Post-Quantum Cryptography Algorithms and Cyber Security

Lucas S. Jorge[1]

[1]Despartment of Computer Science – Instituto Tecnológico de Aeronáutica (ITA)
Department of Science and Aerospace Tecnology (DCTA)
São José dos Campos – SP – Brazil

lucas1jorge@gmail.com, lucas.jorge@ga.ita.br

Abstract. *Quantum Computing is a hot topic among academia, industry and even governments. Large companies are hurrying to develop their own libraries and tools to gain influence in the field. The two most popular quantum algorithms clarify why there's so much impact coming with the arrival of such technology: The time complexity of important tasks in nowadays computing can be strongly decreased . Shor's algorithm was designed in 1994 to solve the problem of Large Integer Factorization (very important in popular security systems such as the RSA cryptography) and might be able to expose a substantial part of the internet security when quantum computers become more accessible. Grover's algorithm was designed in 1996 to perform searches in unstructured databases, and currently is more generally applicable to reverse functions such as hashs (base of the popular SHA-256 and other cryptosystems).*

Resumo. *A computação quântica é um tema em alta nos meios academico, industrial e até em organizações governamentais governos. Grandes empresas estão apressadas para desenvolver suas próprias bibliotecas e ferramentas para ganhar influência no campo. Os dois algoritmos quânticos mais populares esclarecem por que há tanto impacto relacionado à chegada dessa tecnologia: a complexidade de tempo de tarefas importantes na computação atual pode ser fortemente reduzida. O algoritmo de Shor foi projetado em 1994 para resolver o problema da Fatoração de Grandes Inteiros (muito importante em sistemas de segurança populares, como a criptografia RSA) e pode ser capaz de expor uma parte substancial da segurança na Internet quando os computadores quânticos se tornarem mais acessíveis. O algoritmo de Grover foi projetado em 1996 para realizar pesquisas em bancos de dados não estruturados, e atualmente é mais geralmente aplicável como reversor de funções, como os hashs (base do popular SHA-256 e outros sistemas criptográficos).*

1. General Information

Most of nowadays cybersecurity is based on the the problems of large integer factorization, discrete logarithm and elliptic-curve discrete logarithm. These 3 mathematical problems are not solvable within practicable time by classical computers, but are less challenging to quantum computing algorithms.

The worldwide expectations for the arrival of quantum processors and algorithms is that a great renovation will be needed for important areas, such as security. For those

reasons, large companies have been rushing to build their quantum computing tools, environments and programming languages, and some are also trying to popularize their own frameworks making them available open-source. Some examples are:

- IBM's Qiskit (Quantum Information Science kit), which is an open-source software development kit (SDK) that allows users to access the IBM Q quantum processors. Programmers can explore the IBM Quantum Experience using python.
- Huawei's HiQ quantum computing simulation cloud platform. Huawei alleged to open to the public for the purpose of research and education.
- Google's QScript, a quantum programming language developed by Google. The company also provides an environment, the Quantum Computing Playground, with many tutorials, example codes, and a simulator with 22 qubits in which programmers can write QScript code. The simulator provides base codes (Shor's and Grover's algorithms, quantum gates circuits and other examples) that the users can read to learn and ameliorate their own code.
- Microsoft's Q#: Runs the primary logic in a classical host processor and, in proper occasions, invokes subroutines with the quantum codes. Therefore, the quantum hardware works as a coprocessor (that's also usual for other hardwares like GPU and FPGA).

There are multiple other popular examples, like the QCL (Quantum Computing Language), a language that resembles C's syntax. A case that might be seen as somehow analog to this were when Google released TensorFlow, in 2015, and then made it open source. The library was originally developed by Google Brain, for the company's internal use. When it went public, many programmers started using the tool, and it popularized, gaining many contributions and improvements. For the company, it was a great deal, since nowadays everyone uses their library. Something similar could happen if one quantum programming language gained more attention of the programming community, and then the owner company would have great influence and competitive advantage in this market.

2. Large Integer Factorization

Starting with the large integer factorization problem, the quantum algorithm developed by Peter Shor (and thus named Shor's algorithm) transform the mathematical problem that was hard for a classical computer into a task considerably easier.

The time complexity of Shor's algorithm to factorize a semiprime (integer obtained by multiplying two primes) is $O((log\ n)^2(log\ log\ n)(log\ log\ log\ n))$. For the classical computing, on the other hand, the most efficient solution to factorize large integers ($n > 10^{10}$) comes from the **general number field sieve**, which runs in $O(exp((\sqrt[3]{\frac{64}{9}} + o(1))(log\ n)^{\frac{1}{3}}(log\ log\ n)^{\frac{2}{3}}))$. This makes the quantum algorithm asymptotically very superior, and would be able to expose the security systems based on RSA cryptography, for example. The comparison of this complexities is exemplified in figure 1.

