Semiprime factorization methods

Gabriele Colapinto

Faculty of computer engineering
of the Polytechnic university of Bari
(Poliba)

Exam project of the subject:
Formal Languages and Compilers

Teacher: Floriano Scioscia

The semiprime factoring problem

The RSA algorithm is based on the product of two very large prime numbers $N = p \cdot q$. p and q are the private key and N is the public key which can be transmitted in clear because it is difficult for a cryptanalyst to derive the two prime numbers starting from their product.

This problem has been widely studied because the security of online communication depends on it and researchers have devised various algorithm to try to break the security offered by the RSA algorithm.

Among these algorithms there are brute force, Fermat's factorization and quadratic sieve which require a prohibiting amount of time to be executed and there is also Shor's algorithm which became popular for being faster than the others if run on a quantum computer.

Contents

The semiprime factoring problem 1

Brute force algorithm 3

Python implementation of the algorithm 4

Time complexity of the algorithm 4

Fermat's factorization algorithm 6

Python implementation of Fermat's factorization algorithm 7

Time complexity of Fermat's factorization algorithm 9

Quadratic sieve algorithm 10

Python implementation of the quadratic sieve algorithm 13

Optimization of the quadratic sieve algorithm 20

Time complexity of the quadratic sieve algorithm 22

Shor's algorithm 23

Implementation of Shor's algorithm in a quantum computer 27

Basics of quantum computing 27

Qubits 27

Superposition of qubits 28

Measuring a qubit 28

Quantum Fourier Transform 29

Generalized QFT for one qubit 29

Generalized QFT formula for strings of qubits 30

QFT using a matrix 31

QFT circuit 33

QFT circuit example 33

Inverse Quantum Fourier Transform 34

Inverse Quantum Fourier Transform of a single qubit 34

Inverse QTF using a matrix 35

Quantum phase estimation 36

Quantum circuit that implements Shor's algorithm 38

Python implementation of the algorithm 41

Time complexity of Shor's algorithm 50

References 52

Brute force algorithm

The brute force algorithm tries to guess the two factors of N by trying to divide it by all integer numbers starting from 2 and increasing the divisor in each iteration. The concept on which this algorithm is based is very simple: if we divide two numbers and the carry of the division is zero it means that the divisor is one of the two factors we are looking for. Once we find one factor, we can calculate the other dividing N by the first factor. The fact that the divisor increases gradually implies that the first factor found is the smallest of the two and this suggests us that in the worst-case scenario the algorithm increases the divisor up to $\sqrt{N}$.

Let us see three examples to better understand this concept.

1) One factor is much smaller than the other:
$$p = 7, \qquad q = 4289, \qquad N = 30023, \qquad \sqrt{N} \cong 173.27$$
In this case the divisor increases only up to 7. This example shows us that the good cases for the execution of this algorithm are the ones in which one factor is small.
2) The two factors are similar:
$$p = 27361, \qquad q = 27239, \qquad N = 745286279, \qquad \sqrt{N} \cong 27299.93$$
In this case the divisor increases up to 27239. This number is not so smaller than the $\sqrt{N}$.
3) The two factors are equal:
$$p = x, \qquad q = x, \qquad N = x^2, \qquad \sqrt{N} = x$$
This is the theoretical worst-case scenario. Regardless of the number x, in this case the denominator increases up to $\sqrt{N}$.

Python implementation of the algorithm

Let us see an implementation of this algorithm in Python:

```
import numpy as np


N = 745286279 #Number to be factored

f1 = f2 = 1 #Factors of N

found = False #Variable telling whether the first factor has been found


while (not(found)):

  f1 += 1

  if(np.mod(N, f1) == 0):

    found = True


#At this point the first factor has been found

f2 = N/f1

print("The two factors of %d are %d and %d\n" % (N, f1, f2))
```

Script 1

Time complexity of the algorithm

Let us now try to calculate the time complexity of this algorithm referring to the number of bits n the number N is made of.

This calculation is fairly simple, we have to multiply the number of iterations for the time complexity of the division of two binary numbers of n bits.

To understand how many iterations the algorithm can make we have to consider that it depends solely on the divisor which starts from 2 and increases up to $\lfloor\sqrt{N}\rfloor$. So it is $\lfloor\sqrt{N}\rfloor - 1$ because we have to exclude the iteration in which the divisor is 1.

The time complexity of the division of two binary numbers of n bits is $O(n^2)$.

At this point it is useful to consider that a number of n bits can range up to $2^n - 1$. For instance, with 3 bits we can range from 0 to 7 which is $2^n - 1$ so in the big-O notation we can say that:

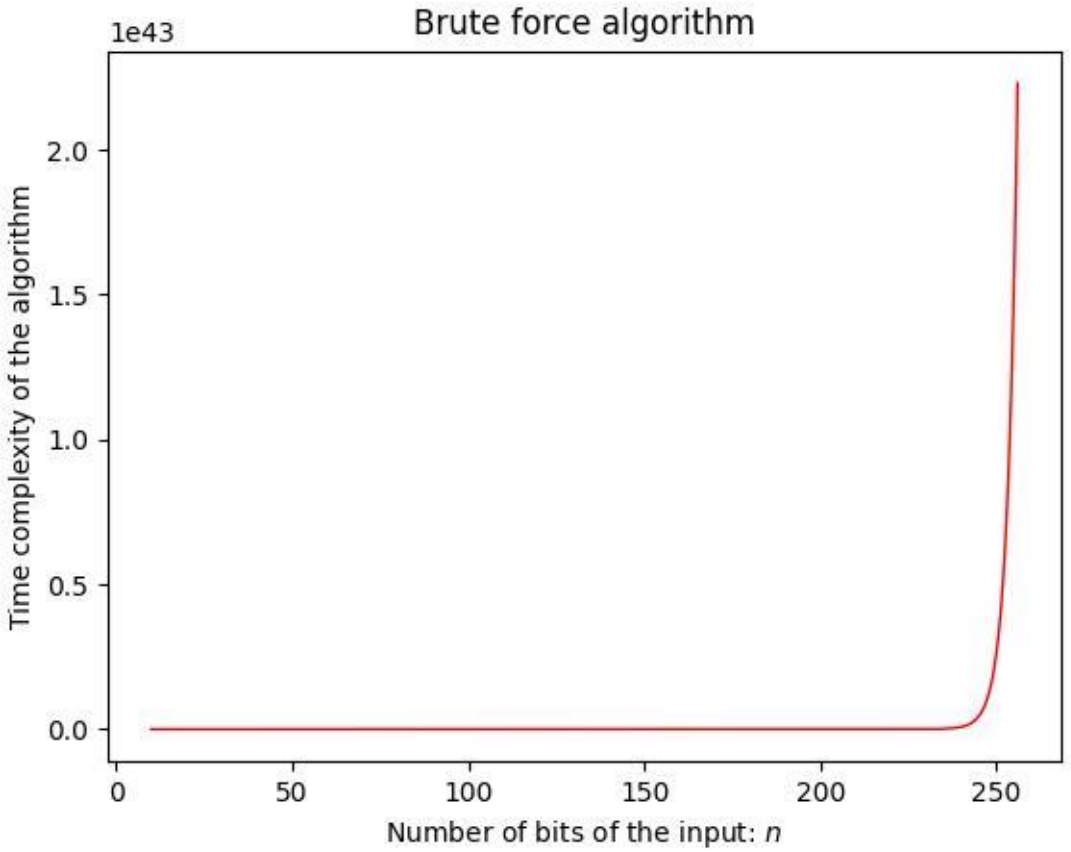
$$O(N) = O(2^n)$$

If we put everything together, we have that the time complexity of the brute force algorithm is:

$$\left(\lfloor\sqrt{N}\rfloor - 1\right) \cdot O(n^2)$$

Considering that in the big-O notation we consider theoretical values and we remove the constants, the time complexity becomes:

$$O\left(\sqrt{N}\cdot n^2\right) = O\left(2^{\frac{n}{2}}\cdot n^2\right) = O(\sqrt{N}\cdot(\log_2 N)^2)$$



Plot 1: Time complexity of the brute force algorithm

From the plot we can see that if we try to use the brute force algorithm to factor a small public key of 256 bits the order of magnitude of the time complexity becomes 10^{43}!

Fermat's factorization algorithm

Fermat's factorization algorithm is another example of simple factorization algorithm. It is based on the identity:

$$x^2 - y^2 = (x - y) \cdot (x + y)$$

If we find two numbers such that:

$$x^2 - y^2 = N, \qquad x, y \in \mathbb{N}$$

It means that the two factors of N are:

$$p = x - y$$

$$q = x + y$$

In order to find the two numbers, let us first rewrite the relation:

$$x^2 - y^2 = N \Leftrightarrow x^2 - N = y^2$$

Considering that square numbers are greater or equal to zero we set the initial value of x as the ceiling approximation of the square root of N. We compute $x^2 - N$ and check if the result is a perfect square different from 0. If not, we increase the value of x by 1 and reiterate the process until we find a perfect square.

Let us see an example of this algorithm:

$$N = 33$$

- Iteration 0:

  $$x = \lceil \sqrt{N} \rceil = 6$$
  $$y^2 = x^2 - N = 36 - 33 = 3$$

  y^2 is not a perfect square.
- Iteration 1:

  $$x = 7$$
  $$y^2 = x^2 - N = 49 - 33 = 16 = 4^2$$

  y^2 is a perfect square.

Now that we have found x and y we can calculate p and q:

$$x = 7, \qquad y = 4$$

$$p = x - y = 3$$

$$q = x + y = 11$$

Python implementation of Fermat's factorization algorithm

The first thing to do is to import the necessary functions from the math library and initialize the variables:

```
from math import sqrt, ceil

#Parameters

N = 91 # Number to be factored


# Loop variables

x_base = ceil(sqrt(N))

x,y_2 = 0,0

result_x, result_y = 0,0


#Two prime factors of N

p,q = 0,0
```

Script 2: Snippets 1 and 2

Now that we have initialized the variables, we define the loop which is the core of the algorithm:

```
# We look for two numbers x and y such that x^2 - N = y^2 and y^2 is a square of an integer

for x in range(x_base, N):

    y_2 = x**2-N

    temp = sqrt(y_2)

    if(float.is_integer(temp)):

        result_x = x

        result_y = int(temp)

        break


print("x = %i and y = %i" % (result_x, result_y))
```

Script 2: Snippet 3

In the loop the x can vary from $\lceil \sqrt{N} \rceil$ to N because we want to make the algorithm as generally applicable as possible. The case in which the x reaches N is the case in which N is a prime number. In every iteration we calculate y^2 and we check if its square root is an integer. If so, we can assign the values of x and y to the result variables and interrupt the loop.

At the end of the snippet, we print the values of x and y. In this case, for $N = 91$, the snippet produces the following output:

x = 10 and y = 3

The last step is to calculate the factors of N:

```
#We calculate the two prime factors of N

p = result_x + result_y

q = result_x - result_y


print("The two factors of %i are %i and %i" % (N, p, q))
```

Script 2: Snippet 4

This snippet produces the following output:

The two factors of 91 are 13 and 7

Time complexity of Fermat's factorization algorithm

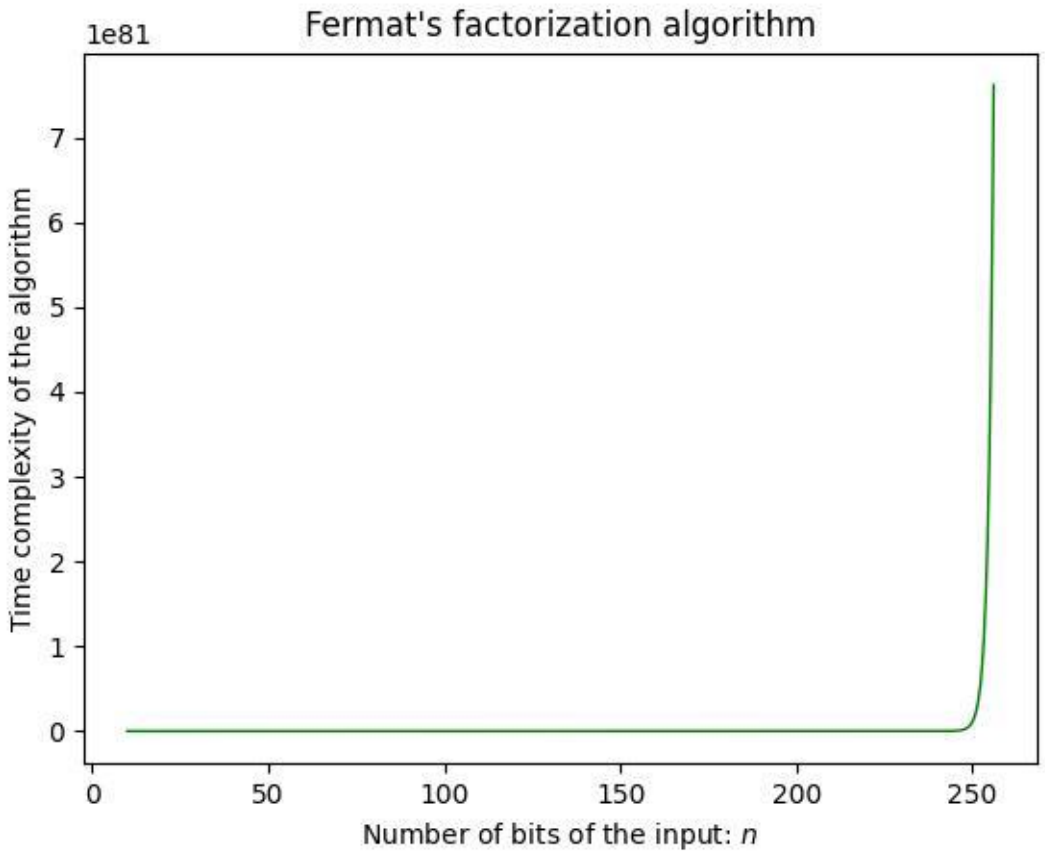
By the implementation of the algorithm, we can see that in the worst-case scenario, that is the case in which N is a prime number, the algorithm iterates $N - \lceil \sqrt{N} \rceil$ times. In each iteration we perform a squaring and a square root which have a time complexity of respectively $O(n^2)$ and $O(\log(N))$. The other operations have a time complexity of $O(1)$ so we will not take them into account.

