ProjectEuler Archive 2: Recurrence relations

YarrowForAlgernon

2 August 2025

Contents

1 Challenge description and introduction

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 sequence is famous for its calculation; any given term in the sequence is the sum of the last two terms. In other words:

Theorem 2.1.

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

Where $n >= 2$

This 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 can 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.

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
    previousTerm = 0
    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.

3 Properties of numbers and the structure of the Fibonacci sequence

3.1 The Even Third Term

We will prove that every term in the Fibonacci sequence whose index is $n >= 2$ and a multiple of 3 is an even number, with the Fibonacci sequence starting from $F_0 = 0$ and $F_1 = 1$

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

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

In accordance with Proposition 3.1, any even term in the Fibonacci sequence must be preceded by either two odd or even terms, in order for their addition to result in an even number. Since the sequence begins with '0, 1, 1, 2, ...' this means there will never be two consecutive even terms in the sequence. We can logically prove this as the next term after the even term in the sequence will be the sum of an odd and even number. According to Proposition 3.2 this will result in an odd number. Since there are no consecutive even numbers, every even number must be preceded by two odd terms and due to Theorem 2.1 th even term will be followed by two odd terms. Thus, every third term in the sequence. e,g F_3, F_6, F_9, etc will be even numbers.

This comes out to be a very predictable sequence, an even term followed by two odd terms repeated throughout the sequence:

```
Even, odd, odd, even, odd, odd, even, ...

The Fibonacci sequence follows an identical structure:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

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?

3.2 A new linear recurrence relation

My initial thought was to use Binet's formula, a formula to compute the nth Fibonacci term without having to compute 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 this reason. 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.

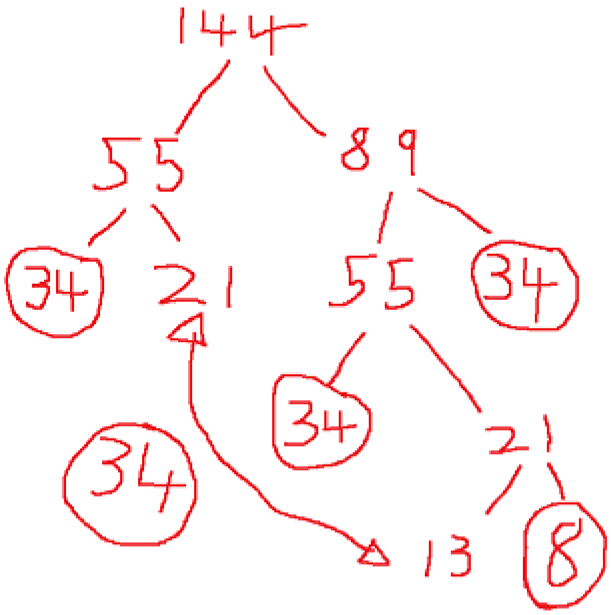
Fibonacci sequence F_n:

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

Even subsequence E_n

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

We can see that each previous even subsequence term E_{n-1} corresponds to F_{n-3} in the Fibonacci sequence. Lets try to express our even subsequence in terms of F_{n-3} using a binary tree (similar to prime decomposition), with all leaf nodes either being expressed as previous terms or their sum resulting in a previous term in the subsequence E_n. For example, 144:



In terms of the Fibonacci sequence F_n we can express this recurrence relation like this:

$$144 = 4(34) + 8 \tag{3.1}$$
$$F_n = 4(F_{n-3}) + F_{n-6} \tag{3.2}$$

Given n is a multiple of 3 (an even term) and n ¿= 6.
This translates to our even subsequence like this:

$$E_n = 4(E_{n-1}) + E_{n-2} \tag{3.3}$$

We can prove this relation when $n >= 2$ through induction:
base case when n = 2.

$$8 = 4(E_{n-1}) + E_{n-2} \tag{3.4}$$
$$8 = 4(2) + 0 \tag{3.5}$$
$$8 = 8 \tag{3.6}$$

Thus, let $k >= 2$ when n = k (n is an arbitrary integer, and thus for all possible values of n), considering n = K+1, show that the relation holds for all subsequent terms. In other words, that $E_n => E_{n+1}$:

$$E_k = 4(E_{k-1}) + E_{k-2} \tag{3.7}$$
$$E_{k+1} = 4(E_{(k+1)-1}) + E_{(k+1)-2} \tag{3.8}$$
$$E_{k+1} = 4(E_k) + E_{k-1} \tag{3.9}$$

3.3 The algorithm

```python
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.

