Shor's Algorithm in Pyquil

Alec Brickner - abrickne@stanford.edu

June 8, 2019

1 Introduction

For this project, we have implemented Shor's algorithm for factoring [4] in Pyquil. The following will explain the details of our implementation, and how this implementation performs across different sizes of inputs.

2 Implementation

Our implementation of the quantum circuit in Shor's algorithm directly follows the circuit described in [1]. The circuit is composed of the following modules: Quantum Fourier Transform (QFT), QFT addition, modular QFT addition, modular multiplication, and the controlled-U_a gate. For each of these modules, our implementation has functions which construct the corresponding circuits, given the input qubits as parameters.

Upon completion of the circuit, we follow the process given by [2] to get the period r of $y^a \mod N$ from the output c. From this period, we can find a solution to $x^2 = \pm 1 \mod N$. Finally, we compute GCD$(x-1,\ N)$ and GCD$(x+1,\ N)$ - if x is nontrivial, then one of these values will be a factor of X.

2.1 Implementation Differences

One particularly interesting finding was that the implementation as described in [1] did not work as expected. Figure 1 shows the output of this implementation, and how it compares to the expected distribution. We see that instead of r = 10 "spikes", the actual distribution only has 5. Further, these 5 spikes are much lower than those from the expected distribution, except for the spike at 0. As a result, we only get outputs from which we can get a period r about 20% of the time, as opposed to the predicted 80%. Given the other requirements for a valid factorization, such as r being even, GCD$(c',\ r)$ being 1, and the factor being non-trivial, it took a very long time for this implementation to find any non-trivial factors.

We discovered that the solution lay in how the QFT^{-1} was being performed after calculating $y^a \mod N$. According to [1], the one-controlling-qubit method uses the control qubit c as a control for $U_{a^{2^0}}$, takes a measurement, and continues with $U_{a^{2^1}}$. This order is reflected in Figure 1 in [1], wherein $2n$ control qubits are used instead of just one. However, applying the QFT$^{-}1$ operation in this way produces the undesirable results discussed previously. By

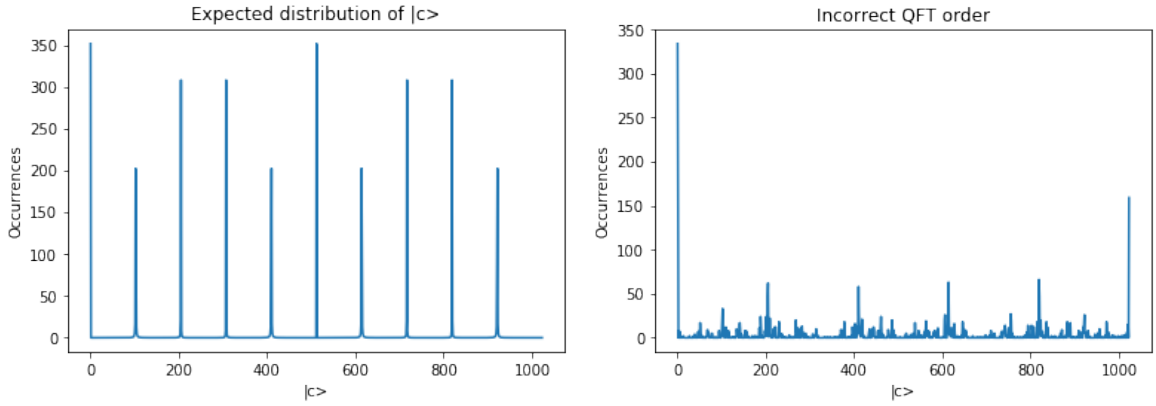


Figure 1: Distributions of $|c\rangle$ from Shor's quantum period-finding algorithm, with N = 25, Q = 1024, y = 4, and r = 10. The left graph shows the expected distribution of —c¿ outputs, while the right graph shows the actual distribution when following the implementation given in [1].

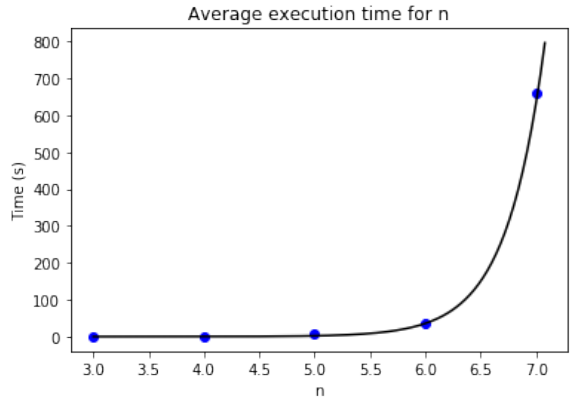
experimenting with the construction of the circuit, we found that while QFT^{-}1 did not work properly, the normal QFT *did* give us the results that we wanted - with the qubits in reverse order. As such, we realized that we needed to perform a normal QFT, rather than QFT^{-1}.