In conclusion the time complexity of the algorithm is:

$$O\left(\left(N - \lceil \sqrt{N} \rceil\right) \cdot (n^2 + \log_2 N)\right)$$

If we consider that in the big-O notation $O(N) = O(2^n)$ we get:

$$O\left(\left(2^n - 2^{\frac{n}{2}}\right) \cdot (n^2 + \log(2^n))\right) = O\left(\left(2^n - 2^{\frac{n}{2}}\right) \cdot (n^2 + \mathrm{n})\right)$$



Plot 2: Time complexity of the Fermat's factorization algorithm

From the plot we can see that if we try to use the Fermat's factorization algorithm to factor a small public key of 256 bits the order of magnitude of the time complexity becomes 10^{81}!

Quadratic sieve algorithm

The quadratic sieve algorithm is an evolution of Fermat's algorithm. It is based on the idea that the difference of two squares can be a multiple of N.

$$x^2 - y^2 = kN, \qquad k \in \mathbb{N} \wedge k > 1$$

We need k to be greater than 1 or else we would be using Fermat's algorithm.

We can rewrite the relation as follows:

$$x^2 - kN = y^2$$

Now, if we use modular arithmetic, we can say that:

$$y^2 = x^2 \ (mod\ N)$$

If we find two integers such that the relation is satisfied, we can say that the two factors of N are:

$$p = \gcd(x - y, N)$$

$$q = \gcd(x + y, N)$$

This algorithm can fail if $x - y$ or $x + y$ are factors of N. In this case the result of the greatest common divisor would be $x - y$ or $x + y$.

In order to have a better understanding of the algorithm let us make an example:

$$N = 20791$$

Just like in Fermat's algorithm, the base value of x is the ceiling approximation of the square root of N:

- Iteration 0:

$$x = \lceil \sqrt{N} \rceil = 145$$
$$x^2 = 21025$$
$$y^2 = x^2 (mod\ N) = 234$$

  y^2 is not a perfect square, so we increase x by 1 and iterate again.

- Iteration 1:

$$x = 146$$

$$x^2 = 21316$$

$$y^2 = x^2 (mod\ N) = 525$$

  y^2 is not a perfect square, so we increase x by 1 and iterate again.

- Iteration 2:

$$x = 147$$

$$x^2 = 21609$$

$$y^2 = x^2 (mod\ N) = 818$$

  y^2 is not a perfect square, so we increase x by 1 and iterate again.

- Iteration 3:

$$x = 148$$

$$x^2 = 21904$$

$$y^2 = x^2 (mod\ N) = 1113$$

y^2 is not a perfect square, so we increase x by 1 and iterate again.

$$\vdots$$

It may seem like this algorithm requires many iterations to lead to a result because we have to find a perfect square. Actually, we can interpret these results differently to make the algorithm much faster.

Let us first store the relations we have found as follows:

$$R_0 = [x_0^2, y_0^2]$$

$$R_1 = [x_1^2, y_1^2]$$

$$R_2 = [x_2^2, y_2^2]$$

$$\vdots$$

$$R_k = [x_k^2, y_k^2]$$

Let us break down in prime factors the values of y^2:

$$y_0^2 = 234 = 2 \cdot 3 \cdot 3 \cdot 13 = 2^1 \cdot 3^2 \cdot 13^1$$

$$y_1^2 = 525 = 3 \cdot 5 \cdot 5 \cdot 7 = 3^1 \cdot 5^2 \cdot 7^1$$

$$y_2^2 = 818 = 2 \cdot 409 = 2^1 \cdot 409^1$$

$$\vdots$$

If we find enough relations, we can select some of them such that if we multiply them together, we can have a product of prime factors with an even exponent.

For instance, let us say that we have found three relations we can use for this operation:

$$y_i^2 = p^1 \cdot q^2 \cdot r^3$$

$$y_j^2 = p^1 \cdot r^1 \cdot s^1$$

$$y_k^2 = p^2 \cdot q^4 \cdot s^1$$

If we multiply these relations, we get:

$$x_{product}^2 = x_i^2 \cdot x_j^2 \cdot x_k^2$$

$$y_{product}^2 = y_i^2 \cdot y_j^2 \cdot y_k^2 = p^4 \cdot q^6 \cdot r^4 \cdot s^2 = (p^2 \cdot q^3 \cdot r^2 \cdot s)^2$$

The product is a perfect square, so the product of the relations satisfies the initial relation:

$$y_{product}^2 = x_{product}^2\ (mod\ N)$$

It means that it can be used to compute the greatest common divisor:

$$x_{product} = \sqrt{x_{product}^2}$$

$$y_{product} = \sqrt{y_{product}^2}$$

$$p = \gcd(x_{product} - y_{product}, N)$$

$$q = \gcd\,(x_{product} + y_{product}, N)$$

Now that we understood the general idea of the algorithm, let us expose the problems of its implementation.

First of all, a computer cannot consider all prime numbers up to infinity. What we do in the implementation of the algorithm is choose an upper bound B for the prime numbers and compute the primes up to B.

For instance, let us choose $B = 50$. The primes up to B are:

$$primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]$$

Choosing a set of primes implies that we have to sieve out the relations in which the y^2 value cannot be broken down completely using the primes in the set. For instance:

$$y_2^2 = 818 = 2 \cdot 409 = 2^1 \cdot 409^1$$

This relation has to be discarded because 409 is not included in the primes array.

When a number can be broken down completely using the primes in the array, we can say that the number can be factored smoothly over the factor base. From this statement we can derive the technical jargon used henceforth in the exposition.

The jargon is:

- A number which can be broken down completely using the primes in the array is called a “smooth number”.
- The relations in which the y^2 is a smooth number are called “smooth relations”.
- The upper bound B can be also referred to as the “smoothness bound”.

Let us say that we kept only the smooth relations. How do we find one or more combinations of relations such that their product can be used to compute p and q?

We have to break down the y^2 values in prime factors and form a matrix in which every row represents a relation and every column a prime number.

$$M \in \mathbb{N}^{\,n_{relations} \times n_{primes}}$$

The elements of the matrix are the exponents of prime factors which make the y^2 value.

Let us make an example of a row considering that we have fifteen primes and considering the following relation:

$$y_1^2 = 525 = 3 \cdot 5 \cdot 5 \cdot 7 = 3^1 \cdot 5^2 \cdot 7^1$$

$$[0 \quad 1 \quad 2 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$

The information we are actually interested in is not the exponent but its parity. So, we have to change the matrix in a way such that its elements are 0 if the number is even and 1 if the number is odd.

So, our row example becomes:

$$[0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$

Once we build the matrix, we have to find a way to calculate the combinations of rows such that the product of the corresponding relations can be used to compute p and q.

This can be done calculating the left null space of M. The left null space of a matrix is a column vector S such that:

$$\underbrace{M^T}_{n_{primes} \times n_{relations}} \times \underbrace{S}_{n_{relations} \times 1} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{n_{primes} \times 1}$$

Now, we can transpose S and use it to compute the linear combination of rows such that its result is a row vector of zeros.

$$\underbrace{S^T}_{1 \times n_{relations}} \times \underbrace{M}_{n_{relations} \times n_{primes}} = \underbrace{[0 \quad 0 \quad \cdots \quad 0]}_{1 \times n_{primes}}$$

In order to find one or more S vectors we need to have a number of relations greater than the number of primes. For this reason, it is a good practice to set the number of relations as a multiple of the number of primes. For instance, we could set:

$$n_{relations} = 6 \cdot n_{primes}$$

Python implementation of the quadratic sieve algorithm

To implement the algorithm, we have to first import the libraries and then initialize some variables:

```
from math import sqrt, ceil

import sympy as sp

from math import gcd


# Number to be factored and upper bound of the prime numbers

N = 20791

B = 50


# We set the initial value of x such that x^2 - N > 0

x_base = ceil(sqrt(N))


# Initialization of the lists

relations = []

smooth_relations = []

factorized_y_2 = []
```

Script 3: Snippets 1 and 2

Among the variables we initialize there are also three lists. The list “relations” contains all the relations we will compute and the list “smooth_relations” contains only the smooth relations we will use to build the matrix. The list “factorize_y_2” contains a list of factors for every y^2 in the smooth relations.

In order to break down the y^2 in prime factors we need to compute all the prime numbers up to the smoothness bound B.

```
# Computation of the prime numbers up to B
primes = [2]
for i in range(3, B, 2):
   is_prime = True
   for j in primes:
      if(i%j == 0):
         is_prime = False
         break
   if(is_prime):
      primes.append(i)
print("The prime numbers up to B are: " + str(primes))
# It is not necessary to compute the prime numbers every time,
# they can be hard coded and we can decide to use them if they
# are smaller than B.
# Here are the prime numbers smaller than 100:
# primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Script 3: Snippet 3

An easy way to compute the prime numbers is to initialize the list of primes so that it contains the number 2 and then use a for loop and check if the index can be divided perfectly by an element in the list of primes. If so, the number is not a prime number. If a number cannot be divided by a prime in the list, it is a prime number and it is added to the list of primes.

Computing the list of primes every time the algorithm is executed makes sense if we change the smoothness bound. If we always use the same B we could also hardcode the list of primes up to B to save time.

Now that we have performed the preliminary operations we can proceed with the algorithm and compute the relations.

```
# Let us compute the relations: y^2 = x^2 (mod N)


# It is a good practice to set the number of relations to at least 6 times
# the upper bound of the prime numbers
n_relations = 6*len(primes)


for x in range(x_base, x_base + n_relations):
    x_2 = x**2
    y_2 = (x_2)%N
    relations.append([x_2, y_2])
```

Script 3: Snippet 4

Now that we have a list of relations, we have to sieve out the non-smooth relations and store the smooth ones in the “smooth_relations” list:

```
# Let us factorize the y^2 values using the list of primes
for i in range(n_relations):
    y_2_element = relations[i][1]
    y_2_factors = []
    j = 0
    while(j < len(primes) and y_2_element > 1):
        # Some number could have prime factors which are not included in the ones listed so
        # we have to make sure that the index of the element is plausible
        while(y_2_element % primes[j] == 0):
            y_2_element = y_2_element / primes[j]
            y_2_factors.append(primes[j])
        j += 1
    # At this point it is possible to have numbers which are not completely factored because
    # all the prime numbers they are made of are not included in the primes list.
    # In this case the y_2_element is greater than one.
    # We can use this property to keep only the y_2 which are completely factorized
    if(y_2_element == 1):
        factorized_y_2.append(y_2_factors)
        smooth_relations.append(relations[i])
```

Script 3: Snippet 5

The way we find the smooth relations is simple. We take a y^2 value and for every prime number we check if it can divide y^2 perfectly. If so, we perform the division and append the prime factor to the list of factors for that y^2. Then we try to divide the value for the same prime number because it is possible for a y^2 to contain a factor more than once. Once we cannot divide y^2 for that prime anymore we reiterate the algorithm with the next prime number.

Now that we have broken down the y^2 value we have to check if the result of the progressive division is 1 to check whether we could factor it completely with our factor base. If so, the relation is smooth and we can keep it in the designated list. Along with the smooth relation we store the list of prime factors of y^2.

Now that we have our list of smooth relations and factors, we can form the matrix M.

```
# At this point we have to create a matrix with one row for every smooth relation
# and a column for every prime number
matrix = sp.zeros(len(smooth_relations), len(primes))

# We have to fill the matrix with the number of each prime factor y_2 is made of
for i in range(len(smooth_relations)):
    for j in range(len(primes)):
        matrix[i, j] = factorized_y_2[i].count(primes[j])

matrix
```

Script 3: Snippet 6

Considering that the matrix will be mainly filled with zeros, we initialize a matrix of zeros of size $n_{smooth} \times n_{primes}$ and we change some of its values using two nested for loops. The exponent of every prime number is the number of times it appears in the list.

We can now turn the matrix into a parity matrix:

```
# To make the computation easier we can transform the elements of the matrix
# in the parities of each exponent
for i in range(len(smooth_relations)):
    for j in range(len(primes)):
        matrix[i, j] = matrix[i, j]%2

matrix
```

Script 3: Snippet 7

We can now compute the left null space of the matrix to find one or more linear combinations of rows such that their result is a row vector of zeros:

```
# We have to calculate the left null space of the matrix to find the rows
# which elements add up to an even number
NS = matrix.T.nullspace()

print("Left Null Space of the Matrix:")
sp.pprint(NS)
```

Script 3: Snippet 8

In the theoretical exposition of the algorithm, we acknowledged that the algorithm can fail because the greatest common divisor can lead to a wrong result. For this reason, it is better for us to compute all the possible $x^2_{product}$ and $y^2_{product}$ using all the possible S vectors in order to maximize the chances of finding the right solution.

