Computing Fibonacci numbers efficiently

Steven J. Miller
sjm1@williams.edu

Arman Rysmakhanov
ar21@williams.edu

Sheafification of G
gsheafified@gmail.com

This article serves as a companion to the video essay [8] by Sheafification of G. We follow the same progression of methods—from naïve recursion to sophisticated Fourier transforms—and elaborate the concepts that were impractical to discuss in video format.

1. Introduction

The president of your college makes a strange offer: you get an 'A+' in every class over the course of your studies if you can find the millionth Fibonacci number by hand in four years. Suppose that you can do three digit operations per second without ever sleeping. Do you take the offer?

Let's start with definitions. The Fibonacci sequence is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$, with initial terms F_0 and F_1. This seemingly simple sequence appears throughout mathematics and nature, making efficient computation of its terms an interesting algorithmic challenge. For the purposes of this article, $F_0 = 1$ and $F_1 = 1$, resulting in the sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584$, and so on. You can continue calculating the terms of the sequence for some time, but what is your limit? How long do you think it will take to compute, for instance, the $1,000,000$-th Fibonacci number by hand? Let's make quick approximations.

Each number in the Fibonacci sequence is the sum of the previous two, and since both previous numbers are positive and growing, we should expect rapid growth. But how rapid, exactly? We can bound this growth from both sides. First, since $F_n = F_{n-1} + F_{n-2}$ and $F_{n-2} < F_{n-1}$, we have $F_n < F_{n-1} + F_{n-1} = 2F_{n-1}$. Similarly, $2F_{n-1} < 4F_{n-2}$, and $4F_{n-2} < 8F_{n-3} < \ldots < 2^n \cdot F_1 = 2^n$. Therefore 2^n is an upper bound for the growth of Fibonacci numbers.

To estimate the lower bound, combining the equations for consecutive terms,

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ F_{n-1} &= F_{n-2} + F_{n-3}, \end{aligned}$$

we get $F_n = 2F_{n-2} + F_{n-3}$, implying $F_n > 2F_{n-2}$, and therefore $2F_{n-2} > 4F_{n-3} > \ldots > 2^{n/2} = \sqrt[n]{2}$. Think of it as dividing F_n by a number that is greater than 2 every two steps, with the total number of steps being $1/n$. Thus we have shown

$$(\sqrt{2})^n < F_n < 2^n.$$

Fibonacci numbers grow *exponentially* fast—which is bad news for manual calculation! But how bad exactly? The number of digits of n-th Fibonacci number, d_n, can be approximated

as

$$\begin{aligned} \log_{10}(\sqrt{2})^n + 1 \;&<\; d_n \;<\; \log_{10} 2^n + 1 \\ \frac{1}{2} n \log_{10} 2 + 1 \;&<\; d_n \;<\; n \log_{10} 2 + 1. \end{aligned}$$

Since

$$\sum_{i=1}^{999,999} n \log_{10} 2 \;=\; \log_{10} 2 \cdot 1,000,000 \cdot \frac{1}{2}(999,999+1) \;\approx\; 1.5 \cdot 10^{11},$$

we see that the number of digit operations you need to perform to get to the millionth Fibonacci number is approximately between $7.5 \cdot 10^{10}$ and $1.5 \cdot 10^{11}$. (Note that it is possible to derive a more exact growth rate of Fibonacci numbers—the point of this exercise was to show how you could quickly estimate it by hand,)

