Shor's Algorithm using Quantum and Classical Modular Exponentiation

Scott McHaffie
Department of Applied Physics
Kungliga Tekniska Högskolan (KTH)
Stockholm, Sweden
mchaffie@kth.se

Jai Anand Iyer
Department of Applied Physics
Kungliga Tekniska Högskolan (KTH)
Stockholm, Sweden
jaii@kth.se

Venkatesh Elayaraja
Department of Applied Physics
Kungliga Tekniska Högskolan (KTH)
Stockholm, Sweden
venela@kth.se

Expected Grade and Work Division

We expect a final grade of **A**. We compared quantum modular exponentiation to classical modular exponentiation and explored the shortcomings of the quantum version. We implemented Shor's algorithm using classical modular exponentiation and applied our algorithm for modulo N = [15, . . . , 100] with bases a = [2, 3]. We compiled and ran our circuits using a noiseless Aer simulator. This report meets the **A** grade.

We show the work done by each individual group member below.

- S.M.: Writing and developed code for – quantum addition modulo N, digital logic to process the results and return prime factors, quantum circuit compilation/execution.
- J.A.I.: Writing and developed code for – Classical Modular Exponential Initialization, Quantum Exponentiation, Quantum Addition Modulo N.
- V.E.: Writing and developed code for – the Quantum Fourier Transform (QFT) and its inverse. Developed unit tests for – the Shor's circuits, and classical code, Optimized visualizations for probability distribution of state vectors

Introduction

We chose to work on the **Shor's Algorithm** project. We investigated two implementations of the **Modular Exponentiation** (ME) stage, namely, **Quantum ME** and **Classical ME**. In the Quantum ME study, we implemented the **Quantum Modulo Adder** (QMA) architecture introduced in [1]. Additionally, we tested its limitations with respect to different inputs and concluded that this architecture could not be used as the building block for quantum modular exponentiation. In the Classical ME study, we implemented modular exponentiation using classical computing, i.e., on our local computers, and used its output to initialize the qubit registers. We implement the remaining stages of Shor's algorithm and obtain the results of the prime factorization of integers $N \in \{15, \dots, 100\}$.

This report is divided into **five sections**. In **Section I** ("Background"), we briefly outline the core principles which make Shor's prime factorization algorithm significantly faster than the best classical algorithms. In **Section II** ("Methodology"), we describe the circuit implementations of each stage of Shor's algorithm pipeline, along with the classical post-processing stages. In **Section III** ("Implementation"), we lay out the implementation details we used for all our experiments. In **Section IV** ("Results"), we showcase the results from our experiments in the Quantum ME and Classical ME studies. In **Section V** ("Discussion"), we analyze the results obtained and discuss the limitations of the QMA, along with classical simulations of Shor's algorithm.

I. Background

Prime factorization is a computationally hard problem that underpins many classical cryptographic systems such as RSA, where security relies on the infeasibility of efficiently factoring large integers. The best known classical algorithm, the General Number Field Sieve (GNFS), achieves a sub-exponential runtime of $\mathcal{O}\left(e^{(\ln N)^{1/3}(\ln \ln N)^{2/3}}\right)$ and becomes impractical for large values of N. Shor's algorithm, proposed in 1994 [2], offers a quantum solution with polynomial runtime $\mathcal{O}\left((\ln N)^3\right)$, providing an exponential speedup over classical methods. The algorithm reduces integer factorization to the problem of finding the period r of the modular exponential function $a^x \bmod N$. This period-finding step leverages quantum parallelism and the Quantum Fourier Transform (QFT) to efficiently extract r, a task that is classically intractable. The importance of Shor's algorithm lies not only in its theoretical efficiency but also in its implications for cryptographic security, motivating studies on its classical simulation and practical realization to better understand its computational characteristics.

II. Methodology

The first building block of Shor's algorithm that we implemented was modular exponentiation, where we multiply the value in a quantum register by $a^x \mod N$. In this expression a is a user specified base, x is a series of positive integer values, and N is the value which is being factorized [3].

We define the series of x values within a quantum register which we will refer to as the **precision register** and we multiply a second quantum register, which we will refer to as the **working register**, by $a^x \mod N$. Let's first look at our implementation of quantum exponentiation.

