OEIS Math Circles Part 2

In part 2, we will talk about recursive sequence in particular the Fibonacci Numbers! I'd like to thank Craig Kaplan for a lot of the ideas in this section!

1 Fibonacci Numbers - A000045

1.1 Introduction

We define a recursive sequence of numbers as follows. Start with the numbers 0 and 1 and then every subsequent term is formed by taking the sum of the previous two terms. So calling the zeroth term 0 and the first term 1 we define the second term by $0+1=1$, the third term by $1+1=2$ the fourth term as $1+2=3$ and the fifth term by $2+3=5$. We continue to form the following sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, ...$$

A **Fibonacci Number** is one of the numbers in the above sequence. We can precisely define this mathematically by the following:

$$f_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{if } n \geq 2 \end{cases}$$

we say that $f_n = f_{n-1}+f_{n-2}$ is the **recurrence relationship** for the **recursive sequence** f_n. In general, a recursive sequence is one where future values of the sequence depend on previous ones. For fun, try doing a Google search on 'recursion' and see if you can find hte hidden Easter Egg joke the Google programmers coded!

Exercise: Which of the Fibonacci numbers above are perfect squares?

Solution

Only 1 and 144 are prefect squares. Interestingly these are the only perfect squares! (A result that was only just proven in the 2000s!)

These numbers were named after Leonardo di Pisa (later known as Fibonacci) who introduced the numbers to Western European mathematics in his 1202 book *Liber Abaci* (the book of calculation) however the sequence has roots in mathematics as early as 200 BCE in work by an Indian mathematician named Pingala on Sanskrit poetry.

There's honestly an endless amount we can talk about with the Fibonacci numbers but we're going to take this opportunity to talk about how to calculate these large Fibonacci numbers computationally.

We're going to do this using the language of Python. The reason for using Python is that the syntax is very easy to understand and allows for large computations with integers. If you're interested in trying this for yourself, you can do so online on the website: `https://replit.com/`

1.2 First Attempt - Naive Method

We start off by trying to write a computer program that directly corresponds to the mathematical definition.

```
1 def fib1(n):
2   if n <= 1: return n
3   return fib1(n-1) + fib1(n-2)
```

Just a brief introduction of the syntax:

- On line 1 above, the keyword `def` tells Python we wish to define a function. The name of the function is `fib1` and it takes a single parameter `n` which we think of as an integer. The colon at the end tells Python we are ready to define what the function does.
- **Very important note!** Note that Python is a **whitespace** language - when lines are more indented than others, it means they belong to the body of the previous line. So here lines 2-3 are the body of the function `fib1`. We will also see this later when we do looping.
- The second line is an `if` statement. When the condition that `n <= 1` is true, we do the command after the colon which in this case is to `return n`. The `return` keyword tells the function to stop running and give back the calling function the value `n`. This gives the correct value when `n` is 0 or 1.
- The third line is only executed if the `if` condition on line 2 evaluates to `False`. This occurs whenever `n >= 2` (assuming we only pass non-negative integers to this function). Recalling that in this case we want the sum of the previous two Fibonacci Numbers, we can make some sense of the third line which returns the value of `fib1(n-1)+ fib1(n-2)`

To use the above function, we can add the line `print(fib1(5))` for example to see the fifth Fibonacci Number.

It is a good idea to try this and the other code snippets in a Python interpreter. You can find one here `https://replit.com/` or install it yourself on your home computer using `https://cscircles.cemc.uwaterloo.ca/run-at-home/` for some initial help.

While the above code works it is woefully inefficient. Try calling `print(fib1(100))`. How long does it take to compute this value (or worse, does your program crash?) What's sad is that if you worked hard, you could probably compute this value by hand without a calculator if you wanted! The reason for this is related to the number of times we call certain values. In fact, if we call `fib(100)`, we can count that the number of times we eventually call `fib(1)` is equal to the integer value of `fib(100)` (which is a **very** large number!

