Abstract

This paper addresses the computation of the Fibonacci sequence with arbitrary precision, recognizing that while the Fibonacci sequence is straightforward from a mathematical perspective, its calculation encounters limitations when using a 64-bit environment, specifically above the 93rd Fibonacci number. To overcome this limitation, we explore various methods for computing the Fibonacci numbers, including the classic method, the Matrix exponentiation method, and the fast-doubling method. Additionally, Binet's direct formula is considered, but its accuracy is found to be constrained when using IEEE754 64-bit floating-point arithmetic. However, in the context of arbitrary precision, we propose an improved version that provides accurate results. Ultimately, a hybrid approach is presented as the preferred solution for efficiently computing the Fibonacci sequence with arbitrary precision.

Introduction

Similar to calculating factorials for integers, computing the Fibonacci sequence in a 64-bit environment has its limitations, reaching its maximum at the 93rd Fibonacci number. We initially examine the straightforward implementation using function recursion, but due to its inefficiency, we discard this method. Instead, we explore enhancements to the recursion by incorporating memoization, which mitigates redundant calculations. Subsequently, we present loop-based computation as a simpler alternative to calculating Fibonacci numbers. Moving towards more advanced techniques, we investigate the fast-doubling method and the Matrix exponentiation method, with an emphasis on further improving the fast-doubling method using memoization. A comparison is made between these two methods to determine their performance characteristics. Additionally, Binet's direct formula is briefly discussed, which, when using IEEE754 64-bit floating-point arithmetic, exhibits accuracy limitations beyond the 71st Fibonacci number. However, in the context of arbitrary precision, we propose an enhancement to this formula that ensures accurate results. The performance of all these methods is evaluated, leading to a recommendation and suggestion for implementing a hybrid approach to achieve optimal computational efficiency.

As customary, the actual C++ source code for the calculations will be provided, utilizing the author's own arbitrary precision Math library [1].

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision

5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision
12. Fast Computation of Stirling's numbers in arbitrary precision. HVE Fast Computation of Stirling numbers in arbitrary precision
13. Fast Prime computation in arbitrary precision. HVE Fast Prime Computation in arbitrary precision
14. Fast PRNG in arbitrary precision. HVE Fast PRNG in arbitrary precision
15. Fast Fibonacci sequence in arbitrary precision. HVE Fast Fibonacci in arbitrary precision
16. Fast Factorial in arbitrary precision. HVE Fast factorial and binomial for arbitrary precision.

Table of Contents

Contents

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handles arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision. Since Fibonacci numbers are integers, we only need to highlight the *int_precision* class.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision ip;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.

One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for *int_precision* variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mNumber;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integer to store our integer precision number.

The method .size() returns the number of internal vector entries needed to hold the number.

The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

For more details see [1].

The goal for implementing the Fibonacci sequence with arbitrary precision

Our objective is to devise an efficient method for calculating the Fibonacci sequence with arbitrary precision, surpassing the limitations of a 64-bit data type. Implementing this sequence requires exploring various algorithms or methods that can accommodate the extended precision requirements. While some algorithms may seem feasible and appropriate for a 64-bit environment, they often prove inadequate in delivering the desired performance and efficiency when dealing with larger precision values.

In this study, we will thoroughly examine several well-known methods for calculating the Fibonacci sequence and assess their performance within the 64-bit environment. Subsequently, we will identify the most promising methods that exhibit scalability beyond the constraints of the 64-bit environment. Our ultimate aim is to find the optimal approach capable of efficiently computing the Fibonacci sequence to any arbitrary precision.

The History of the Fibonacci Sequence

The history of the Fibonacci sequence traces back to a mathematician named Leonardo of Pisa, who is fondly known as Fibonacci. Hailing from 12th and 13th century Italy, Fibonacci introduced this sequence to the Western world through his influential work, "Liber Abaci" or the "Book of Calculation."

Interestingly, Fibonacci was not the originator of the sequence itself. Before his time, Indian mathematicians like Virahanka, Pingala, and Gopāla had already described similar sequences. However, Fibonacci's contribution lies in popularizing and disseminating the sequence to a broader audience.

The encounter between Fibonacci and the sequence occurred during his studies on a problem related to the breeding of rabbits. This conundrum, often referred to as the "rabbit problem" or the "problem of the growth of a population," aimed to determine the number of rabbits that would be born in a year under specific conditions. It was through this investigation that Fibonacci derived the sequence of numbers now known as the Fibonacci sequence.

In his seminal work, "Liber Abaci," Fibonacci presented the sequence along with its intriguing properties. He explored its recurrence relation, which defines each term as the sum of the two preceding terms. Additionally, he showcased the practical applications of the sequence, such as its relevance in interest rate calculations, currency exchange, and other mathematical quandaries.

Although Fibonacci may not have comprehended the full extent of the sequence's significance and diverse applications during his time, his work laid the foundation for the widespread recognition and popularity of the Fibonacci sequence. Subsequent mathematicians, scientists, and enthusiasts have since delved deeper into its properties and applications, propelling its significance in various fields of study, including mathematics, nature, and art.

Application for Fibonacci Sequence

The Fibonacci sequence, with its fascinating patterns and inherent mathematical beauty, finds applications in a multitude of domains. Let's explore some of the areas where the Fibonacci sequence is utilized and appreciated.

Mathematics: In mathematics, the Fibonacci sequence serves as an example of a recursive sequence. It has been extensively studied for its unique properties and connections to other mathematical concepts. The sequence is often used to introduce recursion, mathematical induction, and number patterns in educational settings.

Nature and Biology: The Fibonacci sequence can be observed in numerous natural phenomena, showcasing its presence in the living world. It appears in the branching patterns of trees, the arrangement of leaves on stems, and the spirals of flowers and pinecones. The sequence even manifests in the growth patterns of seashells and the breeding patterns of certain animal populations.

Art and Design: Artists, designers, and architects draw inspiration from the Fibonacci sequence to create aesthetically pleasing compositions. The ratios derived from consecutive Fibonacci numbers, such as the golden ratio (approximately 1.618), are considered visually harmonious and are often utilized to create balanced and visually appealing designs in various art forms.

Technical Analysis and Finance: In the realm of finance, the Fibonacci sequence finds application in technical analysis. Traders and analysts use Fibonacci retracement levels, derived from ratios within the sequence, to identify potential support and resistance levels in price charts. This technique aids in making informed decisions regarding market trends and price movements.