A. *Quantum Exponentiation*

We began constructing the Shor's algorithm circuit by first implementing the quantum modular exponentiation step, which multiplies the working register by 2^x. Initially, we chose a fixed base of $a = 2$ to simplify the design, with the goal of later generalizing to arbitrary bases.

We built the exponentiation circuit using a modular approach. As a first step, we constructed a controlled multiplication-by-2 logic block that conditionally multiplies a specified register by 2 depending on the value of a control bit. The multiplication by 2 was implemented as a *left shift* operation on the register.

We then initialized the working register in the state $|1\rangle$, and initialized the precision register by applying Hadamard gates to all of its qubits, thereby creating an equal superposition of all possible states within the precision register. By applying a series of controlled multiplications by 2 from successive qubits in the precision register, the working register was multiplied by a superposition of 2^1, 2^2, $\dots$, $2^{2^{n_p-1}}$, where n_p denotes the number of qubits in the precision register.

An example circuit diagram for five working qubits and three precision qubits is shown in Fig. 1.

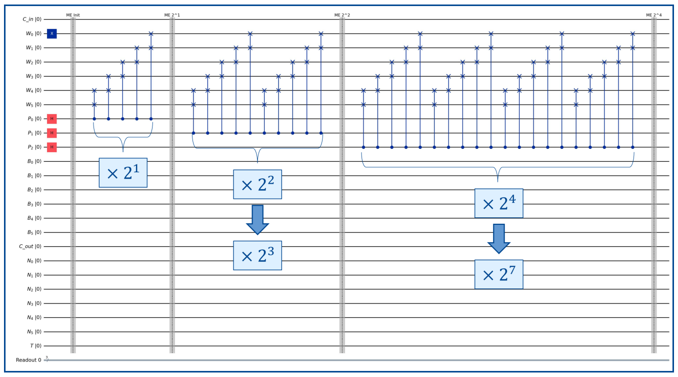


Fig. 1. A quantum exponentiation circuit generated using Qiskit, demonstrating the implementation for five working qubits (denoted with W) and three precision qubits (denoted with P).

In this example, it is evident that the precision register conditionally multiplies the working register into a superposition corresponding to multiplication by 2^1, 2^2, $\dots$, 2^7.

B. *Quantum Modulo Adder*

With the quantum exponentiation circuit implemented, we were able to successfully multiply the working register by 2^x. However, to perform modular exponentiation ($2^x \bmod N$), we also needed to construct a modulo operator logic block. To achieve this, we referred to a paper by Vedral et al. [1], which presents the logic for a quantum adder modulo N. The authors begin by introducing the quantum adder modulo N circuit shown in Fig. 2.

This adder circuit serves as a building block for constructing a multiplication modulo N routine, where multiplication is represented as repeated addition. Finally, exponentiation modulo N can be achieved through repeated use of the modular multiplication block.

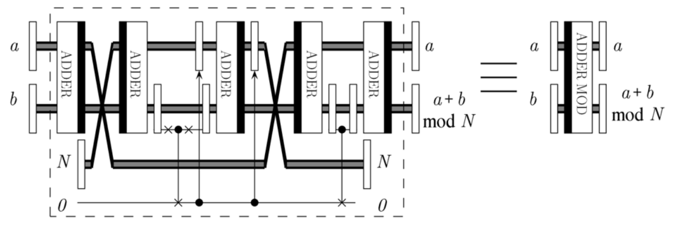


Fig. 2. Quantum circuit implementing an adder modulo N, taken from [1].

We implemented the adder modulo N circuit from [1] in Qiskit, and the resulting circuit diagram for five working qubits, three precision qubits, and $N = 15$ is shown in Fig. 3.

