Fibonacci Number Base

Zixuan Chen, Ke Liu, and Franklin Volcic

April 19, 2022

Abstract

Number bases allow us to write down numbers. In the modern era, base 10 is the common base used in speech. Base 2 is what computers commonly use. Each base has it's strengths and weaknesses. In this paper, we consider a unique number base that can represent all positive integers. This number base is the Fibonacci number base. Throughout this paper, we will define the Fibonacci number base, then give a number of algorithms for converting a number to the Fibonacci number base. Further, we also provide algorithms to do different mathematical operations on numbers in the Fibonacci number base (FNB), such as addition and subtraction.

1 Introduction

The Fibonacci sequence is defined by $f_0 = 0, f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geqslant 2$. So we have $(f)_n$=0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Definition 1. A **Fibonacci representation** of a positive integer x, (also called a **Fibonacci based number**) is defined as a string of digits $a_n a_{n-1} ... a_1 a_0$ where each digit $a_i \in \{0, 1\}$ such that $x = \sum_0^n a_i f_i$. The value that this string of digits takes on in base 10 can be expressed by multiplying each digit a_i by the corresponding Fibonacci number, f_i, and then summing all these values together.

Namely,

$$(a_n a_{n-1} ... a_1 a_0)_{Fib} = a_n \cdot f_n + a_{n-1} \cdot f_{n-1} + \cdots + a_2 \cdot f_2 + a_1 \cdot f_1 + a_0 \cdot f_0$$

To see an example of this, consider the number 1010 in base Fibonacci. This value in base 10 is equivalent to

$$(1010)_{\text{Fib}} = 1 \cdot f_3 + 0 \cdot f_2 + 1 \cdot f_1 + 0 \cdot f_0 = 1 \cdot 2 + 1 \cdot 1 = 3$$

Fibonacci representations for 3 are not unique:

$$(1101)_{\text{Fib}} = 1 \cdot f_3 + 1 \cdot f_2 + 0 \cdot f_1 + 1 \cdot f_0 = 1 \cdot 2 + 1 \cdot 1 = 3$$

What's more, note that the following is not a Fibonacci representation of 5, because $a_3 = 2 \notin$\{0,1\}.

$$(2010)_{Fib} = 2 \cdot f_3 + 1 \cdot f_1 = 5$$

Problem: How to find a unique representation for each positive integer? How to do addition, subtraction and multiplication of two Fibonacci based numbers? Can the above arithmetic be done in linear time with respect to the number of digits? Even better, can we follow a particular order, like from right to left when we do binary addition, and process each digit for only constant amount of steps?

2 Greedy Fibonacci representation

Before trying to do arithmetic, it is natural to ask: For every positive integer x, does there exists a Fibonacci representation for it?

Also, we notice that sometimes Fibonacci representations for a positive integer are not unique. For instance, if we take $x = 4$,

$$\begin{aligned}4 &= 3 + 1 = f_4 + f_2 = (10100)_{Fib}\\ &= 3 + 1 = f_4 + f_1 = (10010)_{Fib}\\ &= 3 + 1 + 0 = f_4 + f_1 + f_0 = (10011)_{Fib}\\ &= 2 + 1 + 1 = f_3 + f_2 + f_1 = (1110)_{Fib}\end{aligned}$$

So we also want to ask: how to assign a unique Fibonacci representation for each positive integer?

We are going to introduce an algorithm that answers the two questions. We call the representation generated by greedy algorithm **greedy Fibonacci representation**.

2.1 Construction of the greedy algorithm

Here we prints greedy Fibonacci representation of x=0,1,...,10.

Algorithm 1 Greedy Algorithm

```
Input: a positive integer
Output: a greedy Fibonacci representation
                                                        ▷ Base case.
$x_1 \leftarrow x$
$k_1 \leftarrow$ the largest natural number such that $f_{k_1} \leqslant x_1 < f_{k_1+1}$
$a_{k_1} \leftarrow 1$
for j from 0 to $k_1 - 1$ do $a_j \leftarrow 0$
end for
$x_2 \leftarrow x_1 - f_{k_1}$
                                                        ▷ Iteration.
for $i > 1$, do
    if $x_i$ is not 0: then
        $k_i \leftarrow$ the largest natural number such that $f_{k_i} \leqslant x_i < f_{k_i+1}$
        $a_{k_i} \leftarrow a_{k_i} + 1$
        $x_{i+1} \leftarrow x_i - f_{k_i}$
    else
        return $a_i a_{i-1} ... a_1 a_0$
    end if
end for
```

