Quantum Information Processing
Quantum Algorithms

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

October 31, 2023

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 some 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 (symbolically denoted as $\neg$).

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

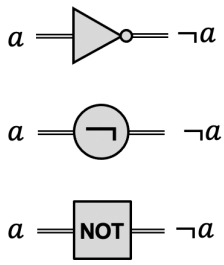

Figure 2: Three notations for the NOT gate.

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 (symbolically denoted as $\wedge$).

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

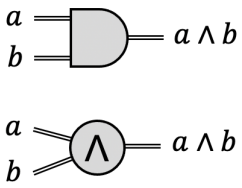

Figure 4: Two notations for the AND gate.

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.

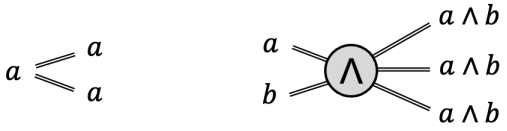

Figure 5: Notation for fan-out: of an input bit (left), and of the output of a gate (right).

What about the logical OR gate and the XOR gate? These are defined as

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

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

Figure 6: Input/output values of the OR and XOR gates (denoted as $\vee$ and $\oplus$, respectively).

We don't need them as our fundamental operations because they can be *simulated* by $\wedge$ and $\vee$ gates. For example, OR can be simuated by three NOT gates and one AND gate using *DeMorgan's Law*

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

We can also simulate an XOR gate in terms of AND and NOT gates, which I leave as an exercise.

Exercise 1.1. *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 7: Table of input/output values of the MAJ_3 function.

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

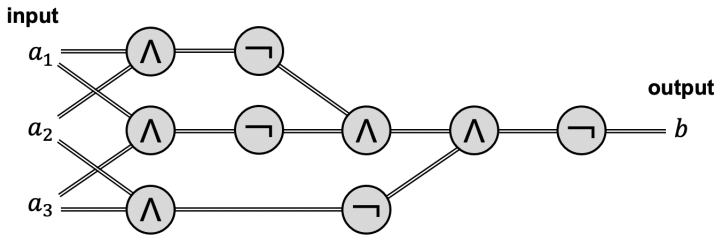

Figure 8: Classical circuit computing the majority value of three bits.

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

$$\mathsf{MAJ}_3(a_1, a_2, a_3) = (a_1 \wedge a_2) \vee (a_1 \wedge a_3) \vee (a_2 \wedge a_3)\,, \tag{2}$$

where the OR gates are simulated by NOT and AND gates along the lines of Eq. (1). 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 $\mathsf{MAJ}_n(a_1, a_2, \ldots, a_n)$, the majority value of $(a_1, a_2, \ldots, a_n)$, for odd $n > 3$? What is the gate count of your construction as a function of n? 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* $\mathsf{MAJ}_n(a_1, a_2, \ldots, a_n)$ *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.

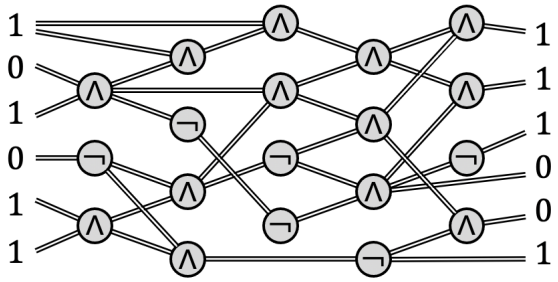

Figure 9: Generic form of a classical circuit (information flows from left to right).

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 (5 in binary) and 111 (7 in binary), the output should be 100011 (35 in binary) 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 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 $O(n^{1.585})$ to $O(n \log n \log \log n)$ 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. To get a feel for this, try factoring the number 91 (or show that it's 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.

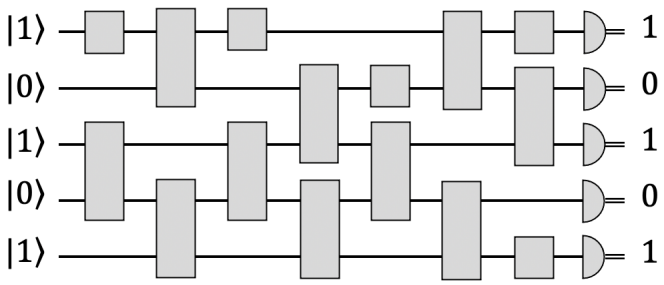

Figure 10: Generic form of a quantum circuit.

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 polynomially bounded 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.

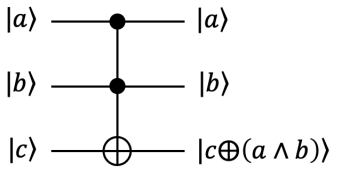

Figure 11: Toffoli gate acting on computational basis states ($a, b, c \in \{0, 1\}$).

On computational basis states, it flips the third qubit if and only if *both* of the first two qubits are in state $|1\rangle$; otherwise it does nothing. The 8×8 unitary matrix for this 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.

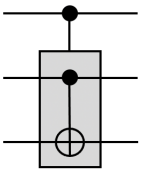

Figure 12: Toffoli gate is a controlled-CNOT gate (also a controlled-controlled-NOT gate).

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 13.

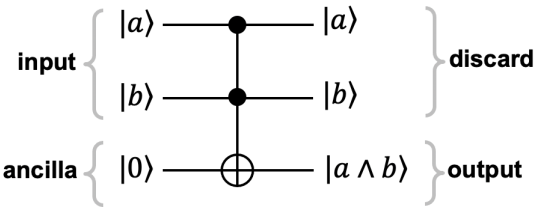

Figure 13: Using a Toffoli gate to simulate a classical AND gate.

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 13. 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.

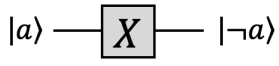

Figure 14: Using an X-gate to simulate a classical NOT gate.

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 15.

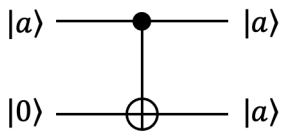

Figure 15: Using a CNOT gate to simulate a classical fan-out gate.

□

Remember that circuit in figure 8 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.

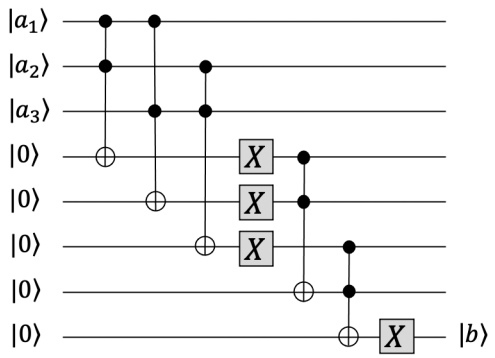

Figure 16: Computing the majority of three bits using Toffoli and X gates.

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 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.

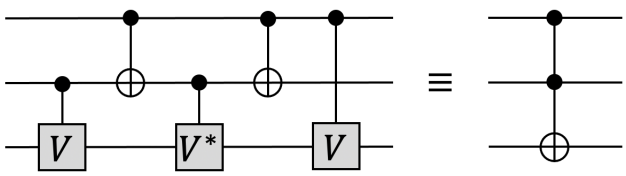

Figure 17: Simulating a Toffoli gate in terms of 2-qubit gates.

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 17 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. First, it avoids a tedious calculation. Second, 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

[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).

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.

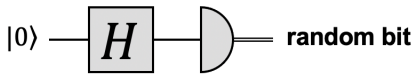

Figure 18: Using a Hadamard and a measurement gate to generate a classical random bit.

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.

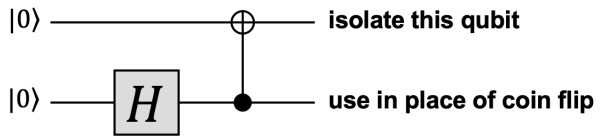

Figure 19: Simulating random bits without using measurements.

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}).

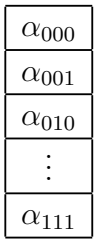

Figure 20: 2^n-dimensional array storing the amplitudes of state $\sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$.

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 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

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

circuit, the result is not very efficient. It's exponential and in fact 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 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

In 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 $f : \{0,1\}^* \to \{0,1\}$, where $*$ denotes the Kleene closure[3] in this context). 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[4] (for some constant c).

BPP (bounded-error probabilistic polynomial time)
Solvable by $O(n^c)$-size probabilistic classical circuits whose worst-case error probability[5] 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] More specifically, $\{0,1\}^* = \varnothing \cup \{0,1\} \cup \{0,1\}^2 \cup \{0,1\}^3 \cup \cdots$.

[4] Technically, we require *uniform circuit families*, one for each input size n, with an algorithmic way of mapping n to the circuit for size n.

[5] 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, in this section, 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 this is our *only* way of acquiring information about f. Such a device is commonly called a *black-box* for f.

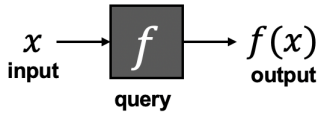

Figure 21: A gate performing an f-query.

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 22: 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 (linear interpolation). If $f : \mathbb{R} \to \mathbb{R}$ is quadratic then it turns out that 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 23: 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 23); or
- $f(0) \neq f(1)$ (the last two cases in figure 23).

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 21. 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 23); 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 24.

Figure 24: 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 23, here are the corresponding mappings on computational basis states.

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

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

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

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

Figure 25: 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.

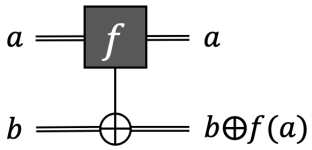

Figure 26: Reversible classical f-query (for $f : \{0,1\} \to \{0,1\}$).

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 $f : \{0,1\} \to \{0,1\}$).

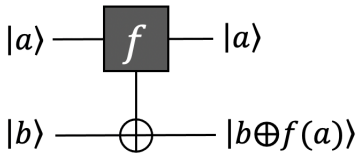

Figure 27: Quantum f-query (for $f : \{0,1\} \to \{0,1\}$).

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[6] *XOR between the m-bit strings b and $f(a)$.*

Here is notation for a general quantum f-query.

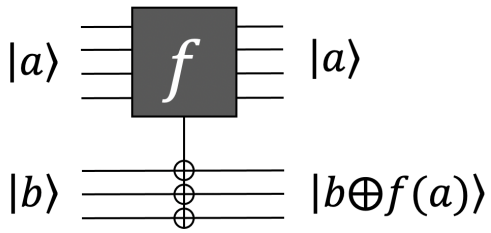

Figure 28: Quantum f-query (for $f : \{0,1\}^n \to \{0,1\}^m$).

[6]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's 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 23). 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.

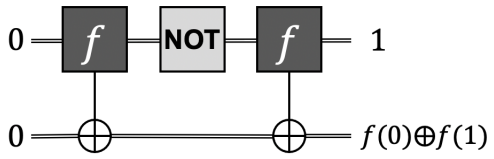

Figure 29: Classical algorithm for Deutsch that makes two reversible classical queries.

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 29. 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!

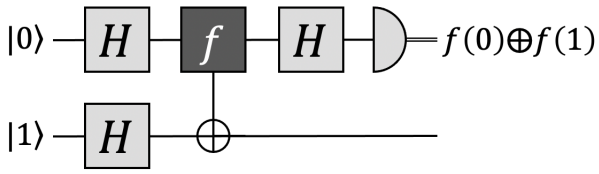

Figure 30: Quantum algorithm for Deutsch that makes just one quantum query.

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.

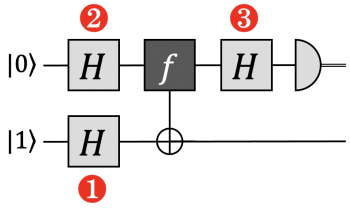

Figure 31: The three separate Hadamard transforms.

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[7] $\frac{1}{\sqrt{2}}|1\rangle - \frac{1}{\sqrt{2}}|0\rangle = -|-\rangle$.

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.

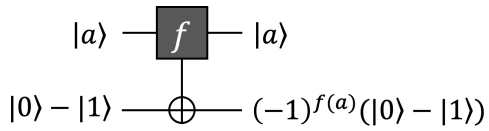

Figure 32: Caption.

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.

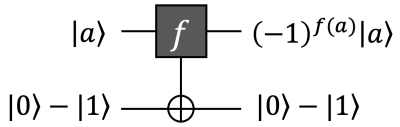

Figure 33: Caption.

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.

[7]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 $(-1)^{f(1)}$.

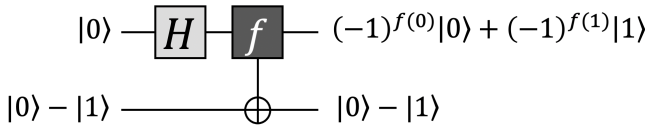

Figure 34: Caption.

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 23. 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.

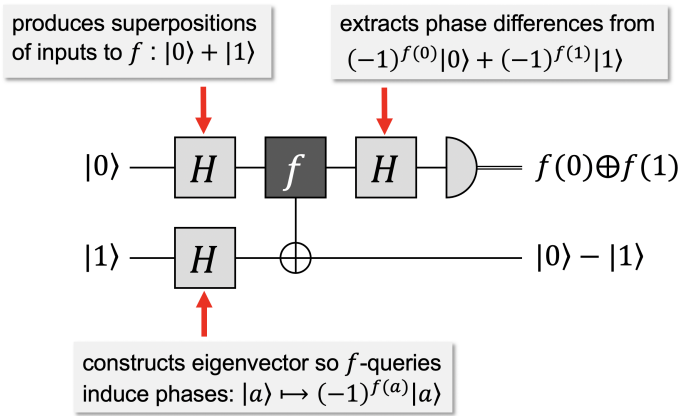

Figure 35: Summary of the role that each H transformation plays in the algorithm.

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 36: 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.

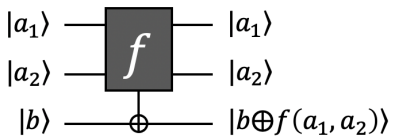

Figure 37: Query gate for a function $f : \{0,1\}^2 \to \{0,1\}$.

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.

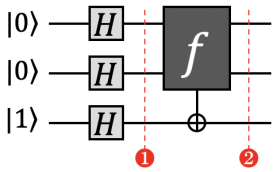

Figure 38: Starting along the lines of Deutsch's algorithm. (Intermediate stages are marked.)

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.

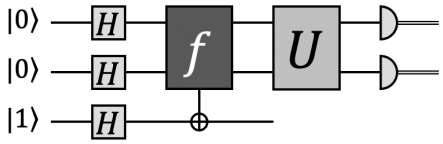

Figure 39: Quantum algorithm for the 1-out-of-4 search problem.

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 40: 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.

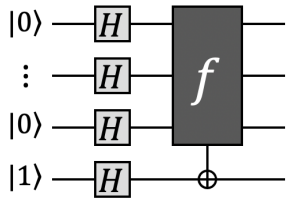

Figure 41: Starting off in a manner similar to the two previous algorithms.

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 a sum of 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 $\pm\,|00\ldots0\rangle = \pm\,|0^n\rangle$. 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.

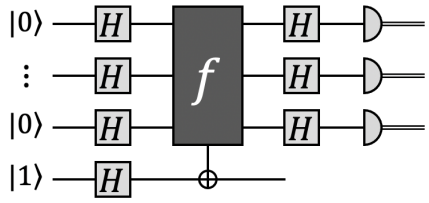

Figure 42: Quantum algorithm for the 1-out-of-4 search problem.

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 a constant that's arbitrarily close to 1, using 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 43: 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 $r \in \{0,1\}^n$ such that: For every colliding pair (a, b), it holds that $a \oplus b = r$.*

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 44: 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 (Simon's problem)**.** *You are given access to a black-box for a 2-to-1 function $f : \{0,1\}^n \to \{0,1\}^n$ 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 above example, $r = 101$. 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[8].

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 that 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

[8]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.

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.

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 $2^n(2^n-1)(2^n-2)(2^n-3)\cdots(2^n-2^{n-1}+1)$ 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 (i.e., for all $1 \leq i < j \leq k-1$, $y_i \neq y_j$). Then all that has been deduced about r is that $r \neq x_i \oplus x_j$, for all $1 \leq i < j \leq k-1$. Therefore, at most $\binom{k-1}{2} = (k-1)(k-2)/2$ possibilities for r have been eliminated (up to one value of r is eliminated for each pair of x_i and x_j). Since there are $2^n - 1$ values of r to begin with (recall that $r \neq 0^n$), it follows that there are at least $2^n - 1 - (k-1)(k-2)/2$ possibilities of r that have not yet been eliminated. When the next query (x_k, y_k) is made, the number of potential collisions arising from it are at most $k-1$ (the number of previous queries). Therefore, the probability of a collision at query k is at most

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

Let's review the expression on the left side of Eq. (14). For the denominator, there are at least $2^n - 1 - (k-1)(k-2)/2$ possibilities of r remaining at the point of query k (assuming that no collisions have occurred yet). And, for the numerator, among those remaining values of r, there are $k-1$ values of r that cause a collision to occur for query x_k, namely the values in the set $\{x_k \oplus x_1,\ x_k \oplus x_2,\ \ldots,\ x_k \oplus x_{k-1}\}$.

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 $m \geq \sqrt{(6/11)2^n}$, 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. The algorithm is only required to deduce r, and it's 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. So 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 $|00\ldots0\rangle = |0^n\rangle$,

$$H^{\otimes n} |0^n\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} |b\rangle . \tag{17}$$

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 $a \in \{0,1\}^n$,*

$$H^{\otimes n} |a\rangle = \tfrac{1}{\sqrt{2^n}} \sum_{b \in \{0,1\}^n} (-1)^{a \cdot b} |b\rangle , \tag{18}$$

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

For example,

$$H \otimes H = \frac{1}{\sqrt{4}} \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 \end{matrix} \\ \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \end{array} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \tag{19}$$

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

$$H^{\otimes n} |a\rangle = \left(H |a_1\rangle\right) \otimes \cdots \otimes \left(H |a_n\rangle\right) \tag{20}$$

$$= \left(\tfrac{1}{\sqrt{2}} |0\rangle + \tfrac{1}{\sqrt{2}}(-1)^{a_1} |1\rangle\right) \otimes \cdots \otimes \left(\tfrac{1}{\sqrt{2}} |0\rangle + \tfrac{1}{\sqrt{2}}(-1)^{a_n} |1\rangle\right) \tag{21}$$

$$= \left(\tfrac{1}{\sqrt{2}} \sum_{b_1 \in \{0,1\}} (-1)^{a_1 b_1} |b_1\rangle\right) \otimes \cdots \otimes \left(\tfrac{1}{\sqrt{2}} \sum_{b_n \in \{0,1\}} (-1)^{a_n b_n} |b_n\rangle\right) \tag{22}$$

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

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

□

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[9] is an *inner product*, and has useful properties. Two vectors are *orthogonal* if and only if their inner product is zero.

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 holds 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.

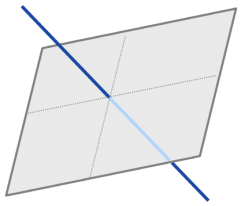

Figure 45: Schematic picture of $\{0,1\}^3$ decomposed into two orthogonal subspaces.

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

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

in the Hilbert space, and it's definitely not the zero vector. It's a quantum state, which is a vector of length one.

6.3.3 Simon's algorithm

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

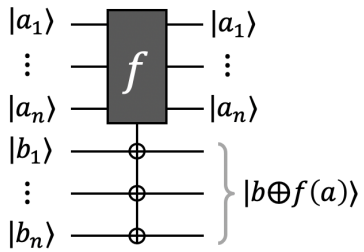

Figure 46: f-query for Simon's Problem.

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.

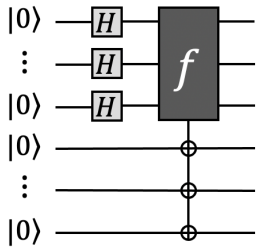

Figure 47: Beginning of Simon's algorithm.

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

$$\frac{1}{\sqrt{2^n}} \sum_{a \in \{0,1\}^n} |a\rangle \, |f(a)\rangle \, . \tag{25}$$

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. (25), 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 44 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 48: Copy of figure 44: 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. (25) as

$$\tfrac{1}{\sqrt{2^n}} \sum_{a \in T} \Big(|a\rangle \, |f(a)\rangle + |a \oplus r\rangle \, |f(a \oplus r)\rangle \Big), \tag{26}$$

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. (26) as

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

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

$$\tfrac{1}{\sqrt{2}} \, |a\rangle + \tfrac{1}{\sqrt{2}} \, |a \oplus r\rangle \, . \tag{28}$$

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

$$H^{\otimes n}\left(\tfrac{1}{\sqrt{2}}|a\rangle + \tfrac{1}{\sqrt{2}}|a \oplus r\rangle\right) = \tfrac{1}{\sqrt{2^{n+1}}}\left(\sum_{b\in\{0,1\}^n}(-1)^{a\cdot b}|b\rangle + \sum_{b\in\{0,1\}^n}(-1)^{(a\oplus r)\cdot b}|b\rangle\right) \tag{29}$$

$$= \tfrac{1}{\sqrt{2^{n+1}}}\left(\sum_{b\in\{0,1\}^n}(-1)^{a\cdot b}|b\rangle + \sum_{b\in\{0,1\}^n}(-1)^{a\cdot b}(-1)^{r\cdot b}|b\rangle\right) \tag{30}$$

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

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

$$\left(1+(-1)^{r\cdot b}\right) = \begin{cases} 2 & \text{if } r\cdot b = 0 \\ 0 & \text{if } r\cdot b = 1. \end{cases} \tag{32}$$

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

$$\begin{cases} \frac{1}{2^{n-1}} & \text{if } r\cdot b = 0 \\ 0 & \text{if } r\cdot b = 1. \end{cases} \tag{33}$$

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

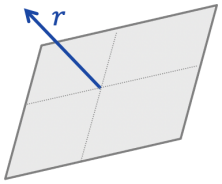

Figure 49: Schematic illustration of $r \in \{0,1\}^n$ and its orthogonal complement.