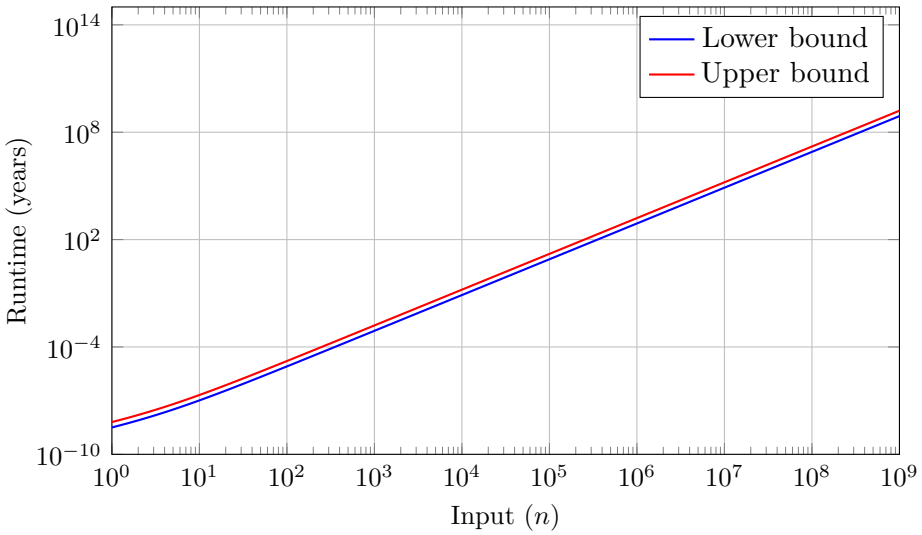

Figure 1: Years needed to calculate the n-th Fibonacci number by hand (log-log scale).

If you can perform 3 single-digit additions on paper per second, and there are $60 \cdot 60 \cdot 24 \cdot 365 \;=\; 31,536,000$ seconds in a year, then it would take you at least around 795 years to complete the task! So we strongly advise you to turn down such an offer if you ever come across one. In fact, as seen in the graph above, one does not have realistic chances of computing beyond the 100,000-th Fibonacci number alone. If you can employ the whole campus, say 5,000 people, then the task becomes more reasonable—at most 1/3 of a year of non-stop calculations. What about the trillionth Fibonacci number? Computing it by hand will take at least around 100,000 years if all of humanity unites in this effort. It is clear that we need to use computer algorithms. In this article, we will compare seven possible methods to find the most efficient, and will try to build our intuition about them from the ground up.

2. Comparison of Algorithms

Before diving into different methods, let's define what we mean by algorithmic efficiency.

Remark (Big O Notation)**.** When we say an algorithm runs in $O(f(n))$ time, we mean that for sufficiently large input size n, the algorithm's runtime is bounded above by some constant multiple of $f(n)$. This gives us a way to compare algorithms' performance as input sizes grow large.

Note that, for instance, both $100n$ and n are $O(n)$, because 100 is a constant factor that is negligible for sufficiently large n. In the same way, $8n^3 + 9n + 5$ is $O(n^3)$ since the quadratic term has by far the most "weight" when n is, say, 10 million. Thus, in most cases, we can ignore both the constants and the lower-order terms when analyzing the efficiency of algorithms.

Here is the graphical summary of all the methods that we will consider:

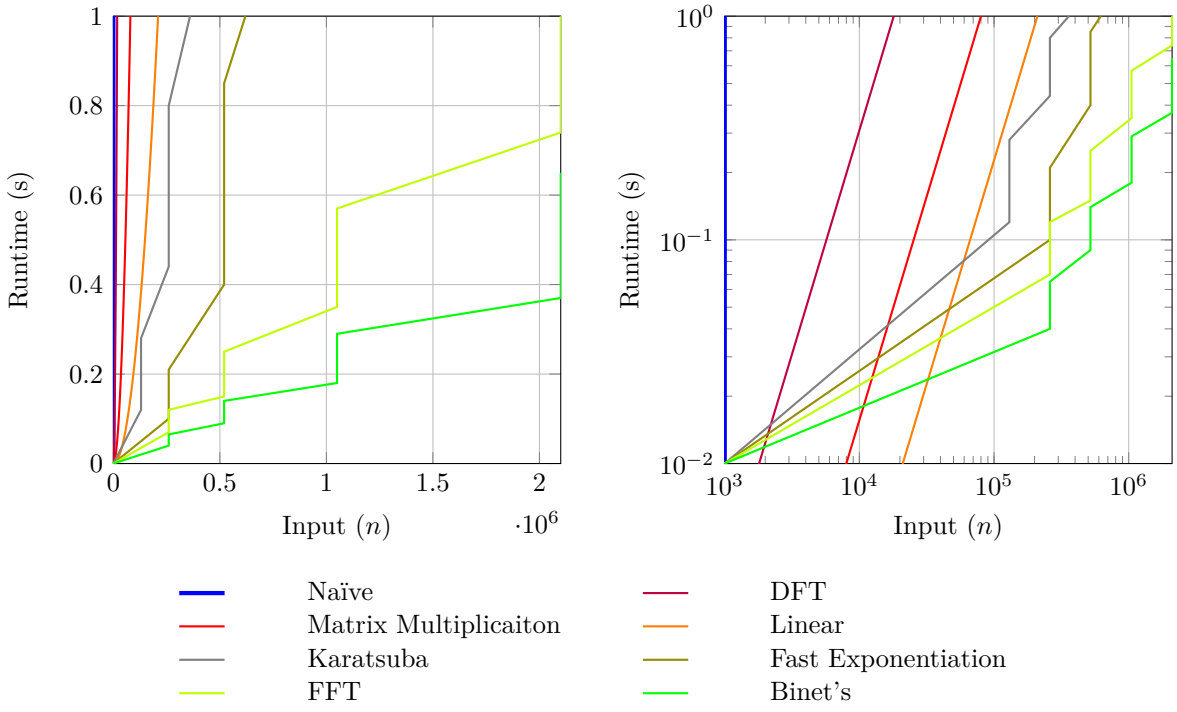

Figure 2: Runtime comparisons of algorithms calculating n-th Fibonacci number. Left: linear scale. Right: log-log scale.

With the Big O notation in mind, let's consider each algorithm in detail, beginning with the most intuitive but least efficient one—naïve recursion.

3. Naïve method

If you were introduced to recursive branching in a computer science class, it is very likely that computing the n-th Fibonacci number was used as an example.

```
def recursive_fibonacci(n):
    if n == 1 or n == 0:
```

```
        return 1
    return recursive_fibonacci(n-1) + recursive_fibonacci(n-2)
```

Although this approach has pedagogical value, it is hopelessly slow. Let's take a look at the callback tree to see why.

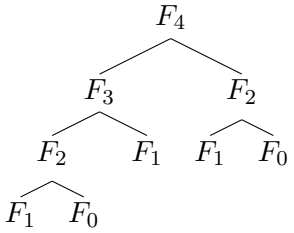

Figure 3: Callback tree of `recursive_fibonacci(n)` for $n = 4$.

Let T(n) denote the number of function calls made by `recursive_fibonacci(n)`. The tree has depth of n and each of its internal nodes calls the function twice, which suggests examining the power of 2.

The maximum number of nodes in a binary tree with depth n is $2^n - 1$, so $T(n) = O(2^n)$. However, we also need to find a lower bound to claim that the algorithm has exponential time complexity. Consider the leftmost path of recursive calls, F_4 to F_1 in our case. At each level k, we have $T(k) = T(k-1)+T(k-2)$. Since computing F_{k-1} implies having computed F_{k-2} on top of other computations, we see that $T(k) > 2T(k-2) > 4T(k-4) > 8T(k-6) > \cdots > 2^{k/2}\ T(0)$. Therefore, T(n) $= O(2^n)$, as it is bounded both above and below by exponential functions with base 2.

Remark (Relating bounds to Fibonacci)**.** A more tight bound for the recursive algorithm is connected to Fibonacci numbers themselves! Remember, we define $F_0 = 1$ and $F_1 = 1$, and F_n is calculated as the sum of all base cases, so the number of all leaf nodes is F_n. Given that the number of leaf nodes in a binary tree is greater than the number of internal nodes by 1, and that $2n = O(n)$, we can obtain a more tight bound of O(ϕ^n)—more on ϕ in the section on Binet's Formula.

4. "Linear" Method

Instead of recalculating values multiple times through recursion, we can compute Fibonacci numbers sequentially. Note that branching recursion implies calculating the same values multiple times: F_2 was calculated twice in the previous algorithm. On top of that, the algorithm's space complexity is $O(n)$ as it is proportional to the tree depth. A natural improvement then would be building F_n from the bottom up.

```
def linear_fibonacci(n):
    i = 0
    a = 1
    b = 1
    while i <= n:
```

```
        temp = b
        b = b + a
        a = temp
        i += 1
    return b
```

It seems that the time complexity should be linear, since we are adding numbers n times. This would have been true if the cost of addition were constant—but Fibonacci numbers get large very quickly ($F_{10,000}$, for example, has 2090 digits), so the cost of addition becomes linear, and the time complexity becomes $O(n^2)$.

