Quantum Computation **(CMU 18-859BB, Fall 2015)**

Lecture 9: Shor's Algorithm

October 7, 2015

Lecturer: Ryan O'Donnell *Scribe: Sidhanth Mohanty*

1 Overview

Let us recall the period finding problem that was set up as a function $f : \mathbb{Z}_N \to$ colors, with the promise that f was periodic. That is, there exists some s for which $f(x + s) = f(x)$ (note that addition is done in $\mathbb{Z}_N$) for all $x \in \mathbb{Z}_N$ and that colors in a block of size s were pairwise distinct.

This setup implies that $s \mid N$, so that greatly narrows down what s could be. This problem is not hard to do classically, but can be done better with a quantum computer. Slight variants of this problem can be solved with a quantum computer too, and we shall explore such a variant in this lecture.

Here is a sketch of the period finding algorithm that was covered during last lecture (see the period finding lecture for a deeper treatment).

- We begin by preparing our favorite quantum state

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

  We then tensor this state with $|0^n\rangle$.

- We pass the state (after tensoring) through an oracle for f and obtain the state

$$\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle |f(x)\rangle$$

- We then measure the qubits representing $|f(x)\rangle$ and obtain a random color c. This causes the overall state to collapse a superposition of states where $|x\rangle$ is in the preimage of c.

$$\sqrt{\frac{s}{N}} \sum_{k=0}^{\frac{N}{s}-1} |x_0 + ks\rangle |c\rangle$$

  The coefficients can be thought of as $f_c(x)\sqrt{\frac{s}{N}}$ where $f_c(x) = 1$ when $f(x) = c$ and 0 otherwise.

- We then apply the Quantum Fourier Transform on this state to obtain a quantum state where the coefficients are $\hat{f}_c(\gamma)\sqrt{\frac{s}{N}}$ where γ is a multiple of $\frac{N}{s}$. From the previous lecture, we know that $\hat{f}_c$ has a period of $\frac{N}{s}$ and hence γ for which $\hat{f}_c(\gamma)$ is nonzero is a multiple of $\frac{N}{s}$.
- Measuring k gives us a random γ in $\left\{0, \frac{N}{s}, \frac{2N}{s}, \cdots, \frac{(S-1)N}{s}\right\}$.
- Take a constant number of samples and take the GCD of all these samples. With high probability, you get $\frac{N}{s}$, from which we can retrieve s.

2 Review of complexity of algorithms involving numbers

In general, an efficient algorithm dealing with numbers must run in time polynomial in n where n is the number of bits used to represent the number (numbers are of order 2^n)

To refresh, let's go over things we can do in polynomial time with integers.

Say P, Q and R are n bit integers.

- $P \cdot Q$ can be computed in polynomial time.
- $\lfloor \frac{P}{Q} \rfloor$ and $P \mod Q$ can be computed in polynomial time.
- P^Q is massive, and writing it out itself would cause the time to go exponential.
- But $P^Q \mod R$ can be done polynomially by computing $p, p^2, p^4, p^8, \ldots, p^{2^n}$ for $2^n \geq Q$.
- The GCD of P and Q can be done polynomially with Euclid's algorithm.
- Now for something more interesting: checking if P is prime. It can be done in $\tilde{O}(n^2)$ using a randomized algorithm (Miller-Rabin) and in $\tilde{O}(n^6)$ using a deterministic algorithm (AKS).
- Now, why not try to factor P? And suddenly we are stuck if we try to approach the problem classically. The best known deterministic algorithm runs in $2^{\tilde{O}(n^{\frac{1}{3}})}$

3 Shor's Algorithm

There are three steps to understanding Shor's algorithm [Sho97].

1. Factoring $\leq$ Order-finding: Factoring reduces to order-finding, which means that if we have an algorithm to solve order-finding efficiently, we can efficiently solve the factoring problem as well by a polynomial time reduction from factoring to order-finding. Note that this reduction can be made classically.

2. Order-finding $\approx$ Period-finding: Vaguely, order-finding is approximately the same problem as period finding for a quantum computer. This will be expanded in more detail this lecture.

3. Identifying simple fractions: This part is necessary in the order-finding algorithm that is crucial for Shor's algorithm and can be done classically as well.

   The second step is the key step in Shor's algorithm.

3.1 What is order finding?

We are given A, M (n-bit numbers) along with a promise that A and M are coprime. The objective is to find the least $s \geq 1$ ($s \leq M$) such that $A^s \equiv 1 \mod M$. s is called the order of A.

Note that s divides $\varphi(M)$, where φ is the Euler Totient function that gives us the number of elements less than M that are coprime with M. As another remark, $\varphi(M)$ is the order of the multiplicative group $\mathbb{Z}_m^*$ and s divides $\varphi(M)$.

3.2 Proof that Factoring $\leq$ Order-finding

In this section, we shall assume that we have an efficient order-finding algorithm.