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

$$b_1 r_1 + b_2 r_2 + \cdots + b_n r_n \equiv 0 \pmod 2. \tag{34}$$

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

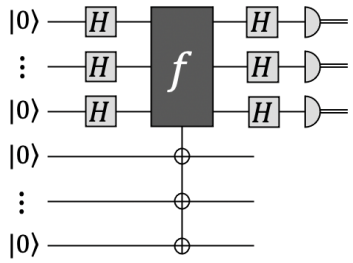

Figure 50: Each execution of this procedure yields a random $b \in \{0,1\}^n$ such that $r \cdot b = 0$.

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)

$$\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,1} & b_{n-1,2} & \cdots & b_{n-1,n} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{35}$$

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 51: 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.

7 The Fourier transform

Up until now, the Hadamard transform $H = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$ (acting on qubits) has played a prominent role in quantum algorithms. There is a natural generalization of H to m-dimensional systems, called the *Fourier transform*, which is also very useful for quantum algorithms.

7.1 Definition of the Fourier transform modulo m

We begin by considering complex numbers the form

$$\omega = e^{2\pi i/m}, \tag{36}$$

which we refer to as *(primitive) m^{th} roots of unity*. Clearly, $\omega^m = 1$. Here's where ω, and all its powers, lie in the complex plane $\mathbb{C}$.

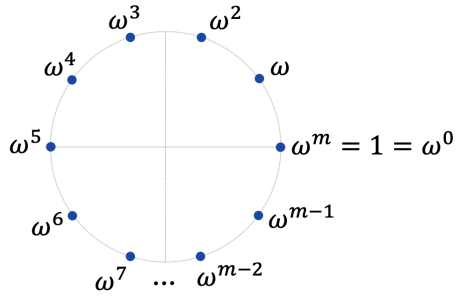

Figure 52: The powers of ω are m equally spaces points on the unit circle in $\mathbb{C}$.

If we sum all these powers of ω, we get zero: $1 + \omega + \omega^2 + \cdots + \omega^{m-1} = 0$. Can you see why? Moreover, if we sum all the powers of ω^k (assuming $1 \leq k \leq m-1$) we also get zero.

Exercise 7.1 (Hint: use formula for sum of geometric sequence)**.** *Prove that, for all $k \in \{1, 2, \ldots, m-1\}$, it holds that*

$$1 + \omega^k + \omega^{2k} + \cdots + \omega^{(m-1)k} = \sum_{j=0}^{m-1} \omega^{jk} = 0. \tag{37}$$

What about the powers of ω^m? Since $\omega^m = 1$, it's obvious that $\sum_{j=0}^{m-1} \omega^{jm} = m$.

Now we can define the Fourier transform modulo m.

Definition 7.1 (Fourier transform)**.** *The Fourier transform modulo m is the $m \times m$ matrix*

$$F_m = \frac{1}{\sqrt{m}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{m-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{m-1} & \omega^{2(m-1)} & \cdots & \omega^{(m-1)^2} \end{bmatrix}. \tag{38}$$

Note that (after the normalization factor $\frac{1}{\sqrt{m}}$) the first column is 1s, the second column is the powers of ω, the third column is powers of ω^2 and so on.

Exercise 7.2. *Prove that F_m is unitary.*

Exercise 7.3. *Prove that the inverse Fourier transform F_m^* is*

$$F_m^* = \frac{1}{\sqrt{m}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(m-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(m-1)} & \omega^{-2(m-1)} & \cdots & \omega^{-(m-1)^2} \end{bmatrix}. \tag{39}$$

For all $a \in \mathbb{Z}_m$, applying the Fourier transform F_m to the computational basis state $|a\rangle$ results in the state

$$F_m |a\rangle = \tfrac{1}{\sqrt{m}} \sum_{j=0}^{m-1} \omega^{ja} |j\rangle, \tag{40}$$

which corresponds to column a of F_m. We call this a *Fourier basis state*. Similarly, applying the inverse Fourier transform F_m^* to $|a\rangle$ results in the state

$$F_m^* |a\rangle = \tfrac{1}{\sqrt{m}} \sum_{j=0}^{m-1} \omega^{-ja} |j\rangle. \tag{41}$$

If there are n registers that are each m-dimensional, and you apply F_m to each register (which is $F_m \otimes F_m \otimes \cdots \otimes F_m = F_m^{\otimes n}$ on the n-register system) then, for all $(a_1, a_2, \ldots, a_n) \in \mathbb{Z}_m^n$,

$$(F_m)^{\otimes n} |a_1, a_2, \ldots, a_n\rangle = \frac{1}{\sqrt{m^n}} \sum_{b \in \mathbb{Z}_m^n} \omega^{a \cdot b} |b_1, b_2, \ldots, b_n\rangle \tag{42}$$

$$(F_m^*)^{\otimes n} |a_1, a_2, \ldots, a_n\rangle = \frac{1}{\sqrt{m^n}} \sum_{b \in \mathbb{Z}_m^n} \omega^{-a \cdot b} |b_1, b_2, \ldots, b_n\rangle, \tag{43}$$

where $a \cdot b = (a_1, a_2, \ldots, a_n) \cdot (b_1, b_2, \ldots, b_n) = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \bmod m$.

7.2 A very simple application of the Fourier transform

We will soon see, in section 9, that it's possible to efficiently solve the celebrated *discrete log problem* (which will be defined in section 8) by reducing it to a generalization of Simon's problem (section 9.2), where the modulus is m (as opposed to 2). This generalization can be solved along the lines of Simon's algorithm—using Fourier transforms in place of Hadamard transforms.

In this section, we consider a very simple query problem where a quantum algorithm that employs the Fourier transform outperforms what any classical query algorithm can do. The reduction in query cost is modest: the quantum algorithm makes one f-query; whereas any classical algorithm must make two f-queries. The purpose of this example is not to demonstrate a dramatic efficiency improvement by a quantum algorithm. Rather, it is to show the Fourier transform in action in a simple setting, where some of its interesting properties are easy to observe and analyze.

Definition 7.2 (Linear coefficient problem)**.** *Let $m \geq 2$. Let $a, b \in \mathbb{Z}_m$ and define $f : \mathbb{Z}_m \to \mathbb{Z}_m$ as*

$$f(x) = ax + b \bmod m, \tag{44}$$

for all $x \in \mathbb{Z}_m$. Assume that you are given access to a black-box for the function f, but you are given no other information about f. You don't know what the linear coefficient a is, nor the additive constant b. Your goal is to find the value of the linear coefficient a.

It's not hard to see that, in the special case where $m = 2$, this is equivalent to Deutsch's problem (section 5.1). In that case, the four functions are all of the form of Eq. (44) and $f(0) = f(1)$ if and only if $a = 0$.

Moreover, it is straightforward to show that, for all $m \geq 2$, any classical algorithm for the linear coefficient problem problem must make at least two f-queries.

Exercise 7.4. *Show that, for classical algorithms,* two *f-queries are necessary and sufficient to solve the linear coefficient problem.*

Next, we will see a quantum algorithm that solves this problem with only one quantum f-query.

We need to define the notion of a *quantum* f-query for functions of the form $f : \mathbb{Z}_m \to \mathbb{Z}_m$. A reasonable definition is as the unitary operation that maps each basis state $|x\rangle\,|y\rangle$ to

$$|x\rangle\,|y+f(x) \bmod m\rangle\,, \tag{45}$$

for all $x, y \in \mathbb{Z}_m$. Here is notation for a quantum f-query (which makes sense for *any* $f : \mathbb{Z}_m \to \mathbb{Z}_m$).

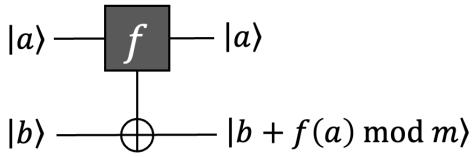

Figure 53: Quantum f-query (for $f : \mathbb{Z}_m \to \mathbb{Z}_m$).

Note the similarity with figure 27 (for the case where $f : \{0,1\} \to \{0,1\}$).

Given the resemblance of the leading coefficient problem to Deutsch's problem, we can draw inspiration from the quantum algorithm for that problem (section 5.1), substituting F_m and F_m^* for Hadamard transforms. In fact, our quantum algorithm will be the following quantum circuit.

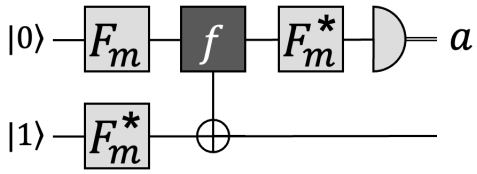

Figure 54: Quantum algorithm for the linear coefficient problem.

Note the resemblance to the quantum circuit in figure 30; when $m = 2$, the two quantum circuits are identical. Why is the measured output of this quantum circuit the linear coefficient a? We will carefully go through then role that each of the Fourier transforms plays in the computation.

In figure 30, the Hadamard transform on the second register creates the special state $|-\rangle$ in the target register of the f-query, so that f-queries have the effect of querying in the phase: $|x\rangle \mapsto (-1)^{f(x)}\,|x\rangle$ (for $x \in \{0,1\}$). An analogue of querying in the phase for $\mathbb{Z}_m$-valued registers is to implement a mapping of the form

$$|x\rangle \mapsto \omega^{f(x)}\,|x\rangle \tag{46}$$

(for $x \in \mathbb{Z}_m$), where $\omega = e^{2\pi i/m}$. This is achieved by setting the second register to the state

$$|\psi\rangle = F_m^* |1\rangle \tag{47}$$

$$= \frac{1}{\sqrt{m}} \sum_{b=0}^{m-1} \omega^{-b} |b \bmod m\rangle . \tag{48}$$

To see why this causes f-queries to implement the mapping in Eq. (46), first define a mod m generalization of the unitary operation X as the m-dimensional unitary operation that, in the computational basis, adds 1 modulo m to a register. Let's call this the *increment modulo m* operation and denote it as X_m. For all $a \in \mathbb{Z}_m$,

$$X_m |a\rangle = |a + 1 \bmod m\rangle . \tag{49}$$

The effect of X_m on $|\psi\rangle$ (the state defined in Eq. (47)) is

$$X_m |\psi\rangle = X_m \left(\tfrac{1}{\sqrt{m}} \sum_{b=0}^{m-1} \omega^{-b} |b\rangle \right) \tag{50}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{b=0}^{m-1} \omega^{-b} |b + 1 \bmod m\rangle \tag{51}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{c=0}^{m-1} \omega^{-(c-1)} |c \bmod m\rangle \tag{52}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{c=0}^{m-1} \omega\omega^{-c} |c \bmod m\rangle \tag{53}$$

$$= \omega |\psi\rangle . \tag{54}$$

More generally, we have $(X_m)^k |\psi\rangle = \omega^k |\psi\rangle$ (in other words, adding k modulo m to state $|\psi\rangle$ results in state $\omega^k |\psi\rangle$). From this it follows that an f-query applied to state $|x\rangle |\psi\rangle$ produces the state $\omega^{f(x)} |x\rangle |\psi\rangle$, thereby implementing a mapping of the form in Eq. (46) (with respect to the first register).

Following the outline of Deutsch's algorithm (section 5.1) as a guide, the Fourier transform on the first qubit creates the superposition

$$F_m |0\rangle = \tfrac{1}{\sqrt{m}} \big(|0\rangle + |1\rangle + \cdots + |m-1\rangle \big) \tag{55}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{x=0}^{m-1} |x\rangle . \tag{56}$$

Applying an f-query to this state (with the second register set to $|\psi\rangle$), results in the state

$$\frac{1}{\sqrt{m}} \sum_{x=0}^{m-1} \omega^{f(x)} |x\rangle = \frac{1}{\sqrt{m}} \sum_{x=0}^{m-1} \omega^{ax+b} |x\rangle \tag{57}$$

$$= \omega^b \left(\frac{1}{\sqrt{m}} \sum_{x=0}^{m-1} \omega^{ax} |x\rangle \right) \tag{58}$$

$$= \omega^b F_m |a\rangle . \tag{59}$$

Since this state is $F_m |a\rangle$ (with a global phase of ω^b), applying F_m^* to the first register after the query results in the state $\omega^b |a\rangle$. Applying a measurement to this first register results in a, as required. Therefore, the quantum circuit in figure 54 does indeed solve the linear coefficient problem.

7.3 Computing the Fourier transform modulo 2^n

The Fourier transform F_m is defined in section 7.1. We're interested in computing F_m efficiently by a quantum circuit consisting of elementary operations acting on qubits. There's an elegant way to do this in the case where $m = 2^n$. Note that F_{2^n} is a unitary operation acting on n qubits. I will show you a fairly simple quantum circuit consisting of $O(n^2)$ gates that computes F_{2^n}.

It's useful to visualize how the powers of a (primitive) 2^n-th root of unity ω are aligned in the complex plane. They are 2^n equally spaced points on the unit circle, and here's what they look like when $n = 3$.

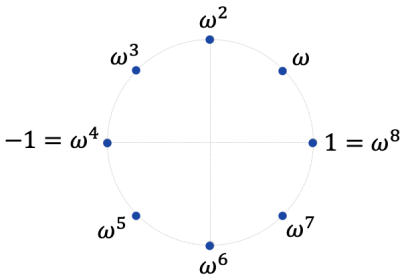

Figure 55: Powers of ω, a primitive 8-th root of unity in $\mathbb{C}$.

For a 2^n-th root of unity ω, a couple of simple but valuable observations are:

1. $\omega^{2^{n-1}} = -1$. (For example, in figure 55, ω is an 8-th root of unity and $\omega^4 = -1$.)
2. ω^2 is a 2^{n-1}-th root of unity. (In figure 55, ω^2 is an 4-th root of unity.)

7.3.1 Expressing F_{2^n} in terms of $F_{2^{n-1}}$

To get a feeling for how this works, let's first consider the case where $n = 3$. For each $a \in \{0,1\}^3 \equiv \{0,1,2,3,4,5,6,7\}$, the corresponding Fourier basis state $F_8 |a\rangle$ is

$$|000\rangle + \omega^a|001\rangle + \omega^{2a}|010\rangle + \omega^{3a}|011\rangle + \omega^{4a}|100\rangle + \omega^{5a}|101\rangle + \omega^{6a}|110\rangle + \omega^{7a}|111\rangle\,.$$

(where the normalization factor $\frac{1}{\sqrt{8}}$ is omitted). Question: Are these three qubits entangled or in a product state? In fact, this state can be written as

$$\begin{aligned} &|0\rangle \otimes \big(|00\rangle + \omega^a|01\rangle + \omega^{2a}|10\rangle + \omega^{3a}|11\rangle\big) \\ &\qquad + \omega^{4a}|1\rangle \otimes \big(|00\rangle + \omega^a|01\rangle + \omega^{2a}|10\rangle + \omega^{3a}|11\rangle\big) \qquad (60) \\ &= \big(|0\rangle + \omega^{4a}|1\rangle\big) \otimes \big(|00\rangle + \omega^a|01\rangle + \omega^{2a}|10\rangle + \omega^{3a}|11\rangle\big) \qquad (61) \\ &= \big(|0\rangle + \omega^{4a}|1\rangle\big) \otimes \big(|0\rangle + \omega^{2a}|1\rangle\big) \otimes \big(|0\rangle + \omega^a|1\rangle\big). \qquad (62) \end{aligned}$$

So $F_8 |a\rangle$ is a product of three 1-qubit states.

This generalizes to $F_{2^n} |a\rangle$, for all $n \geq 1$, as follows.

Lemma 7.1. *For all $n \geq 1$ and $a \in \{0,1\}^n$,*

$$F_{2^n}|a\rangle = \big(|0\rangle + \omega^{2^{n-1}a}|1\rangle\big) \otimes \cdots \otimes \big(|0\rangle + \omega^{4a}|1\rangle\big) \otimes \big(|0\rangle + \omega^{2a}|1\rangle\big) \otimes \big(|0\rangle + \omega^a|1\rangle\big), \quad (63)$$

where there is a normalization factor of $\frac{1}{2^{n/2}}$.

Our quantum circuit for computing the Fourier transform will make use of this structure. Remember that, if we compute a linear operator that matches the Fourier transform on all computational basis states then it must be the Fourier transform.

Let's return our attention to the $n = 3$ case, where the Fourier basis states are

$$F_8 |a\rangle = \big(|0\rangle + \omega^{4a}|1\rangle\big) \otimes \big(|0\rangle + \omega^{2a}|1\rangle\big) \otimes \big(|0\rangle + \omega^a|1\rangle\big), \quad (64)$$

for all $a \in \{0,1\}^3$. Each $a = a_2 a_1 a_0$ denotes an element of $\{0,1,2,3,4,5,6,7\}$ in the usual way (a_0 is the low-order bit and a_2 is the high-order bit).

Consider the state of the first qubit in Eq. (64). Since $\omega^4 = -1$, the state of the first qubit simplifies to $|0\rangle + (-1)^a |1\rangle$. Note that this is $|+\rangle$ when a is even and $|-\rangle$ when a is odd. The parity of a is determined by its low-order bit a_0. Therefore the first qubit of $F_8 |a\rangle$ is simply $H |a_0\rangle$.

What about the remaining qubits? Let $|\psi\rangle$ denote the second and third qubit in Eq. (64). That is,

$$|\psi\rangle = \big(|0\rangle + \omega^{2a}|1\rangle\big) \otimes \big(|0\rangle + \omega^a|1\rangle\big). \quad (65)$$

Now, please look at the $n = 2$ case of the expression in Eq. (63) in Lemma 7.1. Does $|\psi\rangle$ look like $F_4 |a\rangle$?

There is certainly a superficial resemblance; however, $|\psi\rangle$ and $F_4 |a\rangle$ are not *exactly* the same. The state $F_4 |a\rangle$ is with respect to a 4-th root of unity—not an 8-th root of unity. Another difference is that F_4 acts on 2 qubits; whereas the parameter a in Eq. (65) is a 3-bit integer.

It will be fruitful to explore in more detail the difference between $|\psi\rangle$ and $F_4 |a\rangle$. Let's see what these states look like in terms of the digits $a_2 a_1 a_0$ of the binary representation of a. We'll use the Greek letter ϖ to denote the 4-th root of unity, while reserving ω for the 8-th root of unity.

If we apply F_4 to $|a_2 a_1\rangle$ (the two higher order digits of a), the result is

$$F_4 |a_2 a_1\rangle = \big(|0\rangle + \varpi^{2[a_2 a_1]} |1\rangle\big) \otimes \big(|0\rangle + \varpi^{[a_2 a_1]} |1\rangle\big) \tag{66}$$

$$= \big(|0\rangle + (\omega^2)^{2[a_2 a_1]} |1\rangle\big) \otimes \big(|0\rangle + (\omega^2)^{[a_2 a_1]} |1\rangle\big) \tag{67}$$

$$= \big(|0\rangle + \omega^{2[a_2 a_1 0]} |1\rangle\big) \otimes \big(|0\rangle + \omega^{[a_2 a_1 0]} |1\rangle\big). \tag{68}$$

In the exponent, I've surrounded the binary representations by square brackets so that they can be clearly read; the two-digit number in the square brackets is either 0, 1, 2, or 3. Eq. (66) is due to Lemma 7.1. Eq. (67) is due to $\varpi = \omega^2$. And Eq. (68) is due[10] to $2[a_2 a_1] = [a_2 a_1 0]$.

Now let's express $|\psi\rangle$ in terms of $[a_2 a_1 a_0]$. This is

$$|\psi\rangle = \big(|0\rangle + \omega^{2[a_2 a_1 a_0]} |1\rangle\big) \otimes \big(|0\rangle + \omega^{[a_2 a_1 a_0]} |1\rangle\big) \tag{69}$$

$$= \big(|0\rangle + \omega^{2[a_2 a_1 0]} \omega^{2a_0} |1\rangle\big) \otimes \big(|0\rangle + \omega^{[a_2 a_1 0]} \omega^{a_0} |1\rangle\big). \tag{70}$$

Comparing Eq. (68) with Eq. (70), we can see precisely where they differ: the factors ω^{2a_0} and ω^{a_0} (highlighted in red). We'll refer to these as *phase corrections*.

Based on the above observations, let's try to compute $F_8 |a_2 a_1 a_0\rangle$ in terms of $F_4 |a_2 a_1\rangle$ and $H |a_0\rangle$, combined with additional gates that perform phase corrections. To perform the phase corrections, we introduce the following new 2-qubit gate.

Definition 7.3 (controlled-phase gate)**.** *For any $r \in \mathbb{Z}$, the 2-qubit* controlled-phase gate (with phase $e^{2\pi i/r}$) *is defined as the unitary operation that, for all $a, b \in \{0, 1\}$,*

[10] This is a simple maneuver. It's the binary equivalent of what we do in base ten when we multiply an integer by ten: we add a zero digit and shift all the other digits left. 10 times 23 is 230.

maps $|a\rangle\,|b\rangle$ *to* $(e^{2\pi i/r})^{ab}\,|a\rangle\,|b\rangle$*. The unitary matrix for the gate is*

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{2\pi i/r} \end{bmatrix} \tag{71}$$

and our circuit notation for this gate is shown in figure 56.

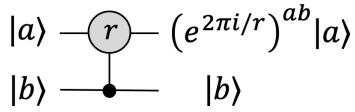

Figure 56: Our notation for the controlled-phase gate, with phase $e^{2\pi i/r}$.

Now we'll analyze the following circuit and show that it computes F_8.

