Implementing and Experimenting Shor's Factorization Algorithm

Eusebiu Volostiuc, Jonas Liley, Tom Sorger

January 2025

1 Introduction

In 1994, Peter Shor introduced an algorithm that marked a milestone in quantum computing and sparked interest worldwide in quantum algorithms with real-world applications. Shor's algorithm addresses one of the most challenging problems in computational mathematics: the factorization of large integers. [14, 13] This problem is computationally difficult for classical computers, especially as the integer's size increases, and it forms the backbone of modern cryptographic systems. RSA encryption, a widely used standard in secure communications, is based on the difficulty of factoring large composite numbers efficiently. [12, 16] Shor's algorithm provides a solution to this problem, significantly reducing the time complexity from exponential to polynomial time, thanks to the principles of quantum mechanics. [13]

The significance of Shor's algorithm extends beyond theoretical computing. Cryptography and information security rely on the assumption that factoring large integers, such as 2048-bit RSA keys, is infeasible within a practical timeframe for classical computers. [7] Shor's algorithm challenges this assumption, as research from 2021 predicts that a sufficiently powerful quantum computer could factor a 2048-bit RSA integer within hours with high success rates. [4] Although today's largest quantum computers still lack the qubit counts necessary to perform such a factorization, projections suggest rapid advancements in quantum hardware, hinting at the potential reality of breaking widely used encryption methods within the coming decades. [5, 2]

This project investigates Shor's factorization algorithm, aiming to translate the theoretical steps into a practical quantum implementation to factor smaller composite numbers, thereby offering hands-on insights into its strengths and limitations.

2 Background on Shor's Algorithm

Shor's algorithm represents one of the earliest and most impactful demonstrations of a quantum algorithm capable of solving a classically "hard" problem efficiently. At its core, Shor's algorithm solves the factorization problem by harnessing the principles of quantum superposition and interference. While classical computers must search for factors sequentially or exhaustively, Shor's algorithm utilizes quantum parallelism and the quantum Fourier transform to identify the period of a function related to modular exponentiation. Once the period is found, classical computation is used to derive the factors of the original number. [14, 13, 10]

Shor's algorithm relies on a mix of classical and quantum steps, often outlined in two primary components: the classical preprocessing and postprocessing steps and the quantum period-finding step. The preprocessing step consists of selecting a random integer that is coprime with the composite number N, which we wish to factor. If the chosen number has a non-trivial common divisor with N, that divisor is already a factor of N, ending the algorithm. Otherwise, the period-finding process is performed using a quantum circuit, which consists of applying modular exponentiation and the quantum Fourier transform. The result, ideally, yields the period r of a certain function, which allows us to deduce the factors of N using basic number-theoretic principles. [14]

The potential impact of Shor's algorithm on cryptography has driven substantial research in quantum computing. [9] Currently, building a quantum computer with enough stable qubits and error correction capabilities to run Shor's algorithm on large integers (such as 2048-bit numbers) remains a technological hurdle. Yet, theoretical and experimental advancements continue, with many industry leaders predicting significant increases in qubit counts in the coming years. [5] In fact, recent work by Gidney and Ekerå suggests that with an error-corrected, 20-million qubit machine, it would be possible to factor a 2048-bit RSA integer in just over five hours with near certainty, emphasizing the urgency for the cryptographic community to consider quantum-safe alternatives. [4]

3 Project Goal

The project will focus on developing a Shor Factorization Algorithm by embracing a quantum computing framework for functionally and effectively factoring integers. More precisely, in this work, the theoretical explanation of this algorithm will be discussed and an actual working of its practical implementation will be presented, which will be used to perform experiments with different composite numbers. A translation of this type of algorithm into workable code would yield insight into the problems and intricacies that accompany quantum programming and offer support for understanding its real-world implications. It will represent a practical application of quantum algorithms to solve real-life problems.

4 Methods and Tools

We will execute it using Python as the programming language, with IBM Qiskit as the main quantum computing framework to attain the project's objectives. Qiskit is an open-source quantum development environment that can create, simulate, and run quantum algorithms on quantum computers, and is the primarily used framework within the course. [3, 11] Trying to produce a neat structure, this implementation would therefore have segments for classical processing and others for quantum processing. Furthermore, we will highlight the synergy between these two programming paradigms.

5 Implementation

5.1 Theory

For trivial cases, such as when N is even or of the form p^q, or for when we find a random a for which $gcd(a, N) \neq 1$, the factors can be efficiently found classically.