1.3 Second Attempt - Saving Previous Values

If the problem is that we recompute a lot of computations above, then perhaps we can save some effort by storing the values in a list and using them. This gives the following:

```
1 def fib2(n):
2   fibs = [0, 1]
3   for i in range(n-1):
4     fibs.append(fibs[-1] + fibs[-2])
5   return fibs[n]
```

Run this in Python! Let's examine what's going on

- Line 1 is similar to the previous example.
- In line 2, we define a new list and initialize the first two numbers to be 0 and 1.
- In lines 3 and 4 we set up a loop. Our loop here runs for every number `i` in `range(n-1)`. This range consists of all numbers from 0 up to *but not including* `n-1` (so we stop at `n-2`). Of course if `n` is 0 or 1, the loop is skipped since there are no valid numbers in this range.
- Line 4 says to append at the end of the list the value `fibs[-1] + fibs[-2]`. We can index lists in Python by negative numbers where `-1` is the last element and `-2` is the second last element. This corresponds to the $n-1$ and $n-2$ in the recursive definition above.
- Line 5 we return `fibs[n]` which is either one of the first two values if $n \leq 1$ or the very last element if $n \geq 2$.

This definitely works better. If you now ask for `print(fib2(100))` you will be able to do it very quickly. However, for very large numbers we still have some issues. Not to mention we have to store a lot of information which is mostly unnecessary...

1.4 Third Attempt - Using Less Space

We can make a quick improvement by noting we only need to save the previous two values.

```
1 def fib3(n):
2   cur, prev = 1, 0
3   for i in range(n):
4     cur, prev = cur + prev, cur
5   return prev
```

Give this a go in Python! Let's examine what's going on

- Line 1 is similar to the previous example.
- In line 2, we set two variables `cur` and `prev` to be 1 and 0 respectively. These will (eventually) correspond to the current Fibonacci Number we are considering and the previous one.
- We set up our loop on line 3 this time to go up to `range(n)`
- ...and on line 4, we change `cur` to be `cur + prev` and `prev` to be `cur`. This moves the numbers along the sequence of Fibonacci numbers. We make use of the fact that `prev` begins at 0 so after one iteration, we get that `cur` is 1 and `prev` is also 1.
- We return the value in `prev` which stores the final answer.

This definitely works better storage wise since we only need to store two numbers. However it doesn't make a huge difference in terms of how much time we need to compute the answer.

1.5 Fourth Attempt - Fractions!

Let's play around with the Fibonacci numbers a bit. Suppose we let $a_n = f_n/f_{n-1}$ for each $n \geq 1$. Then notice that

$$1 + 1/a_n = 1 + f_{n-1}/f_n = (f_n + f_{n-1})/f_n = f_{n+1}/f_n = a_{n+1}$$

Thus, the numerator of these a_n give the next Fibonacci number! Let's code this

```
1 from fractions import Fraction
2 def fib4(n):
3   if n <= 1: return n
4   a = Fraction(1, 1)
5   for i in range(n - 2):
6     a = 1 + 1/a
7   return a.numerator
```

Try this out in Python! For an explanation:

- In line 1, we import a tool to help us deal with fractions better in Python. This information is stored in the `fractions` package.
- Lines 2 and 3 we've seen before
- Line 4 we start our fraction with the term $1/1 = f_2/f_1$.
- Then on lines 5 and 6 for each number from 0 to $n-3$ inclusive, we compute the next term in the a_n sequence, namely that $a_{n+1} = 1 + 1/a_n$. Since we only need the previous term to get the next term, we reuse the variable `a` to store this new value.
- Lastly, once the loop ends, we execute line 7 which returns the numerator of this recursive sequence a_n.

Neat but this still struggles on large values. (But see Binet's Formula for a neat alternative for using this idea to get a closed form formula!)

1.6 Fifth Attempt - Wizardry!

The Fibonacci numbers actually have a surprising amount of hidden identities - some of which you'll discover in the exercises below. Let's examine two now that will help us compute these values!