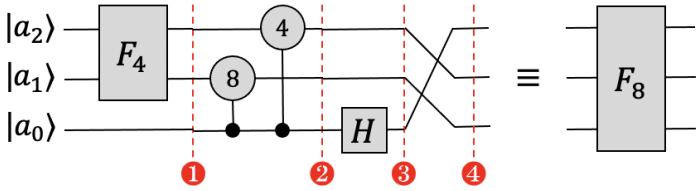

Figure 57: Caption.

We will trace through the stages of this circuit.

State at stage ❶

From Eq. (68), this is $\left(|0\rangle + \omega^{2[a_2a_10]}\,|1\rangle\right) \otimes \left(|0\rangle + \omega^{[a_2a_10]}\,|1\rangle\right) \otimes |a_0\rangle$.

State at stage ❷

The two controlled-phase gates change the state to

$$\left(|0\rangle + \omega^{2[a_2a_10]}\omega^{2a_0}\,|1\rangle\right) \otimes \left(|0\rangle + \omega^{[a_2a_10]}\omega^{a_0}\,|1\rangle\right) \otimes |a_0\rangle \tag{72}$$

$$= \left(|0\rangle + \omega^{2[a_2a_1a_0]}\,|1\rangle\right) \otimes \left(|0\rangle + \omega^{[a_2a_1a_0]}\,|1\rangle\right) \otimes |a_0\rangle \tag{73}$$

$$= \left(|0\rangle + \omega^{2a}\,|1\rangle\right) \otimes \left(|0\rangle + \omega^{a}\,|1\rangle\right) \otimes |a_0\rangle\,. \tag{74}$$

State at stage ❸

Applying a Hadamard gate to the third qubit changes the state to

$$\left(|0\rangle + \omega^{2a}|1\rangle\right) \otimes \left(|0\rangle + \omega^{a}|1\rangle\right) \otimes \left(|0\rangle + (-1)^{a_0}|1\rangle\right) \tag{75}$$

$$= \left(|0\rangle + \omega^{2a}|1\rangle\right) \otimes \left(|0\rangle + \omega^{a}|1\rangle\right) \otimes \left(|0\rangle + \omega^{4a}|1\rangle\right). \tag{76}$$

Notice that this is the state in Eq. (64), except the qubits are in the wrong order.

State at stage ❹

Moving the third qubit to the left and shifting the other two qubits right yields

$$\left(|0\rangle + \omega^{4a}|1\rangle\right) \otimes \left(|0\rangle + \omega^{2a}|1\rangle\right) \otimes \left(|0\rangle + \omega^{a}|1\rangle\right) = F_8|a\rangle\,. \tag{77}$$

What we have shown is a special case of the following more general recurrence for F_{2^n}.

Theorem 7.1. *For all $n \geq 1$, the Fourier transform F_{2^n} can be expressed as*

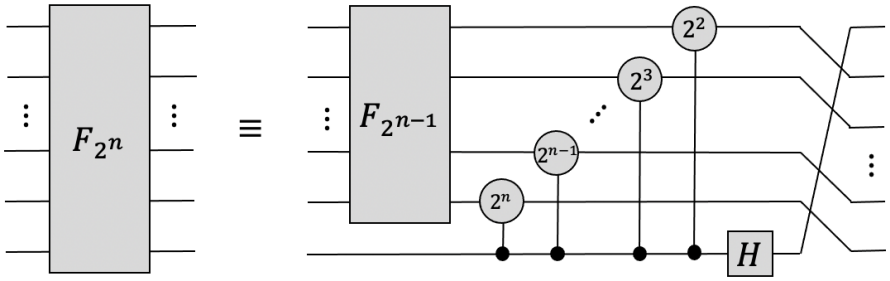

Figure 58: Quantum circuit for F_{2^n} recursively constructed in terms of $F_{2^{n-1}}$.

This is straightforward to verify, along the lines of the analysis for the $n = 3$ case.

7.3.2 Unravelling the recurrence

By repeatedly applying Theorem 7.1, we can construct a quantum circuit for the Fourier transform for any $n \geq 1$. The recurrence bottoms out at $n = 1$, where $F_2 = H$. Here is the circuit for the $n = 4$ case.

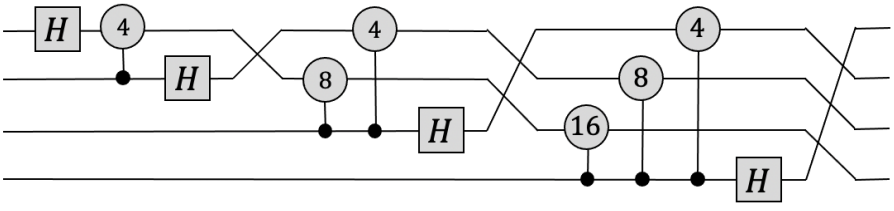

Figure 59: Quantum circuit for F_{2^4}.

Notice that there are places where the qubits are rearranged. Instead of doing these rearrangements, we can move the gates around. Then there's just one net rearrangement at the very end, which turns out to be a reversal of the order of the qubits.

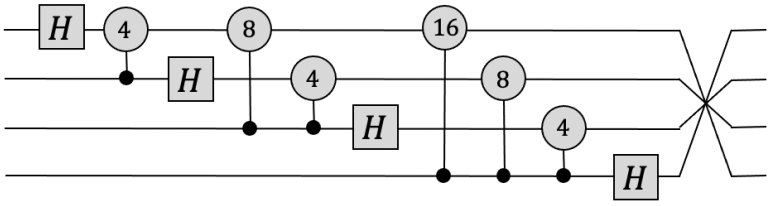

Figure 60: Quantum circuit for F_{2^4}, with rearrangements deferred until the end.

How do we perform this reversal of the order of the qubits? We can define a 2-qubit SWAP gate.

Definition 7.4 (SWAP gate)**.** *The 2-qubit* SWAP gate *is defined as the unitary operation that, for all $a, b \in \{0, 1\}$, maps $|a\rangle\,|b\rangle$ to $|b\rangle\,|a\rangle$. The unitary matrix for the gate is*

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{78}$$

and circuit notation for this gate is shown in figure 61.

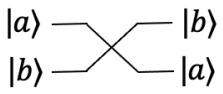

Figure 61: Notation for SWAP gate.

The reversal consists of around $n/2$ SWAP gates.

Intuitively, the notation for SWAP gate suggests a physically movement of the qubits. In principle, one could do that, but it would be nice not to be adding a brand new elementary operation into our repertoire of 2-qubit gates (if we can avoid it). It turns out this this SWAP gate can be computed by three CNOT gates.

Exercise 7.5. *Show that the 2-qubit SWAP gate can be implemented by a sequence of three CNOT gates (appropriately oriented).*

Finally, let's count the number of gates in our circuit construction (which is the natural generalization of the construction in figure 60 to F_{2^n}). There are n Hadamard gates, one for each qubit. The number of controlled-phase gates is

$$1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2). \tag{79}$$

And there are most $\frac{n}{2}$ SWAP gates—which translates into $\frac{3n}{2}$ CNOT gates. That's a total of $O(n^2)$ gates for computing the Fourier transform F_{2^n}.

Exercise 7.6 (straightforward)**.** *Give a quantum circuit that computes the inverse Fourier transform $F_{2^n}^*$ with $O(n^2)$ gates. (There are two different approaches that lead to slightly different circuits.)*

8 Definition of the discrete log problem

The quantum "algorithms" that we've seen up until now are not algorithms in usual sense. They are in the black-box model, where one is given an unknown function and the goal is to extract information about the function with as few queries to the function as possible. Often, the unknown function is promised to have a special kind of structure (such as the function arising in Simon's problem).

In the next section I will describe a remarkable algorithm for solving the *discrete log problem (DLP)*. This is not a black box problem! It is a conventional computational problem where the input is given as a binary string and the output is also a binary string. No classical polynomial algorithm is known for this problem. In fact, the presumed hardness of this problem has been the basis of cryptosystems (such as the Diffie-Hellman key exchange protocol). In 1994, Peter Shor discovered a polynomial quantum algorithm for this problem, and it's based on the underlying methodologies used in Simon's algorithm—which may sound surprising, since Simon's problem is a black-box problem.

In this section, I define the discrete log problem. In the next section I will describe Shor's polynomial quantum algorithm for this problem. In order to define the discrete log problem, we first need to be familiar with two sets, called $\mathbb{Z}_m$ and $\mathbb{Z}_m^*$.

8.1 Definitions of $\mathbb{Z}_m$ and $\mathbb{Z}_m^*$

Definition 8.1 (of the ring $\mathbb{Z}_m$)**.** *For any positive integer $m \geq 2$, define $\mathbb{Z}_m$ as the set $\{0, 1, \ldots, m-1\}$, equipped with addition and multiplication modulo m.*

A more familiar example of a ring is the set of all integers $\mathbb{Z}$ equipped with "ordinary" addition and multiplication. Note that there are only two elements of $\mathbb{Z}$ that have multiplicative inverses (namely $+1$ and -1). What are the elements of $\mathbb{Z}_m$ that have multiplicative inverses? It depends on m. Let's look at the case where $m = 9$. The elements of $\mathbb{Z}_9$ that have multiplicative inverses are 1, 2, 4, 5, 7, and 8 (0, 3, and 6 do not have multiplicative inverses).

Definition 8.2 (of the group $\mathbb{Z}_m^*$)**.** *For any positive integer $m \geq 2$, the set $\mathbb{Z}_m^*$ consists of all elements of $\mathbb{Z}_m$ that have multiplicative inverses. $\mathbb{Z}_m^*$ is a group.*

For the purposes of DLP, we need only consider $\mathbb{Z}_p^*$ for prime p. In that case, it's known that every non-zero element of $\mathbb{Z}_p^*$ has a multiplicative inverse. Therefore we can use this simple definition of the group $\mathbb{Z}_p^*$ for case where p is prime.

Definition 8.3 (of the group $\mathbb{Z}_p^*$)**.** *For any prime p, define $\mathbb{Z}_p^*$ as the group with elements $\{1, \ldots, p-1\}$, where the operation is multiplication modulo p.*

8.2 Generators of $\mathbb{Z}_p^*$ and the exponential/log functions

An element g of the group $\mathbb{Z}_p^*$ is called a *generator* of the group if the set of all powers g is the group. Whenever a group has such an element, it is called *cyclic*.

Let's consider the case where $p = 7$ as an example. $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$. Obviously 1 is not a generator, since all powers of 1 are just 1. What about 2? 2 is not a generator either, because of we list the powers of 2, which are 2^0, 2^1, 2^2, and so on, we get the sequence $1, 2, 4, 1, 2, 4, \ldots$ so we only get the set $\{1, 2, 4\}$, which is a proper subgroup of $\mathbb{Z}_7^*$. What about 3? 3 *is* a generator, since the powers of 3 are $1, 3, 2, 6, 4, 5, 1, 3, 2, \ldots$, which is the entire set $\mathbb{Z}_7^*$. It turns out that $\mathbb{Z}_p^*$ is cyclic for any prime p.

Theorem 8.1. *For any prime p, there exists $g \in \mathbb{Z}_p^*$ such that*

$$\mathbb{Z}_p^* = \left\{ g^k : k \in \{0, 1, \ldots, p-2\} \right\}. \tag{80}$$

Notice that the exponents run from 0 to $p-2$. This makes sense because the size of $\mathbb{Z}_p^*$ is $p-1$ (it's not p because $0 \notin \mathbb{Z}_p^*$). So the set of exponents of a generator is $\mathbb{Z}_{p-1}$ (it's not $\mathbb{Z}_p$). And, if the elements of $\mathbb{Z}_p^*$ are expressed as powers of a generator g, then $g^x g^y = g^{x+y \bmod p-1}$ holds for any $x, y \in \mathbb{Z}_{p-1}$. In other words, multiplication in $\mathbb{Z}_p^*$ corresponds to addition mod $p-1$ in the exponents of g. Formally, there is an isomorphism between the additive group $\mathbb{Z}_{p-1}$ and the multiplicative group $\mathbb{Z}_p^*$.

Definition 8.4 (discrete exp function)**.** *Relative to a prime modulus p and a generator g of $\mathbb{Z}_p^*$, let's first define the* discrete exponential function $\exp_g : \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$ *as, for all $r \in \mathbb{Z}_{p-1}$,*

$$\exp_g(r) = g^r. \tag{81}$$

Definition 8.5 (discrete log function)**.** *Relative to a prime modulus p and a generator g of $\mathbb{Z}_p^*$, define the* discrete log function $\log_g : \mathbb{Z}_p^* \to \mathbb{Z}_{p-1}$ *as the inverse of the discrete exponential function* $\exp_g$*. The input to* $\log_g$ *is some $s \in \mathbb{Z}_p^*$, and the output is the value of $r \in \mathbb{Z}_{p-1}$ such that $g^r = s$.*

8.3 Discrete exponential problem

For the *discrete exp problem*, the input is (p, g, r), where

- p is an n-bit prime.
- g is a generator of $\mathbb{Z}_p^*$.
- $r \in \mathbb{Z}_{p-1}$.

And the output is: $\exp_g(r)$, which is g^r.

How hard is it to compute this function? Of course, g^r is equal to g multiplied by itself r times. So $\exp_g(r)$ can obviously be computed by r multiplications (the precise number of multiplications is actually $r - 1$). But r is an n-bit number, so r can be roughly as large as 2^n. That's an exponentially large number of multiplication operations! Using this approach, the circuit-size would be exponential in n. But there's a simple trick for doing this more efficiently, called the *repeated squaring trick*.

8.3.1 Repeated squaring trick

The idea is the following. You multiply g by itself to get g^2. Then you multiply g^2 by itself to get g^4. And then you multiply g^4 by itself get g^8, and so on. This way, you can compute g^{2^n} at the cost of only n multiplications. That's how the repeated squaring trick works when the exponent r is a power of 2.

What if r is not a power of 2? The above idea can be adjusted to compute *any* n-bit exponent r with fewer than $2n$ multiplications. The idea is based on the fact that g^r can be written as

$$g^{r_n \cdots r_3 r_2 r_1} = ((\cdots (g^{r_n})^2 \cdots g^{r_3})^2 g^{r_2})^2 g^{r_1}, \tag{82}$$

where $r_n ... r_3 r_2 r_1$ is the binary representation of an n-bit exponent r.

Note that $O(n)$ multiplications, at cost $O(n \log(n))$ gates each, leads to a classical gate cost of order $O(n^2 \log(n))$ for computing the discrete exponential function.

8.4 Discrete log problem

For the *discrete log problem (DLP)*, the input is (p, g, s), where

- p is an n-bit prime.
- g is a generator of $\mathbb{Z}_p^*$.

- $s \in \mathbb{Z}_p^*$.

And the output is: $\log_g(s)$, which is the $r \in \mathbb{Z}_{p-1}$ for which $g^r = s$.

9 Shor's algorithm for the discrete log problem

The overall idea behind Shor's algorithm is to convert the discrete log problem (DLP) into a generalization of Simon's problem, and then to solve that generalization of Simon's problem. Since DLP is not a black-box problem, how can such a conversion work? It works by creating a function with a property similar to Simon's *that can be efficiently implemented* so as to simulate black-box queries to the function. This is Shor's function.

9.1 Shor's function with a property similar to Simon's

Recall that an instance of the discrete log problem consists of three n-bit numbers: p (an n-bit prime), g (a generator of $\mathbb{Z}_p^*$), and s (an element of $\mathbb{Z}_p^*$).

Shor's function $f : \mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$ is defined as

$$f(a_1, a_2) = g^{a_1} s^{-a_2}, \tag{83}$$

for all $a_1, a_2 \in \mathbb{Z}_{p-1}$. What's interesting about this function is where the collisions are. When is $f(a_1, a_2) = f(b_1, b_2)$?

Theorem 9.1. *For an instance (p, g, s) of DLP, the function $f : \mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$ defined by Eq. (83) has the property that*

$$f(a_1, a_2) = f(b_1, b_2) \quad \textit{if and only if } (a_1, a_2) - (b_1, b_2) \textit{ is a multiple of } (r, 1), \tag{84}$$

where $r = \log_g(s)$.

The significance of this is that it resembles the Simon property. It's the analogue of the Simon property when we switch from mod 2 arithmetic to mod $p-1$ arithmetic. To see this, recall that the Simon property can be stated as: $f(a) = f(b)$ if and only if $a \oplus b$ is either a string of n zeroes or the string r. Notice that: $a \oplus b$ is the same as $a - b$ in mod 2 arithmetic. So we can write the Simon property as:

> **Simon property for** $f : (\mathbb{Z}_2)^n \to (\mathbb{Z}_2)^n$
> $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$ if and only if $(a_1, \ldots, a_n) - (b_1, \ldots, b_n)$ is a multiple of $(r_1, \ldots, r_n)$ in mod 2 arithmetic.

And here's again is the property that Shor's function has:

Property of Shor's function for $f : \mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1} \to \mathbb{Z}_p^*$
$f(a_1, a_2) = f(b_1, b_2)$ if and only if $(a_1, a_2) - (b_1, b_2)$ is a multiple of $(r, 1)$ in mod $p - 1$ arithmetic.

Proof of Theorem 9.1. The proof is elementary, but we'll go through it carefully. Although we do not know what r is, we do know that an r exists such that $s = g^r$. This means that $s^{a_2} = g^{ra_2}$. Therefore, $f(a_1, a_2) = g^{a_1} g^{-ra_2} = g^{a_1 - ra_2}$. It's nice that to write f this way, because then

$$f(a_1, a_2) = f(b_1, b_2) \tag{85}$$
$$\text{if and only if} \quad a_1 - ra_2 = b_1 - rb_2 \tag{86}$$
$$\text{if and only if} \quad (a_1, a_2) \cdot (1, -r) = (b_1, b_2) \cdot (1, -r) \tag{87}$$
$$\text{if and only if} \quad \big((a_1, a_2) - (b_1, b_2)\big) \cdot (1, -r) = 0. \tag{88}$$

In equation (88), the dot-product being zero is like an orthogonality relation between the vector $(a_1, a_2) - (b_1, b_2)$ and the vector $(1, -r)$. Here's a schematic sketch of the vector $(1, -r)$.

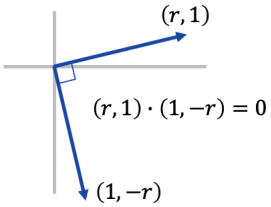

Figure 62: Schematic illustration of $(r, 1)$ orthonormal to $(1, -r)$.

Notice that the vector $(r, 1)$ is orthogonal to the vector $(1, -r)$ in that their dot-product is zero. Now, in a two-dimensional space, the vectors that are orthogonal to a particular vector are all multiples of *one* of the orthogonal vectors. So, we might expect $(a_1, a_2) - (b_1, b_2)$ to be a multiple of $(r, 1)$. This intuition (valid for vector spaces with inner products) is confirmed in our context by a simple calculation:

$$(v_1, v_2) \cdot (1, -r) = 0 \tag{89}$$
$$\text{if and only if} \quad v_1 = rv_2 \tag{90}$$
$$\text{if and only if} \quad v_2 = k \ \text{ and } \ v_1 = rk, \ \text{ for some } k \in \mathbb{Z}_{p-1} \tag{91}$$
$$\text{if and only if} \quad (v_1, v_2) = k(r, 1), \ \text{ for some } k \in \mathbb{Z}_{p-1}. \tag{92}$$

Therefore, $f(a_1, a_2) = f(b_1, b_2)$ if and only if $(a_1, a_2) - (b_1, b_2)$ is a multiple of $(r, 1)$. □

9.2 The Simon mod m problem

We've established that Shor's function satisfies a variant of Simon's property where the domain of the function is changed from $\{0,1\}^n = (\mathbb{Z}_2)^n$ to $(\mathbb{Z}_{p-1})^2$.

Now, what we're going to do is digress from DLP to investigate the generalization of Simon's problem, where the modulus is changed from 2 to m. We'll first find an efficient quantum black-box algorithm for the Simon mod m problem, where we are given a function

$$f : (\mathbb{Z}_m)^d \to T, \tag{93}$$

where T can be any set—but for convenience we'll assume that $T = \{0,1\}^k$ for some k (any T can be enlarged so that it's size is a power of 2).

Definition 9.1 (of an m-to-1 function)**.** *A function is m-to-1 if, for every value attained by the function, there are exactly m preimages. In other words, all colliding sets are of size m.*

Recall that, in the Simon's original problem, we had colliding pairs. Now, here is a mod m analogue of the original Simon property.

Definition 9.2 (Simon mod m property)**.** *An m-to-1 function $f : (\mathbb{Z}_m)^d \to T$ satisfies the* Simon mod m property*, if there exists a non-zero $r \in (\mathbb{Z}_m)^d$ such that: for all $a, b \in (\mathbb{Z}_m)^d$, it holds that $f(a) = f(b)$ if and only if $a - b$ is a multiple of r.*

Notice that the Simon mod m property is equivalent to the property that every colliding set of f is of the form $\{a, a+r, a+2r, \ldots, a+(m-1)r\} = \{a+kr : k \in \mathbb{Z}_m\}$, for some $a \in (\mathbb{Z}_m)^d$. Figure 63 is a schematic diagram of the two-dimensional case.

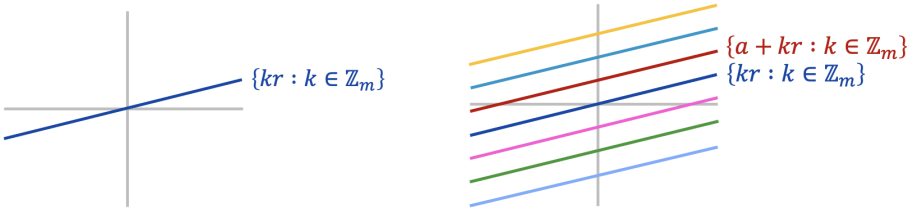

Figure 63: Each colliding set is one of the colored lines (an offset of the line $\{kr : k \in \mathbb{Z}_m\}$).

