BINARY FIBONACCI ARITHMETIC

SARAH QUINN, ANDREW L. TRATTNER, AND YUWEI ZHANG

Abstract. This paper introduces a binary Fibonacci representation for positive integers. We define maximal representation and show a unique correspondence between integers and their maximal representations in binary Fibonacci. We introduce linear time algorithms for converting numbers to their maximal representations as well as adding and subtracting in binary Fibonacci. A quadratic time algorithm is described for multiplication. We conclude with brief remarks on promising sequences for further research.

1. Introduction

This paper introduces a methodology for representing and performing arithmetic operations on positive integers using the Fibonacci sequence as a number base. Although it may seem to be, this problem is not a purely theoretical endeavor. The winning strategy for the game of Nim has been shown to rely on the Fibonacci numbers [1], so efficient computing with them is desirable. In their paper on estimating Jones polynomials, Peter Shor and Stephen Jordan used Fibonacci in their representation of closed braids to show that approximating the Jones polynomial of the trace closure of a braid is one-clean-qubit complete [2]. Before publishing the paper, the authors were looking for an addition algorithm—like the one presented herein—for a step of their proof. While Shor and Jordan found a different method to complete their proof, computation in binary Fibonacci is still relevant.

In this paper, we show that all positive integers have representations in the binary Fibonacci scheme. We provide linear time algorithms for adding and subtracting in base Fibonacci. We also provide a quadratic time multiplication algorithm. The time needed for these procedures is strongly polynomial in the size of the representation.

In what follows, we explain the concept of the maximal binary Fibonacci representation for a number and then derive an algorithm to convert any binary Fibonacci representation to maximal form. We then proceed to algorithms for performing addition, subtraction, and multiplication. The paper concludes with suggestions for future research.

2. Maximal Representation

We can represent a positive integer x in base Fibonacci by representing x as the sum of a subset of Fibonacci numbers.

$$x = F_{k_0} + F_{k_1} + ... + F_{k_n}$$

where $j \in \mathbb{N}$ and F_{k_j} is a Fibonacci number and $F_{k_j} > F_{k_{j-1}} \forall j$. To represent this sum in binary, we denote each F_k in the sum with a 1 in the kth digit. We represent

Date: October 10, 2016.

all unincluded Fibonacci numbers less than the greatest F_k included with a 0 in their respective digits. Hence, we can represent 4, which is the sum of Fibonacci numbers 1 and 3 as 10100 (including $F_0 = 0$ and $F_1 = 1$ in the sequence). Because the first two members of the Fibonacci sequence are unnecessary for the computations in this paper, we represent 4 as 101 instead.

It is worth noting that many numbers have multiple base Fibonacci representations since they can be expressed as multiple different sums of Fibonacci numbers. For instance, $4 = 3 + 1 = 2 + 1 + 1$. In this paper, we are only interested in the maximal representation, which we define as the representation with the largest binary value. This is also known as the Zeckendorf representation.

Let us understand the maximal representation further through an exploration of some of its properties.

Lemma 2.1. *The maximal representation does not contain consecutive ones and a representation with no consecutive ones is necessarily maximal.*

Proof. To rephrase this lemma in terms of the Fibonacci sequence, a maximal representation of a number never contains two consecutive Fibonacci numbers. This is because the Fibonacci sequence is defined as

$$a_n = a_{n-1} + a_{n-2}$$

Thus, two consecutive ones can always be substituted by the next largest member of the sequence, resulting in a larger binary value representation. Section 3 expounds on how to handle converting cases with two or more consecutive ones to the maximal representation. The conversion procedure demonstrates that only consecutive ones in a given representation will require a conversion. Then if no consecutive ones exist, no conversion occurs because that representation must already be maximal. □

Let that F_n be the nth member of the Fibonacci sequence and that each a_i is the indicator variable (taking a value of 0 or 1) for the ith member of the Fibonacci sequence in the sum representing x. Recall that we do not use a_1 or a_0 in this paper.

Lemma 2.2. *$F_n > x$, where x is a number with maximal binary base Fibonacci representation $a_{n-1}a_{n-2}...a_2$.*

Proof. Since $a_{n-1}a_{n-2}...a_2$ is a Fibonacci-based number in maximal representation, from Lemma 2.1, we do not have consecutive 1's in the sequence. Thus, $a_{n-1} = 1$, $a_{n-2} = 0$, and

$$a_{n-1}a_{n-2}...a_1 < F_{n-1} + F_{n-2} = F_n.$$

□

We follow these claims with a proof that a maximal representation does indeed exist for every positive integer.

