Lecture 7: Arithmetic/number-theoretic problems; reversible computation

February 9, 2006

Arithmetic/number-theoretic problems

Before we move on to Shor's Algorithm, we need to understand more about the topic with which it is concerned: computational number theory.

Let us begin with an example—suppose we are given two integers x and y, represented as strings of bits using binary notation[1], and our goal is to determine $x + y$. A specification of this problem is as follows:

Integer Addition

Input: $x, y \in \mathbb{Z}$.

Output: $x + y$.

One can formalize the notion of an *efficient* algorithm for a given problem in various ways by describing precise models of computation. For the purposes of this discussion let us just think in terms of Boolean circuits and identify an algorithm with a collection of acyclic circuits (one for each possible input length) that produce the correct outputs for all possible input strings. The circuits should be composed of AND, OR, and NOT gates, along with FANOUT operations (which we often would not consider as gates at all), and in order to be considered efficient the total number of gates in each circuit should be *polynomial* in the number of input bits.

The number of bits needed to write down the number x in binary notation is

$$\lg(x) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } x = 0 \\ \lfloor \log_2 |x| \rfloor + 2 & \text{if } x \neq 0, \end{cases}$$

assuming we keep a sign bit for each non-zero integer, and so the number of input bits of the above problem is $\lg(x) + \lg(y)$. Now, if you consider the usual method for adding numbers that you probably learned in elementary school or before, but using base 2 instead of base 10, you could turn that method into a construction of Boolean circuits of size $O(\lg x + \lg y)$ for solving this problem. This is in fact *linear* in the input size, and so this algorithm will be considered to be very efficient.

By the way, I should mention that it is typical to place additional constraints on collections of Boolean circuits in order for the circuits to be identified with efficient algorithms. In particular, the circuits should satisfy certain *uniformity constraints*, which means that not only do they have polynomial size, but moreover there should exist an efficient construction of the circuits themselves. This issue can safely be ignored in this class, however.

[1] We will take it as implicit from now on that integers are represented in binary notation in computational problems.

Summing up, we would simply say that the Boolean circuit complexity, or "bit complexity" of computing $x + y$ for integers x and y is $O(\lg x + \lg y)$. Or, even more simply, the bit complexity of adding two n bit integers is $O(n)$.

Here are some other arithmetic and number theoretic problems:

Integer Multiplication

Input: $x, y \in \mathbb{Z}$.

Output: xy.

The "elementary school" multiplication algorithm establishes that the bit complexity of multiplying x and y is $O((\lg x)(\lg y))$, or more simply that the bit complexity of multiplying two n bit numbers is $O(n^2)$. In fact, there are asymptotically better methods, such as the Schönhage-Strassen Algorithm that has bit complexity $O(n \log n \log \log n)$. (Large constant factors in the running time make this particular method less efficient than other methods until n becomes quite large—several thousand bits perhaps.)

Integer Division

Input: Integers x and $y \neq 0$.

Output: Integers a and b, with $0 \leq b < |y|$ such that $x = ay + b$.

The cost is roughly the same as integer multiplication—the "elementary school" division method gives an algorithm with bit complexity $O((\lg x)(\lg y))$. (In fact it is actually somewhat better: $O((\lg a)(\lg y))$.)

Greatest Common Divisor

Input: Nonnegative integers x and y.

Output: $\gcd(x, y)$.

Euclid's Algorithm, which is over 2,000 years old, computes $\gcd(x, y)$ in $O((\lg x)(\lg y))$ bit operations.

Modular Integer Addition

Input: A positive integer N and integers $x, y \in \mathbb{Z}_N \overset{\text{def}}{=} \{0, \ldots, N-1\}$.

Output: $x + y \pmod{N}$.

The bit complexity of this problem is $O(\lg N)$.

Modular Integer Multiplication

Input: A positive integer N and integers $x, y \in \mathbb{Z}_N$.

Output: $xy \pmod{N}$.

Here the bit complexity is $O((\lg N)^2)$ by the elementary school algorithm, and again asymptotically faster methods exist.