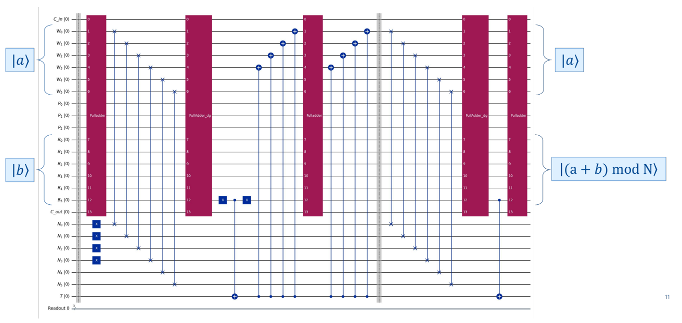


Fig. 3. Quantum logic implemented in Qiskit to realize the adder modulo N circuit described in [1]. The example shown corresponds to an adder modulo 15.

After testing this implementation, we observed that the circuit is only valid for cases where $a + b < 2N$, which is a significant limitation for this building block and our implementation of the modular exponentiation. This issue is discussed further in Sec. IV and Sec. V.

C. *Classical Modular Exponentiation*

In implementing the modular exponentiation step of Shor's algorithm, the computation is first performed classically to facilitate circuit initialization. In Step 1, the function $a^x \bmod N$ is evaluated for all possible values of x corresponding to the states of the precision register. The results are stored as binary strings in a list, denoted as `signal_binary`, with a total length of `signal_size = 2^{precision_bits}`.

For example, we plot the classical signal for $2^x \bmod 21$ for the case of 4-qubit precision registers (see Fig. 4).

In Step 2, these precomputed binary values are used to **conditionally initialize** the working register based on the basis state $|x\rangle$ of the precision register, to the state $|a^x \bmod N\rangle$. This ensures that the combined quantum state is initialized to $\sum |a^x \text{ mod N}\rangle \otimes |x\rangle$.

For example, the corresponding conditional initialization circuit for $2^x \bmod 21$ on a 4-qubit precision register is generated as follows (see Fig. 5).

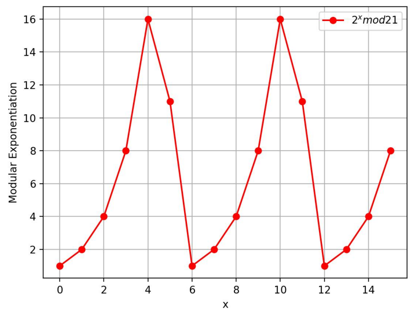


Fig. 4. Classical signal for 2^x mod N, $x \in \{0, .., 15\}$

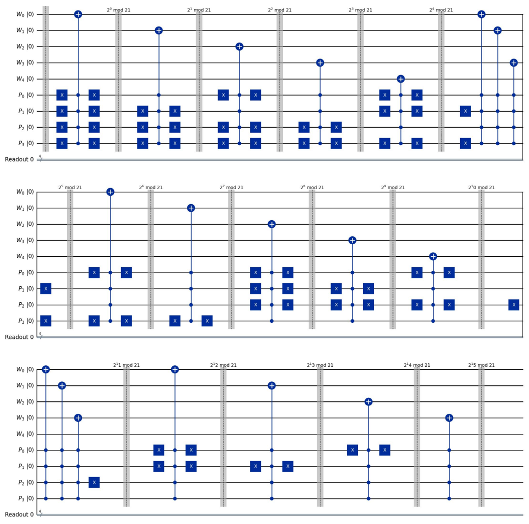


Fig. 5. Conditional Initialization Circuit for 2^x mod N

D. Quantum Fourier Transform

In Shor's algorithm, the Quantum Fourier Transform (QFT) is the key quantum subroutine used to extract the period r of the function $f(x) = a^x \bmod N$. The value of r is then used in the classical post-processing step to factor N. The overall method is: (1) prepare a uniform superposition over all inputs $|x\rangle$, (2) compute $f(x)$ in a second register so that the amplitudes encode a periodic structure, and (3) apply the QFT to transform this periodicity into peaks in the frequency domain, which can be measured. The schematic representation of the QFT circuit is shown in Figure 6, illustrating the sequence of Hadamard and controlled-phase gates followed by a qubit reversal.

For an n-qubit register with $N = 2^n$ basis states, the QFT is defined on computational basis states as

$$\mathrm{QFT}_N\big(|x\rangle\big) = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi i x y}{N}} |y\rangle. \quad (1)$$

This operation is the quantum analogue of the classical Discrete Fourier Transform, but it acts coherently on the amplitudes of a quantum state.

