Challenge 2

Optimize Fibonacci Sequence Generation

The main idea was to go for matrix exponentiation. The Fibonacci sequence can be seen as a system of linear equation of the form $\vec{\mathbf{F}}_{n+1} = \mathbf{A}\vec{\mathbf{F}}_n$:

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} F_2 + F_1 \\ F_2 \end{bmatrix}$$

More generally:

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

By Induction, we see that:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

$$\mathbf{AA} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_3 & F_2 \\ F_2 & F_1 \end{bmatrix}$$

$$\mathbf{A}^2\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} F_4 & F_3 \\ F_3 & F_2 \end{bmatrix}$$

Thus:

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

This can be implemented using Numpy as follows `np.linalg.matrix_power(A, n-1)[0, 0]`. The first argument being the matrix $\mathbf{A}$ and the second argument being the exponent. I used `n-1` due to Pythons indexing and `[0,0]` is used to access F_{n+1}. This has a time complexity of $\mathcal{O}(1)$ space and $\mathcal{O}(\log n)$ operations.

An even faster method is to use fast doubling, which circumvents the redundant calculation in the matrix, i.e. F_n is computed twice. The maths were taken from here.

To find the 2n-th element in the Fibonacci sequence, we can use:

$$F_{2n} = F_n[2 \cdot F_{n+1} - F_n]$$

This method is faster by a constant factor compared to matrix exponentiation. To further speed up the computation faster multiplication algorithm could be used, since the integers that are multiplied are in the range of 10^{200000}. There is the Karatsuba algorithm or Schönhage–Strassen algorithm. I implemented the Karatsuba algoritm in Python, but it is actually slower because Python is an interpret-language.

The sum of Fibonacci Numbers

To iterate through all Fibonacci numbers up to $4 \cdot 10^6$ does not make sense as the number of operations explodes. An idea is to iterate in steps of 3 since the Fibonacci sequence has the form *odd, odd, even, odd, odd, even, ...*, but also this improvement is not good enough.

The sum of all Fibonacci numbers is $\sum_n^i F_i = F_1 + F_2 + ... + F_n$

n	F_n	Sum
1	1	1
2	1	2
3	2	4
4	3	7
5	5	12
6	8	20
7	13	33
8	21	54
9	34	88

It can be shown that the sum of the n-th Fibonacci number is the $F_{n+2} - 1$-th Fibonacci number.

$$\sum_n^i F_i = F_{n+2} - 1$$

$$F_n = F_{n+2} + \cancel{F_{n+1}}$$
$$F_{n-1} = \cancel{F_{n+1}} + \cancel{F_n}$$
$$...$$
$$F_3 = \cancel{F_2} + F_1$$

In a similar way it can be shown that the sum of the even Fibonacci numbers is:

$$F_2 + F_4 + ... + F_{2n} = F_{2n+1} - 1$$

Proof : Math Stack Exchange
The largest Fibonacci number that is even and below $4 \cdot 10^6$ is 3'999'998

The Algos

Algo1

The principle is to use the above mentioned matrix exponentiation.

1. Initialize variables $\mathsf{F_i}$ (it should be $\mathsf{S_i}$ aktschually) (sum of Fibonacci number), and the iterator i. Both are set to $\mathsf{0}$.

2. Begin of the `while` loop with the condition that $\mathsf{F_i} < \mathsf{4e6}$. Here I applied a little trick. By mistake I found that the sum of all Fibos below $\mathsf{4e6}$ is itself below $\mathsf{4e6}$ and thus the algorithm stops at the right time.
   But why? This allows you to skip the creation of a `sum` variable!

3. Use $\mathbf{A}^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$, where $\mathsf{F_i}$ is in the anti-diagonal of A. Add $\mathsf{F_i}$ to itself.

4. Increase iterator `i` by 3, since Fibonacci sequence is *odd, odd, even, odd, odd, even, etc.* This allows to skip checking `F_i mod 2` and reduces the number of iterations.

Conclusion: Matrix exponentiation is master-race when n becomes huge, but tiny Fibonacci's like Fibo(35) are a joke. The method is out-competed by the other algorithms due to a bigger function overhead, plus it requires `import` of **numpy**.

Algo2

The principle is to use the above mentioned matrix exponentiation, but instead of directly summing in every iteration, this algo chooses to append $\mathsf{F_i}$ to a list and sum at the end.

1. Initialize variables $\mathsf{F_i}$ (list of Fibonacci number), and the iterator i. Variables are set to ![]! and $\mathsf{0}$, respectively.

2. Begin of the `while` loop with the condition that $\mathsf{F_i} < \mathsf{4e6}$. Here the trick is not applied and the length of the current Fibonacci number is evaluated every time.

3. Use $\mathbf{A}^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$, where $\mathsf{F_i}$ is in the anti-diagonal of A. Append $\mathsf{F_i}$ to list.

4. Increase iterator `i` by 3, since Fibonacci sequence is *odd, odd, even, odd, odd, even, etc.* This allows to skip checking `F_i mod 2` and reduces the number of iterations.

5. Summation of all numbers minus the last index.

Conclusion: It's worse than **algo1**, handling lists is less efficient.

Algo3

This one uses a really efficient loop that skips the use of checking for even numbers and also iterates in steps of 3. It is the most efficient.

1. Initialize variables F_n, F_{n+1}, and F_{n+2} in the usualy Fibonacci manners. This algo also uses a list to sum the Fibonacci numbers.
2. Begin of the `while` loop with the condition that $F_{n+2} < 4e6$. The efficiency comes from the loop:

Iteration 0	Iteration 1	Iteration ...
$F_n = 1 = F_1$	$F_n = 1 + 2 = F_4$	...
$F_{n+1} = 1 = F_2$	$F_{n+1} = 2 + 3 = F_5$	...
$F_{n+2} = 2 = F_3$	$F_{n+2} = 3 + 5 = F_6$	...

F_{n+2} is always even.

3. Just wait until done.

Conclusion: That's the best one, since the iterations are super efficient and just consist of 3 additions and 3 assignments.

Algo4

This one is also pretty fast. It uses the Binet's formula.

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

1. Quite a number of variables are initialized:
   - sqrt5 : $\sqrt{5}$ (reduces the number of times the square root is calculated)
   - phi : $\frac{\sqrt{5}+1}{2}$ (same here)
   - i : 0 (iterator)
   - F_i : 0 (sum of Fibonacci numbers)
2. Begin of the `while` loop with the condition that $F_{n+2} < 4e6$. The same trick as in Algo1 can be used.
3. Sum all F_i's.

Conclusion: That one is also very fast since the computation are few and fast.