Quantum Factorising Algorithms

Pedro S. Gentil

Universidad Complutense de Madrid

May 28, 2025

1 Abstract

This work provides a comparative analysis of Shor's Factorising Algorithm and an alternative based on Grover's Search Algorithm [WK23], with the latter being able to find the prime factors of a semiprime number i.e., numbers composed of exactly two primes. Accordingly, the algorithms were implemented using the Qiskit [JATK+24] framework. They were then compared using small test cases to observe the limitations and advantages of each approach. Although Shor's algorithm offers an exponential speed-up over classical factorization methods, while the Grover-based alternative provides only a quadratic speed-up, this work demonstrates that, in practice, a basic implemented version of Shor's algorithm can be outperformed by a more carefully optimized Grover-based approach.

2 Introduction

Factorising is widely regarded as a computationally hard problem [Sho97]. This holds particularly true in the context of classical computation. Because of the assumption that this problem would remain a hard one, it has been adopted in cryptography, where the methods used to encrypt data must be resistant to attacks and must provide computational security. The main example of a cryptographic system that relies on the difficulty of factorising is RSA (from Rivest–Shamir–Adleman). RSA is a widely used asymmetric public-key cryptosystem. Examples of its use range from web communication protocols to bank transactions and digital signatures.

However, cryptosystems using factorization's hard computational nature in a classical context are threatened by the uprising of quantum computing. Quantum computation is theoretically able to factorise a composite number in polynomial time. This would mean that a cryptographic system such as RSA would become obsolete. Nonetheless, this has not happened yet. At present, quantum computation is significantly prone to errors and susceptible to noise, meaning that the physical and logical accuracy needed to perform such a task has not yet been achieved. Additionally, the qubits available in Quantum Processing Units (QPUs) are not sufficient to break a modern version of these cryptosystems. This poses an ongoing challenge for people designing and implementing quantum algorithms to find ways to solve the factorising problem more efficiently, by using fewer qubits or trying to decrease the noise embedded in the QPUs.

Shor's algorithm, designed by Peter Shor in 1995, can factorise a composite number in polynomial time [Sho97]. This algorithm has a mathematical foundation in group theory that will be explained later. It is composed of different steps, but this work will focus on one in particular, the period (or order) finding subroutine. This part of Shor's algorithm will run on a quantum computer. Without the quantum advantage, the period finding subroutine would not be efficient, meaning this part is the one carrying the burden of calling this a quantum algorithm. This subroutine will be discussed later as well.

The factorising algorithm based on Grover's search quantum algorithm is a much more recent development (2023) and presents an alternative to Shor's. However, this algorithm's basic implementation

can only factorise numbers composed of exactly two primes larger than 3. With this in mind and the general goal of improving the efficiency of quantum algorithms, the aim of this work is to provide clear comparisons between both of them. Grover's search algorithm general scheme will be explained, as well as how it was adjusted to solve the factorising problem.

3 Theoretical Background

3.1 Classical Factorization

As mentioned previously, factorisation is considered a computationally hard problem. Over time, several factorising algorithms have been developed for classical computers, yet none could achieve polynomial-time performance in the general case. Considering that the composite number n to factorise is not a perfect power, i.e., $n = x^y : \exists x, y \in \mathbb{Z}^+$, some of these algorithms require n to have a special form or rely on certain properties. Examples include trial division, Pollard's rho algorithm, Pollard's $p - 1$ algorithm, the elliptic curve algorithm, and the special number field sieve [MVO96]. Other more general algorithms, such as the quadratic sieve and the general number field sieve, do not have constraints over the input, although they do scale sub-exponentially with the size of n, making them unpractical for large composite numbers as inputs [MVO96].

3.2 Shor's algorithm

3.2.1 Explanation:

Shor's algorithm can factorise a composite number in polynomial time:

> "Our quantum factoring algorithm takes asymptotically O((logn)2(loglogn) (log loglogn)) steps on a quantum computer, along with a polynomial (in logn) amount of post-processing time on a classical computer that is used to convert the output of the quantum computer to factors of n."[Sho97]

However, it needs the composite number to meet certain conditions. The first condition it must meet, being N the number to factorise such that it is not a prime number nor a prime number's perfect power, is that there must exist a non-trivial solution to the equation:

$$x^2 = 1 \ (mod \ N), x \neq \pm 1 \tag{1}$$

Furthermore, this algorithm is a probabilistic algorithm, meaning it partially depends on randomness. Given a random number $a : 1 < a < N \ \wedge gcd(a, N) = 1$ (gcd : greatest common divisor), we try to compute the period of the function (**this is the part computed by a quantum computer**):

$$f(z) = a^z \ (mod \ N), \ z \in \mathbb{Z}^+ \tag{2}$$

Finding the period of this function is equivalent to finding the minimum $r : r > 0$ such that $a^r \ mod \ N = 1$. We can see in Figure 1 an example of the periodic function and a visual representation of its period. It can also be restructured as the problem of finding the order of the element a in the multiplicative group of N, i.e., $ord(a, \mathbb{U}(\mathbb{Z}_N)) : \mathbb{U}(\mathbb{Z}_N) = \{k : 1 \leq k < N \ \wedge \ gcd(k, N) = 1\}$. After finding the period r the following holds true:

$$a^r = 1 \ (mod \ N) \tag{3}$$

If r is **even** we can find a solution to equation 1:

$$x = a^{r/2} \tag{4}$$

With a solution to x we can replace it in the equation and manipulate it to reach our objective, which