Theorem 2.3. *$\forall x \in \mathbb{Z}^+$, a maximal binary Fibonacci base representation exists.*

Proof. We can prove existence using strong induction.

We begin with the base cases. We can represent 1 as 1, 2 as 10, and 3 as 100 since they are all members of the Fibonacci sequence. We omit, as stated above, the F_0 and F_1 from the representations because they are not necessary.

We also use 4 as a base case since, unlike the previous cases, it is not a Fibonacci number. We represent 4 maximally as 101 $(1 + 3)$.

Having established the base cases, we proceed to the inductive step. We assume by strong induction that the theorem holds for all positive integers less than x. We now have to prove that under these circumstances, the maximal representation of x will always exist. There are two cases here:

1. x is a Fibonacci number—in this event, we are done. We can represent $x = F_k$ as 10...0, where there are $k-2$ zeros (if we kept F_0 and F_1, there would be k zeros). This is a maximal representation.

2. x is not a Fibonacci number—in this case, we know that x must be between two Fibonacci numbers such that $F_k > x > F_{k-1}$ for some $k \in \mathbb{N}$. Call $x' = x - F_{k-1}$. We know that x' must be less than F_{k-2} because otherwise, by 2.1, we would be able to represent x with F_k included in the sum and x would be greater than or equal to F_k.

As a result, we can represent the maximal representation of x as the maximal representation of x' with an additional one in the F_{k-1} position. We know that x' doesn't already have a one in this position because x' is less than F_{k-2}. Furthermore, we can conclude that x' also doesn't have a one in the F_{k-2} position, which would result in consecutive ones and make the representation non-maximal. From this procedure, we find the maximal representation for x. □

Theorem 2.4. *No positive integer has two different maximal representations.*

Proof. For the sake of contradiction, consider two different maximal representations of a positive integer x, such that A is one set of nonconsecutive Fibonacci numbers that sum to x and B is the other, distinct set of nonconsecutive Fibonacci numbers that sum to x. Now set $A' = A \backslash B$ and $B' = B \backslash A$ so that $A' \cap B' = 0$. We know that A' and B' must have the same sum because only $A \cap B$ was removed from both A and B, which have the same sum.

Let F_A be the largest element of A' and F_B be the largest element of B'. We know that $F_A \neq F_B$ because we subtracted out all overlapping elements of A and B to get A' and B'. Assume that $F_A < F_B$ without loss of generality. We know that the sum of the elements in A' must be less than F_{A+1} because, if not, A' would not be a maximal representation. Therefore, the sum of the elements in A' must be less than F_B and, thus, less than the sum of the elements in B'. The only way this is possible is if both A' and B' are empty but this is a contradiction since A and B must then be the same set. Thus, a number only has one maximal representation. □

Lemma 2.5. *The maximal representation of n uses $O(\log n)$ binary digits.*

Proof. This lemma follows from the exponential growth rate of the Fibonacci sequence, $\phi = \frac{1+\sqrt{5}}{2}$. □

3. Conversion To Maximal Representation

3.1. **Overview and Preliminaries.** We notice that a number can be written in different ways in binary Fibonacci representation. For example, the number 8 can

have several Fibonacci representations:

$$\begin{aligned} 8 &= 1011 && F_4 + F_2 + F_1 = 5 + 2 + 1 = 8 \\ &= 1100 && F_4 + F_3 = 5 + 3 = 8 \\ &= 10000 && F_5 = 8 \end{aligned}$$

We would like to consider the algorithm for converting any number in binary Fibonacci representation to the maximal representation, as this will prove to be useful as a subroutine for arithmetic operations such as addition and subtraction. Given any number in binary Fibonacci representation of length n, we are able to convert it to the maximal representation in linear time, $O(n)$.

