Notes 2.111/8.370/18.435 Fall 2022

Lecture 17 Prof. Peter Shor

On Friday, we covered Simon's algorithm. Simon's algorithm is a quantum algorithm that solves Simon's problem. This isn't a problem that arose from any practical application, but a problem that was invented to show that quantum computers could be faster than classical computers.

What is Simon's problem? We are given a function f that maps length-n binary strings to length-n binary strings. This function has the property that it is 2-to-1, and that there is some binary string c such that

$$f(x) = f(x \oplus c)$$

where $\oplus$ is bitwise exclusive or. For example, such a function might be

$$\begin{array}{ll} f(000) = 101 & f(100) = 011 \\ f(001) = 010 & f(101) = 100 \\ f(010) = 011 & f(110) = 101 \\ f(011) = 100 & f(111) = 010 \end{array}$$

Simon's problem is to find c, which is 110 in the above example.

How can we find c? First, let's look at classical algorithms for the problem. If we find two function inputs x_1 and x_2 so $f(x_1) = f(x_2)$, then $c = x_1 \oplus x_2$. How can we do this? The obvious way is to try inputs at random until we find such a pair.

Let's try an example. By symmetry, it doesn't matter which input we test first, so let's try 000. And let's try 001 for the second input. We've discovered two function values 101 and 010, and because these are different, can conclude that $c \neq 001$.

Now, let's try $f(010)$. This gives another distinct output, and now we can see that f is not

$$\begin{aligned} 000 \oplus 001 &= 001 \\ 000 \oplus 010 &= 010 \\ 001 \oplus 010 &= 011 \end{aligned}$$

Let's try one more input value. if we choose 011, we won't eliminate any possible values of c. but if we try 101, we will eliminate three more values of c,

$$\begin{aligned} 000 \oplus 101 &= 101 \\ 001 \oplus 101 &= 100 \\ 010 \oplus 101 &= 111 \end{aligned}$$

So the most number of values we can eliminate on the jth call to the function is $j - 1$. This means after t function evaluations, we have eliminated at most

$$\sum_{j=1}^{t} j - 1 = \frac{t(t-1)}{2}$$

This says to be sure of finding c, we need $t(t-1)/2 \geqslant 2^n - 1$, or $t \approx 2^{(n+1)/2}$. There is an easier way of seeing this — we will know c as soon as we find $2^{n-1}+1$ different values of $f(x)$, since f is 2-to-1.

Now, let's think about randomized algorithms. Suppose that f is a random function with Simon's property. Before we've found c, the value of c is equally likely to be any of the values that we have not yet eliminated. Thus, for us to have a fifty percent chance of finding c, we need eliminate roughly half the possible values of c, which gives $t \approx \frac{2}{3}2^{n-1}$. So the running time of the best classical algorithm for this problem is around $\frac{1}{3}2^n$.

What about a quantum algorithm for this problem? Before we can give the quantum algorithm, we need to say how the function is given to the quantum computer. We will use an oracle O_f that behaves as

$$O_f \left|x\right\rangle \left|z\right\rangle = \left|x\right\rangle \left|z \oplus f(x)\right\rangle .$$

If $z = 0$, then the function computes $f(x)$ in the second register. We arrange the oracle this way so that it will be reversible, because functions on quantum computers have to be reversible.

Also recall that the Hadamard transform $H^{\otimes n}$ takes a binary string $\left|j\right\rangle$ of length n to a superposition of all binary strings of length n:

$$H^{\otimes n} \left|j\right\rangle = \sum_{k=0}^{2^n-1} (-1)^{j\cdot k} \left|k\right\rangle$$

We can now give Simon's algorithm. We start with two registers of n qubits each, initialized in the state $\left|0^n\right\rangle$:

$$\left|0^n\right\rangle \left|0^n\right\rangle .$$

We now apply a Hadamard transform to the first register:

$$\frac{1}{2^{n/2}} \sum_{j=0}^{2^n-1} \left|j\right\rangle \left|0^n\right\rangle ,$$