Say M is a number that we want to factor. The key to solving the factoring problem using order-finding lies in finding a nontrivial square root of $1 \mod M$, that is, a number r with $r^2 \equiv 1 \mod M$ and $r \not\equiv \pm 1 \mod M$. Then we know that $(r+1)(r-1) \equiv 0 \mod M$ and both $r+1$ and $r-1$ are nonzero $\mod M$ and are factors of some multiple of M. (A nontrivial square root may not always exist, for instance, when M is a power of an odd prime, but we'll see how to handle that case)

Computing the GCD of M and $r-1$ would give us a nontrivial factor of M, called c. We can divide out c from M, check if c or $\frac{M}{c}$ are prime and for each of c and $\frac{M}{c}$, if they are not prime, we recursively factor them, and if they are prime, we store them as prime factors and wait until the rest of the terms are factored. We then return the set of all prime factors. (Recall that we can efficiently test primality.)

Note that the number of recursive calls made is logarithmic in M because there are at most $\log M$ prime factors of M and each recursive call increases how many numbers we have not split by 1. Hence, after $\log M - 1$ recursive calls, there are about $\log M$ numbers that we have not split. Splitting further would force the number of prime factors to exceed $\log M$, which is not possible.

Now, one might ask how one would go about finding a nontrivial square root of $1 \mod M$. We take a random $A \in \mathbb{Z}_M^*$, and find its order s.

Perhaps, we get lucky and have s be even, so we could set $r \equiv A^{\frac{s}{2}} \mod M$ (then $r^2 \equiv A^s \mod M \equiv 1 \mod M$). Maybe we could push our luck a bit more and hope $r \not\equiv -1 \mod M$. But turns out, we can actually make these two lucky things happen, thanks to a number theory lemma!

Lemma 3.1. *Suppose M has ≥ 2 distinct odd prime factors. Then if we pick $A \in \mathbb{Z}_M^*$ uniformly at random, the probability that the order s of A is even and that $A^{\frac{s}{2}} \ncong 1$ is at least $\frac{1}{2}$.*

Proof. See Lemma 9.2 and Lemma 9.3 of Vazirani's course notes [Vaz04] □

One can pick a uniformly random $A \in \mathbb{Z}_M^*$ by randomly picking elements A from $\mathbb{Z}_M$ and computing $\text{GCD}(M, A)$ until it we find A for which the GCD is 1. And with at least $\frac{1}{2}$ chance, our 'lucky conditions' are satisfied. Repeatedly picking A boosts this probability further. If we cannot find such a number A after picking randomly many times, then it means that M is an odd prime power, in which case, we factorize it by binary searching the k-th root of M where k is guaranteed to be an integer in $[1, \log M]$.

3.3 Quantum algorithm for Order-Finding

By establishing that Factoring $\leq$ Order-Finding, we showed that if we could somehow find the order of $A \in \mathbb{Z}_M^*$, we could then classically factorize M.

Now, we shall see how one actually finds the order. Given n bit integers A and M, let $N = 2^{\text{poly}(n)} >> M$ where $\text{poly}(n)$ is something ridiculously large like n^{10}. Such a number N can still be written in $\text{poly}(n)$ bits.

Define $f : \{0, 1, \ldots, N-1\} \to \mathbb{Z}_M$ to be $f(x) = A^x \mod M$. Notice that $A^0 = A^s = 1$, all powers in between are distinct and then it repeats. So it is almost s-periodic, but not quite, because we do not know if s divides N. But we shouldn't have much trouble modifying period-finding slightly to solve this variant of the original problem.

Just like in period-finding, we start with our favorite state

$$\frac{1}{\sqrt{N}} \sum_{x \in \{0,1\}^n} |x\rangle$$

And then we tensor this state with $|0^n\rangle$ and pass the overall quantum state through an oracle for f, O_f and end up with the state

$$\frac{1}{\sqrt{N}} \sum_{x \in \{0,1\}^n} |x\rangle\,|f(x)\rangle$$

And we measure the second register, collapsing the state to a superposition of states that involve $|x\rangle$ where $f(x)$ is a random element c in the subgroup generated by A. This is where order-finding starts getting different from period finding.

Note that s does not divide N, so we cannot be sure of the exact number of times each color c appears. Instead, we can say that appears only D times where D is either $\lfloor \frac{N}{s} \rfloor$ or $\lceil \frac{N}{s} \rceil$. We will now see how taking a massive N comes in handy.

We apply a Quantum Fourier Transform on our state to obtain the state

$$\sqrt{\frac{1}{ND}} \sum_{\gamma=0}^{N-1} \sum_{j=0}^{D-1} \omega^{\gamma \cdot s \cdot j} |\gamma\rangle |c\rangle$$

In the above state, $\omega = e^{\frac{2\pi i}{N}}$. And sampling γ from this state gives us some fixed γ_0 with probability

$$\mathbf{Pr}[\text{sampling } \gamma_0] = \frac{D}{N} \left| \frac{1}{D} \sum_{j=0}^{D-1} \omega^{\gamma_0 \cdot s \cdot j} \right|^2$$

The reason we separate $\frac{1}{D}$ as $\frac{D}{D^2}$ and move the denominator into the square is that it is nice to think of the sum being squared as an average. We want γ we select by sampling to be of the form $\lfloor \frac{kN}{s} \rceil$ (this is notation for nearest integer) for k uniformly distributed in $\{0, 1, \ldots, s-1\}$. The idea is that if γ is of the given form, then $\frac{\gamma}{N}$ is a real number that is extremely close to the simple fraction $\frac{k}{s}$ where it is known that both k and s are n-bit integers. More formally, given $\frac{\gamma}{N}$ within $\pm \frac{1}{2N}$ of $\frac{k}{s}$, we claim we can find $\frac{k}{s}$.