Think of $\{kr : k \in \mathbb{Z}_m\}$ as a line in $(\mathbb{Z}_m)^d$ passing through the origin. And think of $\{a+kr : k \in \mathbb{Z}_m\}$ as an *offset* of the line by $a \in (\mathbb{Z}_m)^d$. Each colliding set is an offset.

Recall that, for the original Simon property, the colliding pairs were of the form $\{a, a \oplus r\}$, which are offsets of $\{0, r\}$. In that case, the colored lines in figure 63 correspond to the colliding pairs.

Definition 9.3. *For a function $f : (\mathbb{Z}_m)^d \to T$, define an f-query as unitary operation U_f such that, for every $(a_1, a_2, \ldots, a_n) \in (\mathbb{Z}_m)^d$ and $b \in T$,*

$$U_f\big(|a_1\rangle\,|a_2\rangle \ldots |a_n\rangle\,|b\rangle\big) = |a_1\rangle\,|a_2\rangle \ldots |a_n\rangle\,|b \oplus f(a_1, a_2, \ldots, a_n)\rangle\,, \tag{94}$$

where the $\oplus$ operation denotes bit-wise XOR (recall our assumption that $T = \{0,1\}^k$).

We use the following notation for f-queries.

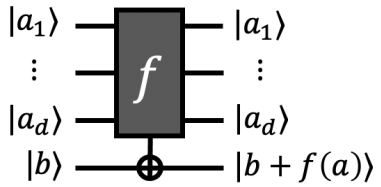

Figure 64: Notation for a quantum f-query gate for $f : (\mathbb{Z}_m)^d \to T$.

Notice that the lines are drawn thicker than those in our previous quantum circuits where the lines carried qubits. This is to help convey the fact that the data flowing through these lines is generally not qubits, but states of dimension higher than 2. The first d lines are carrying an m-dimensional quantum states, and the last line is carrying a $|T|$-dimensional state.

Definition 9.4 (Simon mod m problem)**.** *For Simon's problem mod m, you're given a black-box that computes an f-query for an unknown function f that is promised to have the Simon mod m property, and your goal is to determine the parameter r, based on queries to f.*

How do we solve the Simon mod m problem? Let's start by recalling the algorithm for the original Simon's problem. It was based on repeated runs of this circuit that makes a single f-query.

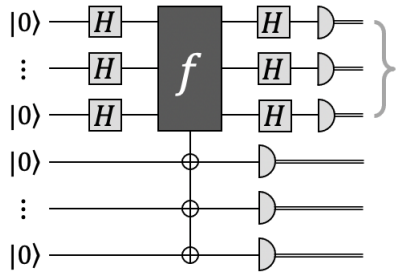

Figure 65: Circuit used for Simon's problem.

Each run of the circuit to produces a uniformly random element of of the orthogonal complement of r. After a few runs, we obtain enough such b's to be able to deduce r by solving a system of linear equations in mod 2 arithmetic.

The proposed algorithm for the Simon mod m problem will be based on a quantum circuit similar to the one in figure 65, but where the horizontal lines represent m-dimensional registers (rather than qubits) and the single-register gates are the Fourier transforms defined in section 7.1 and their inverses (rather than Hadamard transforms).

9.3 Query algorithm for Simon mod m

The algorithm is based on the circuit in figure 66.

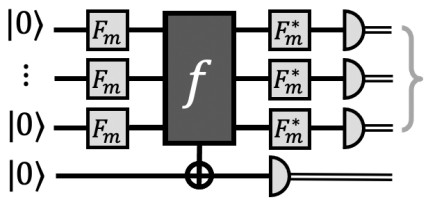

Figure 66: Proposed circuit for the Simon mod m problem.

Note that, in the special case where $m = 2$ case, the circuit in figure 66 is almost the same as the circuit in figure 65. What's the output of this circuit? We'll analyze this for the case where $d = 2$, which the circuit in figure 67.

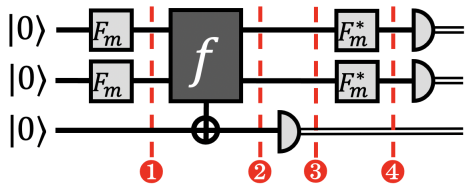

Figure 67: Circuit for Simon mod m problem in the case where $d = 2$.

Let's trace through the evolution of the state of the registers.

State at stage ❶

Since $F_m |0\rangle = \frac{1}{\sqrt{m}}(|0\rangle + |1\rangle + \cdots + |m-1\rangle)$, this state is

$$\frac{1}{\sqrt{m^2}} \sum_{a \in \mathbb{Z}_m \times \mathbb{Z}_m} |a_1, a_2\rangle |0\rangle . \tag{95}$$

State at stage ❷

Since the query maps each basis state $|a_1, a_2\rangle |0\rangle$ to $|a_1, a_2\rangle |f(a_1, a_2)\rangle$, when the input is superposition, the output of the query is

$$\frac{1}{\sqrt{m^2}} \sum_{a \in \mathbb{Z}_m \times \mathbb{Z}_m} |a_1, a_2\rangle |f(a_1, a_2)\rangle . \tag{96}$$

State at the first two registers at stage ❸

Measuring the state of the third register causes its state to collapse to some value of f and the state of the first two registers to collapse to a uniform superposition of all the pre-images of that value of f. This preimage is one of the collapsing sets of f (one of the colored lines in figure 63). Therefore, the state after this measurement is

$$\frac{1}{\sqrt{m}} \sum_{k \in \mathbb{Z}_m} |(a_1, a_2) + k(r_1, r_2)\rangle , \tag{97}$$

for some $(a_1, a_2) \in \mathbb{Z}_m \times \mathbb{Z}_m$.

This state is identical to the state that is the outcome of the following process:

1. Randomly choose one of the colliding sets (the colored lines in figure 63).
2. Take the uniform superposition of the elements of that colliding set.

State at the first two registers at stage ❹

$$F_m^* \otimes F_m^* \left(\tfrac{1}{\sqrt{m}} \sum_{k \in \mathbb{Z}_m} |(a_1, a_2) + k(r_1, r_2)\rangle \right) \tag{98}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{k \in \mathbb{Z}_m} \left(F_m^* \otimes F_m^* |(a_1, a_2) + k(r_1, r_2)\rangle \right) \tag{99}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{k \in \mathbb{Z}_m} \left(\tfrac{1}{\sqrt{m^2}} \sum_{b \in \mathbb{Z}_m \times \mathbb{Z}_m} \omega^{-(a+kr)\cdot b} |b_1, b_2\rangle \right) \tag{100}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{k \in \mathbb{Z}_m} \left(\tfrac{1}{m} \sum_{b \in \mathbb{Z}_m \times \mathbb{Z}_m} \omega^{-a\cdot b} \omega^{-k(r\cdot b)} |b_1, b_2\rangle \right) \tag{101}$$

$$= \tfrac{1}{\sqrt{m}} \sum_{b \in \mathbb{Z}_m \times \mathbb{Z}_m} \omega^{-a\cdot b} \left(\tfrac{1}{m} \sum_{k \in \mathbb{Z}_m} \omega^{-k(r\cdot b)} \right) |b_1, b_2\rangle \,. \tag{102}$$

Notice that Eq. (102) contains an expression for the amplitude of $|b_1, b_2\rangle$ for any $(b_1, b_2) \in \mathbb{Z}_m \times \mathbb{Z}_m$. Since $|\omega^{-a\cdot b}| = 1$, what's important in Eq. (102) is the expression in parentheses, which is

$$\tfrac{1}{m} \sum_{k \in \mathbb{Z}_m} \omega^{-k(r\cdot b)} = \begin{cases} 1 & \text{if } (r_1, r_2) \cdot (b_1, b_2) = 0 \\ 0 & \text{if } (r_1, r_2) \cdot (b_1, b_2) \neq 0. \end{cases} \tag{103}$$

This implies that, for any $(b_1, b_2) \in \mathbb{Z}_m \times \mathbb{Z}_m$,

$$\Pr[(b_1, b_2)] = \begin{cases} \frac{1}{m} & \text{if } (r_1, r_2) \cdot (b_1, b_2) = 0 \\ 0 & \text{if } (r_1, r_2) \cdot (b_1, b_2) \neq 0. \end{cases} \tag{104}$$

Therefore the output of the circuit in figure 67 is a uniformly random sample from the set $\{(b_1, b_2) \in \mathbb{Z}_m \times \mathbb{Z}_m : b_1 r_1 + b_2 r_2 = 0\}$, which we loosely call the *orthogonal complement* of (r_1, r_2).

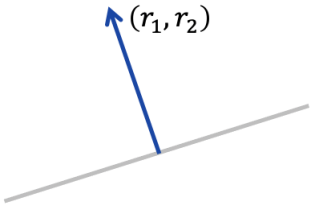

Figure 68: Schematic illustration of the orthogonal complement of (r_1, r_2).

All of the preceeding analysis generalizes in a straightforward way from two dimensions to d dimensions.

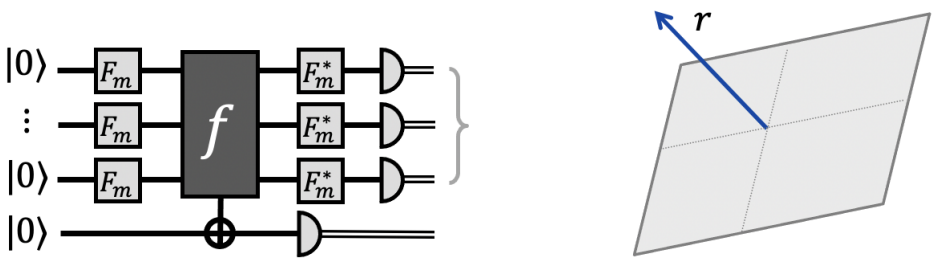

Figure 69: Simon mod m circuit produces a random element of the orthogonal complement of r.

In that case, output from the first d registers is a uniformly random element of the set $\{b \in \mathbb{Z}_m^d : b \cdot r = 0\}$ (the orthogonal complement of r).

As with Simon's algorithm, after repeated runs of the circuit in figure 69, we obtain several b's, from which we can hope to deduce r by solving a system of linear equations in mod m arithmetic (we'll return to this matter of solving the linear equation(s) in the context of the discrete log problem in section 9.5.1).

9.4 Returning to the discrete log problem

Now that we've analyzed the Simon mod m problem, let's get back to the original problem that we're concerned with, which is the discrete log problem. The input is: p, an n-bit prime number; g, a generator of $\mathbb{Z}_p^*$; and s, an element of $\mathbb{Z}_p^*$ (all three inputs are binary strings of length n). And the goal to produce $\log_g(s)$, which is the unique $r \in \mathbb{Z}_{p-1}$ for which $s = g^r$.

The reason why we turned our attention to the Simon mod m problem is that there is a special Shor function that satisfies the Simon mod m property (with m set to $p-1$), and a solution to that instance of Simon mod $p-1$ yields $r = \log_g(s)$.

How can we use our solution to Simon mod m to solve an instance of DLP, which is not in the query framework? The idea is to efficiently *implement* the query gate of the Shor function, as well as the other parts of the query circuit in terms of qubits and elementary gates (1- and 2-qubit gates). The elements of $\mathbb{Z}_{p-1}$ and $\mathbb{Z}_p^*$ can be represented by their n-bit binary representations as n-bit strings. Imagine a $3n$-qubit quantum circuit of the form of figure 70, where each grey box represents a quantum circuit consisting of elementary gates acting on qubits.

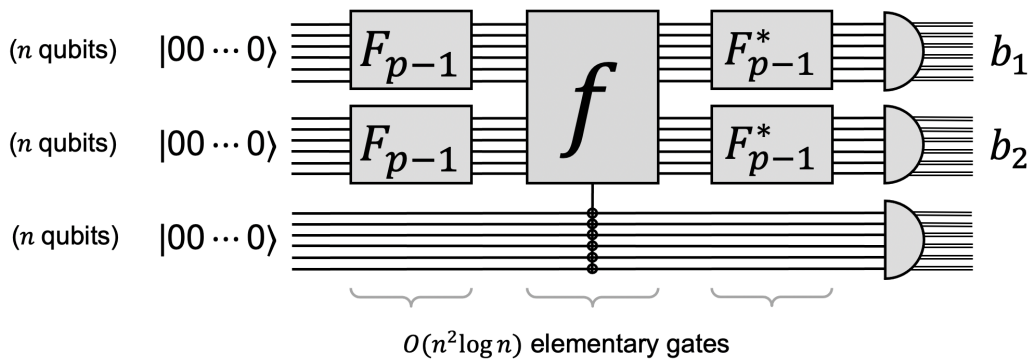

Figure 70: Implementation of the query algorithm in figure 71.

This circuit is an implementation of the circuit in figure 71.

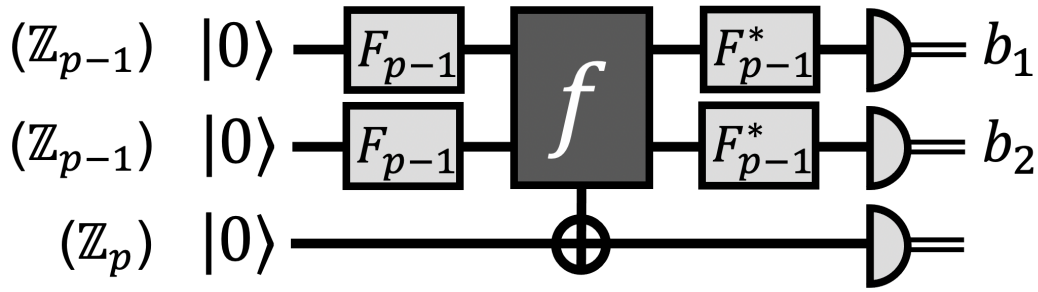

Figure 71: This is what is simulated by the circuit in figure 70.

A property of the Shor function f (defined is Eq. (83)) that's crucial to make this work is that $f(a_1, a_2) = g^{a_1} s^{a_2}$ can be computed *efficiently* by a classical circuit. The exponentiations can be computed efficiently using the repeated squaring trick (explained in section 8.3.1).

That's the overall idea behind Shor's algorithm for the discrete log problem. However, there are a few loose ends that have not been explained.

9.5 Loose ends

One loose end is that concerns technicalities in extracting r from repeated runs of the circuit in figure 67. This is discussed in section 9.5.1.

Another loose end is an important issue that arises in implementing the f-query, which is discussed in sections 9.5.2 and 9.5.3.

Finally, there's the matter of efficiently computing F_m, which is discussed in section 9.5.4.

9.5.1 How to extract r

As was done for the original Simon problem, we can set up a system of linear equations in mod m arithmetic. However, a complication arises when $\mathbb{Z}_m$ is not a field—and in fact, it's *not* a field in our case of interest, where $m = p - 1$, for a prime p.

Recall that, for DLP, the quantum circuit in figure 67 produces a uniformly random (b_1, b_2) such that

$$(b_1, b_2) \cdot (r, 1) \equiv 0 \pmod{p-1}. \tag{105}$$

How do we calculate r from this? From Eq. (105), we can solve for r as

$$r = -b_2/b_1 \bmod p - 1. \tag{106}$$

But, for this to work, b_1 must have an inverse modulo $p - 1$. It might not.

It can be proven that the fraction of elements of $\mathbb{Z}_{p-1}$ that have inverses is not too small. I won't get into further details here about how this is quantified. But, as a result of this, we can simply repeat the process of running the circuit in figure 67 a few times until we obtain a (b_1, b_2) where b_1 has an inverse in $\mathbb{Z}_{p-1}$.

9.5.2 How *not to* compute an f-query

Suppose that there is an efficient classical circuit that computes a function f. How do we simulate an f-query (as in Definition 9.3) in terms of elementary operations on qubits?

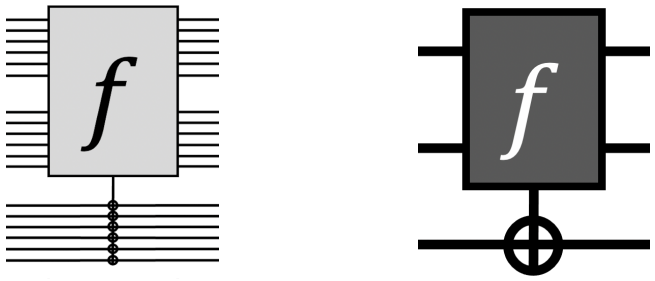

Figure 72: A circuit on qubits (left) so as to simulate an f-query (right).

We know from the lectures notes in [*Part 1: A primer for beginners*] that any classical circuit can be simulated by a quantum circuit with the same efficiency (up to a constant factor). However, that simulation used ancilla qubits and resulted in a quantum quantum circuit that maps each computational basis state of the form $|a\rangle\,|0^m\rangle\,|b\rangle$

to $|a\rangle\,|g(a)\rangle\,|f(a)\oplus b\rangle$, where $|0^m\rangle$ is a string of several $|0\rangle$ qubits that are used as ancillas, and $g(a)$ consists of some intermediate results of the computation.

To help clarify the point, here again in the quantum circuit from [*Part 1*], for computing the majority of three bits.

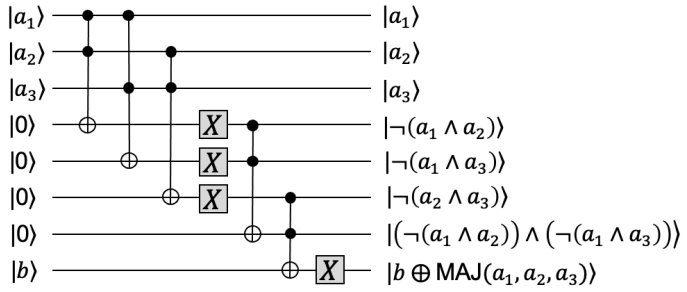

Figure 73: Quantum circuit that simulates classical circuit for the majority function.

The first three qubits contain the input to the function. The last qubit is where the output is XORed. And there are four ancilla qubits, whose states at the end of the computation are shown. We can refer to this as the "garbage", because we might think of disposing of those four qubits. Or ignoring them.

Is this OK for simulating an f-query? No, this leads to a problem if the f-query is applied at a state that's a superposition of computational basis states, such as

$$\sum_{a,b} \alpha_{a,b}\,|a\rangle\,|b\rangle\,|0^m\rangle\,. \tag{107}$$

The above purported implementation of an f-query maps this state to

$$\sum_{a,b} \alpha_{a,b}\,|a\rangle\,|b\oplus f(a)\rangle\,|g(a)\rangle \neq \Big(\sum_{a,b} \alpha_{a,b}\,|a\rangle\,|b\oplus f(a)\rangle\Big)\,|g(a)\rangle\,. \tag{108}$$

The state of the first two registers need not be

$$\sum_{a,b} \alpha_{a,b}\,|a\rangle\,|b\oplus f(a)\rangle\,. \tag{109}$$

(Note that, in Eqns. (107)(108)(109), I've moved the garbage register to the end so that the statement of inequality is simpler to write.)

But this problem has a simple remedy.

9.5.3 How *to* compute an f-query

The first step is to compute f with the ancilla registers. After that, the computation is reversed, so as to restore the ancilla registers back to state $|0\rangle$. But just before the reversal, the answer is XORed to another register, using CNOT gates.

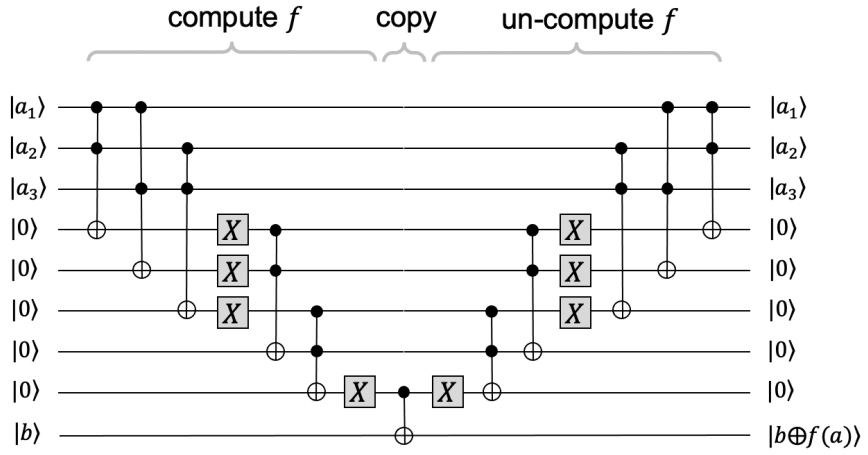

Figure 74: Clean computation of the majority function (the ancilla qubits are reset to $|0\rangle$).

The resulting circuit computes the f-query and restores the ancilla qubits back to their original states. Therefore, this works even if the f-query is applied to a superposition of computational basis states as

$$\sum_{a,b} \alpha_{a,b} |a\rangle |b \oplus f(a)\rangle |0^m\rangle = \Big(\sum_{a,b} \alpha_{a,b} |a\rangle |b \oplus f(a)\rangle\Big) |0^m\rangle . \tag{110}$$

The garbage register ends end up in a product state with the other registers.

Computing the function this way roughly doubles the number of gates needed.

9.5.4 How to compute the Fourier transform F_{p-1}

Another issue, is how to compute the Fourier transform mod $p-1$. It's an exponentially large matrix, and we need to compute it with a polynomial number of elementary gates acting on n qubits.

In fact, for modulus $p-1$, computing F_{p-1} is tricky. Shor's algorithm doesn't actually compute use this. Rather, it uses a Fourier transform with modulus a power of 2, which was shown to be easy to compute efficiently in section 7.3. The power of 2 is set so as to be close to $p-1$ (within a factor of 2 of $p-1$).