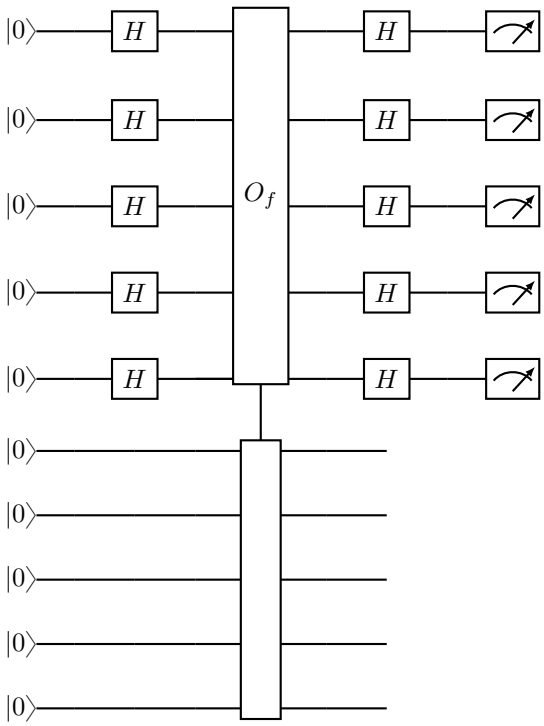
where $\left|j\right\rangle$ is the representation of the integer j in binary. We next apply the function O_f:

$$\frac{1}{2^{n/2}} \sum_{j=0}^{2^n-1} \left|j\right\rangle \left|f(j)\right\rangle .$$

Now, we apply another. Hadamard transform to the first register:

$$\frac{1}{2^n} \sum_{j-0}^{2^n-1} \left(\sum_{k=0}^{2^n-1} (-1)^{j\cdot k} \left|k\right\rangle \right) \left|f(j)\right\rangle = \frac{1}{2^n} \sum_{k=0}^{2^n-1} \left(\sum_{j=0}^{2^n-1} (-1)^{j\cdot k} \right) \left|k\right\rangle \left|f(j)\right\rangle$$

And finally, we measure the registers (although we will only use the result from the first register).



To analyse the algorithm, we need to compute the probability of seeing a pair of values $|k\rangle|f(j)\rangle$ in the registers. This will be equal to the square of the amplitude on this state. First, note that there are only two values of j that will produce $f(j)$; these are j and $j \oplus c$. So there are only two terms in the sum over j that contribute to this amplitude, and the amplitude is:

$$\frac{1}{2^n}\left((-1)^{j\cdot k} + (-1)^{(j\oplus c)\cdot k}\right).$$

We now use the distributive law $(r \oplus s)\cdot t = r\cdot t \oplus s\cdot t$, and we get

$$\begin{aligned}\frac{1}{2^n}\left((-1)^{j\cdot k} + (-1)^{(j\oplus c)\cdot k}\right) &= \frac{1}{2^n}\left((-1)^{j\cdot k} + (-1)^{j\cdot k \oplus c\cdot k}\right)\\ &= \frac{1}{2^n}\left((-1)^{j\cdot k} + (-1)^{j\cdot k}(-1)^{c\cdot k}\right)\\ &= \frac{1}{2^n}(-1)^{j\cdot k}\left(1 + (-1)^{c\cdot k}\right)\end{aligned}$$

But if $c\cdot k = 1$, this expression is 0, and if $c\cdot k = 0$, this expression is $\pm\frac{2}{2^n}$. We thus have that the probability of seeing a value k is 0 if $c\cdot k = 0$, that is, if k is not perpendicular to c, and $\frac{1}{2^{2n-2}}$ if k is perpendicular to c.

This lets us use linear algebra over $\mathbb{Z}_2$ to find c. Suppose we repeat the process above to find $n-1$ linearly independent vectors perpendicular to c. If you have $n-1$

vectors in an n-dimensional space, there is only one vector perpendicular to them. This is c; all we need to do is find it.

Linear algebra over a finite field (like $\mathbb{Z}_2$) is different in several ways from linear algebra over $\mathbb{R}$ or $\mathbb{C}$. The most significant difference is that a vector can be perpendicular to itself; for example, $1010011 \cdot 1010011 = 4 = 0 \pmod 2$. However, many of the techniques and theorems of linear algebra over $\mathbb{Z}_2$ are the same as those over $\mathbb{C}$. Rather than explaining the differences in detail, I'll just demonstrate how we find c for an example. It should be straightforward to deduce the general algorithm from the example, and you should be able to see why it always works.

