Euler Two with Julia

R (Chandra) Chandrasekhar

2023-11-27 | 2023-12-02

In a recent blog I chronicled my efforts at solving the Project Euler Problem 1 using the programming language `Rust`. At the conclusion of the blog, I was less enthusiastic about `Rust` than I was at the start.

Was there another relatively new programming language that was better suited to my temperament and that held promise to become mainstream? I thought of `Julia` which has an impressive pedigree of academics from reputed institutions behind its invention.

I also happened to watch recently a talk by an academic from India, Professor Sourish Das, belonging to the reputed Chennai Mathematical Institute, in which he pitched for Julia as *the* programming language of the future for the field of Data Science. The good professor compared Julia with current mainstream languages used in Data Science, and explained with convincing facts why he was batting for `Julia` to be the leader in a few years.

Solving Project Euler Problem One

To compare apples with apples, I thought I would first try to solve Euler Problem 1 using `Julia`.

The first thing I looked for was vectorization and the use of the simple `start:step:end` syntax that was familiar to me from Octave. I searched for vectors and ranges and landed up here.

From there, it was almost trivial to solve Euler One with this one-liner in `Julia`:

```
sum(3:3:999) + sum(5:5:999) - sum(15:15:999)
```

Note that we have the same `end` value for the three vectors because of the words `below 1000` in the problem. We did not need to separately compute the number of terms in each case. And the fact that ranges in vectors are inclusive at both ends, all make for a simple and succinct solution. The answer, of course, is `233168` as before.

With Euler One out of the way, we are now free to analyze and solve Euler Two using `Julia`.

Statement of problem: Euler Two

The Euler Project Problem 2 is stated as follows:

> Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with and 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.

Parsing the problem

The problem states "By starting with and 1 and 2," indicating that the starting point is not universally accepted as 1 and 2.

Indeed, the Fibonacci sequence at OEIS starts off like this:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots \tag{1}$$

I prefer this latter sequence for aesthetic reasons and will use it in this problem.

The second important phrase asks for the sum of the "even-valued terms". The *even-numbered* terms are those occupying even *positions* in the sequence. The *even-valued* terms are those whose *values* are even. The two need not be the same, and are not the same in this case.

The third important phrase is "whose values do not exceed four million". A number less than x does not exceed x. Equally, a number that equals x also does not exceed x. So, the mathematical condition is "$\leq 4,000,000$". Moreover, the stopping condition refers to the *whole Fibonacci sequence.* Let this upper bound be called:

$$F_m = F_{\max} \leq 4,000,000. \tag{2}$$

We do not know the value of either m or $F_{\max}$ at present.

Recurrence relation for the Fibonacci sequence

The recurrence relation for the Fibonacci sequence is:

$$\begin{aligned} F_1 &= 0 \\ F_2 &= 1 \\ F_3 &= F_2 + F_1 = 1 \\ F_4 &= F_3 + F_2 = 1 + 1 = 2 \\ F_n &= F_{n-1} + F_{n-2} \text{ for } n \in \mathbb{N} \text{ and } n > 2 \end{aligned} \tag{3}$$

While there *is* an explicit formula for the n^{th} Fibonacci number, called Binet's formula, its use involves the irrational, algebraic number $\sqrt{5}$, and programs using it will suffer from rounding errors. However, this does not preclude methods based on Binet's formula, provided they are used knowledgeably.

The even-valued Fibonacci subsequence

If we look at Equation (1), we will notice that, assuming zero is even, the even terms are:

$$0, 2, 8, 34, 144, \ldots$$

and their position in the sequence is

$$1, 4, 7, 10, 13, \ldots$$

spaced at every *three* terms apart. Note that the indices of the even-valued Fibonacci subsequence actually form an arithmetic sequence with $a = 1$ and $d = 3$ and n^{th} term $3n - 3$. If we use this sequence to filter out the Fibonacci sequence and sum it, we will be done. This is one approach. We will explore other approaches later.

So, the sum we are after, assuming that Equation (3) holds, is

$$\sum_{k=1}^{m} F_{3k-2}. \tag{4}$$

where m is the index of the largest even-valued Fibonacci number that does not exceed 4,000,000. Let us call this term F_{max}. We will consider this subsequence in its own right toward the end of this blog.