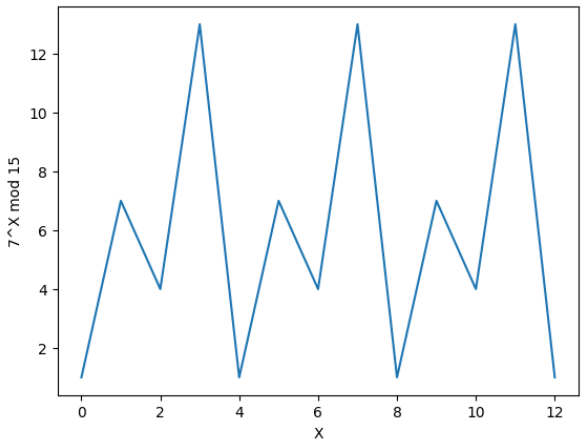


Figure 1: Example of periodic function $f(z) = 7^z \ mod\ 15$ with period $r = 4$

would be to factorise N:

$$(a^{r/2})^2 = 1\ (mod\ N) \tag{5}$$

$$(a^{r/2})^2 - 1 = 0\ (mod\ N) \tag{6}$$

$$(a^{r/2} - 1)(a^{r/2} + 1) = 0\ (mod\ N) \tag{7}$$

Consequently, for N the next equation holds true:

$$(a^{r/2} - 1)(a^{r/2} + 1) = c * N, \exists c \in \mathbb{N} \tag{8}$$

Thus, we can efficiently compute (using the Euclidean algorithm) the greatest common divisor between N and $Y_1 = (a^{r/2} - 1), Y_2 = (a^{r/2} + 1)$, resulting in at least one factor of N if $a^{r/2} \neq -1\ (mod\ N)$.

There are some details that must be taken into account. One, for example, being the probability of choosing an a randomly such that $1 < a < N\ \wedge\ gcd(a, N) = 1$ and that its period r is even. If $y = gcd(a, N) > 1$ then y would be a non-trivial factor of N, and the algorithm would finish. However, if this is not the case, then we would require the period r to be even, or else $r/2$ would not belong to $\mathbb{Z}$. The probability of finding an a that satisfies this requirement depends on the number of distinct odd prime factors of N (k). Shor states that the probability P is [Sho97]:

$$P \geq 1 - \frac{1}{2^{k-1}} \tag{9}$$

If we are trying to factorise a semiprime number $M = pq : p, q$ are primes larger than 2, then the probability of finding an a with even period r is of at least $1 - \frac{1}{2} = 0.5$. This means that if we choose 10 random numbers in the range $1 < a < M$, the probability of having at least one a with even period would be larger than 0.999.

Another detail to take into account is that the algorithm can fail if $a^{r/2} = -1$:

$$a^{r/2} = -1\ (mod\ N) \Rightarrow \tag{10}$$

$$a^{r/2} + 1 = 0\ (mod\ N) \Rightarrow \tag{11}$$

$$a^{r/2} + 1 = d * N, \exists d \in \mathbb{N} \Rightarrow \tag{12}$$

$$(a^{r/2} - 1)(a^{r/2} + 1) = c * d * N, \exists d, c \in \mathbb{N} \tag{13}$$

From here, $gcd(a^{r/2} + 1, N) = 1$ and it is not certain that $gcd(a^{r/2} - 1, N)$ will result in a factor of N. In this case, the algorithm should start again with a different random number a.

3.2.2 Steps:

1. Choose a number a randomly such that $1 < a < N$
2. If $b = gcd(a,\ N) > 1$ we have output b and we stop
3. Find the period of $a\ mod\ N$, with $r > 0$ such that $a^r = 1\ mod\ N$ (**Quantum Part**)
4. If r is odd we go to step 1
5. Compute $x = a^{r/2} + 1\ mod\ N$ and $y = a^{r/2} - 1\ mod\ N$
6. If $x = 0$ we go back to **Step 1**. If $y = 0$ we take $r = r/2$ and go back to step 4
7. We compute $p = gcd(x,\ N)$ and $q = gcd(y,\ N)$, at least one of them will be a non-trivial factor of N

3.2.3 Period-finding subroutine in Shor's algorithm:

The quantum period-finding subroutine provides the factorising algorithm with a remarkable time efficiency. The goal of this subroutine, as its name indicates, is to find the period (r) of a function. The function in this case has the form of $f(z) = a^z\ mod\ N$, which is why this subroutine found in Shor's algorithm is also known as the "order-finding subroutine". With the output of the quantum subroutine, some classical post-processing must be performed to recover r. An important detail to keep in mind is that this subroutine is an instance of the phase estimation algorithm in quantum computing. To formally understand how this subroutine works, one must have a clear idea of how the phase estimation algorithm functions. In this section, the goal is to build the subroutine step-by-step as a quantum circuit, and to gain some understanding of how it works.

The quantum order-finding subroutine starts by initializing two quantum registers, one to store the number to be factorised N (Y register with n qubits), and another to store the period estimation (X register with m qubits). We set the X register to 1 (we apply a Pauli-X gate in qubit 0 of register):

$$|X\rangle|Y\rangle = |0\rangle^{\otimes m}|1\rangle^{\otimes n} \tag{14}$$

After applying Hadamard gates to each qubit in the X register:

$$\frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle|1\rangle^{\otimes n} \tag{15}$$

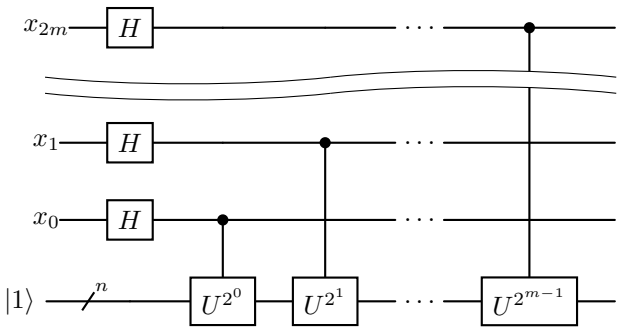
This step is crucial, as it prepares the X register in a uniform superposition over all possible input states, enabling the quantum algorithm to evaluate the function for all values simultaneously. Before continuing, a new operation U must be introduced. This operation (linear transformation) must be unitary, which means $UU^\dagger = I$ ($U^\dagger$ is the conjugate transpose of U). This property is required to implement the operation as a quantum gate. For any quantum state $|y\rangle$ it must have the following effect:

$$U|y\rangle = |y * a\ mod\ N\rangle,\ 1 < a < N \tag{16}$$

It is important to note that for this operation to be unitary, $gcd(a, N) = 1$. It demands having an inverse (for which the operation is reversible). Consequently, there must be another number b, $1 < b < N$ that verifies $a * b = 1\ mod\ N$, and this is only possible if a and N have no common divisors. If $|y\rangle : y \geq N$, applying the operation U should have no effect on the state, for the purpose of making U unitary. Accordingly:

$$U^j|y\rangle = |y * a^j\ mod\ N\rangle \tag{17}$$

The implementation of the U gate will be explained later. Continuing from expression 15, controlled-U^j operations are applied to the Y register using the qubits in the X register as control. Specifically, a gate per qubit in the X register should be applied. The way in which this is done is by equating j (power of the gate) to 2^x, x being the position of the qubit in the X register that controls the operation:



The qubit x_t, $0 \leq t < m$ (the $m-t$ most significant bit in the X register) controls the gate that will perform the U^{2^t} operation in the Y register, which was initially set to $|1\rangle^{\otimes n}$. This means that if qubit x_t is equal to $|1\rangle$, the Y register suffers from a multiplication in the form of $|y * a^{2^t} mod\ N\rangle$. For any state $|x\rangle$ in register X, we can decompose the state as $|x_{m-1} * 2^{m-1} + ... + x_2 * 2^2 + x_1 * 2^1 + x_0 * 2^0\rangle$ being $x_{m-1}...x_2x_1x_0$ the binary representation of x, which is the reason why the U gate is raised by the power of 2^t, being t the position of the qubit that controls it. As the X register is in the uniform state, i.e. all its qubits are in superposition, the expression for both registers is left as:

$$\frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle |a^x\ mod\ N\rangle \tag{18}$$

Although is not necessary throughout the circuit, if the Y register were to be measured at this point, collapsing to a state $|y\rangle$, $y \in \{1, a, a^2, .., a^{r-1}\}$, with r being the order of a, i.e. period of function $f(z) = a^z\ mod\ N$, then for that particular $|y\rangle$ there would be more than one x that could have produced that result. Hence, the collapse of the Y register would still leave room for superposition in the X register:

$$\frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x_0 + r * x\rangle,\ M = \lfloor \frac{2^m - 1 - x_0}{r} \rfloor,\ x_0 \in \{0, 1, .., r-1\} \tag{19}$$

It is noticeable that if the Y register was measured several times for the same collapsed $|y\rangle$, there would be a periodic structure in the X register (see Figure 2). In fact, the probability state vector for the X register would resemble a periodic function with period r. The problem is that it would contain an offset of x_0, and it is not possible to make the same $|y\rangle$ state collapse every time the circuit is executed because it depends on the value inside the X register (as it controls the controlled-U^j operations),

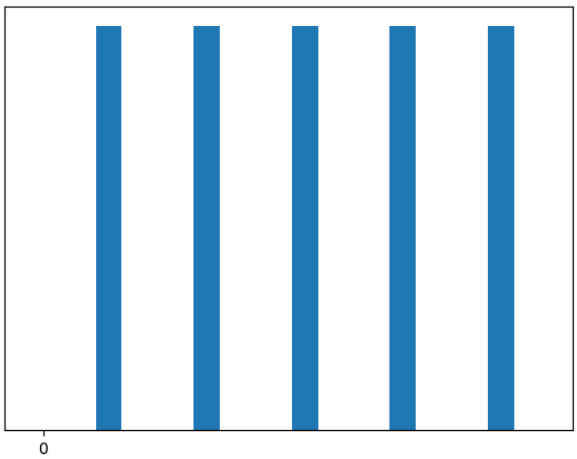


Figure 2: Structure resembling probability statevector of register X before applying $QFT\dagger$ for a collapsed state $|y\rangle$ in the Y register

which is in uniform superposition. The $|y\rangle$ state, if not measured, will be in a superposition of the states $\{|a^0\rangle, |a^1\rangle, ..., |a^{r-1}\rangle\}$.

Finally the $QFT\dagger$ (Inverse Quantum Fourier Transform) is applied to register X in order to inflict a constructive interference at states $|c\rangle \approx |s * \frac{2^m}{r}\rangle,\ s \in \mathbb{Z}^+$. The number s is a positive integer, and its value is arbitrary, although $2^m s/r < 2^m$. The precision of the approximation depends on the period r, if it divides 2^m and the number of qubits in the X register (m). After computing an approximation to $|s * \frac{2^m}{r}\rangle$ the period r must be retrieved. To that end, some classical post-processing is required.

$$|c\rangle \approx |s * \frac{2^m}{r}\rangle \tag{20}$$

$$|\frac{c}{2^m}\rangle \approx |\frac{s}{r}\rangle \tag{21}$$

To retrieve period r successfully, the closest approximation $|\frac{s'}{r'}\rangle$ to $|\frac{s}{r}\rangle$ must be computed. It might happen that r does not divide 2^m, therefore the estimation for $2^m s/r$ will not be totally accurate. In either way, to accomplish this, a continued fractions expansion can be computed in polynomial time[Sho97]. This method is useful because both the numerator and the denominator of the rational number can be retrieved (basically the fraction). The continued fraction expansion of a number is computed by representing it in the following way:

$$\frac{b}{c} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + ...}} \Rightarrow [a_0, a_1, a_2, ...];\ a_0, a_1, a_2, ... \in \mathbb{Z} \tag{22}$$

If one computes the continued fraction expansion of a real number, it might not converge (the produced list could have infinite elements). However, given an integer $k \geq 0$, the k-convergent of the number can be computed. For example, if $k = 2$ is chosen, the previous expression would only expand until the term a_2 and would converge to:

$$\frac{b}{c} \approx a_0 + \frac{1}{a_1 + \frac{1}{a_2}} \Rightarrow [a_0, a_1, a_2];\ a_0, a_1, a_2 \in \mathbb{Z} \tag{23}$$