2.1. Shor's Algorithm Derivation

One famous theorem in the study of numbers theory and mathematics for cryptography is Fermat's little theorem. It states that, if p is prime, for any integer a the expression $a^p - a$ gives an integer multiple of p, and therefore:

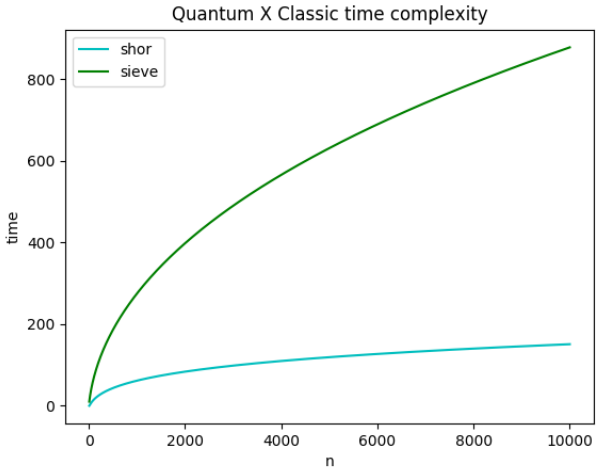

Figura 1. Asymptotic comparison of quantum and classic algorithms for the large integer factorization problem. The curves were designed with python library matplotlib, and use as reference the mathematical expressions for the algorithm, and not sampled data from real running simulations.

$$a^{p-1} \equiv 1 \pmod{p}$$

This popular theorem is an specific case of a more general statement that will be very useful to understand this algorithm's strategy: Euler's theorem. This theorem states that, for any integers a and n relatively coprime (have no common prime factors):

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Where the integer $\phi(n)$, known as Euler's totient, is the number of integers less than n that are coprime to n. Notice that Euler's theorem becomes Fermath's little theorem when n is prime, since any a will also be coprime with n. Furthermore, the exponent $a^{\phi(n)}$ is equal to $n-1$ when n is prime.

Then, applying this theorem to factorize large integer, let's take a large N which is supposed to be a product of two integers. We don't know what are the 2 prime factors, so we take any initial guess g smaller than N. Then, Euler's theorem guarantees that, for some exponent p:

$$g^p - 1 \equiv 0 \pmod{N} \tag{1}$$

$$g^p - 1 = m * N \tag{2}$$

$$\implies (g^{\frac{p}{2}} - 1) * (g^{\frac{p}{2}} + 1) = m * N \tag{3}$$

Let's recall: in the above formula, N is the integer we want to factorize to break an encryption and g is an initial guess for a factor of N (and probably is a wrong guess). What this mathematical manipulation shows is that $g^{\frac{p}{2}} \pm 1$ is a very improved guess for that factor. Since in the right side N appears multiplied by an integer m, then $g^{\frac{p}{2}} + 1$ and $g^{\frac{p}{2}} - 1$ might be the factors of N multiplied by factors of m. So we just need to apply Euclid's algorithm to efficiently find common divisors between $g^{\frac{p}{2}} \pm 1$ and N.

Using a small real example to see the application: let's find the prime factors of 39. Initially, take 5 as poor guess. Since $GCD(5, N) = 1$, we know that 5 does not share prime factors with N. Then next we try $5^2 - 1$, $5^3 - 1$, until we get, for $5^4 - 1$, the value 624, which leaves modulo 0 in the division by 39:

$$5^4 = 625 \equiv 1 \pmod{39} \therefore \tag{4}$$

$$p = 4 \tag{5}$$

$$5^{\frac{p}{2}} + 1 = 26 \tag{6}$$

$$5^{\frac{p}{2}} - 1 = 24 \tag{7}$$

Then 24 and 26 as better guesses for the desired factors. 26 itself is not a factor of $N = 39$, but using Euclid's algorithm for GCD (Greatest Common Divisor), we get 13 as a common factor of both integers. Then $39/13 = 3$ is the other factor (we could also have started with $GDC(24, 39) = 3$ and then find $\frac{39}{3} = 13$). Then we found the answer for the factorization: $39 = 3 \cdot 13$.

$$GCD(26, 39) = 13 \tag{8}$$

$$GCD(24, 39) = 3 \tag{9}$$

$$\therefore 39 = 3 \cdot 13 \tag{10}$$

This is the essential math for Shor's algorithm. Meanwhile, there are some major reasons why this cannot be directly applied in classic computers.

One question that might appear along this algorithm derivation is: why not trying to guess the exponent p of the random guess g by brute force? The answer is that such approach in worst case could be even worse than trying BF to find a prime factor itself, and for sure is worse than the most efficient classic algorithms to find factors. Since the exponent $\phi(n)$ in Euler's theorem (known as Euler's totient) might have an order of magnitude similar to n, guessing the exponent (or half of it) for classic computers is prohibitively slow.