Modular Inverse

Input: A positive integer N and an integer $x \in \mathbb{Z}_N^* \overset{\text{def}}{=} \{a \in \mathbb{Z}_N \,:\, \gcd(a, N) = 1\}$.
Output: $y \in \mathbb{Z}_N$ such that $xy \equiv 1 \pmod{N}$.

The bit complexity is $O((\lg N)^2)$.

Modular Exponentiation

Input: A positive integer N, an integer $x \in \{0, \ldots, N-1\}$, and any integer k.
Output: $x^k \pmod{N}$.

Using the method of repeated squaring, the bit complexity is $O((\lg k)(\lg N)^2)$.

Primality Testing

Input: A positive integer N.
Output: "prime" if N is a prime number, "not prime" otherwise.

Randomized algorithms having bit complexity $O((\lg N)^3)$ that allow a probability $1/4$, say, of making an error have been known for many years. In a breakthrough a few years ago, Agrawal, Kayal and Saxena gave a deterministic prime testing algorithm that gives (the last I heard) a $O((\lg N)^6)$ deterministic algorithm for testing primality.

Integer Factoring

Input: A positive integer N.
Output: a prime factorization $N = p_1^{a_1} \cdots p_k^{a_k}$ of N.

No classical algorithm is known to give a polynomial bit complexity for factoring—the best known algorithm asymptotically gives a bit complexity of

$$2^{O\left((\log N)^{1/3}(\log\log N)^{2/3}\right)}.$$

We will see that when we turn to quantum algorithms, Shor's algorithm will solve the Integer Factoring problem using $O((\lg N)^3)$ operations, giving a remarkable speed-up over known classical algorithms for this task.

Reversible computation

In order to implement Shor's algorithm, it is necessary to efficiently perform some of the arithmetic operations described above (modular exponentiation, in particular) with a quantum computer. Intuitively this seems like it should not be a problem, given that there is an efficient way to implement these operations classically. But does an efficient classical implementation necessarily imply the existence of an efficient quantum implementation? The fact that classical algorithms can use non-unitary operations such as ANDs and ORs, while quantum computers are restricted to unitary operations, makes this question non-trivial to answer.

Before we address the question of whether efficient classical algorithms for arithmetic problems (or any other type of computational problem) imply efficient quantum algorithms for these problems, we need to briefly discuss the notion of quantum circuit complexity. What does it mean to have an efficient quantum algorithm for some task?

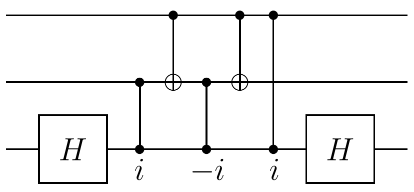
Our answer to this question will be similar to the classical case. In order to completely specify a quantum algorithm, it is necessary to give a construction that produces a quantum circuit solving the problem at hand for any given input size. Also, as in the classical case, one assumes the quantum circuit is composed entirely of quantum gates acting on some small number of qubits—in analogy to the classical case, one may require that all gates in the quantum circuit act on just one or two qubits. Again, to be considered efficient, one usually requires that the number of gates is polynomially related to the number of input qubits.

There is, however, a crucial difference between the classical and quantum cases. Whereas there are a finite number of classical one and two bit operations, and it is well known that AND and NOT operations (for instance) are sufficient to generate any such operation, there are infinitely many one and two qubit unitary operations. Although there are interesting technical aspects of this issue, such as the degree to which a finite number of quantum gates can approximate arbitrary gates, we will essentially sweep the issue under the rug by saying that arbitrary one and two qubit gates are allowed in our circuits.

By the way, it will be very helpful to use Toffoli gates to construct quantum circuits for various tasks. Recall that a Toffoli gate performs the transformation