		8	5	3	2	1	1	0
x	Greedy(x)	a_6	a_5	a_4	a_3	a_2	a_1	a_0
0	Greedy(0)							0
1	Greedy(1)					1	0	0
2	Greedy(2)				1	0	0	0
3	Greedy(3)			1	0	0	0	0
4	Greedy(4)			1	0	1	0	0
5	Greedy(5)		1	0	0	0	0	0
6	Greedy(6)		1	0	0	1	0	0
7	Greedy(7)		1	0	1	0	0	0
8	Greedy(8)	1	0	0	0	0	0	0
9	Greedy(9)	1	0	0	0	1	0	0
10	Greedy(10)	1	0	0	1	0	0	0

2.2 Existence and uniqueness of greedy Fibonacci representation

Theorem 1. *For any positive integer x, the Greedy algorithm always generate a unique greedy Fibonacci representation* $(a_n a_{n-1} ... a_1 a_0)_{Fib} = Greedy(x)$ *for x such that* $\forall i$, $a_i \in \{0, 1\}$ *and* $a_n a_{n-1} ... a_1 a_0)_{Fib} = (a_n \cdot f_n + a_{n-1} \cdot f_{n-1} + \cdots + a_2 \cdot f_2 + a_1 \cdot f_1 + a_0 \cdot f_0$.

Proof. **Existence**

Notice that $f_n \to \infty$ as $n \to \infty$. Therefore, $\forall x_i$, $\exists k_1$ such that $f_{k_1+1} > x_i > 1 = f_2$. Also notice that $f_0 = 0 < f_1 = 1 < x_i$ for any positive integer x_i. So the algorithm is always able to output a result.

Now we'll prove that the output of greedy algorithm will not contain any $a_i \geqslant 2$: Notice that $\forall k > 2$, $f_k > f_{k-1}$, so $2f_{k_i} > f_{k_i-1} + f_{k_i} = f_{k_i+1}$.In the i-th step, if the greedy algorithm picked k_i, then $f_{k_i} \leqslant x_i < f_{k_i+1} \leqslant 2f_{k_i}$ we have $x_{i+1} = x_i - f_{k_i} < f_{k_i}$. So in the (i+1)-th step, the algorithm will not pick the same Fibonacci number again.Also note that x_j is always decreasing. Therefore, each digit a_j will not increase more than once.

That complete the proof of existence. Hence the greedy algorithm will always give a well defined Fibonacci representation for every valid input. □

Proof. **Uniqueness** Since in each recursion, the algorithm pick the largest possible k_i, the output is unique. □

2.3 Properties of greedy Fibonacci representation

Listing outputs Greedy(1),Greedy(2),...,Greedy(10), You may observe that the last two digits are always 00.

Theorem 2. *Greedy Fibonacci representation always ends in 00.*

Proof. For any positive integer x, suppose Greedy algorithm on x runs n steps. Considering the last x_n that is nonzero, there are two cases.

> Case 1. $x_n > 1$. Then x_n must be a Fibonacci number f_{k_n} for some $k_n > 1$ exactly. Then $x_{n+1} = x_n - f_{k_n} = 0$, and the algorithm return a result, leaving a_1 and a_0 both zeros.
> Case 2. $x_n = 1$. Though f_2 and f_1 are both equal to 1, $k_n = 2$ is picked in the n-th step, because $2 > 1$ and 2 is the largest index. Then x_{n+1} is set to be $x_n - 1 = 0$. Then the algorithm return a result, leaving a_1 and a_0 both zeros.

□

Another property of greedy Fibonacci representation is as follows.

Theorem 3. *There are no consecutive 1's in the greedy Fibonacci representation.*

Proof. We prove by contradiction. Suppose $a_j = a_{j-1} = 1$. Then we know that in step i for some i$\in \mathbb{N}$, the algorithm picked k_j, which implies $f_{k_j} \leqslant x_i < f_{k_j+1}$; and in the following step i+1, the algorithm picked k_{j-1}, which implies $f_{k_{j-1}} \leqslant x_{i+1} = x_i - f_{k_j}$. Now we have

$$f_{k_{j+1}} = f_{k_j} + f_{k_{j-1} \leqslant x_i < f_{k_{j+1}}}.(1)$$

Which implies that in step i the algorithm should pick $f_{k_{j+1}}$. Contradiction. □

The theorem says there are no "...11...", "...111...", "...11.(m 1's)..11..." in the output of our greedy algorithm. If we transfer each number in to greedy Fibonacci representation before doing arithmetic, we can significantly decrease the cases to be considered.