```
# At this point we could pick either one among the left null spaces and use it
# to compute the product of the relations we are looking for but unfortunately this algorithm
# can fail or lead to a partial solution so we have to try all the null spaces to increase the chances
# of finding p and q.
products_array = []

for i in range(len(NS)):
    x_2_product, y_2_product = 1, 1
    for j in range(len(NS[i])):
        if(NS[i][j] != 0):
            x_2_product = x_2_product*smooth_relations[j][0]**abs(NS[i][j])
            y_2_product = y_2_product*smooth_relations[j][1]**abs(NS[i][j])
    products_array.append([x_2_product, y_2_product])
```

Script 3: Snippet 9

We initialize a list of product relations $[x^2_{product}, y^2_{product}]$ and we fill it using two nested for loops.

The outer loop iterates over the list of S vectors and the inner loop iterates over the single S vectors.

In order to avoid computing powers which will lead to a multiplication by 1 we check if the exponent of the prime number is different from 0 before proceeding with the computation.

Now we have to perform a counter-intuitive step. The exponent of the relation has to be the absolute value of the element of the S vector or else the product will be 0. This makes sense if we think about what the kernel does, but it is not the result we are looking for. If we use the absolute value of the element as an

exponent, we get to a perfect square which is another correct result of the product $S \times M$ because it leads to a row vector of zeros.

Now that we have an array of product relations, we have to iterate over it to find p and q:

```
solution_p, solution_q = 0,0


for i in range(len(products_array)):
    p = gcd(int(sqrt(products_array[i][0])) - int(sqrt(products_array[i][1])), N)
    q = gcd(int(sqrt(products_array[i][0])) + int(sqrt(products_array[i][1])), N)
    #I have to distinguish the cases in which p and q are the solution, a partial solution or not a solution
    if(p != 1 and p != N and q != 1 and q != N and p != q):
        # In this case we have the complete solution
        solution_p = p
        solution_q = q
        break
    elif((p != 1 and p != N and (q == 1 or q == N)) or (q != 1 and q != N and (p == 1 or p == N))):
        # In this case we have a partial solution
        if(p != 1 and p != N):
            solution_p = p
            solution_q = N/p
        else:
            solution_q = q
            solution_p = N/q
        break
```

Script 3: Snippet 10

We initialize two variables for the solutions and we calculate p and q for each product relation. During the iterations we check if the calculated values make a complete solution or a partial solution. As soon as we find a solution, we stop the iterations because there is no need to compute other values.

If at the end of all the iterations, we can't find a solution the solution variables will still be equal to zero.

```
# Now we have to show the outcome of the algorithm

if(solution_p != 0 and solution_q != 0):

    print("The two factors of %i are %i and %i" % (N, solution_p, solution_q))

else:

    print("The algorithm could not find a solution. Try to change the upper bound B and run the algorithm again")
```

Script 3: Snippet 11

Now it is finally time to show the results of the algorithm. If we have found a solution, we can show the two factors p and q or else we display a message telling the user to run the algorithm again increasing the smoothness bound.

Optimization of the quadratic sieve algorithm

We exposed the algorithm in a way that makes every step simple to understand. But it is not the optimal way to do it because we can perform some steps faster. There are many ways to optimize the algorithm depending on the language it is implemented in and the machine that runs it but there are two optimizations which are simpler than the others:

- Forming the parity matrix
- Computing the list of prime numbers

The simplest step we can do to optimize the algorithm is forming the parity matrix in one step.

```
# At this point we have to create a parity matrix with one row for every smooth relation

# and a column for every prime number

matrix = sp.zeros(len(smooth_relations), len(primes))


# We have to fill the matrix with the number of each prime factor y_2 is made of

for i in range(len(smooth_relations)):

    for j in range(len(primes)):

        matrix[i, j] = factorized_y_2[i].count(primes[j]) % 2


matrix
```

Script 4: Snippet 6

We simply added a modulo two operator to the code we used in the non-optimized version of the algorithm to form the exponent matrix.

The other step is to use an efficient algorithm to compute the list of prime numbers.

In the basic version of the algorithm, we computed the list of prime numbers by using an index of a for loop and checking whether we could divide it perfectly with a prime number we detected in a previous iteration. If not, we added the index to the list of primes.

We can do it faster using the sieve of Eratosthenes algorithm:

```
# We compute the prime numbers up to B using Sieve of Eratosthenes algorithm

numbers = [*range(2, B+1)]

# We reduce the upper bound by 1 because the index of the numbers list starts from 0

for i in range(ceil(sqrt(B)) - 1):

  if(numbers[i] != 0):

    for j in range(i+1, len(numbers)):

      if(numbers[j] != 0 and numbers[j] % numbers[i] == 0):

        numbers[j] = 0


# Now we have to create a list containing only the prime numbers

primes = []

for number in numbers:

  if(number != 0):