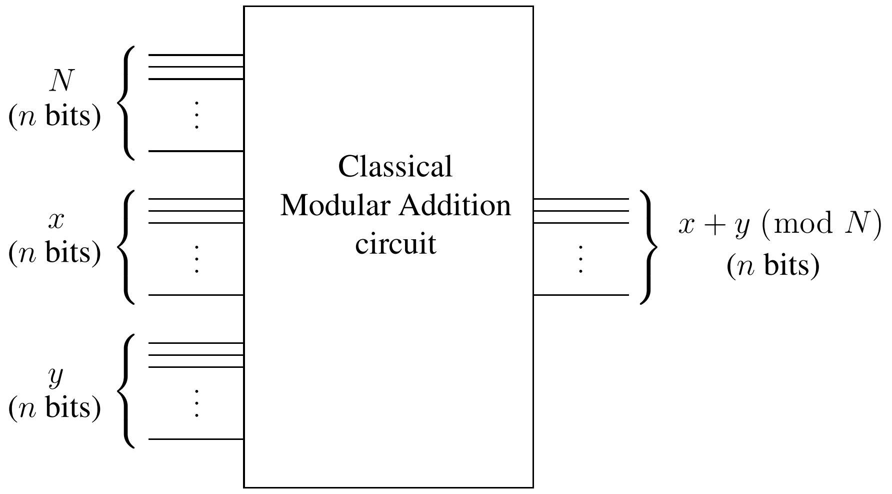
$$T\,|a\rangle\,|b\rangle\,|c\rangle = |a\rangle\,|b\rangle\,|c \oplus (a \wedge b)\rangle$$

for $a, b, c \in \{0, 1\}$. A Toffoli gate can be decomposed into one- and two-qubit quantum gates as follows:



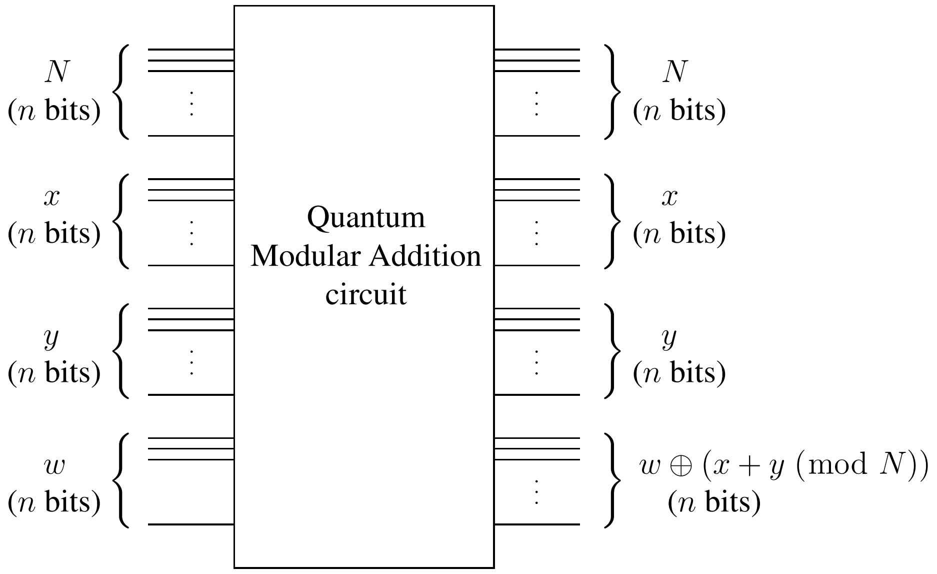
A careful check of the 8 standard basis states shows that this circuit implements a Toffoli gate as claimed.

Now let us return to the question of whether an arbitrary efficient classical algorithm can be converted to an efficient quantum algorithm. Suppose that we have some classical Boolean circuit. For the sake of example, suppose it performs modular addition as suggested in the following diagram.


N (n bits)
x (n bits)
y (n bits)
Classical Modular Addition circuit
$x + y \pmod{N}$ (n bits)

Obviously such a circuit is non-unitary—the number of input and output qubits do not even agree. If we are to somehow transform this circuit into a valid quantum circuit, it will have to correspond to a unitary transformation.

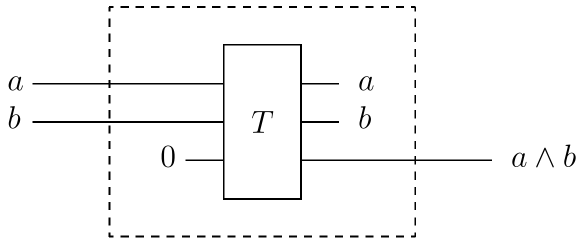
Based on our previous discussions concerning black-box transformations, we might hope to implement this operation unitarily as follows:



This is a special type of unitary transformation, because it always maps classical states to classical states, therefore giving rise to a permutation matrix. We will show how to implement such transformations using only gates that map classical states to classical states, so none of what we are discussing right now needs to be thought of as inherently quantum. Usually when we are talking about quantum gates and circuits that always map classical states to classical states, we refer to them as *reversible* gates or circuits. In other words, a reversible gate is a special type of quantum gate that maps classical states to classical states, or equivalently gives rise to a permutation matrix.