The convergents give a method to approximate any real number. As mentioned above, the denominator and the numerator can be retrieved. For example, for the expression 23, the numerator and denominator would be:

$$Numerator : a_0 * (a_1 + \frac{1}{a_2}) \tag{24}$$

$$Denominator : a_1 + \frac{1}{a_2} \tag{25}$$

It is known that the period r will verify that $r < N$ [NC00], thus the convergent of interest will be the most accurate (with the largest k possible), yet having the denominator $r' < N$[Sho97]. It is possible that r' will end up being a divisor of the original period r (if $gcd(s, r) \neq 1$), which is why it is recommended that the circuit is executed more than once, obtaining the values $r'_1, r'_2, ...$ and then computing the lowest common multiple of the results. It would be enough to repeat the process n times ($\lceil log_2(N) \rceil$) [NC00]. Another possibility is that the circuit returns a poor estimate for s/r, although the probability of this occurring can decrease by increasing the circuit's size [NC00].

With this subroutine, the period r can be computed. A significant detail for the implementation is the number m of qubits assigned to the X register for estimation. Due to the fact that the number $2^m s/r$ might not have a precise representation in binary and there is a need to compute an approximation, the X register must have m qubits such that $2^m \geq N^2$ [Sho97]. This is why, for the implementation that follows, $m = 2n$:

$$m \geq 2 * n \Rightarrow \tag{26}$$

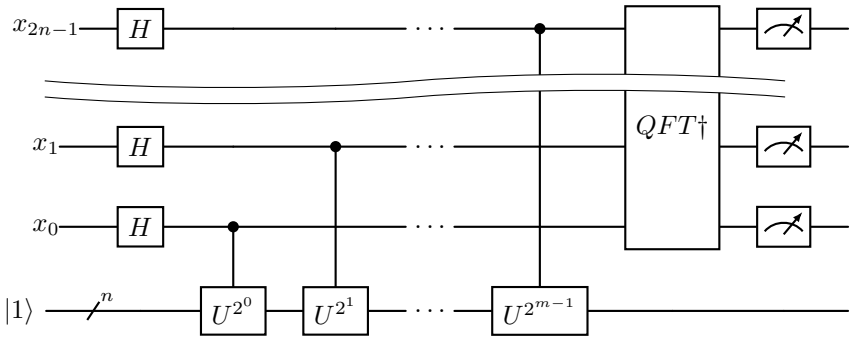
$$m \geq 2 * log_2(N) \Rightarrow \tag{27}$$

$$m \geq log_2(N^2) \Rightarrow \tag{28}$$

$$2^m \geq N^2 \tag{29}$$

$$\tag{30}$$

The implemention below (4.1) uses $3n$ qubits, n for storing the composite number N and $2n$ for the estimation. The final design of the period/order finding algorithm explained is:



3.3 Grover-based factorising algorithm

Lov Grover developed Grover's algorithm in 1996 as a quantum search algorithm, giving a quadratic speed-up for searching an unsorted database [Gro96]. In classical computation, one needs $\mathcal{O}(N)$ probes in the worst case to an N-element unsorted list. The Grover's algorithm, however, can solve the searching problem in $\mathcal{O}(\sqrt{N})$ iterations, due to the advantage given by quantum superposition. The algorithm is based on the successive application of two main components: the oracle, which marks the solution state by applying a phase shift, and a diffuser, that amplifies the probability of measuring the marked state. While Grover's algorithm is not directly applicable to factorising, it has been adjusted to solve this problem by encoding it as a search over possible divisors of a semiprime number. This implementation of Grover's search algorithm is especially appealing as it provides a more space-efficient alternative to Shor's algorithm, as it uses less qubits. Today, even scaling down the time or space complexity of a quantum algorithm could bring new breakthroughs due to the limitation of quantum hardware brought by noise interference and the restricted number of available qubits.

3.3.1 Explanation

Given a semiprime number N, the goal is to find p, q such that $N = p * q$. The special aspect of this algorithm is that it **decreases** the size of the problem by expressing N, p, q differently. Let M, a, b be integers greater or equal to 0 and $s, S = \pm 1$, N, p, q can be restructured as:

$$N = 6 * (M + 1) + S \tag{31}$$

$$p = 6 * (a + 1) + s \tag{32}$$

$$q = 6 * (b + 1) + s * S \tag{33}$$

For these equations to be true, p and q must be primes strictly greater than 3 [WK23]. Furthermore, given a semiprime number N, there is a unique value for M(35) and S(34). Due to the decomposition performed on p, q, the problem went from finding these two primes to finding numbers a, b that satisfy the equations. As s can only take on two values (± 1), the way to find the correct one can be achieved by trial and error. The next equations are derived from the ones above:

$$S = 1.5 - 0.5 * (N \ mod \ 6) \tag{34}$$

$$M = \frac{N - S}{6} - 1 \tag{35}$$

It is important to note that $N \ mod \ 6$ can only be equal to 1 or 5, due to the restriction imposed on p, q, which prevents them from being equal to 2 or 3. Therefore, M can be expressed in terms of a and

b as follows [WK23]:

$$M = \frac{N - S}{6} - 1 = \frac{p * q - S}{6} - 1 \tag{36}$$

$$M = \frac{(6(a+1)+s)(6(b+1)+sS) - S}{6} - 1 \tag{37}$$

$$M = \frac{6^2(a+1)(b+1) + 6(a+1)sS + 6(b+1)s + s^2S - S}{6} - 1,\ s^2 = 1 \tag{38}$$

$$M = g(a,b) = 6(a+1)(b+1) + s(b+1) + sS(a+1) - 1 \tag{39}$$

$$M = f(a',b') = 6a'b' + sb' + sSa - 1 \tag{40}$$

With this final expression, the function $f(a,b)$ is defined. This function is the heart of the oracle that is going to be used. The oracle implementation will be discussed later (Section 4.2). The problem of finding p, q such that $N = pq$ was transformed into the problem of finding a, b such that $M = f(a,b)$.

