Quantum Information Processing
Quantum Algorithms (I)

Richard Cleve
Institute for Quantum Computing & Cheriton School of Computer Science
University of Waterloo

September 22, 2021

Abstract

The goal of these notes is to explain the basics of quantum information processing, with intuition and technical definitions, in a manner that is accessible to anyone with a solid understanding of linear algebra and probability theory.

These are lecture notes for the second part of a course entitled "Quantum Information Processing" (with numberings QIC 710, CS 768, PHYS 767, CO 681, AM 871, PM 871 at the University of Waterloo). The other parts of the course are: a primer for beginners, quantum information theory, and quantum cryptography. The course web site http://cleve.iqc.uwaterloo.ca/qic710 contains other course materials, including video lectures.

I welcome feedback about errors or any other comments. This can be sent to cleve@uwaterloo.ca (with "Lecture notes" in subject heading, if at all possible).

Contents

1 Classical and quantum algorithms as circuits

In this section, we'll see see a basic picture of classical and quantum algorithms as circuits. We'll consider simulations between classical and quantum circuits and we'll see the Toffoli gate.

1.1 Classical logic gates

Recall that the NOT gate takes one bit as input and outputs the logical negation of the bit.

a	$\neg a$
0	1
1	0

Figure 1: Table of input/output values of the NOT gate.

Here is the traditional notation for the gate that's used in electrical engineering logic diagrams, followed by more modern ways of denoting the gate.

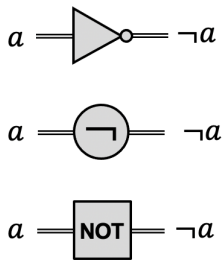

Figure 2: Three notations for the NOT gate.

The logical AND gate takes two input bits and produces one output bit, which is 1 if and only if both input bits are 1.

ab	$a \wedge b$
00	0
01	0
10	0
11	1

Figure 3: Table of input/output values of the AND gate.

Here is the traditional electrical engineering notation for the AND gate followed by more modern notation.

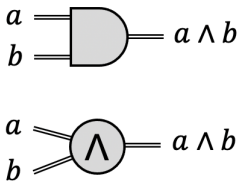

Figure 4: Two notations for the AND gate.

The *fan-out* operation is is often implicitly used in circuit diagrams. It's essentially a copying operation, whose output bits are multiple copies of its input bit. Recall that it's possible to copy classical information.

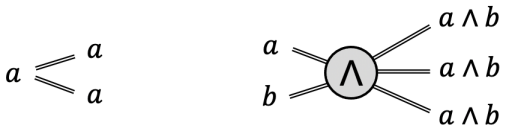

Figure 5: Notation for fan-out: of an input bit (left), and of the output of a gate (right).

What about the logical OR gate? We don't need it as one of our fundamental operations because it can be simulated by three NOT gates and one AND gate using *DeMorgan's Law*

$$a \vee b = \neg(\neg a \wedge \neg b). \tag{1}$$

Similarly, we don't need an XOR gate as a fundamental operation, which I leave as an exercise.

Exercise 1.1. *Define an XOR gate as mapping two input bits a and b to one output bit $a \oplus b$. Show that an XOR gate can be simulated by AND and NOT gates.*

1.2 Computing the majority of a string of bits

Every binary string of odd length has either more zeroes than ones or more ones than zeroes. Define the *majority* of an odd-length string as the most common value. Here's a table of values of the majority function for 3-bit strings, denoted MAJ_3.

a	$\mathsf{MAJ}_3(a)$
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

Figure 6: Caption

Here's a circuit that computes this majority function for 3-bit strings.

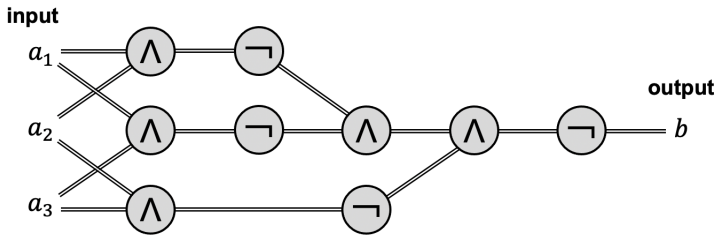

Figure 7: Classical circuit computing the majority value of three bits.

The input bits a_1, a_2, a_3 are on the left, information flows from left to right, and the output bit is $b = \mathsf{MAJ}_3(a_1, a_2, a_3)$. The circuit is based on this formula

$$\mathsf{MAJ}(a_1, a_2, a_3) = (a_1 \wedge a_2) \vee (a_1 \wedge a_3) \vee (a_2 \wedge a_3)\,, \tag{2}$$

where the OR gates are simulated by NOT and AND gates. The total number of OR and NOT gates is nine (and there are three implicit fan-out gates at the inputs).

Can you construct a classical circuit for the majority of odd-length strings larger than three? What is the gate count of your construction as a function of the string length? Does it grow polynomially or exponentially with respect to n?

Exercise 1.2 (level of difficulty depends on your prior familiarity with logic circuits)**.** *Construct a classical circuit that computes the majority of n bits for arbitrarily large odd n using a number of gates that scales polynomially with respect to n.*

1.3 Classical algorithms as logic circuits

We can represent algorithms as logic circuits along the lines of this diagram.

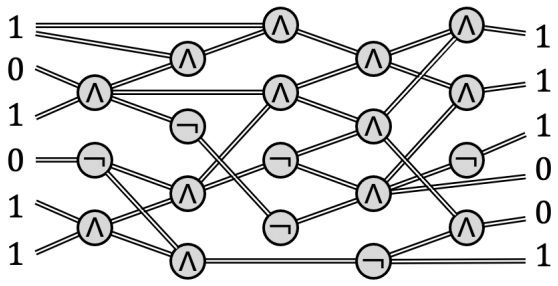

Figure 8: Generic form of a classical circuit (information flows from left to right).

The inputs are on the left, time flows left to right and the outputs are on the right. Classical computer algorithms are usually described in high level languages, but they implicitly represent many low-level logic operations, like this circuit.

A reasonable overall cost measure is the number of gates. There are other measures that we may care about such as the *depth*, which is the length of the longest path from an input to an output. Or the *width*, which is the maximum number of gates at any level, assuming that the gates are arranged in levels. But, to keep things simple, let's use the gate count as our main cost measure.

The number of gates depends on what the elementary operations are. We'll take these to be AND and NOT gates (and let's not count the fan-out operations). What's important is that a gate does a constant amount of computational work. If we allowed arbitrary gates then any circuit could be reduced to just one supergate, but the cost of that one gate would not be very meaningful, since we expect large gates to be expensive to implement.

1.4 Multiplication problem and factoring problem

Let's consider the *multiplication problem* where one is given two n-bit binary integers as input and the output is their product (a $2n$-bit integer). For example, for inputs 101 and 111, the output should be 100011 (because the product of 5 and 7 is 35).

Think of the number of bits n as being quite large, larger than the number of bits of the arithmetic logic unit of your computer. Do you know of an algorithm for performing this? I believe that you do know such an algorithm, because you learned one in grade school. In principle, you could multiply two numbers consisting

of thousands of digits by pencil and paper using that method. And it can be coded up as an algorithm (commonly referred to as the *grade school multiplication algorithm*). How efficient is this algorithm? Assuming you learned the algorithm that I did, its running time scales quadratically in its input size. By quadratic, I mean some constant times n^2 (for sufficiently large n).

The constant depends on the exact gate set that you use and we'll often use the *big-oh* notation to suppress that.

Definition 1.1. *For $f, g : \mathbb{N} \to \mathbb{N}$, $f(n) \in O(g(n))$ if $f(n) \leq cg(n)$ for some constant $c > 0$ (and for sufficiently large n).*

Of course if we want to *implement* an algorithm then we do care about what the constant multiple is. But if we're looking at things theoretically then the big-oh notation helps reduce clutter and focus attention on *asymptotic* growth rates, which tend to be more important for large n.

There are more more efficient algorithms than the grade school method. The current record is $O(n \log n)$ which is quite good. (There's a rich history of multiplication algorithms that evolved from $O(n^2)$ to $O(n \log n)$, but we won't get into that here.)

Now, let's look at the *factoring problem*, which can be roughly thought of as the inverse of the multiplication problem. The input is an n-bit number and the output is its prime factors. For example, for input 100011, the outputs are 101 and 111 (the prime factors of 35).

You probably also learned this algorithm in grade school for factoring numbers: First check if the number is divisible by 2. Then check for divisibility by 3. You can skip 4, since that's a multiple of 2. Then check 5, skip 6, check 7, and so on. You can stop once you get to the square root of the number because if you haven't found a divisor by then, the number must be prime.

How expensive is this computationally? For large n, there are lots of divisors to check, exponentially many. The cost is exponential in n, namely $O(\exp(n))$. There are more sophisticated algorithms than this method. The best currently-known classical algorithm for factoring is the so-called *number field sieve algorithm*, which scales exponentially in the cube root of n (to be more precise, $\exp(n^{1/3} \log^{2/3}(n))$). Since the cube root is in the exponent, this is still essentially exponential. There is no currently-known classical algorithm for factoring whose cost scales polynomially with respect to n.

In fact, the presumed difficulty of factoring is the basis of the security of many cryptosystems.

1.5 Quantum algorithms as quantum circuits

Now let's consider quantum circuits.

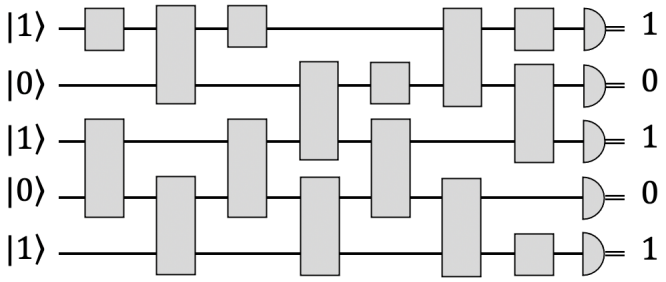

Figure 9: Generic form of a quantum circuit.

The input data is on the left, as computational basis states. Data flows from left to right as a series of unitary gates. Then measurement gates occur at the end, which yields the output of the computation.