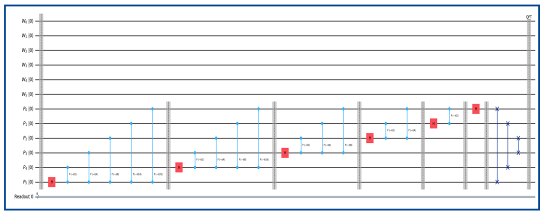


Fig. 6. QFT circuit diagram showing the combination of Hadamard and controlled phase rotation gates used to perform the transformation.

In circuit form, the QFT can be decomposed into single-qubit Hadamard gates H, controlled phase rotations R_k, and a final qubit reversal (SWAPs). The controlled rotation acting on a target qubit is

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}. \quad (2)$$

This means the QFT can be implemented using $O(n^2)$ one- and two-qubit gates, where $n = \log_2 N$, which is exponentially more efficient than performing a full classical Fourier transform on N elements.

After modular exponentiation, the input register is in a superposition of states separated by the unknown period r. Applying the QFT maps this periodic structure to sharp peaks at frequencies proportional to multiples of $1/r$. The resulting frequency-space peaks can be measured to yield an outcome y, as shown conceptually in Figure 6, such that

$$\frac{y}{N} \approx \frac{s}{r}, \quad (3)$$

for some integer s, from which r can be recovered using continued fractions.

E. Classical Components of Shor's Algorithm

Shor's algorithm combines both quantum and classical computation steps to achieve efficient integer factorization. While the Quantum Fourier Transform (QFT) is responsible for finding the hidden period, the surrounding classical components handle initialization, verification, and final factor extraction, as shown in Fig. 7.

The procedure begins by selecting an integer m such that $1 < m < N$ and $\gcd(m, N) = 1$. The quantum part of the algorithm is then used to estimate a value $j \approx c\, 2^{2L}/r$, where r is the unknown period of the modular exponential function

$$f(k) = m^k \bmod N. \quad (4)$$

Once the quantum step outputs j, the classical post-processing attempts to determine the period r from this result. If r is even and $m^{r/2} \not\equiv \pm 1 \pmod N$, then two non-trivial factors of N can be computed as

$$N_1 = \gcd(m^{r/2} - 1, N), \qquad N_2 = \gcd(m^{r/2} + 1, N). \quad (5)$$

If these conditions fail, the algorithm repeats with a different value of m.

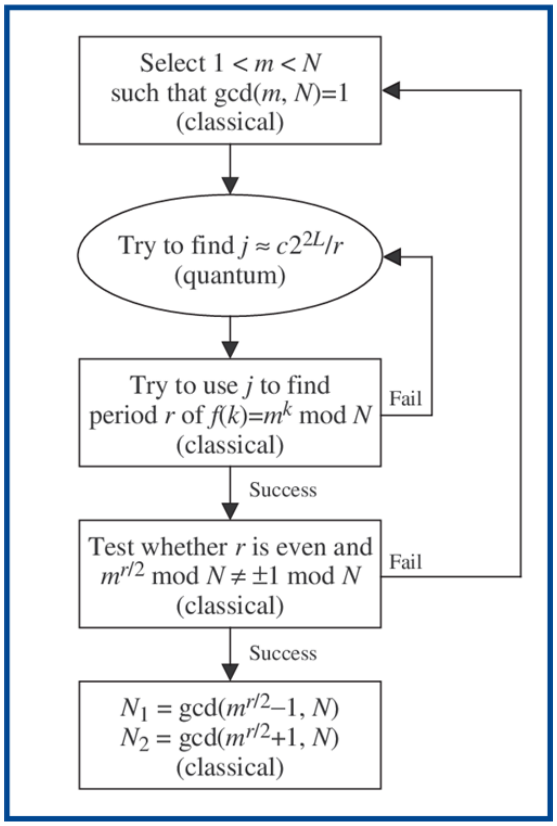

Select $1 < m < N$ such that $\gcd(m, N)=1$ (classical)
Try to find $j \approx c2^{2L}/r$ (quantum)
Try to use j to find period r of $f(k)=m^k \bmod N$ (classical)
Fail
Success
Test whether r is even and $m^{r/2} \bmod N \neq \pm 1 \bmod N$ (classical)
Fail
Success
$N_1 = \gcd(m^{r/2}-1, N)$
$N_2 = \gcd(m^{r/2}+1, N)$
(classical)

Fig. 7. Classical components of Shor's algorithm showing the iterative interaction between classical and quantum computation.