The quantum circuit designed to implement the algorithm will have three registers. One for M (X with), another for a (A) and the final for b (B). The basic explanation for this algorithm using Grover's search is that registers A, B, with states $|a\rangle, |b\rangle$ respectively, will be in a uniform superposition (we apply Hadamard gates to each qubit) of all the possible states they can store, and the oracle will perform the operation $|M - f(a,b)\rangle$ on the X register (that was initially set to $|M\rangle$). The register A, B will contain correct values if the operation $|M - f(a,b)\rangle = |0\rangle$. Therefore, the oracle will mark the state $|0\rangle$ (changing its phase $|0\rangle \rightarrow -|0\rangle$) and then reverse the operation, by performing $|M + f(a,b)\rangle$ in the X register. Finally, the diffuser operator (canonical) will act upon the A, B registers, amplifying the probability of measuring the values that marked the state $|0\rangle$, meaning the values that made $|M - f(a,b)\rangle = |0\rangle$. As it is based on Grover's search algorithm. this process needs to be iterated $k : k \in \mathbb{Z}^+$ times.

The number of iterations the search can optimally have is given by [WK23]:

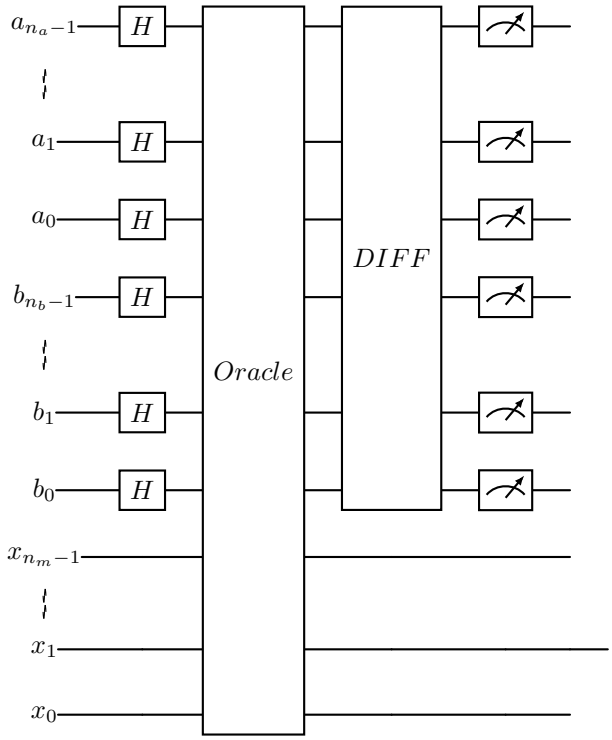
$$k = \lfloor (\frac{\pi}{4})\ 2^{\frac{n_a + n_b}{2}} \rfloor \tag{41}$$

The constants n_a and n_b are the numbers of qubits in registers A and B respectively. Due to the nature of the problem, these registers can achieve optimal size as follows:

$$n_a = \lfloor \frac{n}{2} - 2 - d \rfloor \tag{42}$$

$$n_b = \lceil \frac{n}{2} - 2 + d \rceil \tag{43}$$

Where d is the bit distance $d = (n_a - n_b)/2$ [WK23]. It refers to the difference in size between the a and the b parameters. The value of d can range from 0 to $\lceil \frac{n}{2} - 2 \rceil$ depending on the problem. If one wanted to try and break the RSA cryptosystem where the number to factorise is semiprime with p, q of similar size, this parameter d would equal 0. However, if this is not the case and using an optimal amount of space is absolutely necessary, the algorithm could be tested several times with a changing value of d (for the range mentioned above). Defining the size of register X as $n_m = n_a + n_b + 3$ (to prevent overflow [WK23]), the optimal implementation of this algorithm would require $2n - 5$ qubits in the worst case, being $n = \lceil log_2(N) \rceil$, and N the semiprime number to factorise. If n_a and n_b are optimally defined as explained above and the value of d is correct, there will only be one solution that would satisfy $|M - f(a,b)\rangle = |0\rangle$. If n_a and/or n_b became larger, then there could be 2 solutions to this implementation of the Grover's search algorithm ($A = |a\rangle, B = |b\rangle \ \vee A = |b\rangle, B = |a\rangle$), and the value for d could remain as 0. The search space for this problem has a size of $2^{n_a + n_b}$. Basic design for Grover's algorithm as a circuit(the Oracle and Diffuser structures are repeated k times):



3.3.2 Steps:

1. Verify that number N is not divisible by 2 or 3.
2. Compute M, S and define $s = +1$
3. **Quantum** Execute Grover's search algorithm with custom oracle (returns a, b)
4. Compute $p = 6a + s, q = 6b + sS$
5. Check if $N = pq$, if not go back to step 3 with $s = -1$
6. return p, q

4 Methodology

An analysis of both Shor's algorithm and the alternative using Grover's search will be conducted through the design and simulation of quantum circuits using Qiskit [JATK+24]. The simulator used for this work was the "Aer simulator" provided by IBM. The circuits were also executed in the IBM Torino QPU (Heron) to test how far they can go in a real quantum computer. The source code will be available on Github [Gen25] in notebook format.

4.1 Implementation for Shor's algorithm

In this section the focus will point towards the quantum implementation of the algorithm, i.e. the period-finding subroutine. As mentioned before, the quantum circuit must have two quantum registers. The first quantum register will be composed of $2n$ qubits, $n = \lceil log_2(N) \rceil$ with N being the number to factorise, and will store the estimation value $2^m s/r$. The second quantum register will be composed of n qubits, and will be affected by the quantum operation U^j, where U is a unitary gate. To build this circuit one needs three components: Hadamard gates, a generic U operator gate and a generic QFT

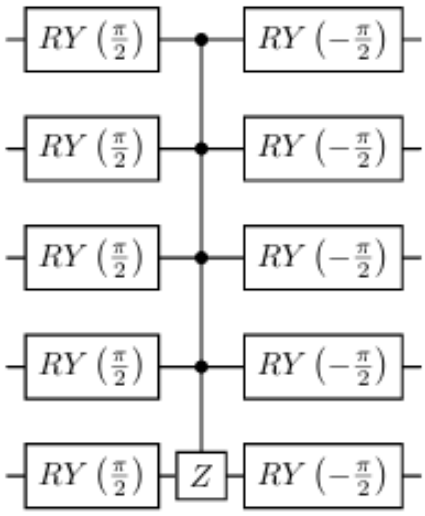


Figure 3: Diffuser operator for register A and B in Grover's search algorithm to factorise (Ref. [WK23])

operator gate. Hadamard gates and the QFT operator are already defined in Qiskit. Therefore, this part will focus on the implementation of the U gate.