Again, we can takes various cost measures for quantum circuits, such as the number of gates, the depth, or the width. Let's focus on the number of gates. As with classical circuits, we need a notion of what an elementary gate is. For now, let's consider the elementary gates to be the set of all 1-qubit and 2-qubit gates. We'll consider simpler gate sets later on.

Consider the factoring problem again. Peter Shor's famous factoring algorithm can be expressed as a quantum circuit for factoring that consists of $O(n^2 \log n)$ gates—a polynomial number of gates! An efficient implementation of this algorithm would break several cryptosystems, whose security is based in the presumed computational hardness of factoring.

2 Simulations between quantum and classical

It is natural to ask how the classical model of computation compares with the quantum model of computation. Can quantum circuits simulate classical circuits? Can classical circuits simulate quantum circuits? How efficient are the simulations?

For quantum circuits to simulate classical circuits, the Pauli X gate (that maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$ can be viewed as a NOT gate, but what quantum gate corresponds to the AND gate? None of the operations that we've considered so far has any resemblance to the AND gate. The Toffoli gate makes such a connection.

2.1 The Toffoli gate

I'd like to show you a 3-qubit gate called the *Toffoli gate*, which will be useful for simulating AND gates. It's denoted this way, like a CNOT gate, but with an additional control qubit.

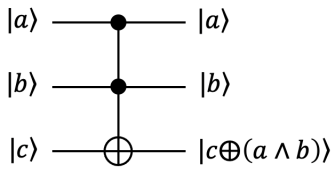

Figure 10: Toffoli gate acting on computational basis states ($a, b, c \in \{0, 1\}$).

On computational basis states $|a\rangle$, $|b\rangle$, and $|c\rangle$ it flips the third qubit if and only if both of the control qubits are in state $|1\rangle$; otherwise it does nothing. The 8×8 unitary matrix for the gate is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3}$$

For arbitrary 3-qubit states, that are not necessarily computational basis states, the action of the gate on the state is to multiply the state vector by this 8×8 matrix.

The Toffoli gate is sometimes called the *controlled-controlled-NOT* gate. This makes sense, because the Toffoli gate is a controlled-CNOT gate.

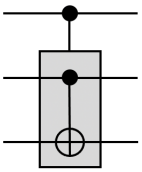

Figure 11: Toffoli gate is a controlled-CNOT gate (also a controlled-controlled-NOT gate).

2.2 Quantum circuits simulating classical circuits

Theorem 2.1. *Any classical circuit of size s can be simulated by a quantum circuit of size $O(s)$, where the gates used are Pauli X, CNOT, and Toffoli.*

Proof. We use Toffoli gates to simulate AND gates as illustrated in figure 12.

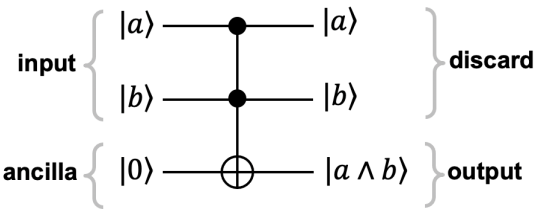

Figure 12: Using a Toffoli gate to simulate a classical AND gate.

Bits are represented as computational basis states in the natural way (bits $a, b \in \{0, 1\}$ are represented by $|a\rangle$ and $|b\rangle$). To compute the AND of a and b, a third ancilla qubit in state $|0\rangle$ is added and then a Toffoli gate is applied as shown in figure 12. This causes the state of the ancilla qubit to become $|a \wedge b\rangle$. The first two qubits can be discarded, or just ignored for the rest of the computation. That's how an AND gate can be simulated.

To simulate NOT gates, we can use the Pauli X gate.

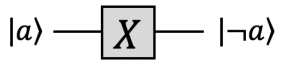

Figure 13: Using an X-gate to simulate a classical NOT gate.

Finally, we can explicitly simulate a fan-out gate by adding an ancilla in state $|0\rangle$ and apply a CNOT gate, as in figure 14.

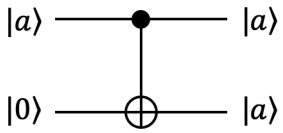

Figure 14: Using a CNOT gate to simulate a classical fan-out gate.

□

Remember that circuit in figure 7 for computing the majority of 3 bits? Here's a quantum circuit that simulates that circuit using Toffoli gates for AND and X gates for NOT.

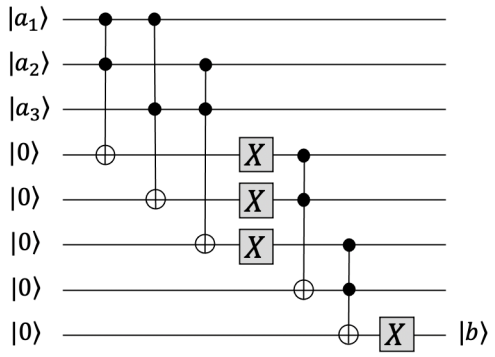

Figure 15: Computing the majority of three bits using Toffoli and X gates.

The output is the last qubit. (Note how this quantum circuit does not need to explicitly simulate the fan-out gates of the inputs.)

You may have noticed that we defined quantum circuits to be in terms of 1-qubit and 2-qubit gates, but our simulation of classical circuits used 3-qubit gates: the Toffoli gate. We can remedy this by simulating each each Toffoli gate by a series 2-qubit gates. I'm going to show how to do this.

To start, we will make use of a 2×2 unitary matrix with the property that if you square it you get the Pauli X. Since X is essentially the NOT gate, this matrix is sometimes referred to as a "square root of NOT". Note that, in classical information, there is no square root of NOT, so the quantum square root of NOT can be regarded as a curiosity. Let's refer to this matrix as V.

Exercise 2.1 (straightforward)**.** *Find a* 2×2 *unitary matrix* V *such that* $V^2 = X$.

Henceforth, I'm going to assume that we have such a matrix V in hand. From this, we can define a controlled-V gate. Also, we can define a controlled-V^* gate. Now, consider the following circuit, consisting of 2-qubit gates.

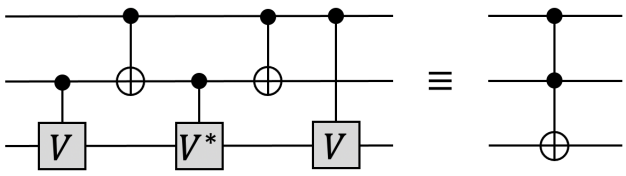

Figure 16: Simulating a Toffoli gate in terms of 2-qubit gates.

I claim that it computes the Toffoli gate. How can we verify this?

Let's discuss two possible approaches. The first approach has the advantage that it's completely mechanical. No creative idea is needed to carry it out. What you can do is work out the 8×8 matrix corresponding to each gate and then multiply[1] the five matrices and see if the product is the matrix in Eq. (3) for the Toffoli gate.

A second approach is to try to find shortcuts based on the logic of the gates, to avoid tedious calculations. In this case, note that it it sufficient to verify that the circuits are equivalent in the case where the first two qubits are in computational basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$.

Consider the $|00\rangle$ case. In this case, none of the controlled-gates have any effect, so the state on the third qubit doesn't change—which is consistent with what the Toffoli gate does in this case. What about the $|01\rangle$ case? Then none of the gates controlled by the first qubit have any effect, so the circuit reduces to a controlled-V followed by a controlled-V^*, acting on the second and third qubits, which simplifies to the identity (since $VV^* = I$). I'll leave it to you to analyze the other two cases.

Exercise 2.2. *Continue the analysis of the correctness of the circuit in figure 16 for the cases where the control qubits are in state* $|10\rangle$ *and in state* $|11\rangle$*.*

This kind of approach, when it can be made to work, has two advantages. It avoids a tedious calculation. In addition, thinking about it this way, we can gain some intuition about why the circuit works. In the future, if you need to design a quantum circuit to achieve something, such intuition can be helpful.

Exercise 2.3 (Challenging)**.** *Show how to simulate a Toffoli gate using only* CNOT *gates and 1-qubit gates. Thus, no controlled-U gates for* $U \neq X$ *are allowed.*

Some classical algorithms make use of random number generators and we have not captured this in our definition of classical circuits. Say we allow our classical algorithms to access random bits, that are zero with probability $\frac{1}{2}$ and one with probability $\frac{1}{2}$. Can we simulate such random bits with quantum circuits? This is actually quite easy, since constructing a $|+\rangle$ state and measuring it yields a random bit.

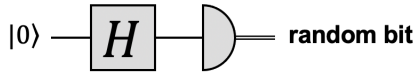

Figure 17: Using a Hadamard and a measurement gate to generate a classical random bit.

[1] A word of caution: gates go from left to right, but the corresponding matrix products go from right to left (because we multiply vectors on the left by matrices).

But this entails an intermediate measurement. Our quantum circuits will not look like the ones we defined earlier, where all the measurements are at the end. Can we simulate random bits without the intermediate measurements? The answer is yes, by the following construction.

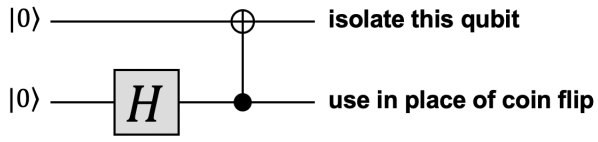

Figure 18: Simulating random bits without using measurements.

Instead of measuring the qubit in the $|+\rangle$ state, we apply a CNOT gate to a target qubit (an ancilla) in state $|0\rangle$. Then we ignore that ancilla qubit for the rest of the computation. The final probabilities when the circuit is measured at the end will be exactly the same as if a random bit had been inserted at that place in the computation. I will leave this as an exercise for you to prove that this works.

Using our decomposition of the Toffoli gate into 2-qubit gates and our results about simulating classical randomness, we can strengthen Theorem 2.1 to the following.

Corollary 2.1. *Any classical probabilistic circuit of size s can be simulated by a quantum circuit of size $O(s)$, consisting of 1-qubit and 2-qubit gates.*

2.3 Classical circuits simulating quantum circuits

Classical circuits can simulate quantum circuits but the only known constructions have exponential overhead.