That's where the subject changes from numbers theory to quantum mechanics. A classical computer has to test each value for the guess exponent, then calculate that power, and finally check if it shares factors with the integer N in the encryption. But a quantum computer can create a superposition of all guesses simultaneously in the input, and receive one single output.

In order for the quantum computer to provide the correct desired output, that is, the one that correspond to a prime factor of N, the input must be arranged in a form that all the other inputs destructively interfere.

Let p be the correct power for which g^p is a factor of N. Of course, since we don't know p, we take the guess g and test is to some powers x_i, for many i's. The following property is always valid:

$$g^p \equiv 1 \pmod{N} \tag{11}$$
$$g^{x_i} \equiv r \pmod{N} \tag{12}$$
$$\implies g^{x_i+p} \equiv r \pmod{N} \tag{13}$$

In other words, the power p is also the period for the powers of the guess g modulo N.

Here's how we can use this into Shor's algorithm: first, to test each power of the guess g, we create an array of pairs $[x_i, g^{x_i}]$, where i is in the range from 1 to N. Then, for each pair in the array, we substitute g_i^x by simply $g^{x_i} \pmod{N}$, since the rest in the division by N is all that matters for us to know if the integer is a multiple of N.

Actually, since we have a quantum computer, instead of creating these pairs iteratively in an array-like structure, we can simply create a superposition of all integers in the range from 1 to N, and give it to the quantum calculator to get the superposition of all $g^{x_i} \pmod{N}$.

If the output of this quantum calculator for the superposition input is measured, we will get one single random pair $[x_i, g^{x_i} \pmod{N}]$. But here another quantum property is valid: when the output is measured, it is guaranteed that only the integers that could have generated that output are left, while all the other inputs must have destructively interfered (in quantum physics theory, the measurement act modifies the measured system).

So, if the superposition contained all the pairs from 1 to N and the measured output was some pair $[x_out, g_{out}^x \pmod{N}]$, it is known that the measured system now contains a superposition of only the pairs $x_i, g_i^x \pmod{N}$ that generate the same modulo in the division by N. Analyzing the property demonstrated in equation (11) in reverse direction (the reverse implication is also valid), we know that the powers of g that leave the same rest in the division by N are spaced by a period p, which is the power that we need to find to factorize the key value N (and then Shor's algorithm is solved).

So the final trick in this algorithm is to find a way to figure out the period p in the superposition of numbers that generated the output $x_i, g_i^x \pmod{N}$, for some x_i. Actually, we don't need to find restrictively the period, we can get something that has a bijective relation with p: it's inverse, the frequency f. The tool used to perform this is the Quantum Fourier Transform (QFT).

The mathematical description of the QFT won't be given in details here, but, in words, what this tool does is to get elements in the frequency domain and convert it to the time domain (and vice versa, since the transform has an inverse). For example, let's say it takes a ket vector $|2\rangle$ as input. This value 2 is the frequency of the sinusoidal wave in the output, which is something of the form $\sum cos(2 * i) |i\rangle$. This resembles the classic Fourier Transform. For the quantum version, if the input is a superposition of ket vectors, the output will be the superposition of each individual output. Applying this into Shor's algorithm, since the input now is a set of elements spaced by the period p, the output will be a superposition of outputs that will destructively interfere for all frequencies except $\frac{1}{p}$ (this is a superficial explanation since QFT is not the focus of this work). Then to get the period p we only need to reverse the output of the QFT. The following equation is a

mathematical definition of the Quantum Fourier Transform.

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i \frac{jk}{N}} x_j \tag{14}$$

In the above formula, y_k is the amplitude of the k-th output for the QFT of the j-th input x_j

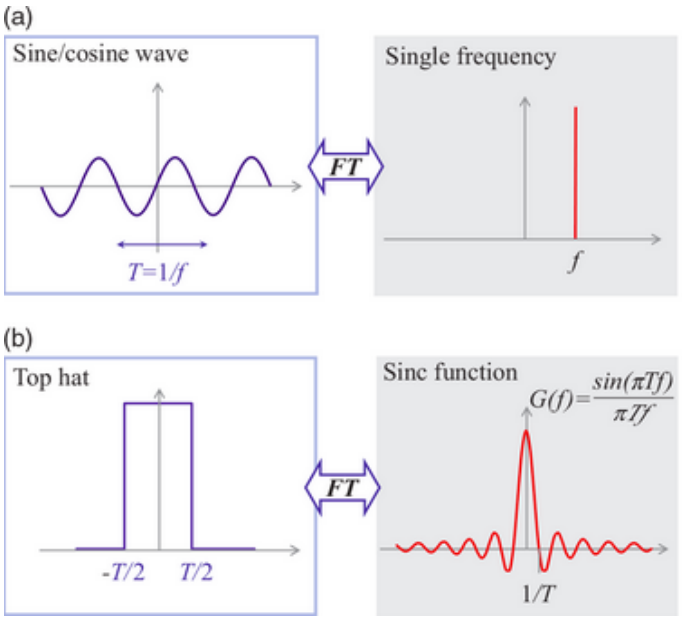

Figura 2. Intuition of how Classic Fourier Transform converts a punctual frequency in it's correspondent wave (a) and converts a range of frequencies into an interference pattern (b). The Quantum Fourier Transform has the same intuition, except for the fact that it is working with a superposition of states instead of a range of frequencies.

