# Shor's Algorithm

*Mat Jones*
*CS360*

## Abstract

Although all integers have a unique breakdown of prime factors, finding those prime factors is thought to be a hard problem in computer science; in fact, most modern cyber security and encryption measures depend on the assumption that solving this problem for integers of a thousand or more digits is not feasibly possible (“Shor’s Algorithm”). However, in 1995, MIT professor Peter Shor presented a quantum algorithm (an algorithm that runs on a quantum computer) to solve the factoring problem in polynomial time. Shor's algorithm thus drastically changed the set of problems in computer science that can be considered feasible.

## Introduction

Shor's algorithm could potentially break nearly all modern cryptography. Most encryption schemes use a public & private key paradigm; each device has two keys (public and private key), and the public keys are exchanged between devices. The keys are generated such that data encrypted using the public key can only be decrypted using the private key. Since the keys must be mathematically related (often based on arithmetic involving prime factors), the private key can theoretically be calculated from the public key. However, finding the prime factors of an integer of a thousand digits or more becomes infeasible on a classical computer; thus, calculating the private key from the public key could take hundreds of years on a classical computer, depending on the encryption scheme used.

Shor's algorithm is a quantum algorithm for efficiently finding the prime factors of large integers, and thus calculating a private key from a public key could become a feasible problem in the near future. For this reason, it is important that software professionals prepare for the eventuality of accessible quantum computing. While theoretically effective post-quantum cryptography has been proposed, existing cyber security infrastructure (made up of classical computers) is not equipped to protect against attacks by a quantum computer.

## Implementation

### Period Finding

It has been widely known by mathematicians and computer scientists since the 1970's that the factorization problem becomes easy if given a "period finding machine"; that is, a machine which can efficiently find a period of the modular exponentiation function (“Shor’s Algorithm”).

The period finding problem is defined as follows: Given two integers $N$ and $a$, find the smallest possible integer $r$ such that $a^r - 1$ is a multiple of $N$. The number $r$ is the period of $a \mod N$. Or, more simply, find the minimum possible value for $r$ such that:

$$a^{x+r} = a^x \pmod N$$

where $x$ is any integer. Consider the following example for $N = 15$ and $a = 7$:

$$7^2 = 4 \pmod{15} = 4$$
$$7^3 = 4 \times 7 = 13 \pmod{15} = 13$$
$$7^4 = 13 \times 7 = 91 \pmod{15} = 1$$

This sequence gives rise to the pattern $7^{x+4} = 7^x \pmod{15}$, and thus $r = 4$ is the period of the modular exponentiation function $7 \pmod{15}$.

## Using Period Finding Machine to Find Prime Factors

Suppose a period finding machine is given that takes two co-prime integers $a$ and $N$ and outputs the period of $a \mod N$. Then, the prime factors $p_1, p_2$ of $N$ can be found by the following procedure:

1. Choose a random integer $a$ between $2..N-1$.
2. Compute the greatest common divisor (GCD) of $a$ and $N$ (can be calculated efficiently using [Euclid's algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm)](https://en.wikipedia.org/wiki/Euclidean_algorithm)); if this value is not equal to $1$, then it equals either $p_1$ or $p_2$, and the other can be computed from the found factor (e.g. $N/p_1$ or $N/p_2$). Otherwise, continue.
3. Let $r$ be the period of $a \mod N$; repeat steps 1-3 until $r$ is even. At this point, a value for $r$ has been found such that $a^r - 1$ is a multiple of $N$.
4. Consider the following identity:
   $a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1)$

Thus, $a^{r/2} - 1$ is not a multiple of $N$ (or else $r$ would equal $r/2$). If $a^{r/2} + 1$ is a multiple of $N$, return to step 1. Otherwise, this means that neither of the values $a^{r/2} \pm 1$ are multiples of $N$, but the product of these two values *is* a multiple of $N$. This is possible if and only if $p_1$ is a prime factor of $a^{r/2} - 1$ and $p_2$ is a prime factor of $a^{r/2} + 1$ or vice versa. Therefore, $p_1$ and $p_2$ can now be computed by $p_1 = GCD(N, a^{r/2} - 1)$ and $p_2 = GCD(N, a^{r/2} + 1)$. Consider the following table showing the calculated prime factors of $N = 15$ given different $a$ values:

| $a$ | $r$ | $GCD(N, a^{r/2} - 1)$ | $GCD(N, a^{r/2} + 1)$ |
|---|---|---|---|
| 1 | 1 | | |
| 2 | 4 | 3 | 5 |
| 4 | 2 | 3 | 5 |
| 7 | 4 | 3 | 5 |
| 8 | 4 | 3 | 5 |
| 11 | 2 | 5 | 3 |
| 13 | 4 | 3 | 5 |
| 14 | 2 | 1 | 15 |

*Figure 1: Resultant factors for various N values.*

Notice that $a = 14$ is the only "unlucky" integer chosen (i.e. it results in $GCD(N, a^{r/2} + 1)$ being a multiple of $N$); in general, it can be shown that selection of "unlucky" integers are rather infrequent ("Shor's Algorithm").

## Using a Quantum Computer to Simulate Period Finding Machine

The true power of Shor's algorithm lies in the ability of a quantum computer to simulate such a "period finding machine" required to efficiently solve the factoring problem. By exploiting quantum parallelism and constructive interference, certain "global properties" can be derived from a complex function. The Deutsch-Jozsa algorithm uses this technique to determine if a function has the global property of being a balanced function ("Deutsch-Jozsa Algorithm"). In Shor's algorithm, it can be used to determine the periodicity of the modular exponentiation function.