We know that for a non-trivial N and a random $2 \leq a < N$ for which $gcd(a, N) = 1$, we can find r such that $a^r \equiv 1 \; mod \; N$. r is called in this case the multiplicative order. We can then compute from a^r a factor of N.

The main idea of Shor's algorithm was that, instead of computing $a^x \; mod \; N$ repeatedly until a previously-found value is reached, one could instead compute only a limited set of values and then use *Quantum Fourier Transformation* (QFT) to infer the period. The typical quantum circuit would thus look akin to Figure 1.

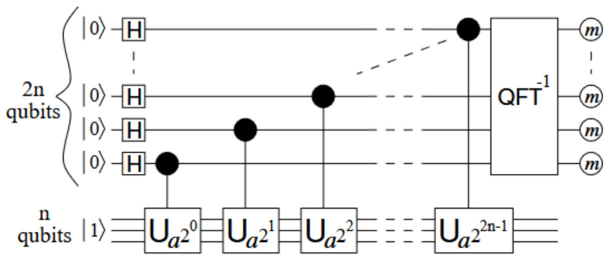


Figure 1: Shor's algorithm typical circuit. U represents the operator for multiplication modulo N.

In 2003, Stéphane Beauregard published a paper [1] in which he presents what he calls *"The one controlling-qubit trick"*. He showed that due to a number of quantum properties, QFT can be applied *semi-classically*: one can apply QFT on the controlling qubit after each multiplication, and thus simulate the QFT being applied on all the results. By doing this, the number of necessary qubits is greatly reduced. The optimized circuit is presented in Figure 2.

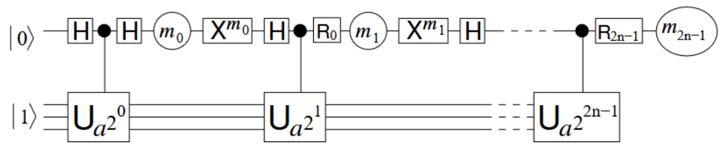
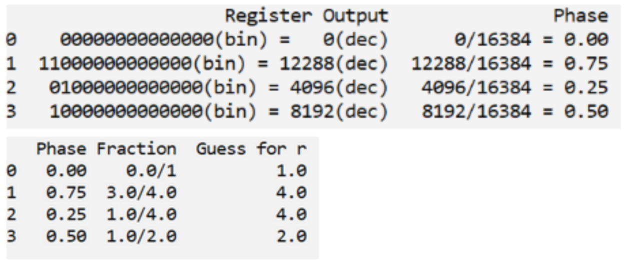


Figure 2: Shor's algorithm circuit with Beauregard optimization.

After obtaining the results of the QFT we have to interpret them to get the period r (QFT does not output the exact r). To do that, we make use of *continued fractions*. Using the "quantum mindset" of running the algorithm multiple times and selecting the result based on probabilities, we perform a number of iterations, transform the outputs to continued fractions, and select the relevant r, as shown in figure 3.


```
                  Register Output             Phase
0    00000000000000(bin) =     0(dec)     0/16384 = 0.00
1  11000000000000(bin) = 12288(dec)  12288/16384 = 0.75
2   01000000000000(bin) =  4096(dec)   4096/16384 = 0.25
3   10000000000000(bin) =  8192(dec)   8192/16384 = 0.50

   Phase Fraction  Guess for r
0   0.00     0.0/1          1.0
1   0.75   3.0/4.0          4.0
2   0.25   1.0/4.0          4.0
3   0.50   1.0/2.0          2.0
```

Figure 3: Sample output from the algorithm. The best candidate for r is 4.

In the code, we included the methods for solving the trivial cases as well, in order to have a more round-up program in the end.

5.2 Code

For our implementation we used *qiskit v1.2.4*, *qiskit-aer v0.15.1*, and *qiskit-ibm-runtime v0.33.2*. The complete code for this project will be published on GitHub[15]. While discussing we will only refer to function names and specific steps, indicated through comments in the code, within the main algorithm. We will also not discuss the entire code but just highlight important parts.

5.2.1 Supporting Functions

One important function is *check_for_simple_factors(N, isPrinting)*. As mentioned before, our code does not solve every factorization using Quantum Computing. The function checks for trivial factors of number N. For example, we check whether N is an even number. This would give us the trivial factors $p = 2$ and $q = N/p$. In the same manner we would get trivial factors if N is a prime number or a prime power. To facilitate it, we use common algorithms like the Miller-Rabin Primality Test or Sieve of Eratosthenes algorithm.