5. Matrix Multiplication Method

The recursive nature of the Fibonacci sequence hints at another solution that could potentially be more efficient. The recurrence relation $F_n = F_{n-1} + F_{n-2}$ is linear, suggesting that we can express it as a matrix transformation. If we find the right matrix, each multiplication will generate the next Fibonacci number, turning the problem into matrix exponentiation.

We can try to improve on the algorithm by taking a linear algebra approach. Consider the matrix equation

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 \cdot F_{n-1} + 1 \cdot F_{n-2} \\ 1 \cdot F_{n-1} + 0 \cdot F_{n-2} \end{bmatrix} = \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}.$$

Note that the first row computes F_n, while the second row carries forward F_{n-1}.

This relationship allows us to conclude:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \ldots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Unfortunately, this algorithm performs even worse than the "Linear" one, since multiplying 2×2 matrices requires 4 additions of numbers that get *very* large. (Note that multiplication has a constant cost since we are always multiplying by 0 or 1, e.g. $110111 \cdot 0 = 0$ and $110111 \cdot 1 = 110111$.)

6. Fast Matrix Exponentiation

After noticing that matrix multiplication requires too many operations with big integers, we need a method that minimizes such operations. The key insight comes from efficiently computing powers of 2.

Consider computing A^{16}: instead of $\underbrace{A \cdot A \cdot A \cdot \cdots \cdot A}_{\text{15 multiplications}}$, we can compute it in just 4 steps:

$$\begin{aligned} A^2 &= A \cdot A \\ A^4 &= A^2 \cdot A^2 \\ A^8 &= A^4 \cdot A^4 \\ A^{16} &= A^8 \cdot A^8. \end{aligned}$$

In the general case, for any power n, we can write it in binary: $n = \sum_{i=0}^{k} b_i 2^i$ where $b_i \in \{0, 1\}$. For example, $A^{11} = A^8 \cdot A^2 \cdot A^1$ since $11 = 1011_2$. This means computing and storing A^1, A^2, A^4, A^8, but only multiplying the matrices whose power corresponds to 1's in the binary representation.

Unlike the previous method where only additions were costly, here each matrix multiplication involves big integer multiplication. Each multiplication of n-digit numbers requires $O(n)$ additions, so the total complexity might seem to be $O(n^2 \log n)$, but it is actually better.

Remark (Master Theorem)**.** In algorithm analysis, the Master Theorem provides the toolkit for analyzing recurrence relations of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ is recursive calls and $b > 1$ is the reduction factor. The time complexity is determined by comparing the growth rates of the recursive term and the combining function. We will see several applications of this theorem later. For more details, see the paper on Master Theorem [3].

Let $T(n)$ be the time complexity to compute F_n. Then,

$$T(n) \leq T(n/2) + kn^2,$$

where kn^2 is the cost of the final matrix multiplication. This is a recurrence relation that can be solved by repeatedly expanding. Expanding the recurrence

$$\begin{aligned} T(n) &\leq T(\frac{n}{2}) + kn^2 \\ &\leq T(\frac{n}{4}) + k(\frac{n}{2})^2 + kn^2 \\ &\leq \cdots \\ &\leq kn^2 \ (1 + \frac{1}{4} + \frac{1}{16} + \cdots) \\ &\leq \frac{4kn^2}{3}, \end{aligned}$$

where the sum in parentheses is a geometric series with the ratio of $1/4$, gives us $O(n^2)$, since $4k/3$ is a constant.

Looking at the graph of the Fast Multiplication algorithm, you might be wondering why it has "jumps" at $n = 131072$, 262144, 524288, etc. These are not arbitrary! They are powers of two, namely 2^{17}, 2^{18}, 2^{19}, marking points where the binary representation requires one more digit and thus one more matrix multiplication in our exponentiation process.

Why do we care more about multiplications than additions? It turns out that addition of n-digit numbers is, in theory, linear, but multiplication requires $O(n^2)$ operations.

This approach works much better than the previous ones, but it still uses big integer operations (arithmetic on numbers too large for built-in integer types), although not as frequently as the previous ones. The most we can squeeze out of this approach is applying the Strassen Algorithm [10] to multiply matrices more efficiently—but even that will only reduce the number of multiplications from 8 to 7 and will not result in substantial improvements. This begs a question—is there a better way to multiply?