Computer Science: The Fibonacci sequence has relevance in computer science and algorithms. It serves as a basis for understanding and implementing recursive algorithms, dynamic programming, and efficient coding practices. The sequence is also encountered in various algorithms related to searching, sorting, and optimization problems.

Music and Rhythm: Musicians and composers have found inspiration in the Fibonacci sequence to create melodically pleasing compositions. The ratios and patterns derived from the sequence are employed to establish rhythmic structures and harmonies that are pleasing to the human ear.

These are just a few examples of the wide-ranging applications and influence of the Fibonacci sequence. Its intrinsic allure continues to captivate researchers, artists, and enthusiasts, reinforcing its status as a fascinating mathematical concept with numerous real-world implications.

The definition of Fibonacci's Sequence

The Fibonacci sequence is defined by the sum of the two previous sequence numbers with the initial condition that the First two numbers are 0 and 1.

The Fibonacci sequence starts with 0 and 1, and each subsequent number is obtained by adding the two numbers before it. Therefore, the Fibonacci sequence begins as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on.

Mathematically, the Fibonacci sequence can be defined using the recurrence relation:

$$F(n) = F(n-1) + F(n-2) \qquad (1)$$

where F(n) represents the n^{th} term in the sequence, F(n-1) represents the $(n-1)^{th}$ term, and F(n-2) represents the $(n-2)^{th}$ term. By convention, F(0) is usually taken as 0, and F(1) is 1, which serves as the starting point for the sequence. In a 32-bit environment you hid the limit after the 47th Fibonacci number while in a 64-bit environment, the limit without overflowing is the 93rd Fibonacci number.

The Fibonacci Sequence and the Golden Ratio

The connection between the Fibonacci sequence and the golden ratio is a significant and fascinating aspect of mathematics. The golden ratio, often denoted by the Greek letter phi (φ), is approximately equal to 1.6180339887.

The relationship between the Fibonacci sequence and the golden ratio arises from the ratios of consecutive Fibonacci numbers. As the Fibonacci sequence progresses, if we take the ratio of each term to its predecessor, the values approach the golden ratio. More precisely, the ratio of consecutive Fibonacci numbers tends to converge to the golden ratio as the sequence grows larger.

For example, if we divide a Fibonacci number by its preceding number, such as 5 divided by 3 or 89 divided by 55, the ratios approximate the golden ratio. As we take larger Fibonacci numbers, the ratios become increasingly closer to the value of φ.

This connection between the Fibonacci sequence and the golden ratio is mathematically expressed as follows:

$$\lim_{n\to\infty} \frac{F(n+1)}{F(n)} = \varphi \qquad (2)$$

where $\lim_{n\to\infty}$ denotes the limit as n approaches infinity, F(n) represents the n^{th} Fibonacci number, and φ represents the golden ratio.

The golden ratio has significant geometric and aesthetic properties. It is often associated with harmonious proportions and is found in numerous art forms, architecture, and nature. The proportions of the golden ratio are visually pleasing and can be seen in famous works of art, such as the Parthenon in Athens and Leonardo da Vinci's paintings.

The connection between the Fibonacci sequence and the golden ratio adds an intriguing layer of mathematical beauty and symmetry to both concepts. It highlights the inherent elegance and interplay between numbers, patterns, and aesthetics, captivating mathematicians, artists, and thinkers throughout history.

The simple computational approach of the Fibonacci sequence

The definition of the Fibonacci sequence in (1) lent itself to an implementation using recursive function calls as outlined below.

```
// Recursive Fibonacci sequence
// F(n) = F(n-1) + F(n-2)
static uintmax_t fibonacci(const uintmax_t n)
{
	if (n <=1) return n;// Base case for Fibonacci sequence
	return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Very simple and self-explanatory recursive implementation of the Fibonacci sequence. While it correctly calculates the Fibonacci numbers, it has certain pros and cons to consider:

Pros:

- *Simplicity:* The recursive implementation follows the mathematical definition of the Fibonacci sequence closely, making it easy to understand and implement.
- *Readability:* The code reflects the recurrence relation of the Fibonacci sequence (F(n) = F(n-1) + F(n-2)), which can aid in understanding the logic behind the sequence.
- *Accuracy:* The implementation accurately computes the Fibonacci numbers for a given input.

Cons:

- *Exponential Time Complexity:* The recursive approach has exponential time complexity. Each call to the fibonacci function results in two additional recursive calls, leading to a significant number of redundant computations. As a result, the time taken to compute larger Fibonacci numbers grows rapidly, making it inefficient for large inputs. This inefficiency is due to the repeated calculations of the same Fibonacci numbers multiple times.
- *Redundant Calculations:* As mentioned above, the recursive implementation recalculates Fibonacci numbers multiple times. For example, when calculating fibonacci (n), it may recursively calculate fibonacci (n-1) and fibonacci(n-2), and each of these recursive calls will further calculate their respective subproblems, resulting in redundant calculations.
- *Stack Overflow:* The recursive implementation is prone to stack overflow errors for larger values of n. With each recursive call, additional function calls are added to the call stack, and if the stack limit is reached, it can cause a stack overflow error.

To address these cons, one can consider using alternative approaches, such as iterative methods or memoization. Iterative methods avoid redundant calculations and have linear time complexity, while memoization stores previously computed Fibonacci numbers to avoid duplicate calculations, significantly improving efficiency.

Our first improvement is to memorize the previously calculated value of Fibonacci which leads us to this source. We used the unordered_map library to store the previously calculated value of Fibonacci.