Small steps toward the solution

Because the syntax of `Julia` is new to me, I will start with trivial scripts that almost single-step toward the solution.

Append the third Fibonacci number to the array

Let us concatenate the third Fibonacci number to the first two. Obviously, we need a one-dimensional array , or vector, `F` to hold the Fibonacci numbers.

The first two elements of `F` are pre-defined. So, only the third element must be defined by the recurrence relation Equation (3), and added to the *end* of the array, or *appended* to it. The code to do this is:

```
# Append the third Fibonacci number to the Fibonacci array
F = [0, 1];
push!(F, F[1] + F[2]);
println(F);
```

It works and gives us `[0, 1, 1]`. So far so good.

First twenty elements of the Fibonacci Sequence

Because we know that there are twenty elements, our task is easier and may be accomplished by a `for` loop.


```
# Generate the first twenty Fibonacci numbers
F = [0, 1];
for i in (3:20)
  push!(F, F[i-1] + F[i-2]); # append to array
end
println(F);
```

This gives the first twenty Fibonacci numbers as `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]`. Array indices in `Julia` begin with 1, and the `[` and the `]` represent the array delimiters. You can download the script here.

So, we are good. Notice that we did not even need to define `F[i] = F[i-1] + F[i-2]` but could simply invoke the RHS (right hand side) of the recurrence relation and append it. Thus far, syntax in `Julia` has tracked the mathematical expression very closely.

How to stop when we need to?

The difficulty is that while we know from Equation (2) the upper bound that should not be exceeded, we know neither the value of the largest Fibonacci number, F_{max}, at which we must stop, nor its index m. We therefore need to allocate an array whose dimensions are not known in advance. This also precludes the use of the `for` iterator for the same reason.

`Julia` also offers a `while` loop. As with `while` loops in all languages, it must be used with care, because of the following possible undesirable outcomes, if used erroneously:

a. It does not execute even once.
b. It stops at one more than the expected condition.
c. It stops at one less than the expected condition.
d. It loops infinitely.

From what I know so far, the program should preferably use a `while` loop that goes on forever and is forcibly terminated when a known stopping condition is encountered within its body.

Following this approach, here is my program to output all the Fibonacci terms which do not exceed $4,000,000$.

```
# Generate the Fibonacci numbers which do not exceed 4 million
const MAX = 4000000;
F = [0, 1, undef]
i = 3 # Array already has three elements
#
while true
  global i
  F[i] = F[i-1] + F[i-2]
  # println("$(i)" , " ", "$(F[i])") # for troubleshooting
  (F[i] + F[i-1]) <= MAX || break
  push!(F, F[i]) # append to array
```



```
  i += 1
end
#
println(i)
println(F)
```

Commentary on the program

This program was written with very little knowledge of `Julia`. *It uses the wrong approach, but gives the right answer. It should not be used as an example to write code in* `Julia`.

Nevertheless, I will go through the above program one line at a time. We first define the constant 4,000,000 using an uppercase name, as is prevalent in `C` as well. Note that we may terminate a line with a semi-colon, or leave it out, as we please.

The array `F` is of `type Any` because we have not assigned a type to it. While type assignment might matter in other situations, not assigning it now is simpler for our purposes.

We are within our rights in allocating values to the first two elements of `F` because the sequence cannot fire up otherwise.

Note that the third element of `F` is `undef` , i.e, it is left undefined. We need to *reserve a place* for the third element because will be evaluating it at the top of the `while` loop, and we could run into an `out of bounds` error otherwise.

Array indices start at `1` in `Julia` and we need `i` to be `3` at the start of the `while` loop. The value of `i` cannot, however, be passed to the `while` loop unless we declare it global. I later found out from the user-community that declaring variables `global` is a strict no-no.

We then progress to the code that actually embodies the recurrence relation in Equation (3):

```
F[i] = F[i-1] + F[i-2]
```

The line after this was used for troubleshooting and may uncommented for that purpose if desired.

The statement `(F[i] + F[i-1]) <= MAX || break` is the condition that operates the `while`. To see why it takes this form, assume that we have the second largest Fibonacci number to be 2.5 million and the next number to be 3.5 million. The next Fibonacci number will therefore be 6 million. The loop will surely stop when it encounters 6 million. But will it stop at 3.5 million which is still within the stipulated bound?

