ProjectEuler Archive 2: Recurrence Relations

YarrowForAlgernon

August 2025

Contents

1 Challenge Description

Challenge Name: Even Fibonacci Numbers

Challenge Description:

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . . By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

2 Naive Solution

2.1 Linear Recurrence Relation

The Fibonacci sequences' calculation is famous; any given term in the sequence is the sum of the last two terms. In other words:

Theorem 2.1 (The Fibonacci Sequence).

$$F_n = F_{n-1} + F_{n-2} \tag{2.1}$$

Where $n >= 2$, $F_0 = 0$ and $F_1 = 1$.

Theorem 2.1 is a linear recurrence relation, which is a way to express the nth term in a sequence as a first order polynomial (highest exponent is 1) equation of its previous terms.

To get the sum of every even term in the Fibonacci sequence under 4,000,000 we calculate every term in the sequence using the recurrence relation described in Theorem 2.1 and test if it is even. If it is even, then we can add it to the sum of all previous even terms and repeat this sequence until we reach the biggest even term ¡= 4,000,000.

2.2 The Algorithm

```
import time

def getNextFibTerm(previousTerm: int, currentTerm: int) -> int:
    '''calculates the next term in the fibonacci sequence'''
    nextTerm = previousTerm + currentTerm
    return nextTerm

def checkIsEven(Term: int) -> bool:
    '''returns true if even, otherwise false'''
    return Term % 2 == 0

def sumTerms():
    '''gets the sum of all even fib terms under 4 million'''
    currentTerm = 1 # F1
    previousTerm = 0 # F0
    evenSum = 0
    while currentTerm < 4000000:
        nextTerm = getNextFibTerm(previousTerm, currentTerm)
        if checkIsEven(nextTerm):
            evenSum += nextTerm
        previousTerm = currentTerm
        currentTerm = nextTerm
    return evenSum


if __name__ == "__main__":
    startTime = time.time_ns()
    evenSum = sumTerms()
    stopTime = time.time_ns()
    print(evenSum)
    print("Time taken for algorithim to execute in nano seconds: "
    + str(stopTime - startTime))
```

The time complexity of this algorithm is O(n), since the number of calculations we perform grows proportionally to the input size. The space complexity is also O(1) since we only need to store at most 3 numbers (F_n, F_{n-1} and F_{n-2}).

3 Properties Of Numbers And The Structure Of The Fibonacci Sequence

3.1 Introduction

This section has a lot of Maths and proofs, many of which are very simple but stated anyway for completeness's sake. These proofs don't need to be understood to understand the rest of the section and can be skipped. The end of a proof is marked by ■.

3.2 The Even Third Term

We will prove using two propositions that every third term in the Fibonacci sequence is an even number (formalised in Proposition 3.3).

Proposition 3.1. If two odd numbers or two even numbers are added or subtracted, it will result in an even number.

Proof. Proof by cases:

The sum of two even numbers is even:

$$2k + 2n = 2(k + n)$$

The difference between two even numbers is even:

$$2k - 2n = 2(k - n)$$

The sum of two odd numbers is even:

$$(2k + 1) + (2n + 1) = 2(k + n + 1)$$

The difference between two odd numbers is even:

$$(2k + 1) - (2n + 1) = 2(k - n)$$

■

Proposition 3.2. If any combination of an even and odd number are added or subtracted from each other, the result is an odd number.

Proof. Proof by cases:

The sum of an even and odd number is odd:

$$2k + (2n + 1) = 2(k + n) + 1$$

The difference between any combination of an even number and an odd number is odd:

$$\begin{aligned} 2k - (2n + 1) &= 2(k - n) - 1 \\ (2n + 1) - 2k &= 2(n - k) + 1 \end{aligned}$$

■

Proposition 3.3. Every term in the Fibonacci sequence F_n whose index n is a multiple of 3 is an even number, with the Fibonacci sequence starting from $F_0 = 0$ and $F_1 = 1$.