7. Karatsuba's Algorithm

Having seen the limitations of regular multiplication, we need a different approach to multiplying large numbers. The key insight comes from representing an n-digit number as $a \cdot B^{n/2} + b$ where B is the base (typically 10 or 2^{32} in practice) and a, b are $n/2$-digit numbers. For example, $1234 = 12 \cdot 10^2 + 34$.

When multiplying two such numbers,

$$(a \cdot B^{n/2} + b)(c \cdot B^{n/2} + d) = ac \cdot B^n + (ad + bc) \cdot B^{n/2} + bd,$$

it seems like we need four multiplications: ac, ad, bc, and bd. However, Karatsuba discovered a very useful trick [7], which says that if we also compute $(a+b)(c+d)$, we can recover $ad+bc$ in the following way:

$$\begin{aligned} (a+b)(c+d) &= ac + ad + bc + bd \\ ad + bc &= (a+b)(c+d) - ac - bd. \end{aligned}$$

This reduces the number of multiplications from four to three! The complete product can then be computed as

$$ac \cdot B^n + [(a+b)\ (c+d) - ac - bd] \cdot B^{n/2} + bd.$$

Let $T(n)$ be the time complexity of multiplying two n-digit numbers using this method. Each multiplication now involves three recursive calls on numbers of size $n/2$, plus some linear work for additions and shifts, giving us

$$T(n) \leq 3T(n/2) + kn,$$

where kn represents the cost of additions and shifts. What if we apply this trick once more on the numbers a, b, c, d and continue applying it until no further decomposition is possible? The recurrence relation describing the time complexity of this procedure is given by

$$\begin{aligned} T(n) &\leq 3T(n/2) + kn \\ &\leq 9T(n/4) + 2.5kn \\ &\leq 27T(n/8) + 4.75kn \\ &\leq \cdots \\ &\leq 3^{\log_2 n}\ T(1) + ckn. \end{aligned}$$

Since $3^{\log_2(n)} = n^{\log_2(3)}$, we obtain $T(n) = O(n^{\log_2(3)})$, where $\log_2 3 \approx 1.585$, which is significantly better than the quadratic complexity of standard multiplication.

However, looking at the graph, we see that Karatsuba's algorithm is worse than Fast Matrix Multiplication. The reason lies in the algorithm's overhead: we need to allocate additional memory for the intermediate results $(a+b)$ and $(c+d)$, which can be up to one digit longer than a, b, c, d themselves. When implementing this recursively, we need approximately 8 times as much workspace as the input size to ensure all recursive calls have enough space. (The factor of 8 comes from needing space for both operands in each of the three recursive calls, plus space for intermediate results.)

This brings us to an important insight: asymptotic complexity improvements do not always translate to performance gains for typical input sizes. In our case, the overhead of memory allocation and management outweighs the theoretical benefits until the numbers become extremely large—well beyond the one-second time limit. Let's continue looking for better ways to multiply.

8. Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a fundamental tool in signal processing that transforms a sequence of numbers into components of different frequencies. While a complete treatment of DFT is beyond our scope, some excellent introductions include these notes [4] and this video [1]. We will focus on how DFT can help us multiply large numbers more efficiently.

To start, note that any integer can be viewed as a polynomial of degree $N-1$ in some base B. For instance, the decimal number 234 can be written as $2B^2 + 3B^1 + 4B^0$. More formally, for all x and y, we can write

$$x \;=\sum_{i=0}^{N-1} x_i B^i, \quad y \;=\sum_{j=0}^{N-1} y_j B^j.$$

Their product can then be expressed as $(f \times g)(B)$ where $f(B)$ and $g(B)$ are the following polynomials:

$$\begin{aligned} f(B) &= x_0 + x_1 B + x_2 B^2 + \cdots + x_{N-1} B^{N-1} \\ g(B) &= y_0 + y_1 B + y_2 B^2 + \cdots + y_{N-1} B^{N-1}. \end{aligned}$$

The product polynomial $(f \times g)(B)$ has degree at most $2N-2$,

$$(f \times g)(B) \;=\; z_0 + z_1 B + z_2 B^2 + \cdots + z_{2N-2} B^{2N-2},$$

where

$$z_n \;=\sum_{m=0}^{N-1} x_m y_{N-1-m}$$

is the convolution of coefficients.