A way in which the U gate can be implemented using Qiskit is by defining a unitary matrix that models its expected behaviour (see expression 16). This is how it is implemented in the source code. To achieve this, the next definition for the matrix was followed, given a random number a,the number to factorise N ($gcd(a, N) = 1 \ \wedge \ 1 < a < N$), and the number $n = \lceil log_2(N) \rceil$ of qubits in the second register:

$$u_{ij,0\leq\ i\ <\ 2^n,0\leq\ j\ <2^n} = \begin{cases} 1 & (((a * j)\ mod\ N = i)\ \wedge\ j < N)\ \vee\ (i = j\ \wedge j \geq N) \\ 0 & otherwise \end{cases} \tag{44}$$

Qiskit provides methods that allow to exponentiate gates, which could be used to create the U^j gates. Furthermore, it also provides a method to generate the controlled version of gates. With these tools the quantum circuit for period finding can be implemented.

4.2 Implementation for alternative factorising algorithm using Grover's search

This section will explain the oracle's and the canonical diffuser's implementation in Qiskit for the quantum part of the algorithm. Firstly, the algorithm requires three registers that will be called A, B, X. The registers A, B store the results a, b respectively at the end of the circuit's execution. These two registers must begin in a state of uniform superposition, meaning a Hadamard gate must be applied for each of their qubits. Register X must store the state $|M\rangle$, the number M mentioned in section 3.3.1, which can be done with a function that receives a number as an argument and returns a Qiskit quantum circuit with X gates at the corresponding qubits. Then the oracle and the diffuser must be applied k times to the circuit. Afterwards, registers A, B must be measured to retrieve a, b. To start with, the diffuser in Grover's search algorithm is canonical, meaning that it can be implemented the same way in all instances of the problem. As this work does not target specific designs with limited access to multi-qubit gates, the way to implement this operator is shown in Figure 3. As illustrated, the diffuser is an application of rotations in the Y-axis followed by a multi-controlled Z gate and ending with the inverse operation over the Y-axis. The oracle, on the other side, is customized for this specific problem. As it was stated in section 3.3.1, the oracle must act on the X register, performing the operation $|M - f(a, b)\rangle$, with a, b stored in registers A, B. The function $f(a, b)$ is defined in section

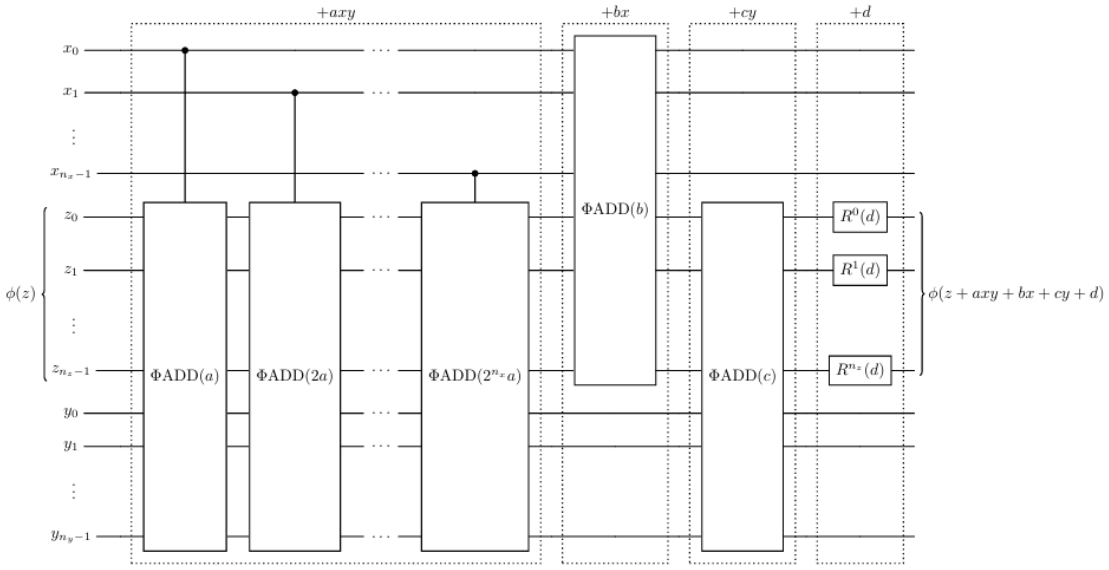


Figure 4: QMA operator (Ref. [WK23])

3.3.1 in equation 40. The following equation holds:

$$|M - f(a,b)\rangle = |M - 6ab - sSa - sb + 1\rangle \tag{45}$$

This is done by creating a parametrized operator Quantum Multiply Add (QMA) that allow us to perform the next operation to any register Z with state $|z\rangle$, and two registers X, Y, with states $|x\rangle, |y\rangle$ respectively, as follows [WK23]:

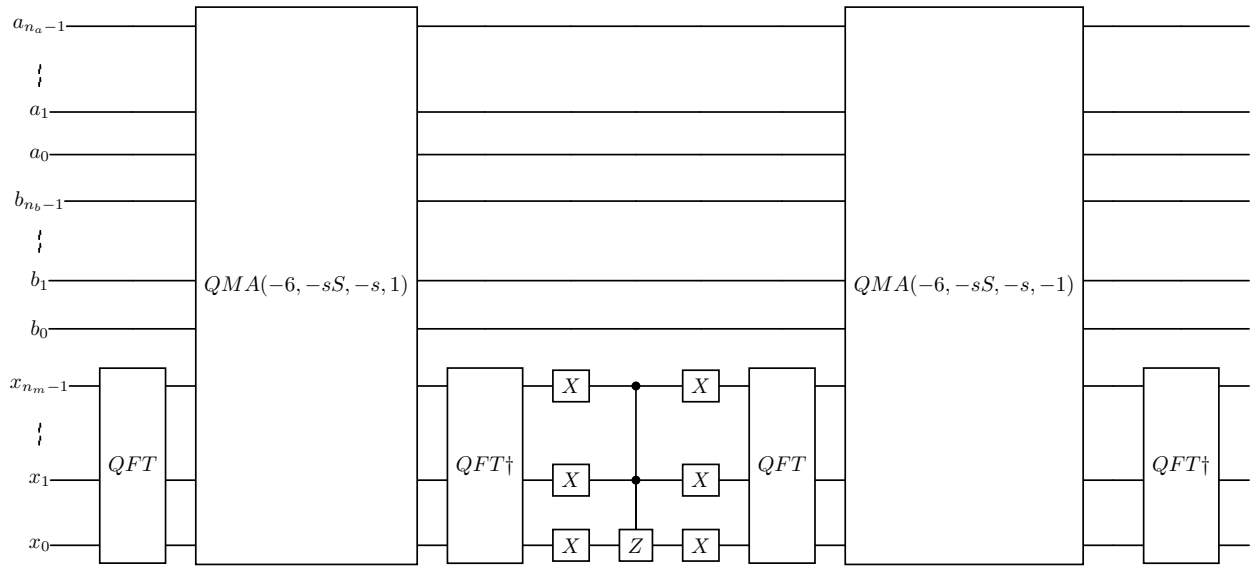
$$QMA_{c,d,e,h}|x\rangle|y\rangle|z\rangle = |x\rangle|y\rangle|z + cxy + dx + ey + h\rangle \tag{46}$$

By applying the QMA operation in the circuit with parameters $c = -6, d = -sS, e = -s, h = 1$ we get:

$$QMA_{-6,-sS,-s,1}|a\rangle|b\rangle|M\rangle = |a\rangle|b\rangle|M - 6ab - sSa - sb + 1\rangle \tag{47}$$

The QMA operator's implementation can be done with Draper adders [Dra00]. Therefore, a QFT operator is also needed, as Draper's quantum adder performs addition in the Fourier basis. However, for it to multiply by a constant this adder must be modified. Its basic implementation performs rotations in one quantum register controlled by the qubits in another quantum register. To multiply, the rotations must be scaled by the constant. For more details of the adder's implementation check Ref. [WK23].

To perform the desired operation, a QFT must be applied to the X register of the circuit (the one containing M), then the QMA operation constructed with Draper's quantum adders must be applied to A, B, X registers, and finally an inverse QFT to go back to the computational basis in the X register. The QMA's implementation follows the design in Figure 4. Afterwards, the goal is to mark the state $|0\rangle$, which is done by applying a column of X-gates in the X register, a multi-controlled Z operation with target qubit x_0 and control qubits $x_1, x_2, ..., x_{n_m-1}$ and finally another column of X-gates. At last, the QMA operation must be reversed to restore the X register to state $|M\rangle$. Oracle design:



Some details about the implementation:

- The first QFT and the last $QFT^\dagger$ of the oracle can be applied once in all the circuit's implementation. There is no need to apply them every iteration.
- All operation between the QMA and its reverse operation can be replaced by a diffuser but applied to register X [WK23]

4.3 Circuit's analysis

The efficiency of the algorithms and even the coherence of the results that each return can depend on their quantum circuit's implementation. In an era where quantum computers have scarce qubits and significant error rate induced by noise interference, implementing quality quantum algorithms might help get better results.

One aspect that impacts a quantum circuit's performance is the depth of the circuit. The depth of the circuit, also called the critical path, is the number of temporal steps the circuit must do to finish its execution. It is an indicator of a circuit's reliability due to the state decoherence qubits suffer when the circuit is running. The larger the depth of a circuit, the more information is probably lost. This is why it is important to compare the circuit's depth for each algorithm. The next tables show the depth information of each circuit when transpiled to different backends.

Depth of each circuit for composite number N (No backend)		
N	Shor's algorithm (Period-finding)	Alternative using Grover's search
35	15	7
77	17	11
221	19	15
437	21	19
667	23	27

Depth of each transpiled circuit for composite number N (Aer simulator - optimization level = 2)		
N	Shor's algorithm (Period-finding)	Alternative using Grover's search
35	37	56
77	43	128
221	49	240
437	55	454

Depth of each transpiled circuit for composite number N (IBM Torino - optimization level = 2)		
N	Shor's algorithm (Period-finding)	Alternative using Grover's search
35	341427	883
77	1590553	2908

It is noticeable that for smaller cases, the algorithm based on Grover's search has lower logical depth than Shor's. As the number to factorise increases, the gap between the logical depth of both circuits reduces. Grover's search algorithm seems to have a more elevated depth increase rate than Shor's. However, things change when the circuit is transpiled. For example, for Aer simulator, the algorithm based on Grover's search seems to have a significantly larger depth value for its circuit, and an exponential increase rate, while Shor's algorithm has a linear increase rate of depth as the problem size gets bigger. Aer simulator has access to many native gates, as it does not depend on specific quantum technology to do the computations. Lastly, for the real quantum computer IBM Torino, there is an unexpected result for the circuit's depth values. When the circuit is transpiled into the IBM Torino backend, Shor's algorithm's circuit has a depth of more than 300000 time steps. Although both circuit's depths increased significantly, Shor's algorithm's circuit for period finding has an unprecedented value for it. This might have been caused by the implementation of the unitary gate with the matrix declaration. This will be discussed in the following section.