The solution seems very good. But, still, some issues might have been unnoticed during the algorithm derivation.

Firstly, let's recall equation (1), where $(g^{\frac{p}{2}} - 1) * (g^{\frac{p}{2}} + 1) = m * N$. In the left side of this equation, one of the factors can be a multiple of N itself, and then no progress would be made from the starting point.

Secondly, the power p to which we need to raise the random guess g and subtract 1 to get a multiple of N might be an odd number, which means that $\frac{p}{2}$ is a fractional number. So, just by trying the integer powers of the random guess g, we will never reach an integer factor of N.

Fortunately, statistically it happens that for almost 40% of the times (it's better approximated as 3/8 = 37.5% of the time), for any random guess g, raising it to 2 and adding or subtracting 1 will actually lead to a factor of N (the reasons for this won't be explored here). So none of the bad cases above will happen, with an reasonable probability, and they should not be a concern for the applicability of the algorithm.

Trials	Failure	Success
1	0.625	0.375
2	0.390	0.609
3	0.244	0.755
4	0.152	0.847
5	0.095	0.904
6	0.059	0.940
7	0.037	0.962
8	0.023	0.976
9	0.014	0.985
10	0.009	0.991

Tabela 1. Probability of getting a proper guess of g to factor N after i trials. The middle column represents the chance that all the trials to guess g will fail after i trials. The last column is the probability of failing i-1 times, and get a good guess in the i-th trial

The table 1 depicts, for the probability of $\frac{3}{8}$ of getting the correct guess g, what is the chance that a correct factor of the key value N will be found after i trials.

Below, all the steps described until here are summarized:

- To break encryption, we need to discover prime factors of N.
- Start by taking a random guess g. If g divides N, we found the answer. Return $\{g, \frac{N}{g}\}$. Else:
- Find a power p so that $g^p - 1$ divides N.
- if p is odd, restart the algorithm.
- If p is even, $g^{\frac{p}{2}} \pm 1$ are much better guesses to share factors with **N**
- If one of the obtained integers is a whole multiple of N, restart the algorithm.
- Else if a factor f is found, return $\{f, \frac{N}{f}\}$. $C \equiv P^e \pmod N$
- Else, use Euclid's algorithm to check if one of the numbers share a factor f with N. Return $f, \frac{N}{f}$

Let's analyze the time complexity of the above implementation . Creating the superposition states in the input takes $O(d)$ by using the Hadamard matrix, where d is the number of digits in N (also proportional to $log(N)$). Raising integers to some power p and taking the modulo N can be performed in $log(p)$ and $O(1)$, respectively. The Quantum Fourier Transform applied to the remaining superposition also depends on the number of digits to represent the input. Finally, the Euclid's GCD has time polynomial in $log^2(max(N, \frac{p}{2}))$. Since each of the above mentioned steps is done sequentially, the overall asymptotic time complexity is the expression with maximum order among those.

In this case, a careful analysis show that the resulting time is polynomial in $log(N)$. This makes Shor's quantum algorithm to factorize integers tremendously faster than the most efficient known classic algorithm for the same task.

So, that's the description of Shor's algorithm. As soon as larger quantum processors are available, this solely is enough to break all Large Integer Factorization based cryptography on the internet, which represents a **substantial** part of current security systems.

A very straightfoward case of application is in the popular RSA publlic key encryption. This system is base on two hard mathematical problems, that are thought to be unsolvable in feasible time by classic computers: the problem of factoring large numbers and the RSA problem (that involves raising the message to be transmitted to an exponent, and then taking the modulo N of the result, where N is the integer whose factors are unknown):

$$C \equiv P^e \pmod{N} \tag{15}$$

Where the cypher C is the encrypted message, P is original message (plain text) which can't be efficiently computed, N is a Large integer, product of two large prime factors, and e is some integers coprime to $\phi(N)$ (reminder: $\phi(N)$, known as Euler's totient, is a function that gives the number of positive integers less than N that are coprime to N). Shor's algorithm can be used here to factorize the semiprime (product of two primes)N.