```
#include <unordered_map>
static std::unordered_map<uintmax_t, uintmax_t> memo;

static uintmax_t fibonacciRecursiveOptimized(const uintmax_t n)
{
	if (n <= 1) return n;  // Base case for Fibonacci sequence

	if (memo.count(n) == 0)
	memo[n] = fibonacciRecursiveOptimized(n - 1) +
	          fibonacciRecursiveOptimized(n - 2);

	return memo[n];
}
```

This approach can address some of the limitations of the previous recursive implementation. Let's discuss the pros and cons of this optimized implementation:

Pros:

- *Improved Time Complexity:* By memorizing previously computed Fibonacci numbers, redundant calculations are eliminated. This leads to a significant improvement in the time complexity compared to the previous recursive implementation. The time complexity becomes linear (O(n)), as each Fibonacci number is computed only once.
- *Increased Efficiency:* The use of memoization reduces the number of recursive function calls, resulting in improved efficiency for larger values of n. This is because the function retrieves the precomputed Fibonacci numbers from the memoization table, rather than recalculating them.
- *Scalability:* The optimized implementation allows for the computation of larger Fibonacci numbers within a reasonable time frame, thanks to the elimination of redundant calculations and the linear time complexity.

Cons:

- *Space Complexity:* The optimized implementation utilizes additional memory to store the memoization table, which can consume memory proportional to the input value n. For very large values of n, this can result in a significant memory footprint. However, the space complexity remains reasonable compared to the exponential space complexity of the previous recursive implementation.
- *Potential Hash Collisions:* If the input value n becomes extremely large, there is a slight possibility of hash collisions in the unordered_map used for memoization. However, this is unlikely to occur in practice and can typically be addressed by using a larger hash table or alternative data structures.

Below is a table that lists the number of recursive calls made in these two implementations

The first x number of Fibonacci value	Recursive call without memoization	Recursive call with memoization
10	177	19
20	21,891	39
30	2,692,537	59
40	331,160,281	79
50	40,730,022,147	99

As can be seen from the table in the first implementation the number of recursive calls goes out of hand and becomes the limitation for making any practical computation of the Fibonacci sequence.

The optimized implementation with memoization significantly improves the efficiency and time complexity of computing Fibonacci numbers. It provides a scalable solution for larger inputs while utilizing additional memory resources for storing the memoization table.

However, we do have a faster alternative in our first simple approaches to a computationally efficient method and that is to use looping instead of recursion. With looping we eliminate the issues with excessive recursion but also the need to memorize all previous numbers. The loop-based source I listed below.

```
// Fibonacci loop based
static uintmax_t fibonacci_loop(const uintmax_t n)
{
      if (n <= 1) return n;
      uintmax_t previous = 0, current = 1, i;
      for (i = 2; i <= n; ++i)
      {
            uintmax_t tmp = current + previous;
            previous = current;
            current = tmp;
      }
      return current;
}
```

In this approach, we only need to keep the last two numbers to compute the next sequence.

The loop-based implementation of the Fibonacci sequence offers a different approach to calculating the Fibonacci numbers. Let's again explore the pros and cons of this implementation:

Pros:

- *Improved Time Complexity:* The loop-based implementation has a linear time complexity of O(n), making it more efficient than the previous recursive implementations. It computes each Fibonacci number iteratively without redundant calculations, resulting in faster execution for larger values of n.
- *Reduced Space Complexity:* The loop-based implementation does not require additional data structures or memorization tables. It only utilizes a few variables to store the current and previous Fibonacci numbers, leading to lower space complexity compared to the memorization-based approach.

Cons:

- *Lack of Readability:* The loop-based implementation may be slightly more challenging to understand for individuals unfamiliar with iterative programming constructs. The calculation of the Fibonacci numbers is done within the loop using temporary variables, which can make the code less intuitive compared to the recursive or memorized versions.
- *Limited Precision:* Depending on the data type used (uintmax_t in this case, typically 64-bit), the loop-based implementation may encounter limitations in representing extremely large Fibonacci numbers accurately. If the Fibonacci numbers exceed the maximum value that can be stored in the chosen data type, the results may be incorrect or wrap around, leading to inaccuracies.

These cons are a little bit far fetching for such a small function. Readability is not an issue and with limited precision, you do have an issue since a 64-bit data type can only hold the compute the first 93 Fibonacci numbers without overflowing, emphasizing the need to have an implementation that utilizes arbitrary precision arithmetic.

The loop-based implementation offers a computationally efficient and straightforward approach to calculating Fibonacci numbers. It avoids the recursive overhead and redundant calculations present in the previous recursive implementations.

The more advanced computation of the Fibonacci sequence.

All of the previous methods required that we start from scratch and then continue finding the next term until we have reached the Fibonacci number we are looking for. For small numbers this approach is OK and the fastest one. However, if we want to find the Fibonacci number for large values of n then this approach does not work any longer. Instead, we can use a technique called matrix exponentiation or the fast-doubling method to find the number.

Matrix Exponentiation

The method consists of raising a particular matrix to a power of n. If we take the base matrix M = $\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$ and raise it to the power of n (in other words calculate M^n, then we get the (n+1)th Fibonacci number as the element at row and column [0, 0] in the resultant matrix. You can generalize the elements as follows.

$$\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^n = \begin{bmatrix}F_{n+1} & F_n\\F_n & F_{n-1}\end{bmatrix} \quad (3)$$

Now this can easily be implemented efficiently using recursion. As outlined in the below algorithm.

```
Function F(n)

M=$\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$
If n is even
```

```
        k = n/2:
        HalfPower=$M^k$
        F(n)=  HalfPower*HalfPower
If n is odd
        k = (n - 1)/2:
        HalfPower=$M^k$
        F(n)=  M*HalfPower*HalfPower
```

Algorithm 1 Matrix exponentiation algorithm for Fibonacci number

The matrix exponentiation method for calculating Fibonacci numbers offers several advantages and some potential drawbacks. Let's explore the pros and cons of this approach:

Pros:

- *Improved Efficiency:* The matrix doubling method provides a significantly faster computation of Fibonacci numbers compared to the recursive or iterative approaches. It achieves a time complexity of O(log n), where n is the desired Fibonacci number. This makes it highly efficient for large values of n, as it avoids redundant calculations and reduces the number of multiplications.
- *Elimination of Redundant Calculations:* The matrix doubling method utilizes matrix exponentiation to raise the base matrix to the power of n-1. By utilizing exponentiation by squaring, it eliminates the need for repetitive calculations of intermediate Fibonacci numbers. This results in a substantial reduction in computational steps.
- *Scalability:* The matrix doubling method is well-suited for computing Fibonacci numbers with very large values of n. It can handle extremely large Fibonacci numbers efficiently due to its logarithmic time complexity, enabling calculations that are infeasible with other methods.

Cons:

- *Increased Complexity:* The matrix exponentiation method introduces additional complexity compared to simple recursive or iterative approaches. It requires the implementation of matrix multiplication and matrix exponentiation algorithms, which can be more challenging to understand and implement correctly.
- *Memory Usage:* The matrix doubling method requires storing and manipulating matrices during the computation. This can consume more memory compared to other methods that only require storing a few variables. However, memory usage is still reasonable and typically not a significant concern unless huge Fibonacci numbers are involved.
- *Precision Limitations:* Depending on the chosen data type and the size of Fibonacci numbers being computed, the matrix doubling method may encounter limitations in accurately representing extremely large Fibonacci numbers. If the chosen data type lacks sufficient precision or the numbers exceed their representable range, the results may become inaccurate or overflow.

Although the matrix exponentiation method offers substantial advantages in terms of efficiency and scalability for calculating Fibonacci numbers it does first show its real advantage for a much higher number of the Fibonacci sequence that can only be computed using higher data types than the 64-bit limit in most system. This means we have to turn to arbitrary precision before this method will have any advantages over the others.