Theorem 2.2. *A quantum circuit of size s acting on n qubits can be simulated by a classical circuit of size $O(sn^2 2^n)$.*

Proof. The idea of the simulation is quite simple. An n-qubit state is a 2^n-dimensional vector $\sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$. Store the 2^n in an array of size 2^n, each with n-bits precision (which means accuracy within 2^{-n}).

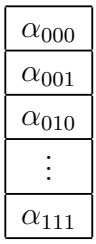

Figure 19: 2^n-dimensional array storing the amplitudes of state $\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$.

The initial state is a computational basis state. Each gate corresponds to $2^n \times 2^n$ matrix and the effect of the gate is to multiply the state vector by that matrix. In fact that matrix will be sparse for 1-qubit gates and 2-qubit gates, with only a constant number of nonzero entries for each row and each column. Therefore, there are a constant number of arithmetic operations per entry of the array. Let's allow $O(n^2)$ elementary gates for each such arithmetic operation[2] (on n-bit precision numbers). The simulation cost is $O(n^2 2^n)$ for each gate in the circuit. We multiply that by the number of gates in the quantum circuit s to get a cost of $O(sn^2 2^n)$. That's the gate cost to get the final state of the computation, just before the measurement.

What about the measurements? For this we, we compute the absolute value squared of each entry of the array. Again, that's 2^n arithmetic operations. At this point, the entries in the array are the probabilities after the measurement.

The output of the quantum circuit is not the probabilities; it's a sample according to the distribution. How do we generate that sample? First we sample the first bit of the output, the probability that that this bit is 0 is the sum of the first half of the entries of the array; the probability that bit is 1 is the sum of the second half. Once we have the first bit, we can use a similar approach for the second bit, using conditional probabilities, conditioned on the value of the first bit, and so on for the other bits. This also costs $O(n^2 2^n)$ elementary classical gates. □

Note that if we take the quantum circuit that results from Shor's algorithm and then simulate it by a classical circuit then simulate that quantum circuit by a classical circuit, the result is not very efficient. It's expoenntial and in fact it it's worse than the best currently-known classical algorithm for factoring.

If we view the aforementioned simulation in the usual high-level language of algorithms, it is exponential *space* in addition to exponential time (because it stores the

[2]Using simple grade-school algorithms for addition and multiplication.

huge array). In fact there's a way to reduce the storage space to polynomial—while maintaining exponential time.

Exercise 2.4 (challenge)**.** *Show how to do the above simulation as an algorithm using only a polynomial amount of space (memory).*

3 A brief look at computational complexity classes

On theoretical computer science, there are various taxonomies of computational problems according to their computational difficulty. I'll briefly show you a few of these and how the power of quantum computers fits in within this classification.

Consider all binary functions over the set of all binary strings (of the form $f : \{0,1\}^* \to \{0,1\}$). The input is a binary string of some length n, where n can be anything. And the output is one bit. These are sometimes called decision problems since the answer is a binary decision, 0 or 1, yes or no, accept or reject. This is a common convention for reasons of simplicity. Most problems that are not naturally decision problems can be reworked to be expressed as decision problems.

P (polynomial time)
Solvable by $O(n^c)$-size classical circuits[3] (for some constant c).

BPP (bounded-error probabilistic polynomial time)
Solvable by $O(n^c)$-size probabilistic classical circuits whose worst-case error probability[4] is $\leq \frac{1}{4}$.

BQP (bounded-error quantum polynomial time)
Solvable by $O(n^c)$-size quantum circuits with worst-case error probability $\leq \frac{1}{4}$.

EXP (exponential time)
Solvable by $O(2^{n^c})$-size classical circuits.

The following containments among these complexity classes are known:

$$\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{EXP}. \tag{4}$$

[3]Technically, we require *uniform circuit families*, one for each input size.

[4]There is some arbitrariness in the error bound $\frac{1}{4}$. Any polynomial-size circuit achieving this can be converted into another circuit whose error probability is $\leq \epsilon$ by repeating the process $\log(1/\epsilon)$ times and taking the majority value. Using $\frac{1}{4}$ is simple, though any constant below $\frac{1}{2}$ would work.

4 The black-box model

In the next few subsequent sections, we are going to see some simple quantum algorithms in a framework called the *black-box model*. First, I'll explain this computational model.

4.1 Classical black-box queries

Imagine that f is some function that is unknown to us and we're given a device that enables us to evaluate f on any particular input, of our choosing, and that's our *only* way of acquiring information about f. Such a device is commonly called a *black-box* for f.

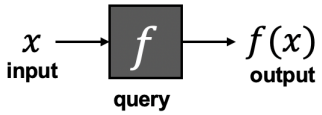

Figure 20: A gate performing an f-query.

If we want to evaluate the function on some input x, we insert x into the black-box and then the value of $f(x)$ pops out, for us to see. We call this process of evaluating the function at an input an *f-query*.

Now, suppose that we want to acquire some specific information about a function f, and that we want to do this with as few queries as possible. We can think of this as a game, where we want to know something about an unknown function f, and we're only allowed to ask questions like "what's the value of f at point x?" for any x of our choosing. How many such questions do we have to ask?

One example that illustrates the general idea is *polynomial interpolation*. Here, one is given a black-box computing an unknown polynomial of degree up to d, and the goal is to determine which polynomial it is. At how many points does the polynomial have to be evaluated to accomplish this?

Figure 21: Interpolating linear and quadratic functions with as few queries as possible.

If $f : \mathbb{R} \to \mathbb{R}$ is an unknown linear function then one evaluation is insufficient for determining what f is; however, two queries suffice. If $f : \mathbb{R} \to \mathbb{R}$ quadratic then three evaluations are necessary and sufficient. In general, if f is a polynomial with degree up to d then the number of queries that are necessary and sufficient is $d + 1$.

We will focus our attention on functions over *finite* domains instead of $\mathbb{R}$, such as $\{0,1\}^n$. Let us begin by considering the simple case where we have an unknown $f : \{0,1\} \to \{0,1\}$. There are only four such functions and here are the tables of values for each of them:

x	$f(x)$
0	0
1	0

x	$f(x)$
0	1
1	1

x	$f(x)$
0	0
1	1

x	$f(x)$
0	1
1	0

Figure 22: The four functions $f : \{0,1\} \to \{0,1\}$.

Suppose that we're given a black-box for such a function, but we don't know of the four functions it is.

Suppose that our goal is to determine whether:

- $f(0) = f(1)$ (the first two cases in figure 22); or
- $f(0) \neq f(1)$ (the last two cases in figure 22).

To be clear, we are not required to determine which of the four functions f is; just whether it's among the first two or the last two. How many queries do we need to accomplish this? Please think about this. We will get back to this question in section 5.1.

4.2 Quantum black-box queries

A classical query is along the lines of figure 20. We can set the input to any x in the domain of f. Then we receive as output the value of $f(x)$. Does it make sense to define a *quantum* query? Let's keep our attention on the simple case where $f : \{0,1\} \to \{0,1\}$ (figure 22); we will consider more general cases later.

I first want to show you a naïve first attempt at defining a quantum query that does *not* work. The classical query maps bits to bits. Define a quantum query (mapping qubits to qubits) that correspondingly maps computational basis states to computational basis states, as in figure 23.

Figure 23: Naïve attempt to define a quantum query—that doesn't work!

For each of the four functions $f : \{0,1\} \to \{0,1\}$ in figure 22, here are the corresponding mappings on computational basis states.

$\lvert a\rangle$	$\lvert f(a)\rangle$
$\lvert 0\rangle$	$\lvert 0\rangle$
$\lvert 1\rangle$	$\lvert 0\rangle$

$\lvert a\rangle$	$\lvert f(a)\rangle$
$\lvert 0\rangle$	$\lvert 1\rangle$
$\lvert 1\rangle$	$\lvert 1\rangle$

$\lvert a\rangle$	$\lvert f(a)\rangle$
$\lvert 0\rangle$	$\lvert 0\rangle$
$\lvert 1\rangle$	$\lvert 1\rangle$

$\lvert a\rangle$	$\lvert f(a)\rangle$
$\lvert 0\rangle$	$\lvert 1\rangle$
$\lvert 1\rangle$	$\lvert 0\rangle$

Figure 24: Input-output relationships for naïvely defined quantum query gate.

By linearity, we get these four linear operators.

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{5}$$

The third and fourth are familiar unitary operations: the identity operation and the Pauli X operation. But notice that the first two are not unitary operations. This violation of unitarity is a serious problem. For example, the first two linear operators both map the state $|-\rangle$ to the zero vector, a two-dimensional vector whose amplitudes are both zero, which makes no sense as a quantum state. So this approach does not work.

In order for a classical mapping to be quantizable in the above manner it must be bijective. Then the underlying linear operator is given by a permutation matrix, which is unitary.

We will first define a *reversible classical f-query* that is bijective whether or not f itself is bijective. In the case where $f : \{0,1\} \to \{0,1\}$, the reversible classical f-query is the mapping $\{0,1\}^2 \to \{0,1\}^2$ defined as $(a,b) \mapsto (a, b \oplus f(a))$ (for all $a, b \in \{0,1\}$). Here is notation for a reversible classical f-query.

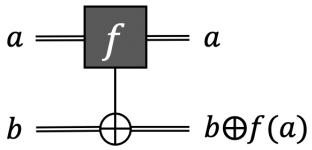

Figure 25: Reversible classical f-query (for $f : \{0,1\} \to \{0,1\}$).

Why is this mapping $\{0,1\}^2 \to \{0,1\}^2$ bijective? One way to see this is to observe that the mapping is its own inverse.

The reversible classical f-query is easy to quantize as the 2-qubit unitary operation that maps $|a\rangle|b\rangle$ to $|a\rangle|b \oplus f(a)\rangle$ (for all $a, b \in \{0,1\}^2$). Here is notation for a quantum f-query (in the case where $f : \{0,1\} \to \{0,1\}$).

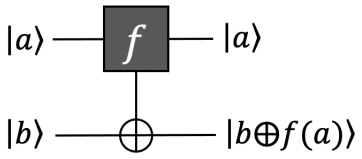

Figure 26: Quantum f-query (for $f : \{0,1\} \to \{0,1\}$).