First, one immediate thought would be to combine consecutive ones - replace 11 with 100. In the above example, we can replace 1100 with 10000. But what if combining consecutive ones creates more consecutive ones to the left, when we replace 1011 with 1100 for example? In fact, our algorithm is able to do it systematically, and we will prove that this will not arise. We convert a number in the form of $1(01)^*1$ to $10(00)^*0$, where the asterisk indicates zero or more occurrences of the sequence in the brackets. In other words,

$$1\underbrace{01\cdots01}_{n\ 01\text{'s}}1 \to 10\underbrace{00\cdots00}_{n\ 00\text{'s}}0, \quad \text{where } n \geq 0.$$

We prove this in the following lemma:

Lemma 3.1. $F_{n+1} = (F_n + F_{n-2} + \cdots + F_{j+2} + F_j) + F_{j-1}$

Proof. We repeatedly use the definition $F_{n+1} = F_n + F_{n-1}$.

$$\begin{aligned} & (F_n + F_{n-2} + \cdots + F_{j+2} + F_j) + F_{j-1} \\ &= F_n + F_{n-2} + \cdots + F_{j+2} + (F_j + F_{j-1}) \\ &= F_n + F_{n-2} + \cdots + (F_{j+2} + F_{j+1}) \\ &= \cdots \\ &= F_n + F_{n-1} \\ &= F_{n+1} \end{aligned}$$

□

3.2. **Procedure and Analysis.** Now we are ready to proceed to the conversion algorithm:

3.2.1. *Procedure.* We scan the number from left to right. Let x denote the sequence of numbers to the right of the digits we are processing at each step. When we see the first $a_i = 1$, we have several cases:

(1) $a_{i-1} = 1$.
$11x \to 100x$. If $a_{i-1} = 1$, we set $a_{i+1} = 1$, $a_i = a_{i-1} = 0$, and continue with x.

(2) $a_{i-1} = 0$
  (a) $a_{i-2} = 0$.
  $100x \to 100x$. If $a_{i-1} = a_{i-2} = 0$, we simply continue with x.
  (b) $a_{i-2} = 1$.
  If $a_{i-1} = 0$, $a_{i-2} = 1$, we continue to scan to the right until we first see a break in 01's, i.e. we hit 00 or 1 following $1(01)^*$. (If we reach the

end without seeing a break in 01's, i.e. we have $1(01)^*$, we are done.) $1(01)^*00x \to 1(01)^*00x$. If we hit 00, we simply move on to the right. $1(01)^*1x \to 10(00)^*0x$. If we hit 1, then we set $a_{i+1} = 1$, $a_i = a_{i-1} = ... = 0$ as proven in Lemma 3.1, and continue with x.

3.2.2. *Correctness.* We can see that after each step in each of the cases, we always set the last two processed digits to 00. Thus, in the next step, we are guaranteed that we can set the preceding digit to 1 without creating consecutive ones to the left - we would not affect the previous sequence of digits.

3.2.3. *Runtime.* It follows from our procedure that we visit each digit at most twice - we scan the digits from left to right first, and then when needed, we set the digits accordingly. Hence, when we reach the end of the number, we have converted it into maximal form in $O(n)$ time.

3.3. **Pseudocode and Example.** To summarize, Case (1) can be incorporated into Case (2)(b), where we have no run of 01's, just 1 immediately followed by 1. We see that only need to change the digits in the case of (2)(b), where $1(01)^*1 \to 10(00)^*0$, and do nothing for other cases. We then simply move on to the right. We present this in the following pseudocode:

Algorithm 1 ToMaxRep(A, n)

Input: A number in binary Fibonacci representation, $A = a_n a_{n-1} \cdots a_1$.
Output: ToMaxRep(A, n) converts A to its maximal representation in place.

```
1:  procedure ToMaxRep(A, i)
2:      if i = 1 then
3:          EXIT                          ▷ We have reached the last digit - we are done!
4:      if a_i = 0 then
5:          ToMaxRep(A, i − 1)
6:      else                                                    ▷ Find the first 1
7:          j ← i − 1
8:          while a_j, a_{j−1} = 0, 1 do                        ▷ Find the run of 01's
9:              j ← j − 2
10:         if a_j = 1 then
11:             a_{i+1}, a_i, ··· , a_j ← 1, 0, ..., 0          ▷ 1(01)*1 → 10(00)*0
12:         ToMaxRep(A, j − 1)
```

Here is a worked example for the conversion to maximal representation as processed step by step by the algorithm. Digits in bold are already processed, and underlined digits are the ones we are looking at at each stage.