Exercise: What is the value of f_7? What is the value of $f_3^2 + f_4^2$? (Note this is the same as $(f_3)^2 + (f_4)^2$).

Solution

They both equal 13.

Exercise: What is the value of f_9? What is the value of $f_4^2 + f_5^2$?

Solution

They both equal 34.

Exercise: Can you extrapolate the above? Suppose that $n = 2k + 1$, that is, suppose n is odd. What does f_{2k+1} equal to in terms of smaller values (that depend on k)?

Solution

$f_{2k+1} = f_k^2 + f_{k+1}^2$

Exercise: What is the value of f_6? What is the value of $f_3(2f_4 - f_3)$?

Solution

They both equal 8.

Exercise: What is the value of f_8? What is the value of $f_4(2f_5 - f_4)$?

Solution

They both equal 21.

Exercise: Can you extrapolate the above? Suppose that $n = 2k$, that is, suppose n is even. What does f_{2k} equal to in terms of smaller values (that depend on k)?

Solution

$f_{2k} = f_k(2f_{k+1} - f_k)$

The summary of the above is the following two identities:

- If n is odd, then $f_n = f_{(n-1)/2}^2 + f_{((n-1)/2)+1}^2$.
- If n is even, then $f_n = f_{n/2}(2f_{(n/2)+1} - f_{n/2})$

we can use these to generate the Fibonacci numbers very quickly!

```
1 def fib5(n):
2   def pairs_fibs(n):
3     if n == 0: return [0, 1]
4     fnhalf, fnhalfplus1 = pairs_fibs(n // 2)
5     fn, fnplus1 = fnhalf*(2*fnhalfplus1 - fnhalf), fnhalf**2
        + fnhalfplus1**2
6     if n % 2 == 0: return [fn, fnplus1]
7     return [fnplus1, fn + fnplus1]
8   return pairs_fibs(n)[0]
```

Definitely try this out in Python! An explanation of the code:

- Line 1 is as before
- On line 2 we define what is called a *local helper function* which just means it is another function we need to get our result. This function returns the pair of Fibonacci numbers f_n and f_{n+1} which we can use to get subsequent values.
- Line 3 is our base case; The starting values we need to make this work.
- Line 4 sets the variables `fnhalf` and `fnhalfplusone` to be $f_{\lfloor n/2 \rfloor}$ and $f_{\lfloor n/2 \rfloor+1}$ respectively where $\lfloor n/2 \rfloor$ means we take n divide by 2 and round down.
- Line 5 computes the values using the identities.
- Line 6 states that if n is even (that is, when the remainder when I divide `n` by 2 is 0, we just return the values computed on line 5.
- Line 7 only runs when n is odd. but when n is odd we need f_{n+1} and f_{n+2} so we compute that using the values of f_n and f_{n+1} and the recurrence relationship.
- Line 8 calls the local helper function and returns the first element (which is f_n).

Notice that above, the code can very quickly compute `print(fib5(1000000))`!!!

1.7 Other Methods

There are actually lots of other different ways to compute the Fibonacci numbers - one can use matrices, binary expansions, Pascal's triangle, Analysis (Binet's Formula) and others. We'll look at some of these in the exercises. The techniques used here can also be applied to other recurrence sequences which we'll also see in the exercises.

In these two lessons we saw two extremely important sequences of numbers, namely the prime numbers (and fun subsequences of the primes) and the Fibonacci Numbers. It can be a lot of fun to check out other OEIS sequences and see what other interesting sequences you can find!

2 Exercises

2.1 Fibonacci Questions

(i) What follows are two lists of identities. By trying some numbers for n see if you can match the values on the left hand side with the values on the right hand side. Note that values on the left hand side might match with multiple values on the right hand side.

