encoding Addition problem. Given any pair of valid Fibonacci Encodings of natural numbers a and b, we trivially run bitwise addition to them to acquire a Bitwise Sum with value $a + b$, feed it into Algorithm 3 and obtain a Fibonacci Sum with value $a + b$, and finally use Algorithm 2 to canonicalize it and return the Fibonacci Encoding of $a + b$.

Chapter 4

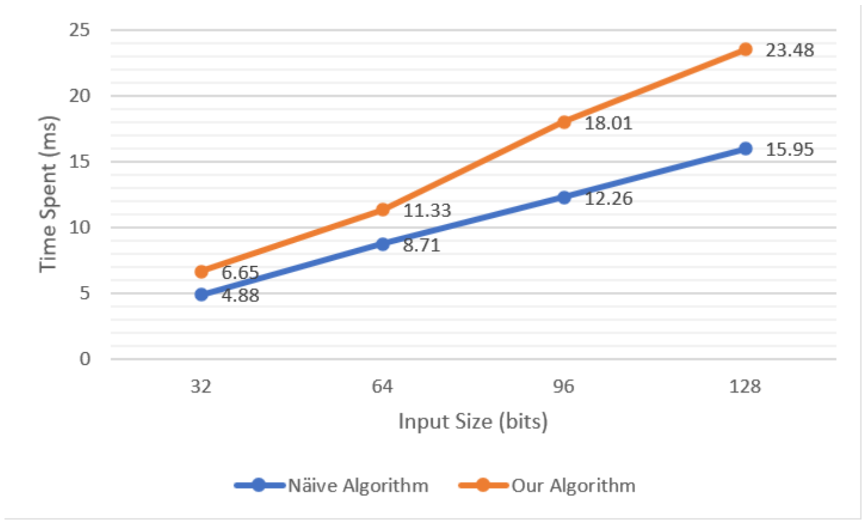
Implementation

A version of this algorithm implemented with bitvectors and bitwise operators in C++ was devised. This implementation, along with an initial realization of this algorithm in Python, can be found at `https://github.com/maoyuans/Linear-Fibonacci-Addition`.

This implementation relies on the `unsigned __int128` type supported by GCC, an 128-bit container capable of supporting bitwise operations. No libraries were used in the implementation.

To store 2-bits and 3-bits in a binary environment, during the 2-elimination part of the algorithm, we keep two 128-bit containers to simulate the first and second bit of a 2-bit long container, capable of storing 0 to 3 bits.

This implementation is tested on 10000 iterations of adding random 128-bit Fibonacci Encodings, against the naïve quadratic algorithm, using greedy encoding and decoding methods and built-in base-2 addition. The time results, which is plotted below, confirmed that our implementation is indeed linear. The results also show that the theoretically quadratic naïve implementation is more efficient than our implementation up to a constant factor, and it also follows an approximately linear runtime with respect to input size.



A possible explanation for this is that, due to the Word-RAM model widely applied to computer architecture currently, addition on two n-bit integers takes $O(1)$ time for small n like 64 or 128. As a result, each iteration of encoding and decoding only requires $O(1)$ runtime as opposed to $O(n)$, which brings the total time complexity from quadratic to linear.

Theoretically, as the size of the input increases beyond 128 bits, the time required to finish the naïve algorithm will start growing quadratically, and should outgrow the runtime of our algorithm soon. Doing so would require implementing the algorithm with containers larger than 128 bits, or multiple containers chained together, with modifications to the algorithm to handle such cases, which is a potential extension to this project.

Chapter 5

Conclusion and Future Work

We have presented an linear-time algorithm for adding Fibonacci Encodings. We have also shown with proof their correctness and time complexity bounds. Although the experimental results does not show supremacy over the naïve algorithm, it does convey growing advantages with respect to increases in data size, as well as potential for improvements.

Conceptually, it is natural to ask next if a similar linear-time algorithm exists for subtraction on Fibonacci Encodings. The basics of substraction on Fibonacci Encoding involve decomposing more significant bits to negate '-1'-bits, or indices where an 1-bit from the minuend is subtracted from a 0-bit in the subtrahend:

$$\begin{aligned} \cdots\ \texttt{0 0 0 0 0 0 1}\ \cdots &\rightarrow \cdots\ \texttt{0 0 0 0 1 1 0}\ \cdots \\ &\rightarrow \cdots\ \texttt{0 0 1 1 0 1 0}\ \cdots \\ &\rightarrow \cdots\ \texttt{1 1 0 1 0 1 0}\ \cdots \\ &\cdots \end{aligned}$$

Potential problems that arise from these rules is that if a '-1'-bit is located an odd number of bits towards the less significant side of the closest 1-bit, the continuous decomposition will pass through and influence the bit immediately to the less significnat side of the '-1'-bit, which might result in the generation of 2-bits.