Suppose two co-prime integers $a$ and $N$ are given; the goal is to find the smallest possible integer $r$ such that $a^r = 1 \pmod{N}$. A unitary operator $U_a$ can be constructed which implements the modular multiplication function such that $U_a : x \rightarrow ax \pmod{N}$. With this definition of $U_a$, it can be shown that the each Eigenvalue of $U_a$ is of the form $e^{i\phi}$ where $\phi = 2\pi k/r$ for some integer $k$ ("Shor's Algorithm").

Eigenvalues of a unitary operator can be efficiently measured using the quantum phase estimation algorithm ("Quantum Phase Estimation"). However, the value of $r$ can only be derived from a given Eigenvalue if the Eigenvalue is measured *exactly* or with exponentially small precision; for example, a factorization of a $1000$ digit integer would require an Eigenvalue measured with a precision of $10^{-2000}$ ("Shor's Algorithm"). Such a level of precision cannot be achieved using the phase estimation algorithm because it would require too large a pointer system.

Instead, define a family of unitary operators $U_b$ with $b = a, a^2, a^4, a^8, \ldots a^{2^P}$ where $P \approx N^2$. Notice that all $U_b$ are integer powers of $U_a$; thus, if $b = a^t$ for some integer $t$, then $U_b = (U_a)^t$, which implies that all $U_b$ have the same Eigenvectors as $U_a$. Further, this means that the Eigenvalues of all $U_b$ can be derived simultaneously. Additionally, the implementation of $U_b$ is as easy as implementing $U_a$; simply pre-compute the powers of $U_a$ by repeated squaring method. Conveniently, each squaring of $a$ reduces the margin of error in the estimation of $U_a$ by a factor of

$1/_2$; this mitigates the requirement for exact or exponentially small precision in the phase estimation algorithm. For example, a precision of $10^{-2000}$ can be achieved by a series of $10^6$ less precise measurements (up to 10% error) of $U_b$, and selecting a few Eigenvalues $\phi = 2\pi k/r$ at random to estimate with a precision of $1/N^2$ is enough to derive an exact value of $r$ using rational approximation to estimate $k/r$ ("Shor's Algorithm").

### Reversible Classical Circuits

A quantum circuit which implements the modular multiplication operator is required in order to use the phase estimation gate. Quantum algorithms can call classical subroutines if the classical subroutine is first transformed into a *reversible form*, i.e. represented by a sequence of reversible logic gates (CNOT gate, Toffoli gate, etc). That is, the number of input wires must equal the number of output wires for each gate. Further, the classical subroutine can use scratch memory for local variables, but the scratch memory must be wiped clean upon termination of the subroutine; leaving residual values in the scratch memory could potentially destroy quantum coherence, preventing the quantum routine from being able to detect interference between quantum states ("Shor's Algorithm").

Consider a standard AND gate; it can be transformed into a reversible equivalent (R-AND) by adding an extra input wire $d$ (so, inputs $a, b, d$), which is a dummy wire expecting a value of $d = 0$, and adding two extra output wires (so, outputs $a, b, c$) where $c$ is the result of $d \oplus (a \wedge b)$ (see figure 2). Thus, all inputs of the R-AND can be derived from its outputs since $c = d \oplus (a \wedge b)$ ("Shor's Algorithm").

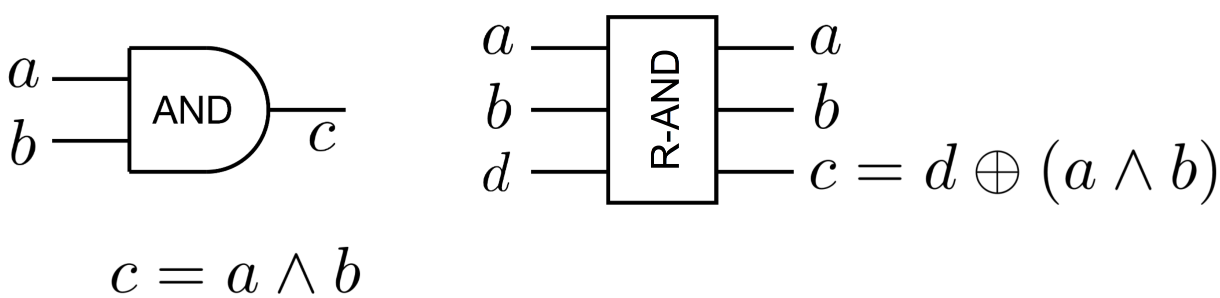



*Figure 2: Reversible AND (R-AND) gate. (Source: "Shor's Algorithm")*

Similar logic can be applied to any gate with two inputs and one output; if gate F computes a boolean function $c = F(a, b)$, then a reversible transformation (R-F gate) would map inputs $a, b, d$ to outputs $a, b, c$ where $c = d \oplus F(a, b)$ ("Shor's Algorithm"). See figure 3.