Another kind of supporting functions are all functions that help us implement the quantum circuit. We added various functions that implement various quantum operations using qiskit. For example, *mod_addr* and *inv_mod_addr* implement modular addition and subtraction operation of a value a modulus N using two control qubits. The modular addition of a value is an important step in computing the period of the function

$$f(x) = a^x \pmod N$$

which we need to factor the integer N. Similarly *c_mult* and *inv_c_mult(circuit, ctrl, q, aux, a, N, n)* implements the Controlled Multiplication gate for modular exponentiation. This gate performs a controlled modular multiplication of the quantum register q with a constant a, modulo N, based on the control qubit $ctrl$. It uses QFT and InvQFT to facilitate the modular arithmetic. A last interesting function is *gen_ua(circuit, ctrl, eigen_vec_reg, aux, a, N, n, isPrinting=True)*. It implements a general unitary operator Ua used in Shor's algorithm for modular exponentiation. It applies a series of the previously mentioned modular exponentiation gates, controlled swaps, and the modular inverse of the base a.

5.2.2 Main Algorithm

1. Screening N for trivial factors: The algorithm first checks for simple factors of N (e.g., even numbers or prime powers) using classical methods. If such factors are found, N is factored without proceeding further.

2. Calculating the number of qubits: The number of qubits required is determined based on N. For a composite number N, $\lceil \log_2 N \rceil$ qubits are needed for the quantum register.

3. Iterating: The algorithm runs for a specified number of iterations (10 for simulators, 1 for real devices) to find valid factors.

4. Setting up quantum registers: Quantum registers are initialized, including an auxiliary register, eigenvector register, and a counting register. The counting register may be optimized to use a single qubit, depending on whether optimization settings are used. If optimization is enabled, the previously mentioned Stéphane Beauregard [1] one controlling-qubit trick is used.

5. Creating the quantum circuit: The quantum circuit is constructed, with either the full counting register or the optimized one.

6. Initializing the eigenvector: The eigenvector register is initialized to the quantum state $|00\ldots01\rangle$.

7. Performing a classical guess: A random a is chosen, and the greatest common divisor (gcd) of a and N is computed. If $\gcd(a, N) > 1$, a factor of N is found classically.

8. Running the order-finding subroutine: If the classical step fails, the quantum phase estimation subroutine is executed to find the order r of a modulo N.
9. Encoding phase information: Phase information related to the eigenvalue of the unitary operator is encoded into the amplitudes of the counting register. This is done either using a serialized process or controlled modular exponentiation gates.
10. Setting up the backend: The quantum circuit is executed on a quantum simulator or real quantum device with an appropriate backend.
11. Analyzing the results: Measurement outcomes are processed to determine the corresponding eigenvalues and their phases.
12. Converting phases to fractions: Phases are converted into fractions using continued fractions to make guesses for the period r.
13. Verifying period and factoring N: Each candidate period r is checked. If r is even, it is used to compute potential factors of N. Valid factors are returned, or the algorithm declares failure if no factors are found.

6 Experimenting

This section will describe the experiments we are doing with the implemented Shor's Factorization Algorithm in section 5. All execution time experiments run on a device with the system details as described in Appendix A.

6.1 Execution Time Analysis

This experiment aims to evaluate the average runtime of our implementation of Shor's Algorithm across a range of inputs. The focus is on determining the computational efficiency of the algorithm in identifying the factors of odd composite numbers.

6.1.1 Methodology

To conduct this experiment, we developed a script that executes our implementation of Shor's Algorithm for various inputs. Specifically, the script profiles the execution time for each input and repeats the process multiple times[1] to compute an average runtime.

The inputs consist of all odd composite numbers between 1 and 1000. Even numbers are excluded because they represent trivial cases for the algorithm, as they can be easily factored by dividing by 2. We also didn't run the experiment for all of these inputs, this is because of the high execution times. Therefore, we let the script run for around ten hours before stopping it.

[1]We decided to repeat the profiling for every input five times, as the difference between the execution times didn't differ that much unless the algorithm did a lucky guess.

6.1.2 Results

Running this experiment gives us the initial values as seen in Figure 4.