III. Implementation

We implemented and executed our quantum circuits using the Qiskit Aer simulator. Each simulation was performed with 2048 shots, and we used up to seven working qubits and six precision qubits in our designed circuits. All simulations were conducted in a noiseless simulator, as the focus of this project was on the ideal implementation of Shor's algorithm and on comparing quantum and classical modular exponentiation performance.

For each value of N, ranging from 15 to 100, we tested our implementation to identify its prime factors. The state measurement results from the quantum circuit were processed classically using a spike-finding algorithm and a factor-finding algorithm. These classical algorithms were applied to **all** measurement results and the program was terminated once any valid prime factors of N were obtained.

IV. Results

A. *Quantum Modulo Exponentiation*

We created our quantum modular exponentiation block by combining our quantum exponentiation circuit (Fig. 1) and the quantum adder modulo N circuit (Fig. 3). The working register was used to perform the modular multiplication, while the auxiliary register b was initialized to $|0\rangle$. The resulting circle plot for $N = 15$ is shown in Fig. 8. In this section we will refer to the working register as register a.

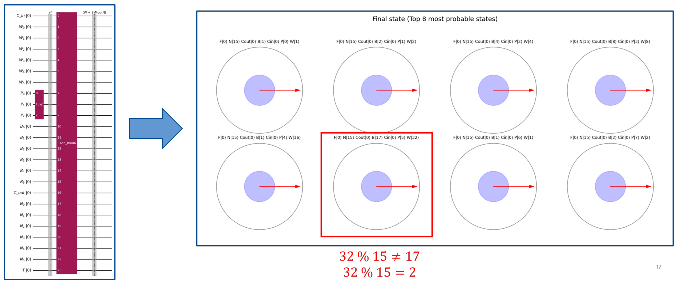


Fig. 8. Circle plot results of quantum modular exponentiation circuit for $N = 15$.

B. *Classical Modulo Exponentiation – Initialization*

We plot the simulated quantum state vector resulting from the the conditional initialization circuit for 2^x mod 21 below (see Fig. 9, zoom into circle plot for state decomposition).

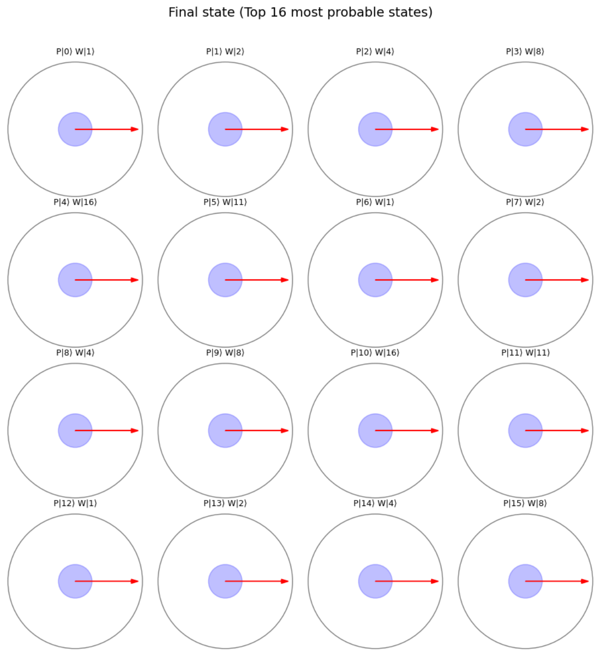


Fig. 9. Working and Precision registers initialized to $\sum |2^x \bmod 21\rangle\,|x\rangle$