Now let us see how we can transform the original classical circuit into such a circuit, obeying the constraint that only reversible gates are used inside the circuit. We do not want to have to reinvent the wheel, so to speak, so we can begin by trying to simply replace the gates of the classical circuit with reversible gates. The Toffoli gates will be handy for doing this.

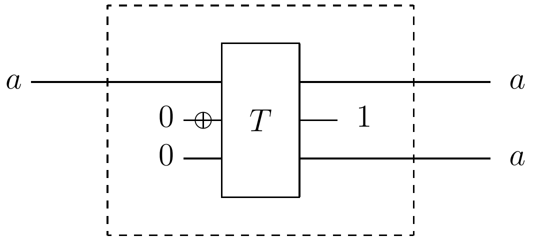
AND gates. We can simulate an AND gate using a Toffoli gate as follows:



NOT gates. These are already unitary, so nothing needs to be done.

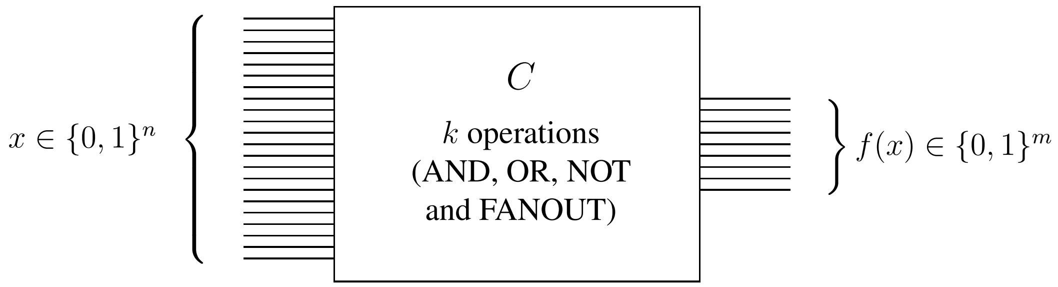
OR gates. These can be replaced by an AND gate and three NOT gates using DeMorgan's Laws.

FANOUT. We often do not even think of this as an operation, but we could have fanout in a classical Boolean circuit, and this needs to be explicitly implemented in a quantum circuit. Note that this does not mean replicating the quantum state of a qubit—it means replicating the classical state. Here is how this operation can be simulated using a Toffoli gate:

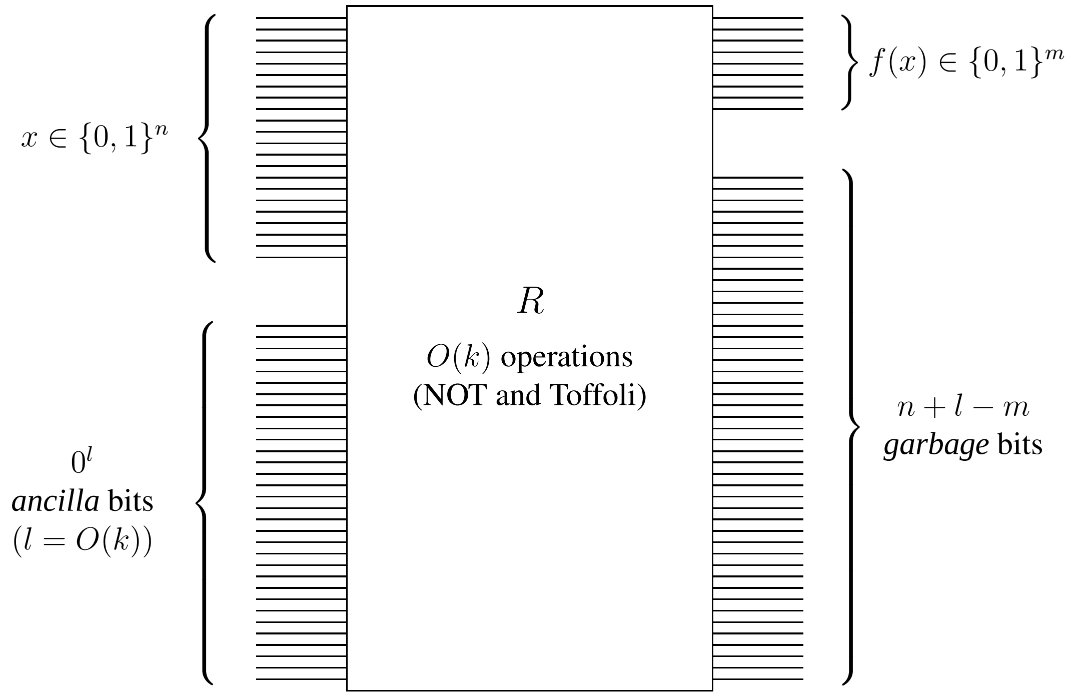


Of course it would have been just as easy to input a 1 into the second input of the Toffoli gate rather than performing a NOT operation on a 0. I've only written it like this because it will be convenient later to have all of the auxiliary inputs start in the 0 state.

Now, we can simply go through the original circuit and replace each of the AND, OR, NOT, and FANOUT operations with reversible gates as just described. If we started with a general circuit C computing a function $f : \{0,1\}^n \to \{0,1\}^m$ as follows:



Then we would end up with a reversible circuit R as follows.



The auxiliary bits needed for the AND and FANOUT simulations in this circuit are sometimes called *ancilla* bits. The bits labeled *garbage* are the extra bits that resulted from the simulations (that were left inside the dashed boxes in the above pictures).

The circuit above is not quite what we want, because the garbage bits are undesirable—we have no control over them, and they would cause problems in many quantum computations. However, we almost have what we are after. By taking advantage of the fact that NOT gates and Toffoli gates are their own inverses, we can simply take a "mirror image" of R to get R^{-1}, which simply undoes the computation of R. Combining R, R^{-1}, and a collection of controlled-NOT gates as described in Figure 1 essentially gives us what we were looking for. The only difference between this circuit and our initial aim is the existence of the ancilla bits. (It should be noted that commonly when people use the term "ancilla", they often expect that such bits are returned to their initial 0 state as in the circuit in Figure 1.) These bits will not have any adverse effects, and often we do not even explicitly mention them.

So, in summary, if you have a classical Boolean circuit C for computing some function

$$f : \{0,1\}^n \to \{0,1\}^m,$$

then the procedure above will transform a description of this circuit into a description of a quantum circuit S_C composed only of Toffoli gates, NOT gates, and controlled-NOT gates that satisfies

$$S_C \left|x\right\rangle \left|y\right\rangle \left|0^l\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle \left|0^l\right\rangle .$$

The number of ancilla qubits l and the total number of gates in S_C is linear in the number of operations in the original circuit C. The presence of the ancilla qubits will have no effect on