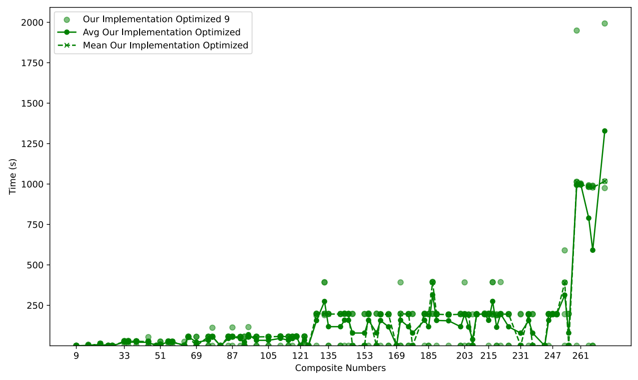


Figure 4: Execution Time of Our Code

We see some jumping values, this is because of the lucky guesses the algorithm can make as described earlier. To get a better picture of the actual execution times we removed the lucky guesses in Figure 5.

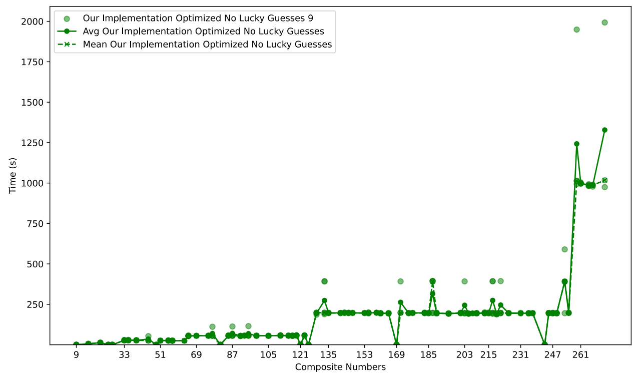


Figure 5: Execution Time of Our Code Without Lucky Guesses

In this figure we can now see very good that there are multiple steps of execution time:

- We slightly see the first step until an input value of 32 (2^5),
- The second step slightly from 33 until 64 (2^6),
- The third step from 65 until 128 (2^7),
- The fourth step from 129 until 256 (2^8), and

- The fifth step from 257 until the end of the graph.

This shows us that the runtime increases when the used input number uses more bits which makes the quantum circuit more complex and therefore also increases the execution time. The small dips and kinks in the graph can be explained as they follow the structure of p^q, a trivial case for the algorithm.

6.2 Execution Time Comparison

In this experiment, we compare the runtime performance of our implementation with the work of Rui Maia and Tiago Leão, whose implementation of Shor's Algorithm was adapted from for an older version of IBM's Qiskit implementation [8, 6]. Their implementation includes two variants: *Normal QFT* and *Sequential QFT*. [8]

Normal QFT This version implements Shor's Algorithm using a quantum circuit based on the first simplification from [1]. It employs the standard Quantum Fourier Transform (QFT) and features a top register with $2n$ qubits.

Sequential QFT This version implements Shor's Algorithm using a quantum circuit based on the second simplification from [1]. It uses the sequential Quantum Fourier Transform (QFT) with a top register of just one qubit, performing $2n$ measurements throughout the circuit. These measurements directly yield x_{final}, eliminating the need for a QFT at the circuit's end [8].

6.2.1 Methodology

We followed the same methodology as described in 6.1.1 for both variants and took the numbers of 6.1 to compare the runtime durations.

6.2.2 Results

Normal QFT Looking at the results for the implementation of the normal QFT in Figure 6, we recognized pretty fast that the implementation was not very good for even smaller numbers. Until the input number of 21 it was working fine; higher inputs let to an internal error always returning false results. We therefore cut down the graph to just include input numbers that worked for us.

We can see that the execution time for the input of $9 = 3^2$ is close to zero seconds as it's a trivial case, for other numbers like 15 or 21, that can be represented with the same amount of bits, also use around the same amount of time to be factored.