A potential idea on this topic is done via shifting bits: The properties of Fibonacci numbers dictate that given a Fibonacci Encoding x, we can express it as the sum of two Fibonacci Sums $x \gg 1 + x \gg 2$, subject to minor modifications on a few indices near 0. Thus, given these two Fibonacci Sums, for any '-1'-bit generated from bitwise subtraction, one of the two Fibonacci Sums must contain an 1-bit of even distance to it. However, after subtracting an 1-bit from one of the Fibonacci Sums, it is possible that there is some 1-bit in the minuend such that the immediately adjacent 1-bits in both Fibonacci Sums are of odd distance to it, thus invalidating the procedure. More analysis thus is needed to polish and expand on this idea.

Another class of encodings similar to Fibonacci Encodings is NegaFibonacci Encodings, which uses negaFibonacci numbers [6] [12] as a base to encode all non-zero integers. Arithmetics on NegaFibonacci Encodings can be investigated and applied as extensions to our algorithm.

On the subject of optimization, there exists a lot of room for improvement. As mentioned above, modifications to the algorithm and data structures can be made to allow storage and bitwise operations on inputs of size greater than 128 bits. Additionally, the application of bitwise operators can be optimized to reduce the amount of operations needed to further improve runtime on a constant factor level.

Algorithmically, the 2-elimination step can potentially be modified to combine the bitwise sum step, as well as the first pass of the canonicalization step, to reduce the number of passes needed and eliminate the need to allocate another container to store 2-bits and 3-bits. Our algorithm is inherently sequential, but possible extensions to allow parallel executions should be discussed as well.

Bibliography

[1] Radim Baca, Václav Snásel, Jan Platos, Michal Krátký, and Eyas El-Qawasmeh. The fast Fibonacci decompression algorithm. *CoRR*, abs/0712.0811, 2007. 2.2

[2] R. Bastys. Fibonacci coding within the Burrows-Wheeler compression scheme. *Elektronika ir Elektrotechnika*, pages 28–32, 01 2010. 1

[3] JL Brown Jr. Zeckendorf's theorem and some appucations. 1964. 1

[4] Jiancheng Zou, R. K. Ward, and Dongxu Qi. A new digital image scrambling method based on Fibonacci numbers. In *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, volume 3, pages III–965, 2004. 1

[5] William H. Kautz. Fibonacci codes for synchronization control. *IEEE Transactions on Information Theory*, 11(2):284–292, Apr 1965. 1

[6] Donald Knuth. NegaFibonacci numbers and the hyperbolic plane. In *San Jose-Meeting of the Mathematical Association of America*, 2008. 5

[7] Donald E. Knuth. Fibonacci multiplication. *Applied Mathematics Letters*, 1(1):57–60, 1988. 1

[8] Murat Kologlu, Gene Kopp, Steven J. Miller, and Yinghui Wang. On the number of summands in Zeckendorf decompositions. 2010. 1

[9] Madhu Mutyam. Fibonacci codes for crosstalk avoidance. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(10):1899–1903, 2012. 1

[10] David S. Simon, Nate Lawrence, Jacob Trevino, Luca Dal Negro, and Alexander V. Sergienko. High-capacity quantum Fibonacci coding for key distribution. *Phys. Rev. A*, 87:032312, Mar 2013. 1

[11] K Somasundaram and P Sumitra. Compression of image using Fibonacci Fode (FC) in jpeg2000. *International Journal of Engineering Science and Technology*, 2(12):7311–7319, 2010. 1

[12] Amelia Carolina Sparavigna. ON THE GROUP OF THE FIBONACCI NUMBERS. May 2018. working paper or preprint. 5

[13] Jiří Walder, Michal Krátký, Radim Bača, Jan Platoš, and Václav Snášel. Fast decoding algorithms for variable-lengths codes. *Information Sciences*, 183(1):66 – 91, 2012. 2.2

[14] Jiri Walder, Michal Krátkỳ, and Jan Platos. Fast Fibonacci encoding algorithm. In *DATESO*, pages 72–83. Citeseer, 2010. 2.2

[15] E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de Fibonacci ou denombres de Lucas. *Bull. Soc. Roy. Sci. Liége*, 41:179–182, 1972. 1