    primes.append(number)
```

Script 4: Snippet 3

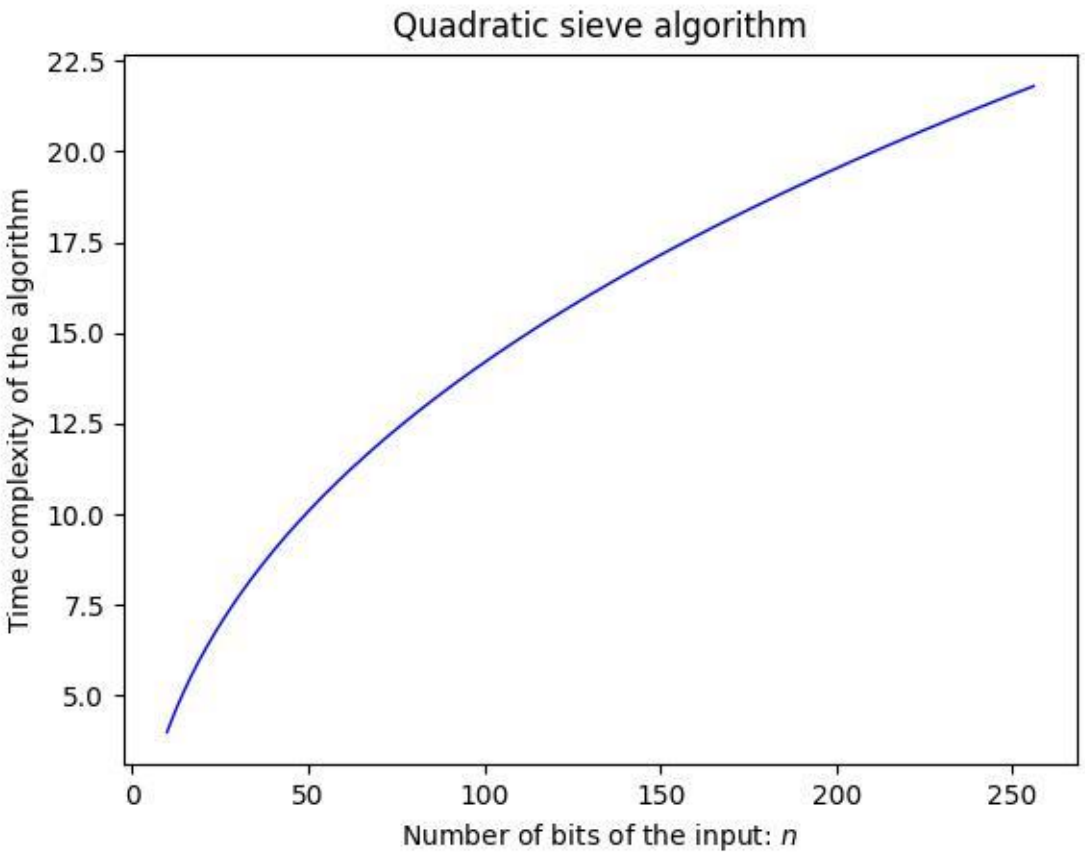
We first create an array of numbers from 2 to B. Then we start from the beginning of the array and take one number to try to use it as a divisor for the numbers after it. If the divisor can divide perfectly one of the numbers after it, we set the dividend to 0. At the end of the algorithm the only numbers greater than 0 in the array are the prime numbers we will use to make the list of primes.

Time complexity of the quadratic sieve algorithm

The theoretical time complexity of the algorithm is:

$$O\left(e^{\sqrt{\log(n)\cdot\log(\log(n))}}\right)$$

Unfortunately, deriving this value from the implementation of the algorithm is difficult and even if we tried to do it, the value we would calculate would be influenced by our particular implementation of it. As we discussed in the previous section, there are different ways to optimize this algorithm.



Plot 3: Time complexity of the quadratic sieve algorithm

From the plot we can see that this algorithm is much faster than the previous two for factoring a number of 256 bits.

Shor's algorithm

Shor's algorithm turns a prime factoring problem into a period finding problem. Starting from the product N we want to find a number a which is smaller than N and coprime with it. This is to make sure that they don't have a common divisor.

$$\gcd(a, N) = 1$$

If this condition is not met it means that a is one of the prime numbers we are looking for and we can find the other one simply by calculating $\frac{N}{a}$. Since this is a rare case, we will consider a and N to be coprime henceforth in the exposition.

Let us define the function:

$$f(x) = a^x mod(N) = y$$

The first step in the algorithm is to find the smallest $x > 0$ such that $y = 1$. Once we have found it, we have found the period of the function and we rename it with r. To understand why it is called period look at the following example.

Let us consider $a = 83$ and $N = 91$:

$$f(1) = 83^1 mod(91) = 83$$

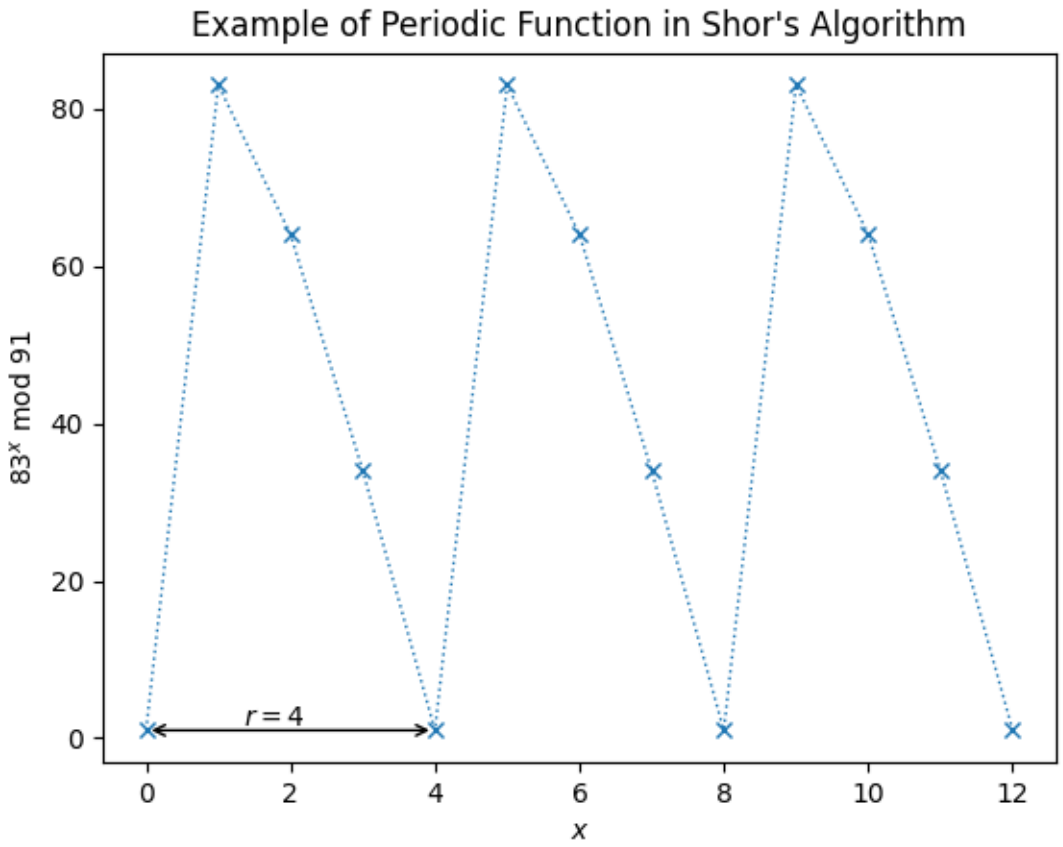
$$f(2) = 83^2 mod(91) = 64$$

$$f(3) = 83^3 mod(91) = 34$$

$$f(4) = 83^4 mod(91) = 1$$

We have found the smallest $x > 0$ such that $y = 1$. Let us see what happens if we keep increasing the exponent:

$$\begin{array}{cccc} f(1) = 83 & f(2) = 64 & f(3) = 34 & f(4) = 1 \\ f(5) = 83 & f(6) = 64 & f(7) = 34 & f(8) = 1 \\ f(9) = 83 & f(10) = 64 & f(11) = 34 & f(12) = 1 \\ f(13) = 83 & f(14) = 64 & f(15) = 34 & f(16) = 1 \end{array}$$



Plot 4: Periodicity of the modular exponentiation

Now it is clearer that the function is periodical with period $r = 4$.

Now that we have the period of the function, we can use modular arithmetic to say:

$$a^r mod(N) = 1 \Leftrightarrow a^r = m \cdot N + 1, \qquad m \in \mathbb{N}$$

$$a^r = m \cdot N + 1 \Leftrightarrow a^r - 1 = m \cdot N$$

At this point we can proceed only if r is an even number, if not we have to try again using a different a.

Assuming that r is an even number we can say that $\frac{r}{2}$ is an integer number. If we consider that we can multiply and divide r by 2 and we can use the properties of the powers, we can say that:

$$a^r - 1 = a^{\frac{r}{2} \cdot 2} - 1 = \left(a^{\frac{r}{2}}\right)^2 - 1$$

This is the difference of two squares. We can use the formula:

$$a^2 - b^2 = (a - b) \cdot (a + b)$$

And get:

$$a^r - 1 = \left(a^{\frac{r}{2}} - 1\right) \cdot \left(a^{\frac{r}{2}} + 1\right) = m \cdot N$$

$\left(a^{\frac{r}{2}} - 1\right)$ contains one factor of N and $\left(a^{\frac{r}{2}} + 1\right)$ contains the other, so if we calculate the greatest common divisor between these two numbers and N we can find the two prime numbers p and q.

To make this point clearer we can say that:

$$\left(a^{\frac{r}{2}}-1\right)=\mu_p\cdot p,\qquad \mu_p\in\mathbb{N}$$

$$\left(a^{\frac{r}{2}}+1\right)=\mu_q\cdot q,\qquad \mu_q\in\mathbb{N}$$

The two coefficients μ_p and μ_q are the product of other prime numbers.

If we break down in prime factors $\left(a^{\frac{r}{2}}-1\right)$, $\left(a^{\frac{r}{2}}+1\right)$ and N we can clearly see that p is the only common factor between $\left(a^{\frac{r}{2}}-1\right)$ and N and q is the only common factor between $\left(a^{\frac{r}{2}}+1\right)$ and N.

N	p
q	q
1	

$\left(a^{\frac{r}{2}}-1\right)$	p
μ_p	μ_p
1	

$\left(a^{\frac{r}{2}}+1\right)$	q
μ_q	μ_q
1	

This algorithm can fail in case one of the μ contains the other factor. In such case the greatest common divisor would be N.

Let us see an example of a favorable case:

$$N=91,\qquad a=83,\qquad r=4$$

$$a^{\frac{r}{2}}-1=6888$$

$$a^{\frac{r}{2}}+1=6890$$

91	13
7	7
1	

6888	2
3444	2
1772	2
861	3
287	7
41	41
1	

6890	2
3445	5
689	13
53	53
1	

In actual practice we can't use this method to find the greatest common divisor because we do not know the prime factors that constitute the numbers and because they are daunting to guess which is the whole point of the security algorithm. The method we actually use is the Euclidian algorithm.

Let us see the same example with the Euclidian algorithm:

$$\gcd(6888,91)$$

Quotient	A	B	Remainder
75	6888	91	63
1	91	63	28
2	63	28	7
4	28	7	0

$\gcd(6890,91)$

Quotient	A	B	Remainder
75	6890	91	65
1	91	65	26
2	65	26	13
2	26	13	0

At the end of Shor's algorithm, we can have three possible outcomes:

1) We find both the non-trivial prime numbers:
   This is the best-case scenario because the algorithm provides us both the prime numbers.

2) We find one non-trivial prime number and 1:
   In this case we have a partial solution, but it is sufficient to find the two numbers simply because if we have p we can calculate q by doing N/p.
   Let us see an example of this scenario:
   $$N = 15, \quad a = 13, \quad r = 2$$
   $$a^{\frac{r}{2}} - 1 = 12, \quad \gcd(15,12) = 3$$
   $$a^{\frac{r}{2}} + 1 = 14, \quad \gcd(15,14) = 1$$

3) We find 1 and N:
   This is the worst-case scenario because the algorithm does not provide any valuable result and we will have to run it again and change the a.
   Let us see an example of this scenario:
   $$N = 15, \quad a = 2, \quad r = 8$$
   $$a^{\frac{r}{2}} - 1 = 15, \quad \gcd(15,15) = 15$$
   $$a^{\frac{r}{2}} + 1 = 17, \quad \gcd(15,17) = 1$$

Implementation of Shor's algorithm in a quantum computer

In order to understand how Shor's algorithm is implemented in a quantum computer we need to understand a few preliminary concepts:

- Basics of quantum computing
- Quantum Fourier transform
- Quantum Fourier inverse transform ($QFT^{\dagger}$)
- Quantum phase estimation

Basics of quantum computing

Qubits

An ordinary computer uses bits to represent information. Similarly, a quantum computer has quantum bits, qubits for short, which are shorter ways to represent state vectors. A single qubit can have two states which are defined as follow:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}a_0\\a_1\end{bmatrix}$$

$$|1\rangle = \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}a_0\\a_1\end{bmatrix}$$

We can also have strings made by more than one qubit which are defined using the tensor product.

$$|a\rangle = \begin{bmatrix}a_0\\a_1\end{bmatrix}$$

$$|b\rangle = \begin{bmatrix}b_0\\b_1\end{bmatrix}$$

$$|a\rangle \otimes |b\rangle = |ab\rangle = \begin{bmatrix}a_0 \cdot b_0\\a_0 \cdot b_1\\a_1 \cdot b_0\\a_1 \cdot b_1\end{bmatrix}$$

Now let us calculate the four possible combinations of two qubits:

$$|0\rangle \otimes |0\rangle = |00\rangle = \begin{bmatrix}a_0 \cdot b_0\\a_0 \cdot b_1\\a_1 \cdot b_0\\a_1 \cdot b_1\end{bmatrix} = \begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix} = \begin{bmatrix}1 \cdot 1\\1 \cdot 0\\0 \cdot 1\\0 \cdot 0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

$$|0\rangle \otimes |1\rangle = |01\rangle = \begin{bmatrix}a_0 \cdot b_0\\a_0 \cdot b_1\\a_1 \cdot b_0\\a_1 \cdot b_1\end{bmatrix} = \begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix} = \begin{bmatrix}1 \cdot 0\\1 \cdot 1\\0 \cdot 0\\0 \cdot 1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

$$|1\rangle \otimes |0\rangle = |10\rangle = \begin{bmatrix}a_0 \cdot b_0\\a_0 \cdot b_1\\a_1 \cdot b_0\\a_1 \cdot b_1\end{bmatrix} = \begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix} = \begin{bmatrix}0 \cdot 1\\0 \cdot 0\\1 \cdot 1\\1 \cdot 0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

$$|1\rangle \otimes |1\rangle = |11\rangle = \begin{bmatrix}a_0 \cdot b_0\\a_0 \cdot b_1\\a_1 \cdot b_0\\a_1 \cdot b_1\end{bmatrix} = \begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix} = \begin{bmatrix}0 \cdot 0\\0 \cdot 1\\1 \cdot 0\\1 \cdot 1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

If we convert the qubit number in decimal, we have respectively 0, 1, 2 and 3. If we observe the position of the 1 inside the state vector we can notice that its index coincides with the number of the qubit. The same reasoning is valid for any string of qubits made by zeros and ones including the case of a single qubit previously discussed.

Superposition of qubits

Until now we have only discussed simple qubits strings in which the state vectors were made by a sequence of zeros and a 1 located in the position coinciding with the number of the qubit. These are not the only existing qubits. There can be more complex qubits, which are the ones that make quantum computation powerful. To turn a single qubit into a superposition of qubits we use the Hadamard operator which is defined as follows:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

Let us apply the Hadamard operator to a single qubit and calculate the result:

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\cdot 1 + 1\cdot 0\\ 1\cdot 1 - 1\cdot 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\cdot 0 + 1\cdot 1\\ 1\cdot 0 - 1\cdot 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ -1\end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

The qubits $H|0\rangle$ and $H|1\rangle$ are also known as respectively $|+\rangle$ and $|-\rangle$ because of the sign of the qubit $|1\rangle$.

Measuring a qubit

Measuring a qubit is not a deterministic process, it depends on probability. This statement implies that the sum of the probabilities of measuring the different qubit states must be 1.

In general, if we consider a qubit as a superposition of qubits, we have that:

$$|a\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad \alpha, \beta \in \mathbb{C}$$

The probabilities of measuring $|0\rangle$ or $|1\rangle$ are:

$$P(|0\rangle) = |\alpha|^2$$

$$P(|1\rangle) = |\beta|^2$$

This implies that the coefficients α and β are such that:

$$|\alpha|^2 + |\beta|^2 = 1$$

This introduces the problem of normalizing the coefficients of the qubits. Normalization is the reason why the Hadamard operator has a coefficient of $\frac{1}{\sqrt{2}}$. In fact, when we measure a qubit in the $|+\rangle$ state we have:

$$\alpha = \frac{1}{\sqrt{2}}$$

$$\beta = \frac{1}{\sqrt{2}}$$

$$P(|0\rangle) = |\alpha|^2 = \frac{1}{2}$$

$$P(|1\rangle) = |\beta|^2 = \frac{1}{2}$$

$$P(|0\rangle) + P(|1\rangle) = 1$$

Likewise, when we measure a qubit in the $|-\rangle$ state we have:

$$\alpha = \frac{1}{\sqrt{2}}$$

$$\beta = -\frac{1}{\sqrt{2}}$$

$$P(|0\rangle) = |\alpha|^2 = \frac{1}{2}$$

$$P(|1\rangle) = |\beta|^2 = \frac{1}{2}$$

$$P(|0\rangle) + P(|1\rangle) = 1$$

Quantum Fourier Transform

The quantum Fourier transform (QFT) transforms between two bases, the computational (Z) basis, and the Fourier basis. It is defined as follows:

$$QFT_N|x\rangle = \frac{1}{\sqrt{N}} \cdot \sum_{y=0}^{N-1} e^{\frac{2\pi \cdot i \cdot x \cdot y}{2^n}} \cdot |y\rangle$$

$$n = number\ of\ bits, \qquad N = 2^n$$

Let us see an example of QFT of one simple qubit:

$$n = 1, \qquad N = 2$$

$$QFT_2|0\rangle = \frac{1}{\sqrt{2}} \cdot \sum_{y=0}^{1} e^{\pi \cdot i \cdot 0 \cdot y} \cdot |y\rangle = \frac{1}{\sqrt{2}} \cdot \sum_{y=0}^{1} |y\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle + |1\rangle) = |+\rangle$$

$$QFT_2|1\rangle = \frac{1}{\sqrt{2}} \cdot \sum_{y=0}^{1} e^{\pi \cdot i \cdot 1 \cdot y} \cdot |y\rangle = \frac{1}{\sqrt{2}} \cdot \sum_{y=0}^{1} e^{\pi \cdot i \cdot y} \cdot |y\rangle = \frac{1}{\sqrt{2}} \cdot (e^{\pi \cdot i \cdot 0} \cdot |0\rangle + e^{\pi \cdot i \cdot 1}|1\rangle) =$$

$$= \frac{1}{\sqrt{2}} \cdot (|0\rangle - |1\rangle) = |-\rangle$$

From this example we can see that using the QFT on a single simple qubit is equivalent to applying the Hadamard operator to it.

Generalized QFT for one qubit

We can generalize this operation on more complex qubits by considering them a superposition of the qubits $|0\rangle$ and $|1\rangle$. Let us consider a generical qubit $|\psi\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad x_0 = \alpha, \qquad x_1 = \beta, \qquad N = 2$$

The QFT of this qubit will be a superposition of the qubits $|0\rangle$ and $|1\rangle$ just like the starting qubit but it will have different coefficients for them. So, the concept behind the generalized QFT of one qubit, and its inverse operation, is considering every qubit as a superposition of the qubits $|0\rangle$ and $|1\rangle$ and use the coefficients of the starting qubit to calculate the coefficients of the target qubit using the following formula:

$$y_k = \frac{1}{\sqrt{N}} \cdot \sum_{j=0}^{N-1} \left(x_j \cdot e^{\frac{2\pi \cdot i \cdot k \cdot j}{N}} \right)$$

Let us apply this formula to the generical qubit $|\psi\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad x_0 = \alpha, \qquad x_1 = \beta, \qquad N = 2$$

$$y_0 = \frac{1}{\sqrt{2}} \cdot \left(x_0 \cdot e^{\frac{2\pi \cdot i \cdot 0 \cdot 0}{2}} + x_1 \cdot e^{\frac{2\pi \cdot i \cdot 0 \cdot 1}{2}} \right) = \frac{1}{\sqrt{2}} \cdot (\alpha + \beta)$$

$$y_1 = \frac{1}{\sqrt{2}} \cdot \left(x_0 \cdot e^{\frac{2\pi \cdot i \cdot 1 \cdot 0}{2}} + x_1 \cdot e^{\frac{2\pi \cdot i \cdot 1 \cdot 1}{2}} \right) = \frac{1}{\sqrt{2}} \cdot (\alpha - \beta)$$

We calculated that in general:

$$QFT_2|\psi\rangle = \frac{1}{\sqrt{2}} \cdot (\alpha + \beta)|0\rangle + \frac{1}{\sqrt{2}} \cdot (\alpha - \beta)|1\rangle$$

In order to use the general formula for the cases $|\psi\rangle = |0\rangle$ and $|\psi\rangle = |1\rangle$ we just need to consider that:

$$|\psi\rangle = |0\rangle, \qquad \alpha = 1, \qquad \beta = 0$$

$$|\psi\rangle = |1\rangle, \qquad \alpha = 0, \qquad \beta = 1$$

Generalized QFT formula for strings of qubits

Let us start from the canonical QFT formula:

$$QFT_N|x\rangle = \frac{1}{\sqrt{N}} \cdot \sum_{y=0}^{N-1} e^{\frac{2\pi \cdot i \cdot x \cdot y}{2^n}} \cdot |y\rangle$$

y is made by more than one qubit so:

$$y = y_1 y_2 y_3 \dots y_n$$

Now we need to remember a general property of numbers. For instance, let us consider the number 7:

$$(7)_{10} = (111)_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

So, in general we can say that a string of qubits y can be seen as follows:

$$y = y_1 \cdot 2^{n-1} + y_2 \cdot 2^{n-2} + y_3 \cdot 2^{n-3} + \cdots + y_n \cdot 2^0$$

In the argument of the exponential y is divided by 2^n so we have:

$$\frac{y}{2^n} = \frac{y_1 \cdot 2^{n-1} + y_2 \cdot 2^{n-2} + y_3 \cdot 2^{n-3} + \cdots + y_n \cdot 2^0}{2^n} = \frac{y_1}{2^1} + \frac{y_2}{2^2} + \frac{y_3}{2^3} + \cdots + \frac{y_n}{2^n} = \sum_{k=1}^{n} \frac{y_k}{2^k}$$

We can rewrite the formula as:

$$QFT_N|x\rangle = \frac{1}{\sqrt{N}} \cdot \sum_{y=0}^{N-1} e^{2\pi \cdot i \cdot x \cdot \left(\sum_{k=1}^{n} \frac{y_k}{2^k} \right)} \cdot |y_1 y_2 y_3 \dots y_n\rangle$$

We can use the properties of the powers and expand the exponential of a sum into a product of exponentials:

$$QFT_N|x\rangle = \frac{1}{\sqrt{N}} \cdot \sum_{y=0}^{N-1} e^{2\pi \cdot i \cdot x \cdot \frac{y_1}{2^1}} \cdot e^{2\pi \cdot i \cdot x \cdot \frac{y_2}{2^2}} \cdot e^{2\pi \cdot i \cdot x \cdot \frac{y_3}{2^3}} \cdot \ldots \cdot e^{2\pi \cdot i \cdot x \cdot \frac{y_n}{2^n}} |y_1 y_2 y_3 \ldots y_n\rangle$$
$$= \frac{1}{\sqrt{N}} \cdot \sum_{y=0}^{N-1} \prod_{k=1}^{n} e^{2\pi \cdot i \cdot x \cdot \frac{y_k}{2^k}} \cdot |y_1 y_2 y_3 \ldots y_n\rangle$$

Now we need to consider that the sum referred to a string of qubits can be broken down into a sequence of sums referring to each single qubits.

$$\sum_{y=0}^{N-1} |y_1 y_2 y_3 \ldots y_n\rangle = \sum_{y_1}^{1} \sum_{y_2}^{1} \sum_{y_3}^{1} \ldots \sum_{y_n}^{1} |y_n\rangle$$

At this point we can break down y in the single bits it is made of and concatenate them using the tensor product:

$$QFT_N|x\rangle = \frac{1}{\sqrt{N}} \cdot \left(\sum_{y_1=0}^{1} e^{2\pi \cdot i \cdot x \cdot \frac{y_1}{2^1}} |y_1\rangle \right) \otimes \left(\sum_{y_2=0}^{1} e^{2\pi \cdot i \cdot x \cdot \frac{y_2}{2^2}} |y_2\rangle \right) \otimes \left(\sum_{y_3=0}^{1} e^{2\pi \cdot i \cdot x \cdot \frac{y_3}{2^3}} |y_3\rangle \right) \otimes$$
$$\cdots \otimes \left(\sum_{y_n=0}^{1} e^{2\pi \cdot i \cdot x \cdot \frac{y_n}{2^n}} |y_n\rangle \right) =$$
$$= \frac{1}{\sqrt{N}} \cdot \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2}} |1\rangle \right) \otimes \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2^2}} |1\rangle \right) \otimes \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2^3}} |1\rangle \right) \otimes \cdots \otimes \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2^n}} |1\rangle \right)$$

Every qubit become a superposition of the qubit $|0\rangle$ and the qubit $|1\rangle$ multiplied by a phase. This formula leads us to think about a generalization of the Hadamard gate for a single qubit.

$$H|x_k\rangle = \frac{(|0\rangle + e^{\frac{2\pi \cdot i \cdot x_k}{2^k}} |1\rangle)}{\sqrt{2}}$$

The denominator of each single qubit is $\sqrt{2}$ so it may seem that using the generalized Hadamard gate is not equivalent to using the generalized QFT formula. The two methods are equivalent because a string is made by n qubits and $\sqrt{2}$ is a scalar, so during the tensor product we can multiply all the denominators at once and get:

$$\left(\frac{1}{\sqrt{2}}\right)^n = \frac{1}{\sqrt{N}}$$

QFT using a matrix

The QFT can be also seen as a matrix to be applied to a state vector. This is the most compact way to define the QFT and it can be implemented in a quantum circuit using a quantum gate. The QFT matrix is defined as follows:

$$F_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2\cdot(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2\cdot(N-1)} & \omega^{3\cdot(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}, \qquad \omega = e^{\frac{2\pi \cdot i}{N}}$$

In order to convince ourselves that the matrix and the general formula are equivalent let us see an example of QTF using 2 qubits.

$$n = 2, \qquad N = 2^n = 4$$

Let us first use the general QFT formula:

$$QFT|x\rangle = \frac{1}{\sqrt{4}} \cdot \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2}}|1\rangle\right) \otimes \left(|0\rangle + e^{\frac{2\pi \cdot i \cdot x}{2^2}}|1\rangle\right) = \frac{1}{2} \cdot \left(|0\rangle + e^{\pi \cdot i \cdot x}|1\rangle\right) \otimes \left(|0\rangle + e^{\frac{\pi}{2} \cdot i \cdot x}|1\rangle\right)$$

$$QFT|0\rangle = \frac{1}{2} \cdot (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) = \frac{1}{2} \cdot \left(\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\1\end{bmatrix}\right) = \frac{1}{2} \cdot \begin{bmatrix}1\\1\\1\\1\end{bmatrix} = |\tilde{0}\rangle$$

$$QFT|1\rangle = \frac{1}{2} \cdot (|0\rangle - |1\rangle) \otimes (|0\rangle + i \cdot |1\rangle) = \frac{1}{2} \cdot \left(\begin{bmatrix}1\\-1\end{bmatrix} \otimes \begin{bmatrix}1\\i\end{bmatrix}\right) = \frac{1}{2} \cdot \begin{bmatrix}1\\i\\-1\\-i\end{bmatrix} = |\tilde{1}\rangle$$

$$QFT|2\rangle = \frac{1}{2} \cdot (|0\rangle + |1\rangle) \otimes (|0\rangle - |1\rangle) = \frac{1}{2} \cdot \left(\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\-1\end{bmatrix}\right) = \frac{1}{2} \cdot \begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} = |\tilde{2}\rangle$$

$$QFT|3\rangle = \frac{1}{2} \cdot (|0\rangle - |1\rangle) \otimes (|0\rangle - i \cdot |1\rangle) = \frac{1}{2} \cdot \left(\begin{bmatrix}1\\-1\end{bmatrix} \otimes \begin{bmatrix}1\\-i\end{bmatrix}\right) = \frac{1}{2} \cdot \begin{bmatrix}1\\-i\\-1\\+i\end{bmatrix} = |\tilde{3}\rangle$$

Now let us use the matrix:

$$\omega = e^{\frac{2\pi \cdot i}{4}} = e^{\frac{\pi}{2} \cdot i}, \qquad F_4 = \frac{1}{2} \cdot \begin{bmatrix}1 & 1 & 1 & 1\\1 & i & -1 & -i\\1 & -1 & 1 & -1\\1 & -i & -1 & i\end{bmatrix}$$

$$QFT|0\rangle = F_4|0\rangle = \frac{1}{2} \cdot \begin{bmatrix}1 & 1 & 1 & 1\\1 & i & -1 & -i\\1 & -1 & 1 & -1\\1 & -i & -1 & i\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix}1\\1\\1\\1\end{bmatrix} = |\tilde{0}\rangle$$

$$QFT|1\rangle = F_4|1\rangle = \frac{1}{2} \cdot \begin{bmatrix}1 & 1 & 1 & 1\\1 & i & -1 & -i\\1 & -1 & 1 & -1\\1 & -i & -1 & i\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix}1\\i\\-1\\-i\end{bmatrix} = |\tilde{1}\rangle$$

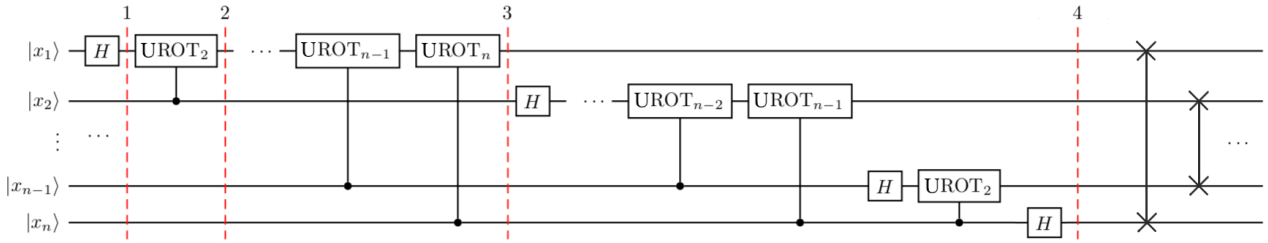
$$QFT|2\rangle = F_4|2\rangle = \frac{1}{2} \cdot \begin{bmatrix}1 & 1 & 1 & 1\\1 & i & -1 & -i\\1 & -1 & 1 & -1\\1 & -i & -1 & i\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} = |\tilde{2}\rangle$$

$$QFT|3\rangle = F_4|3\rangle = \frac{1}{2} \cdot \begin{bmatrix}1 & 1 & 1 & 1\\1 & i & -1 & -i\\1 & -1 & 1 & -1\\1 & -i & -1 & i\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix}1\\-i\\-1\\+i\end{bmatrix} = |\tilde{3}\rangle$$

The two methods lead to the same results.

Understanding the QFT matrix is crucial because it greatly simplifies the understanding of the inverse QFT function also known as $QFT^{\dagger}$ (QFT dagger).

QFT circuit



The circuit that implements QFT uses two quantum gates: the first is the Hadamard gate and the second is the controlled rotation.

The controlled rotation operation is defined as follows:

$$CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix}$$

Where the unitary rotation is:

$$UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{2\pi \cdot i}{2^k}\right) \end{bmatrix}$$

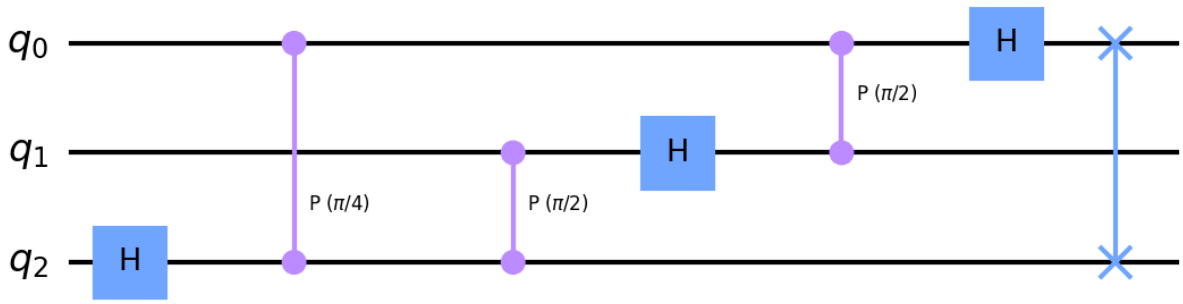
The operator is called “controlled rotation” because the first qubit is the control qubit and the second qubit rotates if the control qubit is set to $|1\rangle$. By this definition we can understand that we cannot apply the Hadamard operator to all qubits at once at the beginning of the circuit because that would cause the control qubits to be in a superposition of states instead of a computational state.

As it is shown in the picture, every qubit is first changed using an Hadamard gate and then it is rotated according to the values of the other bits up to the nth one. For instance, the first qubit is rotated according to the qubit from the second to the nth, the second qubit from the third to the nth and the nth one only passes through an Hadamard gate.

In the circuit, after the gates it is possible to change the qubits ordering because it depends on the convention used by the quantum computer user.

QFT circuit example

Let us make an example of 3-qubits QFT circuit:



At the end of the circuit the qubits become:

$$QFT|x\rangle = \frac{1}{\sqrt{2}} \cdot \left(|0\rangle + \exp\left(\frac{2\pi \cdot i}{2} \cdot x_2\right) \cdot |1\rangle\right) \otimes \frac{1}{\sqrt{2}} \cdot \left(|0\rangle + \exp\left(\frac{2\pi \cdot i}{2^2} \cdot x_2 + \frac{2\pi \cdot i}{2} \cdot x_1\right) \cdot |1\rangle\right) \otimes \frac{1}{\sqrt{2}} \cdot \left(|0\rangle + \exp\left(\frac{2\pi \cdot i}{2^3} \cdot x_2 + \frac{2\pi \cdot i}{2^2} \cdot x_1 + \frac{2\pi \cdot i}{2} \cdot x_0\right) \cdot |1\rangle\right)$$

This formula is different from the canonical one we have already discussed but it is equivalent to it. In order to convince ourselves about the equivalence we could try to calculate the QFT of a number using both formulas and comparing the results.

Inverse Quantum Fourier Transform

The quantum Fourier transform (QFT) transforms between two bases, the computational (Z) basis, and the Fourier basis. This change of basis is essential to perform some operations such as the quantum phase estimation. After we have performed these operations, we must transform the qubits back in the computational basis to be able to measure them and provide the results to a classical computer. To perform this operation, we need the inverse quantum Fourier transform.

Inverse Quantum Fourier Transform of a single qubit

In order to calculate the inverse QFT of a single qubit we need to consider it a superposition of the qubits $|0\rangle$ and $|1\rangle$ and use the coefficients of the starting qubit to calculate the coefficients of the target qubit using the following formula:

$$x_k = \frac{1}{\sqrt{N}} \cdot \sum_{j=0}^{N-1} \left(y_j \cdot e^{-\frac{2\pi \cdot i \cdot k \cdot j}{N}}\right)$$

Let us calculate the $QFT^\dagger$ of the qubits $|+\rangle$ and $|-\rangle$:

$$|+\rangle = \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle, \qquad y_0 = \frac{1}{\sqrt{2}} = y_1, \qquad N = 2$$

$$x_0 = \frac{1}{\sqrt{2}} \cdot \left(y_0 \cdot e^{-\frac{2\pi \cdot 0 \cdot 0}{2}} + y_1 \cdot e^{-\frac{2\pi \cdot 0 \cdot 1}{2}}\right) = \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\right) = 1$$

$$x_1 = \frac{1}{\sqrt{2}} \cdot \left(y_0 \cdot e^{-\frac{2\pi \cdot 1 \cdot 0}{2}} + y_1 \cdot e^{-\frac{2\pi \cdot 1 \cdot 1}{2}}\right) = \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\right) = 0$$

$$|-\rangle = \frac{1}{\sqrt{2}} \cdot |0\rangle - \frac{1}{\sqrt{2}} \cdot |1\rangle, \qquad y_0 = \frac{1}{\sqrt{2}}, \qquad y_1 = -\frac{1}{\sqrt{2}}, \qquad N = 2$$

$$x_0 = \frac{1}{\sqrt{2}} \cdot \left(y_0 \cdot e^{-\frac{2\pi \cdot 0 \cdot 0}{2}} + y_1 \cdot e^{-\frac{2\pi \cdot 0 \cdot 1}{2}}\right) = \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\right) = 0$$

$$x_1 = \frac{1}{\sqrt{2}} \cdot \left(y_0 \cdot e^{-\frac{2\pi \cdot 1 \cdot 0}{2}} + y_1 \cdot e^{-\frac{2\pi \cdot 1 \cdot 1}{2}}\right) = \frac{1}{\sqrt{2}} \cdot \left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\right) = 1$$

Inverse QTF using a matrix

As the QFT can be implemented in a quantum gate using a matrix so can the inverse QFT. It is simple to derive the $QFT^{\dagger}$ starting from the QFT matrix and vice versa. All we have to do is to transpose the matrix and calculate its complex conjugate.

$$F_N \xrightarrow{complex\ conjugate\ transpose} F_N^{\dagger}$$