So Shor's algorithm uses the "wrong" Fourier transform. But it's not too wrong. Shor uses careful error analysis to show that, if the modulus is only off by a factor of two then the resulting wrong output state of the circuit is not too wrong. The result of the measurement still succeeds with some constant probability. I won't go into the details of this error analysis here. If you would like to see these details, you can find them in section 6 of Shor's paper[11] (https://arxiv.org/abs/quant-ph/9508027).

In the next section, I'll show you a simple efficient way of computing the Fourier transform when the modulus is a power of 2. Such Fourier transforms also have other applications, beyond the discrete log problem.

[11] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1484–1509, October 1997.

10 Phase estimation algorithm

In this section, we are going to investigate the *phase estimation problem*, which involves finding an eigenvalue of an unknown unitary operation, given as a black-box. This is an abstract problem that turns out to be a powerful algorithmic primitive.

10.1 A simple introductory example

Let U be an n-qubit unitary and $|\psi\rangle$ be an eigenvector of U with eigenvalue either $+1$ or -1. Suppose that we're given a controlled-U gate as a black-box

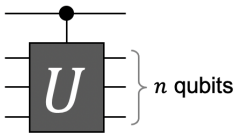

Figure 75: A controlled-U gate as a black box.

and we are also given n qubits that are in state $|\psi\rangle$. We're given no other information about U. Our controlled-U gate is essentially a black-box. Also, we're given no information about what state $|\psi\rangle$ is. All we know is that $|\psi\rangle$ is an eigenvector of U with eigenvalue $\lambda \in \{+1, -1\}$. Our goal is to determine whether λ is $+1$ or -1.

It turns out that we can solve this problem with just one single query to the controlled-U gate. I'd like to you to think about how this can be done. What state should you set the control qubit to? What state should you set the n target qubits to? Now is a good time to stop reading and think about this ...

So how did it go? Did you come up with a quantum circuit for this? If you did not then I'd like to urge you to try again. The solution is a pretty simple circuit. And here is a hint: it's similar to the very first quantum algorithm that we saw, for Deutsch's problem. So please give it another shot ...

Spoiler alert: a solution is on the next page.

Here's a quantum circuit that works.

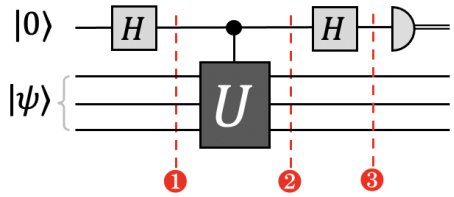

Figure 76: Quantum circuit determining the eigenvalue of U in the special case.

Let's trace through this to see how it works.

State at stage ❶

Applying H to the control qubit results in the state

$$\left(\tfrac{1}{\sqrt{2}}\left|0\right\rangle + \tfrac{1}{\sqrt{2}}\left|1\right\rangle\right)\left|\psi\right\rangle = \tfrac{1}{\sqrt{2}}\left|0\right\rangle\left|\psi\right\rangle + \tfrac{1}{\sqrt{2}}\left|1\right\rangle\left|\psi\right\rangle. \tag{111}$$

State at stage ❷

After the controlled-U gate, the state is

$$\tfrac{1}{\sqrt{2}}\left|0\right\rangle\left|\psi\right\rangle + \tfrac{1}{\sqrt{2}}\left|1\right\rangle U\left|\psi\right\rangle = \tfrac{1}{\sqrt{2}}\left|0\right\rangle\left|\psi\right\rangle + \tfrac{1}{\sqrt{2}}\left|1\right\rangle\lambda\left|\psi\right\rangle \tag{112}$$

$$= \left(\tfrac{1}{\sqrt{2}}\left|0\right\rangle + \tfrac{1}{\sqrt{2}}\lambda\left|1\right\rangle\right)\left|\psi\right\rangle. \tag{113}$$

Notice that, since $\lambda \in \{+1, -1\}$, the control qubit is in either the state $\left|+\right\rangle$ or state $\left|-\right\rangle$ (depending in λ).

State at stage ❸

It follows that, when H is applied again to the control qubit becomes

$$\begin{cases} \left|0\right\rangle & \text{if } \lambda = +1 \\ \left|1\right\rangle & \text{if } \lambda = -1. \end{cases} \tag{114}$$

We have just solved a special case of the *phase estimation problem*.

I will show you how the general case is defined—and then we'll see that it turns out not to be not that much harder than this case to solve. In order to define the phase estimation problem in full generality, we first need to extend our notion of a controlled-U gate.

10.2 Multiplicity controlled-U gates

Let's begin with a review of what a standard controlled-U gate is.

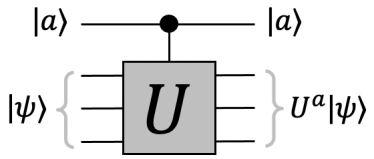

Figure 77: Controlled-U gate.

The unitary matrix is the block matrix

$$\begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}. \tag{115}$$

When the control qubit is in computational basis state $|a\rangle$ and the target qubit is in state $|\psi\rangle$, we can write the output state of the target qubit succinctly as $U^a|\psi\rangle$ (where $U^0 = I$ means "do nothing", and $U^1 = U$ means "apply U"). So, in the computational basis, the control qubit is a number which indicates how many times U should be applied to the target: 0 times or 1 time.

We can define a more general type of controlled-U gate where there are ℓ control qubits, and the number of times that U is applied is an ℓ-bit integer.

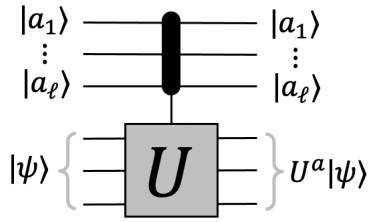

Figure 78: Multiplicity-controlled-U gate.

In this case, U can be applied zero times, U can be applied once, twice, three times, all the way up to $2^\ell - 1$ times. For example, if $\ell = 5$ and the control qubits are in state $|11010\rangle$ then U gets applied 26 times. Why 26? because 11010 is the number 26 in binary.

The unitary matrix of this kind of controlled-U gate is the block matrix

$$\begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ 0 & U & 0 & \cdots & 0 \\ 0 & 0 & U^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U^{2^\ell - 1} \end{bmatrix}. \tag{116}$$

Let's call this a *multiplicity-controlled-U gate*. In the computational basis, the state of the control qubits specifies how many times U should be applied.

As an aside, I want to distinguish this kind of gate from a different generalization of a controlled-U that we saw previously in the Toffoli gate.

10.2.1 Aside: multiplicity-control gates vs. AND-control gates

For the Toffoli gate, there are two control qubits and the unitary U is the Pauli X gate (a.k.a. NOT gate). For the Toffoli gate, the NOT is applied to the target qubit if both control qubits are $|1\rangle$, and nothing happens for the other computational basis states $|00\rangle$, $|01\rangle$, $|10\rangle$.

Our notation distinguishes between these controlled gates and our multiplicity-controlled gate.

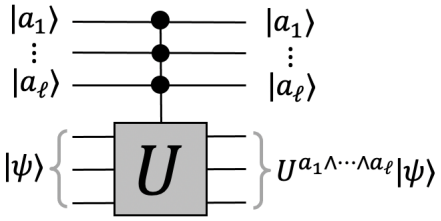

Figure 79: AND-controlled-U gate.

When we have a separate *dot* at each control qubit, that means that the U gets applied once if *all* control qubits are in state $|1\rangle$ and, for other computational basis states, nothing happens. So the unitary matrix is the block matrix

$$\begin{bmatrix} I & 0 & \cdots & 0 & 0 \\ 0 & I & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & I & 0 \\ 0 & 0 & \cdots & 0 & U \end{bmatrix}. \tag{117}$$

Our multiplicity-controlled-U gates are denoted differently, with *one single dot*, that's stretched out so as to cover all of the control qubits (figure 78).

10.3 Definition of the phase estimation problem

Now, we can define the phase estimation problem in generality.

Let U be an arbitrary unknown unitary operation acting on n qubits. Let $|\psi\rangle$ be an eigenvector of U. But now the eigenvalue is not restricted to being $+1$ or -1. The eigenvalue can be any complex number on the unit circle. So the eigenvalue is of the form $e^{2\pi i\varphi}$, where φ can be any element of the interval $[0, 1]$.

Definition 10.1 (Phase estimation problem)**.** *In the* phase estimation problem *we are given: a black-box for a multiplicity-controlled-U gate with ℓ control qubits and*

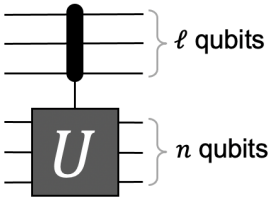

Figure 80: Caption.

one copy of state $|\psi\rangle$, which is an eigenvector of U with eigenvalue $e^{2\pi i\varphi}$ for some $\varphi \in [0, 1)$. The goal is to determine an ℓ-bit approximation of the value of φ. (Since φ can take on a continuum of values, determining it exactly *is unreasonable.)*

There's an *exact case* of this where φ is an ℓ-bit binary fraction. This case is simpler to work with than the general case. An ℓ-bit binary fraction is a rational number with 2^ℓ in the denominator and an integer $a \in \{0, 1, \ldots, 2^\ell - 1\}$ in the numerator. In other words, the binary representation of φ can be written exactly with only ℓ bits occurring after the radix point. For example, for $\ell = 6$,

$$\frac{19}{2^6} = 0.010011. \tag{118}$$

The *general case* is where φ is arbitrary. For example, if $\varphi = \frac{1}{3}$ then

$$\phi = 0.\overline{01} = 0.010101010101\ldots \tag{119}$$

(where the pattern repeats forever). That's not a binary fraction for any ℓ. The 8-bit approximation of this number is 0.01010101 (it's $\frac{1}{3}$ rounded down to the closest

8-bit binary fraction). The 7-bit approximation of this number is 0.0101011 (note that last bit is 1, not 0, because we round *up* to the closest 7-bit binary fraction. We always round φ towards the ℓ-bit binary fraction that's closest. Sometimes that means rounding down and sometimes that means rounding up.

For our applications, we will want to solve the general case of phase estimation, but it's conceptually useful to to first solve this problem in the exact case.

10.4 Solving the exact case

Here we will solve the phase estimation problem in the exact case—which turns out to be quite easy. In the exact case,

$$\varphi = \frac{a}{2^\ell} = 0.a_1 a_2 \dots a_\ell \,, \tag{120}$$

where $a \in \{0, 1, \dots, 2^\ell - 1\}$. Note that the eigenvalue can be written as

$$e^{2\pi i \varphi} = e^{2\pi i a/2^\ell} = \left(e^{2\pi i/2^\ell}\right)^a = \omega^a, \tag{121}$$

where ω is a primitive 2^ℓ-th root of unity. It clarifies things to think of the eigenvalue as ω^a in this manner. Note that the goal of determining the eigenvalue parameter φ is now equivalent to determining the value of the number a.

We'll start by putting the control qubits into a uniform superposition of all computational basis sates on ℓ qubits, which is accomplished by ℓ Hadamard transforms. And then we'll perform the multiplicity-controlled-U gate.

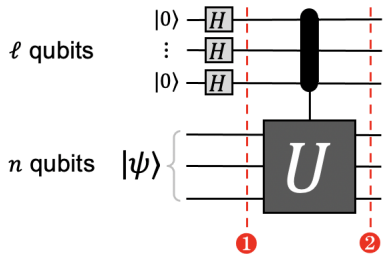

Figure 81: Caption.

Let's determine the output state of this quantum circuit.

State at stage ❶

The $H^{\otimes \ell}$ operation on $|0\rangle^{\otimes \ell}$ produces that state

$$\Big(|0...00\rangle + |0...01\rangle + |0...10\rangle + |0...11\rangle + \cdots + |1...11\rangle\Big)\,|\psi\rangle$$
$$= \Big(|0\rangle + |1\rangle + |2\rangle + |3\rangle + \cdots + \big|2^\ell - 1\big\rangle\Big)\,|\psi\rangle\,. \tag{122}$$

State at stage ❷

The multiplicity-controlled-U gate changes the state to

$$|0\rangle\,|\psi\rangle + |1\rangle\,U\,|\psi\rangle + |2\rangle\,U^2\,|\psi\rangle + |3\rangle\,U^3\,|\psi\rangle + \cdots + \big|2^\ell - 1\big\rangle\,U^{2^\ell - 1}\,|\psi\rangle$$
$$= |0\rangle\,|\psi\rangle + |1\rangle\,\omega^a\,|\psi\rangle + |2\rangle\,\omega^{2a}\,|\psi\rangle + |3\rangle\,\omega^{3a}\,|\psi\rangle + \cdots + \big|2^\ell - 1\big\rangle\,\omega^{(2^\ell - 1)a}\,|\psi\rangle \tag{123}$$
$$= \Big(|0\rangle + \omega^a\,|1\rangle + \omega^{2a}\,|2\rangle + \omega^{3a}\,|3\rangle + \cdots + \omega^{(2^\ell - 1)a}\,\big|2^\ell - 1\big\rangle\Big)\,|\psi\rangle\,. \tag{124}$$

Now, do you recognize this state of the first ℓ qubits? Haven't you seen the state $|0\rangle + \omega^a\,|1\rangle + \omega^{2a}\,|2\rangle + \cdots + \omega^{(2^\ell - 1)a}\,\big|2^\ell - 1\big\rangle$ before?

It's the Fourier basis state $F_{2^\ell}\,|a\rangle$ (see Eq. (40)). Remember that our goal is to determine a. The fact that the Fourier basis states for all the potential values of parameter a are orthogonal is a good sign. It means that they are distinguishable in principle. But we can also explicitly compute the value of a using a polynomial number of gates. Can you see how?

We can compute a by applying the the *inverse Fourier transform*. If we apply $F^*_{2^\ell}$ to $F_{2^\ell}\,|a\rangle$, the result is $|a\rangle$. So our phase estimation algorithm for the exact case is the quantum circuit in figure 82.

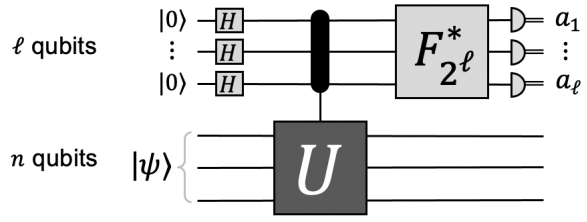

Figure 82: Quantum algorithm for phase estimation in the exact case.

The output of the ℓ measured qubits is a, the numerator of the binary fraction for φ, or, equivalently, the ℓ bits of the binary representation of φ.

Note that the algorithm makes only one query to the multiplicity-controlled-U gate, and all the other operations in the algorithm (the grey boxes) can be implemented with a polynomial number of elementary gates (with respect to ℓ, the number of control qubits). The dominant cost is that of computing $F^*_{2^\ell}$, which costs $O(\ell^2)$ elementary gates (section 7.3).

By the way, the $\ell = 1$ version of the exact case is that simple example that we solved in section 10.1 (and in that case $\omega = -1$).

10.5 Solving the general case

Now, what happens if we use this same algorithm for the general case, where φ can be *any* real number between 0 and 1? Recall that, in that case, we need to determine the ℓ-bit binary number that best approximates the true value of φ. We can write φ as an ℓ-bit binary fraction plus a quantity δ that represents the remaining bits that get rounded up[12] or down. More precisely,

$$\varphi = \frac{a}{2^\ell} + \delta, \tag{125}$$

where $a \in \{0, 1, \ldots, 2^\ell - 1\}$ and

$$|\delta| \leq \frac{1}{2^{\ell+1}}. \tag{126}$$

If φ gets rounded down then δ is positive; if φ gets rounded up then δ is negative. If $\delta = 0$, we are in the exact case. Let's analyze what the circuit that we used for the exact case does in the case where $\delta \neq 0$.

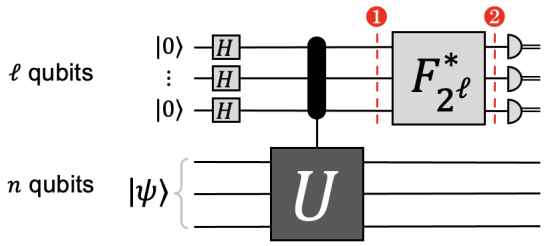

Figure 83: Quantum algorithm for phase estimation applied in the general case.

Since $|\psi\rangle$ is an eigenvector of U, the state of the second register (the last n qubits) does not change. All the action is with the first register (the first ℓ qubits). We trace through the evolution of the first ℓ qubits.

[12] Since $e^{2\pi i \cdot 1} = e^{2\pi i \cdot 0}$, we define distances within the interval $[0, 1)$ with "wrap-around", representing $\varphi = 1$ as $\varphi = 0$. By this convention, if $1 - \frac{1}{2^{\ell+1}} < \varphi < 1$ then φ rounds *up* to 0.

State at stage ❶

After the multiplicity-controlled-U gate, the state of the first ℓ qubits is

$$\frac{1}{\sqrt{2^\ell}} \sum_{b=0}^{2^\ell-1} \left(e^{2\pi i\varphi}\right)^b |b\rangle = \frac{1}{\sqrt{2^\ell}} \sum_{b=0}^{2^\ell-1} e^{2\pi i\left(\frac{a}{2^\ell}+\delta\right)b} |b\rangle \tag{127}$$

$$= \frac{1}{\sqrt{2^\ell}} \sum_{b=0}^{2^\ell-1} \omega^{ab} e^{2\pi i\delta b} |b\rangle\,, \tag{128}$$

where $\omega = e^{2\pi i/2^\ell}$. The effect of δ is highlighted in red. If $\delta = 0$ then $e^{2\pi i\delta b} = 1$ and the expression is $F_{2^\ell} |a\rangle$, which is consistent with what we obtained for the exact case.

State at stage ❷

After applying the inverse Fourier transform, the state becomes

$$F_{2^\ell}^* \left(\frac{1}{\sqrt{2^\ell}} \sum_{b=0}^{2^\ell-1} \omega^{ab} e^{2\pi i\delta b} |b\rangle \right) = \frac{1}{\sqrt{2^\ell}} \sum_{b=0}^{2^\ell-1} \omega^{ab} e^{2\pi i\delta b} \left(\frac{1}{\sqrt{2^\ell}} \sum_{c=0}^{2^\ell-1} \omega^{-bc} |c\rangle \right) \tag{129}$$

$$= \frac{1}{2^\ell} \sum_{c=0}^{2^\ell-1} \left(\sum_{b=0}^{2^\ell-1} \omega^{b(a-c)} e^{2\pi i\delta b} \right) |c\rangle\,. \tag{130}$$

Now, recall that the correct outcome for the phase estimation problem is a (the bits of an ℓ-bit approximation of φ). What's the probability that measuring the state in Eq. (130) results in outcome a? To figure this out, we look at the amplitude of the $|a\rangle$ term in this expression. Notice that if we substitute a for c then the factor $\omega^{b(a-c)}$ simplifies to $\omega^0 = 1$. So the amplitude of the $|a\rangle$ term in Eq. (130) is the sum

$$\frac{1}{2^\ell} \sum_{b=0}^{2^\ell-1} e^{2\pi i\delta b}. \tag{131}$$

As a reality check, what is this sum in the exact case, where $\delta = 0$? Can you see why it's 1? This makes sense, because, for the exact case, we've already seen that the measurement outcome is a with probability 1.

Returning to our case of interest, where $0 \neq |\delta| \leq 1/2^{\ell+1}$, we're interested in the absolute value squared of the amplitude in Eq. (131). At first glance, the sum may look a little daunting, but there's a closed-form expression for it. Can you see what it is?

The sum in Eq. (131) is a geometric series.[13] Therefore, we can use the formula for the sum of a geometric series to obtain

$$\frac{1}{2^\ell}\sum_{b=0}^{2^\ell-1} e^{2\pi i\delta b} = \frac{1}{2^\ell}\frac{1-(e^{2\pi i\delta})^{2^\ell}}{1-e^{2\pi i\delta}}. \tag{132}$$

What we're going to do next is use some simple geometric reasoning to show that the absolute value of this expression is at least $\frac{2}{\pi}$.

Lemma 10.1. *If δ is such that $0 \neq |\delta| \leq 1/2^{\ell+1}$ then*

$$\frac{1}{2^\ell}\left|\frac{1-(e^{2\pi i\delta})^{2^\ell}}{1-e^{2\pi i\delta}}\right| \geq \frac{2}{\pi}. \tag{133}$$

Proof. To lower-bound the expression, we'll show that the numerator is not too small and the denominator is not too large. We give the proof for the case where $\delta > 0$ (the case where $\delta < 0$ is similar, with upside-down versions of figures 84 and 85).

First, let's show that the denominator is not too large. Figure 84 shows 1 and $e^{2\pi i\delta}$ as points in the complex plane.

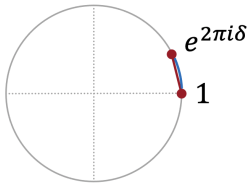

Figure 84: Geometric view of $|1-e^{2\pi i\delta}|$ as the distance between two points in $\mathbb{C}$.

The length of the red line segment is $|1-e^{2\pi i\delta}|$. The length of the red line is upper bounded by the arc-length between the two points, which is $2\pi\delta$. Therefore, we can upper bound of the denominator as

$$|1-e^{2\pi i\delta}| \leq 2\pi\delta. \tag{134}$$

Next, we'll lower bound the numerator. In this case, $|1-e^{2\pi i\delta 2^\ell}|$ is the length of the red line in figure 85.

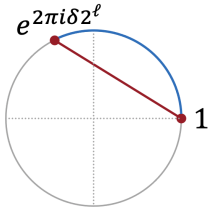

Figure 85: Geometric view of $|1-e^{2\pi i\delta 2^\ell}|$ as the distance between two points in $\mathbb{C}$.

[13] For $s \neq 1$, the formula for the sum of the geometric series $1+s+s^2+\cdots+s^{m-1} = \frac{1-s^m}{1-s}$.