5 Results

This section will post the results of running both algorithms in simulation (Aer simulator) and in a quantum computer (IBM Torino - Heron). For comparison's sake, the test cases that will be used are for factorising semiprime numbers, so that the algorithm based on Grover's search can be analysed.

5.1 Simulation

To simulate both algorithms, the Qiskit Aer simulator was used. In simulation both algorithms were able to run some basic test cases. For example, for $N = 35$ and $N = 77$, both recovered the correct factors for each. However, Shor's algorithm has shown difficulties when N was increased to $143 = 11 * 13$, as the simulation takes significant amount of time to finish. In simulation, the algorithm based on Grover's search seems to have better performance. The probable cause for this is the declaration of the Unitary Gate with a custom matrix. If the unitary operator for the modular exponentiation in Shor's algorithm was implemented using the gates provided by Qiskit's framework, then it could be possible to enhance Shor's performance. On the other hand, the algorithm based on Grover's search has shown no difficulties with numbers like $35, 77, 143, 221, 437, 667, 713$. This difference between these two algorithms in simulation probably comes from their implementation, as mentioned before. Using matrix-defined unitary gates calls for an expensive use of gates that scales rapidly when increasing the circuit's size. Furthermore, the creation of these gates is time-expensive, as the simulator needs to verify they are unitary.

The conclusion drawn from simulation is that the implementation of the order-finding unitary gate (modular exponentiation) must be modified to allow Shor's algorithm to run faster. Also, at least in simulation, the algorithm based on Grover's search showed great performance reaching factorisation of semiprime numbers up to $N = 4757$.

5.2 Running on a Quantum Computer

Both algorithms were executed in a quantum computer (IBM Torino, Heron). This section posts their results and some metrics.

5.2.1 Shor's algorithm

Shor algorithm was tested by running with $N = 15, N = 21, N = 35, N = 77$. The first two were used to test the circuit's functionality in a real backend. The last two were used to compare the execution with Grover's search algorithm.

Executing Shor's factorising algorithm in IBM Torino (Heron)		
N	Successful	Time in Quantum Computer
15	Yes	4s
21	No	7s
35	No	14s
77	No	19s

Some comments about the executions:

- The circuit's results from $N = 21$ upwards were inconsistent.
- For $N = 15$, repetitive execution of the circuit showed good results. Increasing the size of the problem (from $n = 4$ to $n = 5$, being n the number of bits necessary to represent N, is the probable cause of the inconsistency.

5.2.2 Algorithm based on Grover's search

Grover's search algorithm was tested by running with $N = 35, N = 77, N = 221$.

Executing algorithm based on Grover's search algorithm in IBM Torino (Heron)		
N	Successful	Time in Quantum Computer
35	Yes	2s
77	No	2s
221	No	9s

Some comments about the executions:

- The circuit's results from $N = 77$ upwards were inconsistent.
- For $N = 35$, the circuit did not give the expected result distribution. The number of shots for each execution was 1000, where around 30% were accurate. This has to be taken into account for further improvement.

6 Conclusions

Given the data observed in the results section and the circuits analysis, some conclusions can be drawn from it. First, Grover's search algorithm's iterative approach is theoretically expensive in terms of gate utilisation and depth. This instance of Grover's search algorithm uses $O(n^3 2^{n/2})$ elementary gates[WK23]. In comparison, a basic implementation for Shor's algorithm should use around $O(n^3 log(n))$ gates [PVG25], which represents an exponential difference from the alternative. However, a basic and generic implementation of Shor's algorithm has a significant problem, which is the modular exponentiation operation. This operation must be implemented as a gate and is generally done with the help of a classical algorithm (fast exponentiation). Although this helps build the period-finding circuit, the practical implementation of this gate remains a bottleneck for it [PVG25]. As seen in Section 4.3, when the period-finding circuit was transpiled to IBM Torino's backend, the circuit depth took a huge leap from what the original circuit design had.

Comparatively, Grover's search algorithm showed unexpected better results in simulation and in the real quantum computer than Shor's algorithm. In a real backend, as seen, this algorithm could

only withstand noise interference until $N = 35$. The implemented Shor's algorithm, on the other hand, only could withstand noise interference only until $N = 15$.

Another detail to take into account is the "quantum-time" metric. This is the time each circuit spends running on the real quantum computer. Grover's algorithm, in this case, surpassed Shor's algorithm, as shown in Section 5.2. As mentioned above, the probable cause of this might come from the implementation of the modular exponentiation gate, that in a real backend adds a significant amount of depth to the circuit. Today, there are some generic implementations of the algorithm that intend to solve the problem related to the modular exponentiation (Ref. [PVG25] for example).

Despite the fact that the algorithm based on Grover's search performed better, it is still important to highlight that the implementation seen is only tailored to semiprime numbers, i.e. number composed of exactly two primes. Additionally, the primes that compose it must be greater than 3. This leaves room to a lot of cases where Shor's algorithm is a better fit for the factorising problem. Furthermore, Shor's algorithm grants the factorisation problem an exponential speed-up compared to classical factorisation techniques, while using Grover's search only grants a quadratic speed-up [WK23]. Therefore, for a large problem size, Shor's algorithm, with the correct implementation, would be preferable.

References

[Dra00] Thomas G. Draper. Addition on a quantum computer, 2000.

[Gen25] Pedro Santiago Gentil. Quantum factorising, 2025. Accessed https://github.com/pgentil/QuantumFactorising : 2025-05-28.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[JATK+24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[NC00] Michael A Nielsen and Isaac L Chuang. *Cambridge series on information and the natural sciences: Quantum computation and quantum information*. Cambridge University Press, Cambridge, England, October 2000.

[PVG25] Abu Musa Patoary, Amit Vikram, and Victor Galitski. A discrete fourier transform based quantum circuit for modular multiplication in shor's algorithm, 2025.

[Sho97] 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.

[WK23] S. Whitlock and T. D. Kieu. Quantum factoring algorithm using grover search, 2023.