2.4 Converting non-greedy Fibonacci representation to greedy Fibonacci representation

Given an arbitrary Fibonacci representation $(d_n d_{n-1} ... d_1 d_0)_{Fib}$ of some positive integer x with $d_0 = d_1 = 0$, how can we transfer it into the greedy Fibonacci representation $(a_n a_{n-1} ... a_1 a_0)_{Fib}$? The answer is very simple. We know that d_i could be either 0 or 1. Just go from right to left, and whenever you see n consecutive 1's, go from the left of these 1's and replace (0)11 with 100 and write 1 when you see a single 1, write 0 when you see 0. Here is an example:

$(d_n d_{n-1} ... d_1 d_0)_{Fib}$		1	1	1	1	0	1	1	1	1	0	0	1	0	0
see 0															0
see 0														0	0
see single 1													1	0	0
see 0												0	1	0	0
see 0											0	0	1	0	0
see 4 consecutive 1's go from left to right						1	0	0			0	0	1	0	0
continue processing consecutive 1's						1	0	1	0	0	0	0	1	0	0
see 4+1 consecutive 1's		1	1	1	1	1	0	1	0	0	0	0	1	0	0
process from the left most 1	1	0	0				0	1	0	0	0	0	1	0	0
process consecutive 1's	1	0	1	0	0		0	1	0	0	0	0	1	0	0
see a single 1	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0

In this way, we converted a non-greedy Fibonacci representation to a representation without consecutive 1's. Since it could be proved that "a Fibonacci representation with last two digits being 00 has no consecutive 1's if and only if it is a greedy representation", in fact what we get after converting $(d_n d_{n-1} ... d_1 d_0)_{Fib}$ is a greedy Fibonacci representation.

3 Addition in the Fibonacci Number Base

In this section, we are going to define an algorithm which takes two numbers in the FNB and adds them, returning their result in the greedy representation.

3.1 Addition Background

To start this section, let us first consider what is actually happening when we add two numbers in the FNB.

Let's first assume that we have two numbers, A and B where $A = a_n a_{n-1} \dots a_0$ and $B = b_n b_{n-1} \dots b_0$. We know that in base 10,

$$(A)_{10} = a_n \cdot f_n + a_{n-1} \cdot f_{n-1} + \cdots + a_0 \cdot f_0$$

and ,

$$(B)_{10} = b_n \cdot f_n + b_{n-1} \cdot f_{n-1} + \cdots + b_0 \cdot f_0$$

Thus, it follows that

$$(A + B)_{10} = (a_n + b_n) \cdot f_n + (a_{n-1} + b_{n-1}) \cdot f_{n-1} + \cdots + (a_0 + b_0) \cdot f_0$$

While this will result in the correct base 10 number after addition, we cannot simply add the digits in base Fibonacci, as some digit's may take the value of 2. For example,

$$\begin{array}{r} 1\,0010\,0000 \\ +\,0\,1010\,0000 \\ \hline 1\,1020\,0000 \end{array}$$

As we can see, if we only add the digits of each value, we may end up digits that take on a value larger than 1. This is clearly not in the greedy representation nor a legal Fibonacci Number Base representation. To address this issue, we introduce something we called **expansion rules**.

3.2 Expansion Rules

In this section, we introduce **expansion rules**. Formally, An **expansion rule** is the greedy representation of some Fibonacci number f_n multiplied by some positive integer k.

We will look at the first 3 expansion rules

Expansion Rules	
	$K \cdot f_n$
$K = 1$	f_n
$K = 2$	$f_{n+1} + f_{n-2}$
$K = 3$	$f_{n+2} + f_{n-2}$

Every expansion rule can be constructed from the definition of the Fibonacci sequence. Now that we have a definition of an **expansion rule**, we can now define an addition algorithm.

3.3 The Algorithm

Here we will define the algorithm, which we will refer to as the **recursive addition algorithm**. This algorithm simply adds two numbers in FNB digit by digit, then apply a small set of rules recursively until the result is in the greedy representation, at which point the algorithm terminates.

Algorithm 2 Recursive Addition Algorithm

```
    Input: Numbers $n_1$ and $n_2$ in FNB
    Output: The result of $n_1 + n_2$ in FNB greedy representation
 1: Add the corresponding digits of $n_1$ and $n_2$, store result in $n_3 = d_k \dots d_0$
 2: if digit $d_i \geq 1, d_{i+1} \geq 1$ then
 3:     $d_i = d_i - 1$
 4:     $d_{i-1} = d_{i-1} - 1$
 5:     $d_{i+1} = d_{i+1} + 1$
 6:     Goto 2
 7: end if
 8: if digit $d_i > 1$ then                    ▷ Apply expansion rule for $k = 2$
 9:     $d_i = d_i - 2$
10:     $d_{i+1} = d_{i+1} + 1$
11:     $d_{i-2} = d_{i-2} + 1$
12:     Goto 2
13: end if
14: if digit $d_0 > 0$ then                    ▷ digit $d_0$ represents $f_0 = 0$
15:     $d_0 = 0$
16:     Goto 2
17: end if
18: if digit $d_1 > 0$ then                    ▷ digit $d_1$ represents $f_1 = 1 = f_2$
19:     $d_2 = d_2 + d_1$
20:     $d_1 = 0$
21:     Goto 2
22: end if
23: Return $n_3$
```