Now, we call upon another lemma to show how such a γ can be sampled.

Lemma 3.2. *For each γ of the form $\lfloor k\frac{N}{s} \rfloor$ with $0 \leq k < s$, there is $\geq \frac{0.4}{s}$ probability of sampling γ.*

Proof. A proof can be found in lemma 9.4 of Vazirani's course notes [Vaz04]. □

We will now show how one can get $\frac{k}{s}$ when they have $\frac{N}{\gamma}$. Continued fractions are a way to approximately describe real numbers in terms of integers. A real number r would look something like

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ldots + \cfrac{1}{a_M}}}}$$

We will use a method involving continued fractions and go over a rough sketch of this method in lecture to use continued fractions to obtain $\frac{k}{s}$ from $\frac{\gamma}{N}$.

First, let us illustrate with an example how one can obtain the expansion of some number with continued fractions.

Consider the fraction $\frac{42}{23}$. We first split the fraction into its integer part and fractional part and express it as the sum of both.

$$\frac{42}{23} = 1 + \frac{19}{23}$$

We then express the fraction as an inversion of its reciprocal.

$$1 + \frac{19}{23} = 1 + \frac{1}{\frac{23}{19}}$$

Now, split the denominator of the second term into its integer part and fractional part and repeat.

$$1 + \frac{1}{1 + \frac{1}{4 + \frac{1}{1 + \frac{1}{3}}}}$$

Now we will see how one could use continued fractions to compute $\frac{k}{s}$. The idea is to use Euclid's algorithm on N and γ and stop when we get some value close to 0 rather than when we get exactly 0, and keep track of quotients of the form $\lfloor \frac{a}{b} \rfloor$ whenever we compute a value of the form $a \mod b$. We will illustrate the method with another example.

If $\frac{k}{s}$ is $\frac{11}{25}$, then $\gamma \approx \frac{11}{25}N$.

Now, we take $N \mod \gamma$ and get approximately $\frac{3}{25}N$ with 2 as the quotient. As the next step, we take $\frac{11}{25}N \mod \frac{3}{25}N$ and get roughly $\frac{2}{25}N$ with 3 as the quotient. Then, we get approximately $\frac{1}{25}N$ as the remainder from $\frac{3}{25}N \mod \frac{2}{25}N$ and get 1 as the quotient. Finally, in the last step, we get the remainder to be approximately 0 and the quotient to be 2 when we take recursively apply Euclidean's algorithm on terms that are approximately $\frac{2}{25}N$ and $\frac{1}{25}N$.

The quotients at any given step in the Euclidean algorithm could be thought of as the integral part, and finding the bigger element modulo the smaller element helps us obtain the fractional part. Using the quotients we obtained, we get the continued fraction approximation for $\frac{\gamma}{N}$ as

$$\frac{1}{2 + \frac{1}{3 + \frac{1}{1 + \frac{1}{2}}}} = \frac{11}{25}$$

To wrap up, we will show how we can eliminate possibilities of failure. If k and s have a common factor, then the fraction $\frac{k'}{s'}$ returned by computing the continued fractions approximation of $\frac{\gamma}{N}$ would be one of simplest form, but with $k' \neq k$ and $s' \neq s$. We will treat this possibility by showing that we can always find $\frac{k}{s}$ with k and s coprime by running the algorithm enough times.

We claim that with probability $\frac{1}{\text{poly}(n)}$, k and s are coprime.

Proof. Note that s has at most $\log s$ prime factors. By the prime number theorem, there are at least $\frac{s}{\log s}$ prime numbers less than s. The order of the number of prime numbers less than s that are coprime with s is about the same, because $\log s$ is asymptotically much less than $\frac{s}{\log s}$ so excluding those primes without losing many elements. Thus, when k is picked uniformly at random between 1 and s, there is a $\frac{1}{\log s}$ chance that it is a prime that is coprime to s. s is at most n bits long, and hence the probability that k is a coprime to s is at least $\frac{1}{\text{poly(n)}}$. □

Repeat the algorithm until you get $\frac{k}{s}$ and $\frac{k'}{s}$ in lowest terms with $\text{GCD}(k, k') = 1$.

Once we accomplish this, we can find s, which is the order of element A. And by using the reduction of factoring to order finding that we proved in the previous section, we can efficiently solve the factoring problem!

References

[Sho97] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.

[Vaz04] Umesh Vazirani. Shor's factoring algorithm. *CS 294-2*, Fall 2004. `http://www.cs.berkeley.edu/~vazirani/f04quantum/notes/lec9.pdf`.