```
// Fibonacci exponentiation formula to calculate the nth Fibonacci number
static uintmax_t fibonacci_exponentiation(const uintmax_t n)
{
	if (n <= 1)
		return n;  // Base case for Fibonacci sequence

	// Lambda for matrix multiplication of two 2x2 matrix
	auto matrixMultiply = [](const std::vector<std::vector<uintmax_t>>& A,
	                         const std::vector<std::vector<uintmax_t>>& B)
	{
		std::vector<std::vector<uintmax_t>> result(2, std::vector<uintmax_t>(2,
0));

		// Unroll the loop for a fixed-size 2x2 matrix
		result[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
		result[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
		result[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
		result[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];

		return result;
	};

	// Exponentiate matrix M to the power of n
	// We need to use the std::function to be able to call the lambda
	// function recursively.
	std::function<std::vector<std::vector<uintmax_t>>(const
std::vector<std::vector<uintmax_t>>&,const uintmax_t)> matrixPower = [&](const
std::vector<std::vector<uintmax_t>>& M, const uintmax_t n)
	{
		if (n == 0)
			// Identity matrix
			return std::vector<std::vector<uintmax_t>> { {1, 0}, {0, 1} };

		if (n == 1)
			return M;

		std::vector<std::vector<uintmax_t>> halfPower;
		if (n % 2 == 0)
		{
			halfPower = matrixPower(M, n / 2);
			return matrixMultiply(halfPower, halfPower);
		}
		else
		{
			halfPower = matrixPower(M, (n - 1) / 2);
			return matrixMultiply(matrixMultiply(halfPower, halfPower), M);
		}
	};

	const std::vector<std::vector<uintmax_t>> baseMatrix = { {1, 1}, {1, 0} };
	std::vector<std::vector<uintmax_t>> resultMatrix = matrixPower(baseMatrix, n -
1);
```

```
	return resultMatrix[0][0];  // Return the Fibonacci number from [0][0]
}
```

We can improve the method further by exploiting some symmetry and relationship between the matrix elements.

$$\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^n = \begin{bmatrix}F_{n+1} & F_n\\F_n & F_{n-1}\end{bmatrix} = \begin{bmatrix}F_{n+1} & F_n\\F_n & F_{n+1}-F_n\end{bmatrix} \quad (4)$$

We notice that with the information in the first row, we can create the second row if needed. This means that we don't need to carry around a 2x2 matrix but can settle for a vector with the two elements.

$$[F_{n+1,}F_n]$$

And then carry out the matrix multiplication to achieve the same result as for a real 2x2 matrix multiplication. Normally when carrying out a 2x2 matrix multiplication it involves 8 multiplication and 4 additions. With this new approach, we get 4 multiplication, 2 addition, and one subtraction. Nearly a ~42% reduction in operations plus we only need to handle a vector of 2 elements instead of a matrix with 4 elements.

The revised source is.

```
static uintmax_t fibonacci_exponentiationOptimized(const uintmax_t n)
{
	if (n <= 1)
		return n;  // Base case for Fibonacci sequence

	// Lambda for matrix multiplication of two 2x2 matrix
	// However, this is not a general matrix multiplication but is optimized ‘
	// with the knowledge that it arises as a power of the base matrix {{1,1},{1,0}}.
	// Therefore, we can exploit some symmetry eliminating 4 multiplications out
	// of 8 multiplication and one addition out of four
	// Instead of a 2x2 matrix we only need the first row as a vector
	auto Multiply = [](const std::vector<uintmax_t>& A, const std::vector<uintmax_t>&
B)
	{
		std::vector<uintmax_t> result(2);

		//Multiply a fixed-size 2x2 matrix disguised as two vectors of the first
rows
		result[0] = A[0] * B[0] + A[1] * B[1];
		result[1] = A[0] * B[1] + A[1] * (B[0] - B[1]);
		return result;
	};

	// Exponentiate matrix M to the power of n
	// We need to use the std::function to be able to call the lambda
	// function recursively.
	std::function<std::vector<uintmax_t>(const std::vector<uintmax_t>&, const
uintmax_t)> matrixPower = [&](const std::vector<uintmax_t>& M, const uintmax_t n)
	{
		if (n == 0)
```

```
                // Identity matrix
                return std::vector<uintmax_t> {1, 0};

        if (n == 1)
                return M;

        std::vector<uintmax_t> halfPower;
        if (n % 2 == 0)
        {
                halfPower = matrixPower(M, n / 2);
                return Multiply(halfPower, halfPower);
        }
        else
        {
                halfPower = matrixPower(M, (n - 1) / 2);
                return Multiply(Multiply(halfPower, halfPower), M);
        }
    };

    const std::vector<uintmax_t> baseMatrix = {1, 1};
    std::vector<uintmax_t> resultMatrix = matrixPower(baseMatrix, n - 1);

    return resultMatrix[0];  // Return the Fibonacci number from resultMatrix[0][1]
}
```

Fast doubling method

Another fast method is derived from the matrix exponentiation method and is called the fast-doubling method using the recursion below.