Remark (Polynomial Interpolation)**.** A polynomial of degree less than M is uniquely determined by its values at any M distinct inputs. This is because interpolating a polynomial from points leads to a system of linear equations—you can't solve it if you have fewer equations (points) than unknowns (coefficients).

Moreover, for any B, $(f \times g)(B) \;=\; f(B)g(B)$; multiplication of polynomials at a specific input is simply a multiplication of numbers.

This suggests a three-step algorithm:

1. Evaluate $f(B)$ and $g(B)$ at $2N$ distinct points (we need $2N$ points because the product polynomial has degree $2N-2$).
2. Multiply these values pointwise (that is, multiply corresponding values at each point).

3. Use the $2N$ resulting values to reconstruct the product polynomial.

Remark (Roots of Unity)**.** We evaluate the functions specifically at $2N$-th roots of unity

$$\omega_{2N}^{-k} \;=\; e^{\frac{2\pi i(-k)}{2N}},$$

which are complex numbers on the unit circle satisfying $\omega_{2N}^{2N} \;=\; 1$ and $\omega_{2N}^{j} \;\neq\; 1$ for all integers $j \;<\; 2N$. These special points make the inverse transformation (step 3) particularly efficient. For a more detailed explanation of roots of unity, see these notes [6].

This approach, while theoretically elegant, has practical drawbacks. The algorithm is $O(n^2)$ due to two nested loops in the DFT computation: we are computing n coefficients, each of which requires summing over n terms. But why has it performed worse than other quadratic algorithms? The reasons are the following.

- We need to compute three polynomials: two forward transforms for f and g, and one inverse transform to recover the coefficients.
- We need to compute exponentials to obtain complex numbers.
- We need to use pairs of double-precision floating point numbers to represent complex numbers, increasing both memory usage and computational overhead.

This might seem like a step backward—but there is a way to fix DFT.

9. Fast Fourier Transform (FFT)

There is a clever way to reorganize the computation in DFT, known as the Fast Fourier Transform (FFT), developed by Cooley and Tukey in 1965 [2], which significantly improves its efficiency.

The main idea is splitting the sum based on even and odd indices. Consider the DFT of a sequence of length M:

$$\widehat{a}_k \;=\; \sum_{\ell=0}^{M-1} a_\ell \omega_M^{-k\ell}.$$

We can separate this into even and odd indexed terms:

$$\begin{aligned}
\widehat{a}_k \;&=\; \sum_{\ell=0}^{\frac{M}{2}-1} a_{2\ell}\; \omega_M^{-2k\ell} + \sum_{\ell=0}^{\frac{M}{2}-1} a_{2\ell+1}\; \omega_M^{-k(2\ell+1)} \\
&=\; \underbrace{\sum_{\ell=0}^{\frac{M}{2}-1} a_{2\ell}\; \omega_{M/2}^{-k\ell}}_{\text{DFT of even terms}} + \omega_M^{-k} \underbrace{\sum_{\ell=0}^{\frac{M}{2}-1} a_{2\ell+1}\; \omega_{M/2}^{-k\ell}}_{\text{DFT of odd terms}}.
\end{aligned}$$

Note that

$$\omega_M^{-2k\ell} \;=\; (\omega_M^2)^{-k\ell} \;=\; \omega_{\frac{M}{2}}^{-k\ell},$$

where $\omega_{M/2}$ is a $(M/2)$-th root of unity. This allows us to rewrite

$$\widehat{a}_k \;=\; E_k + \omega_M^{-k}\; O_k,$$

where E_k and O_k are the DFTs of the even and odd subsequences, each of length $M/2$.

Another crucial property of the M-th root of unity is that $\omega_M^{k+M/2} = -\omega_M^k$. This means

$$\widehat{a}_{k+\frac{M}{2}} = E_k - \omega_M^{-k} O_k.$$

This symmetry property shows us that once we compute E_k and O_k for $k < M/2$, we already know the other values. The computational savings are substantial: instead of doing M computations, we only need to do $M/2$ computations at each level.