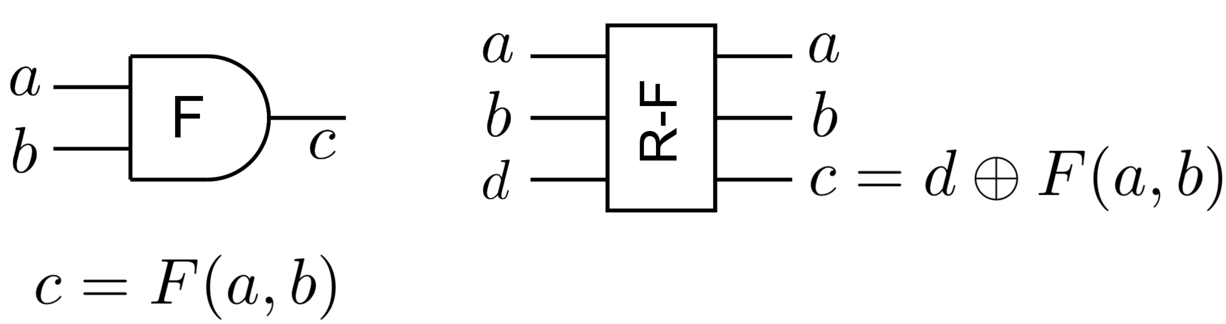



*Figure 3: Reversible F (R-F) gate. (Source: "Shor's Algorithm")*

### Quantum Circuit for Modular Multiplication

The modular exponentiation function can be derived from a series of modular multiplication functions. Let the modular multiplication function be defined as $f(x) = ax \pmod{N}$. If the integer results of $f(x)$ are stored as $n$-bit strings, then $f(x)$ can be implemented as a classical circuit $U_a$ (unitary operator described above) using 3-bit reversible classical gates with $n$ inputs and $n$ outputs via the reversible gate transformation technique described above ("Shor's Algorithm"). Since $U_a$ is now composed completely of reversible classical circuits, it can be called as a classical subroutine from a quantum routine.

### IBM Quantum Experience

IBM has a platform called the IBM Quantum Experience which allows curious computer scientists to experiment with quantum computing concepts. IBM quantum hardware runs [OpenQASM (https://github.com/QISKit/openqasm)](https://github.com/QISKit/openqasm) quantum assembly code, but the web interface also features called "Composer" which allows quantum and classical gates to be click-and-dragged onto a virtual circuit diagram. IBM also has a Python SDK which allows developers to implement and run quantum subroutines from within Python code, either on a local quantum simulator, an IBM Quantum Experience simulator, or on real IBM quantum hardware.

A basic, crude implementation of Shor's algorithm can be found below; if running in a Jupyter Notebook, it can be run interactively. The full Python script can be found in the appendix.

In [8]:

```python
# %load QuantumProgramRunner.py
#!/usr/bin/env python3
from Runner import run

if __name__ == "__main__":
    while run(None):
        pass
```

```
Factorizing N=6378689...
Chose unlucky 'a' value, trying again with new 'a' value (18th try so fa
r)...
Selected random value a=4854918 to find period.
Found common period between N=6378689 and a=4854918
Took 18 guesses for 'a' value.
Found factors: 37 X 172397 = 6378689
Available programs:
  1. find_period: Takes two integers, a and N, and finds the period of th
e modular exponentiation function.
  2. factorize_N: Takes an integer N and finds factors of N using Shor's
 algorithm.
Select a program to run, or type 'exit' to quit.
> exit
```

## Results

Because I could only get the QASM code to run properly on a quantum simulator, and not real quantum hardware at IBM, I was unable to get valid or usable results from my attempted benchmarking. The test data, results, and graphs I generated, as well as the Python scripts used to generate them, can be found in the appendix.

The time complexity of integer factorization and Shor's algorithm is a function of the number of digits $d$. The brute force solution for an integer $N$ simply iterates through prime numbers $p$ up to $\sqrt{N}$ and checks if $N \pmod{p} \stackrel{?}{=} 0$. A more efficient solution, known as the quadratic sieve technique, searches for two integers $a, b$ such that $(a^2 - b^2) \mod N \stackrel{?}{=} 0$; this method has a runtime bounded by $O(\sqrt{d})$ where $d$ is the number of digits in $N$. The most efficient known classical algorithm, the general number field sieve achieves a complexity of $O(d^{1/3})$. Shor's algorithm achieves a time complexity *polynomial* in terms of $d$ ("Shor's Algorithm").

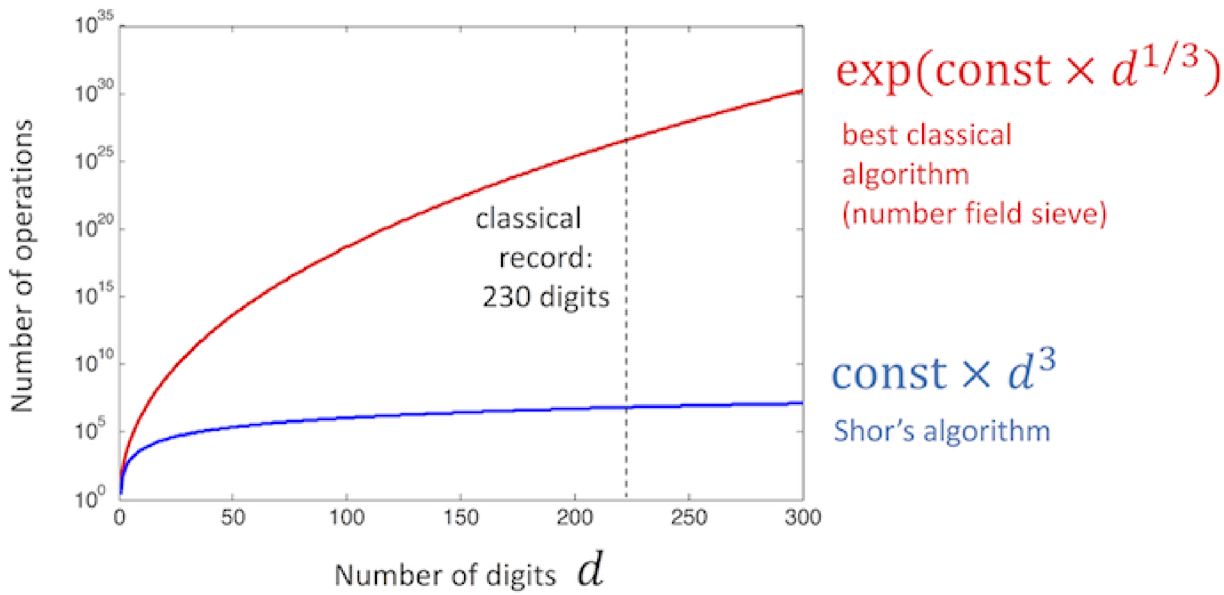



*Figure 4: Shor's algorithm vs. the general number sieve algorithm. (Source: "Shor's Algorithm")*

## Extensions

To date, Shor's algorithm is the only known algorithm for prime factorization of large integers in polynomial time. On classical computers, the Sieve of Eratosthenes can be used to find prime factors of integers in $O(nlg(lg(n)))$ time (*Sorenson 1990*). Despite being published in 1995, Shor's algorithm is still at the bleeding edge of prime factorization algorithms and can be shown to have a time complexity of $O((lg\, n)^2(lg\, lg\, n)(lg\, lg\, lg\, n))$ (*Beckman et al. 1996*).