Now that we have defined the algorithm, we will use it to add the following numbers in FNB: 101000 and 100000

$$\begin{array}{r} 101000 \\ +\,100000 \\ \hline 201000 \\ 1002000 \\ 1010010 \\ 1010100 \end{array}$$

As we can see, applying the rules 1 by 1, we are able to add these two numbers and get a resulting number in base Fibonacci which in also in the greedy representation.

3.4 Algorithm Termination

This algorithm clearly adds the one example that we used it on, but how can we be certain that our algorithm will add numbers and always terminate? In this section, we will prove that this algorithm always terminates using proof by contradiction.

To prove this, we will need to note two lemmas.

Lemma 4. *For any positive integer N, there exists some Fibonacci number f_i such that $f_i > N$*

Proof. To prove this, we first note that since the n-th Fibonacci number, f_n, is the sum of the prior two Fibonacci numbers, $f_{n-1} + f_{n-2}$, it follows that $f_n > f_{n-1}$ when $n > 2$.

We also note that the n-th Fibonacci number is greater than or equal to n when $n \geq 5$. Namely, $n \leq f_n$. We know this as when $n = 5$, $f_n = 5$. Since $f_n > f_{n-1}$ and the Fibonacci numbers are always integers, f_n will always be at least equal to n after $n = 5$.

Using these two facts, we can now say that for any positive integer N, $f_N \geq N$, and therefore $f_{N+1} > N$, completing the proof.

□

Lemma 5. *Any time a rule is applied to some set of digits $a_j, a_{j-1}, \ldots$ in the addition algorithm, a more significant digit a_n is incremented where $n > j, j-1, \ldots$*

Proof. To prove this, we simply must prove that each case in the algorithm increments a more significant digit. If two contiguous digits are greater than or equal to 1, then we will increment the next largest digit, and decrement the two digits. If $d_1 > 0$, then we increment d_2 and decrement d_1. In the final case when a digit $d_i > 1$, we increment the digit d_{i+1} which is a more significant digit. As we can see, all cases increment a more significant digit, completing the proof. □

Theorem 6. *The recursive addition algorithm terminates for all inputs with a finite number of digits.*

Proof. Assume that we are adding two numbers, n_1 and n_2, in base Fibonacci, and assume the result of $n_1 + n_2 = n_3$. Finally, assume that the algorithm never terminates.

From theorem 4, we know that there exists some Fibonacci number f_i where $f_i > n_3$. From definition of numbers in the Fibonacci base, any time a digit larger than or equal to the $i - th$ digit is 1, the resulting value is larger than equal to the value f_i. Since every iteration of our addition algorithm applies a rule that sets a more significant digit than any of the current digits, we can only have a finite number of iterations until a more significant digit than the $i - th$ digit is set, leading to a value which is larger than n_3. This is a contradiction as the result of the algorithm is equal to n_3, and therefore the algorithm must always terminate.

□

4 Linear Addition

Let's recall the question asked in the beginning. Can we add two number Fibonacci based numbers following a particular order, like from right to left when we do binary addition, and process each digit for only constant amount of steps? Here is an addition method that satisfy these requirements.

4.1 Properties of direct sum

Before we going to the detailed method, let's first look at some useful properties that follows form greedy representation.

Definition 2. Direct sum of two greedy represented Fibonacci based number $(a_n a_{n-1} ... a_1 a_0)_{Fib}$ and $(b_n b_{n-1} ... b_1 b_0)_{Fib}$ is given by $(c_n c_{n-1} ... c_1 c_0)$, where $c_i = b_i + a_i$, $\forall i$.

It follows from $a_i, b_i \in \{0, 1\}$ that $c_i \in \{0, 1, 2\}$.

Definition 3. $n \cdot (ab) = abab...ab$ where ab appears n times. a, b are some integers, n is some positive integer. Similarly, $n \cdot (c) = cc...c$ where c appears n times. c is some integer and n is some positive integer.