The time complexity follows a beautiful recursive pattern. Let $T(M)$ be the time complexity of FFT with M inputs,

$$T(M) \leq 2T(\frac{M}{2}) + kM,$$

where kM represents the combining step at each level. Expanding the recurrence results in

$$\begin{aligned} T(M) &\leq 4T(M/4) + kM + 2k(M/2) \\ &\leq 8T(M/8) + kM + 2k(M/2) + 4k(M/4) \\ &\vdots \\ &\leq kM \log_2 M. \end{aligned}$$

The final sum follows because we have $\log_2 M$ levels, and at each level we do work proportional to M. This results in the complexity of $O(n \log n)$, a substantial improvement that makes large-scale multiplication practical, which is reflected in the graph of this algorithm as compared to others.

10. Binet's Formula

While FFT improved the matrix multiplication, there is another approach that fundamentally changes how we handle the transition matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = P \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} P^{-1}$$

A matrix is diagonalizable if it can be written as PDP^{-1} where D is a diagonal matrix and P is an invertible matrix. This is useful for us since it allows for easier computation of matrix powers. However, not all matrices can be diagonalized—it can be done if and only if the matrix has n linearly independent eigenvectors, with n being the dimension of the matrix. That is equivalent to having n distinct eigenvalues because distinct eigenvalues result in linearly independent eigenvectors. For more details on diagonalization see, for example, these notes [5].

Our 2×2 Fibonacci transition matrix has two distinct eigenvalues, $\lambda_1 = \phi = \frac{1+\sqrt{5}}{2}$ (the

golden ratio) and $\lambda_2 = \psi = \frac{1-\sqrt{5}}{2}$ (its conjugate), allowing us to diagonalize it:

$$\begin{aligned}\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= P \begin{bmatrix} \phi^n & 0 \\ 0 & \psi^n \end{bmatrix} P^{-1} \\ &= \begin{bmatrix} 1 & 1 \\ \phi & \psi \end{bmatrix} \begin{bmatrix} \phi^n & 0 \\ 0 & \psi^n \end{bmatrix} \begin{bmatrix} 1 & 1 \\ \phi & \psi \end{bmatrix}^{-1} \\ &= \frac{1}{\psi - \phi} \begin{bmatrix} -\phi^n + \psi^n \\ -\phi^{n+1} + \psi^{n+1} \end{bmatrix}.\end{aligned}$$

This gives us Binet's formula:

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}} = \frac{(1+\sqrt{5})^n}{2^n\sqrt{5}} - \frac{(1-\sqrt{5})^n}{2^n\sqrt{5}}.$$

It was mentioned above that using this formula directly would not work, as we would be raising irrational numbers to large powers, accumulating rounding errors rapidly. There is a trick—we can operate in terms of numbers of the form $a+b\sqrt{5}$ where a, b are integers, forming what is called the ring $\mathbb{Z}[\sqrt{5}]$. You can learn more about such algebraic structures here [9].

Remark (Working in $\mathbb{Z}[\sqrt{5}]$)**.** We can encode elements of $\mathbb{Z}[\sqrt{5}]$ as pairs of integers (a, b) representing $a + b\sqrt{5}$. The arithmetic operations are

$$\begin{aligned}(a, b) + (a', b') &= (a + a', b + b') \\ (a, b) \times (a', b') &= (aa' + 5bb', ab' + a'b).\end{aligned}$$

These rules come from the usual algebra of $a + b\sqrt{5}$, keeping track of rational and irrational parts separately.

Since $|\psi| < 1$, we have $\psi^n \to 0$ as $n \to \infty$, allowing us to approximate the n-th Fibonacci number:

$$F_n \approx \frac{\phi^n}{\sqrt{5}} = \frac{(1+\sqrt{5})^n}{2^n\sqrt{5}} = \frac{1}{2^n} \text{ [coefficient of } \sqrt{5} \text{ in } (1+\sqrt{5})^n].$$

While this algorithm still requires multiplying large numbers, we are only dealing with pairs of numbers rather than 2×2 matrices. Using FFT for the multiplication operations gives us $O(n \log n)$ complexity, but with better constant factors than the matrix multiplication methods since we are operating on less data.