C. *Classical Peak Finding from QFT*

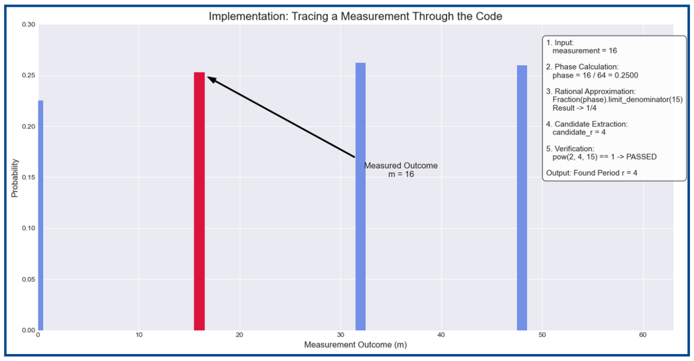
After performing the Quantum Fourier Transform (QFT), the quantum register is measured to obtain an outcome m, which corresponds to one of the peaks in the probability distribution. These peaks occur at values that encode information about the hidden period r of the modular function $f(x) = a^x \bmod N$. The resulting measurement distribution is shown in Figure 10, where distinct peaks indicate candidate values related to the true period.

The classical post-processing step interprets this measurement result to extract r. The process involves the following steps:

1) Compute the phase as phase $= m/2^n$.

2) Use a rational approximation method (e.g., continued fractions) to estimate the fraction $\frac{s}{r} \approx$ phase.
3) Extract r from the denominator of the best rational approximation.
4) Verify the candidate period by checking whether $a^{r/2} \not\equiv \pm 1 \pmod{N}$.

A valid measurement outcome produces a peak corresponding to a rational approximation that satisfies the verification step, confirming the correct period r. This procedure bridges the quantum output and the classical factorization step.



```
1. Input:
   measurement = 16

2. Phase Calculation:
   phase = 16 / 64 = 0.2500

3. Rational Approximation:
   Fraction(phase).limit_denominator(15)
   Result -> 1/4

4. Candidate Extraction:
   candidate_r = 4

5. Verification:
   pow(2, 4, 15) == 1 -> PASSED

Output: Found Period r = 4
```

Fig. 10. Classical peak extraction and rational approximation from the measured outcome of the QFT. The highlighted peak at $m = 16$ corresponds to a detected period $r = 4$.

D. *Prime Factorization using Shor's Algorithm*

In Table I, we showcase the factors found for integers $N \in \{15, ..., 100\}$, base $a \in \{2, 3\}$.

V. Result Discussion

A. *Failure of the Quantum Modulo Adder*

We observed that the output of the modulo adder circuit behaved as expected for $a+b < 2N$, returning $a+b \mod N$. However, for $a+b \geq 2N$ the output is actually $a+b-N$, rather than the expected $a + b \bmod N$. This represents a significant limitation of the circuit, particularly when it is used as a building block for quantum modular exponentiation. Once the value represented by a qubit in the precision register exceeds $2N$, the circuit fails to produce the correct modular result.

Initially, the circuit behaved as expected and correctly produced the value $b = |2^x \bmod N\rangle$ in the b register. This specifically occurs when $|a+b\rangle < |2N\rangle$. However, when $|a+b\rangle \geq |2N\rangle$, the circuit failed to compute the correct modular result. In such cases, the b register instead stored $b = |a+b-N\rangle$.

For example, when $x = 5$, the working register stored the value $a = |2^5\rangle = |32\rangle$. The value stored in the b register is then expected to be $b = |32 \mod 15\rangle = |2\rangle$. However, according to the circle plot diagrams, the value in register b is actually $b = |17\rangle$, which corresponds to $|2^5 - 15\rangle$.