## Conclusion

Shor's algorithm is perhaps the most radical example of how quantum computing has changed (and can continue to change) the set of problems considered feasible. In the past, prime factorization was a problem considered so infeasible that a significant portion of existing cryptography infrastructure is based on this assumption; this means that quantum computing technology, along with Shor's algorithm, could be used to crack nearly any common encryption used today. Shor's algorithm is on the bleeding edge of quantum computing; as quantum computing technology has gained traction in recent years, it has been used as a sort of litmus test for quantum computer viability. IBM first demonstrated a simple implementation of Shor's algorithm on one of their early quantum computers in 2001, factorizing the number 15 into its factors 3 and 5 (*"IBM's Test-Tube Quantum Computer Makes History" 2001*). Significant progress has been made since 2001 in the realm of quantum computing technology, and for this reason, its important to begin considering it as an inevitability which will come to fruition sooner rather than later.

# Works Cited


Beckman, David, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. 1996. "Efficient Networks for Quantum Factoring." Cornell University. https://arxiv.org/abs/quant-ph/9602016.

"Deutsch-Jozsa Algorithm." Documentation. *IBM Q Experience Documentation.* https://quantumexperience.ng.bluemix.net/proxy/tutorial/full-user-guide/004-Quantum_Algorithms/080-Deutsch-Jozsa_Algorithm.html.

"IBM's Test-Tube Quantum Computer Makes History." 2001. *IBM News Room*. December 19. https://www-03.ibm.com/press/us/en/pressrelease/965.wss.

"Quantum Phase Estimation." Documentation. *IBM Q Experience Documentation.* https://quantumexperience.ng.bluemix.net/proxy/tutorial/full-user-guide/004-Quantum_Algorithms/100-Quantum_Phase_Estimation.html.

"Shor's Algorithm." Documentation. *IBM Q Experience Documentation.* https://quantumexperience.ng.bluemix.net/proxy/tutorial/full-user-guide/004-Quantum_Algorithms/110-Shor's_algorithm.html.

Sorenson, Jonathan. 1990. "An Introduction to Prime Number Sieves." University of Wisconsin-Madison. http://research.cs.wisc.edu/techreports/1990/TR909.pdf.


# Appendix

## GitHub

https://github.com/mrjones2014/CS360-shors-algorithm (https://github.com/mrjones2014/CS360-shors-algorithm)