Proof. By induction:

Base case when n = 3:

$$\begin{aligned} &F_n = F_{n-1} + F_{n-2} \\ &F_3 = F_2 + F_1 \\ &2 = 1 + 1 \\ &2 = 2 \end{aligned}$$

Inductive hypothesis, n = 3(k) and assuming F_{3k} is even:

$$F_{3k} = F_{3k-1} + F_{3k-2}$$

Inductive step, n = 3(k+1):

$$\begin{aligned} &F_{3(k+1)} = F_{3(k+1)-1} + F_{3(k+1)-2} \\ &F_{3k+3} = F_{3k+2} + F_{3k+1} \end{aligned}$$

Now we can rewrite F_{3k+2} as a sum of its previous terms and substitute it back into the equivalence for F_{3k+3}:

$$\begin{aligned} F_{3k+2} &= F_{3k+1} + F_{3k} \\ F_{3k+3} &= (F_{3k+1} + F_{3k}) + F_{3k+1} \\ F_{3k+3} &= 2(F_{3k+1}) + F_{3k} \end{aligned}$$

Since we assumed F_{3k} is even, according to Proposition 3.1, F_{3k+3} must be an even number. ■

Because every third term in the sequence is even, all other terms are odd, due to Proposition 3.1 and Proposition 3.2. This comes out to be a very predictable sequence, an even term followed by two odd terms.

So, we could avoid our evenness check and only add every 3rd term in the sequence, which would reduce the number of calculations we would need to make. However, we would still need to compute every Fibonacci term between even terms. Is there a way to avoid this?

3.3 A New Linear Recurrence Relation

My initial thought was to use Binet's formula, a formula that calculates the nth Fibonacci term without having to calculate any of its previous terms. However in practice not only is this not very time efficient but it is also inaccurate for large Fibonacci terms due to difficulties in floating point precision. While this accuracy error doesn't apply for our challenge, I chose to avoid the formula since it doesn't see much practical use for these reasons, despite being a very useful formula.

Instead, we can construct a new recurrence relation by creating a subsequence (a sequence created from another sequence by *only* removing some of its terms) of exclusively even terms from our original Fibonacci sequence. Since a linear recurrence relation is defined as an expression of its previous terms, let us try to express it that way.

Fibonacci sequence F_n:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Even subsequence E_n

0, 2, 8, 34, 144, ...

Since subsequence E_n is made up of terms only from the Fibonacci sequence, we can construct a new linear recurrence relation from the terms of the original sequence.

In order to build this recurrence relation, we must express each term in E_n in terms of F_n. This is quite easy, since the subsequence is just every third term (even number) in the Fibonacci sequence, we can express F_n as a linear recurrence relation given each term has its index n as a multiple of 3.

Starting with Theorem 2.1:

Assuming n is a multiple of 3:

$F_n = F_{n-1} + F_{n-2}$

Lets take F_n and try to express it in terms of its previous even numbered terms F_{n-3} and F_{n-6}

$F_{n-1} = F_{n-2} + F_{n-3}$

$F_n = 2(F_{n-2}) + F_{n-3}$

$F_{n-2} = F_{n-3} + F_{n-4}$

$F_n = 3(F_{n-3}) + 2(F_{n-4})$

$F_{n-4} = F_{n-5} + F_{n-6}$

$F_n = 3(F_{n-3}) + F_{n-4} + F_{n-5} + F_{n-6}$

$F_n = 4(F_{n-3}) + F_{n-6}$

This gives us the linear recurrence relation for every third term in the Fibonacci sequence:

Theorem 3.1 (Even Subsequence Linear Recurrence Relation).

$$E_n = 4(E_{n-1}) + E_{n-2} \tag{3.1}$$

Where $n >= 2$, $E_0 = 0$ and $E_1 = 2$.

We can now use Theorem 3.1 to skip every odd numbered term in the sequence.

3.4 The Algorithm

```
def computeNextEvenTerm(previous3rdTerm: int, currentTerm: int) -> int:
    nextTerm = previous3rdTerm + (4 * currentTerm)
    return nextTerm

def sumEvenTerms():
    currentTerm = 8
    previousTerm = 2
    sum = 2
    while currentTerm < 4000000:
        sum += currentTerm
        nextTerm = computeNextEvenTerm(previousTerm, currentTerm)
        previousTerm = currentTerm
        currentTerm = nextTerm
    return sum

if __name__ == "__main__":
    evenSum = sumEvenTerms()
    print(evenSum)
```

While the time complexity is still O(n), we only need to calculate 1/3 of the sequence we would in Section 2.2, a very marginal improvement. The space complexity is still O(1) since the number of integers we store doesn't change with the size of our input.