Sequential QFT Looking at the results for the implementation of the sequential QFT in Figure 7, we recognized a similar step pattern as in experiment 6.1, with certain steps. These steps are namely:

- We slightly see the first step until an input value of 16 (2^4),

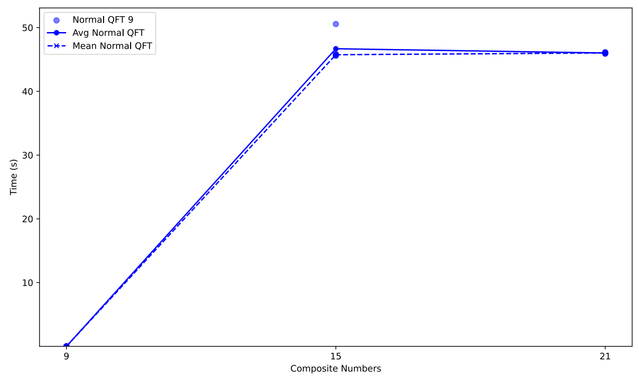


Figure 6: Implementation by Tiago Leão and Rui Maia – Normal QFT

- The second step from 17 until 32 (2^5),
- The third step from 33 until 64 (2^6), and
- The fourth step from 65 until the end of the graph.

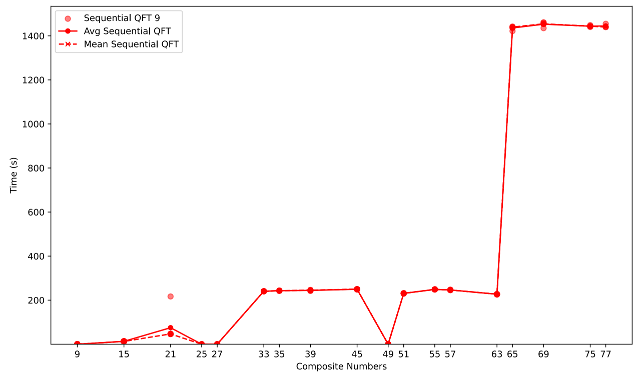


Figure 7: Implementation by Tiago Leão and Rui Maia – Sequential QFT

This also shows us that the runtime increases when the used input number uses more bits which makes the quantum circuit more complex and therefore also increases the execution time exactly like in experiment 6.1. The small dips and kinks in the graph can also be explained as they follow the structure of p^q, a trivial case for the algorithm that we can also see in experiment 6.1.

Comparing Normal and Sequential QFT A comparison of both implementations of Rui Maia and Tiago Leão can be seen in Figure 8.

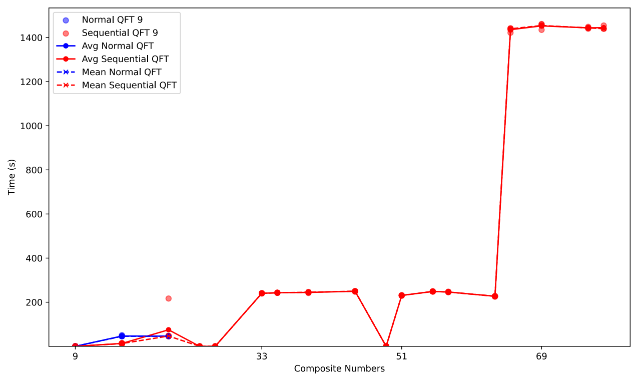


Figure 8: Implementation by Tiago Leão and Rui Maia – Comparing Normal and Sequential QFT.

We can directly see that the sequential QFT implementation is quite more effective than the normal QFT implementation, as we have way more data available. Also for the inputs both implementations have data for, we can see that the sequential implementation is more efficient.

Comparing Our Code, Normal, and Sequential QFT A comparison of our implementation and both implementations of Rui Maia and Tiago Leão can be seen in Figure 9.

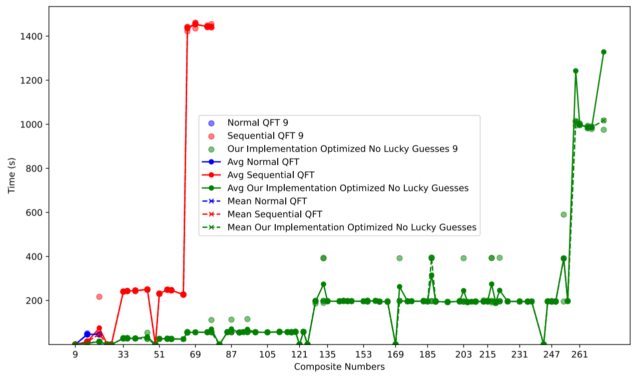


Figure 9: Comparing Our Code, Normal, and Sequential QFT

As described earlier, it does not make sense to compare the normal QFT implementation, therefore, we will just compare the sequential QFT implementation with our implementation. Also, as described earlier, we can see that both implementations have a step-like increase in runtime when the input value increases by their bit size. Comparing the execution times themselves, we can also see that our implementation is way faster than the sequential QFT implementation.