## QuantumCircuits.py

```
from qiskit import QuantumProgram
from qiskit import QuantumCircuit
from qiskit import QuantumRegister
from qiskit import ClassicalRegister
from pyspin.spin import make_spin, Default
from IPython.display import clear_output
from math import sqrt; from itertools import count, islice
from qiskit import Result
import PrintUtils
import QConfig
import random
import math

class Circuits:
    # String constants for circuit names
    PERIOD = "circuit_period" # find_period() quantum circuit

class QRegs:
    # String constants for quantum register names
    PERIOD = "qreg_period" # find_period() quantum register

class CRegs:
    # String constants for classical register names
    PERIOD = "creg_period" # find_period() classical register

class QuantumPrograms:
    PROGRAMS = {
        "find_period": "Takes two integers, a and N, and finds the p
eriod of the modular exponentiation function.",
        "factorize_N": "Takes an integer N and finds factors of N us
ing Shor's algorithm."
    }
    """
    A class containing quantum circuits used in Shor's algorithm.
    Constructor takes an instance of a QuantumProgram object from QI
SKit module
    in order to create the circuits/run code on the IBM Quantum Expe
rience hardware.
    """

    def __init__(self, quantum_program: QuantumProgram, qconfig: QCo
nfig):
        """Store QuantumProgram instance in self."""
        self.qp = quantum_program
        self.qconf = qconfig

    def gcd(self, a, b):
        """Find Greatest Common Divisor (GCD) using Euclid's algorit
hm"""
```

```
        while b != 0:
            (a, b) = (b, a % b)
        return a

    def isPrime(self, n):
        return n > 1 and all(n%i for i in islice(count(2), int(sqrt(
n)-1)))

    def factorize_N(self, N, numRetries=0):
        """Factorize N using Shor's algorithm."""
        PrintUtils.printInfo(f"Factorizing N={N}...")
        if numRetries > 0:
            clear_output()
            if numRetries > 1:
                PrintUtils.delete_last_lines(6)
            else:
                PrintUtils.delete_last_lines(5)
            PrintUtils.printInfo(f"Factorizing N={N}...")
            PrintUtils.printWarning(f"Chose unlucky 'a' value, tryin
g again with new 'a' value ({PrintUtils.toOrdinal(numRetries + 1)} t
ry so far)...")

        # Step 1: check if N is even; if so, simply divide by 2 and
 return the factors
        if N % 2 == 0:
            return [2, int(N/2)]
        # Step 2: choose random value for 'a' between 2..(N-1)
        a = random.randint(2, N-1)
        PrintUtils.printInfo(f"Selected random value a={a} to find p
eriod.")
        # Step 3: determine if common period exists
        t = self.gcd(N, a)
        if t > 1:
            PrintUtils.printInfo(f"Found common period between N={N}
 and a={a}")
            PrintUtils.printSuccess(f"Took {numRetries + 1} guesses
 for 'a' value.             ")
            return [t, int(N/t)]
        # Step 4: t=1, thus, N and a do not share common period. Fin
d period using Shor's method.
        PrintUtils.printInfo("Using Shor's method to find period..."
)
        r = self.find_period(a, N)
        factor1 = self.gcd((a**(r/2))+1, N)
        if factor1 % N == 0 or factor1 == 1 or factor1 == N or not(s
elf.isPrime(factor1)):
            return self.factorize_N(N, numRetries + 1)
        factor2 = N/factor1
        if not self.isPrime(factor2):
```

```
                return self.factorize_N(N, numRetries + 1)
            PrintUtils.printSuccess(f"Took {numRetries + 1} guesses for 'a' value.                ")
            return [int(factor1), int(factor2)]

    @make_spin(Default, "Finding period using Shor's method...", "\r\n")
    def find_period(self, a, N):
        """
        Find the period of the modular exponentiation function,
        i.e. find r where (a^x) % N=(a^[x + r]) % N where x is any integer.
        For example:
        (7^2) % 15 = 4 % 15 = 4
        (7^3) % 15 = (4 * 7) % 15 = 13 % 15 = 13
        (7^4) % 15 = (13 * 7) % 15 = 91 % 15 = 1
        Thus, for a=7 and N=15, the periodic sequence is  (7^x) % 15 = (7^[x + 4]) % 15 for any integer x;
        therefore, the period for the modular exponentiation function for a=7 and N=15 is r=4.
        Returns a tuple containing the value of r and the number of iterations required to find r,
        (r, iterCount)
        """
        self.create_modular_multiplication_circuit()
        iterCount = 0
        r = math.inf # initialize r to infinity
        while not ((r%2 == 0) and (((a**(r/2))+1)%N != 0) and (r != 0) and (r != 8)):
            iterCount += 1
            result: Result = self.qp.execute([Circuits.PERIOD], backend=self.qconf.backend, shots=self.qconf.shots, timeout=self.qconf.timeout)
            # print(result)
            data = result.get_counts(Circuits.PERIOD)
            # print(data)
            data = list(data.keys())
            # print(data)
            r = int(data[0])
            # print(r)
            l = self.gcd(2**3, r)
            # print(l)
            r = int((2**3)/l)
            # print(r)
        return r

    def create_modular_multiplication_circuit(self):
        qr = self.qp.create_quantum_register(QRegs.PERIOD, 5)
        cr = self.qp.create_classical_register(CRegs.PERIOD, 3)
```

```
        self.qp.create_circuit(Circuits.PERIOD, [qr], [cr])
        # re-fetch circuit and registers by name
        circuit: QuantumCircuit  = self.qp.get_circuit(Circuits.PERI
OD)
        qreg: QuantumRegister = self.qp.get_quantum_register(QRegs.P
ERIOD)
        creg: ClassicalRegister = self.qp.get_classical_register(CRe
gs.PERIOD)

        ## Set up the quantum circuit
        # Initialize: set qreg[0] to |1> and
        # create superposition on top 8 qbits
        circuit.x(qreg[0])

        ## Step 1: apply a^4 % N
        circuit.h(qreg[2])
        # Controlled Identity gate
        circuit.h(qreg[2])
        circuit.measure(qreg[2], creg[0]) # store the result
        # Reinitialize to |0>
        circuit.reset(qreg[2])
        ## Step 2: apply a^2 % N
        circuit.h(qreg[2])
        # Controlled Identity gate
        if creg[0] == 1:
            circuit.u1(math.pi/2.0, qreg[2])
        circuit.h(qreg[2])
        circuit.measure(qreg[2], creg[1]) # store the result
        # Reinitialize to |0>
        circuit.reset(qreg[2])
        ## step 3: apply a % N
        circuit.h(qreg[2])
        # Controlled NOT (C-NOT) gate in between remaining gates
        circuit.cx(qreg[2], qreg[1])
        circuit.cx(qreg[2], qreg[4])

        ## Feed forward
        if creg[1] == 1:
            circuit.u1(math.pi/2.0, qreg[2])
        if creg[0] == 1:
            circuit.u1(math.pi/4.0, qreg[2])
        circuit.h(qreg[2])
        circuit.measure(qreg[2], creg[2]) # store the result
        # print(circuit.qasm()) # print QASM code
```

**QConfig.py**

```python
class QConfig:
    def __init__(self, backend, shots, timeout, program=None):
        self.backend = backend
        self.shots = shots
        self.timeout = timeout
        self.program = program
```

### Runner.py

```
import QuantumCircuits
import ExperimentUtils
import SignalUtils
import PrintUtils
import random
import sys

def run(args):
    return run_experiment(ExperimentUtils.setup_experiment(args))

def run_experiment(experiment):
    print("")
    program = experiment.qconf.program.strip()
    timeout = experiment.qconf.timeout
    if program == "exit":
        try:
            sys.exit()
        except:
            try:
                quit()
            except:
                return
    elif program == "find_period":
        N = int(input("Enter a value for N:\nN = "))
        a = input("Enter a value for a (or type 'rand' for random va
lue between 2..N-1):\na = ")
        if a == "rand":
            a = random.randint(2, N-1)
        else:
            a = int(a)
        def run_expr():
            r = experiment.find_period(a, N)
            PrintUtils.printSuccess(f"Found period r={r} for a={a} a
nd N={N}.")
        SignalUtils.tryExecuteWithTimeout(run_expr, timeout, f"\nFai
led to find period within timeout: {timeout} seconds.")
        return True
    elif program == "factorize_N":
        N = int(input("Enter a value N to factorize:\nN = "))
        def run_expr():
            factors = experiment.factorize_N(N)
            PrintUtils.printSuccess(f"Found factors: {factors[0]} X
 {factors[1]} = {N}")
        SignalUtils.tryExecuteWithTimeout(run_expr, timeout, f"Faile
d to factorize {N} within timeout: {timeout} seconds.")
        return True
    else:
        PrintUtils.printErr(f"Invalid program '{program}'")
```