```
Function F(n)
If n is even

        k = n/2:
        F(n) = [2*F(k-1) + F(k)]*F(k)

If n is odd

        k = (n + 1)/2:
        F(n) = F(k)*F(k) + F(k-1)*F(k-1)
```

Algorithm 2. Fast doubling algorithm

This optimized recursive method for calculating Fibonacci numbers, also known as the "Fast-Doubling" method, has its own set of advantages and disadvantages compared to other approaches like naive recursion, iterative methods, and matrix exponentiation.

Pros:

- *Efficient Time Complexity:* The "Fast Doubling" method has a time complexity of O(log n), which is significantly faster than the naive recursive approach with a time complexity of O(2^n) and even faster than the iterative methods with a time complexity of O(n).

- *Avoids Redundant computation:* If the method uses memoization, caching previously computed Fibonacci numbers, to avoid redundant computations. This reduces the number of recursive calls and improves the performance, especially for larger values of n.
- *Simple Implementation:* The recursive nature of the algorithm makes it easy to implement and understand, making the code concise and readable.
- *No Precision Issues:* Unlike some other methods that rely on floating-point arithmetic, the "Fast Doubling" method works with integers, eliminating potential precision issues, but still subject to the limitation of the integer data type.
- *No Memory Overhead:* The memory overhead is relatively low compared to matrix exponentiation-based methods, as it does not require storing large matrices.

Cons:

- *Recursive Stack Limit:* For very large values of n, the recursive nature of the method might cause a stack overflow due to the limitation of the call stack size. Although memoization helps avoid redundant calls, extremely large Fibonacci numbers can still lead to stack overflow issues.
- *Space Complexity:* The method uses memoization to store previously computed Fibonacci numbers, which increases the space complexity to O(n). This can be a concern for calculating Fibonacci numbers for very large values of n, consuming a significant amount of memory.
- *Precision Issues with Large Integers:* For Fibonacci numbers with extremely large values (larger than the maximum representable integer), the method may face precision issues, especially on 64-bit systems.
- *Integer Overflow:* Despite using integers, if the calculated Fibonacci number exceeds the maximum representable value for the integer type used, it may cause an integer overflow and lead to incorrect results.
- *Complexity in Multi-Threaded Environments:* The recursive nature of the method may introduce complexities when parallelizing computations in multi-threaded environments.

The optimized recursive "Fast Doubling" method is an efficient way to calculate Fibonacci numbers with a time complexity of O(log n). However, it may suffer from recursion depth and memory overhead issues for extremely large values of n. Depending on the specific use case and constraints, this method can be a good choice for most practical scenarios, especially when the Fibonacci numbers to be calculated are within the representable range of the integer data type being used. For even more efficient computations, matrix-based methods like matrix exponentiation can be considered, which have a lower space complexity but require more sophisticated implementation.

Fast doubling without using memoization.

```
//F(n) = [2 * F(k - 1) + F(k)] * F(k)
//F(n) = F(k) * F(k) + F(k - 1) * F(k - 1)
// Without Memorization
static uintmax_t fibonacciRecursiveOptimized2A(uintmax_t n)
{
	if (n <= 1)
		return n;

	uintmax_t k = (n % 2 == 0) ? n / 2 : (n + 1) / 2;
	uintmax_t fk = fibonacciRecursiveOptimized2A(k);
	uintmax_t fkMinus1 = fibonacciRecursiveOptimized2A(k - 1);

	uintmax_t result;
	if (n % 2 == 0)
		result = (2 * fkMinus1 + fk) * fk;

	else
		result = fk * fk + fkMinus1 * fkMinus1;

	return result;
}
```

Now since we are repeatedly recursive using both f(k) and f(k-1) we could see if we can gain any advantage using memorization as in the source below.

```
//F(n) = [2 * F(k - 1) + F(k)] * F(k)
//F(n) = F(k) * F(k) + F(k - 1) * F(k - 1)
// with memorization
#include <unordered_map>
static std::unordered_map<uintmax_t, uintmax_t> memo;

static uintmax_t fibonacciRecursiveOptimized2B(uintmax_t n)
{
	if (n <= 1)
		return n;

	if (memo.count(n) > 0)
		return memo[n];

	uintmax_t k = (n % 2 == 0) ? n / 2 : (n + 1) / 2;
	uintmax_t fk = fibonacciRecursiveOptimized2B(k);
	uintmax_t fkMinus1 = fibonacciRecursiveOptimized2B(k - 1);

	uintmax_t result;
	if (n % 2 == 0)
		result = (2 * fkMinus1 + fk) * fk;

	else
		result = fk * fk + fkMinus1 * fkMinus1;