11. Practical Limitations of FFT

The fundamental issue lies in how computers represent complex numbers. In our implementation, each complex number is encoded as a pair of double-precision floating-point numbers (for real and imaginary parts). These doubles follow the standard that provides a 52-bit mantissa (the significant digits of a number) [11], plus one implicit bit, plus potentially one bit for rounding. This gives us effectively 54 bits of precision.

Remark (Floating-Point Representation)**.** A floating-point number is stored as $\pm m \times 2^e$ where m is the mantissa (a binary fraction) and e is the exponent. The mantissa determines how many significant digits we can represent accurately.

What does that mean for FFT? When we transform an N-byte number (where each byte represents a base-256 digit), here is what happens:

1. Each input digit requires 8 bits (one byte) of precision.
2. The position of each digit contributes $\log N$ additional bits.
3. Therefore, transforming one number needs $(8 + \log N)$ bits.
4. When multiplying two such numbers, intermediate results can require twice the precision.

This leads to our precision requirement:

$$\begin{aligned} 16 + 2\log_2 N &\leq 54 \\ \log_2 N &\leq 19. \end{aligned}$$

This means N, the byte-length of our numbers, is bounded by approximately 2^{19}. Since each byte represents a base-256 digit, the upper limit will be around $F_{6{,}000{,}000}$. Beyond this point, the accumulation of rounding errors in the floating-point arithmetic would make the results unreliable.

12. Conclusion

Let's return to the president's offer and think what would change if you could use your computer. What if it was F_{100}? Accept on the spot, even if it is to be done by hand. The millionth Fibonacci number? It can be done, but make sure to select an efficient algorithm to finish it in a reasonable amount of time. The trillionth Fibonacci number? Even with the best algorithms, you would run into the problem of operating on numbers with hundreds of billions of digits, so it would be wise to pass on the offer.

Building intuition for different algorithms for computing Fibonacci numbers showed us several things.

The progression from exponential to quadratic to $n \log n$ complexity demonstrates the power of mathematical approaches in computational problems. We have seen how each improvement—from recursion to matrix multiplication to Fourier transforms—showcases a different way of thinking about the same problem. Yet even our most sophisticated methods eventually hit limits, whether from floating point precision or the sheer size of numbers, serving as a reminder of the importance of hardware constraints.

References

[1] Steve Brunton. *The Discrete Fourier Transform (DFT)*. 2020. URL: https://www.youtube.com/watch?v=nl9TZanwbBk.

[2] J. Cooley and J. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301.

[3] T. Cormen et al. *Introduction to Algorithms*. 2nd ed. Sections 4.3 (The master method) and 4.4 (Proof of the master theorem). MIT Press and McGraw–Hill, 2001, pp. 73–90. ISBN: 0-262-03293-7.

[4] B. Delgutte and J. Greenberg. *The Discrete Fourier Transform*. 1999. URL: https://web.mit.edu/~gari/teaching/6.555/lectures/ch_DFT.pdf.

[5] *Diagonalization and Powers of A*. 2011. URL: https://ocw.mit.edu/courses/18-06sc-linear-algebra-fall-2011/d05795355f5cbe35356a4a1b39658336_MIT18_06SCF11_Ses2.9sum.pdf.

[6] *Fast Fourier Transform*. 2023. URL: https://www.cs.cmu.edu/~15451-s23/lectures/lec24-fft.pdf.

[7] A. Karatsuba and Yu. Ofman. "Multiplication of many-digital numbers by automatic computers". In: *Dokl. Akad. Nauk SSSR* 145.2 (1962), pp. 293–294.

[8] Sheafification of G. *One second to compute the largest Fibonacci number I can*. 2023. URL: https://www.youtube.com/watch?v=KzT9I1d-LlQ.

[9] P. Stevenhagen. *Number Rings*. 2004. URL: https://kconrad.math.uconn.edu/math5230f08/stevenhagen.pdf.

[10] V. Strassen. "Gaussian Elimination is not Optimal". In: *Numer. Math.* 13.4 (1969). S2CID: 121656251, pp. 354–356. DOI: 10.1007/BF02165411.

[11] *Type float*. 2021. URL: https://learn.microsoft.com/en-us/cpp/c-language/type-float.