Note that the above defines the effect of the f-query on computational basis states. This determines a unitary operator that defines the effect of the f-query on arbitrary quantum states.

We can generalize the above definition to arbitrary functions $f : \{0,1\}^n \to \{0,1\}^m$. The f-query is a unitary operation acting on $n+m$ qubits defined as follows.

Definition 4.1. *Let function $f : \{0,1\}^n \to \{0,1\}^m$. Then a* quantum f-query *is defined as the unitary operation acting on $n+m$ qubits with the property that, for all $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$,*

$$|a\rangle|b\rangle \mapsto |a\rangle|b \oplus f(a)\rangle, \tag{6}$$

where $b \oplus f(a)$ denotes the bit-wise[5] *XOR between the m-bit strings b and $f(a)$.*

Here is notation for a general quantum f-query.

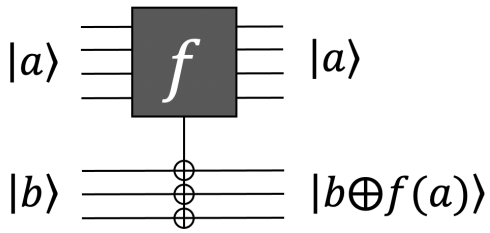

Figure 27: Quantum f-query (for $f : \{0,1\}^n \to \{0,1\}^m$).

[5]The *bit-wise* XOR between $(b_1, b_2, \ldots, b_m)$ and $(c_1, c_2, \ldots, c_m)$ is $(b_1 \oplus c_1, b_2 \oplus c_2, \ldots, b_m \oplus c_m)$.

5 Simple quantum algorithms in black-box model

The simple quantum algorithms in this section are admittedly curiosities, rather than practical algorithms. It's hard to think of real-world applications that fit their framework, and where it worth the trouble to build a quantum device to solve these problems. What you should pay attention to are the maneuvers that these quantum algorithms make. After these algorithms, we'll be seeing increasingly sophisticated extensions of these maneuvers, that accomplish more dramatic algorithmic feats.

5.1 Deutsch's problem

The first problem that we'll consider involves functions $f : \{0,1\} \to \{0,1\}$ (the four such functions are shown in figure 22). Remember the question of how many queries are necessary to determine whether or not $f(0) = f(1)$? This is the definition of Deutsch's Problem.

Definition 5.1. Deutsch's Problem *is defined as the problem where one is given as input a black box for some* $f : \{0,1\} \to \{0,1\}$ *and the goal is to determine whether or not* $f(0) = f(1)$ *by making queries to* f*.*

Let's first consider classical queries necessary to solve Deutsch's problem. One query is not sufficient. To see why this is so, suppose that you make one query at some $a \in \{0,1\}$ to acquire $f(a)$. This gives absolutely no information about the *other* value, $f(\neg a)$. It is possible that $f(\neg a) = f(a)$ and it is possible that $f(\neg a) \neq f(a)$. Therefore, two queries necessary and clearly two queries are also sufficient.

You may wonder whether the number of *reversible* classical queries is different. In fact, reversible classical query at (a, b) provide exactly the same amount of information as a simple classical queries of at a. The output of the reversible query at (a, b) is $(a, b \oplus f(a))$, and note that (a, b) are already known. Therefore, there are only two possibilities of interest for the output:

$$\begin{cases} b \oplus f(a) = b, \text{ which occurs if any only if } f(a) = 0; \\ b \oplus f(a) \neq b, \text{ which occurs if any only if } f(a) = 1. \end{cases} \tag{7}$$

Therefore, even with reversible classical queries, one query is not sufficient.

Now let's see what an algorithm solving Deutsch's Problem with two reversible classical queries looks like. Here's a classical circuit expressing such an algorithm.

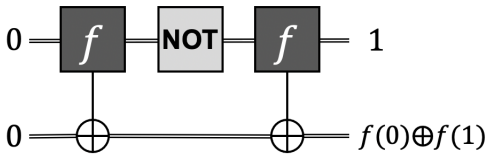

Figure 28: Classical algorithm for Deutsch that makes two reversible classical queries.

The first query XORs the value of $f(0)$ to the second bit. Then the first bit is flipped to 1, so the second query XORs the value of $f(1)$ to the second bit. At the end, the second bit contains the value of $f(0) \oplus f(1)$, which is the solution to Deutsch's problem.

There is an obvious 2-query quantum algorithm that solves Deutsch's problem just like the circuit in figure 28. But quantum circuits need not be restricted to these types of operations, where states are always in computational basis states. This quantum circuit that solves Deutsch's problem with just one single query!

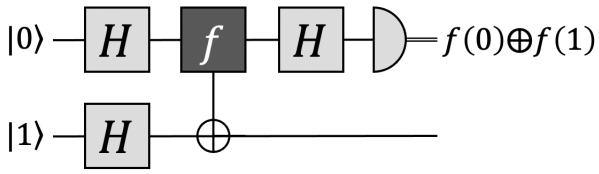

Figure 29: Quantum algorithm for Deutsch that makes just one quantum query.

How does it work? It's very different from any classical algorithm. There are three Hadamard transforms, and each one plays a different role in the computation. Let's look at how each Hadamard contributes to the computation in this order.

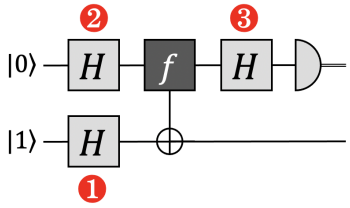

Figure 30: The three separate Hadamard transforms.

❶ What the first Hadamard does

This is the Hadamard applied to the second qubit, in state $|1\rangle$. Obviously, this creates the $|-\rangle$ state. What's interesting about creating this state here is that it's an

eigenvector of the Pauli X. Performing an X-operation on $|-\rangle$ causes it to become[6] $\frac{1}{\sqrt{2}}|1\rangle - \frac{1}{\sqrt{2}}|0\rangle = -|-\rangle$.

With the second qubit in state $|-\rangle$, if the first qubit is in the computational basis state $|a\rangle$ then the f-query causes the second qubit to change by a factor of -1 if and only if $f(a) = 1$, as shown in the following circuit diagram.

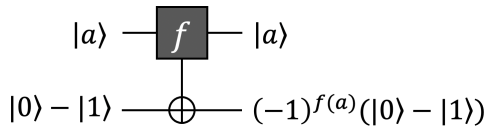

Figure 31: Caption.

Note that $(-1)^{f(a)}$ is a succinct notation for expressing the two cases, since

$$(-1)^{f(a)} = \begin{cases} +1 & \text{if } f(a) = 0 \\ -1 & \text{if } f(a) = 1. \end{cases} \tag{8}$$

Notice that the 2-qubit output state is $(-1)^{f(a)}|a\rangle|-\rangle$ and the $(-1)^{f(a)}$ does not really belong to a specific qubit of the two. We can equivalently write the circuit this way.

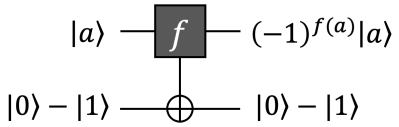

Figure 32: Caption.

But this is an interesting way of thinking about what the query does when the second qubit is in state $|-\rangle$. Namely, the first qubit picks up a phase of $(-1)^{f(a)}$, and the second qubit doesn't change. We sometimes call this *querying in the phase*.

At this point, you might wonder why we should care about this, since this is a global phase, and we know that global phases don't matter. But it's only a global phase if the first qubit is in a computational basis state, of the form $|a\rangle$. It's *not* a global phase if the first qubit is in superposition. This brings us to the role of the second Hadamard.

[6]Let's keep track of the distinction between between $|-\rangle$ and $-|-\rangle$, even it's only a global phase. The significance of this will become clear shortly.

❷ What the second Hadamard does

This brings us to the role of the second Hadamard, that's applied to the first qubit. That causes the first input qubit to the query to be in an equally weighted superposition of $|0\rangle$ and $|1\rangle$, namely, the $|+\rangle$ state. Now the state of the first output qubit after the query is in a superposition of $|0\rangle$ and $|1\rangle$ with respective phases $(-1)^{f(0)}$ and $(-1)^{f(1)}$.

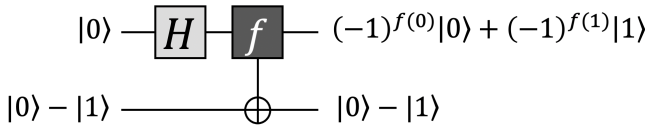

Figure 33: Caption.

So, after the query, the state of the first qubit is $\frac{1}{\sqrt{2}}(-1)^{f(0)}\,|0\rangle + \frac{1}{\sqrt{2}}(-1)^{f(1)}\,|1\rangle$. Let's look at what this state is for each of the four possible functions back in figure 22. The corresponding states of the first qubit are respectively

$$\tfrac{1}{\sqrt{2}}\,|0\rangle + \tfrac{1}{\sqrt{2}}\,|1\rangle \quad -\tfrac{1}{\sqrt{2}}\,|0\rangle - \tfrac{1}{\sqrt{2}}\,|1\rangle \quad \tfrac{1}{\sqrt{2}}\,|0\rangle - \tfrac{1}{\sqrt{2}}\,|1\rangle \quad -\tfrac{1}{\sqrt{2}}\,|0\rangle + \tfrac{1}{\sqrt{2}}\,|1\rangle\,. \tag{9}$$

Notice that, for the first two cases (where $f(0) = f(1)$), the state is $\pm\big(\frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle\big)$; and for the last two cases (where $f(0) \neq f(1)$), the state is $\pm\big(\frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle\big)$. Therefore, to solve Deutsch's problem, we need only distinguish between $\pm\,|+\rangle$ and $\pm\,|-\rangle$. Which brings us to the third Hadamard.

❸ What the third Hadamard does

This Hadamard maps $\pm\,|+\rangle$ to $\pm\,|0\rangle$ and $\pm\,|-\rangle$ to $\pm\,|1\rangle$. Measuring the resulting qubit in the computational basis yields

$$\begin{cases} 0 & \text{if } f(0) = f(1) \\ 1 & \text{if } f(0) \neq f(1). \end{cases} \tag{10}$$

Therefore, the output bit of the algorithm is the solution to Deutsch's problem.