6.3 Quantum Circuit Optimization

In this experiment we want to figure out how much performance gains we achieve through the quantum circuit optimization, using the one controlling-qubit trick.

6.3.1 Methodology

We followed the same methodology as described in 6.1.1 and took the numbers of 6.1 to compare the runtime durations. The times of 6.1 already use the optimized quantum circuits. We therefore run the experiment now with not optimized quantum circuits.

6.3.2 Results

Looking at the results for the implementation of the sequential QFT in Figure 10, we see a lot of jumping values again because of the lucky guesses.

We therefore remove these lucky guesses from the graphic and end up with Figure 11.

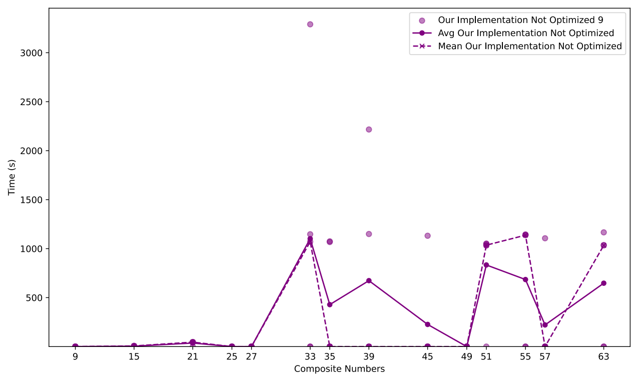


Figure 10: Results all

It might again be a bit hard to see, but we again have a certain step-like increment. These steps are namely:

- We slightly see the first step until an input value of 16 (2^4),
- The second step from 17 until 32 (2^5), and
- The third step from 33 until 64 (2^6).

This step-like increase would be visually better to detect when having the result for an input number bigger than 64. We run this implementation with 65 as input but even after multiple hours we didn't get a result.

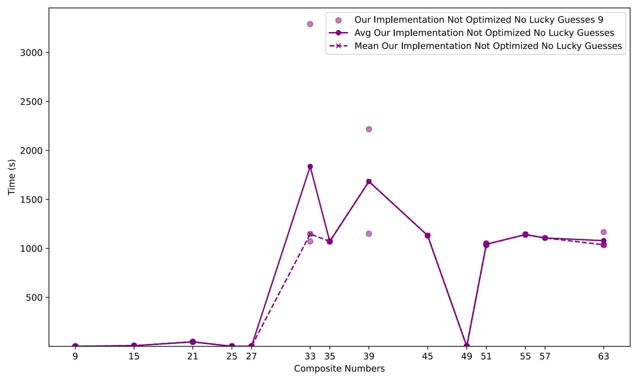


Figure 11: Results No lucky guesses

Compare Not Optimized and Optimized Quantum Circuit Implementation Comparing both, the optimized and the not optimized quantum circuit implementation in Figure 12, we can see that the implemented optimization actually improves the execution time of the algorithm a lot.

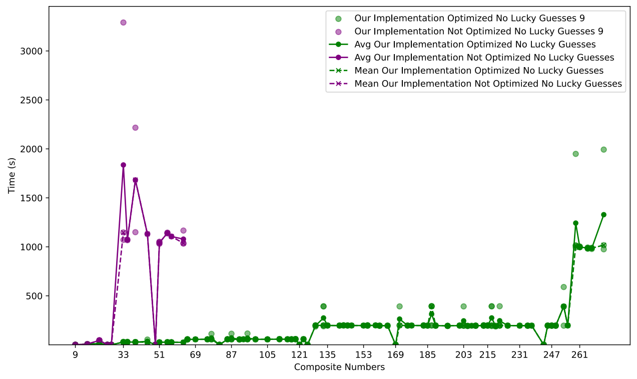


Figure 12: Results No lucky guesses compared to 5.