$$\underline{11}00101011$$
$$\Downarrow$$
$$\mathbf{100}\underline{00}101011$$
$$\Downarrow$$
$$\mathbf{10000}\underline{101011}$$
$$\Downarrow$$
$$\mathbf{10001000000}$$

We check that indeed

$$1100101011 = F_{10} + F_9 + F_6 + F_4 + F_2 + F_1 = 89 + 55 + 13 + 5 + 2 + 1 = 165,$$
$$10001000000 = F_{11} + F_7 = 144 + 21 = 165.$$

4. Addition

4.1. **Overview and Preliminaries.** When given two numbers in binary Fibonacci representation, we can often compute the sum via simple superposition, such as $10001 + 00100 = 10101$. In this section, we consider how to perform addition in linear time for more complicated cases such as $10100 + 00101 = 10201$, which we will see reduces to 100010.

4.2. **Procedure and Analysis.** Our procedure adds the given numbers by showing how to reduce twos (as above) and threes to ones and zeroes in a single pass over the digits, which are shown to never be four or greater.

4.2.1. *Procedure.* We start with two nonnegative integers given in binary Fibonacci representation. Convert both to maximal form so $x = a_n a_{n-1} \dots a_1$, $y = b_n b_{n-1} \dots b_1$, where we pad the smaller number with zeros such that both are of length n, without loss of generality. We are now ready to find the sum by adding x to y, so we will be left with x and $x + y$. Denote the digits of this result b'_i.

First instantiate an additional bit $b_{n+1} = 0$ and insert it into y's representation, since $x + y$ is represented by at most $n + 1$ bits. This follows from Lemma 2.2 and the fact that $F_{i+2} > F_i + F_i$.

We will find it useful to observe that we originally have no more than two ones in any window of two summed digits $a_i + b_i$ and $a_{i-1} + b_{i-1}$ because Lemma 2.1 guarantees no consecutive ones in the maximal representation.

Now begin scanning right on x and y, starting at digit n. At digit i, we may encounter the easy $a_i = b_i = 0$ where we let $b_i = 0$ and continue by simple superposition. If either a_i or b_i is nonzero, we consider a few cases, listed by priority:

(1) two consecutive nonzero digits ($a_i + b_i$ and $a_{i-1} + b_{i-1}$ are nonzero): subtract one from each and add one to b_{i+1}. This operation maintains the result by case (1) of section 3. We now have $b'_{i+1} = 1$ since other cases will show that b_{i+1} must have previously been zero and $b_i \leq 1$. We continue scanning.
(2) $a_i = 1$ and $b_i = 0$ or $a_i = 0$ and $b_i = 1$: add by simple superposition so $b'_i = 1$. Since we did not have two consecutive nonzero digits and we did not modify any digits to the right of i (less than i), we know that $b'_{i-1} = 0$ and $a_{i-2} + b_{i-2}$ is at most 2. If it is 2, then we know $b'_{i-3} = 0$ by Lemma 2.2.

(3) $a_i + b_i = 2$: we observe that $F_i + F_i = F_i + F_{i-1} + F_{i-2} = F_{i+1} + F_{i-2}$ by definition of the Fibonacci numbers, so we add one to b_{i+1} and b_{i-2}, setting $b_i = 0$. b_{i+1} must have previously been zero, else we would have had an earlier case. The same goes for b_{i-1}.
(4) $a_i + b_i = 3$: we observe that $F_i + (F_i + F_i) = F_i + (F_{i+1} + F_{i-2}) = F_{i+2} + F_{i-2}$ by definition of the Fibonacci numbers, so we add one to b_{i+2} and b_{i-2}, setting $b_i = 0$. This case only occurs as a result of $a_{i+2} + b_{i+2} \geq 2$, so we know that $b_{i+2} = 0$ and adding one will not be an issue. We also know that $a_{i-1} + b_{i-1} = 0$ by Lemma 2.2.

To conclude the procedure, convert the new string to maximal form and return. Note that modifications to b_0 are transferred to b_1 since we omit F_1 from our representation. Modifications to indices less than zero are simply discarded.