Here is a summary of the 1-query quantum algorithm for Deutsch's problem, and the role that each of the three Hadamard transforms plays in it.

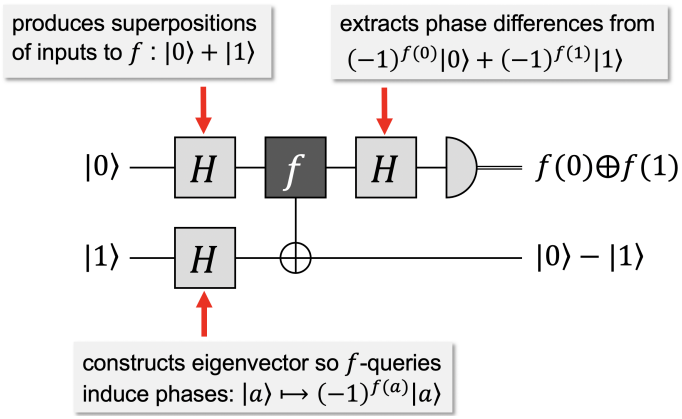

Figure 34: Summary of the role that each H transformation plays in the algorithm.

This quantum algorithm accomplishes something with one query that would require two classical queries by any classical algorithm.

Notice what this algorithm does *not* do. It does not somehow extract both values of the function, $f(0)$ and $f(1)$, with one single query. In fact, if you run this algorithm, then you get *no* information about the value of $f(0)$ itself. Also, you get *no* information about the value $f(1)$ itself. You *only* get information about $f(0) \oplus f(1)$.

5.2 One-out-of-four search

Now let's try to generalize this methodology to another problem.

Let $f : \{0,1\}^2 \to \{0,1\}$ with the special property that it attains the value 1 at exactly one of the four points in its domain. There are four possibilities for such a function and here are their truth tables.

x	$f_{00}(x)$
00	1
01	0
10	0
11	0

x	$f_{01}(x)$
00	0
01	1
10	0
11	0

x	$f_{10}(x)$
00	0
01	0
10	1
11	0

x	$f_{11}(x)$
00	0
01	0
10	0
11	1

Figure 35: The four functions $f : \{0,1\}^2 \to \{0,1\}$ that take value 1 at a unique point.

Now, suppose that you're given a black-box computing such a function. You're promised that it's one of these four, but you're not told which one. Your goal is to determine which of the four functions it is.

First of all, how many classical queries do you need to do this? The answer is three queries. You can query in the first three places and see if one of them evaluates to 1. If none of them do then, by process of elimination, you can deduce that the 1 must be the fourth place.

Now, what about quantum queries? Let's try to build a good quantum algorithm for this. For functions mapping two bits to one bit, the queries look like this.

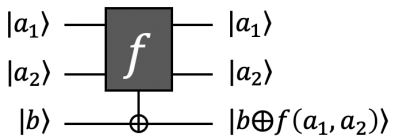

Figure 36: Query gate for a function $f : \{0,1\}^2 \to \{0,1\}$.

There are two qubits for the input, and a third qubit for the output of the function. In the computational basis, the value of $f(a1, a2)$ is XORed onto the third qubit. Of course, that description is for states in the computational basis. But, there is a unique unitary operation that matches this behavior on the computational basis states.

Let's start, along the lines that we did for Deutsch's algorithm: by setting the target qubit to state ket-minus so as to query in the phase; and then querying the inputs in a uniform superposition.

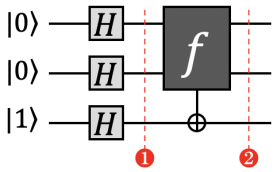

Figure 37: Starting along the lines of Deutsch's algorithm. (Intermediate stages are marked.)

Let's trace through the execution of this quantum circuit. At each stage, we will determine the state of the three qubits at that stage.

State at stage ❶

The state after the Hamadard operations, but just before the query is

$$\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big)\,|-\rangle\,. \tag{11}$$

State at stage ❷

The query does not affect the state of the third qubit, but it changes the state of the first two qubits to

$$\begin{cases} -|00\rangle + |01\rangle + |10\rangle + |11\rangle & \text{in the case of } f_{00} \\ |00\rangle - |01\rangle + |10\rangle + |11\rangle & \text{in the case of } f_{01} \\ |00\rangle + |01\rangle - |10\rangle + |11\rangle & \text{in the case of } f_{10} \\ |00\rangle + |01\rangle + |10\rangle - |11\rangle & \text{in the case of } f_{11}. \end{cases} \tag{12}$$

Looking at these four states, what noteworthy property do they have? They're orthogonal! If you take the dot-product between any two of these vectors then you get two positive terms and two negative terms, which cancel out. Since they're orthogonal, we can measure with respect to this basis. In other words, there exists a unitary operation U that maps these states to the computational basis, and then we can measure in the computational basis.

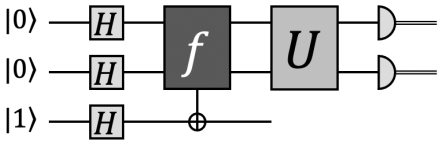

Figure 38: Quantum algorithm for the 1-out-of-4 search problem.

So this quantum circuit only makes one quantum query and it correctly identifies the function (among the four). And this would require 3 classical queries.

As an aside, a small challenge question is to give a quantum circuit with only H and CNOT gates that computes the above U. I'll leave this for you to consider.

It is possible to get a more dramatic quantum vs. classical query separation than 1 vs. 3?

5.3 Constant vs. balanced

Next we'll see a problem where a quantum algorithm solves a problem with exponentially fewer queries than any classical algorithm.

Let $f : \{0,1\}^n \to \{0,1\}$. We call such a function *constant* if either its value is 0 everywhere, or its value is 1 everywhere. We call such a function *balanced* if its value is 0 in exactly the same number of places that its value is 1. There are 2^n possible

inputs to such a function. Therefore, a balanced function takes value 0 in exactly 2^{n-1} places and it takes value 1 in exactly 2^{n-1} places.

Here are some examples of tables of functions for the $n = 3$ case:

a	$f_1(a)$	$f_2(a)$	$f_3(a)$	$f_4(a)$	$f_5(a)$
000	0	1	0	1	0
001	0	1	0	1	0
010	0	1	0	0	0
011	0	1	0	1	0
100	0	1	1	0	0
101	0	1	1	0	0
110	0	1	1	0	1
111	0	1	1	1	0

Figure 39: Examples of functions that are constant, balanced, and neither.

The first two functions, f_1 and f_2, are the two constant functions: the all-zero function and the all-one function. The third function f_3 is a balanced function with all the zeroes before all the ones. And f_4 is another balanced function, but with the positions of the zeros and ones mixed up. How many balanced functions are there? Exponentially many (the number is approximately $\frac{2^n}{\sqrt{n}}$). And f_5 is neither constant nor balanced—just as a reminder that this third category exists.

For the *constant-vs-balanced problem*, we're given a black-box computing a function that is promised to be either constant or balanced, but we're not told which one. Our goal is to figure out which of the two cases it is, with as few queries to f as possible.

First of all, how many queries do we need to solve this problem by a classical algorithm? If you think about this, you'll probably realize that, even if you query the function in many spots and always see the same value, you still might not know which way it goes. It could be constant. But it could also be that, in many of the places that you did not query, the function takes the opposite value and it's actually a balanced function. It's only after you've queried in more than half of the spots that you can be sure about which case it is. Therefore, for number of queries that a classical algorithm must make is $2^{n-1} + 1$. That's a lot of queries when n is large.

How many queries can a quantum algorithm get by with? In fact, this problem can be solved by a quantum algorithm that makes just one single query! Let's see how that works.

We'll start off as usual, setting the target qubit to state $|-\rangle$ and setting the the other qubits—those where the inputs to f are—into a uniform superposition of all n-qubit basis states. That's what applying a Hadamard to each of n qubits in state $|0\rangle$ does.

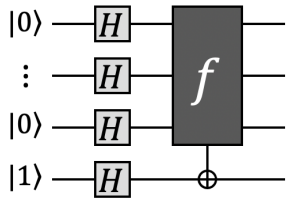

Figure 40: Starting off in a manner similar to the two previous algorithms.

So the state right after the query is

$$\frac{1}{\sqrt{2^n}}\left(\sum_{a\in\{0,1\}^n}(-1)^{f(a)}\,|a\rangle\right)\otimes|-\rangle\,. \tag{13}$$

The first n qubits are the interesting part of this state, which is a uniform superposition of all 2^n computational basis states of n qubits, with a phase of $(-1)^{f(a)}$ for each $|a\rangle$.

What does this state looks like in the constant case? In that case, either all the phases are $+1$ or all the phases are -1. So the state remains in a uniform superposition of computational basis states and just picks up a global phase of $+1$ or -1.

Now, what can we say about the state in the balanced case? There are lots of possibilities, depending on which particular balanced function it is. But one thing we can say is that the state is orthogonal to the state that arises in the constant case. Why? Because, in the computational basis, the state will have $+1$ in half of its components and -1 in the other half. So when you take the dot product, with a vector that has (say) $+1$ in each component you get an equal number of $+1$s and -1s, which cancel and results in zero.

Something that we've seen a few times before is that, when the cases that we're trying to distinguish between result in orthogonal states, we're in good shape. Being orthogonal means that, in principle, the states are perfectly distinguishable.

Let's make this more explicit. Suppose that we now apply a Hadamard to each of the first n qubits, and consider what happens in the constant case and in the balanced case. In the constant case, this transforms the state to the $\pm\,|00\ldots0\rangle = \pm\,|0^n\rangle$. In

the balanced case, since unitary operations preserve orthogonality relationships, the state is transformed to some state that is orthogonal to $|0^n\rangle$.

After this final layer of Hadamards, we measure the state of the first n qubits.

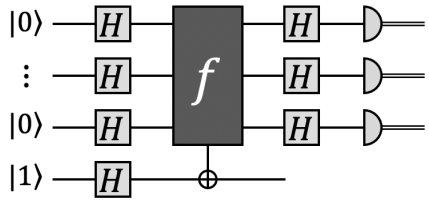

Figure 41: Quantum algorithm for the 1-out-of-4 search problem.