$$F_N^{\dagger} \xrightarrow{complex\ conjugate\ transpose} F_N$$

Let us see an example with two qubits in which we calculate the qubits from $|0\rangle$ to $|3\rangle$ starting from the qubits from $|\tilde{0}\rangle$ to $|\tilde{3}\rangle$:

$$F_4 = \frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & -1 & -i\\ 1 & -1 & 1 & -1\\ 1 & -i & -1 & i\end{bmatrix} \xrightarrow{complex\ conjugate\ transpose} F_4^{\dagger} = \frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & +i\\ 1 & -1 & 1 & -1\\ 1 & +i & -1 & -i\end{bmatrix}$$

$$F_4^{\dagger}|\tilde{0}\rangle = \frac{1}{2}\cdot\frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & +i\\ 1 & -1 & 1 & -1\\ 1 & +i & -1 & -i\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \frac{1}{4}\cdot\begin{bmatrix}4\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

$$F_4^{\dagger}|\tilde{1}\rangle = \frac{1}{2}\cdot\frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & +i\\ 1 & -1 & 1 & -1\\ 1 & +i & -1 & -i\end{bmatrix}\begin{bmatrix}1\\i\\-1\\-i\end{bmatrix} = \frac{1}{4}\cdot\begin{bmatrix}0\\4\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

$$F_4^{\dagger}|\tilde{2}\rangle = \frac{1}{2}\cdot\frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & +i\\ 1 & -1 & 1 & -1\\ 1 & +i & -1 & -i\end{bmatrix}\begin{bmatrix}1\\-1\\1\\-1\end{bmatrix} = \frac{1}{4}\cdot\begin{bmatrix}0\\0\\4\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

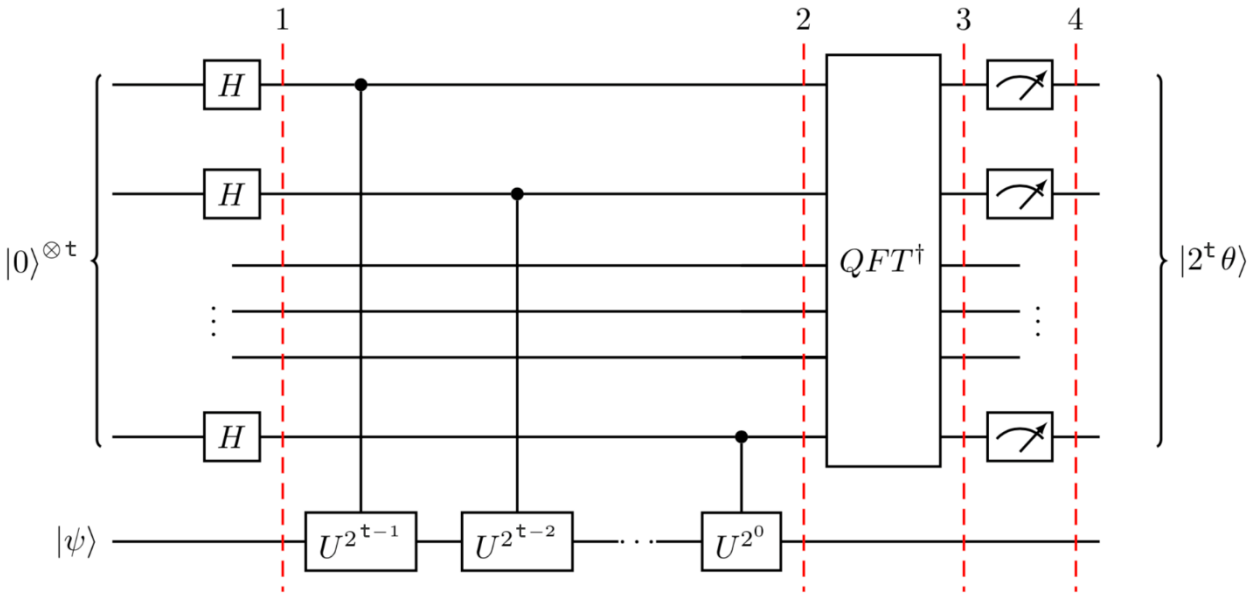
$$F_4^{\dagger}|\tilde{3}\rangle = \frac{1}{2}\cdot\frac{1}{2}\cdot\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & +i\\ 1 & -1 & 1 & -1\\ 1 & +i & -1 & -i\end{bmatrix}\begin{bmatrix}1\\i\\-1\\-i\end{bmatrix} = \frac{1}{4}\cdot\begin{bmatrix}0\\0\\0\\4\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Quantum phase estimation

Quantum phase estimation is one of the most important subroutines in quantum computation. Given an eigenvector $|\psi\rangle$ with a phase θ_ψ to be measured and a set of counting qubits, the process of quantum phase estimation can encode θ_ψ in the counting qubits and measure it.

The intuition on which this process is based is that we can turn the state of the counting qubits from the computational one to the Fourier one to apply a phase to them according to $|\psi\rangle$. At the end of this step the phase θ_ψ is encoded in the counting qubits which have the value $|2^t \cdot \theta_\psi\rangle$ where t is the number of counting qubits. In order to measure this value, we have to convert the qubits back to the computational state using the inverse QFT function. The measurement will not always give us the exact value of θ_ψ, it will probably give us an estimation of it. This implies that we can have problems with the accuracy of the estimated value which we can increase by increasing the number of counting qubits.

Let us analyze the circuit which implements the quantum phase estimation:



0) We start from a set of t counting qubits and the eigenvector $|\psi\rangle$. We will refer to the set of all qubits as ψ with a subscript that indicates the step inside the circuit.

$$|\psi_0\rangle = |0\rangle^{\otimes t} \otimes |\psi\rangle$$

1) We apply the Hadamard gate to the counting qubits to change their basis from the computational one to the Fourier one.

$$|\psi_1\rangle = \frac{1}{2^{\frac{n}{2}}}(|0\rangle + |1\rangle)^{\otimes t} \otimes |\psi\rangle$$

2) In order to understand the second step, we need to introduce the U operator. U is a unitary operator defined as follows:

$$U|\psi\rangle = e^{2\pi \cdot i \cdot \theta_\psi}|\psi\rangle$$

So U extracts the phase from $|\psi\rangle$.
At this point we can define a controlled version of the U operator called CU which rotates the counting qubit according to the phase of $|\psi\rangle$ only if the control qubit in the eigenvector $|\psi\rangle$ is set to $|1\rangle$.
Now we need to consider the following relationship:

$$U^{2j}|\psi\rangle = (e^{2\pi \cdot i \cdot \theta_\psi})^{2j}|\psi\rangle = e^{2\pi \cdot i \cdot 2j \cdot \theta_\psi}|\psi\rangle$$

If we apply the t controlled operations CU^{2^j} with $0 \leq j \leq t-1$ and we use the following relationship:

$$|0\rangle \otimes |\psi\rangle + |1\rangle \otimes e^{2\pi \cdot i \cdot \theta_\psi}|\psi\rangle = \left(|0\rangle + e^{2\pi \cdot i \cdot \theta_\psi}|1\rangle\right) \otimes |\psi\rangle$$

The qubits become:

$$|\psi_2\rangle = \frac{1}{2^{\frac{n}{2}}}(|0\rangle + e^{2\pi \cdot i \cdot \theta_\psi \cdot 2^{n-1}}|1\rangle) \otimes \cdots \otimes (|0\rangle + e^{2\pi \cdot i \cdot \theta_\psi \cdot 2^1}|1\rangle) \otimes (|0\rangle + e^{2\pi \cdot i \cdot \theta_\psi \cdot 2^0}|1\rangle) \otimes |\psi\rangle$$

3) At this point the control qubits have become a sequence of superpositions of the qubit $|0\rangle$ and the qubit $|1\rangle$ with a phase applied to it. This reminds us of the quantum Fourier transform of a sequence of qubits and this means that if we apply the inverse QFT function to $|\psi_2\rangle$ we will get a sequence of qubits in the computational state.

$$|\psi_3\rangle = QFT^\dagger |\psi_2\rangle$$

4) Now that the counting qubits are back into computational form they can be measured and their value is $2^t \cdot \theta_\psi$:

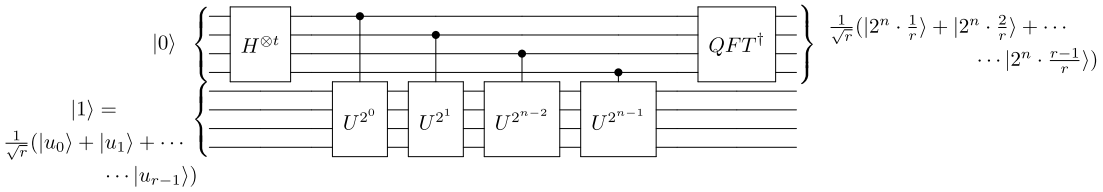
$$|\psi_4\rangle = |2^t \cdot \theta_\psi\rangle \otimes |\psi\rangle$$

So the phase θ_ψ we were looking for can be easily calculated by dividing the counting qubits by 2^t.

Quantum circuit that implements Shor's algorithm

Shor's algorithm is not fully implemented in a quantum computer. The quantum computer only solves the period finding problem and then it communicates this value to a classical computer which does the rest of the mathematical operations.

The way the quantum computer finds the period is by using an adapted version of the quantum phase estimation circuit using $|\psi\rangle = |1\rangle$ for reasons we will discuss in the exposition.



Let us first define the U function for this circuit as:

$$U|y\rangle \equiv |a \cdot y \ (mod\ N)\rangle$$

Let us consider $a = 83$, $N = 91$ and $r = 4$, let us start from the state $|1\rangle$ and apply the U operator multiple times:

$$U^1|1\rangle = |83\rangle$$

$$U^2|1\rangle = |64\rangle$$

$$U^3|1\rangle = |34\rangle$$

$$U^4|1\rangle = U^r|1\rangle = |1\rangle$$

If we apply the U operator r times to the same state we get the same state we started from.

Now, if we consider a superposition of the eigenstates of U we can prove that it is an eigenstate of U.

$$|u_0\rangle = \frac{1}{\sqrt{r}} \cdot \sum_{k=0}^{r-1} |a^k (mod\ N)\rangle$$

Let us see an example:

$$a = 83, \qquad N = 91, \qquad r = 4$$

$$|u_0\rangle = \frac{1}{2} \cdot (|1\rangle + |83\rangle + |64\rangle + |34\rangle)$$

$$U|u_0\rangle = \frac{1}{2} \cdot (U|1\rangle + U|83\rangle + U|64\rangle + U|34\rangle) = \frac{1}{2} \cdot (|83\rangle + |64\rangle + |34\rangle + |1\rangle) = |u_0\rangle$$

Starting from $|u_0\rangle$ we can add a phase to the eigenstates and make it proportional to the index of the sum:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \cdot \sum_{k=0}^{r-1} e^{-\frac{2\pi \cdot i \cdot k}{r}} |a^k (mod\ N)\rangle$$

If we apply the U operator to $|u_1\rangle$ we get:

$$|u_1\rangle = \frac{1}{2} \cdot (|1\rangle + e^{-\frac{\pi \cdot i \cdot 1}{2}} |83\rangle + e^{-\frac{\pi \cdot i \cdot 2}{2}} |64\rangle + e^{-\frac{\pi \cdot i \cdot 3}{2}} |34\rangle)$$

$$U|u_1\rangle = \frac{1}{2} \cdot (|83\rangle + e^{-\frac{\pi \cdot i \cdot 1}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot 2}{2}}|34\rangle + e^{-\frac{\pi \cdot i \cdot 3}{2}}|1\rangle)$$

At this point we can factor out $e^{\frac{\pi}{2} \cdot i}$ and get:

$$U|u_1\rangle = \frac{1}{2} \cdot e^{\frac{\pi}{2} i} \cdot (e^{-\frac{\pi \cdot i \cdot 1}{2}}|83\rangle + e^{-\frac{\pi \cdot i \cdot 2}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot 3}{2}}|34\rangle + \underbrace{e^{-\frac{\pi \cdot i \cdot 4}{2}}}_{1}|1\rangle)$$

$$= \frac{1}{2} \cdot e^{\frac{\pi}{2} \cdot i} \cdot (|1\rangle + e^{-\frac{\pi \cdot i \cdot 1}{2}}|83\rangle + e^{-\frac{\pi \cdot i \cdot 2}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot 3}{2}}|34\rangle)$$

$$U|u_1\rangle = e^{\frac{\pi}{2} \cdot i} \cdot |u_1\rangle$$

In general, we can say that:

$$U|u_1\rangle = e^{\frac{2\pi \cdot i}{r}}|u_1\rangle$$

The usage of r in the eigenvalue makes sure that the phase differences between the r computational basis states are equal.

We can generalize the eigenstates further by adding another index s in the phase of the eigenvalue:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \cdot \sum_{k=0}^{r-1} e^{-\frac{2\pi \cdot i \cdot s \cdot k}{r}}|a^k (mod\ N)\rangle$$

If we apply the U operator to $|u_s\rangle$ we get:

$$|u_s\rangle = \frac{1}{2} \cdot (|1\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 1}{2}}|83\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 2}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 3}{2}}|34\rangle)$$

$$U|u_s\rangle = \frac{1}{2} \cdot (|83\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 1}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 2}{2}}|34\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 3}{2}}|1\rangle)$$

At this point we can factor out $e^{\frac{\pi}{2} \cdot i \cdot s}$ and get:

$$U|u_s\rangle = \frac{1}{2} \cdot e^{\frac{\pi}{2} \cdot i \cdot s} \cdot (e^{-\frac{\pi \cdot i \cdot s \cdot 1}{2}}|83\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 2}{2}}|64\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 3}{2}}|34\rangle + e^{-\frac{\pi \cdot i \cdot s \cdot 4}{2}}|1\rangle)$$

$$U|u_s\rangle = e^{\frac{\pi}{2} \cdot i \cdot s} \cdot |u_s\rangle$$

In general, we can say that:

$$U|u_s\rangle = e^{\frac{2\pi \cdot i \cdot s}{r}}|u_s\rangle$$

We now have a unique eigenstate for each integer value s of where $0 \leq s \leq r - 1$. Very conveniently, if we sum up all these eigenstates, the different phases cancel out all computational basis states except $|1\rangle$:

$$\frac{1}{\sqrt{r}} \cdot \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle$$

Let us apply this formula to our example:

$$|u_0\rangle = \frac{1}{2} \cdot (|1\rangle + |83\rangle + |64\rangle + |34\rangle)$$

$$|u_1\rangle = \frac{1}{2} \cdot (|1\rangle + e^{-\frac{2\pi \cdot i}{4}}|83\rangle + e^{-\frac{2\pi \cdot i \cdot 2}{4}}|64\rangle + e^{-\frac{2\pi \cdot i \cdot 3}{4}}|34\rangle)$$

$$|u_2\rangle = \frac{1}{2}\cdot(|1\rangle + e^{-\frac{2\pi\cdot i\cdot 2}{4}}|83\rangle + e^{-\frac{2\pi\cdot i\cdot 2\cdot 2}{4}}|64\rangle + e^{-\frac{2\pi\cdot i\cdot 2\cdot 3}{4}}|34\rangle)$$

$$|u_3\rangle = \frac{1}{2}\cdot(|1\rangle + e^{-\frac{2\pi\cdot i\cdot 3}{4}}|83\rangle + e^{-\frac{2\pi\cdot i\cdot 3\cdot 2}{4}}|64\rangle + e^{-\frac{2\pi\cdot i\cdot 3\cdot 3}{4}}|34\rangle)$$

$$\frac{1}{\sqrt{r}}\cdot\sum_{s=0}^{r-1}|u_s\rangle = \frac{1}{2}\cdot(|u_0\rangle + |u_1\rangle + |u_2\rangle + |u_3\rangle) =$$

$$= \frac{1}{2}\cdot\frac{1}{2}\cdot(4\cdot|1\rangle + \left(1 + e^{-\frac{\pi}{2}\cdot i} + e^{-\pi\cdot i} + e^{-\frac{3}{2}\pi\cdot i}\right)\cdot|83\rangle + \left(1 + e^{-\pi\cdot i} + e^{-2\pi\cdot i} + e^{-3\cdot\pi\cdot i}\right)\cdot|64\rangle +$$
$$+ \left(1 + e^{-\frac{3}{2}\pi\cdot i} + e^{-3\pi\cdot i} + e^{-\frac{9}{2}\pi\cdot i}\right)\cdot|34\rangle) =$$
$$= \frac{1}{4}\cdot(4\cdot|1\rangle + (1 - i - 1 + i)\cdot|83\rangle + (1 - 1 + 1 - 1)\cdot|64\rangle + (1 + i - 1 - i)\cdot|34\rangle) =$$
$$= \frac{1}{4}\cdot 4\cdot|1\rangle = |1\rangle$$

This proves that the eigenstate $|1\rangle$ is a superposition of the states $|u_s\rangle$ which is the reason why we use it as the eigenvector $|\psi\rangle$. So its phase is:

$$\phi = \frac{s}{r}, \qquad 0 \le s \le r - 1$$

At the end of the quantum phase estimation, we measure $0 \le \phi < 1$ so we have to guess the fraction that gave us this result using the continued fractions algorithm. Let us see an example of this process:

$$\phi = \frac{s}{r}, \qquad r = 4, \qquad 0 \le s \le 3$$

This means that according to the value of s we can measure these possible values:

$$\phi = \{0, 0.25, 0.50, 0.75\}$$

- Case $\phi = 0$:

  $$s = 0, \qquad r = ?$$

  In this case we can only say that the numerator is zero and the denominator is different from zero.

- Case $\phi = 0.25$:

  $$\phi = 0.25 = \frac{25}{100} = 0 + \frac{1}{\frac{100}{25}} = 0 + \frac{1}{4} = \frac{1}{4}$$
  $$s = 1, \qquad r = 4$$

- Case $\phi = 0.50$:

  $$\phi = 0.50 = \frac{50}{100} = 0 + \frac{1}{\frac{100}{50}} = 0 + \frac{1}{2} = \frac{1}{2}$$
  $$s = 1, \qquad r = 2$$

- Case $\phi = 0.75$:

  $$\phi = 0.75 = \frac{75}{100} = 0 + \frac{1}{\frac{100}{75}} = 0 + \frac{1}{1 + \frac{25}{75}} = 0 + \frac{1}{1 + \frac{1}{\frac{75}{25}}} = 0 + \frac{1}{1 + \frac{1}{3}} = \frac{1}{\frac{3 + 1}{3}} = \frac{1}{\frac{4}{3}} = \frac{3}{4}$$
  $$s = 3, \qquad r = 4$$

As we can see from the results the period found by the algorithm is not always correct. That is why we may need to run the algorithm multiple times or run the algorithm again changing the a.

Python implementation of the algorithm

In this section we will discuss the phases of the implementation of Shor's algorithm in Python.