4.2.2. *Correctness.* We have shown how the addition algorithm reduces registers which contain two or three. We may not have any more than three in a given digit place since multiple ones would need to be added, but carry operations when $a_i + b_i \geq 2$ clearly may not occur in adjacent digit places, nor when the previous digits indexed at $i + 1 or i + 2$, respectively, are nonzero.

4.2.3. *Runtime.* Because we are guaranteed to modify previous digits only when doing so ensures they remain one or less, we do not return left in the scan. Since we proceed one digit to the right after performing a few operations at each index, the asymptotic running time is $O(n)$.

4.3. **Pseudocode and Example.** We give the following example to demonstrate the algorithm in practice: $100101 + 1010101$. Underlined digit(s) are the ones we are considering at each stage.

$$
\begin{aligned}
x &= \underline{1}010101 \\
y &= \underline{1}010010 \\
&\Downarrow \text{ case (3)} \\
&10\underline{1}0101 \\
&100\underline{2}0010 \\
&\Downarrow \text{ case (4)} \\
&1010\underline{10}1 \\
&11000\underline{11}0 \\
&\Downarrow \text{ case (1)} \\
&1010\underline{101} \\
&11001\underline{000} \\
&\Downarrow \text{ case (2)} \\
&1010101 \\
&11001101 \\
&\textsc{ToMaxRep} \implies 100010001
\end{aligned}
$$

Algorithm 2 ADD(x, y)

Input: Binary Fibonacci representations $x = a_n a_{n-1} \cdots a_1$ and $y = b_n b_{n-1} \ldots b_1$.
Output: ADD(x, y) returns x, $x + y$ in maximal representation.

```
1: procedure ADD(x, y):
2:     x, y ⟵ ToMaxRep(x, n), ToMaxRep(y, n + 1)
3:     i = n
4:     while i ≥ 1 do
5:         if x[i] == 0 ∩ y[i] == 0 then
6:             i− = 1, continue
7:         else
8:             y[i] = x[i] + y[i]
9:             if y[i] > 0 ∩ x[i − 1] + y[i − 1] > 0 then
10:                y[i − 1] = x[i − 1] + y[i − 1] − 1
11:                y[i]− = 1
12:                y[i + 1]+ = 1
13:                i− = 1, continue
14:            if y[i] == 1 then
15:                i− = 1, continue
16:            if y[i] == 2 then
17:                y[i]− = 2
18:                y[i + 1]+ = 1
19:                y[i − 2]+ = 1
20:                i− = 2, continue
21:            if y[i] == 3 then
22:                y[i]− = 3
23:                y[i + 2]+ = 1
24:                y[i − 2]+ = 1
25:                i− = 2, continue
26:    return ToMaxRep(x, n), ToMaxRep(y, n + 1)
```

4.4. **Conclusion.** We have shown $O(n)$ running time to sum two integers.

5. Subtraction

5.1. **Overview and Preliminaries.** Now that we have seen addition for numbers in binary Fibonacci representation, a natural step would be to consider subtraction. Given that we can do the addition of positive integers, it is not immediately obvious how we can subtract 1 from 10000, for example. Given two numbers in binary Fibonacci representation A and B, we present an algorithm that is also able to compute $A - B$ in linear time.

Before we begin, we simplify the computation by preprocessing A and B as follows: first, we convert them into the maximal form. This takes linear time in the length of the input as given in Section 3. Next, without loss of generality assume that $A > B$ (for $A < B$, swap A and B and simply add the negative sign to the final result). If A and B are of the same length, we scan the digits of A and B from

left to right, cancel out where both A and B have the same digit, and stop at the first position where A and B have different digits.

We have now come to the first differing digit, where it is 1 for A and 0 for B (If A has more digits than B, it will just be the first digit of A). We have $A = a_n a_{n-1} \cdots a_1$ and $B = b_m b_{m-1} \cdots b_1$, where $n > m$.

Now, we rewrite $A - B = (A - F_n) + (F_n - B)$. Note that F_n is a Fibonacci number greater than B as $n > m$, and $F_n > B$ by Lemma 2.2. We perform the computation in three steps:

1. $C = A - F_n$
2. $D = F_n - B$
3. result $= C + D = (A - F_n) + (F_n - B) = A - B$

5.2. **Procedure and Analysis.**

5.2.1. *Procedure.* We first compute $C = A - F_n$. Since $A = a_n a_{n-1} \cdots a_1$, C is simply $C = A - F_n = a_{n-1} \cdots a_1$.