For instance, $3 \cdot (20) = 202020$. $1 \cdot (1) = 1$. It follows that,

Theorem 7. *Any direct sum of two Fibonacci based number is composed of the following three patterns: $n_1 \cdot (0)$, $n_2 \cdot (1)$, $n_3 \cdot (20)$ for some n_1, n_2, n_3 being positive integer.*

Theorem 8. *Whenever 2 appears in the direct sum of two Fibonacci based number, the left neighbour and right neighbour of 2 are always 0.*

This is because there are no consecutive 1's in greedy representation. Suppose $c_i = 2$, then we must have $a_i = b_i = 1$, which implies $a_{i-1} = a_{i+1} = b_{i-1} = b_{i+1} = 0$. Therefore, $c_{i-1} = c_{i+1} = 0$.
For instance,

$$\begin{array}{lcccccccccc}
 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
+ & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\hline
= \text{(direct sum)} & 2 & 0 & 1 & 1 & 0 & 2 & 0 & 2 & 0 & 0
\end{array}$$

4.2 Linear Addition algorithm

Linear Addition Algorithm

Input: Fibonacci based number $(a_n a_{n-1} ... a_1 a_0)_{Fib}$ and $(b_n b_{n-1} ... b_1 b_0)_{Fib}$, both in Greedy Fibonacci representation

Output the addition result $d = (d_n d_{n-1} ... d_1 d_0)_{Fib}$in greedy representation.

```
1: d← $n \cdot (0)$; $c = (c_n c_{n-1} ... c_1 c_0) \leftarrow$ direct sum of $(a_n a_{n-1} ... a_1 a_0)_{Fib}$ and
   $(b_n b_{n-1} ... b_1 b_0)_{Fib}$
2: Look at $(c_n c_{n-1} ... c_1 c_0)$ from right to left, check the patterns instead of single
   digits each time:
3: if see $n \cdot (0)$ or $n \cdot (01)$ for some n then                              ▷ Case 0
4:     copy what is seen to corresponding digits of d.
5: else if see $n \cdot (1)$ then                                                     ▷ case 1
6:     go from the left of these 1's and replace (0)11 with 100 and write 1 when
   there left a single 1 (see similar idea in section 2.4).
7:     Move to the leftmost 1's and go to step 2.
8: else if see $k \cdot (02) m \cdot (10)0$ then                                      ▷ Case 2
9:     if k=1 then
10:        Add 1 to c's digit corresponding to 02's left neighbour, write 00 in
   d's digits corresponding to 02, write 00 in d's digits corresponding to the
   leftmost 10Copy the remaining 10 to d's corresponding digits. Fill unwritten
   digits of d using 0.
11:    else
12:        Add 1 to c's digit corresponding to leftmost 02's left neighbour, write
   00 in d's digits corresponding to leftmost 02; Write 01 in d's digits corresponding
   to the else 02's. write 00 in d's digits corresponding to the leftmost
   10 . Copy the remaining 10 to d's corresponding digits. Fill unwritten digits
   of d as 0.
13:    end if
14:    Move to the left most 02's 2's position and go to step 2.
15: else if see $k \cdot (02)00$ then                                                 ▷ Case 3
16:    if k=1 i.e. see 0200 then
17:        write 1001 in d's correspoding digits.
18:    else if k=2 i.e. see 020200 then
19:        Add 1 to c's digits corresponding to leftmost 02's left neighbor, write
   000001 in d's digits corresponding to 020200. Fill unwritten digits of d as
   0.
20:    else
21:        Add 1 to c's digits corresponding to leftmost 02's left neighbor, write
   01 in d's digits corresponding to 00. Write 00 in d corresponding to the
   rightmost and leftmost 02. Write 01 in d correspoding to the remaining
   02's. Fill unwritten digits of d as 0.
22:    end if
23:    Move to the left most 02's 2's position and go to step 2.
24: else if see $k \cdot (02) m \cdot (01)00$ then                                    ▷ Case 4
25:    write 10 in d for each 01 in c.
26:    if k=1 then
27:        write 10 in d for 02 in c, go to step 2.
28:    else
29:        Add 1 to c's digit corresponding to left neighbour of leftmost 02.
   Write 00 in d for left most and right most 02. write 01 for the else 02, if
   there are.
```

```
30:     end if
31:     Go to step 2.
32: end if
33: return d
```

4.3 Example

a		0	1	0	1	0	0		0		0	1	0		1	0	1	0	0	
b		0	1	0	1	0	0		1		0	1	0		0	1	0	0	0	
c		0	2	0	2	0	0		1		0	2	0		1	1	1	0	0	
d																		0	0	case 0 see 00
d														(c +1)	0	0	1	0	0	case 1 see111
d										(c +1)	0	0	0		0	0	1	0	0	case2 k=1 see 02100
d								(c+1)	0		0	1	0		0	0	1	0	0	case 3 k=1 see 0200
d	(c +1)	0	0	0	0	1	0		0		0	1	0		0	0	1	0	0	case 4 02020100
d	1	0	0	0	0	1	0		0		0	1	0		0	0	1	0	0	case 0 see 01 return d

4.4 Complexity

This algorithm will check digits and write each digit at most 2 times, so it takes O(n) operations. Hence is linear.