Suppose we have some vectors perpendicular to c. First, we need to check that they are all linearly independent (or eliminate the linearly dependent ones, if there are any). We use Gaussian elimination. Let's say we have the vectors

$$\begin{aligned} k_1 &= 11011 \\ k_2 &= 01011 \\ k_3 &= 01111 \\ k_4 &= 11010\,. \end{aligned}$$

Our first step will be to make the first column all 0's, except for the top row. We can do this by subtracting (i.e., XORing) the first row with any row that begins with 1. This gives $k_4' = k_4 \oplus k_1$, and we have:

$$\begin{aligned} k_1 &= 11011 \\ k_2 &= 01011 \\ k_3 &= 01111 \\ k_4' &= 00001\,. \end{aligned}$$

Now, we do the same for the second column, for all rows below the 2nd row. This gives $k_3' = k_3 \oplus k_2$, and we have:

$$\begin{aligned} k_1 &= 11011 \\ k_2 &= 01011 \\ k_3' &= 00100 \\ k_4' &= 00001\,. \end{aligned}$$

Now we're done, because this matrix is in row-echelon form, and we see that the original four vectors were linearly independent. However, in general we might have to use this procedure on all the columns.

What we do now is figure out which vector is perpendicular to all these rows. Looking at the last row, k_4', we see that the last coordinate of c must be 0, so $c = ???0$. We don't have any rows of our matrix that start $0001?$, which means that the fourth coordinate isn't constrained—it could be either 0 or 1. if we take it to be 0, we will end up with the 0 vector, so we take it to be 1, and we have $c = ??10$. Now, the fact that c must be perpendicular to k_3' means that the third coordinate is 0, so we get $c = ?010$. Making c perpendicular to the second row, k_2, gives $c = 1010$, and making

it perpendicular to the first row gives $c = 01010$. And indeed, you can check that $c = 01010$ is perpendicular to every row in our first matrix.

Note that in general, if you have $n-1$ linearly independent vectors perpendicular to c, and you carry out this procedure, all but one of the coordinates of c will be determined by a row in the row-echelon matrix, and this procedure will give you a unique non-zero vector c.

How long does this algorithm take? That is, how many vectors will we have to sample to find $n-1$ vectors that are linearly independent. For the first step, there is only one vector that is linearly dependent ($000\ldots0$), and there are 2^{n-1} possible vectors perpendicular to c. Our calculations showed that we find each of them with equal probability, so the probability of the first vector not being linearly independent is $\frac{1}{2^{n-1}}$. On the second step, there are two vectors that are not linearly independent (k_1 and 0), so the probability of the second vector not being linearly independent is $\frac{1}{2^{n-2}}$. And in general, the probability that the ith vector is linearly dependent on the first $i-1$ is $\frac{1}{2^{n-i}}$. All of these probabilities are less than $\frac{1}{2}$, so the expected number of tries it takes to get a linearly independent vector on the ith step is at most $\frac{1}{2}$, and we see that the expected number of steps is less than $2n$. Since each step takes $O(n)$ time, the total time taken is $O(n^2)$. This contrasts with the expected running time of at least $2^{n/2}$ for the classical algorithm, giving an exponential advantage for quantum computation.

In fact, with a more clever analysis you can show that the expected number of steps is always less than $n+2$.

Now, let's look at the circuit for Simon's algorithm more closely. You will notice that we first do a Hadamard transform on the first register. We then computer take the value of $|x\rangle$ in the first register and compute $|f(x)\rangle$ in the second register. However, we never look at the second register again. I don't know hwo many of you know about optimizing compilers, but a classical optimizing compiler would look at this circuit, and ask: why are you computing $f(x)$ in the second register if you never look at the value again? (And in fact, this was one of the questions that was asked in class on Friday.) So why is the computation of the oracle necessary for the algorithm.

It is easy to see that it is necessary; without the oracle, you would just be applying an H gate to each wire in the first register, and then applying another H gate to these wires. Since $H^2 = I$, we would have a circuit that did nothing. But how can computing the value of $|f(x)\rangle$ in the second register change the values of the first register. This is an example of a process called *back action*.