4 Non-Lucas Fast Doubling

4.1 Preamble

The algorithm that will be described in this section is actually practically slower for computing small nth terms of the Fibonacci sequence which is what is needed for our challenge. However, there is a much faster way to compute the nth term when n is large, as the solution in Section 3.4 grows quickly with n.

This algorithm is computed without using the Lucas sequence, which is a sequence very similar to the Fibonacci sequence and has some very useful identities with it. Using the Lucas sequence identities with the following algorithm should be faster than the method described in this section, as shown in [1].

4.2 Summations And Fast Doubling

There is a way to calculate the sum of all previous Fibonacci terms using the following formula:

Theorem 4.1.

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1 \tag{4.1}$$

Proof. By induction:

Base case n = 3:

$$\sum_{i=1}^{n=3} F_i = F_5 - 1$$
$$0 + 1 + 1 + 2 = 5 - 1$$
$$4 = 4$$

Inductive hypothesis When n = k:

$$\sum_{i=1}^{k} F_i = F_{k+2} - 1$$

Inductive step - when n = K+1:

$$\sum_{i=1}^{k+1} F_i = F_{(k+1)+2} - 1$$
$$F_0 + F_1 + ... + F_k + F_{k+1} = F_{(k+1)+2} - 1$$
$$F_0 + F_1 + ... + F_k = F_{k+2} - 1$$
$$F_{k+2} - 1 + F_{k+1} = F_{k+3} - 1$$
$$F_{k+3} = F_{k+3}$$

∎

This formula works for the normal Fibonacci sequence, but will not for the subsequence of only even Fibonacci terms. There is a very simple way to fit this formula to our sequence. Since the even terms are the result of summing the previous two odd terms, this means that combined those two previous terms share the same value as the even term. So for every even term we sum in our sequence, two odd terms of the same combined value have been excluded.

This gives us a new equation:

Corollary 4.1.1 (Summation Of Only Even Fibonacci Terms).

$$\sum_{i=1}^{n} F_i = \frac{F_{n+2} - 1}{2} \tag{4.2}$$

Where n is a multiple of 3. The formula will not work if n isn't a multiple of 3 as therefore odd terms are included in the calculations.

4.3 Matrix Exponentiation

The fastest way to get to n, such that n is the largest even Fibonacci term below 4,000,000 (and the summation can be quickly computed as defined in Corollary 4.1.1) is best explained as an evolution from other fast algorithms. A very fast algorithm for calculating the nth Fibonacci term is matrix exponentiation. This takes a specific 2x2 matrix and raises it to the nth power to produce the following equivalence and allow us to calculate F_n:

Theorem 4.2 (Matrix Exponentiation).

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \tag{4.3}$$

This is done using binary exponentiation (a fast way to calculate large powers), otherwise using the recurrence relation defined in Theorem 3.1 is faster.

We calculate the nth term in the Fibonacci sequence, along with its adjacent terms just by looking at specific cells on the matrix they correspond to. While fast, there are some redundant calculations, such as calculating F_n twice, and F_{n-1} which we don't need. In order to stop doing this, we can formulate some general identities in order to streamline the calculations. We can do this by changing n to $2n$ in our matrix equations:

Lemma 4.3 (Derivation Of Identities).

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} = \begin{bmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{bmatrix} \tag{4.4}$$

Since the left hand side of Theorem 4.2 is the exact same as the right hand side, we can apply the squaring to the right hand to get the following equation:

$$\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^{2n} = \begin{bmatrix}F_{n+1} & F_n\\F_n & F_{n-1}\end{bmatrix}^2 \tag{4.5}$$

Now we analyse both Matrices with Fibonacci terms:

$$\begin{bmatrix}F_{2n+1} & F_{2n}\\F_{2n} & F_{2n-1}\end{bmatrix} = \begin{bmatrix}F_{n+1} & F_n\\F_n & F_{n-1}\end{bmatrix}^2 \tag{4.6}$$

Expanding the right Hand side of the equation above:

$$\begin{bmatrix}F_{2n+1} & F_{2n}\\F_{2n} & F_{2n-1}\end{bmatrix} = \begin{bmatrix}F_{n+1}^2 + F_n^2 & F_nF_{n+1} + F_nF_{n-1}\\F_nF_{n+1} + F_nF_{n-1} & F_n^2 + F_{n-1}^2\end{bmatrix} \tag{4.7}$$