1. f_{2n}
2. f_{3n}
3. f_{n+1}^2
4. $(-1)^n$

a) $f_{n-1}f_{n+1} - f_n^2$

b) $f_{n+1}^2 - f_{n-1}^2$

c) $f_{n+1}^3 + f_n^3 - f_{n-1}^3$

d) $f_n(f_{n+1} + f_{n-1})$

e) $4f_nf_{n-1} + f_{n-2}^2$

f) $f_n(f_n + 2f_{n-1})$

Solution

1. $f_{2n} = f_n(f_n + 2f_{n-1}) = f_n(f_{n+1} + f_{n-1}) = f_{n+1}^2 - f_{n-1}^2$
2. $f_{3n} = f_{n+1}^3 + f_n^3 - f_{n-1}^3$
3. $f_{n+1}^2 = 4f_n f_{n-1} + f_{n-2}^2$
4. $(-1)^n = f_{n-1}f_{n+1} - f_n^2$

(ii) In this question we explore some other ways to compute Fibonacci Numbers.

1. Here, we compute Fibonacci numbers using matrices. A matrix for us will be a two by two array of numbers of the form $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ where a, b, c, d are all integers. We can multiply two matrices in the following (albeit strange) way:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ f & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Perform the multiplication $\begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. Can you explain why this equals $\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$?

Solution

$$\begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_n \cdot 1 + f_{n-1} \cdot 1 & f_n \cdot 1 + f_{n-1} \cdot 0 \\ f_{n-1} \cdot 1 + f_{n-2} \cdot 1 & f_{n-1} \cdot 1 + f_{n-2} \cdot 0 \end{bmatrix} = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

Warning - the next explanation is a bit challenging feel free to skip to the next warning comment!

Now we can reason using mathematical induction which we'll explain informally. First, we have that

$$\begin{bmatrix} f_{1+1} & f_1 \\ f_1 & f_{1-1} \end{bmatrix} = \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Which gives us that

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

when n is 1. Now, when n is 2, we see that

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Then, since above we have shown that our **special identity**:

$$\begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

We can substitute $n = 2$ into this equation to get that

$$\begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_3 & f_2 \\ f_2 & f_1 \end{bmatrix}$$

and so

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f_3 & f_2 \\ f_2 & f_1 \end{bmatrix}$$

We can actually keep this pattern going! If we assume that

$$\begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

Then using the **special identity** replacing n, we can see that:

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} = \begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Thus, by Mathematical Induction, we have for all values of n greater than or equal to 1, we have

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

End of Warning

The math above might have been tricky to follow but computationally, all we care about is that smart people way before us have determined that

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

and we can write a program using it!

```
1  def mul( M, N ):
2    a, b, c, d = M
3    e, f, g, h = N
4    return [a*e+b*g, a*f+b*h, c*e+d*g, c*f+d*h]
5  def fib6(n):
6    M = [1, 1, 1, 0]
7    A = [1, 0, 0, 1]
8    for i in range( n ):
9      A = mul( M, A )
10   return A[1]
```

Note above that any matrix multiplied by $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ gives the original matrix back.

2. Let's do one better than the above. To compute large powers of numbers, there's an algorithm known as the square and multiply algorithm. It works based on the following:

$$a^n = \begin{cases} a(a^2)^{(n-1)/2} & \text{if } n \text{ is odd} \\ (a^2)^{n/2} & \text{if } n \text{ is even} \end{cases}$$

Exercise: Use the above to compute 3^7 without a calculator!

Solution

$$3^7 = 3 \cdot (3^2)^3 = 3 \cdot 9^3 = 3 \cdot 9 \cdot (9^2)61 = 27 \cdot 81 = 2187$$

Exercise (Challenging!) Use the above to write code that computes the nth Fibonacci Number.