Let's also consider the arc length between the two points, which is $2\pi\delta 2^\ell$. Note that the arc-length is *not* a lower bound of the distance—it's longer than the line segment. Think of the arc-length as the length of the line times some stretch-factor. How big is the stretch-factor? If the line segment is short then the stretch-factor is just slightly larger than 1 (in that case, the line and arc have almost the same lengths).

But the line and arc need not be short. The extremal case is where δ is as large as possible: $\delta = 1/2^{\ell+1}$. In that case, $(e^{2\pi i\delta})^{2^\ell} = e^{\pi i} = -1$, so the length of the line is 2 and the length of the arc is π. Thus, the maximum possible stretch-factor is $\pi/2$.

Therefore, we can lower bound the length of the line by the arc-length times $2/\pi$. Multiplying the arc-length by the reciprocal of the maximum stretch-factor compensates for how much longer the arc-length can be than the line segment. Therefore, the numerator is lower bounded as

$$|1 - e^{2\pi i\delta 2^\ell}| \geq 2\pi\delta 2^\ell\Big(\frac{2}{\pi}\Big) = 4\delta 2^\ell. \tag{135}$$

Now, we can substitute in our bounds in Eq. (134) and Eq. (135) for the numerator and the denominator to obtain

$$\frac{1}{2^\ell}\left|\frac{1-(e^{2\pi i\delta})^{2^\ell}}{1-e^{2\pi i\delta}}\right| \geq \frac{1}{2^\ell}\frac{4\delta 2^\ell}{2\pi\delta} = \frac{2}{\pi}. \tag{136}$$

□

This means that, if we measure (in the computational basis), we get the correct answer a with probability at least $4/\pi^2 = 0.405...$ (squaring the amplitude). That's slightly more than 40%. This might not seem like a very high success probability. But, for our purposes, what's important is that it's a *constant* (independent of ℓ and n). In the contexts where we will use phase estimation to achieve something else (such as to factor a number), we will be able to amplify the success probability by repeating the entire process a few times.

In summary, for the phase estimation problem, you are given an multiplicity-controlled-U gate with ℓ control-qubits and a state $|\psi\rangle$ that's an eigenvector of U with eigenvalue $e^{2\pi i\varphi}$. Your goal is to find an ℓ-bit approximation of φ. The following quantum circuit solves this problem with success probability at least $\frac{4}{\pi^2} = 0.405\ldots$.

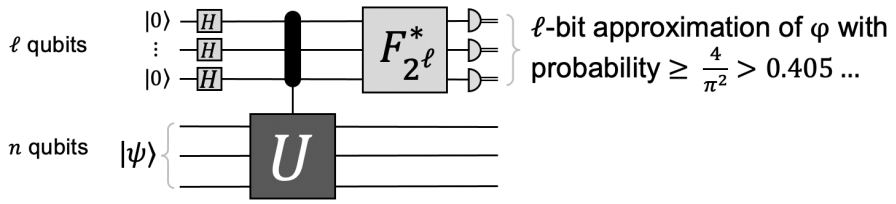

Figure 86: Summary of phase-estimation algorithm for approximating the eigenvalue $e^{2\pi i\varphi}$ of $|\psi\rangle$.

The algorithm makes one query to the multiplicity-controlled-U gate and has $O(\ell^2)$ elementary gates (the dominant cost being the implementation of $F^*_{2^\ell}$).

10.6 The case of superpositions of eigenvectors

Before ending this section, I'd like to bring your attention to a slightly different scenario. Suppose that, instead of being given an eigenvector of U, we're given a superposition of two eigenvectors with different eigenvalues.

Suppose that we're provided with a state of the form $\alpha_1 |\psi_1\rangle + \alpha_2 |\psi_2\rangle$, where:

- $|\psi_1\rangle$ is an eigenvector of U with eigenvalue $e^{2\pi i\varphi_1}$,
- $|\psi_2\rangle$ is an eigenvector of U with eigenvalue $e^{2\pi i\varphi_2}$,
- $|\alpha_1|^2 + |\alpha_2|^2 = 1$, and $\varphi_1 \neq \varphi_2$.

What happens if we run our phase estimation algorithm with this state in the target register?

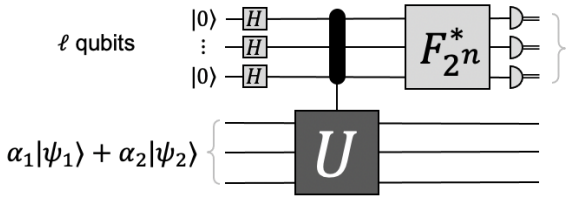

Figure 87: Scenario where input to phase algorithm is a superposition of two eigenvectors.

Exercise 10.1. *For the exact case, where φ_1 and φ_2 are both ℓ-bit binary integers, show that the output of the circuit in figure 87 is*

$$\begin{cases} \varphi_1 & \text{with probability } |\alpha_1|^2 \\ \varphi_2 & \text{with probability } |\alpha_2|^2. \end{cases} \tag{137}$$

So the net result is exactly the same as what occurs if, instead of a superposition of eigenvalues, we had just randomly selected one of the eigenvectors as

$$\begin{cases} |\psi_1\rangle & \text{with probability } |\alpha_1|^2 \\ |\psi_2\rangle & \text{with probability } |\alpha_2|^2 \end{cases} \tag{138}$$

and then input the selected eigenvector to the target register.

That's for the exact case. For the general case, it's similar, with the statement of the result qualified as "with success probability at least $4/\pi^2$". Also, although I've described the case of a superposition of *two* eigenvectors, there's nothing so special about two. There are similar results for the case of superpositions of more than two eigenvectors.

11 The order-finding problem

11.1 Greatest common divisors and Euclid's algorithm

In section 12, I will show you quantum algorithms for two problems: one is called the *order-finding problem* and the other is the *factoring problem*. I will show you how these algorithms can be based on the framework of the phase estimation problem that we saw in Part II.

In the present section, I will review some basics about divisors, common divisors, and the greatest common divisor of two integers.

Of course, we know that factoring a number x (a positive integer) is about finding its divisors (where the divisors are the numbers that divide x). If we have two numbers, x and y, then the *common divisors* of x and y are the numbers that divide both of them. For example, for the numbers 28 and 42 the common divisors are: 1, 2, 7, and 14. (4 is not a common divisor because, although 4 divides 28, it does not divide 42.)

Definition 11.1 (greatest common divisor (GCD))**.** *For two numbers x and y, their* greatest common divisor *is the largest number that divides both x and y. This is commonly denoted as* $\gcd(x, y)$*.*

The GCD of 28 and 42 is 14. What is the GCD of 16 and 21?
The answer is 1, since there is no larger integer that divides both of them.

Definition 11.2 (relatively prime)**.** *We say that numbers x and y are* relatively prime *if* $\gcd(x, y) = 1$*. That is, they have no common divisors, except the trivial divisor 1.*

Note that 16 and 21 are not prime but they are relatively prime.

Superficially, the problem of finding common divisors resembles the problem of finding divisors (which is the factoring problem). If x and y are n-bit numbers then a brute-force search would have to check among exponentially many possibilities.

In fact, the problem of finding greatest common divisors is considerably easier than factoring. It has been known for a long time that there is an efficient algorithm for computing GCDs. By long time, I mean approximately 2,300 years! That's how long ago Euclid published his algorithm (now known as the *Euclidean algorithm*). Of course, there were no computers back then, but Euclid's algorithm could be carried out by a hand calculation and it requires $O(n)$ arithmetic operations for n-digit

numbers. This translates into a gate cost of $O(n^2 \log n)$. Let's remember Euclid's GCD algorithm, because we're going to make use of it.

Now, I'd like to briefly review a few more properties of numbers. Let $m \geq 2$ be an integer that's not necessarily prime. Recall that $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$ and

$$\mathbb{Z}_m^* = \big\{x \in \mathbb{Z}_m : x \text{ has a multiplicative inverse mod } m\big\}, \tag{139}$$

where the latter is a group, with respect to multiplication mod m. For example,

$$\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}. \tag{140}$$

It turns out that x has a multiplicative inverse mod m if and only if $\gcd(x, m) = 1$. For example, you can see that 3, 6, and 7 (and several other numbers in $\mathbb{Z}_{21}$) are excluded from $\mathbb{Z}_{21}^*$ because they have common factors with 21.)

Now, let's take an element of $\mathbb{Z}_{21}^*$ and list all of its powers. Suppose we pick 5. The powers of 5 (5^0, 5^1, 5^2, ...) are

$$1, 5, 4, 20, 16, 17, 1, 5, 4, 20, 16, 17, 1, 5, 4, 20, 16, \ldots \tag{141}$$

Notice that it's periodic ($5^6 = 1$, and then the subsequent powers repeat the same sequence). The *period* of the sequence is 6.

If we were to pick 4 instead of 5 then the period of the sequence is different. The powers of 4 are: $1, 4, 16, 1, 4, 16, \ldots$ so the period is 3.

Definition 11.3 (order of $a \in \mathbb{Z}_m^*$)**.** *For any $a \in \mathbb{Z}_m^*$ define* the order of a modulo m (denoted as $\mathrm{ord}_m(a)$) *as the minimum positive r such that $a^r \equiv 1 \pmod{m}$. This is the period of the sequence of powers of a.*

Related to all this, we can define the computational problem of determining $\mathrm{ord}_m(a)$ from m and a. This problem has an interesting relationship to the factoring problem.

11.2 The order-finding problem and its relation to factoring

Definition 11.4 (order-finding problem)**.** *The input to the order-finding problem is two n-bit integers m and a, where $m \geq 2$ and $a \in \mathbb{Z}_m^*$. The output is $\mathrm{ord}_m(a)$, the order of a modulo m.*

A brute-force algorithm for order-finding is to compute the sequence of powers of a $(a, a^2, a^3, \ldots)$ until a 1 is reached. That can take exponential-time when the modulus

is an n-bit number because the order can be exponential in n. Although each power of a can be computed efficiently by the repeated-squaring trick, but there are potentially exponentially many different powers of a to check. In fact, there is no known polynomial-time *classical* algorithm for the order-finding problem.

But why should we care about solving the order-finding problem efficiently? It might come across as an esoteric problem in computational number theory. One reason to care is that the factoring problem *reduces* to the order-finding problem. If we can solve the order finding-problem efficiently then we can use that to factor numbers efficiently.

What I'm going to do next is show how any efficient algorithm (classical or quantum) for the order-finding problem can be turned into an efficient algorithm for factoring. This is true for the general factoring problem; however, for simplicity, I'm only going to explain it for the case of factoring numbers that are the product of two distinct primes. That's the hardest case, and the case that occurs in cryptographic applications.

Let our input to the factoring problem be an n-bit number m, such that $m = pq$, where p and q are distinct primes. Our goal is to determine p and q, with a gate-cost that is polynomial in n.

The first step is to select an $a \in \mathbb{Z}_m^*$ and use our quantum order-finding algorithm[14] as a subroutine to compute $r = \mathrm{ord}_m(a)$. Note that, since $a^r \equiv 1 \pmod{m}$, it holds that $a^r - 1$ is a multiple of m. In other words, m divides $a^r - 1$.

Now, the order r is either an even number or an odd number (it can go either way, depending on which a we pick). Something interesting happens when r is an even number. If r is even then a^r is a square, with square root $a^{r/2}$, and we can express $a^r - 1$ using the standard formula for a difference-of-squares as

$$a^r - 1 = (a^{r/2} + 1)(a^{r/2} - 1). \tag{142}$$

Since m divides $a^r - 1$ and $m = pq$, it follows that p and q each divide $a^r - 1$. It's well known that if a prime divides a product of two numbers then it must divide one of them. Also, it's not possible that both p and q divide the factor $a^{r/2} - 1$. Why not? Because that would contradict the fact that r is, by definition, the *smallest* number for which $a^r \equiv 1 \pmod{m}$. Therefore, there are three possibilities for the factors that p and q divide:

1. p divides $a^{r/2} + 1$ and q divides $a^{r/2} - 1$.

[14] In the next section, we'll see that there is an efficient quantum algorithm for the order-finding problem that we can use for this.

2. q divides $a^{r/2}+1$ and p divides $a^{r/2}-1$.

3. p and q both divide $a^{r/2}+1$.

In the first two cases p and q divide different factors; in the third case, they divide the same factor. Our reduction works well when r is even *and* p and q divide different factors. With this in mind, let's define a to be "lucky" when these conditions occur.

Definition 11.5 (of a *lucky* $a \in \mathbb{Z}_m^*$)**.** *With respect to some $m \geq 2$, define an $a \in \mathbb{Z}_m^*$ to be* lucky *if $ord_m(a)$ is an even number and m does not divide $a^{r/2}+1$.*

Given an $a \in \mathbb{Z}_m^*$ that is lucky in the above sense, it holds that

$$\gcd(a^{r/2}+1, m) \in \{p, q\}. \tag{143}$$

This enables us to easily factor m by computing $\gcd(a^{r/2}+1, m)$. The repeated-squaring trick enables us to compute $a^{r/2}$ with a gate cost of $O(n^2 \log n)$ and Euclid's algorithm enables us to compute $\gcd(a^{r/2}+1, m)$ with a gate cost of $O(n^2 \log n)$.

The outstanding matter is to find a lucky $a \in \mathbb{Z}_m^*$. How do we do that? Unfortunately, there is no efficient *deterministic* method known for finding a lucky $a \in \mathbb{Z}_m^*$. However, it's known that, for any $m \geq 2$, the number of lucky $a \in \mathbb{Z}_m^*$ is quite large.

Lemma 11.1. *For all $m = pq$, where p and q are distinct primes, at least half of the elements of $\mathbb{Z}_m^*$ are lucky.*

So what we can do is *randomly* select an $a \in \mathbb{Z}_m^*$. The selection will be lucky with probability at least $\frac{1}{2}$, and therefore the resulting procedure successfully factors m with probability at least $\frac{1}{2}$. We can repeat the process to boost the success probability.

So that's how an efficient algorithm for the order-finding problem can be used to factor a number efficiently. Now we can focus our attention on finding an efficient algorithm for order-finding.

12 Shor's algorithm for order-finding

In this section, I will first show you an efficient quantum algorithm for the order-finding problem. And then, in section 13, I will show you an efficient algorithm for factoring (using reduction to the order-finding algorithm from section 11.2).

Let's begin with an input instance to the order-finding problem, m and a, which are both n-bit numbers and for which $a \in \mathbb{Z}_m^*$. The goal of the algorithm is to determine $r = \mathrm{ord}_m(a)$.

12.1 Order-finding in the phase estimation framework

Our approach makes use of the framework of the phase-estimation algorithm (explained in Part II). Recall that, for this framework, we need a unitary operation and an eigenvector. Our unitary operation is $U_{a,m}$ defined such that, for all $b \in \mathbb{Z}_m^*$,

$$U_{a,m} |b\rangle = |ab\rangle \tag{144}$$

(where by ab we mean the product of a and b modulo m). As for the eigenvector, we'll begin with

$$|\psi_1\rangle = \tfrac{1}{\sqrt{r}}\Big(|1\rangle + \omega^{-1}|a\rangle + \omega^{-2}|a^2\rangle + \cdots + \omega^{-(r-1)}|a^{r-1}\rangle\Big) \tag{145}$$

$$= \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-j} \left|a^j\right\rangle, \tag{146}$$

where $\omega = e^{2\pi i(1/r)}$ (a primitive r-th root of unity).

To see why $|\psi_1\rangle$ is an eigenvector of $U_{a,m}$, consider what happens if we apply $U_{a,m}$ to $|\psi_1\rangle$. Note that $U_{a,m}$ maps each $|a^k\rangle$ to $|a^{k+1}\rangle$, where $|a^r\rangle = |1\rangle$. Therefore,

$$U_{a,m} |\psi_1\rangle = \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-j} \left|a^{j+1}\right\rangle \tag{147}$$

$$= \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega\, \omega^{-(j+1)} \left|a^{j+1}\right\rangle \tag{148}$$

$$= \omega |\psi_1\rangle. \tag{149}$$

Therefore $|\psi_1\rangle$ is an eigenvector of $U_{a,m}$ with eigenvalue $\omega = e^{2\pi i(1/r)}$. What's interesting about this is that, if we can estimate the eigenvalue's parameter $1/r$ with

sufficiently many bits of precision, then we can deduce what r is. In order to do this using the phase estimation algorithm, we need to efficiently simulate a multiplicity-controlled-$U_{a,m}$ gate and also to construct the state $|\psi_1\rangle$.

12.1.1 Multiplicity-controlled-$U_{a,m}$ gate

Here I will show you a rough sketch of how to efficiently compute a multiplicity-controlled-$U_{a,m}$ gate. Consider the mapping of the multiplicity-controlled-$U_{a,m}$ gate with ℓ control qubits.

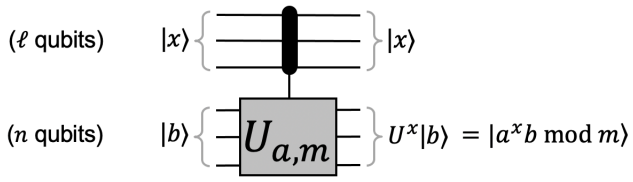

Figure 88: Multiplicity-controlled-$U_{a,m}$ gate.

For each $x \in \{0,1\}^\ell \equiv \{0,1,\ldots,2^\ell - 1\}$ and $b \in \mathbb{Z}_m^*$, this gate maps $|x\rangle\,|b\rangle$ to

$$|x\rangle\,(U_{a,m})^x\,|b\rangle = |x\rangle\,|a^x b\rangle \tag{150}$$

(which is equivalent to $U_{a,m}$ being applied x times).

We can compute this efficiently[15] (with respect to ℓ and n) by using the repeated squaring trick to exponentiate a. The number of gates is $O(\ell n \log n)$.

12.1.2 Precision needed to determine $1/r$

Now, how many bits of precision ℓ should we use in our phase estimation of $\frac{1}{r}$? It should be sufficient to uniquely determine $\frac{1}{r}$. We know that $r \in \{1,2,3,\ldots,m\}$ and the corresponding reciprocals are $\left\{\frac{1}{1},\frac{1}{2},\frac{1}{3},\ldots,\frac{1}{m}\right\}$. Here are the positions of these reciprocals on the real line.

[15]However, please note that, *in general*, if we can efficiently compute some unitary U then it does *not* always follow that we can compute a multiplicitly-controlled-U gate efficiently. The $U_{a,m}$ that arises here has special structure that permits this.

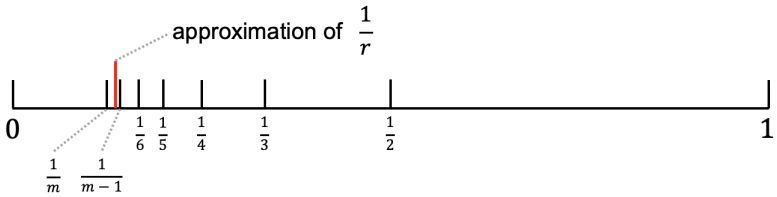

Figure 89: The positions of $\left\{\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{m}\right\}$ on the real line.

The smallest gap occurs between $\frac{1}{m}$ and $\frac{1}{m-1}$. The size of this gap is

$$\frac{1}{m-1} - \frac{1}{m} = \frac{m-(m-1)}{m(m-1)} = \frac{1}{m(m-1)} \approx \frac{1}{m^2}. \tag{151}$$

This means the approximation must be within $\frac{1}{2}\frac{1}{m^2}$ of $\frac{1}{r}$. Since m is an n-bit number, $m \approx 2^n$ and $\frac{1}{2}\frac{1}{m^2} \approx \frac{1}{2^{2n+1}}$ which corresponds to setting $\ell = 2n+1$.

12.1.3 Can we construct $|\psi_1\rangle$?

So far so good. But to efficiently implement the phase estimation algorithm, we also need to be able to efficiently create the eigenvector $|\psi_1\rangle$. Of course, if we already know what r is, then this is straightforward. But the algorithm does not know r at this stage; r is what the algorithm is trying to determine.

It seems very hard to construct this state without knowing what r is. This is a serious problem; it's not known how to construct this state directly. Instead of producing that state $|\psi_1\rangle$, our way forward will be to use a different state.

12.2 Order-finding using a random eigenvector $|\psi_k\rangle$

Let's consider alternatives to the eigenvector $|\psi_1\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-j} \left|a^j\right\rangle$ (with $\omega = e^{2\pi(\frac{1}{r})}$). Other eigenvectors of $U_{a,m}$ are

$$|\psi_2\rangle = \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-2j} \left|a^j\right\rangle \tag{152}$$

$$\vdots \qquad \vdots$$

$$|\psi_k\rangle = \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-kj} \left|a^j\right\rangle \tag{153}$$

$$\vdots \qquad \vdots$$

$$|\psi_r\rangle = \tfrac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \omega^{-rj} \left|a^j\right\rangle . \tag{154}$$

For each $k \in \{1, 2, \ldots, r\}$, the eigenvalue of $|\psi_k\rangle$ is $e^{2\pi(\frac{k}{r})}$.

Suppose that we are able to devise an efficient procedure that somehow produces a random sample from the r different eigenvectors (where each $|\psi_k\rangle$ occurrs with probability $\frac{1}{r}$). Can we deduce r from this? There are two versions of this scenario, depending on whether or not the procedure reveals k.

12.2.1 When k is known

First, let's suppose that the output of the procedure is the pair $(k, |\psi_k\rangle)$, with each possibility occurring with probability $\frac{1}{r}$. In this case, we can use the phase estimation algorithm (with the random $|\psi_k\rangle$) to get an approximation of $\frac{k}{r}$ and then divide the approximation by k to turn it into an approximation of $\frac{1}{r}$, and proceed from there as with the case of $|\psi_1\rangle$. The details of this case are straightforward.