**ExperimentUtils.py**

```python
from QuantumCircuits import QuantumPrograms
from qiskit import QuantumProgram
from QConfig import QConfig
import PrintUtils
import subprocess
import os

def print_available_programs(programs):
    PrintUtils.printHeader("Available programs:")
    i = 1
    for progname in programs.keys():
        PrintUtils.printInfo(f"  {i}. {progname}: {programs[prognam
e]}")
        i += 1

def getParams():
    programs = QuantumPrograms.PROGRAMS

    backend = 'local_qasm_simulator'
    shots = 1024
    timeout = 120
    program = None

    engine = QuantumProgram()

    def tryParseInt(value):
        try:
            return int(value)
        except:
            return None

    print_available_programs(programs)
    program = input("Select a program to run, or type 'exit' to qui
t.\n> ").strip()
    progNum = tryParseInt(program)
    while program not in programs and program != "exit":
        if progNum is not None:
            if progNum > len(programs):
                PrintUtils.printErr(f"Invalid program '{progNum}'")
            else:
                program = list(programs.keys())[progNum - 1]
                break
        else:
            PrintUtils.printErr(f"Invalid program '{program}'")
        print_available_programs(programs)
        program = input("Run which program?\n> ").strip()
        progNum = tryParseInt(program)
    return QuantumPrograms(engine, QConfig(backend, shots, timeout,
program))
```

```
def setup_experiment(args):
    if args is None:
        return getParams()
    else:
        programs = QuantumPrograms.PROGRAMS

        apiToken = None
        backend = None
        shots = None
        timeout = None
        program = None

        # get API token
        if args.apitoken is None:
            try:
                apiToken = open("./.qiskit_api_token", "r").read()
            except:
                apiToken = input("Enter your IBM Quantum Experience
 API token: \n> ")
        else:
            apiToken = args.apitoken

        engine = QuantumProgram()
        engine.set_api(apiToken, 'https://quantumexperience.ng.bluem
ix.net/api')

        # get backend
        backends = get_backend_dict(engine.available_backends())
        if args.backend is None:
            PrintUtils.printHeader("Available backends:")
            for key, value in backends.items():
                PrintUtils.printInfo(f"  {key}: {value}")
            backend = get_backend(backends[int(input("Run on which b
ackend?\n> "))], backends)
        else:
            if args.backend in backends.values():
                backend = args.backend
            elif int(args.backend) in backends.keys():
                backend = backends[int(args.backend)]
            else:
                raise ValueError(f"Invalid backend: '{args.backen
d}'")

        # set shots default value
        if args.shots is None:
            shots = 1024
        else:
            shots = int(args.shots)
```

```
        # set timeout default value
        if args.timeout is None:
            timeout = 120
        else:
            timeout = int(args.timeout)

        # validate program
        if args.program is None:
            print_available_programs(programs)
            program = input("Run which program?\n> ")
            while program not in programs:
                PrintUtils.printErr(f"Invalid program '{program}'")
                print_available_programs(programs)
                program = input("Run which program?\n> ")
        else:
            if args.program not in programs.keys():
                raise ValueError(f"Invalid program '{args.program}'"
)
            else:
                program = args.program

        return QuantumPrograms(engine, QConfig(backend, shots, timeo
ut, program))

def get_backend_dict(backends):
    dict = {}
    i = 1
    # ensure simulators are at top of list
    for b in backends:
        if "simulator" in b:
            dict[i] = b
            i += 1
    for b in backends:
        if "simulator" not in b:
            dict[i] = b
            i += 1
    return dict

def build_backend_dict(backends):
    dict = {1: "local_qasm_simulator"}
    i = 2
    # ensure simulators are at top of list
    for b in backends:
        if "simulator" in b["name"]:
            dict[i] = b["name"]
            i += 1
    for b in backends:
        if "simulator" not in b["name"]:
```

```
            dict[i] = b["name"]
            i += 1
    return dict

def get_backend(i, available_backends):
    backend = None
    if i in available_backends.keys():
        return available_backends[i]
    elif str(i) in available_backends.keys():
        return available_backends[str(i)]
    elif str(i) in available_backends.values():
        return str(i)
    else:
        PrintUtils.printErr(f"Invalid backend '{backend}', using def
ault simulator...")
        return next(iter(available_backends.values())) # first value
 in dict

def request_input_file():
    files = [i for i in os.listdir(".") if i.endswith(".qasm")]
    PrintUtils.printHeader("QASM files found in current directory: "
)
    for i in files:
        PrintUtils.printInfo(f"    ./{i}")
    file = subprocess.check_output('read -e -p "Enter path to QASM f
ile to run: \n> " var ; echo $var',shell=True).rstrip()
    return file
```

### SignalUtils.py

```
import signal
import PrintUtils

class TimeoutError(Exception):
    pass

def handler(signum, frame):
    raise TimeoutError()

def tryExecuteWithTimeout(func, timeout, failMessage):
    signal.signal(signal.SIGALRM, handler)
    signal.alarm(timeout)
    try:
        func()
    except TimeoutError:
        PrintUtils.printErr(failMessage)
```