This leads us to ask: since the paper specifies using QFT^{-}1, why does using QFT get us the correct answer? First, we note that QFT starts from the MSB, while QFT^{-1} starts from the LSB. This tells us that if we want the correct answer, we need to run the QFT starting from the MSB. Looking carefully through [2], we see that this is how the QFT is implemented in that paper - similar to the inverse QFT in [1], but starting from the MSB.

As it turns out, this is the only significant difference. The controlled phase shift gate can have its arguments reversed - PS(A, B) = PS(B, A). As such, every phase shift gate in the QFT circuit can have its qubit arguments swapped. Further, note that while using inverse phase shift gates instead of standard phase shift gates makes a difference in the complex coefficient of each state, it does not actually change the magnitude of the probability of that state. For example, we see $(0 - 0.5i)\,|01\rangle$ without the inverse gates, and $(0.5 + 0i)\,|01\rangle$ when using inverse gates - both have a probability of 0.25. This might be an important difference if we need to perform calculations on the output of the QFT, but since the only operation we perform after the final QFT is measurement, the difference in the coefficients has no effect.

Implementing this change required a small change in the one-controlling-qubit algorithm. In our algorithm, we start by precomputing the a^{2^i} values for all i in advance, rather than using repeated squaring. We then compute $y^a \mod N$ starting from $U_{a^{2^{2n-1}}}$, which is possible since U_a gates commute. For each control qubit position i, after we compute $U_{a^{2^i}}$ and the partial QFT for that qubit, we measure the control qubit into position $2n-1-i$, which reads the qubits into the readout in reverse order. Since we found earlier that QFT computed the reverse order of the $|c\rangle$ values that we were expecting, this puts the bits in the correct order.

We are not sure whether this is simply an issue with [1], or whether there is an issue in our implementation. Our assumption is that there is an error in our implementation somewhere, but having checked over our circuit multiple times, we have been able to find nothing different from the circuit in [1]. Of note, however, is how the description of the QFT



n	Time (s)	σ
3	0.24	0.01
4	1.13	0.06
5	6.00	0.24
6	36.32	0.26
7	663.34	6.28

Figure 2: Runtime of the one controlling-qubit period-finding algorithm over different values of n. The specific N values used were $7, 15, 25,$ and 35, respectively.

differs between [1] and [2]; since we have determined that the order in which the QFT runs is important to getting the right answer, this could very well mean that [1] is incorrect.

This does not, however, explain why running the QFT starting from the MSB gives us the expected bytes in reverse order. Perhaps it is a detail left out of [2], or perhaps some other bug in our implementation causes this to occur. A mathematical analysis on why this occurs is left for future investigation.

3 Benchmarking

This section will discuss how well the algorithm performs across different numbers of qubits. Since this is only a simulation on the QVM, we will not get the polynomial time results expected by Shor.

3.1 Runtime of Quantum Period-Finding Algorithm

Here, we will analyze the runtime of the quantum period-finding algorithm across various n, where n is the number of bits in the number N to be factored. See Figure 2 for a graph of the results.

We see a roughly exponential curve here, with each n taking approximately 6 times longer to compute than previous values of n. If we simplify this curve to $t = 6^{n-4}$, we see that factoring a 128-bit N with this algorithm will take $3.1 * 10^{96}$ seconds.

Even so, the one controlling-qubit method performs a lot better than using $2n$ controlling qubits. Due to the fact that the $2n$ qubit method takes a far longer time, we were only able to get results for $n = 3$ and $n = 4$. We find that when $n = 3$, the algorithm takes 2 seconds per run (compared to 0.26), and when $n = 4$, the algorithm takes 222.78 seconds per run (compared to 1.13). Note that this large difference is due to the fact that the $2n$ controlling-qubits method uses $4n + 2$ qubits, while the one controlling-qubit method uses $2n + 3$ qubits. Since the QVM must simulate all of its qubits at the same time, adding more qubits makes the QVM take much longer. On a real quantum computer, we might see fairly

n	N	y	r	Correct	Close	Wrong
4	15	13	4	5	5	0
5	25	4	10	4	6	0
6	35	17	12	2	8	0
7	65	17	12	1	4	0

Table 1: Correctness of Shor's period-finding algorithm (i.e. the period of $f(a) = y^a \mod N$) for various values of N and y. "Correct" means that the r value found was correct, while "Close" means that the r value was off by a factor due to finding a c' value such that $\text{GCD}(c', r) > 1$ held.

similar runtimes for the two methods. In fact, the $2n$ controlling-qubits method may be faster, since we don't have to take measurements during the computation, or use the results of those measurements for future QFT computations.