Now we can extract the identities that are useful for us directly from the matrix:

$$F_{2n+1} = (F_{n+1})^2 + (F_n)^2 \tag{4.8}$$

$$F_{2n} = F_nF_{n+1} + F_nF_{n-1} \tag{4.9}$$

We can simplify F_{2n} further:

$F_{2n} = F_n(F_{n+1} + F_{n-1})$

We can express F_{n-1} in terms of F_{n+1} and F_n by doing the following:

$F_{n+1} = F_n + F_{n-1}$

$F_{n-1} = F_{n+1} - F_n$

Now we can substitute this back into our original equation to get:

$F_{2n} = F_n(F_{n+1} + (F_{n+1} - F_n))$

$F_{2n} = F_n(2(F_{n+1}) - F_n)$

This gives us the following identities:

Theorem 4.4 (Fast Doubling Identities).

$$F_{2n} = F_n(2(F_{n+1}) - F_n) \tag{4.10}$$

$$F_{2n+1} = (F_{n+1})^2 + (F_n)^2 \tag{4.11}$$

Using these identities we can calculate the nth Fibonacci term in O(log(n)) time.

4.4 The Application Of Fast Doubling Identities In Base 2

The actual algorithm is slightly more complex.

We first convert the nth term into into base 2, and using this representation decide how to apply our identities. We store and keep track of four numbers, F_n, F_{n+1} and either F_{2n} and F_{2n+1} or F_{2n+1} and F_{2n+2}.

We store the binary form of n as a boolean array, where True corresponds to 1 and False corresponds to 0. The first bit we process is the most significant bit of the number.

If our binary digit is a 1, we set the following values:

$$F_n \to F_{2n+1} \tag{4.12}$$
$$F_{n+1} \to F_{2n+2} \tag{4.13}$$

Remember that F_{2n+2} is just $F_{2n+1} + F_{2n}$, as it is just the next term in the Fibonacci sequence.

And If the binary digit is 0:

$$F_n \to F_{2n} \tag{4.14}$$
$$F_{n+1} \to F_{2n+1} \tag{4.15}$$

The reason for this conditional logic is to calculate terms that would otherwise be unreachable by just one set of identities. For example, without processing a 1 and performing the following assignments, the Fibonacci sequence would not move beyond F_0 and F_1.

4.5 The Algorithm

```
from time import time_ns

def convertIntegerToBase2(Integer: int) -> list:
    '''Calculates the binary form of an unsigned integer
    as a list of boolean values.
    '''
    binaryForm = []
    while Integer > 0:
        if Integer % 2 != 0:
            binaryForm.append(True)
        else:
            binaryForm.append(False)
        Integer //= 2
    return binaryForm


def fastDoubling(n: int) -> int:
    '''Returns the nth term in the fibonacci sequence starting from 0.
    O(log2(n)).'''
    a = 0 #F0
    b = 1 #F1
    binaryForm = convertIntegerToBase2(n)
    while binaryForm:
        mostSignificantBit = binaryForm[-1]
        c = a * ((2 * b) - a) #Identity F2n
        d = (a**2) + (b**2) #Identity F2n+1
        if not mostSignificantBit: # if bit is 0
            a, b = c, d
        else:
            a, b = d, c + d
        binaryForm.pop(-1)
    return a


def main(n: int = 33) -> float:
    '''Returns the sum of all even Fibonacci terms under the nth term.'''
    n = setNthTermToEvenNumber(n)
    evenSummation = fastDoubling(n+2) - 1
    evenSummation //= 2
    return evenSummation


def setNthTermToEvenNumber(n: int) -> int:
    '''Sets the value of n to a multiple of 3 to ensure it is
    an even number.'''
    return (n - (n % 3))
```

```

if __name__ == "__main__":
    startTime = time_ns()
    evenSum = main()
    stopTime = time_ns()
    print(evenSum, "\nNumber of digits: " + str(len(str(evenSum))),
    "\nTime (ms): " + str((stopTime - startTime) / 1000000))
```

The time complexity of this algorithm is O(log(n)).

5 Bibliography

References

[1] D. Takahashi, "A fast algorithm for computing large fibonacci numbers," *Information Processing Letters*, vol. 75, no. 6, pp. 243–246, Nov. 2000. DOI: 10.1016/s0020-0190(00)00112-5.