To finish this section, a real simulation of the algorithm is analyzed. For this simulation, we used the Quantum Computing Playground, Google's open environment for quantum programming. The environment compiles and runs the language QScript, also developed by the company, and offers various other resources, like tutorials, quantum circuits and gates simulators, templates and sample codes. The simulator still can offer only 22 qubits, and therefore our analysis is limited to test the factorization of small integers only (quantum registers must be use to store not only the input and output, but also some intermediate values).

In table 2 we show the output and logs of the algorithm running with the example mentioned in the beginning of this section: factorize 39.

3. Grover's algorithm

Grover's algorithm was created in 1996 to perform search in unstructured databases with quadratic speed up when compared with classical algorithms. It is expected that such algorithm gains great importance in the future. With more and more data coming out, and with IoT (Internet of Things) on the rise, it becomes difficult to keep data organized in structured databases. So the quantum strategies to perform operations in database like search considerably faster than classical computers are very attractive. A more general way of describing Grover's algorithm application is to say that it discovers the input that generates a given output of a black box function.

From the cyber security perspective, this could break many sorts of hash based encryption systems, such as SHA-256 (in which bitcoin is based), scrypt (base of lite-

Success	.
Initial guess:	11
Qubit number out of range:	16
Qubit number out of range:	17
Qubit number out of range:	18
Qubit number out of range:	19
Qubit number out of range:	20
Measured	10923 (170.671875)
fractional approximation is	-512/-3
Possible period is	-3
Unable to determine factors, try again.	
Measured	8192 (128)
fractional approximation is	128/1
Odd denominator, trying to expand by	2.
Possible period is	2

Tabela 2. Log from the simulation for factorizing 39. As mentioned, the simulator has only 22 qubits. In this particular implementation (code is available in the Apendix section), the size of vetor to store values were limited to 16, so qubits from 16 and above will result in 'Qubit number out of range' log. The steps mentioned in the algorithm description throughout this section can be identified in the logs: the initialization guess (11); the first measurement correspond to an odd exponent, giving fractional value when divided by two. Then, in the second trial, the period 2 is identified. It means that $11^{\frac{2}{2}} \pm 1$ is a better guess for some factor. Since $11^{\frac{2}{2}} + 1 = 12$ shares the factor 3 with N, then the other factor is $\frac{39}{3} = 13$ and we found the answer: $39 = 3 \cdot 13$

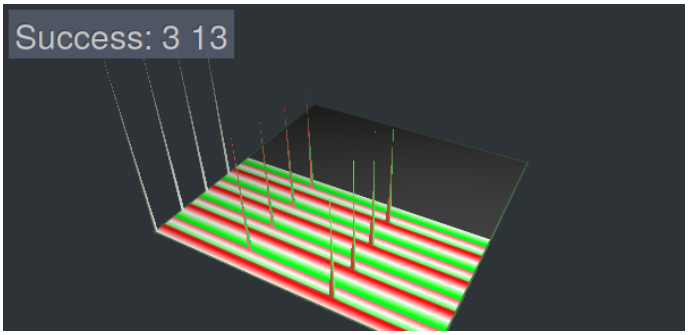

Figura 3. Graphic output from Shor's algorithm factorization of the semiprime 39, provided by the Quantum Computing Playground

coin), and MD5 (**Obs:** though MD5 is no longer in largely used specifically for security purpose for being considered too vulnerable, it is still applied for data integrity verification functions like checksum, and can lead to attack surfaces) for example. These hash based encryptions work with the fact that, given an input, the hash function converts it into an output that can't be reverted back to the input. So only someone that knew the original input has the key to the encryption. The hash function is public and well know, so the system is the exact definition of Grover's algorithm application: given a black box function and it's output, find the input that generated that output.

So in this section the Grover's search algorithm will be discussed in more details. A classic algorithm to search an specific element in an unsorted list has to look at each element by brute force. This is in worst case $O(n)$ (and looks at $\frac{n}{2}$ elements in average). Grover's algorithm can reduce this to $\sqrt{n}$, which is a substantial reduction in time (quadratic reduction), that can only be performed with quantum computing. However, it is also proved that this $\frac{n}{2}$ is also the efficiency limit for a quantum algorithm.

This algorithm also does not take into consideration list data structure's specific properties, so it's application extends to more general cases. Though the speed up that Grover's algorithm offer for database search is not as critic as Shor's speed up for factorization, the impact of this algorithm is also very relevant.

As Shor's algorithm analyzed in the last section had an interesting algebraic derivation, this algorithm has a good geometrical interpretation. It also starts with creating a superposition of states with the Hadamard transform (referred as the initialization phase).

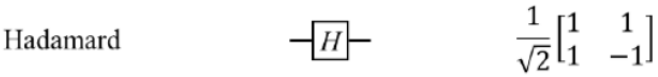

Figura 4. Hadamard gate and matrix used to create a superposition of states as an input for the algorithm

The second phase of the algorithm is called the Grover's Iteration, and it consists of applying an Oracle to the list. The oracle is a black box function that can operate in the quantum superposition system without collapsing it to a classical state.

