ProjectEuler: Archive 2 - Recurrence relations

YarrowForAlgernon

2 August 2025

ABSTRACT Using the linear recurrence relation of the Fibonacci sequence to calculate the sum of its even terms has a time complexity of O(n), though a new recurrence relation can be constructed from a subsequence to improve runtime marginally.

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.*

This document focuses mainly on this problem, but does contain a basic explanation of a faster algorithm for calculating Fibonacci terms in section 4 which is only practically faster for nth terms larger than what is needed to solve the challenge.

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 1.1

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

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.

2.2 Naive algorithm

To get the sum of every even term in the Fibonacci sequence under 4,000,000 we can calculate every term in the sequence and test if it is even. If it is even, then we can add it to the rolling sum of all previous even terms. Here is some python code that implements this:

```
import time

def getNextFibTerm(previousTerm: int, currentTer
    '''calculates the next term in the fibonacci
    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
    currentTerm = 1
    previousTerm = 0
    evenSum = 0
    while currentTerm < 4000000:
        nextTerm = getNextFibTerm(previousTerm,
        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
```

2.3 Complexities

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 O(n) as well, as the integers we store in memory become larger proportionally to n.

3 Efficient Solutions

3.1 Properties of numbers and the structure of the Fibonacci sequence

Property 1.1 If two odd numbers are added or subtracted, it will result in an even number.

Property 1.2 If any combination of an even and odd number is added or subtracted, the result is an odd number.

In accordance with Property 1.1, any even term in the Fibonacci sequence must be preceded by two odd terms, in order for their addition to result in an even number. Lucky for us, property 1.2 states that the next two terms after the even number must be odd. This comes out to be a very predictable sequence.

0, odd, odd, even, odd, odd, even, odd, odd, even, ...

The Fibonacci sequence follows an identical structure:

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

Every 3rd term is an even number. 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 fib 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 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, 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 of only 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.

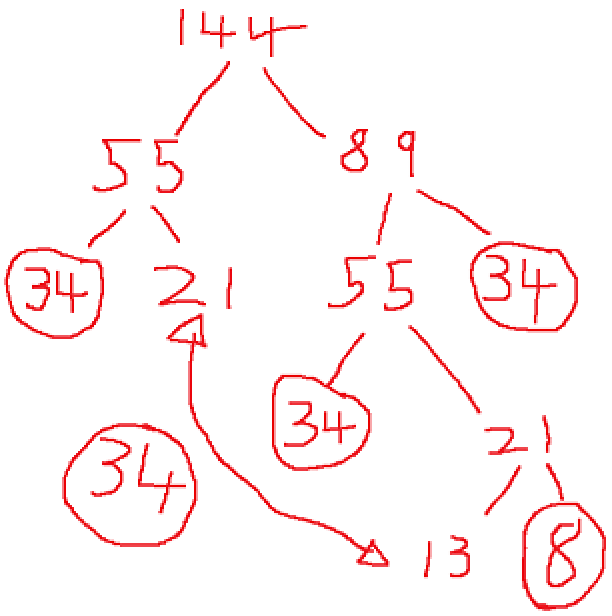
Fibonacci sequence $F()$:

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

Even subsequence $E()$

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

We can see that each previous term in $E()$, 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 $E()$. For example, 144:



In terms of the Fibonacci sequence $F()$ we can express this recurrence relation like this:

$$144 = 4(34) + 8 \tag{1}$$
$$F_n = 4(F_{n-3}) + F_{n-6} \tag{2}$$

Given n is a multiple of 3 (an even term) and n ¿= 6.

And in our even subsequence:

$$E_n = 4(E_{n-1}) + E_{n-2} \tag{3}$$

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

base case when n = 2.

$$8 = 4(E_{n-1}) + E_{n-2} \tag{4}$$
$$8 = 4(2) + 0 \tag{5}$$
$$8 = 8 \tag{6}$$

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 $E_n => E_{n+1}$:

$$E_k = 4(E_{k-1}) + E_{k-2} \tag{7}$$
$$E_{k+1} = 4(E_{(k+1)-1}) + E_{(k+1)-2} \tag{8}$$
$$E_{k+1} = 4(E_k) + E_{k-1} \tag{9}$$

3.3 The algorithm

Here is some python code that implements this, very similar to the naive solution:

```
def computeNextEvenTerm(previous3rdTerm: int, cu
    nextTerm = previous3rdTerm + (4 * currentTer
    return nextTerm

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

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

3.4 Complexity

While the time complexity is still O(n), we only need to calculate 1/3 of the sequence we would otherwise. Similarly, the space complexity is also still O(n). /par

4 Beyond the efficient solution

4.1 Preamble

This section doesn't need to be read if the above implementation is satisfactory for the challenge, and is actually faster for computing small nth terms of the Fibonacci sequence. However, there is a much faster way to compute the nth term when n is large, as the above solutions' runtime grows very quickly with n.

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 1.1

$$\sum_{k=1}^{n} F_k = F_{n+2} - 1 \tag{10}$$

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:

Theorem 1.2

$$\sum_{k=1}^{n} F_k = \frac{F_{n+2} - 1}{2} \tag{11}$$

Given k is a multiple of 3, AKA F_n is an even number.

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:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \tag{12}$$

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:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} = \begin{bmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{bmatrix} \tag{13}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} = \begin{bmatrix} F_{n+1} & F_{n} \\ F_{n} & F_{n-1} \end{bmatrix}^2 \tag{14}$$

Thus:

$$\begin{bmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_{n} \\ F_{n} & F_{n-1} \end{bmatrix}^2 \tag{15}$$

$$\begin{bmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1}^2 + F_n^2 & F_nF_{n+1} + F_nF_{n-1} \\ F_nF_{n+1} + F_nF_{n-1} & F_n^2 + F_{n-1}^2 \end{bmatrix} \tag{16}$$

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

$$F_{2n+1} = (F_{n+1}) + (F_n)^2 \tag{17}$$

$$F_{2n} = F_n F_{n+1} + F_n F_{n-1} \tag{18}$$

We can simplify F_{2n} further:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}) \tag{19}$$
$$F_{2n} = F_n(2F_{2n+1} - F_n) \tag{20}$$

This simplification is due to the identity:

$$F_{n+1} = F_n + F_{n-1} \tag{21}$$
$$F_{n-1} = F_{n+1} - F_n \tag{22}$$

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

4.3 The algorithm

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:

$$F_n \rightarrow F_{2n+1} \tag{23}$$
$$F_{n+1} \rightarrow F_{2n+2} \tag{24}$$

If it is 0:

$$F_n \rightarrow F_{2n} \tag{25}$$
$$F_{n+1} \rightarrow F_{2n+1} \tag{26}$$

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.

Here is the python code that implements this:

```
    from math import log
from time import time_ns


def calculateBinaryForm(number: int) -> list:
    '''Calculates the binary form of an unsigned
    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 seq
    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 multipl
    a positive term. A quotient division is usec
    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 t
    in order to calculate the sum of all previou
    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(stopTim
```

4.4 Complexities

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