```
from qiskit.utils import QuantumInstance

import numpy as np

from qiskit import QuantumCircuit, Aer, execute

from qiskit.tools.visualization import plot_histogram

import math

from fractions import Fraction

import pandas as pd

print("Imports succesful")
```

Script 5: Snippet 1

The first phase consists in importing the necessary libraries for the algorithm to work. NumPy, math and fractions are libraries related to mathematics and are needed to perform the mathematical operations. Qiskit is the library containing what we need to implement a quantum circuit and run the algorithm. Pandas is a library containing the tools to show, analyze and manipulate data. We will use it to show the tables containing the results of our operations.

```
# Parameters settings

N = 15

a = 2

# I calculate the smallest allowed values for the auxiliary qubits and counting qubits

m = math.ceil(math.log(N, 2))

n_count = 2*m
```

Script 5: Snippet 2

The second phase consists in defining the parameters of the algorithm. This is the only portion of code the user has to edit in order to run the algorithm differently. A user can decide to change the values of N and a only and allow the algorithm to calculate the default value of the other two parameters or they can decide to set bigger values manually.

```
# Functions definition
def c_amodN(a, power):
    U = QuantumCircuit(m)
    for iteration in range(power):
        for qubit in range (m-1, 0, -1):
            U.swap(qubit-1, qubit)
        for q in range(m):
            U.x(q)
    U=U.to_gate()
    U.name="  %i^%i mod%i" %(a, power, N)
    c_U = U.control()
    return c_U


def qft_dagger(n):
    # n-qubit QFTdagger the first n qubits in circ
    qc = QuantumCircuit(n)
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT†"
    return qc
```

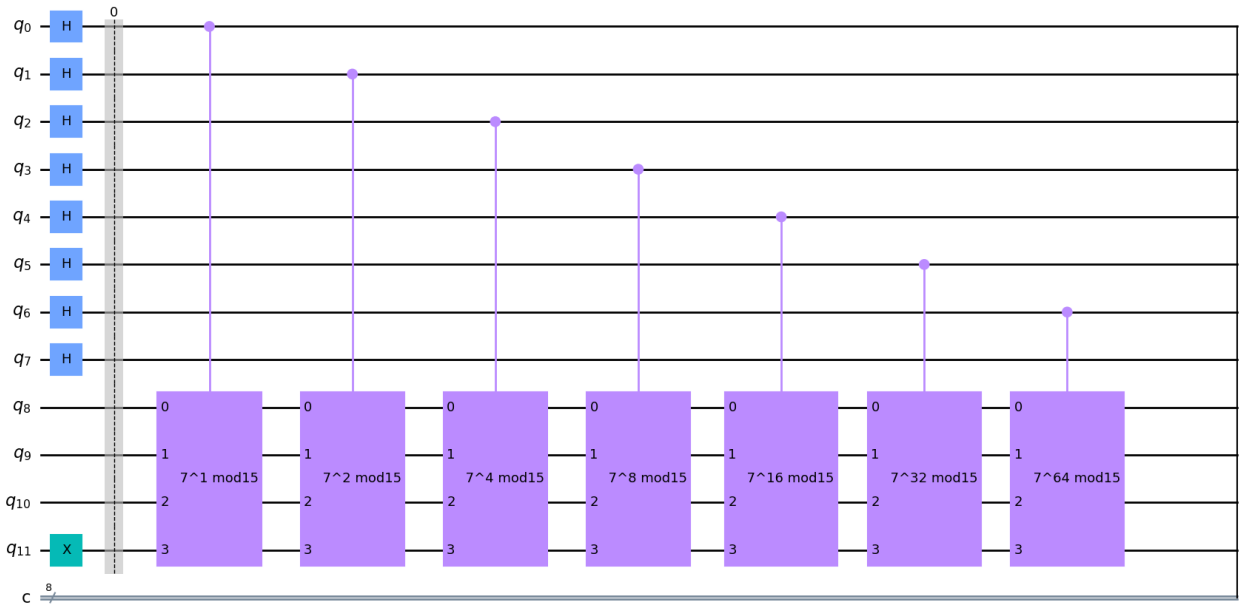
Script 5: Snippet 3

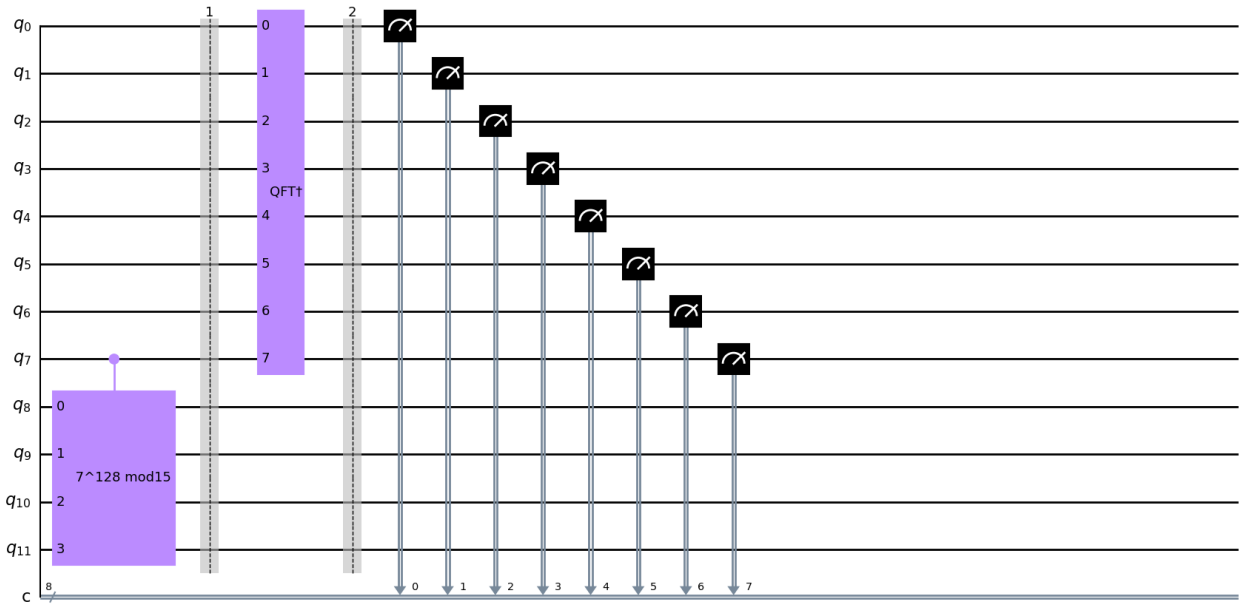
The next phase consists in defining the functions to be used to build the circuit. A basic definition of these functions is documented on Qiskit. We have adapted these functions to use the parameters defined previously.

```
# I build the quantum circuit
qc = QuantumCircuit(n_count + m, n_count)

# I apply an Hadamard gate to the counting qubits
for q in range(n_count):
    qc.h(q)

# I set the last qubit to 1
qc.x(m - 1 + n_count)

qc.barrier(label="0")

for q in range(n_count):
    qc.append(c_amodN(a, 2**q), [q]+[i+n_count for i in range(m)])

qc.barrier(label="1")

qc.append(qft_dagger(n_count), range(n_count))

qc.barrier(label="2")

qc.measure(range(n_count), range(n_count))

qc.draw(output='mpl')
```

Script 5: Snippet 4

In this phase we build the quantum circuit and print it.





The circuit is structured as follows:

- From the beginning to barrier 0:

  In this phase we initialize the qubits, we apply the Hadamard gate to the counting qubits and set to $|1\rangle$ the last qubit because the modular exponentiation requires it.

- Between barrier 0 and barrier 1:

  In this part of the circuit we modify the counting qubits using the modular exponentiation function.

- Between barrier 1 and barrier 2:

  We use the inverse quantum Fourier transform to convert the qubits from the Fourier state to the

computational state so that we can measure them.

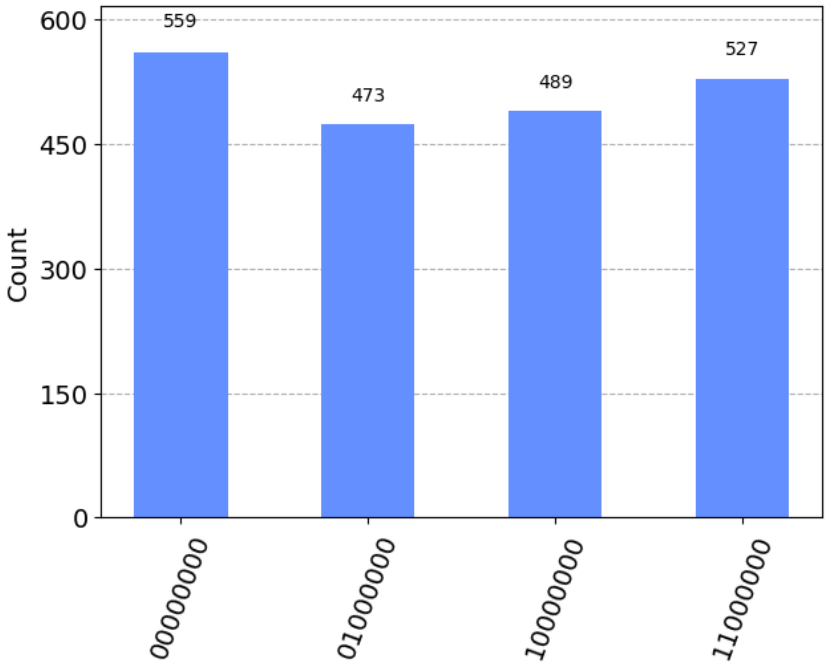
- After barrier 2:

  We measure the counting qubits and store the result in the register at the bottom of the circuit.