First, the following oracle matrix operation is applied:

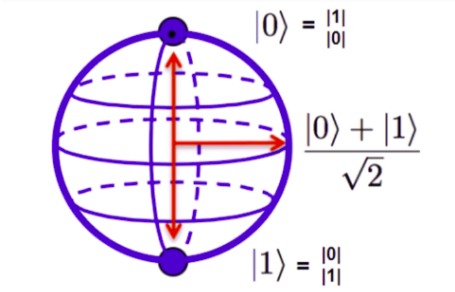

Figura 5. Intuitive illustration of how Hadamard matrix operates to convert 0 and 1 vector into a superposition state. The north pole in the figure represents 0 and the south is 1. The horizontal state is the superposition.

$$U_w |x\rangle = (-1)^{f(x)} |x\rangle \tag{16}$$

This operations reverses the amplitude of the matching element in the list, and leaves all other positions unchanged. The function f passed to the oracle is defined as $f(x) = 1$ if the searched element is in the position x, and $f(x) = 0$ otherwise.

The last operation inverts the signal of the amplitude in the correct index, but does nothing to it's module. For this reason, if we measure the superposition at this point, the probability of getting any index as answer is the same, since the amplitudes are still the initial ones. This probability is $\frac{1}{N}$ or $\frac{1}{2^n}$, where the lowercase n is the number of bits needed to represent the positions in the list. This quantum state is defined as:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \tag{17}$$

which is denominated a uniform superposition quantum state.

So let's call the initial superposition state $|s\rangle$ and the superposition after the oracle matrix is $U_w |s\rangle$. Next, the oracle applies a reflection around the previous superposition state $|s\rangle$ in the vector space (the state $|s\rangle$ must be kept in a variable). This operation will be denoted U_s and the resulting state is now $U_s U_w |s\rangle$. This reflection is what causes the quantum state to get it's amplitude amplified in the direction of the correct answer, and contracted in all other directions. This operations become more clear when visualized graphically in figure 6.

The figure shows the first step in Grover's iteration, which amplifies the superposition resulting vector in the directions of the correct answer, and shrinks it's components in other directions. After performing this step once, if we measure the superposition, the probability of the observed value to collapse in the correct index is greater than any other individually, but, of course, it's still not statistically trustworthy.

In order to reach acceptable certainty that the observed state is the amplified direction, the number of iterations needed is proportional to the square root of number of elements n. More precisely, the complexity of the algorithm is proportional to $\frac{\pi}{4} * \sqrt{2n}$. The $\frac{\pi}{4}$ comes from the fact that in the initial superposition all qubits are $\frac{\pi}{4}$ distant from

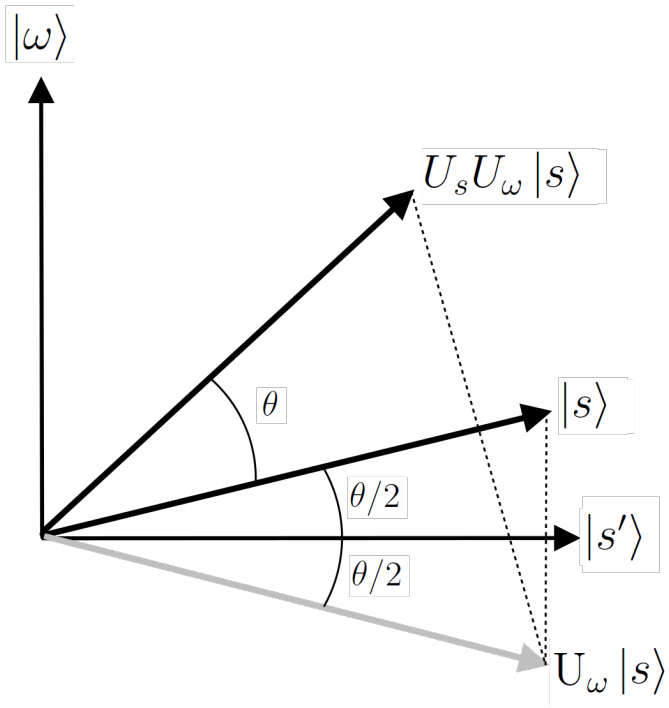

Figura 6. Grover iteration graphic interpretation. The state $|w\rangle$ (conventionally denominated winner position) is the correct answer and $|s'\rangle$ is the superposition minus it's component in $|w\rangle$'s direction. The first oracle matrix reverses the superposition $|s\rangle$ around $|s'\rangle$ and the second operation reflects the resulting state around the original $|s\rangle$. As it can be noticed in the figure, this approximates the resulting state vector of the correct direction, i.e., the component in $|w\rangle$ increases and all other components decrease (it is shown in 2 dimensions, but the logic works for n dimensions).