12.2.2 When k is not known

Now, let's change the scenario slightly and assume that we have an efficient procedure that somehow generates a random $|\psi_k\rangle$ (uniformly distributed among the r possibilities) as output—but without revealing k. Can we still extract r from $|\psi_k\rangle$ alone?

Using the phase estimation algorithm, we can obtain a binary fraction $0.b_1b_2\ldots b_\ell$ that estimates $\frac{k}{r}$ within ℓ-bits of precision (where we can set the value of ℓ). But how can we determine k and r from this? This is a tricky problem, because we don't know

what k to divide by in order to turn an approximation of $\frac{k}{r}$ into an approximation of $\frac{1}{r}$. But there is a way to do this using the fact that we have an upper bound m on the denominator.

Let's carefully restate the situation as follows. We have an n-bit integer m and we are also given an ℓ-bit approximation $0.b_1b_2\ldots b_\ell$ of one of the elements of

$$\left\{ \frac{k}{r} \;\middle|\; \text{where } r \in \{1,2,\ldots,m\} \text{ and } k \in \{1,2,\ldots,r\} \right\}. \tag{155}$$

Our goal is to determine which element of the set is being approximated.

Consider the example where $m = 8$ (so the maximum denominator is m). Figure 90 shows all the *candidates* for $\frac{k}{r}$.

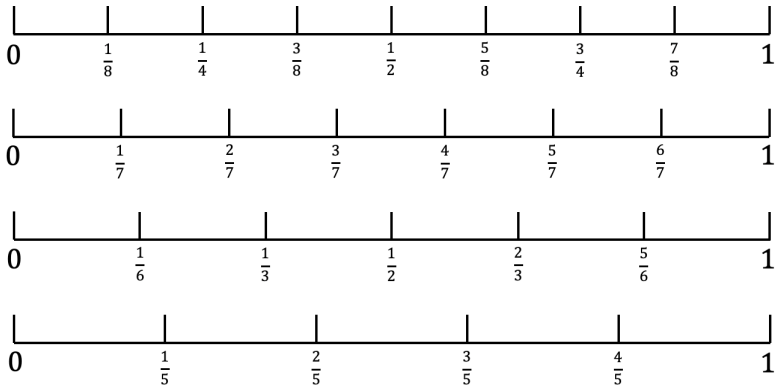

Figure 90: The set of candidates for $\frac{k}{r}$ in the $m = 8$ case.

Note that denominators 2, 3, and 4 are covered by the cases of 6 and 8 in the denominator. Figure 91 shows what all these candidates look like when they're combined on one line.

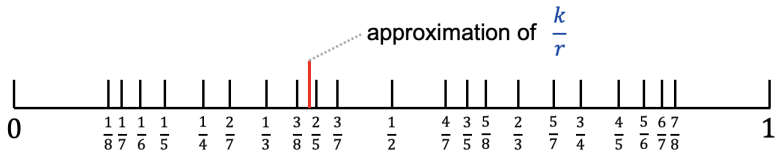

Figure 91: The set of candidates for $\frac{k}{r}$ in the $m = 8$ case (shown combined).

Now, suppose that we are able to obtain a 7-bit approximation of one of these candidates. Say it's 0.0110011. This number is exactly 51/128, but that's not one of the candidates. The rational number $\frac{2}{5}$ is the *unique* candidate within distance $\frac{1}{2^7}$ from 0.0110011.

In general, how good an approximation do we need in order to single out a unique $\frac{k}{r}$? This depends on the smallest gap among the set of candidates. It turns out that the gap is smallest at the ends: $\frac{1}{m-1} - \frac{1}{m} \approx \frac{1}{m^2}$. Therefore, setting $\ell = 2n+1$ provides the correct level of precision to guarantee that there is a unique rational number $\frac{k}{r}$ that's close to the approximation.

But how do we find the rational number? If m is an n-bit number then there are exponentially many candidates to search among. So a brute-force search is not going to be efficient.

It turns out that there is a known efficient algorithm that's tailor-made for this problem, called the *continued fractions algorithm*.[16] The continued-fractions algorithm enables us to determine k and r efficiently from a $(2n+1)$-bit approximation of $\frac{k}{r}$.

But perhaps you've already noticed that there is a major problem with all this: that of *reduced fractions*.

12.2.3 A snag: reduced fractions

The problem of reduced fractions is that different k and r can correspond to exactly the same number. For example, if $k = 34$ and $r = 51$ then $\frac{k}{r} = \frac{34}{51} = \frac{2}{3}$. There is no way to distinguish between $\frac{34}{51}$ and $\frac{2}{3}$. The continued fractions algorithm only provides a k and r in reduced form, which does not necessarily correspond to the actual r. If it happens that $\gcd(k, r) = 1$ then this problem does not occur. In that case, the fraction is already in reduced form.

Recall that, in our setting, we are assuming that k is uniformly sampled from the set $\{1, 2, \ldots, r\}$. What can we say about the probability of such a random k being relatively prime to r? This probability is the ratio of the size of the set $\mathbb{Z}_r^*$ to the set $\mathbb{Z}_r$. The ratio need not be constant, but is not too small. It is known to be at least $\Omega(1/\log\log r)$ in size. When the modulus is n-bits, this is at least $\Omega(1/\log n)$.

This means that $O(\log n)$ repetitions of the process is sufficient for the probability of a k that's relatively prime to r to arise with constant probability. Procedurally, in the case where $\gcd(k, r) > 1$, the result is an r that's smaller than the order (which can be detected because in such cases $a^r \bmod m \neq 1$). The $O(\log n)$ repetitions just introduces an extra log factor into the gate cost.

[16] The continued fractions algorithm was discovered in the 1600s by Christiaan Huygens, a Dutch physicist, astronomer, and mathematician who made several amazing contributions to diverse fields (for example, he discovered Saturn's rings, and he contributed to the theory of how light propagates).

In fact, we can do better than that. If we make two repetitions then there are two rational numbers $\frac{k_1}{r}$ and $\frac{k_2}{r}$ (where r is the order, which is unknown to us). Call the reduced fractions that we get from the continued fractions algorithm $\frac{k'_1}{r'_1}$ and $\frac{k'_2}{r'_2}$ (where r is a multiple of r'_1 and of r'_2). It turns out that, with constant probability, the *least common multiple* of r'_1 and r'_2 is r. So, with just *two* repetitions of the phase estimation algorithm, we can obtain the order r with constant success probability (independent of n), assuming we use an independent random eigenvector in each run.

12.2.4 Conclusion of order-finding with a random eigenvector

Now, lets step back and see where we are. We're trying to solve the order-finding problem using the phase-estimation algorithm.

We have a multiplicity-controlled-$U_{a,m}$ gate that's straightforward to compute efficiently.

Regarding the eigenvector, *if* we could construct the eigenvector state $|\psi_1\rangle$ *then* we could determine the order r from that. Unfortunately, we don't know how to generate the state $|\psi_1\rangle$ efficiently. We also saw that: if we could generate a randomly sampled pair $(k, |\psi_k\rangle)$ then we could also determine r from that. Unfortunately, we don't know how to generate such a random pair efficiently.

Next, we saw that: if we could generate a randomly sampled $|\psi_k\rangle$ (without the k) then we could still determine r from that. In that case, we had to do considerably more work. But, using the continued-fractions algorithm and some probabilistic analysis, we could make this work. So ... can we generate a random $|\psi_k\rangle$? Unfortunately, we don't know how to efficiently generate such a random $|\psi_k\rangle$ either.

Are we at a dead end? No, in fact we're almost at a solution! What I'll show next is that if, instead of generating a random $|\psi_k\rangle$ (with probability $\frac{1}{r}$ for each k), we can instead generate a *superposition* of the $|\psi_k\rangle$ (with amplitude $\frac{1}{\sqrt{r}}$ for each k) then we can determine r from this. And this is a state that we *can* generate efficiently.

12.3 Order-finding using a superposition of eigenvectors

Remember the phase estimation algorithm (explained in Part II), at the very end I discussed what happens if the target register is in a superposition of eigenvectors? In that case, the outcome is an approximation of the phase of a random eigenvector of the superposition, with probability the amplitude squared. Therefore, setting the

target register to state

$$\tfrac{1}{\sqrt{r}} \sum_{k=1}^{r} |\psi_k\rangle \tag{156}$$

results in the same outcome that occurs when we use a randomly generated eigenvector state—which is the case that we previously analyzed in section 12.2. So we have an efficient algorithm for order-finding using the superposition state in Eq. (156) in place of an eigenvector (it succeeds with constant probability, independent of n).

12.4 Order-finding without the requirement of an eigenvector

How can we efficiently generate the superposition state in Eq. (156)? In fact, this is extremely easy to do. To see how, expand the state as

$$\begin{aligned}
\tfrac{1}{\sqrt{r}} \sum_{k=1}^{r} |\psi_k\rangle \;&=\; \tfrac{1}{r}\Big(|1\rangle \;+\; \omega\,|a\rangle \;+\; \omega^2\,\big|a^2\big\rangle \;+\; \cdots \;+\; \omega^{r-1}\big|a^{r-1}\big\rangle\Big) \\
&\quad + \tfrac{1}{r}\Big(|1\rangle \;+\; \omega^2\,|a\rangle \;+\; \omega^4\,\big|a^2\big\rangle + \cdots + \omega^{2(r-1)}\big|a^{r-1}\big\rangle\Big) \\
&\quad + \tfrac{1}{r}\Big(|1\rangle \;+\; \omega^3\,|a\rangle \;+\; \omega^6\,\big|a^2\big\rangle + \cdots + \omega^{3(r-1)}\big|a^{r-1}\big\rangle\Big) \\
&\qquad \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \\
&\quad + \tfrac{1}{r}\Big(|1\rangle \;+\; \omega^r\,|a\rangle + \omega^{2r}\,\big|a^2\big\rangle + \cdots + \omega^{r(r-1)}\big|a^{r-1}\big\rangle\Big) \qquad (157) \\
&= |1\rangle \qquad (158)
\end{aligned}$$

where the simplification to $|1\rangle$ is a consequence of $\omega^r = 1$ and the fact that, for all $k \in \{1, 2, \ldots, r-1\}$, it holds that $1 + \omega^k + \omega^{2k} + \cdots + \omega^{(r-1)k} = 0$.

So we're left with $|1\rangle$. The n-bit binary representation of the number 1 is 00...01. So the superposition of eigenvectors in Eq. (156) is merely the computational basis state $|00...01\rangle$, which is indeed trivial to construct. The quantum part of the order-finding algorithm looks like this.

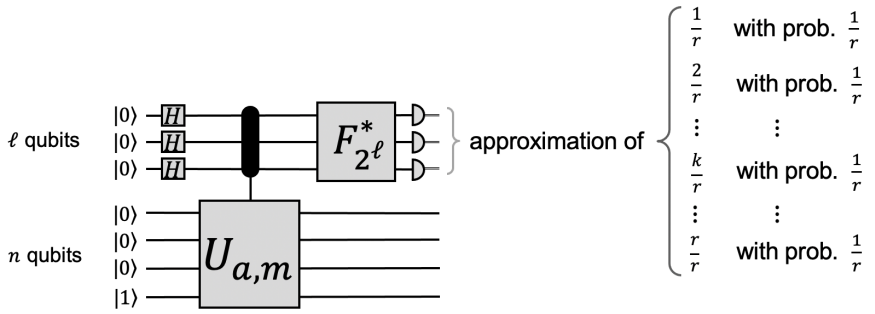

Figure 92: Caption.

The accuracy parameter ℓ is set to $2n+1$. The output of the measurement is then post-processed by the classical continued fractions algorithm, which produces a numerator k and denominator r. After two runs, taking the least common multiple of the two denominators, we obtain r, with constant probability. This constant probability is based on the phase estimation algorithm succeeding, in addition to r occurring via the continued fractions algorithm. The total gate cost is $O(n^2 \log n)$.

12.4.1 A technicality about the multiplicity-controlled-$U_{a,m}$ gate

In section 12.1.1, I glossed over some technical details about how the multiplicity-controlled-$U_{a,m}$ gate can be implemented. That gate is a unitary operation such that

$$|x, b\rangle \mapsto |x, a^x b\rangle, \tag{159}$$

for all $x \in \{0,1\}^\ell$ and $b \in \mathbb{Z}_m^*$. But our framework for computing classical functions in superposition (in Part II: 2.6.2 & 2.6.3) is different from this. What we showed there is that we can compute an f-query

$$|x, c\rangle \mapsto |x\rangle\,|f(x) \oplus c\rangle \tag{160}$$

in terms of a classical implementation of f.

There are two remedies. One is a separate construction for efficiently computing the mapping in Eq. (159). Such a efficient construction exists; however, I will not explain it here. Instead, I will show how to use a mapping of the form in Eq. (160) *instead of* the mapping in Eq. (159).

Define $f_{a,m} : \{0,1\}^\ell \to \{0,1\}^n$ as $f_{a,m}(x) = a^x$. Then, since there is a classical algorithm for efficiently computing $a^x b \bmod m$ in terms of (x, a, b) we can efficiently compute the mapping $|x, c\rangle \mapsto |x, f_{a,m}(x) \oplus c\rangle$. And then we can use the circuit in figure 93 instead of the circuit in figure 92.

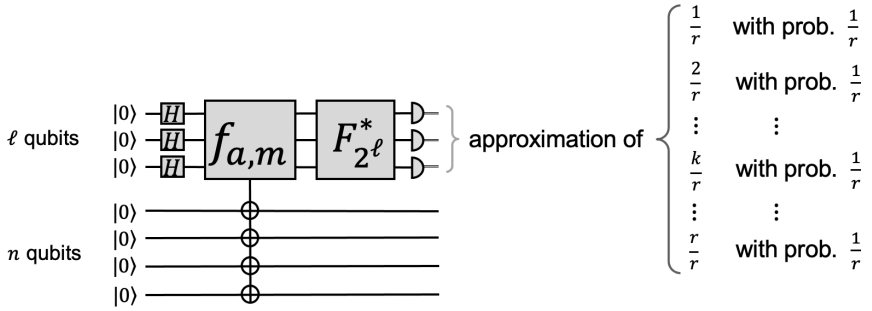

Figure 93: Caption.

The outputs of the two circuits are the same because, for both circuits, the state right after the multiplicity-controlled-$U_{a,m}$ gate (figure 92) and $f_{a,m}$-query (figure 93) is

$$\frac{1}{\sqrt{2^\ell}} \sum_{x \in \{0,1\}^\ell} |x\rangle \, |a^x\rangle \, . \tag{161}$$

13 Shor's factoring algorithm

Figure 94 describes the factoring algorithm (for factoring an n-bit number m), where the heavy lifting is done by a quantum subroutine for the order-finding problem (section 12). The algorithm is based on the reduction of the factoring problem to the order-finding problem which is explained in section 11.2.

```
choose a random a ∈ {1, 2, ..., m − 1}
g ← gcd(a, m)
if g = 1 then   (case where a ∈ Z*_m)
    r ← ord_m(a)   (the quantum part of the algorithm)
    if r is even then
        f ← gcd(a^{r/2} + 1, m)
        if f properly divides m then
            output f
        end if
    end if
else   (case where g > 1, so g is a proper divisor of m)
    output g
end if
```

Figure 94: Factoring algorithm (using quantum algorithm for order-finding as a subroutine).

The algorithm samples uniformly from $\mathbb{Z}_m^*$ by sampling an a uniformly from the simpler set $\{1, 2, \ldots, m-1\}$ and then checking whether or not $\gcd(a, m) = 1$. If $\gcd(a, m) = 1$ then $a \in \mathbb{Z}_m^*$ and the algorithm proceeds. If $\gcd(a, m) > 1$ then, although the algorithm could randomly sample another element from $\{1, 2, \ldots, m-1\}$, that's not necessary because, in that event, $\gcd(a, m)$ is a proper divisor of m.

In the event that $a \in \mathbb{Z}_m^*$, the probability that a is lucky (as defined in section 11.2) is at least $\frac{1}{2}$. If a is lucky then the algorithm computes $\gcd(a^{r/2} + 1, m)$ to obtain a factor of m. This part can be computed by repeated squaring and Euclid's algorithm.

The implementation cost of this algorithm is $O(n^2 \log n)$ gates. Although we only analysed this algorithm for the case where m is the product of two distinct primes, it turns out that this factoring algorithm succeeds with probability at least $\frac{1}{2}$ for *any* number m that's not a prime power. For the prime power case (where $m = p^k$ for a prime p) there's a simple classical algorithm. That algorithm can be run as a preprocessing step to the algorithm in figure 94.

14 Grover's search algorithm

In this section, I will show you Grover's search algorithm, which solves several search problems quadratically faster than classical algorithms. Although quadratic improvement is less dramatic than the exponential improvement possible by Shor's algorithms for factoring and discrete log, Grover's algorithm is applicable to a very large number of problems.

Let's begin by defining an abstract black-box search problem. You are given a black-box computing a function $f : \{0,1\}^n \to \{0,1\}$. Here's what the black-box looks like in the classical case (in reversible form) and the quantum case.

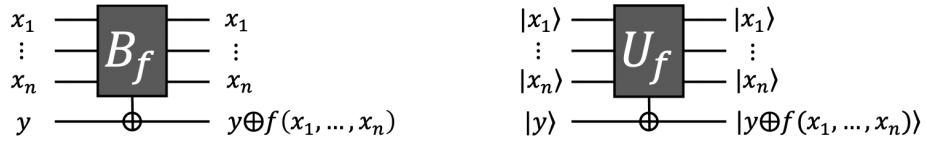

Figure 95: Classical reversible f-query (left) and quantum f-query (right).

In the notation of figure 95, $B_f : \{0,1\}^{n+1} \to \{0,1\}^{n+1}$ is a bijection and U_f is a unitary operation acting on $n+1$ qubits.

The goal is to find an $x \in \{0,1\}^n$ such that $f(x) = 1$. This is called a *satisfying assignment* (or *satisfying input*) because it is a setting of the logical variables so that the function value is "true" (denoted by the bit 1). It is possible for the function to be the constant zero function, in which case there does not exist a satisfying assignment. In that case, the solution to the problem is to report that f is "unsatisfiable". There is also a related *decision problem*, where the goal is to simply determine whether or not f is satisfiable. For that problem, the answer is one bit.

How many classical queries are needed to solve this search problem? It turns out the 2^n queries are necessary in the worst case. If f is queried in $2^n - 1$ places and the output is 0 for each of these queries then there's no way of knowing whether f is satisfiable or not without querying f in the last place.

Remember back in Part I we saw that there can be a dramatic difference between the classical deterministic query cost and the classical probabilistic cost? For the constant-vs-balanced problem, we saw that exponentially many queries are necessary for for an exact solution, but that there is a classical probabilistic algorithm that succeeds with probability (say) $\frac{3}{4}$ using only a constant number of queries.

So how does the probabilistic query cost work out for this search problem? What's the classical probabilistic query cost if we need only succeed with probability $\frac{3}{4}$? It

turns out that order 2^n queries are still needed. Suppose that f is satisfiable in exactly one place that was set at random. Then, by querying f in random places, on average, the satisfying assignment will be found after searching half of the inputs. So the *expected number* of queries is around $\frac{1}{2}2^n$. Based on these ideas, it can be proven that a constant times 2^n queries are *necessary* to find a satisfying x with success probability at least $\frac{3}{4}$. So, asymptotically, this problem is just as hard for probabilistic algorithms as for deterministic algorithms.

What's the quantum query cost? We will see that it's $O(\sqrt{2^n})$ by Grover's algorithm, which is the subject of the rest of this section. Notice that the query cost is still exponential. The difference is only by a factor of 2 in the exponent. But this speed-up should not be dismissed. If you're familiar with algorithms then you are probably aware of clever fast sorting algorithms that make $O(n \log n)$ steps instead of the $O(n^2)$ steps that you get if you use the most obvious algorithm. The improvement is only quadratic, but for large n this quadratic improvement can make a huge difference. Similarly, in signal processing, there's a famous *Fast Fourier Transform algorithm*, whose speed-up is again quadratic over trivial approaches to the same problem.

A natural application of Grover's algorithm is when a function $f : \{0,1\}^n \to \{0,1\}$ can be computed in polynomial-time, so there is a quantum circuit of size $O(n^c)$ that implements an f-query.

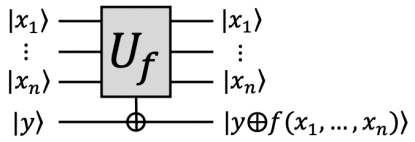

Figure 96: Implementation of a quantum f-query by a quantum circuit.

We can use Grover's algorithm to find a satisfying assignment with a circuit of size $O(\sqrt{2^n}n^c)$ (which is $O(\sqrt{N}\log^c N)$ if $N = 2^n$). For many of the so-called NP-complete problems, this is quadratically faster than the best classical algorithm known.

The order-finding algorithm and factoring algorithm use interesting properties of the Fourier transform. Grover's algorithm uses properties of simpler transformations.

14.1 Two reflections is a rotation

Consider the two-dimensional plane. There's a transformation called a *reflection about a line*. The way it works is: every point on one side of the line is mapped to a mirror image point on the other side of the line, as illustrated in figure 97.

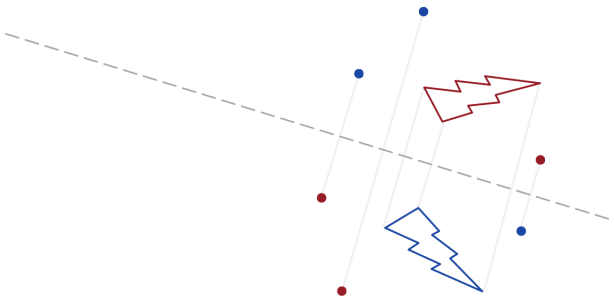

Figure 97: The reflection about the line of each red item is shown in blue.