If the outcome is $00\ldots0 = 0^n$ then we report "constant" and if the outcome is any string that's not 0^n (not all zeroes) then we report "balanced". Note that, in the balanced case, it's impossible to get outcome 0^n, because measuring a state orthogonal to $|0^n\rangle$ cannot result in outcome 0^n.

That's how the one-query quantum algorithm for the constant-vs-balanced problem works. It achieves something with one single query that requires exponentially many queries by any classical algorithm.

5.4 Probabilistic vs. quantum query complexity

Although everything I've said about the classical and the quantum query cost for the constant-vs-balanced problem is true, there's something unsatisfying about the "exponential reduction" in the number of queries that the quantum algorithm attains. The classical query cost is expensive *only if we require absolutely perfect performance.* If a classical procedure queries f in random locations then, in the case of a balanced function, it would have to be very unlucky to always draw the same bit value.

Here's a classical probabilistic procedure that makes just two queries and performs fairly well. It selects two locations randomly (independently) and then outputs "constant" if the two bits are the same and "balanced" if the two bit values are different.

What happens if f is constant? In that case the algorithm always succeeds. The two bits will always be the same. What happens if f is balanced? In that case the algorithm succeeds with probability $\frac{1}{2}$. The probability that the two bits will be different will be $\frac{1}{2}$.

By repeating the above procedure k times, we can make the error probability exponentially small with respect to k. Only 4 queries are needed to obtain success

probability $\frac{3}{4}$. And the success probability can be made to any constant, arbitrarily close to 1, with a constant number of queries.

In summary, we've considered three problems in the black-box model. For each problem, a quantum algorithm solves it with just one query, but more queries are required by classical algorithms.

problem	quantum	classical deterministic	classical probabilistic
Deutsch	1	2	2
1-out-of-4 search	1	3	3
Const. vs. balanced	1	$2^{n-1}+1$	$O(1)$

Figure 42: Summary of query costs for problems considered so far.

For Deutsch's problem, any classical algorithm requires 2 queries. For the 1-out-of-4 search problem, any classical algorithm requires 3 queries. And, for the constant-vs-balanced problem, any classical algorithm requires exponentially many queries to solve the problem perfectly; however, there is a probabilistic classical algorithm that makes only a constant number of queries and solves the problem with bounded error probability.

Along this line of thought, the following question seems natural: Is there a black-box problem for which the quantum-vs-classical query cost separation is stronger? For example, for which even probabilistic classical algorithms with bounded-error probability require exponentially more queries than a quantum algorithm?

We'll address this question in the next section.

6 Simon's problem

We are going to investigate a black-box problem called *Simon's Problem*. For this problem, there is an exponential difference between the probabilistic classical query cost and the quantum cost. Also the quantum algorithm for this problem introduces some powerful algorithmic techniques in a simple form. Simon's Problem is a slightly more complicated black-box problem than those that we've seen up to now, involving functions from n bits to n bits.

It is interesting for two reasons:

1. It improves on the progression of black-box problems where quantum algorithms outperform classical algorithms. The quantum algorithm requires exponentially fewer queries than even probabilistic classical algorithms that can err with constant probability, say $\frac{1}{4}$.

2. The quantum algorithm for Simon's problem introduces a technique that transcends the black-box model. When looked at the right way, the ideas introduced in Simon's algorithm lead naturally to Shor's algorithm for the discrete log problem—which is not a black-box problem! Shor discovered his algorithms (for discrete log and factoring) soon after seeing Simon's algorithm, and in his paper, he acknowledges that he was inspired by Simon's algorithm.

6.1 Definition of Simon's Problem

The problem concerns functions $f : \{0,1\}^n \to \{0,1\}^n$ that are *2-to-1 functions*, which means that, for every point in the range, there are exactly two pre-images. In other words, for every value that the function attains, there are exactly two points, a and b, in the domain that both map to that value. We call such a pair of points a *colliding pair*. If $a \neq b$ and $f(a) = f(b)$ then a and b are a colliding pair.

Now, there's a special property that some 2-to-1 functions have, that we'll call the *Simon property*.

Definition 6.1. *A 2-to-1 function $f : \{0,1\}^n \to \{0,1\}^n$ has the* Simon property *if there exists $r \in \{0,1\}$ such that: For every colliding pair, (a, b), the $a \oplus b = r$.*

For $a, b \in \{0,1\}^n$, $a \oplus b$ denotes their bit-wise XOR. That is, you take the XOR of the first bit of a with the first bit of b, and then you take the XOR of the second bit of a with the second bit of b, and so on.

Let's look at an example. Here's a 2-to-1 function $f : \{0,1\}^3 \to \{0,1\}^3$.

a	$f(a)$
000	011
001	101
010	000
011	010
100	101
101	011
110	010
111	000

Figure 43: Example of function satisfying the Simon property for $n = 3$.

I've color coded the colliding pairs. For example, if you look at the two green points in the domain, 000 and 101, you can see that f maps both of these points to 011 (and those are the only points that are mapped to 011). So the two green points are a colliding pair. Also, the two red points 011 and 100 are a colliding pair, both mapping to 010. And there is a blue colliding pair and a purple colliding pair.

OK, so this f is a 2-to-1 function. But it also has the additional Simon property. If you take any colliding pair (a, b) and then $a \oplus b$ is always the same 3-bit string. Can you see what that string r is in this example?

Did you get $r = 101$? If you pick any color and take the bit wise XOR of the two strings of that color you'll get 101. For example, for the red pair, $011 \oplus 110 = 101$.

Note that, for an arbitrary 2-to-1 function, the bit-wise XORs can be different for different colliding pairs. So the 2-to-1 functions that satisfy the Simon property are special ones, for which these bit-wise XORs are always the same.

Now we can define Simon's problem.

Definition 6.2. *You are given a black box for $f : \{0,1\}^n \to \{0,1\}^n$ which is a 2-to-1 function that has the Simon property, but you are given no other information about f. You don't know what the colliding pairs are and you don't know the value of r. Your goal is to find the value of r. In the example, it would be 101. For a general function $f : \{0,1\}^n \to \{0,1\}^n$, r could potentially be any non-zero n-bit string.*

6.2 Classical query cost of Simon's problem

Let's try to think about how hard this black-box problem is. In the example, we could find r by taking the bit-wise XOR of any colliding pair. So, once we have a colliding

pair, it's easy to find r. Therefore, one way to solve this is to find a colliding pair. But notice that any two distinct n-bit strings *could* be a colliding pair for some r. So if we make a query at some point a, we learn the value of $f(a)$, but we have no idea for which $b \neq a$, you get a colliding pair.

Here's an example of a classical algorithm for Simon's problem. If we a have a budget of m queries, query f at m random points in $\{0,1\}^n$ and then check if there's a collision among them. If any two are a colliding pair then their bit-wise XOR is the answer r. If there are no collisions then the algorithm is out of luck; in that case it fails to find r.

How large should m be set to so that the probability of a collision is at least $\frac{3}{4}$? There are 2^n points in the domain, and each pair of queries will collide with probability $\frac{1}{2^n}$. Since there are around m^2 pairs, the expected number of collisions is roughly m^2 times $\frac{1}{2^n}$. Setting $m \approx \sqrt{2^n}$ suffices to make this expectation a constant[7].

So there's a classical algorithm that solves this with $O\left(\sqrt{2^n}\right)$ queries. It's better than querying f everyhere, which would cost 2^n queries. But the square root just amounts to a reduction by a factor of 2 in the exponent. It's still exponentially many queries. What's interesting is that the above is essentially the best possible classical algorithm.

Theorem 6.1. *Any classical algorithm the solves Simon's problem with worst-case success probability at least $\frac{3}{4}$ must make $\Omega\left(\sqrt{2^n}\right)$ queries.*

How do we know this? We *cannot* deduce this simply based in the fact that this is the best algorithm that we could come up with. How can we be sure that there isn't some clever trick that we haven't discovered? Also, notice that f is not an arbitrary 2-to-1 function. It is a 2-to-1 with the Simon property, which is a very special structure. So it's conceivable that a classical algorithm can somehow take advantage of that structure to find a collision in an unusual way.

Theorem 6.1 is true, but it requires a proof. The proof is not as trivial as the argument that two queries are needed for Deutsch's problem, or that 3 queries are needed for the 1-out-of-4 search problem. It requires a more careful argument. If you're interested in seeing the proof, it is in the next subsection 6.2.1. The proof isn't particularly hard, but it requires some set-up. Feel free to skip past subsection 6.2.1 on a first reading.

[7]You may recognize in this analysis the so-called "birthday paradox", where you consider what the chances are that there are two people in a group of (say) 23 people who have the same birthday. It's 50%, assuming that people's birthdays are uniformly distributed.

6.2.1 Proof of classical lower bound for Simon's problem (Theorem 6.1)

In this section, we prove Theorem 6.1. The proof uses some standard techniques that arise in computational complexity; however, this account assumes no prior background in the area.

The first part of the proof is to "play the adversary" by coming up with a way of generating an instance of f that will be hard for any algorithm. Note that picking some *fixed* f will not work very well. A fixed f has a fixed r associated with it and the first two queries of the algorithm *could* be 0^n and r, which would reveal r to the algorithm after only two queries. Rather, we shall *randomly* generate instances of f. First, we pick r at random, uniformly from $\{0,1\}^n - 0^n$. Picking r does not fully specify f but it partitions $\{0,1\}^n$ into 2^{n-1} colliding pairs of the form $\{x, x \oplus r\}$, for which $f(x) = f(x \oplus r)$ will occur. Let us also specify a representative element form each colliding pair, say, the smallest element of $\{x, x \oplus r\}$ in the lexicographic order. Let T be the set of all such representatives: $T = \{s : s = \min\{x, x \oplus r\} \text{ for some } x \in \{0,1\}^n\}$. Then we can define f in terms of a random one-to-one function $\phi : T \to \{0,1\}^n$ uniformly over all the $2^n(2^n-1)(2^n-2)\cdots(2^n-2^{n-1}+1)$ possibilities. The definition of f can then be taken as

$$f(x) = \begin{cases} \phi(x) & \text{if } x \in T \\ \phi(x \oplus r) & \text{if } x \notin T. \end{cases}$$

We shall prove that no classical probabilistic algorithm can succeed with probability $\frac{3}{4}$ on such instances unless it makes a very large number of queries.