0 or 1, an then they'll have to be rotated until the final iteration. The $\sqrt{n}$ comes from the fact that, in the beginning, all amplitudes are $\frac{1}{\sqrt{n}}$, so that the probability of measuring each qubit is uniformly equal to $\frac{1}{n}$. Since each complete iteration step increases linearly the amplitude of the winner direction $|w\rangle$, the probability of measuring increments in the square of that rate. So $\sqrt{n}$ iterations are enough to cause the probability to increase proportional to n.

Finally, we can measure the resulting state: $|s_{final}\rangle = (U_s U_w)^{\frac{\pi}{4} * \sqrt{2n}} |s\rangle$. This measurement, with trustworthy certainty, will cause the final superposition to collapse to the correct position for the searched element.

So here is the overall view of algorithm's strategy:

- We need to search an element in an unstructured list.
- Start by creating a superposition in a register, where each qubit represents an index in the list.
- **Important:** the oracle will perform operations in the qubits that maximize the probability of the correct state being measured without collapsing the quantum superposition. But, for this to be possible, we can't look at the system until the

final step of the algorithm.

- Apply the oracle matrix $U_w |x\rangle = (-1)^{f(x)} |x\rangle$ to the qubits, responsible for reversing only the amplitude corresponding to the correct index, and leaving the other indexes unchanged.

- Apply the reflection U_s of the resulting state ($U_w |s\rangle$) relative to the direction of the previous $|s\rangle$.

- Repeat the Grover iteration $\frac{\pi}{4} * \sqrt{2n}$ times and obatain $(U_s U_w)^{\frac{\pi}{4} * \sqrt{2n}} |s\rangle$.

- measure the final state. The value observed is reliably the correct winner position for the searched element.

As mentioned, although the described implementation was originally designed as a search algorithm, Grover's algorithm is more generally seen as a function inverter algorithm. Some authors describe Grover's task as: given the output that comes out of a black box function, find the input that generated it.

Grover's algorithm is also applicable to solve the *collision problem*. This problem consists in finding the minimum number of operations needed to certainly discover if, for a given function f, each output y can be generated by a single input x (the function f is injective) or if there are r different x's that map to each y (called $r - to - 1$ function). This problem is closely related to cryptography, since hashing functions must have a probability of collision as small as possible in order to be considered good to hide and authenticate information.

The function reverser algorithm could be used to reverse hash functions within practicable time, exposing passwords and many sorts of authentication based security. This also has implications in the blockchain technology, since each block authenticates the previous block's identity through it's hash. Therefore, block forgery becomes possible.

Additionally, in the search algorithm, if there is more than one instance of the desired element in the database, and the number of matches is known, the algorithm can be optimized.

Grover's algorithm can be found explained in details in IBM's Quantum Experience documentation. Quantum computing is a hot topic at the moment, and many quality materials are available online. Also, although for someone having it's first contact with quantum computing this algorithm (and also Shor's algorithm discussed in the last section) might appear a little futuristic, or very hard to implement with nowadays tools, they are already a reality, easy to implement with tools available open source. The only real limitation in present technologies is scalability, i.e., the accessible simulators and open cloud quantum computing services contain only a few qubits for now. But it's a matter of time until quantum technology becomes more accessible for worldwide programmers.

4. Conclusion

Quantum computing has been a promise for the future and is now popular not only in academy. The leading companies in the technological industry, as well as governments,

are hurrying not to be left behind in the race to dominate quantum technology.

The algorithms analyzed in this work are the most popular in this initial phase of propagation of quantum computing interest. They demonstrate how the smart and strategic use of the extra laws of physics possible with these computers might speed up important and recurrent tasks in a asymptotic scale, which is the reason why they are so impacting.

Shor's algorithm's impact is so huge that a substantial part of all Internet security is considered to be exposed as soon as bigger quantum processors are available to the public.

Grover's algorithm, on the other hand, is less impacting due to it's smaller reduction in the classical analogue's complexity. Nevertheless, the algorithm has huge expectations for the future, with the raise of IoT era. In an environment where tons of data are generated at every moment, an efficient approach to perform search is a key skill for any process.

Since theses algorithms improvement is in terms not only of constants, but of time complexity, straightforward strategies like increasing the size of cryptographic keys, developing best hash functions or using more powerful hardware for defense would not be effective. This obligates the defensive side to develop quantum resistant algorithms, which act through different routes that can't be violated by quantum strategies. This is giving rise to the Post Quantum Cryptography as a countermeasure.

5. References

Referências

Aaronson, S. (2004). Limits on efficient computation in the physical world. *arXiv preprint quant-ph/0412143*.

edX, B. (2013). Qft, period finding & shor's algorithm. *University of California*.