Next, we compute $D = F_n - B$ using recursion. At each stage, we compute $F_j - B'$, where $B' = b_i b_{i-1} \cdots b_1$ in several cases:

(1) If $j = i+1$, since $F_{i+1} = F_i + F_{i-1}$, we have $F_j - F_i = F_{i+1} - F_i = F_{i-1}$. As B is in maximal form, we have $b_i = 1$ and $b_{i-1} = 0$. Write $d_{i+1} = d_i = 0$, and continue the procedure to compute $F_{i-1} - b_{i-2} \cdots b_1$. Note that if $B' = 1$ in this case, then we have $10 - 1 = 1$ $(3 - 2 = 1)$, so we write $d_1 = 1$ and we are done.
(2) If $j = i + 2$, since $F_{i+2} = F_{i+1} + F_i$, we have $F_j - F_i = F_{i+2} - F_i = F_{i+1}$. Write $d_{i+2} = d_{i+1} = d_i = 0$, and continue the procedure to compute $F_{i+1} - b_{i-1} \cdots b_1$.
(3) If $j > i + 2$, since $F_j = F_{j-1} + F_{j-2}$, write $d_{j-1} = 1$. We continue the procedure to compute $F_{j-2} - b_i ... b_1$.
(4) If $B' = 0$, we simply write $d_j = 1$ and we are done.

Finally, our last step would be to add C and D using the addition algorithm in Section 4, and after converting the result to the maximal representation, we obtain $A - B$.

5.2.2. *Correctness.* We see that at each stage, we maintain the fact that F_j has more digits than B', i.e. $j > i$, and thus by Lemma 2.2, $F_j > B'$. Hence, we are able to continue the recursion with the sequence to the right.

5.2.3. *Runtime.* We compute $C = A - F_n$ in $O(1)$. We compute $D = F_n - B$ in $O(n)$, as we go through the digits from left to right once. Finally, we compute the result $= C + D$ in $O(n)$ as well. Overall, the subtraction procedure runs in $O(1) + O(n) + O(n) = O(n)$ time.

5.3. **Pseudocode and Example.** We present the subtraction algorithm in the following pseudocode:

Algorithm 3 SUBTRACT(F_n, B)

Input: F_n and $B = b_m b_{m-1} \cdots b_1$, where $n > m$.
Output: $D = F_n - B$ from SUBTRACT(F_n, B)

```
1:  D = 0
2:  procedure SUBTRACT(F_j, B')
3:      if B' = 0 then
4:          d_j = 1
5:          EXIT                                          ▷ We are done!
6:      if j = i + 1 then
7:          if B' = 1 then
8:              d_1 = 1
9:              EXIT                                      ▷ 10 − 1 = 1, and we are done!
10:         else
11:             d_{i+1} = d_i = 0
12:             SUBTRACT(F_{i−1}, b_{i−2} ··· b_1)
13:     else if j = i + 2 then
14:         d_{i+2} = d_{i+1} = d_i = 0
15:         SUBTRACT(F_{i+1}, b_{i−1} ··· b_1)
16:     else                                              ▷ j > i + 2
17:         d_{j−1} = 1
18:         SUBTRACT(F_{j−2}, b_i ··· b_1)
```

Here is a worked example for the subtraction of $F_n - B$ as processed step by step by the algorithm. Digits in bold are the ones we are looking at at each stage.

$$\begin{array}{rccccccc}
 & \mathbf{1} & \mathbf{0} & 0 & 0 & 0 & 0 & 0 \\
- & & \mathbf{1} & 0 & 0 & 1 & 0 & 1 \\
\hline
 & & & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 & 0 \\
 & & - & & & \mathbf{1} & 0 & 1 \\
\hline
 & & & & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
 & & & - & & & & \mathbf{1} \\
\hline
 & & & & & & \mathbf{1} & \mathbf{0} \\
 & & & & & - & & \mathbf{1} \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}$$

We check that

$$\begin{aligned}
1000000 &= F_7 = 21, \\
100101 &= F_6 + F_3 + F_1 = 13 + 3 + 1 = 17, \\
101 &= F_3 + F_1 = 3 + 1 = 4.
\end{aligned}$$

Indeed, $1000000 - 100101 = 21 - 17 = 4 = 101$.