We also tested this implementation for $N = 21$ and observed similar behavior. In this case, when $x = 6$ the a register contained $a = |2^6\rangle = |64\rangle$. The value stored in the

TABLE I
Prime Factors computed through Shor's Algorithm

Number (N)	Factors (Base = 2)	Factors (Base = 3)
15	(3, 5)	(3, 5)
16	(2, 8)	–
21	(7, 3)	(7, 3)
24	(2, 12)	(2, 12)
25	–	(5, 5)
32	(2, 16)	(8, 4)
33	(11, 3)	(3, 11)
35	(7, 5)	(7, 5)
39	(3, 13)	(13, 3)
40	(20, 2)	–
45	(9, 5)	(9, 5)
48	(2, 24)	(6, 8)
49	–	(7, 7)
51	(3, 17)	–
55	(11, 5)	(11, 5)
56	(4, 14)	(2, 28)
57	(19, 3)	(19, 3)
63	(7, 9)	(7, 9)
64	–	(8, 8)
65	(5, 13)	(13, 5)
69	(23, 3)	(3, 23)
72	(36, 2)	(2, 36)
75	(3, 25)	–
77	(7, 11)	(11, 7)
80	(40, 2)	–
81	–	(9, 9)
85	(5, 17)	–
87	(3, 29)	–
88	(44, 2)	(22, 4)
91	(7, 13)	(13, 7)
93	(31, 3)	(3, 31)
95	(19, 5)	(19, 5)
96	(2, 48)	(8, 12)
99	–	(11, 9)

b register was simulated to be $b = |43\rangle$. which is given as $b = |64 - 21\rangle$ and not $|64 \mod 21\rangle$.

Due to this limitation of the modulo adder, we decided to pivot toward an implementation of Shor's algorithm that employed classical modular exponentiation instead of a fully quantum version.

B. *Prime Factorization*

In the classical simulation of Shor's algorithm, using base 2 yielded successful factorization for all tested composite numbers of the form $N = p \times q$, where p and q are prime. This indicates that the modular exponentiation and period-finding steps are well-resolved with relatively low precision requirements. In contrast, simulations using base 3 produced correct factors for a smaller subset of N, suggesting that higher base values require larger precision registers to accurately capture the periodicity of $a^x \bmod N$. This increase in register size directly impacts computation time and memory usage in classical emulation. Furthermore, attempts to simulate bases of 5 and above become increasingly inefficient due to the rapid growth of intermediate values in modular exponentiation. To address this, repeated modular reduction or optimized exponentiation techniques could be incorporated to maintain numerical stability and scalability in higher-base simulations.

Conclusion

These results highlight the inherent limitations of classical simulations of Shor's algorithm. As the base and problem size increase, the exponential scaling of state-space and precision requirements quickly renders classical computation impractical. In an actual quantum implementation, these operations would be executed in parallel through quantum superposition, allowing efficient period finding with polynomial resources. The contrast between the classical and quantum performance underscores the fundamental advantage of quantum computation for problems such as integer factorization and illustrates why Shor's algorithm remains a central benchmark for evaluating quantum hardware capabilities.

VI. Individual Contributions

The implementation of Shor's algorithm was a collaborative effort, with each team member contributing to specific components of the quantum and classical workflow. The primary contributions are summarized as follows:

- **Scott McHaffie:** Developed the code for Quantum Addition Modulo N, implemented the routines for running the quantum circuit, and designed the digital logic required to process the measurement results and return the corresponding prime factors.
- **Jai Anand Iyer:** Implemented the Classical Modular Exponential Initialization, Quantum Exponentiation, and Quantum Addition Modulo N components, ensuring proper integration between the classical preprocessing and quantum execution stages.
- **Venkatesh Elayaraja:** Developed the Quantum Fourier Transform (QFT) and its inverse circuits, created comprehensive unit tests for both the quantum and classical parts of Shor's algorithm, and optimized the visualization routines for displaying the probability distribution of state vectors.

Each member's contribution played a key role in realizing a complete and functional implementation of Shor's algorithm, from modular arithmetic to quantum circuit execution and result interpretation.

References

[1] V. Vedral, A. Barenco, and A. Ekert, "Quantum Networks for Elementary Arithmetic Operations," *Physical Review A*, vol. 54, no. 1, pp. 147–153, Jul. 1996, arXiv:quant-ph/9511018. [Online]. Available: http://arxiv.org/abs/quant-ph/9511018

[2] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 124–134. [Online]. Available: https://ieeexplore.ieee.org/document/365700

[3] S. Markidis, "Lecture slides: DD2367 Quantum Computing for Computer Scientists." Recorded.