7 Discussion

Comparing all implementations in Figure 13 we can say that the needed execution time of all implementations increases with the bit size of the input number under the condition that the input is not a trivial case. We see that the different implementations also differ in their effectiveness when it comes down to their runtime. In case one would have to rank them, the following ranking would be applicable:

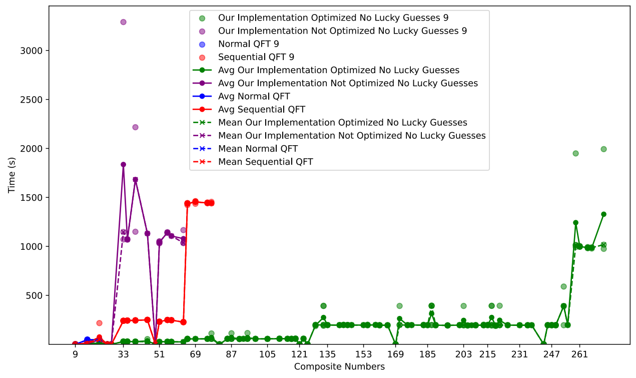


Figure 13: Compare all.

1. Our Implementation Optimized
2. Sequential QFT
3. Our Implementation Not Optimized
4. Normal QFT

Using only one controlling qubit instead of $2n+1$ qubits in Shor's algorithm is powerful because it dramatically reduces the quantum resources required, making the algorithm more practical for implementation on current and near-term quantum devices. Quantum hardware is limited in the number of available qubits and their coherence times, so minimizing qubit usage reduces state preparation and measurement errors. Additionally, using a single controlling qubit, even though it sequentializes certain parts of the circuit, does not increase computation times, it even decreases the execution time due to savings in circuit complexity. This efficiency gain makes Shor's algorithm significantly easier to run on less powerful quantum hardware, which is especially important due to the limited capabilities of today's quantum technology.

Lastly, we have to bring into discussion the difficulties of running this algorithm on real hardware. One of the big limitations of current-times quantum hardware is the amount of noise in the qubits and the lack of error correction. Given the scale of the circuit needed to factor a number of a practical length (e.g. at least 1024 bits), the accumulation of noise would make it so that the results are either unusable, or they require multiple runs, to the point of undermining the speed benefits of the algorithm. That is, of course, given the access to the necessary amounts of qubits in the first place. Beauregard's optimization greatly reduces the number of necessary qubits, making it more feasible in the near future. However, in the process, the circuit is lengthened, and the same qubits are reused for an extended amount of time compared to the normal version. This makes error correction that much more imperative. The reality is thus that, without a breakthrough in the field, it will be long before RSA will be ”broken by quantum”, probably long enough for most of the industry to move on to quantum-safe cryptographic algorithms.

8 Conclusion

To conclude, this project was a great way for us to peer into the world of real quantum. We got to read numerous papers, learn about how Shor's algorithm evolved over time, play with multiple implementations of it, and discover not only how it works but also what it is capable of. While the use of this algorithm is still currently more theoretical than practical due to hardware limitations, progress is being done in the field, and new ideas keep emerging. We are confident that at some point, after enough advancements will be made in terms of hardware, and with the right algorithms, we will be able to tap in into the true powers of quantum computing.

References

[1] Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits, 2003.

[2] Spencer Feingold and Filipe Beato. Us unveils new tools to withstand encryption-breaking quantum, aug 2024.

[3] Python Software Foundation. Python programming language.

[4] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, April 2021.

[5] IBM. Ibm quantum roadmap, 2024.

[6] IBM Quantum Documentation. qiskit.algorithms.shor.

[7] Edmond K. Machie. *Network security traceback attack and react in the United States Department of Defense network.* Trafford, Mar 2013.

[8] Rui Maia and Tiago Leão. Shoralgqiskit, 2019.

[9] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security Privacy*, 16(5):38–41, 2018.

[10] PRMJ. Quantum supremacy, Sep 2024.

[11] IBM Quantum. Qiskit, 2024.

[12] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[13] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[14] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[15] Eusebiu Volostiuc, Jonas Liley, and Tom Sorger. Shors-factorization-algorithm, 2024.

[16] Matthew J. Zanto and Helena Montana. The rsa encryption system and the factorization of large numbers, 2002.

A System Details

A.1 Hardware Information

- **Model**: Lenovo ThinkPad T14s Gen 4
- **Memory**: 32.0 GiB
- **Processor**: AMD Ryzen™ 7 PRO 7840U w/ Radeon™ 780M Graphics × 16
- **Graphics**: AMD Radeon™ Graphics

A.2 Software Information

- **Firmware Version**: R2EET37W (1.18)
- **OS Name**: Ubuntu 24.04.1 LTS
- **OS Type**: 64-bit
- **GNOME Version**: 46
- **Windowing System**: Wayland
- **Kernel Version**: Linux 6.8.0-49-generic