4.5 Validity of Linear Addition Algorithm

We know from previous properties that 2 will always appear with left and right neighbours being 0 in c. Because of case 1, the right hand side of what step 2 is looking at will always have no consecutive 1's. Since the transfer will only affect the area corresponding to 1111 and the left neighbor of the leftmost 1, whenever there are 0202...02 in original c, the right most 02's right neighbour will only be 0 or 1, not 2. What's more, notice that pattern of 0101...01 will have a right neighbour 0. It follows that we have these five cases.

Those actions to do in each cases are inducted from (0200)=(1001) and (100)=(011). Here are some data.

Case 1	00210101010000	0020210101010000
	10000101010000	1000100101010000
	002020210101010000	00202020202101010100000
	100010100101010000	10001010101001010100000
Case 2	0020000	002020000
	0100100	100000100
	00202020000	0020202020200000
	10001000100	100010101000100
Case 3	002010101010000	00202010101010000
	10101010100100	10000101010100100
	0020202010101010000	002020202020101010100000
	1000100101010100100	10001010100101010100100

5 Subtraction Algorithm

The next goal for the project is to find subtraction rule that proceed in linear time in the number of digits. With the subtraction rule, we can take any two numbers using the greedy representation, which has a binary symbols 0 or 1 in every digit, and subtract them to get the result in a Fibonacci number base's greedy representation. In the subtraction algorithm, the input are two numbers a, b, where $a > b$, and both are in greedy representation, noted as $(a)_{Fib}$ and $(b)_{Fib}$. The output are $a - b$ in greedy representation, noted as $(a-b)_{Fib}$.

5.1 Intuitive Subtraction Algorithm

First, we study an intuitive algorithm of subtraction to show that by using the expansion rule $f_k = f_{k-1} + f_{k-2}$, we can get a different form of number a in binary Fibonacci base, and use the new a to subtract with b to get the final result.

Here is the example showing the steps of intuitive subtraction algorithm. In the example 5 minus 4, we have $a = 5 = (100000)_{Fib}$, $b = 4 = (10100)_{Fib}$.

Step 1:

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

Step 2: In 3rd digit, we have 0 minus 1, which means a's 3_{rd} digit $<$ b's 3_{rd} digit, then we need to continuously read the digit of a from current place 3_{rd} to the left, until we find the first 1 in the 6_{th} digit. Then, we shall apply the expansion rule to the 6_{th} digit, which means the 100000 will be changed to 011000. Below shows the result after step 2:

Algorithm 3 Intuitive Subtraction Algorithm (Pseudocode)

```
    **Input:** $(a)_{Fib}$ and $(b)_{Fib}$, where $a > b$
    **Output:** $(a - b)_{Fib}$ in greedy representation
    Time complexity: $O(n^2)$
Read the digit $i_{th}$ of both a and b from right to left
**while** $a$'s $i_{th}$ digit $<$ $b$'s $i_{th}$ digit **do**
    Read the digit of a from right to left, started from $i^{th}$ digit, until reach the first 1
    Expand the digit 1 (change digits "100" → "011")
**end while**
When finishing reading the leftmost digit of b, subtract the altered number $a$ with $b$
Modify the last two digits of the result to get a greedy representation
```

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

Step 3: Still, we noticed that a's 3_{rd} digit $<$ b's 3_{rd} digit. So the algorithm will apply the expansion rule to the 4_{th} digit, which is a 1. Now the $a = 011000$ is changed to the 010110.

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

Step 4: There is no digit for a's i_{th} digit $<$ b's i_{th} digit, so we break out from the while loop, and we can output the raw solution as below.

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

Step 5: The raw result which is 000010 is not in greedy representation, because the last two digits are not both 0. In this case, we will use the f_2 digit to replace f_1 digit. So 000010 is changed to 000100. Finally, we can remove the leading 0s in this Fibonacci number base. So the final output is 100, and $(100)_{Fib} = 1$.

Now let's look into the time complexity in this algorithm. We will find that by reading the digits of a and b together from right to left, it takes $O(n)$ time complexity in regard to the number of digits. Inside the while loop, we read the digit of number a from right to left, which also costs $O(n)$ times. The step of expanding the digits only takes linear time. So the total time complexity is $O(n) \times O(n) = O(n^2)$.