The next part of the proof is to show that, with respect to the above distribution among inputs, we need only consider *deterministic* algorithms (by which we mean ones that make no probabilistic choices). The idea is that any probabilistic algorithm is just a probability distribution over all the deterministic algorithms, so its success probability p is the average of the success probabilities of all the deterministic algorithms (where the average is weighted by the probabilities). At least one deterministic algorithm must have success probability $\geq p$ (otherwise the average would be less than p). Therefore (because we have a fixed probability distribution of the input instances), we need only consider deterministic algorithms.

Next, consider some deterministic algorithm and the first query that it makes: $(x_1, y_1) \in \{0,1\}^n \times \{0,1\}^n$, where x_1 is the input to the query and y_1 is the output of the query. The result of this will just be a uniformly random element of $\{0,1\}^n$, independent of r. Therefore the first query by itself contains absolutely no information about r.

Now consider the second query (x_2, y_2) (without loss of generality, we can assume that the inputs to all queries are different; otherwise, the redundant queries could be eliminated from the algorithm). There are two possibilities: $x_1 \oplus x_2 = r$ (collision) or $x_1 \oplus x_2 \neq r$ (no collision). In the first case, we will have $y_1 = y_2$ and so the algorithm can deduce that $r = x_1 \oplus x_2$. But the first case arises with probability only $\frac{1}{2^n - 1}$. With probability $1 - \frac{1}{2^n - 1}$, we are in the second case, and all that the algorithm deduces about r is that $r \neq x_1 \oplus x_2$ (it has ruled out just one possibility among $2^n - 1$).

We continue our analysis of the process by induction on the number of queries. Suppose that $k - 1$ queries, $(x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$ have been made without any collisions so far. (No collision so far means that, for all $1 \leq i < j \leq k - 1$, $y_i \neq y_j$.) Then all that has been deduced about r is that it is not $x_i \oplus x_j$ for all $1 \leq i < j \leq k-1$. In other words, up to $(k-1)(k-2)/2$ possibilities for r have been eliminated. When the next query (x_k, y_k) is made, the number of potential collisions arising from it are at most $k - 1$ (there are $k - 1$ previously made queries to collide with). Therefore, the probability of a collision at query k is at most

$$\frac{k-1}{2^n - 1 - (k-1)(k-2)/2} \leq \frac{2k}{2^{n+1} - k^2}. \tag{14}$$

Since the collision probability bound in Eq. (14) holds all k, the probability of a collision occurring somewhere among m queries is at most the sum of the right side of Eq. (14) with k varying from 1 to m:

$$\sum_{k=1}^{m} \frac{2k}{2^{n+1} - k^2} \leq \sum_{k=1}^{m} \frac{2m}{2^{n+1} - m^2} \leq \frac{2m^2}{2^{n+1} - m^2}. \tag{15}$$

If this quantity is to be at least $\frac{3}{4}$ then

$$\frac{2m^2}{2^{n+1} - m^2} \geq \frac{3}{4}. \tag{16}$$

It is an easy exercise to solve for m in the above inequality, yielding

$$m \geq \sqrt{\tfrac{6}{11} 2^n}, \tag{17}$$

which gives the desired bound.

Actually, there is a slight technicality remaining. We have shown that $\sqrt{(6/11)2^n}$ queries are necessary *to attain a collision* with probability $\frac{3}{4}$; whereas the algorithm is not technically required to make queries that include a collision. Rather, the algorithm

is just required to deduce r, and it is conceivable that an algorithm could deduce r some other way without a collision occurring. But any algorithm that deduces r can be modified so that it makes one additional query that collides with a previous one. Hence, we have a slightly smaller lower bound of $\sqrt{(6/11)2^n} - 1$, but this is still $\Omega(\sqrt{2^n})$.

6.3 Quantum algorithm for Simon's problem

Before showing you the algorithm for Simon's problem, I'd like to show you a particularly useful way of viewing the multi-qubit Hadamard transform $H \otimes H \otimes \cdots \otimes H$ and the structure of $\{0,1\}^n$.

6.3.1 Understanding $H \otimes H \otimes \cdots \otimes H$

To start with, let's look at how $H \otimes H \otimes \cdots \otimes H = H^{\otimes n}$ (a Hadamard transform on each of n qubits) affects the computational basis states.

First, for the state $|00\ldots0\rangle = |0^n\rangle$,

$$H^{\otimes n} |0^n\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} |b\rangle . \tag{18}$$

It turns out that there's this nice expression for applying $H^{\otimes n}$ to any computational basis state $|a\rangle$ ($a \in \{0,1\}^n$). It's a uniform superposition of the computational basis states, but with certain phases.

Theorem 6.2. *For all $a \in \{0,1\}^n$,*

$$H^{\otimes n} |a\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} |b\rangle , \tag{19}$$

where $a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \bmod 2$.

For example,

$$H \otimes H = \frac{1}{\sqrt{4}} \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 \end{matrix} \\ \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \end{array} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \tag{20}$$

Note that, for each $a, b \in \{0,1\}^2$, the sign of entry (a,b) is $(-1)^{a \cdot b}$.

Proof of Theorem 6.2. The proof is this simple calculation

$$H^{\otimes n}|a\rangle = \left(H|a_1\rangle\right) \otimes \cdots \otimes \left(H|a_n\rangle\right) \tag{21}$$

$$= \left(\tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}(-1)^{a_1}|1\rangle\right) \otimes \cdots \otimes \left(\tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}(-1)^{a_n}|1\rangle\right) \tag{22}$$

$$= \left(\tfrac{1}{\sqrt{2}} \sum_{b_1 \in \{0,1\}} (-1)^{a_1 b_1} |b_1\rangle\right) \otimes \cdots \otimes \left(\tfrac{1}{\sqrt{2}} \sum_{b_n \in \{0,1\}} (-1)^{a_n b_n} |b_n\rangle\right) \tag{23}$$

$$= \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} (-1)^{a_1 b_1} \cdots (-1)^{a_n b_n} |b\rangle \tag{24}$$

$$= \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} (-1)^{a_1 b_1 + \cdots + a_n b_n} |b\rangle . \tag{25}$$

□

So we have a nice expression for applying $H^{\otimes n}$ on computational basis states. In particular, notice how an expression that looks like a dot-product of two n-bit strings (in modulo 2 arithmetic) arises.

6.3.2 Viewing $\{0,1\}^n$ as a discrete vector space

Now let's think about the set $\{0,1\}^n$. It's often useful to associate these strings with mathematical structures, such as the integers $\{0, 1, 2, \ldots, 2^n - 1\}$.

But we can also think of the set $\{0,1\}^n$ as an n-dimensional vector space, where the components of each vector are 0 and 1, and the arithmetic is modulo 2 (which is equivalent to using $\oplus$ for addition and $\wedge$ for multiplication). This is different from a vector space over the field $\mathbb{R}$ or $\mathbb{C}$. But the set $\{0,1\}$, with addition and multiplication modulo 2 (which we'll denoted as $\mathbb{Z}_2$), is a *field*, meaning that it shares some key structural properties that the real and complex numbers have (I won't go into the details of these properties here). It's perfectly valid to have a vector space over a finite field like $\mathbb{Z}_2$. The linear algebra notions of *subspace*, *dimension* and *linear independence*, make perfect sense over such vector spaces.

And this brings us to that dot-product expression that arose in the n-fold tensor product of the Hadamard. For vector spaces over $\mathbb{R}$ and $\mathbb{C}$, the dot-product[8] is an *inner product*, and has useful properties. Two vectors are *orthogonal* if and only if their inner product is zero.

[8] In the case of field $\mathbb{C}$ we need to take complex conjugates one of the vectors in the dot product.

Technically, the dot-product in our finite field $\mathbb{Z}_2$ scenario is *not* an inner product. An inner product has the property that, for any vector v, it hold that $v \cdot v = 0$ if and only if $v = 0$ (the zero vector). In our finite field vector space, there are non-zero binary strings whose inner products with themselves are 0. Can you think of one? Any binary string with an even number of 1s has dot product 0 with itself.

Nevertheless, this dot product does have *some* nice properties. For example, the space can be decomposed into "orthogonal" subspaces whose dimensions add up to n. Two spaces are deemed "orthogonal" if every point in one has dot-product 0 with every point in the other.

Here is a schematic picture of a decomposition of $\{0,1\}^3$ decomposed into a 1-dimensional space and an orthogonal 2-dimensional space.

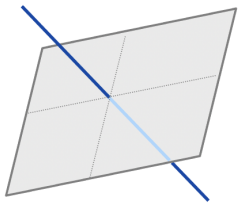

Figure 44: Schematic picture of $\{0,1\}^3$ decomposed into two orthogonal subspaces.

Of course the picture is really for vector spaces over the field $\mathbb{R}$, not the finite field $\mathbb{Z}_2$. So it should just be seen as an intuitive guide, rather than a literal depiction.

A word of caution: for n-qubit systems, there are two spaces in play. One space is the n-dimensional discrete vector space $\{0,1\}^n$ which is over the finite field $\mathbb{Z}_2$. This is associated with the *labels* of the computational basis states. The other space is the 2^n-dimensional space over $\mathbb{C}$, spanned by the 2^n computational basis states (technically, a *Hilbert space*). Do not conflate these different spaces! For example, $00\ldots0$ is the zero vector in the finite vector space. But $|00\ldots0\rangle$ lives in the Hilbert space, and it's definitely not the zero vector. It's a quantum state, which is a vector of length is one.

6.3.3 Simon's algorithm

Applying definition 4.1, the f-queries look like this.

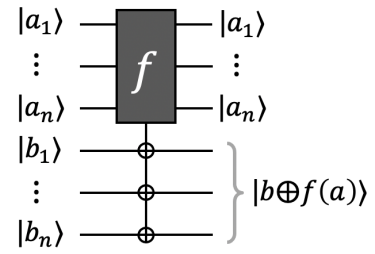

Figure 45: f-query for Simon's Problem.