## PrintUtils.py

```python
class bcolors:
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    HEADER = BOLD + UNDERLINE + OKGREEN

def printErr(str):
    print(f"{bcolors.FAIL}{str}{bcolors.ENDC}")

def printSuccess(str):
    print(f"{bcolors.OKGREEN}{str}{bcolors.ENDC}")

def printInfo(str):
    print(f"{bcolors.OKBLUE}{str}{bcolors.ENDC}")

def printWarning(str):
    print(f"{bcolors.WARNING}{str}{bcolors.ENDC}")

def printHeader(str):
    print(f"{bcolors.HEADER}{str}{bcolors.ENDC}")

def delete_last_lines(n):
    CURSOR_UP_ONE = '\x1b[1A'
    ERASE_LINE = '\x1b[2K'
    for i in range(0, n):
        print(CURSOR_UP_ONE, end="")
    for i in range(0, n):
        print(ERASE_LINE)
    for i in range(0, n):
        print(CURSOR_UP_ONE, end="")

def toOrdinal(n):
    if n == 1:
        return "1st"
    elif n == 2:
        return "2nd"
    elif n == 3:
        return "3rd"
    else:
        return f"{n}th"
```

## BenchmarkRunner.py

```python
#!/usr/bin/env python3

from QuantumCircuits import QuantumPrograms
from qiskit import QuantumProgram
from QConfig import QConfig
from SignalUtils import tryExecuteWithTimeout
from random import randint
import time
import sys

def setup_quantum_program():
    timeout = 210 # 3.5 minutes
    # timeout = 80 # for debugging
    shots = 1024
    backend = 'local_qasm_simulator'
    program = 'factorize_N'

    engine = QuantumProgram()
    apiToken = None
    try:
        apiToken = open("./.qiskit_api_token", "r").read()
    except:
        apiToken = input("Enter your IBM Quantum Experience API token: \n> ")

    engine.set_api(apiToken, 'https://quantumexperience.ng.bluemix.net/api')
    config = QConfig(backend, shots, timeout, program)
    return QuantumPrograms(engine, config)

def run_benchmark(qp: QuantumPrograms, numberToFactor: int):
    initial = time.perf_counter()
    qp.factorize_N(numberToFactor)
    return (time.perf_counter() - initial)

def random_with_N_digits(n):
    range_start = 10**(n-1)
    range_end = (10**n)-1
    return randint(range_start, range_end)

if __name__ == "__main__":
    console = sys.stdout
    sys.stdout = None # stifle program output
    num_inputs = 10
    results = { 15: [] }

    for i in range(1, num_inputs):
        results[random_with_N_digits(i + 2)] = []
```

```
    num_trials = 10
    # num_trials = 3 # for debugging

    engine = setup_quantum_program()

    for i in results.keys():
        for j in range(0, num_trials):
            def run_experiment():
                res = run_benchmark(engine, i)
                results[i].append(res)
            tryExecuteWithTimeout(run_experiment, engine.qconf.timeout, f"Failed to factorize {i} within {engine.qconf.timeout} seconds.")
            if len(results[i]) <= j:
                results[i].append(-1) # use value of -1 to indicate timeout failure
    sys.stdout = console
    for i in results.keys():
        print(f"N={i}")
        count = 1
        resultSum = 0
        numNonZeroResults = 0
        for j in results[i]:
            if j > 0:
                numNonZeroResults += 1
                resultSum += j
            print(f"    Trial#{count}: {j}")
        results[i] = (resultSum / numNonZeroResults) # average of trials for each number
    try:
        sys.exit()
    except:
        try:
            quit()
        except:
            pass
```