Solution

The above was stated for integers but it easily extends to matrices to give the following code:

```
1.  def mul( M, N ):
2.    a, b, c, d = M
3.    e, f, g, h = N
4.    return [a*e+b*g, a*f+b*h, c*e+d*g, c*f+d*h]
5.  def fib7(n):
6.    M = [1, 1, 1, 0]
7.    A = [1, 0, 0, 1]
8.    while n > 0:
9.      if (n % 2) == 1:  A = mul( M, A )
10.     M = mul( M, M )
11.     n = n // 2
12.   return A[1]
```

Some notes for the above:

- The `while` loop on lines 8-11 continues until `n` is eventually 0. On each pass, we perform the square multiply algorithm, namely multiplying `A` by an extra copy of $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ whenever `n % 2 == 1` (which is just another way to write when n is odd).
- On each pass we square M as well.
- Note that before `n` reaches 0, we must execute line 9 so in the end `A` stores our final answer.
- Line 10 divides `n` by 2 and rounds down (so we do eventually reach 0 as long as `n` started out as positive).
- The last line returns `A[1]` which is where f_n is stored.

This code is incredibly fast and can easily compute the millionth Fibonacci number!

2.2 Lucas Numbers

The Lucas Numbers (see `https://oeis.org/A000032`) is a related recurrence sequence to the Fibonacci Numbers. The recurrence relationship is the same however we begin with $L_0 = 2$ and $L_1 = 1$ and then $L_n = L_{n-1} + L_{n-2}$ for all $n \geq 2$.

(i) Write the first 10 terms in the Lucas Number Sequence

Solution

$$2, 1, 3, 4, 7, 11, 18, 29, 47, 76,$$

(ii) Recalling that we use f_n to denote the nth Fibonacci Number, match the identities on the left with the terms on the right

1. $4(-1)^n$
2. f_n
3. f_{n+1}
4. f_{2n}
5. f_n^2

a) $f_n L_n$

b) $\frac{1}{5}(L_{2n} - 2(-1)^n)$

c) $\frac{1}{2}(f_n + L_n)$

d) $\frac{1}{5}(L_{n-1} + L_{n+1})$

e) $L_n^2 - 5f_n^2$

Solution

- $4(-1)^n = L_n^2 - 5f_n^2$
- $f_n = \frac{1}{5}(L_{n-1} + L_{n+1})$
- $f_{n+1} = \frac{1}{2}(f_n + L_n)$
- $f_{2n} = f_n L_n$
- $f_n^2 = \frac{1}{5}(L_{2n} - 2(-1)^n)$

(iii) Using the above ideas (like was done with the Fibonacci Numbers), write code that computes the nth Lucas number. How fast can you make your code?

Solution

I'll do the fast matrix solution just as an example. You can verify for yourself that

$$\begin{bmatrix} L_{n+1} & L_n \\ L_n & L_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}$$

(basically the matrix power is the recurrence relationship and the extra matrix is what L_1, L_0, L_0, L_{-1} need to be in order for the initial set up to work!)

This gives the following code:

```
1.  def mul( M, N ):
2.    a, b, c, d = M
3.    e, f, g, h = N
4.    return [a*e+b*g, a*f+b*h, c*e+d*g, c*f+d*h]
5.  def fib7(n):
6.    M = [1, 1, 1, 0]
7.    A = [1, 0, 0, 1]
8.    while n > 0:
9.      if (n % 2) == 1:  A = mul( M, A )
10.     M = mul( M, M )
```

```
11.     n = n // 2
12.   return mul(A, [1, 2, 2, -1]) [1]
```

2.3 For the Live Session

One of the things we will do if we have time in the Live session is to share your favourite sequence. Find a sequence on OEIS, figure out how to compute the sequence, whether or not your sequence is recursive or not and be prepared to share one or two interesting facts about the sequence! I'll share some examples below but feel free to go off the list and find your own interesting sequence!

- Euler's Totient Function (`https://oeis.org/A000010`)
- Tribonacci Numbers (`https://oeis.org/A000073`)
- Catalan Numbers (`https://oeis.org/A000108`)
- Triangular Numbers (`https://oeis.org/A000217`)
- Fermat Numbers (`https://oeis.org/A0001215`)
- Abundant Numbers (`https://oeis.org/A005101`)
- Look and Say Sequence (`https://oeis.org/A005150`)
- Recamán's Sequence (`https://oeis.org/A005132`)
- Thue-Morse Sequence (`https://oeis.org/A010060`)