Experience, I. Q. (2017). Grover's algorithm.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING*, pages 212–219. ACM.

Knill, E. (1996). Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States).

Lin, F. X. (2014). Shor's algorithm and the quantum fourier transform. *McGill University*.

Naihin, S. (2019). Breaking cryptosystems with quantum computers.

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring.

University, S. P. S. (2019). The introduction to quantum computing.

6. Apendix

This section contains the simulated Shor Algorithm in QScript. This is a modification of Google's Quantum Computing Playground to fit the format of the algorithm described in this work. The code is commented and the name of the variables are the same used in section 2. **Obs:** the code below contains text formatting for better visualization. So if it is

Algorithm 1 Shor's algorithm to factorize a semiprime N into it's two prime integers.

```
1:  proc Factorize N
2:  // the method getWidth return the number
3:  // of qubits needed to store N
4:  N_bits = QMath.getWidth(N)
5:  total_bits = 2 * N_bits + 3
6:
7:  // create the initial random guess for a factor
8:  g = 0
9:  for g; (QMath.gcd(N, g) > 1||(g < 2); g
10:     g = Math.floor(Math.random() * 10000)%N
11: endfor
12:
13: Print "Initial guess: "+ g
14:
15: // create a superposition of states in each
16: // digit using the Hadamard transform
17: for i = 0; i < total_bits; i++
18:     Hadamard i
19: endfor
20:
21: // calculates g**total_bits(modN)
22: ExpModN g, N, total_bits
23: // measuring bits cause the superposition to colapse into
24: // only the inputs correspondednt to the measured output
25:
26: // since we took the modulo N in the previous step, we
27: // only need to measure up to N_bits, since all numbers
28: // modulo N will be in range from 0 to N.
29: for i = 0; i < N_bits; i++
30:     MeasureBit total_bits + i
31: endfor
32:
33: // the reversed of the QFT will turn the p-spaced periodic
34: // superposition in the input into the 1/p frequency in the output
35: InvQFT 0, total_bits
36:
37: // inverse the digits, since the output
38: // in last step was in reversed order
39: for i = 0; i < total_bits/2; i++
40:     Swap i, total_bits - i - 1
41: endfor
42:
```

```
43: // as explained along the theory, the initial guess might be a bad
44: // value, with 3/8 chance of being a good choice. Then, it is very
45: // likely that the correct result is achieved within the first 10 trials
46: for trials = 100; trials >= 0; trials - -
47: Measure
48: c = measured_value
49:
50: p = 1 << N_bits
51: Print c
52: Print p
53: Print "Measured "+ c + "("+ c / p + ")"
54:
55: tmp = QMath.fracApprox(c, p, N_bits)
56:
57: c = tmp[0];
58: p = tmp[1];
59:
60: Print "fractional approximation is "+ c + "/"+ p
61: // after the QFT, the measurement is in the frequency domain, and
62: // we wish that the denominator corresponds to the period
63: if (p%2 == 1)&&(2 * p < (1 << N_bits))
64:     Print "Odd denominator, trying to expand by 2."
65:     p *= 2
66: endif
67:
68: // if the denominator is odd and can't be multiplied by 2 in N_bits,
69: // then we had a bad guess g and should repeat the algorithm
70: if p%2 == 1
71:     Print "Odd period, try again."
72:     continue
73: endif
74:
75: Print "Possible period is "+ p
76:
77: // if the period is even, we can calculate gˆ (p/2) +- 1
78: // to get good guesses for factors of N
79: a = QMath.ipow(g, p/2) + 1%N
80: b = QMath.ipow(g, p/2) - 1%N
81:
82: // a and b might not be factors of N, but share factor with N
83: a = QMath.gcd(N, a)
84: b = QMath.gcd(N, b)
```

```
85: if a > b
86:         factor = a
87: else
88:         factor = b
89: endif
90:
91: if (factor < N)&&(factor > 1)
92:         Display «h2 > Factorization : " + factor + " " + N/factor
93:         Breakpoint
94: else
95:         Print "Unable to Factorize. Please try again."
96:         continue
97: endif
98:
99: endfor
100:
101: endproc
```

copied and pasted in this exact way in the compiler, it won't work. The pure code (compiling and functional) used is available in the author's github: Lucas1Jorge/Shor.qscript.

Next, we present the mentioned Euclid's algorithm for calculating the Greatest Common Divisor between two integers, used when the obtained values for $g^{\frac{p}{2}} \pm 1$ is not a factor of N, but shares factors with it.

Algorithm 2 Euclid's algorithm for Greatest Common Denominator

```
procedure Euclid (a, b)
    // Input: Two non negative integers a and b
    // Output: gcd(a, b)
    if b == 0
        return a
    else
        return Euclid(b, a%b)
```