## CsvDataWriter.py

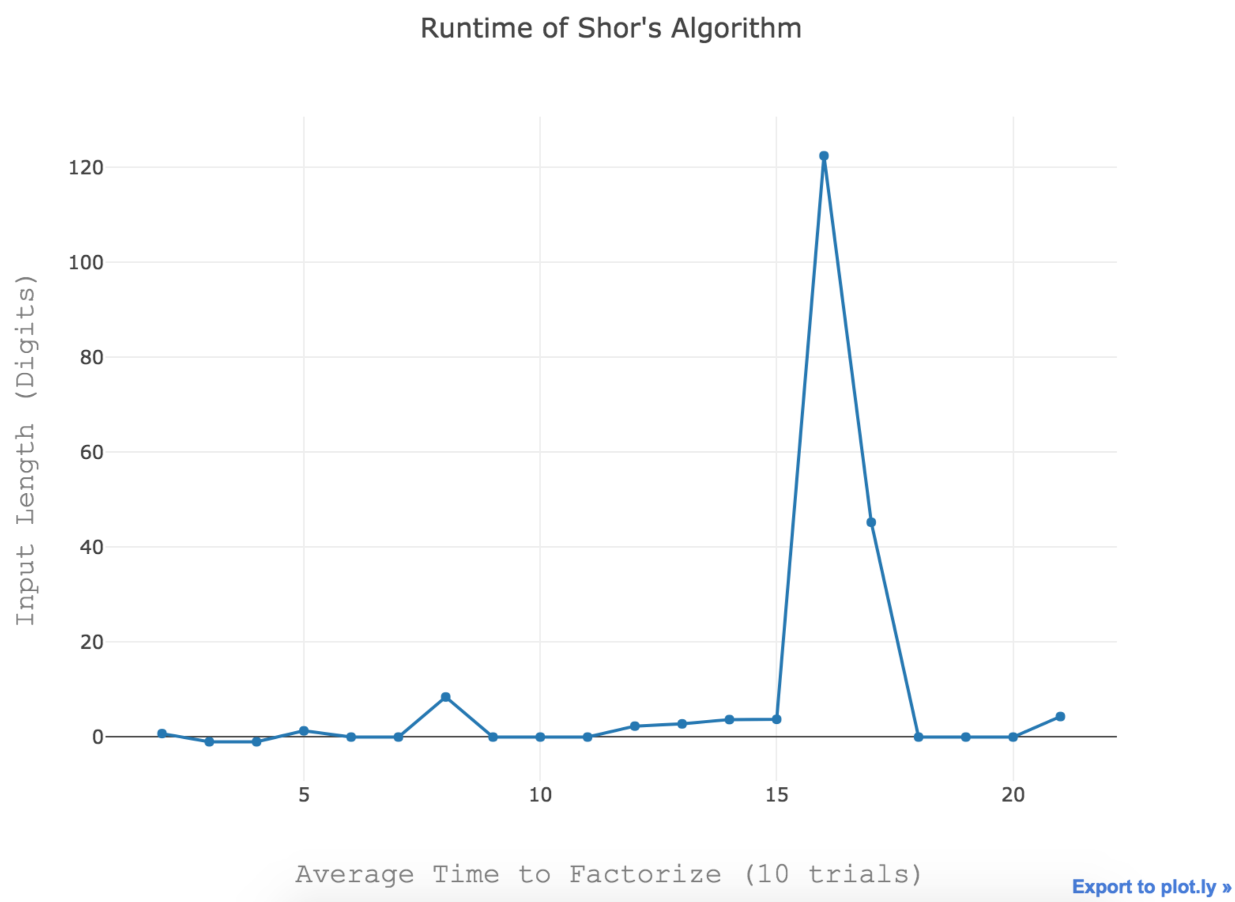
```python
import csv
import time

def get_filename():
    """Returns a filename with a timestamp in it to ensure unique filenames"""
    timestr = time.strftime("%Y-%m-%d--%H-%M-%S")
    return f"benchmark_data/benchmark-{timestr}"

def average(arr):
    """Takes an array of numbers and returns the average."""
    arrSum = 0
    count = 0
    try:
        for i in arr:
            if i > -1:
                arrSum += i
                count += 1
        return (arrSum / count)
    except:
        return -1

def transform_data(data_dict: dict):
    """
    Takes the original data dict and converts it to a dict containing the fieldnames
    and an array of dict in a form that can be used by csv.DictWriter
    (each element of array representing a CSV row), e.g.
    {fieldnames: ["input_len", "trial1", ...], results: [{'input': 15, 'trial1': 4738, 'trial2': 4367, ..., 'average': 47248}]}
    """
    field_names = ["input_len"]
    for i in range(1, len(data_dict[list(data_dict.keys())[0]]) + 5): # length of array of first value
        field_names.append(f"trial{i}")
    field_names.append("average")
    results = []
    for i in data_dict.keys():
        theDict = {"input_len": len(str(i)), "average": average(data_dict[i])}
        trialNum = 1
        for j in data_dict[i]:
            theDict[f"trial{trialNum}"] = j
            trialNum += 1
        results.append(theDict)
    return {"fieldnames": field_names, "results": results}

def write_data(data_dict: dict, filename=None):
```

```
    """
    Takes a dictionary of data of the form {inputNum: [result1, resu
lt2, ...]}
    and outputs the data as a CSV for the form:
    input, trial1, trial2, trial3, ..., average
    """
    data = transform_data(data_dict)
    if filename is None:
        filename = get_filename()
    with open(filename, 'w') as csvfile:
        fieldnames = data["fieldnames"]
        rows = data["results"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(rows)
    return filename

def test():
    print("Running CsvDataWriter test...")
    test_dict = {
        1: [1, 2, 3, 4],
        5: [123, 423, 532, 748],
        15: [647, 616, 679, 686]
    }
    filename = get_filename()
    write_data(test_dict, filename=filename)
    return filename

if __name__ == "__main__":
    test()
```

### BenchmarkPlotter.py and Generated Graph

```
#!/usr/bin/env python3

import plotly.offline as py
from plotly.offline import download_plotlyjs, init_notebook_mode, pl
ot, iplot
import plotly.graph_objs as go
import CsvDataWriter
import pprint
import csv
import glob
import os

pp = pprint.PrettyPrinter(indent=4)

def get_most_recent_data_file():
    list_of_files = glob.glob('./benchmark_data/*') # * means all if
 need specific format then *.csv
    return max(list_of_files, key=os.path.getctime)

def _plot(data):
    """Takes a dict in the form {"x": [x axis data], "y": [y axis da
ta]} and generates a plot."""
    trace = go.Scatter(
        x=data["x"],
        y=data["y"],
        mode='lines+markers',
        name='lines+markers'
    )
    plotData = [trace]
    layout = go.Layout(
        title="Runtime of Shor's Algorithm",
        width=800,
        height=600,
        xaxis=dict(
            title='Average Time to Factorize (10 trials)',
            titlefont=dict(
                family='Courier New, monospace',
                size=18,
                color='#7f7f7f'
            )
        ),
        yaxis=dict(
            title='Input Length (Digits)',
            titlefont=dict(
                family='Courier New, monospace',
                size=18,
                color='#7f7f7f'
            )
        )
```

```
    )
    figure = go.Figure(data=plotData, layout=layout)
    # py.image.save_as(figure, filename="test-plot.png")
    return figure

def make_plot(filename):
    return _plot(parse_data(filename))

def transform_data(data):
    x_axis = []
    y_axis = []
    for i in data:
        x_axis.append(i[0])
        y_axis.append(i[len(i) - 1])

    plotData = {"x": x_axis, "y": y_axis}
    # print(plotData)
    return plotData

def parse_data(filename):
    x_axis = []
    y_axis = []
    with open(filename) as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            x_axis.append(row["input_len"])
            y_axis.append(row["average"])
    data = {"x": x_axis, "y": y_axis}
    return data


if __name__ == "__main__":
    init_notebook_mode(connected=True)
    iplot(make_plot(get_most_recent_data_file()), image_width=800, i
mage_height=600, filename="shors/plot")
```



Runtime of Shor's Algorithm
Input Length (Digits)
0
20
40
60
80
100
120
5
10
15
20
Average Time to Factorize (10 trials)
Export to plot.ly »