5.2 Advanced Subtraction Algorithm

With the goal to obtain a the subtraction algorithm with $O(n)$ time complexity, we raised the Advanced Subtraction Algorithm. It works by reading the digits from right to left, and deals with the 3 consecutive digits ($i_{th}, (i-1)_{th}, (i-2)_{th}$ digit) in each reading step. Here, we'll introduce the concept of borrowing digit, that is when at the i_{th} digit, borrow 1 from the left digit $(i+1)_{th}$, and give it to the $i_{th}, (i-1)_{th}$ digits. Note that i is increasing in the direction from right to left, indexing from 0. So the rightmost digit is 1_{st} digit.

To represent the implementation of borrowing digit clearly, we will have four lines of digits: first line is the borrow line(marked with b), being used to indicate if this column of digit is going to borrow a digit from its left side; second line is the minuend number a; the third line is the modifed line(marked with m), being used to show that the borrowed digit from left-hand side is being add to current column of digit; and the fourth line is the subtrahend number b. To calculate the result, we only need to combine all numbers in $2_{nd}, 3_{rd}$, and 4_{th} line.

By having the algorithm in mind, we can look into the example of 5 minus 4, which is $(100000)_{Fib} - (10100)_{Fib}$. Since we move from right to left step by step, and in each step, we will need to handle not only the i_{th} digit's output itself, but also the output of $(i-1)_{th}$ and $(i-2)_{th}$ digit, we will use a box with 3-digit width to simulate the whole process. Also, we will include a borrow line storing the borrow digit, and a modifying line indicating the modified digit.

Step 1: Get the output of last two digits as 0

```
  b:
- - - - - - - - - - - - - - - -
      1   0   0   0   0   0
  m:
  -       1   0   1   0   0
-------------------------------
                      0   0
```

and we will write it in the form for later convenience:

```
  b:
- - - - - - - - - - - - - - - -
      1   0   0   0   0   0
  m:
         -1   0  -1   0   0
-------------------------------
                      0   0
```

Step 2: Start at 3_{rd} digit, this is the $0-1$ case, so it needs to borrow a digit from its left-hand side. Thus, we should change its borrow digit to -1, and change its modified digit to +1. Also, we shall write its output to 0 as a result of combining it's original digit and modified digit($-1+1=0$).

```
                      ¦             ¦
  b:                  ¦             ¦
- - - - - - - - - - - ¦ - - - - - - ¦
      1   0   0       ¦ 0   0   0   ¦
  m:                  ¦             ¦
         -1   0       ¦-1   0   0   ¦
----------------------¦-------------¦
                      ¦     0   0   ¦
                      ¦             ¦
```

Algorithm 4 Advanced Subtraction Algorithm

```
    Input: $(a)_{Fib}$ and $(b)_{Fib}$, where $a > b$
    Output: $(a - b)_{Fib}$ in greedy representation
    Time Complexity: $O(n)$
Read the digit $i^{th}$ of both $a$ and $b$ from right to left    ▷ assume that the digits in the left of $b$'s leftmost digit are 0s
if $a$'s $i_{th}$ digit $<$ $b$'s $i_{th}$ digit then    ▷ $0 - 1$
    Change the borrow digit to -1, and change the modified digit to +1
    Write the output digit to 0
else if $a$'s $i_{th}$ digit $=$ $b$'s $i_{th}$ digit then    ▷ $0 - 0$, $1 - 1$
    if $(i - 1)_{th}$ borrow digit $= 0$ then
        Write the output digit to 0
    else if $(i - 1)_{th}$ borrow digit $= 1$ then
        Change the borrow digit to -1, and the $(i - 1)_{th}$'s borrow digit +1;
        the $(i - 2)_{th}$ borrow digit +1 if it need to borrow,
        or the $(i - 2)_{th}$ modified digit +1 if it doesn't need to borrow

        Write the output digit to 0
        Update the $(i - 1)_{th}$, $(i - 2)_{th}$ output digits
    end if
else if $a$'s $i_{th}$ digit $>$ $b$'s $i_{th}$ digit then    ▷ $1 - 0$
    if $(i - 1)_{th}$ borrow digit $= 0$ then
        Write the output digit to 1
    else if $(i - 1)_{th}$ borrow digit $= 1$ then
        Change the borrow digit to -1, and the $(i - 1)_{th}$ digit +1;
        the $(i - 2)_{th}$ borrow digit +1 if it need to borrow,
        or the $(i - 2)_{th}$ modified digit +1 if it doesn't need to borrow

        Write the output digit to 0
        Update the $(i - 1)_{th}$, $(i - 2)_{th}$ output digits
    end if
end if
Stop when finish reading the leftmost digit of $a$
Modify the last two digits of the result to get a greedy representation
```