Therefore, we need to *evaluate the next Fibonacci number* and test it against the stopping condition. If it is within bounds, we append it to the array with `push!(F, F[i])` and then increment the array index `i`. Otherwise, we will abruptly `break` and exit the program.

At the end, we print out the last `i` value, which is `34` and the complete Fibonacci sequence whose values do not exceed 4,000,000. The largest permissible Fibonacci number, obeying this condition, is `35245781`. The full sequence is:

`[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578]`. It appears prefixed by the word `Any` which is an artifact of the language and should be ignored.

We may now append a second script to sum the even numbers in this sequence.

Sum of even-valued Fibonacci numbers not exceeding 4 million

We are a stone's throw away from solving the set problem. All we need do is to sum the even numbers in the array `F`. The indices of these are 1, 4, 7, etc. So we *slice* `F` as shown below to get the even sequence `E`. The sum of `E` is our desired result.

```
#
# Sum of even Fibonacci numbers in F
#
E = F[1:3:end]
println(E)
println(sum(E)) # This is what we want
```

The answer to the problem is `4613732`. For completeness the full program may be downloaded here.

Things I got wrong with the above script

Since I am just learning `Julia`, and had some doubts, I posted a question at the community forum where experienced `Julia` programmers respond to queries. Their replies were very definite on what to do and what not to do, and also revealed a diversity of approaches to solve the problem.

The experts were in one voice saying that `global` variables *should not* be used. They advised me to define a function and use it instead. Indeed, they even gave me examples of code to solve Euler Two.

I have slightly modified one of the solutions I received, and list it below. It uses functions to circumvent the need for `global` variables. Please treat my previous solution as a mistake, and consider the code fragment below as one of many proper solutions.

```
function myfib(maxval)
    val = 1
    F = [0, val] # assign values to the first two Fibonacci terms
    while val <= maxval
        push!(F, val) # the third term is also 1
        val = F[end] + F[end-1] # the recurrence relation; same as v += F[end-1]
    end
    return F # return the whole array
end

#
```



```
# Obtain the Fibonacci sequence for all values not exceeding 4 million
#
A = myfib(4000000)
println(A, " ", length(A)) # The length gives the index of the largest admissible Fibo

#
# Extract and sum the even-valued terms
#
E = A[1:3:end]
println(sum(E))
```

The sum of the even-valued terms in the Fibonacci sequence not exceeding four million is `4613732`. This file may be accessed here.

A second look at the problem

In our first pass, we stuck very closely to the Fibonacci sequence, evaluated it in full until the prescribed limit, and then obtained the even-valued sum. From a programming point 0f view, this may be deemed "naive" and "wasteful" in a way because:

1. We evaluate the *entire* sequence and throw away more than half the terms. It would be more efficient to obtain a *recurrence relation for the even-valued subsequence* and use that instead.
2. After assembling the array, we sum it to get a *single* number. We are aggregating a vector into a scalar. Would it not be better to *sum as we go along* and keep accumulating the result in a single scalar, rather than maintaining a vector that we sum at the end?

These observations are generic keys to efficient programming, and could result in more frugal programs in terms of memory and execution times. This "philosophy" will help us write better programs. The rest of this blog explores this approach.

The even-valued Fibonacci subsequence

Let us call the *even-valued subsequence of the Fibonacci sequence* the *even sequence* for short, remembering that it is not the *even terms* but the *even values* that we are after.

Two important factors will guide our efforts at getting a recurrence relation for the even sequence:

1. The original recurrence relation applies to the Fibonacci sequence.
2. The even sequence is predictably distributed in the Fibonacci sequence, with even values every three places.