	memo[n] = result;
	return result;
}
```

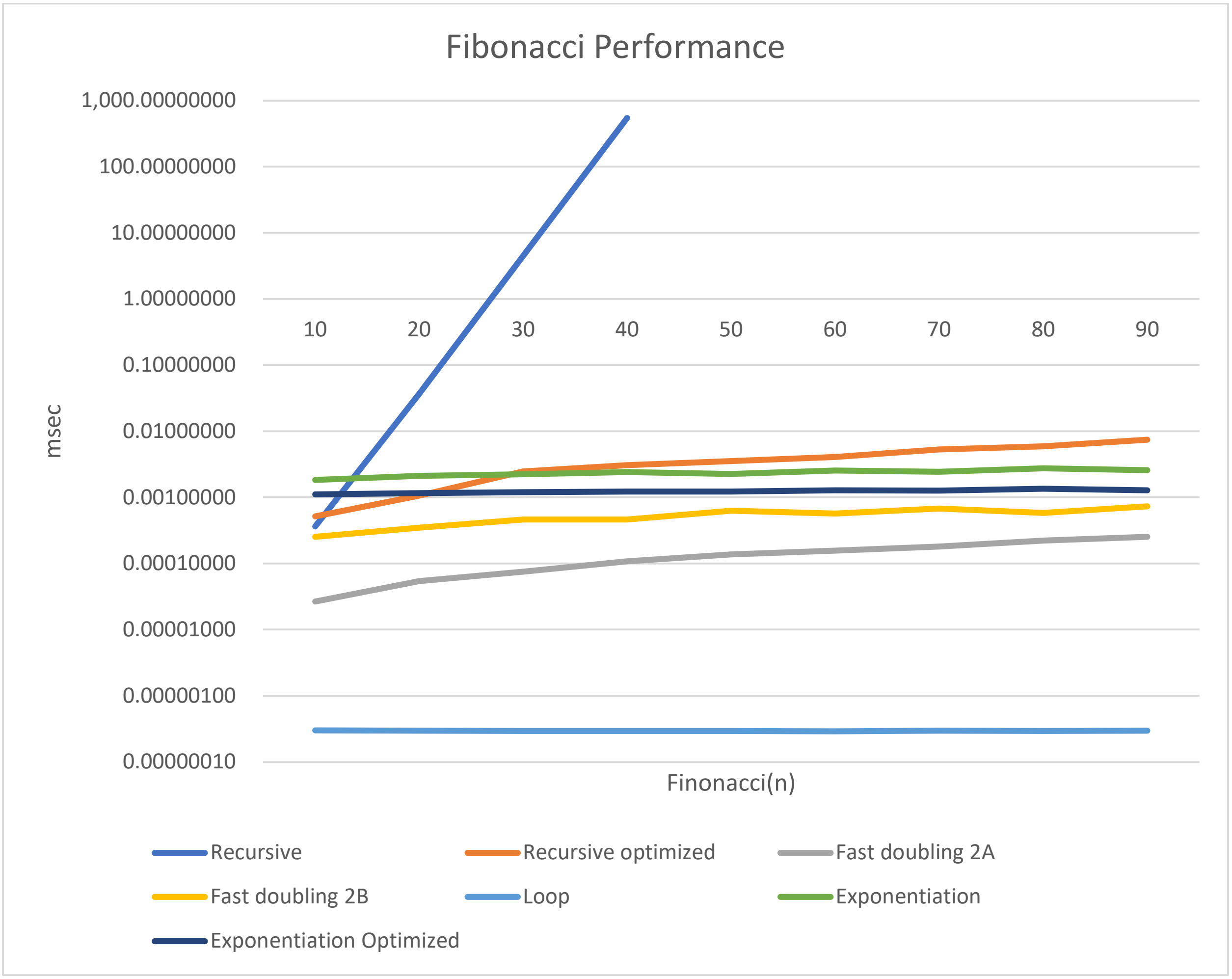
Memorization does reduce the number of recursive calls significantly.

The first x number of Fibonacci value	Recursive call without memoization	Recursive call with memoization
10	19	11
20	39	15
30	45	19
40	63	19
50	81	25

60	91	23
70	107	27
80	127	23
90	147	29

Whether we can gain any advantage from using memoization is discussed in the performance of the Fibonacci methods below.

Performance of the methods in a 64-bit environment



The use of 64-bit arithmetic sets limits on how high the sequence of the Fibonacci number we can generate within the 64-bit space. As indicated in the above figure we time the performance of generated the first 10, 20, …, up to 90 (Maximum is 93 in a 64-bit environment).
As can be seen, the use of a pure recursion has an exponential slow behavior as we increase the Fibonacci number, and above 40-50 it becomes useless for any practical purpose.
Recursion optimized is the standard recursion using memoization and it scales a lot better than the original version but is the slowest among the other methods. In general, we can't recommend

the use of these two recursive methods.
The matrix exponentiation is faster but due to the complexity of using matrix operations, it is not the fastest method for Fibonacci numbers below 90. The fast-doubling method is the most efficient but not the fastest either. Interestingly enough, the use of memoization is not faster than not using memoization. This is properly because we are still dealing with small numbers of the Fibonacci sequence and the memoization does not kick into gears at these low Fibonacci numbers.

Interestingly enough the brute force method is the fastest of them all in a 64-bit environment.

Approximation of the Fibonacci sequence using Binet's formula

Binet's formula for the Fibonacci sequence is.

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

Were

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803 \ldots \qquad (6)$$

φ is the golden ratio. And

$$\psi = \frac{1-\sqrt{5}}{2} \qquad (7)$$

Usually, you don't implement the algorithm using this formula, but instead observe that $\frac{\psi^n}{\sqrt{5}} < \frac{1}{2}$ for all n greater or equal to zero and then you can simplify the formula to the nearest integer to $\frac{\varphi^n}{\sqrt{5}}$.

$$F_n = \frac{\varphi^n}{\sqrt{5}} \qquad (8)$$

However, the accuracy of the computation depends on the accuracy of the underlying IEEE754 arithmetic implemented in most systems. For 64-bit IEEE754 floating-point you get an accurate result up to F_{70} whereafter rounding errors make the result inaccurate.

```
// Binet's Fibonacci approximation formula
static uintmax_t fibonacci_binet(const intmax_t n)
{
	if (n <= 1) return n; // Fibonacci base cases
	double phi = (1.0 + sqrt(5.0) ) / 2;
	return uintmax_t(round(pow(phi, n) / sqrt(5)));
}
```

Recommendation for Fibonacci methods in a 64-bit environment.

Since the simple loop is faster compared to the other methods and the simplest to implement, I recommend using that method for the valid range of the Fibonacci sequence between 0 and 93.

Fibonacci number in arbitrary precision

As the max limit is the 93rd Fibonacci number in a 64-bit environment you quickly need to go to arbitrary precision to compute a higher number of the Fibonacci sequence. Now based on the previous performance figure we can discard the least efficient method and look at these three methods and how they scaled with higher Fibonacci numbers. The four interesting methods are

1. The brute force loop-based, method
2. The Matrix exponentiation method
3. The fast-doubling method
4. The Binet's direct formula if we can get it to scale well and maintain the accuracy of the result

It is relatively easy to convert the first 3 methods to arbitrary precision where you replace the *uintmax_t* (64-bit unsigned integer) to *int_precision* which is the data type for the arbitrary precision integers in [1].

Source Fibonacci loop-based method.

```
// Fibonacci loop based in arbitrary precision
static int_precision fibonacci_loop(const int_precision& n)
{
	const int_precision c1(1);
	if (n <= c1) return n;     // Fibonacci Base cases

	int_precision previous(0), current(1), i(2);
	for (; i <= n; ++i)
	{
		int_precision tmp = current + previous;
		previous = current;
		current = tmp;
	}
	return current;
}
```

Source Fibonacci Optimized Matrix exponentiation method.