4 Non-Lucas Fast Doubling

4.1 Preamble

The algorithm that will be described in this section is actually 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 3.3 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, as described in [**takahashi**].

4.2 Summations and fast doubling

There is a way to calculate the sum of all previous fib terms as a formula in the form of Fibonacci terms:

Theorem 4.1.

$$\sum_{k=1}^{n} F_k = F_{n+2} - 1 \tag{4.1}$$

[proof inductive needed]

This formula works for the normal Fibonacci sequence, but will not for our even termed Fibonacci sequence. 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 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.

$$3\sum_{k=1}^{k} F_k = \frac{F_{n+2} - 1}{2} \tag{4.2}$$

Where k is the quotient of n (the original upper bound) divided by 3, which subsequently leads to F_n being an even number.

4.3 Matrix Exponentiation

The fastest way to get to n, such that n is the largest even Fibonacci term below 4,000,000 is best explained as an evolution from other fast algorithms. A very fast algorithm for calculating the nth Fibonacci term is matrix exponentiation. By raising the left hand Matrix to the nth power we can get the following equivalence:

Theorem 4.2.

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

To the nth power (using binary exponentiation, otherwise using the recurrence relation is faster), we can 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:

Corollary 4.2.1.

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

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

Thus:

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

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

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

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

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

We can simplify F_{2n} further:

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

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

This simplification is due to the identity:

$$F_{n+1} = F_n + F_{n-1} \tag{4.12}$$

$$F_{n-1} = F_{n+1} - F_n \tag{4.13}$$

Hence, using iteration we can take the nth term, and using these identities, calculate it in O(log(n)) time.

4.4 The True Algorithm Is A Little harder

The actual algorithm is slightly more complex. Using the binary form of n, we compute both F_{2n} and F_{2n+1} and we store them along with F_n and F_{n+1}.

We compute the binary form of n as a boolean array, where true is the presence of 1 and False is the presence of 0. The first bit we process is the most significant byte of the number.

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

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

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

If it is 0:

$$F_n \to F_{2n} \tag{4.16}$$

$$F_{n+1} \to F_{2n+1} \tag{4.17}$$

The reason for this conditional 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.

4.5 The True Algorithm

```
from math import log
from time import time_ns


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


def fastDoubling(n: int) -> int:
    '''returns the nth term in the fibonacci sequence starting from 0.
    O(log2(n)).'''
    a = 0
    b = 1
    nBinaryForm = calculateBinaryForm(n)
    while nBinaryForm:
        n = nBinaryForm[0]
        c = a * ((2 * b) - a)
        d = (a**2) + (b**2)
        if not n: # if bit is 0
            a, b = c, d
        else:
            a, b = d, c + d
        nBinaryForm.pop(0)
    return a

```

```

def sumEvenTerms(n: int) -> float:
    '''Only works if the value of n is a multiple of 3 and therefore a
    positive term. A quotient division is used to save time and space
    in divisions involving large numbers.'''
    isOdd = False
    evenSum = fastDoubling(n+2) - 1
    if evenSum % 2 != 0:
        isOdd = True
    evenSum //= 2
    if isOdd:
        evenSum += 0.5
    return evenSum


def main(n: int = 10) -> int:
    '''if n isn't an even term, it will set it to the last even term,
    in order to calculate the sum of all previous even terms.'''
    if n % 3 != 0:
        n = n - (n % 3)
    finalSum = sumEvenTerms(n)
    return finalSum


if __name__ == "__main__":
    startTime = time_ns()
    evenSum = main()
    stopTime = time_ns()
    print(evenSum, "\nTime (ns): " + str(stopTime - startTime))
```

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