```
# I run the algorithm
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend, shots=2048).result()
counts = results.get_counts()
plot_histogram(counts)
```

Script 5: Snippet 5

Now that we have built our circuit, we can run the algorithm multiple times and show the possible results using a histogram.



At the bottom of the histogram, we can see the binary strings containing the possible results. The least significant bit is located at the top of the string so the results from the leftmost to the rightmost are: 0, 64, 128, 192.

At this point we can consider that if we decided to run the algorithm only once the process could fail because we can get the results 0 and 2 which respectively don't lead to a solution and lead to a partial solution.

Now, regardless of the count of each result, we have all the possible results of the algorithm and we can work on them to find the correct solution.

```
rows, measured_phases = [], []
for output in counts:
    decimal = int(output, 2)  # Convert (base 2) string to decimal
    phase = decimal/(2**n_count)  # Find corresponding eigenvalue
    measured_phases.append(phase)
    # Add these values to the rows in our table:
    rows.append([f"{output}(bin) = {decimal:>3}(dec)",
                 f"{decimal}/{2**n_count} = {phase:.2f}"])
# Print the rows in a table
headers = ["Register Output", "Phase"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

Script 5: Snippet 6

At this point we can calculate the measured phase by dividing the measured value by 2^t and show a table containing the results for convenience.

	Register Output	**Phase**
0	00000000(bin) = 0(dec)	0/256 = 0.00
1	11000000(bin) = 192(dec)	192/256 = 0.75
2	10000000(bin) = 128(dec)	128/256 = 0.50
3	01000000(bin) = 64(dec)	64/256 = 0.25

```
rows = []
r_array = []
for phase in measured_phases:
    frac = Fraction(phase).limit_denominator(N)
    if(frac.denominator not in r_array):
        r_array.append(frac.denominator)
    rows.append([phase,
                 f"{frac.numerator}/{frac.denominator}",
                 frac.denominator])
# Print as a table
headers = ["Phase", "Fraction", "Guess for r"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

Script 5: Snippet 7

Now that we have the phase expressed as a decimal number, we have to express it as a fraction to calculate its denominator. In Python we don't have to implement the continued fractions algorithm by hand because it is implemented in the class "Fraction". The only problem deriving from the usage of this class is that it provides a fraction which leads to the exact result even if the numerator and the denominator are large numbers. To solve this problem, we can limit the denominator to the number N and make the class give the fraction which value is closest to the decimal value provided in input with a denominator which is lesser or equal to N.

For conveniency we can print a table containing the phases, the fractions and the guesses for r and we can store the unique values of the denominator in an array to use them later.

	Phase	**Fraction**	**Guess for r**
0	0.25	1/4	4
1	0.75	3/4	4
2	0.50	1/2	2
3	0.00	0/1	1

In the case in which $s = 0$ the "Fraction" class assumes that the numerator is 0 and the denominator is 1 even if, mathematically speaking, we can only say that the numerator is 0 and the denominator is different from 0.

```
# I create an array of unique and valid possible values of r
print("All the unique values found for r are: " + str(r_array))
valid_r = []
for element in r_array:
    if(element%2 == 0 and element not in valid_r):
        valid_r.append(element)
print("The valid values for Shor's algorithm are: " + str(valid_r))
```

Script 5: Snippet 8

Shor's algorithm requires the r to be an even number so we have to discard the odd numbers and put the even ones in another array.

The output of this snippet is as follows:

All the unique values found for r are: [4, 2, 1]

The valid values for Shor's algorithm are: [4, 2]

```
# I create a table containing the r value and the guesses for p and q
# In the process I can add a comment to the row indicating whether the result found is the solution,
# a partial solution or not a solution.
# I save the solution in an array and show it at the end

results_rows = []
solution = []
partial_solution = []
for value in valid_r:
    p = math.gcd(int(a**(value/2)-1), N)
    q = math.gcd(int(a**(value/2)+1), N)

    if((p == N or q == N) and (p == 1 or q == 1)):
        comment = "Not a solution"
    elif(p == 1 or p == N or q == 1 or q == N or p == q):
        comment = "Partial solution"
        if(len(partial_solution) == 0):
            partial_solution.append(p)
            partial_solution.append(q)
    else:
        comment = "Valid solution"
        if(len(solution) == 0):
            solution.append(p)
            solution.append(q)

    results_rows.append([value, p, q, comment])

results_headers = ["r value", "p", "q", "Comment"]
results_table = pd.DataFrame(results_rows, columns=results_headers)
print(results_table)
```

Script 5: Snippet 9

For each valid value of r we can calculate the two numbers p and q and check whether they make a valid solution, a partial solution or they don't make a solution. In the process, we can add a comment to the result and store a valid solution and a partial solution. There is no need to store more than one valid solution and one partial solution even if, in the second case, one partial solution can contain a nontrivial prime number and another partial solution can contain the other nontrivial prime number because if we have one of them we can calculate the other.

This snippet generates the following table:

	r value	p	q	Comment
0	2	3	1	Partial solution
1	4	3	5	Valid solution

```
if(len(solution) > 0):
    print("The solution is: " + str(solution))
elif(len(partial_solution) > 0):
    print("The algorithm found a partial solution: " + str(partial_solution) + "\n")
    # We can derive the complete solution starting from the nontrivial number
    if(partial_solution[0] != 1 and partial_solution[0] != N):
        solution.append([partial_solution[0], int(N/partial_solution[0])])
    else:
        solution.append([partial_solution[1], int(N/partial_solution[1])])
    print("From the partial solution we can derive the two prime numbers: " + str(solution))
else:
    print("This run of the algorithm could not find a solution, try running the algorithm again with a different \"a\"")
```

Script 5: Snippet 10

Now we can finally print the result of the factoring algorithm. We start by checking if the algorithm found a valid solution and if it did not, we can calculate the solution starting from a partial one. In case the algorithm could not find a solution we show a message that tells the user to run the algorithm again changing the a value.

In this example the algorithm finds a valid solution, so the output of the last snippet is:

```
The solution is: [3, 5]
```

Time complexity of Shor's algorithm

The documented time complexity of Shor's algorithm is:

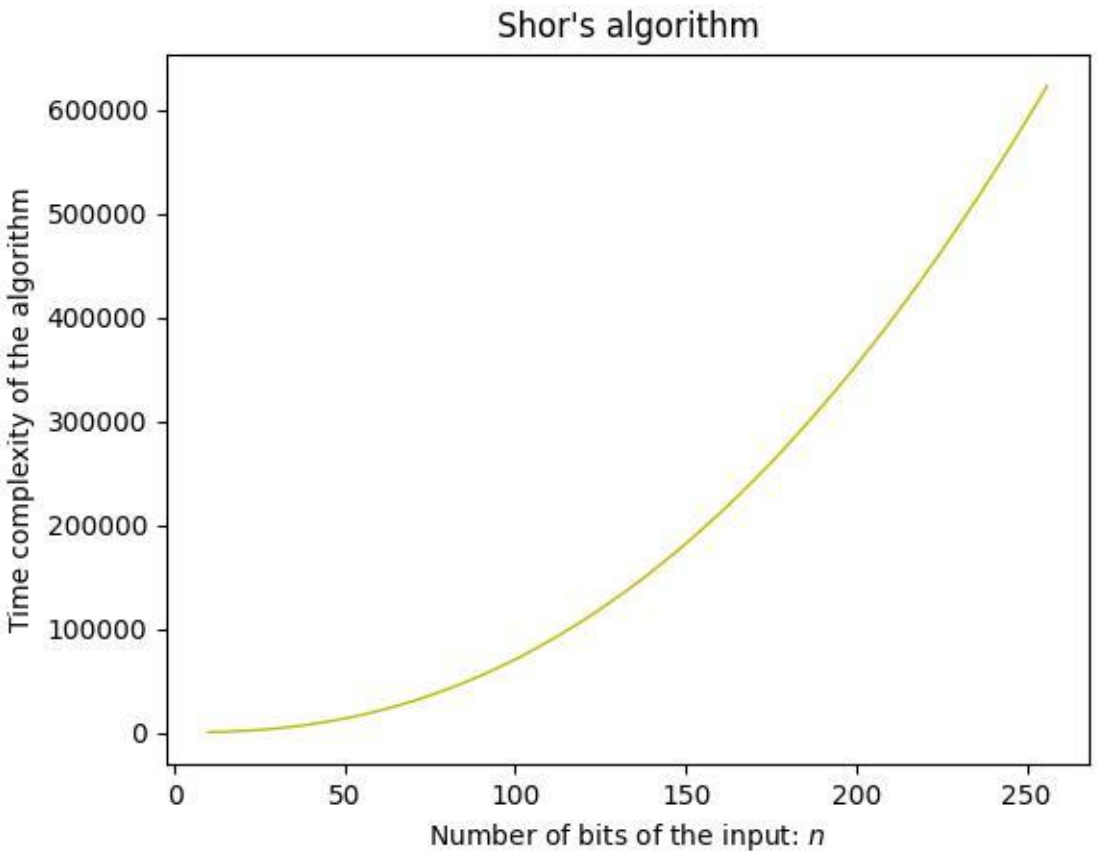
$$O(n^2 \cdot \log(n) \cdot \log(\log(n)))$$

Cryptanalysts find this result particularly interesting because it does not contain an exponential and it contains logarithms. This means that with the progress of this technology this will become the fastest

algorithm to break RSA cryptosystems and it will become a standard to refer to during the development of new encryption algorithms.

In the current state of things this technology is still not capable of factoring an RSA public key of 256 bits or more in an acceptable amount of time. For now, we can factor small semiprimes such as $15 = 5 \cdot 3$ or $22 = 2 \cdot 11$ quickly. If we try to factor bigger semiprimes, even if they are smaller than 100, such as $91 = 13 \cdot 7$ the quantum computer takes a long amount of time.

This is reassuring because it means that for the moment the privacy of our sensitive data and credit card numbers is still granted by the current encryption systems!



Plot 5: Time complexity of Shor's algorithm

References

Brute force algorithm:

- Runtime complexity of a brute force factoring algorithm: https://cs.stackexchange.com/questions/115019/runtime-complexity-of-a-brute-force-factoring-algorithm-in-terms-of-bits

Fermat's factorization algorithm:

- Fermat's factorization method: https://en.wikipedia.org/wiki/Fermat%27s_factorization_method
- Time complexity of binary multiplication: https://math.stackexchange.com/questions/226394/time-complexity-of-binary-multiplication

Quadratic sieve algorithm:

- Quadratic sieve: https://en.wikipedia.org/wiki/Quadratic_sieve
- Here's How Quadratic Sieve Factorization Works: https://medium.com/nerd-for-tech/heres-how-quadratic-sieve-factorization-works-1c878bc94f81
- SIEVE OF ERATOSTHENES ALGORITHM: https://www.topcoder.com/thrive/articles/sieve-of-eratosthenes-algorithm
- SymPy 1.12 documentation: https://docs.sympy.org/latest/index.html

Quantum computing:

- Qiskit Textbook: https://qiskit.org/learn
- Qiskit Aer: https://github.com/Qiskit/qiskit-aer
- Quantum Fourier transform: https://en.wikipedia.org/wiki/Quantum_Fourier_transform https://cnot.io/quantum_algorithms/qft/ https://learn.qiskit.org/course/ch-algorithms/quantum-fourier-transform
- Quantum Phase Estimation: https://learn.qiskit.org/course/ch-algorithms/quantum-phase-estimation
- Quantum Period Finding: https://anonymousket.medium.com/quantum-period-finding-qpf-a6bd3d95e24c

Shor's algorithm:

- NumPy documentation: https://numpy.org/doc/1.23/
- pandas documentation: https://pandas.pydata.org/pandas-docs/version/2.0/index.html
- Shor's algorithm: https://en.wikipedia.org/wiki/Shor%27s_algorithm https://learn.qiskit.org/course/ch-algorithms/shors-algorithm https://www.youtube.com/watch?v=lvTqbM5Dq4Q https://www.youtube.com/watch?v=EdJ7RoWcU48

https://www.youtube.com/watch?v=mAHC1dWKNYE
https://www.youtube.com/watch?v=pq2jkfJlLmY

- Euclidean algorithm:
  https://en.wikipedia.org/wiki/Euclidean_algorithm
  https://www.youtube.com/watch?v=yHwneN6zJmU
- Continued fraction:
  https://en.wikipedia.org/wiki/Continued_fraction
- Scalable Shor's Algorithm:
  https://learn.qiskit.org/course/ch-labs/lab-7-scalable-shors-algorithm

Useful resources:

- Matplotlib 3.7.2 documentation:
  https://matplotlib.org/stable/index.html
- List of prime numbers up to 1 000 000 000 000 (1000 billion):
  http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php