```
// Fibonacci doubling formula optimized to calculate the nth Fibonacci number
// Instead of a 2x2 matrix we only need the info from the first row and therefore it is implemented as a vector with two elements
// This speeds up the calculation with a factor of 2 to 3 times.
static int_precision fibonacci_exponentiationOptimized(const int_precision& n)
{
	const int_precision c1(1);
	if (n <= c1)
		return n;  // Base case for Fibonacci sequence

	// Lambda for matrix multiplication of two 2x2 matrix
	// However, this is not a general matrix multiplication but is optimized with
	// the knowledge that it arises as a power of the base matrix {{1,1},{1,0}}.
	// Therefore, we can exploit some symmetry
	// result[1][0]=result[0][1] and result[1][1]=result[0][0]-result[0][1]
	// eliminating 4 multiplications out of 8 multiplication and one addition
	// out of four
	// Instead of a 2x2 matrix we only need the first row as a vector
	auto Multiply = [](const std::vector<int_precision>& A, const
std::vector<int_precision>& B)
	{
		std::vector<int_precision> result(2);

		//Multiply a fixed-size 2x2 matrix disguised as two vectors of the first
rows
		result[0] = A[0] * B[0] + A[1] * B[1];
		result[1] = A[0] * B[1] + A[1] * (B[0] - B[1]);
		return result;
	};

	// Exponentiate matrix M to the power of n
	// We need to use the std::function to be able to call the lambda function
	// recursively.
	std::function<std::vector<int_precision>(const std::vector<int_precision>&, const
int_precision&)> matrixPower = [&](const std::vector<int_precision>& M, const
int_precision& n)
	{	const int_precision c1(1), c2(2);
		if (n.iszero() )
			// Identity matrix
			return std::vector<int_precision> {1, 0};

		if (n == c1)
			return M;

		std::vector<int_precision> halfPower;
		if ((n % c2).iszero())
		{
			halfPower = matrixPower(M, n / c2);
			return Multiply(halfPower, halfPower);
		}
		else
		{
			halfPower = matrixPower(M, (n - c1) / c2);
			return Multiply(Multiply(halfPower, halfPower), M);
		}
	};

	const std::vector<int_precision> baseMatrix = { 1, 1 };
	std::vector<int_precision> resultMatrix = matrixPower(baseMatrix, n - c1);

	return resultMatrix[0];  // Return the Fibonacci number from resultMatrix[0][1]
}
```

Source Fibonacci Optimized fast doubling method with memoization.

```
static std::unordered_map<uintmax_t, int_precision> ipmemo;

//F(n) = [2 * F(k - 1) + F(k)] * F(k)
//F(n) = F(k) * F(k) + F(k - 1) * F(k - 1)
static int_precision fibonacciRecursiveOptimized3B(int_precision& n)
{
	const int_precision c1(1), c2(2);
	if (n <= c1)
		return n;

	if (ipmemo.count(n) > 0)
		return ipmemo[n];

	int_precision k = ((n % 2).iszero()) ? n / c2 : (n + c1) / c2;
	int_precision fk = fibonacciRecursiveOptimized3B(k);
	int_precision fkMinus1 = fibonacciRecursiveOptimized3B(k - c1);

	int_precision result;
	if ((n % c2).iszero())
		result = (c2 * fkMinus1 + fk) * fk;
	else
		result = fk * fk + fkMinus1 * fkMinus1;

	ipmemo[uintmax_t(n)] = result;
	return result;
}
```

Binet’s formula in arbitrary precision

The drawback of Binet’s formula is the lack of accuracy with higher Fibonacci numbers due to IEEE754 64-bit fixed accuracy. Luckily, we can circumvent that issue using arbitrary precision floating point since we can dynamically increase the accuracy as needed for higher Fibonacci numbers. In [2] they give an approximation of the needed decimal accuracy as:

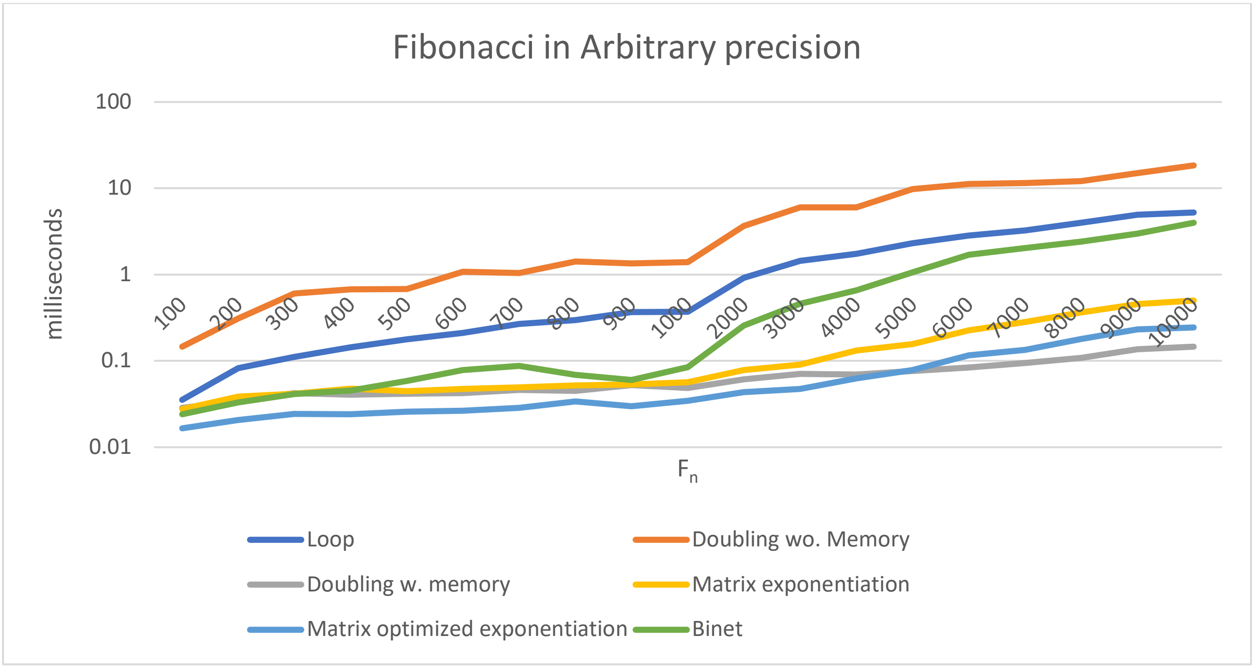
$$n log_{10}(\varphi) = 0.209n \qquad (9)$$

E.g. for F_{100} you need 0.209n=20.9decimal accuracy in your computation of the 100th Fibonacci number. We can therefore lay out the source code for the arbitrary precision version of Binet’s approximation formula.