box:

	a_3	a_2	a_1
b:	-1		
	0	0	0
m:	+1		
	-1	0	0
	0	0	0

Step 3: For 4_{th} digit, this is a $0 - 0$ case. It has a borrow digit in its right-hand side, as the 3_{rd} digit is -1. By following the advanced subtraction algorithm, it should borrow a digit itself, and lend it to the 3_{rd} and 2_{nd} digit. So the current borrow digit is -1, and add 1 to 3_{rd} borrow digit. Note that the 2_{nd} borrow digit is 0, so we should add 1 to 2_{nd} modified digit. By combining the original digits of a and b with the modified digit, we can update 3_{rd} and 2_{nd}'s output digit as below.

b:				-1		
	1	0	0	0	0	0
m:				+1		
		-1	0	-1	0	0
				0	0	0

box:

	a_4	a_3	a_2
b:	-1	-1+1	+1
	0	0	0
m:		+1	0
	0	-1	0
	0	0	1

then,

	a_4	a_3	a_2
b:	-1	0	0
	0	0	0
m:		+1	+1
	0	-1	0
	0	0	1

Step 4: For 5_{th} digit, this again is a $0 - 1$ case. So we can change its borrow digit to -1, and add 1 to its modified digit. Finally, write the output digit to 0 as a result of combining +1 and -1.

b:			-1	0	0	
	1	0	0	0	0	0
m:				+1	+1	
		-1	0	-1	0	0
			0	0	1	0

box:

	a_5	a_4	a_3
b:	-1	-1	0
	0	0	0
m:	+1		+1
	-1	0	-1
	0	0	0

Step 5: For 6_{th} digit, this is a $1 - 0$ case. Therefore, we have a 1 in current digit to borrow. So we need to change current borrow digits to -1, and since we have borrow digit -1 in both 4_{th} and 3_{rd} digits, the borrow digit in both 4_{th} and 3_{rd} will be added 1.

b:		-1	-1	0	0	
	1	0	0	0	0	0
m:		+1		+1	+1	
		-1	0	-1	0	0
		0	0	0	1	0

box:

	a_5	a_4	a_3
b:	-1	-1+1	-1+1
	1	0	0
m:		+1	
	0	-1	0
	0	0	0

then,

	a_5	a_4	a_3
b:	-1	0	0
	1	0	0
m:		+1	
	0	-1	0
	0	0	0

Step 6: We reached to the leftmost digit of a, so now we have the output 000010 as below:

b:	-1	0	0	0	0	
	1	0	0	0	0	0
m:		+1		+1	+1	
-		1	0	1	0	0
	0	0	0	0	1	0

Step7: By modifying the last two digit to 0, and removing the leading 0s, we can convert the output to a greedy representation $(000010)_{Fib} \rightarrow (100)_{Fib}$.

The above example wraps up our implementation with the advanced subtraction algorithm. The three cases of $1 - 0$ case, $0 - 1$ case, and $1 - 1, 0 - 0$ case can conclude all the cases we are going to handle by sliding the 3-digit width box from right to left. This is a $O(n)$ time complexity by reading from right to left, and the process we had with each box is within linear time. Thus, the whole algorithm is within $O(n)$ time which satisfy our goal of subtraction method.

6 Conclusion

In conclusion, for all positive integers, we designed the Greedy Fibonacci representation that can help to generate a unique representation, along with the implementation of Addition Algorithm and Subtraction Algorithm. These algorithms are quite useful in handling the binary digits in Fibonacci Base Number using the concept of Fibonacci expansion rule, and it allows to explore a brand new way of arithmetic calculation(addition and subtraction) in $O(n)$ time complexity, and might be further expand to multiplication and division.

For future expectation, we hope these algorithms can be adapted with negative numbers. Though we are only exercising with positive number in this paper, it is not hard to find that negative numbers can also be represented in Fibonacci Number Base if given a set of rules. Once proved that the representation is exist and unique, the expansion rules can still be applied to all numbers. With this idea, this topic is worth digging deeper for more possible cases.

Acknowledgments

Section 1, 2 and 4 are written by Zixuan. Section 5 and 6 are written by Ke. Section 3 is written by Franklin.

References

[1] Fibonacci. 2015 *Jwilson.coe.uga.edu.* `http://jwilson.coe.uga.edu/emt668/emt668.folders.f97/norton/Final/Fibonacci.html` [Accessed 19 March 2015].