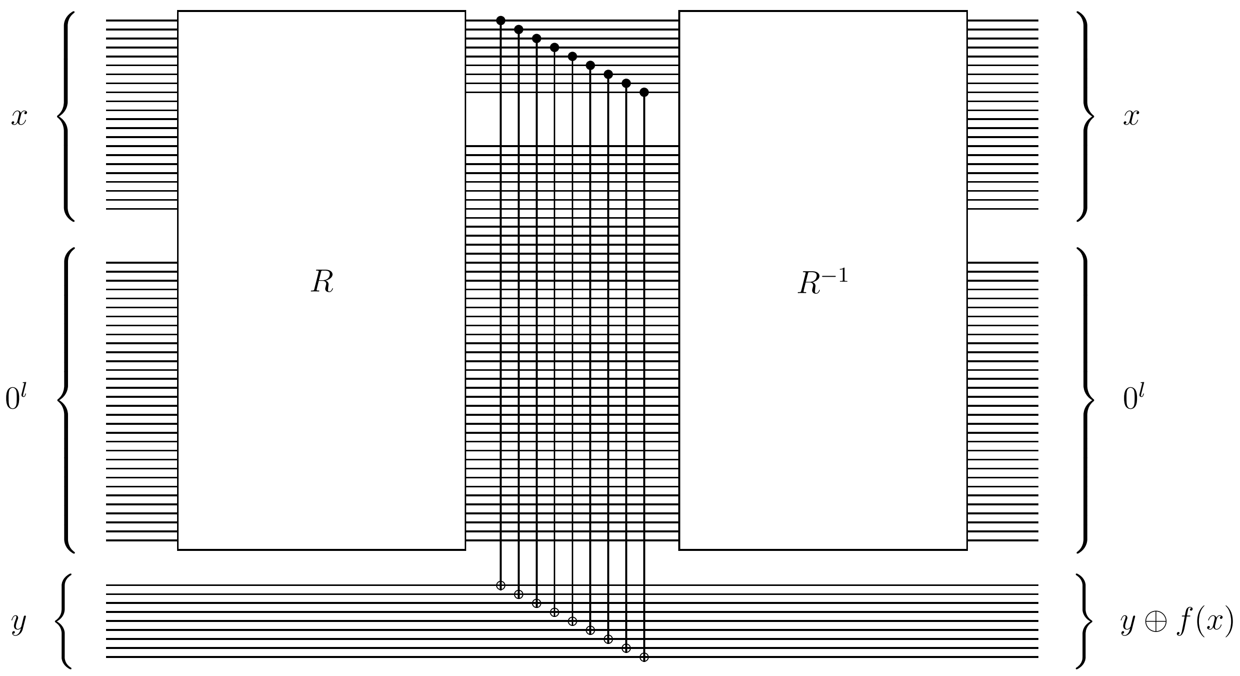


Figure 1: The final reversible circuit S_C implementing the transformation we are after.

any algorithm that uses the circuit S_C, and in fact they could be re-used to perform some other transformation after S_C is performed. For this reason, we will usually say that S_C performs the transformation

$$S_C \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle,$$

leaving it implicit that some number of ancilla qubits may have been used to implement S_C.

Reversible implementation of invertible functions

Finally, suppose that you have a function of the form

$$f : \{0,1\}^n \rightarrow \{0,1\}^n$$

that itself is invertible (one-to-one and onto), and suppose that f is efficiently computed by some Boolean circuit C. We already know that it is possible to build a reversible circuit S_C that performs the transformation

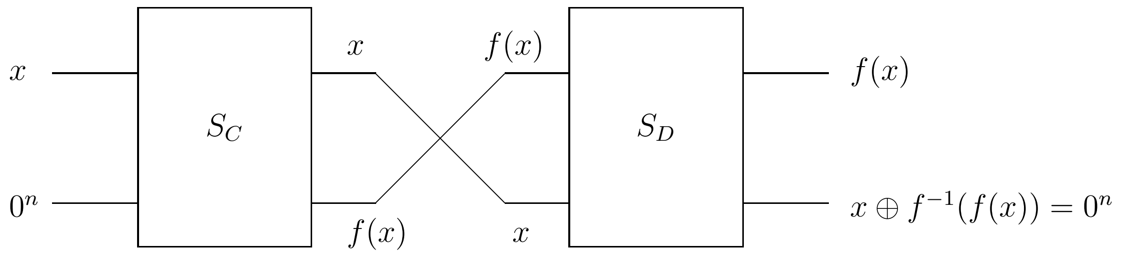
$$S_C \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle$$

for all $x, y \in \{0,1\}^n$ (possibly using some number of ancilla bits). Is it possible to go further and build a reversible circuit Q, possibly using some ancilla bits, that efficiently performs the transformation

$$Q \left|x\right\rangle = \left|f(x)\right\rangle$$

for all $x \in \{0,1\}^n$? Because f is assumed to be invertible, the transformation to be performed by Q is indeed reversible, so in principle it is possible to build Q. The question, however, is about efficiency.

Without any further assumptions it is not known how to construct Q given only C. However, if in addition we have a Boolean circuit D that efficiently computes f^{-1}, then Q can be constructed that efficiently performs the required transformation as follows:



(In order to simplify the picture, each "wire" represents n bits and the ancilla bits needed by the circuits S_C and S_D have not been shown explicitly.)