3.2 Correctness

Here, we will analyze how often the period-finding algorithm produces a correct result. Due to how Shor's algorithm works, the output of the period-finding algorithm may not always be correct, nor will it always lead to a correct period r. If r does not divide q, then the period-finding algorithm will not always return a multiple of $\frac{q}{r}$. Further, even if a correct r is found, this r must be even, and the $\frac{c'}{r}$ approximated by $\frac{c}{q}$ has the restriction that GCD(c',r) = 1. This tends to cut out a lot of potential results, especially if there are many numbers $< r$ which share factors with r. If r is even, this means all the even numbers less than r, plus $\frac{r}{2}$ - at least half. Nevertheless, all we need is a single r and c' that satisfy our constraints.

Figure 1 shows correctness results for small values of N. In the case where $n = 4$, where q equals 256, we see that q is divisible by r, so we do not expect any wrong solutions. However, it is interesting to note that we get no wrong solutions for $n = 5$ or $n = 6$. Using the probability distribution of $|c\rangle$ given in [2], we expect the probability of non-wrong c to be about 0.899 for the instance where $N = 25$ and $y = 4$. This number was computed by calculating the probability of each c value, then summing up the probabilities that were greater than or equal to .01 (which correspond to values of c close to a multiple of $\frac{q}{r}$). If we were to run this for more trials, we might see a distribution of results that more closely aligns with this number.

In the instance where $N = 35$ and $y = 17$, we find that this probability of correctness is also about 0.899. Further, when $N = 65$ and $y = 17$, the probability of correctness becomes 0.903. In neither case in the above table do we see any wrong answers, though again these may come given more trials.

We now look at the "close" and "correct" columns. Given some r, we expect the proportion of correct r to be based on $\Phi(r)$ - that is, the number of integers less than r which are coprime to r. Particularly, we expect $\frac{Correct}{Trials} \approx \frac{\Phi(r)}{r}$ to hold. The results can be seen in Table 2.

In the first two cases, we see that the proportions are identical. In the latter two, they are slightly off ($\frac{1}{3}$ vs $\frac{1}{5}$). Again, with more trials, these proportions may be more accurate.

n	r	$\Phi(r)$	Trials	Correct
4	4	2	10	5
5	10	4	10	4
6	12	4	10	2
7	12	4	5	1

Table 2: Proportions of correct responses to the number of trials, compared to $\Phi(r)$ and r.

3.3 Factoring Correctness

We have yet to speak of factoring - one of the primary goals of this algorithm. Up until now, we have chosen specific values of y that have even values for r, and not considered the fact that certain r values might produce trivial factors. Running the full factorization algorithm for $N = 15$ gets us a correct answer for 12 out of 30 trials, or 40% of the time (rather than the expected 50%). It is not entirely clear why this is the case - when $N = 15$, the quantum function should return a value of c' which is not coprime to r exactly 50% of the time (as r can only be 2 or 4), yet this function seems to consistently work only 40% of the time. Again, this might balance out with more trials.

We continue by factoring 27. This produces far worse results - only 1 out of the 10 trials produces a correct answer, and one gets an even period but finds a trivial solution to $x^2 = 1 \mod N$. This is largely due to the fact that most values of y have a period of either 9 or 18. 9 won't work, since it is odd, and $\Phi(18) = 6$, meaning that only 6 values of c will work.

To see if we can get more success factoring the product to primes, we then try to factor 21. This gives us a little more success - 6 out of 20 trials give a correct factorization. Only two out of the 20 trials produced a trivial factorization - the remaining 12 generally found periods of 3 (or found a c' whose GCD with r is greater than 1.)

4 Conclusions and Future Work

We were able to successfully implement Shor's algorithm using Pyquil, then run it for small values of N. Despite minor confusion regarding the implementation, the program runs as expected. The issue with the reverse readout is interesting, and merits some more looking into.

In the future, we might look into algorithms such as [3], which provide speedups for Shor's algorithm that use pre-computed values from constants to create faster circuits. It may be interesting to see how they perform on Pyquil.

The code for the implementation can be found here.

References

[1] Stephane Beauregard. "Circuit for Shor's algorithm using 2n + 3 qubits". In: *arXiv preprint quant-ph/0205095* (2002).

[2] Artur Ekert and Richard Jozsa. "Quantum computation and Shor's factoring algorithm". In: *Reviews of Modern Physics* 68.3 (1996), p. 733.

[3] Igor L. Markov and Mehdi Saeedi. "Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation". In: *Quantum Information & Computation* 12 (2012), pp. 361–394.

[4] Peter W Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". In: *SIAM review* 41.2 (1999), pp. 303–332.