```
// Binet's formula using arbitrary precision
static int_precision fibonacci_binetfp(const intmax_t n)
{
	if (n <= 1)   // Fibonacci Base cases
		return n;

	float_precision  phi, sq5(5);
	//Compute precision and add a few extra digits to be sure
	intmax_t prec = intmax_t(n * 0.209 + 4);
	if (prec < 20) prec = 20; // use a minimum of 20 decimal digits
	phi.precision(prec); sq5.precision(prec);	// Change the precision
	sq5 = sqrt(sq5);
	phi = (float_precision(1) + sq5) / float_precision(2);
	phi = pow(phi, float_precision(n)) / sq5;
	return int_precision(round(phi));
}
```

Performance using arbitrary precision



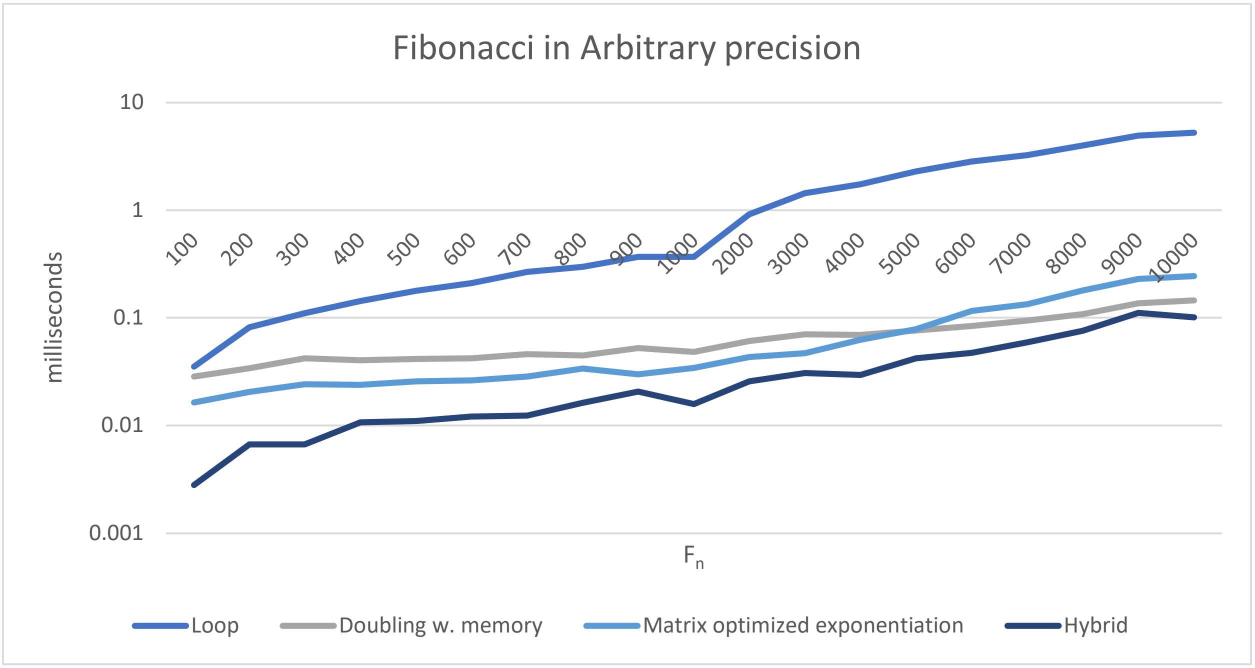
Interestingly enough the picture changes quite a bit while using arbitrary precision. The basic loop algorithm which was faster within a 64-bit environment fell short in arbitrary precision. The doubling algorithm with or without memoization performs the same in 64-bit it is dramatically different in arbitrary precision. The reason is of course that memoization helps reduce the number of recursive calls and also reduce the number of arbitrary precision operations. As we saw in the 64-bit environment matrix exponentiation the unoptimized performs poorly compared to the optimized version and the optimized version is the faster method up to around the first 5,500 Fibonacci numbers whereafter the doubling method with memoization takes over. Binets Fibonacci formula is an exciting method but the need for increased accuracy for higher Fibonacci numbers makes the method less efficient compared to the other methods.

Recommendation for Fibonacci in arbitrary precision

The two preferred methods are the doubling method with memoization and the Matrix exponentiation method. For the Fibonacci number between 100 and 5,500, the matrix method is faster but then the Fibonacci doubling method takes over. To balance the two methods, I recommend a hybrid approach that automatically switches between the most efficient method including the loop-based method that can be handled by 64-bit integers.

Our Final Hybrid version

Below is the final performance of our hybrid version that takes advantage of the 64-bit arithmetic as possible, with memoization and the fast-doubling algorithm.



Notice that the Hybrid algorithm performs much better for small to large Fibonacci numbers and then the asymptotic approach the Fast-doubling algorithm for very large Fibonacci numbers.

```
// Final hybrid version of the Fibonacci sequence F(n)
// If n <= 93 then use 64-bit arithmetic and the loop-based Fibonacci method
// Otherwise, we use the Fast-doubling method with memorization and call of
// 64-bit Fibonacci loop when needed
//
static int_precision fibonacci(int_precision& n)
{
	const int_precision c1(1),c2(2), limit64bit(93);

	auto loop = [&](const int_precision& n)
	{// Handle Fibonacci sequence up to 93 with is the limit in a 64-bit environment
		if (n <= c1) return n;// Handle Fibonacci base cases
		uintmax_t previous = 0, current = 1;
		for (int i = int(n); i >= 2; --i)
		{
			uintmax_t tmp = current + previous;
			previous = current;
			current = tmp;
		}
		return int_precision(current);
	};

	std::unordered_map<uintmax_t, int_precision> ipmemo; // only used in doubling
	std::function<int_precision(const int_precision&)> doubling = [&](const
int_precision& n)
	{
		if (ipmemo.count(n) > 0)
			return ipmemo[n];

		int_precision result;
		if (n <= limit64bit)
		{
			result = loop(n);
		}
		else
		{
			int_precision k = ((n % c2).iszero()) ? n / c2 : (n + c1) / c2;
			int_precision fk = doubling(k);
			int_precision fkMinus1 = doubling(k - c1);

			if ((n % c2).iszero())
				result = (c2 * fkMinus1 + fk) * fk;
			else
				result = fk * fk + fkMinus1 * fkMinus1;
		}
		ipmemo[uintmax_t(n)] = result;
		return result;
	};

	ipmemo.clear();
	return doubling(n);
}
```

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages
2) Wikipedia. Fibonacci sequence. Fibonacci sequence - Wikipedia