6. Multiplication

6.1. **Overview and Preliminaries.** Naturally, we want to include multiplication in our arithmetic. Given x, y one might naïvely perform $O(x)$ additions of y to

itself for a pseudo-polynomial running time of $O(xn)$. In this section, we show how to perform multiplication in strongly polynomial $O(n^2)$ time.

6.2. **Procedure and Analysis.**

6.2.1. *Procedure.* We start with two nonnegative integers given in binary Fibonacci representation. Convert both to maximal form so $x = a_n a_{n-1} \dots a_1, y = b_n b_{n-1} \dots b_1$, where we pad the smaller number with zeroes such that both are of length n, without loss of generality. Obviously, if either x or y is zero, then the product is zero and we return this immediately.

We proceed by generating a table of n entries corresponding to 1, 10, 100, ... We will multiply y by each of these Fibonacci numbers and add the corresponding products for each Fibonacci number which appears in x to find the correct result.

For the first entry, write the representation of y. For the second entry, write $10 \times b_n b_{n-1} \dots b_1 = y + y$ in $O(n)$. For the third entry onward, add the previous two entries in $O(n)$. We can generate the multiplication table in this way since $y \times F_i = y \times (F_{i-1} + F_{i-2})$.

Once the table is generated, perform add on each entry which corresponds to a one in the representation of x. Return this sum, which has been already converted to maximal form.

6.2.2. *Correctness.* Follows from $\sum_i a_i F_i \times \sum_k b_k F_k = \sum_i [a_i F_i \times \sum_k b_k F_k]$.

6.2.3. *Runtime.* Since there are at most $O(n)$ of the table entries, and each addition takes $O(n)$, we have a total of $O(n^2)$ running time. This incorporates the $O(n^2)$ to generate the table as well.

It is important to recognize that the length of y, even when multiplied by F_n, remains $O(n)$ because of Lemma 2.5.

6.3. **Pseudocode and Example.** We give the following example and pseudocode to demonstrate the algorithm in practice: 1001×1010. We generate the table

1	1001
10	**10101**
100	101000
1000	**1010001**

and add bold rows to obtain $10101 + 1010001 = 10010000$, which is 42 as expected.

Algorithm 4 MULTIPLY(x, y)

Input: Binary Fibonacci representations $x = a_n a_{n-1} \cdots a_1$ and $y = b_n b_{n-1} \dots b_1$.
Output: MULTIPLY(x, y) returns $x \times y$ in maximal representation.

```
1: procedure MULTIPLY(x, y):
2:     x, y ⟵ TOMAXREP(x, n), TOMAXREP(y, n)
3:     T ⟵ [y, ADD(y, y)]
4:     for (i = 2; i ≤ n; i++) do
5:         T.append(ADD(T[i − 1], T[i − 2]))
6:     product = 0
7:     for i = 1; i ≤ n; i++ do
8:         ADD(product, x[i] × T[i − 1])
9:     return product
```

6.4. **Conclusion.** We obtain multiplication in strongly polynomial $O(n^2)$.

7. Conclusion

In this paper, we have introduced linear time addition, subtraction, and conversion algorithms and a quadratic time multiplication algorithm, which makes binary base Fibonacci a potentially useful and efficient system of computation.

Further research could explore the use of other sequences as bases for arithmetic. Sequences that are similar in structure to the Fibonacci sequence and are more dense in terms of differences between consecutive members of the sequence should be able to represent all numbers in binary form. In this paper, we note how Fibonacci base addition makes use of the sequence definition, and we would imagine that other sequences behave similarly. Such research could be important if there are sequences other than Fibonacci that conveniently represent phenomena so that we can analyze them without having to convert to base ten. One sequence that could be analyzed is the Lucas sequence, $2a_n + a_{n-1}$, as well as versions of the Fibonacci sequence with different initial cases.

References

[1] Robert Silber. "Wythoff's Nim and Fibonacci Representations". In: *The Fibonacci Quarterly 15* (1977), pp. 85–88.

[2] Peter W Shor and Stephen P Jordan. "Estimating Jones polynomials is a complete problem for one clean qubit". In: *arXiv preprint arXiv:0707.2831* (2007).