Back action is a fundamental principal of quantum mechanics. It says that any time two systems interact, if the first system has an effect on the second, then the second also has an effect on the first. Maybe for this course, the best illustration of back action is the CNOT gate.

Recall that the Hadamard gate has the following properties:

$$\begin{aligned} H^2 &= I \\ H\sigma_z H &= \sigma_x \\ H\sigma_x H &= \sigma_z \end{aligned}$$

Now, we will deal with an identity involving CNOT and Hadamard gates. First,

let's consider the CNOT gate, and input the state $|-\rangle|-\rangle$. What happens?

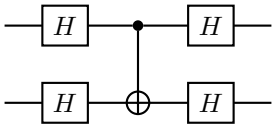
$$\begin{aligned}|-\rangle|-\rangle &= \frac{1}{2}\big(|0\rangle - |1\rangle\big)\big(|0\rangle - |1\rangle\big) \\ &= \frac{1}{2}\big(|00\rangle - |01\rangle - |10\rangle + |11\rangle\big)\end{aligned}$$

So because CNOT$|10\rangle = |11\rangle$ and CNOT$|11\rangle = |10\rangle$, we get

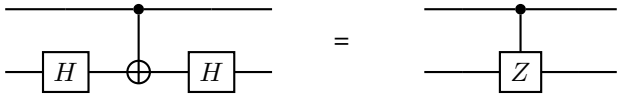
$$\begin{aligned}CNOT\frac{1}{2}\big(|00\rangle - |01\rangle - |10\rangle + |11\rangle\big) &= \frac{1}{2}\big(|00\rangle - |01\rangle - |11\rangle + |10\rangle\big) \\ &= \frac{1}{2}\big(|0\rangle + |1\rangle\big)\big(|0\rangle - |1\rangle\big) = |+\rangle|-\rangle.\end{aligned}$$

Here, the target qubit has stayed the same and the control bit has changed. This is the opposite of what you might think should happen. Classically, when you apply an "if" statement on variable A, and use the results to modify variable B, then variable A doesn't change. Here, a CNOT looks like an "if" statement, but the value of variable A has changed. We'll say a little more about this phenomenon at the end of these lecture notes.

Let's look at the quantum circuit

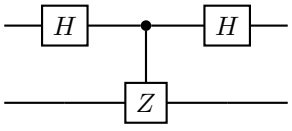


What does this do? First, let's look at a piece of this circuit: We will show that



If the top wire starts in $|0\rangle$, the CNOT acts as identity on the bottom wire, and we get $HH = I$. If the top wire states in $|0\rangle$, the CNOT acts as a σ_x on the bottom wire, and we get $H\sigma_x H = \sigma_z$. Thus, the above circuit is a controlled σ_z, or a C-Z.

We can make this substitution in our original circuit, to get the equivalent circuit

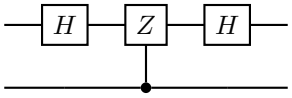


However, a C-Z with the first qubit as the control and the second qubit as the target is the same as a C-Z with the second qubit as the control and the first qubit as the target. You can see this by noticing that the action on the two qubits is symmetric—a phase of -1 is applied if and only if the state is $|11\rangle$. You can also see this by matrix

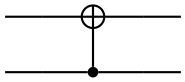
multiplication. You can compute the action of the gate with the qubits swapped using SWAP · C-Z · SWAP = C-Z where

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \text{C-Z} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

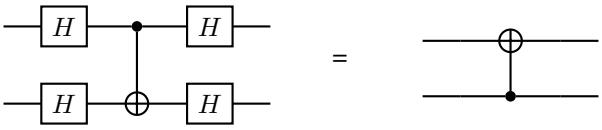
Thus, the circuit is equivalent to



But we've just computed that this is the same as:



So now we've shown that putting two H gate before and after a CNOT reverses the direction of the CNOT:



This is actually an example of a general principle in quantum mechanics: if system A has some effect on system B, then system B will also have an effect on system A. The effect of system B on system A is called "back-action". One consequence of this is that you cannot measure a quantum system without also affecting the quantum system being measured; in a standard von Neumann measurement, the system being measured is projected onto one of a set of quantum states, or a set of subspaces. However, there are more general kinds of measurement, that we won't discuss in this class, and the principle that there will always be a back-action still applies.