The f-query acts on $2n$ qubits and, in the computational basis, f of the first n bits is XORed onto the last n bits. With queries like this, it's not so clear how we can "query in the phase", as we did for all the quantum algorithms in section 5.

We'll start out differently, with all the n target qubits just in state $|0\rangle$. But we will put the inputs to f into a uniform superposition of all the n-bit strings. And then we'll perform an f-query.

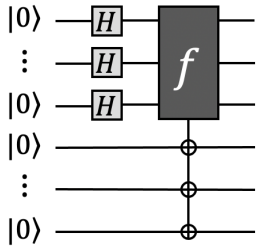

Figure 46: Beginning of Simon's algorithm.

What's the output state of this circuit? The state right after the query is

$$\frac{1}{\sqrt{2^n}} \sum_{a \in \{0,1\}^n} |a\rangle \, |f(a)\rangle \, . \tag{26}$$

Let's consider this state. It's sometimes said that what makes quantum computers so powerful is that they can actually perform several computations at the same time. But this view is misleading. The state was obtained by making just one single query, and it appears to contain information about all of the values of the function f. However, it's not possible to extract more than one value of the function from this state. In particular, if we measure this state in the computational basis then what we end up with is just one pair $(a, f(a))$, where a is sampled from the uniform distribution. And you don't need any quantum devices to generate such a sample. You could just flip a coin n times to create a random a and then perform one query to get $f(a)$.

So how *should* we think about quantum algorithms? With a state like that in Eq. (26), rather than measuring in the computational basis, we can instead—via a unitary operation—measure with respect to *another* basis. In some cases, for a well-chosen basis, we can acquire information about some *global property* of f (*instead of* the value of f at some specific point).

Let's try to find a useful measurement basis for the case where f satisfies the Simon property. The inputs to the function partition into colliding pairs. It helps to look at our example in figure 43 again for reference. It's reproduced here for convenience.

a	$f(a)$
000	011
001	101
010	000
011	010
100	101
101	011
110	010
111	000

Figure 47: Copy of figure 43: A function satisfying the Simon property.

Let's define a set T so as to consist of one element from each colliding pair. In the example, we take one of the two green points, one of the two blue points, one of the two purple points and one of the two red points. Which one we choose won't matter; what's important is that there exists such a set T. In the example, we could set $T = \{000, 100, 010, 011\}$.

Notice that, for each element $a \in T$, the *other* element of the colliding pair is $a \oplus r$. We don't know what r is (that's what we're trying to find by the algorithm). But we know that f satisfies the Simon property and that *there exists* an associated r. (In the example, $r = 101$.)

Therefore, if we combine the elements of T with the elements of the set $T \oplus r$ then we get all of $\{0,1\}^n$. This enables us to rewrite the state in Eq. (26) as

$$\frac{1}{\sqrt{2^n}} \sum_{a \in T} \Big(|a\rangle \, |f(a)\rangle + |a \oplus r\rangle \, |f(a \oplus r)\rangle \Big), \tag{27}$$

where we are only summing over the elements of T, but, for each $a \in T$, we include two terms: one for a and one for $a \oplus r$.

Now, since f satisfies the Simon property with the associated r, for each a, we have that $f(a) = f(a \oplus r)$. That's exactly what the Simon property says. So we can write the expression in Eq. (27) as

$$\tfrac{1}{\sqrt{2^n}} \sum_{a \in T} \big(|a\rangle + |a \oplus r\rangle \big) \, |f(a)\rangle \, . \tag{28}$$

Suppose that we now measure the last n qubits in the computational basis. Then the state of the last n qubits collapses to some value $f(a)$ and the residual state of the first n qubits is a uniform superposition of the pre-images of that value. That is, the state of the first n qubits becomes

$$\tfrac{1}{\sqrt{2}} \, |a\rangle + \tfrac{1}{\sqrt{2}} \, |a \oplus r\rangle \, . \tag{29}$$

for a random $a \in \{0,1\}^n$. What can we do with this state? If we could somehow extract both a and $a \oplus r$ from this state then we could deduce the value of r (by taking their XOR, $a \oplus (a \oplus r) = r$). But we can only measure the state once, after which its state collapses.

If we measure in the computational basis, then we just get either a or $a \oplus r$, neither of which is sufficient to learn anything about the value of r. We can think of measuring in the computational basis this way: we first randomly choose a color (that's what happens when we measure the last n qubits), and then we randomly choose one of the two elements of that color (that's what happens when we measure the first n qubits). So the net effect of all this is to get just a random n-bit string, which is devoid of any information about the structure of f. So, if we want make progress than we should definitely *not* measure the first n qubits in the computational basis.

Something quite remarkable happens if we apply a Hadamard transform to each of the n qubits before measuring in the computational basis. We can calculate the state resulting from the Hadamard transform using Theorem 6.2 as

$$H^{\otimes n}\big(\tfrac{1}{\sqrt{2}} \, |a\rangle + \tfrac{1}{\sqrt{2}} \, |a \oplus r\rangle\big) = \tfrac{1}{\sqrt{2^{n+1}}} \Bigg(\sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} \, |b\rangle + \sum_{b \in \{0,1\}^n} (-1)^{(a \oplus r) \cdot b} \, |b\rangle \Bigg) \tag{30}$$

$$= \tfrac{1}{\sqrt{2^{n+1}}} \Bigg(\sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} + \sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} (-1)^{r \cdot b} \, |b\rangle \Bigg) \tag{31}$$

$$= \tfrac{1}{\sqrt{2^{n+1}}} \Bigg(\sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} \big(1 + (-1)^{r \cdot b} \big) \, |b\rangle \Bigg) . \tag{32}$$

Why is this interesting? Let's think about what happens if we measure this state in the computational basis. Notice that, for each $b \in \{0,1\}^n$,

$$\left(1 + (-1)^{r \cdot b}\right) = \begin{cases} 2 & \text{if } r \cdot b = 0 \\ 0 & \text{if } r \cdot b = 1. \end{cases} \tag{33}$$

Therefore, the probability of each $b \in \{0,1\}^n$ occurring as the outcome is

$$\begin{cases} \frac{1}{2^{n-1}} & \text{if } r \cdot b = 0 \\ 0 & \text{if } r \cdot b = 1. \end{cases} \tag{34}$$

In other words, the outcome of the measurement is a uniformly distributed random b in the orthogonal complement of r (that is, for which $r \cdot b = 0$).

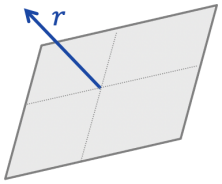

Figure 48: Schematic illustration of $r \in \{0,1\}^n$ and its orthogonal complement.

This b does not produce enough information for us to deduce r, but it reveals *partial information* about r. Namely that the bits of r satisfy the linear equation

$$b_1 r_1 + b_2 r_2 + \cdots + b_n r_n \equiv 0 \pmod 2. \tag{35}$$

And we can acquire more information about r by repeating the procedure.

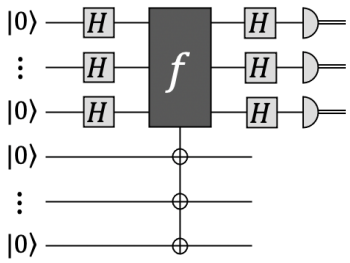

Figure 49: Each execution of this procedure yields a random $b \in \{0,1\}^n$ such that $r \cdot b = 0$.

Each execution of the procedure produces an independent random $b \in \{0,1\}^n$ that is orthogonal to r (in the sense that $r \cdot b = 0$). Suppose that we repeat the process

$n-1$ times (so the number of f-queries is $n-1$). Then, combining the resulting b's, we obtain a system of $n-1$ linear equations (in mod 2 arithmetic)

$$\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,1} & b_{n-1,2} & \cdots & b_{n-1,n} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{36}$$

If the $n-1 \times n$ matrix has rank $n-1$ then there is a unique nonzero r solution to the system, which can be easily found by Guassian elimination. What's the probability that this matrix has full rank? It turns out that this probability is a constant independent of n.

Exercise 6.1. *Show that, if each of the rows of an $n-1 \times n$ matrix is an independent sample form the set of $b \in \{0,1\}^n$ such that $b \cdot r = 0$, then the probability that the matrix has rank $n-1$ is at least $\frac{1}{4}$.*

So we now have a quantum algorithm that makes $n-1$ queries in all and succeeds in finding r with probability at least $\frac{1}{4}$. This fails with probability at most $\frac{3}{4}$. It's easy to reduce the failure probability to $(\frac{3}{4})^5 < \frac{1}{4}$ by repeating the entire procedure five times.

In conclusion, $\Omega(\sqrt{2^n})$ queries are necessary for any classical algorithm to attain success probability $\frac{3}{4}$, whereas order $O(n)$ queries are sufficient for a quantum algorithm to attain success probability $\frac{3}{4}$.

6.4 Significance of Simon's problem

Now, what should we make of Simon's problem and Simon's algorithm?

It's a black-box problem that was specially designed to be very hard for probabilistic classical algorithms and easy for quantum algorithms—thereby improving on previous classical vs quantum query cost separations.

problem	quantum	classical deterministic	classical probabilistic
Deutsch	1	2	2
1-out-of-4 search	1	3	3
Const. vs. balanced	1	$2^{n-1}+1$	$O(1)$
Simon	$O(n^2)$	$\Omega(2^{n/2})$	$\Omega(2^{n/2})$

Figure 50: Summary of query costs for problems considered so far.

But Simon's problem doesn't immediately look like a problem that one would care about in the real world. When Simon's work first came out, people were wondering what to make of it. Although it provided a very strong classical vs. quantum query cost separation, it looked like a contrived problem. Moreover, a contrived *black-box* problem, which is not even a conventional computing problem, involving input data.

This may be one's first impression, but there's actually more to it than that. Look again at the Simon property: for all a, $f(a) = f(a \oplus r)$. This is kind of like a periodicity property of a function, which would be written as: for all a, $f(a) = f(a + r)$. And periodic functions arise naturally in many contexts. We'll soon see that variations of Simon's problem in this direction are very fruitful.