This means that we should be able to get a recurrence relation by writing the relationships in terms of the even sequence alone, starting from the full Fibonacci sequence. The even-valued terms are three positions apart. So, we span seven terms from index k to index $(k+6)$ to get the recurrence relation:

$$\begin{aligned} F_{k+6} &= F_{k+5} + F_{k+4} \text{ (expand RHS solely in terms of } F_{k+3} \text{ and } F_k) \\ &= F_{k+4} + F_{k+3} + F_{k+3} + F_{k+2} \\ &= F_{k+3} + F_{k+2} + F_{k+3} + F_{k+3} + F_{k+2} \text{ (expand } F_{k+2} = F_{k+1} + F_k) \\ &= 3F_{k+3} + F_{k+2} + F_{k+1} + F_k \text{ (since } F_{k+3} = F_{k+2} + F_{k+1}) \\ &= 3F_{k+3} + F_{k+3} + F_k \\ &= 4F_{k+3} + F_k \end{aligned} \tag{5}$$

If we write Equation (5) in terms of the even sequence, E_n, noting that indices k, $(k+3)$, and $(k+6)$ represent its successive terms, which we shall call n, $(n+1)$, and $(n+2)$, we get from Equation (5),

$$E_{n+2} = 4E_{n+1} + E_n \tag{6}$$

We need to initialize $E_1 = 0$ and $E_2 = 2$. Thenceforth, we may generate the whole even sequence. Let us write this using `Julia` syntax.

Summing the even-valued sequence in `Julia`

Three variables are used in the recurrence relation. A fourth is needed for the sum. So, with four variables and some judicious code, we should be able to solve the problem directly, efficiently, and fast.

```
function fibonacci_even_sum(maxval)
  current, previous = 2, 0
  sum = current + previous
  next = 4current + previous # recurrence relation for even-valued Fibonacci numbers
  while (next <= maxval)
    sum += next
    current, previous = next, current
    next = 4current + previous # recurrence relation for even-valued Fibonacci numbers
  end
  return sum
end
```

Observe the following:

1. The assignment `current, previous = 2, 0` is idiom for `(current, previous) = (2, 0)` and is the syntax for `tuples` assignment in `Julia`. One could either use the parentheses or leave them out altogether.
2. The expression `4current` is shorthand for `4*current` and is yet another aspect of the language that makes its syntax mathematics-friendly and nimble.

3. The `while` condition need not have parentheses, although I have shown it here with them.
4. Lines need not end with a `;` and the return value need not be wrapped in parentheses. Code written without syntactic clutter like this is easier on the eye and also simpler to decode to reveal the underlying algorithm.

The function must be called separately from its definition and the value of the sum should be printed out. If, therefore, we append the following two lines to the above function, we would get our final result:

```
evensum = fibonacci_even_sum(4000000)
println(evensum)
```

which gives `4613732` as before, and all is well.

Final assessment

The `Julia` programming language is refreshingly original in its syntax and allows the programmer to solve the problem in very many ways.

In the case of Euler Project Problem 2, I found out that I ran into trouble, mostly because I was running foul of doing things the "right way". The language gently nudges one to think again before coding. It coaxes rather than coerces the programmer to adopt efficient and safe coding practices.

The existence of a knowledgeable user-community who were ready to help, and who could illuminate the problem from different angles, made learning `Julia` enjoyable, educational, and enriching. It is a language that I will spend time learning properly, and use in the future.

Caveat Lector! or Reader Beware! or Disclaimer

I am new to `Julia`. What I have written here represents my efforts at learning, errors and all. Experienced "Julians"[1] who find errors are requested to email me with their corrections. ☹

Acknowledgements

The `Julia`-user community is knowledgeable, courteous, and helpful, and they are very enthusiastic about their programming language. The website to ask for help is Julia Programming Language - A forum for users and developers.

When I encountered difficulties with my code, I sought the community's help in this thread [1]. The wealth of information given there is enough to keep one busy for quite a while, learning different ways to *solve* the same problem and also different ways to *think* about problems in general.

[1] `Rust` pogrammers call themselves "Rustaceans". `Python` programmers call themselves "Pythonistas". I propose that `Julia` programmers call themselves "Julians".

Feedback

Please email me your comments and corrections.

A PDF version of this article is available for download here:

https://swanlotus.netlify.app/blogs/euler-two-with-julia.pdf

References

[1] Various. 2023. Help with Project Euler #2 undef inits, printing, multiplication by juxtaposition, and more. Retrieved 2 December 2023 from https://discourse.julialang.org/t/help-with-project-euler-2-undef-inits-printing-multiplication-by-juxtaposition-and-more/106930