Now suppose that we add a second line of reflection, with angle θ between the lines.

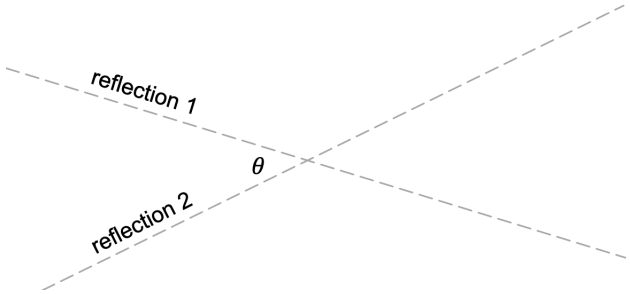

Figure 98: Two refection lines with angle θ between them.

What happens if you compose the two reflections? That is, you apply reflection 1 and then reflection 2? Let the *origin* be where the two lines intersect and think of each point as a vector to that point. Now, start with any such vector.

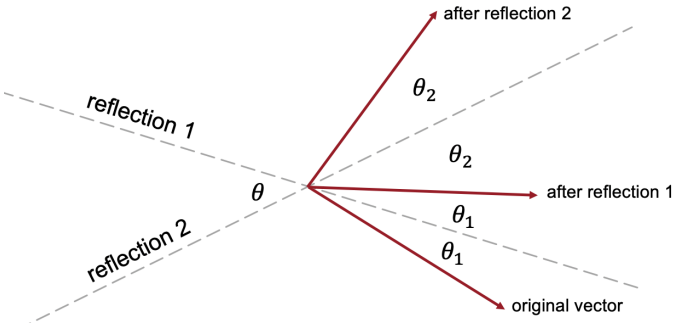

Figure 99: Effect of reflection 1 followed by reflection 2 on a vector.

This vector makes some angle—call it θ_1—with the first line. After we perform the first reflection, the reflected vector makes the same angle θ_1 on the other side of the

line. The reflected vector also makes some angle with the second line. Call that angle θ_2. Notice that $\theta_1 + \theta_2 = \theta$. Now, if we apply the second reflection then the twice reflected vector makes an angle θ_2 on the other side of the second line. So what happened to the original vector as a result of these two reflections? It has been rotated by angle 2θ.

Notice that the amount of the rotation depends only on θ, it does not on depend on what θ_1 and θ_2 are. This is an illustration of the following lemma.

Lemma 14.1. *For any two reflections with angle θ between them, their composition is a rotation by angle 2θ.*

The picture in Figure 99 is intuitive, but is not a rigorous proof. A rigorous proof can be obtained by expressing each reflection operation as a 2×2 matrix, and then multiplying the two matrices. The result will be a rotation matrix by angle 2θ. The details of this are left as an exercise.

Exercise 14.1. *Prove Lemma 14.1.*

This simple geometric result will be a key part of Grover's algorithm.

14.2 Overall structure of Grover's algorithm

Grover's algorithm is based on repeated applications of three basic operations.

One operation is the f-query that we're given as a black-box.

Another operation that will be used, is one that we'll call U_0 that is defined as, for all $a \in \{0,1\}^n$ and $b \in \{0,1\}$,

$$U_0 \left|a\right\rangle \left|b\right\rangle = \left|a\right\rangle \left|b \oplus [a = 0^n]\right\rangle, \tag{162}$$

where $[a = 0^n]$ denotes the predicate

$$[a = 0^n] = \begin{cases} 1 & \text{if } a = 0^n \\ 0 & \text{if } a \neq 0^n. \end{cases} \tag{163}$$

In other words, in the computational basis, U_0 flips the target qubit if and only if the first n qubits are all $\left|0\right\rangle$. Our notation for the U_0 gate is shown in Figure 100 (which also shows one implementation in terms of an n-ary Toffoli gate).

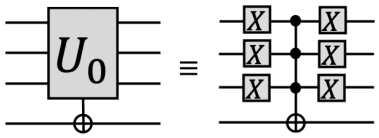

Figure 100: Notation for U_0 gate (implementable by Pauli X gates and an n-fold Toffoli gate).

A third operation that we'll use is an n-qubit Hadamard gate, by which we mean $H^{\otimes n}$ (the n-fold tensor product of 1-qubit Hadamard gates). However, in this context, we will denote this gate as H (without the superscript $\otimes n$).

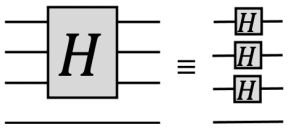

Figure 101: Notation for $H^{\otimes n}$ gate.

Also, we will use the U_f and U_0 gates with the target qubit set state $|-\rangle$, which causes the function value to be computed in the phase as, for all $x \in \{0,1\}^n$,

$$U_f |x\rangle |-\rangle = (-1)^{f(x)} |x\rangle |-\rangle \tag{164}$$

$$U_0 |x\rangle |-\rangle = (-1)^{[x=00\ldots0]} |x\rangle |-\rangle . \tag{165}$$

Here's what the main part of Grover's algorithm looks like in terms of the three basic operations.

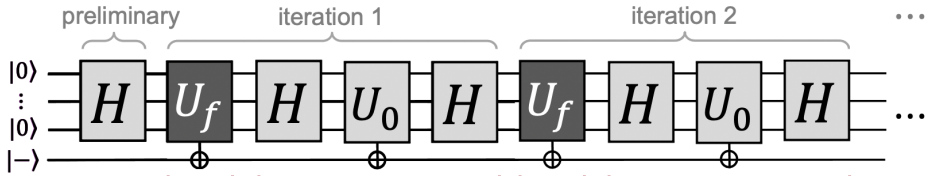

Figure 102: Quantum circuit for Grover's algorithm.

It starts with a preliminary step, which is to apply H to the state $|0...0\rangle$.

Next, the sequence of operations U_f, H, U_0, H is applied several times. Each instance of this sequence, HU_0HU_f, is called an *iteration*. After the iterations, the state of the first n qubits is measured (in the computational basis) to yield an $x \in \{0,1\}^n$. Then a classical query on input x is performed to check if $f(x) = 1$. If it is then the algorithm has solved the search problem.

Here's a summary of the algorithm in pseudocode, where k is the number of iterations performed.

1: construct state $H|00\ldots0\rangle|-\rangle$
2: **repeat** k **times:**
3: apply $-HU_0HU_f$ to the state
4: measure state, to get $x \in \{0,1\}^n$, and check if $f(x) = 1$

Figure 103: Pseudocode description of Grover's algorithm (including classical post-processing).

You might notice that, in the pseudocode, a minus sign in the iteration $-HU_0HU_f$. This is just a global phase, which has no affect on the true quantum state, but it makes the upcoming geometric picture of the algorithm nicer.

What is k, the number of iterations, set to? This is an important issue that will be addressed later on.

What I will show next is that U_f is a reflection and $-HU_0HU_f$ is also a reflection (and then we'll apply Lemma 14.1 about two reflections being a rotation). The function f partitions $\{0,1\}^n$ into two subsets, A_0 and A_1, defined as

$$A_1 = \{x \in \{0,1\}^n : f(x) = 1\} \tag{166}$$

$$A_0 = \{x \in \{0,1\}^n : f(x) = 0\}. \tag{167}$$

The set A_1 consists of the satisfying inputs to f and A_0 consists of the unsatisfying inputs to f. Let $s_1 = |A_1|$ and $s_0 = |A_0|$. Then $s_1 + s_0 = N$ (where $N = 2^n$).

Note that, if f has many satisfying assignments (for example if half the inputs are satisfying) then it's easy to find one with high probability by just random guessing. The interesting case is where there are very few satisfying assignments to f. To understand Grover's algorithm, it's useful to focus our attention on the case where s_1 is at least 1 but much smaller than N (for example, if $s_1 = 1$ or a constant). In that case, finding a satisfying assignment is nontrivial.

Now let's define these two quantum states

$$|A_1\rangle = \tfrac{1}{\sqrt{s_1}} \sum_{x \in A_1} |x\rangle \tag{168}$$

$$|A_0\rangle = \tfrac{1}{\sqrt{s_0}} \sum_{x \in A_0} |x\rangle\,. \tag{169}$$

The states $|A_1\rangle$ and $|A_0\rangle$ make sense as orthonormal vectors as long as $0 < s_1 < N$.

Consider the two-dimensional space spanned by $|A_1\rangle$ and $|A_0\rangle$. Notice that $H|0...0\rangle$ (the state of the first n qubits right after the preliminary Hadamard operations) resides within this two-dimensional space, since

$$\frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} |x\rangle = \sqrt{\frac{s_0}{N}}\left(\frac{1}{\sqrt{s_0}} \sum_{x\in A_0} |x\rangle\right) + \sqrt{\frac{s_1}{N}}\left(\frac{1}{\sqrt{s_1}} \sum_{x\in A_1} |x\rangle\right) \tag{170}$$

$$= \sqrt{\frac{s_0}{N}}\,|A_0\rangle + \sqrt{\frac{s_1}{N}}\,|A_1\rangle\,. \tag{171}$$

Let θ be the angle between $|A_0\rangle$ and $H|0...0\rangle$, as illustrated in figure 104.

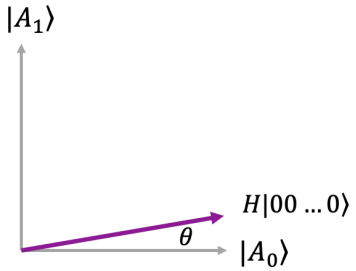

Figure 104: The state $H|0...0\rangle$ within the subspace spanned by $|A_0\rangle$ and $|A_1\rangle$.

Then $\cos(\theta) = \sqrt{\frac{s_0}{N}}$ and $\sin(\theta) = \sqrt{\frac{s_1}{N}}$. In the case where $1 \leq s_1 \ll N$, the angle θ is small and approximately $\sqrt{\frac{s_1}{N}}$.

If the system were in state $|A_0\rangle$ then measuring (in the computational basis) would result in a random unsatisfying assignment. If the system were in state $|A_1\rangle$ then measuring would result in a random satisfying assignment. After the preliminary step, the system is actually in state $H|0...0\rangle$, and measuring at that point would result in an unsatisfying assignment with high probability.

A useful goal is to somehow rotate the state $H|0...0\rangle$ towards $|A_1\rangle$. Note that, although the states $|A_0\rangle$ and $|A_1\rangle$ exist somewhere within the 2^n-dimensional space of all states of the first n qubits, the algorithm does not have information about what they are. It's not possible to directly apply a rotation matrix in the two-dimensional space without knowing what $|A_0\rangle$ and $|A_1\rangle$ are.

The key idea in Grover's algorithm is that the iterative step $-HU_0HU_f$ consists of two reflections in this two-dimensional space, whose composition results in a rotation. In the analysis below, we omit the last qubit, whose state is $|-\rangle$ and doesn't change. We write $U_f|x\rangle = (-1)^{f(x)}|x\rangle$ as an abbreviation of $U_f|x\rangle|-\rangle = (-1)^{f(x)}|x\rangle|-\rangle$ (and similarly for U_0).

The first reflection is U_f, which is a reflection about the line in the direction of $|A_0\rangle$. Why? Because $U_f |A_0\rangle = |A_0\rangle$ and $U_f |A_1\rangle = -|A_1\rangle$.

The second reflection is $-HU_0H$. This is a reflection about the line in the direction of $H|0...0\rangle$. To see this requires a little bit more analysis.

Lemma 14.2. *$-HU_0H$ is a reflection about the line passing through $H|0...0\rangle$.*

Proof. First, note that $(-HU_0H)H|0...0\rangle = H|0...0\rangle$ because

$$(-HU_0H)H|0...0\rangle = -HU_0|0...0\rangle \tag{172}$$
$$= -H(-|0...0\rangle) \tag{173}$$
$$= H|0...0\rangle. \tag{174}$$

Next, consider any vector v that is orthogonal to $H|0...0\rangle$. Then Hv is orthogonal to $|0...0\rangle$. Therefore, $U_0Hv = Hv$, from which it follows that $-HU_0Hv = -v$. We have shown that the effect of $-HU_0H$ on any vector w is to leave the component of w in the direction of $H|0...0\rangle$ fixed and to multiply any orthogonal component by -1. □

Now, since each iteration $-HU_0HU_f$ is a composition of two reflections, and the angle between them is θ, the effect of $-HU_0HU_f$ is a rotation by 2θ. And the effect of k iterations is to rotate by $k2\theta$, as illustrated in figure 105.

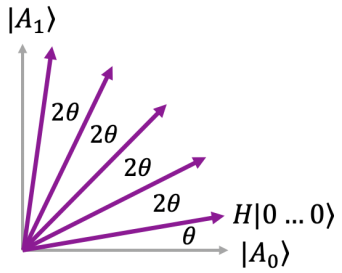

Figure 105: The state $|0...0\rangle$ rotates by angle 2θ after each iteration.

What should k be set to? How many iterations should the algorithm make to move the state close to $|A_1\rangle$? Let us begin by focusing on the case where there is one unique satisfying input to f.

14.3 The case of a unique satisfying input

Consider the case where there is one satisfying assignment (that is, where $s_1 = 1$). In that case, $\sin(\theta) = \frac{1}{\sqrt{N}}$. Since $\frac{1}{\sqrt{N}}$ is small, we have $\theta \approx \frac{1}{\sqrt{N}}$.

Then setting $k = \frac{\pi}{4}\sqrt{N}$ (rounded to the nearest integer) causes the total rotation to be approximately $\frac{\pi}{2}$ radians (that is, 90 degrees). The resulting state will not be exactly $|A_1\rangle$. In most cases, $|A_1\rangle$ will be somewhere between two rotation angles. But the state will be close to $|A_1\rangle$, so measuring will produce a satisfying assignment with high probability (certainly at least $\frac{3}{4}$).

The resulting algorithm finds the satisfying x with high probability and makes $O(\sqrt{N})$ queries to f (one per iteration).

14.4 The case of any number of satisfying inputs

What if f has more than one satisfying input? One might be tempted to think intuitively that the "hardest" case for the search is where there is just one satisfying input. Finding a needle in a haystack is harder if the haystack contains a single needle. So how well does the algorithm with $k \approx \frac{\pi}{4}\sqrt{N}$ iterations do when there are more satisfying inputs? Unfortunately, it's not very good.

Suppose that there are four satisfying inputs. Then $\theta \approx \frac{2}{\sqrt{N}}$ (double the value in the case of one satisfying input), which causes the rotation to overshoot. The total rotation is by 180 degrees instead of 90 degrees. The state starts off almost orthogonal to $|A_1\rangle$. Then, midway, it gets close to $|A_1\rangle$. And then it keeps on rotating and becomes almost orthogonal to $|A_1\rangle$ again.

When there are four satisfying inputs, the ideal number of iterations is half the number for the case of one satisfying input. More generally, when there are k satisfying inputs, the ideal number of iterations is $k \approx \frac{\pi}{4}\sqrt{\frac{N}{k}}$. If we know the number of satisfying inputs in advance then we can tune k appropriately.

But suppose we're given a black box for f without any information about the number of satisfying inputs that f. What do we do then? For any particular setting of k, it's concievable that number of satisfying inputs to f is such that the total rotation after k iterations is close to a multiple of 180 degrees.

I will roughly sketch an approach that resolves this by setting k to be a random selection from the set $\left\{1, 2, \ldots, \lceil \frac{\pi}{4}\sqrt{N} \rceil\right\}$. To see how this works, let's begin by considering again the case where there is one satisfying input. Figure 106 shows the success probability as a function of the number iterations k, for any $k \in \left\{1, 2, \ldots, \lceil \frac{\pi}{4}\sqrt{N} \rceil\right\}$.

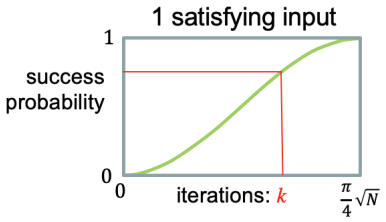

Figure 106: Success probability of measurement as a function of k, the number of iterations.

The curve is sinusoidal (of the form of a sine function). In the case of a single satisfying input, setting $k = \lceil \frac{\pi}{4}\sqrt{N} \rceil$ results in a success probability close to 1. But if k is set randomly then the success probability is the average value of the curve. By the symmetry of the sine curve, the area under the curve is half of the area of the box, so the success probability is $\frac{1}{2}$.

Now, let's consider the cases of two, three or four satisfying assignments. These cases result in a larger θ, and the corresponding success probability curves are shown in figure 107 (they are sinusoidal curves corresponding to more total rotation).

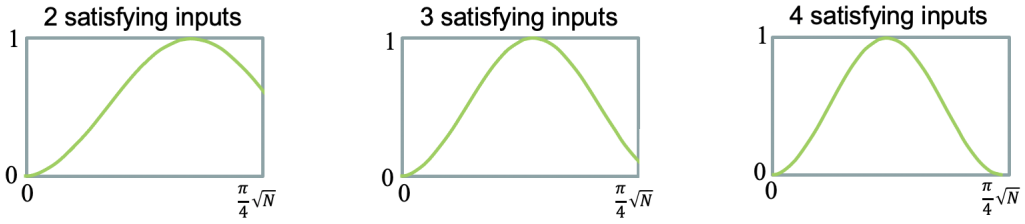

Figure 107: Success probability function in cases of 2, 3, and 4 satisfying inputs.

By inspection, the area under each of these curves is at least $\frac{1}{2}$. Each curve is a 90 degree rotation, for which the average is $\frac{1}{2}$, followed by another rotation less than or equal to 90 degrees, for which the average can be seen to be more than $\frac{1}{2}$.

But there are cases where the average is less than $\frac{1}{2}$. Let's look at the case of six satisfying inputs.

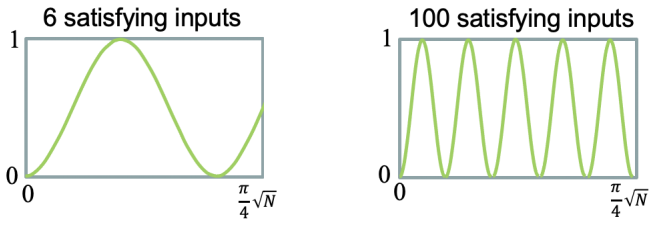

Figure 108: Success probability function in cases of 6 and 100 satisfying inputs.

There are two 90-degree rotations, followed by a rotation less than 90 degrees for which the average is less than $\frac{1}{2}$. The average for the whole curve can be calculated to be around 0.43. For larger numbers of satisfying inputs, the proportion of the domain associated with the last partial rotation becomes small enough so as to be inconsequential.

The above roughly explains why choosing k randomly within the range results in a success probability that's at least 0.43, regardless of the number of satisfying assignments. That's one way to solve the case of an unknown number of satisfying inputs to f.

To summarize, we have a quantum algorithm that makes $O(\sqrt{N})$ queries and finds a satisfying assignment with constant probability. By repeating the process a constant number of times, the success probability can be made close to 1.

What happens if there is no satisfying assignment? In that case, the algorithm will not find a satisfying assignment in any run and outputs "unsatisfiable".

There's a refined version of the search algorithm that stops early, depending on the number of satisfying inputs. The number of queries is $O\left(\sqrt{\frac{N}{s_1}}\right)$, where s_1 is the number of satisfying inputs. This refined algorithm does not have information about the value of s_1. When the algorithm is run, we don't know in advance for how long it will need to run. I will not get into the details of this refined algorithm here here.

Finally, you might wonder if Grover's search algorithm can be improved. Could there be a better quantum algorithm that performs the black-box search with fewer than order $\sqrt{N}$ queries? Say, $O(N^{1/3})$ or $O(\log N)$ queries? Actually, no. It can be proven that the above query costs are the best possible, as explained in the next section.

15 Optimality of Grover's search algorithm

In this section, I will show you a proof that Grover's search algorithm is optimal in terms of the number of black-box queries that it makes.

Theorem 15.1 (optimality of Grover's algorithm)**.** *Any quantum algorithm for the search problem for functions of the form* $f : \{0,1\}^n \to \{0,1\}$ *that succeeds with probability at least* $\frac{3}{4}$ *must make* $\Omega\left(\sqrt{2^n}\right)$ *queries.*

In this proof, I make two simplifying assumptions. First, I assume that the function is always queried in the phase (as occurs with Grover's algorithm).

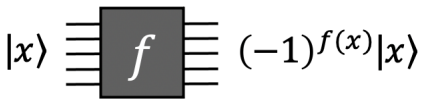

Figure 109: Query of f in the phase.

Second, I assume that the algorithm uses only n qubits. That is, no ancilla qubits are used. The purpose of these simplifying assumptions is to reduce clutter. It is very straightforward to adjust this proof to the general case, where the function is queried in the standard way, and where the quantum algorithm is allowed to use ancilla qubits. The more general proof will be just a more cluttered-up version of the proof that I'm about to explain. Nothing of importance is being swept under the rug in this simplified proof.

Under our assumptions, any quantum algorithm that makes k queries is a circuit of the following form.

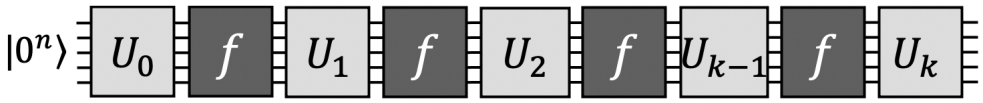

Figure 110: Form of a k query quantum circuit.

The initial state of the n qubits is $|0^n\rangle$. Then an arbitrary unitary operation U_0 is applied, which can create any pure state as the input to the first query. Then the first f-query is performed. Then an arbitrary unitary operation U_1 is applied. Then the second f-query is performed. And so on, up the k-th f-query, followed by an arbitrary unitary operation U_k. The *success probability* is the probability that a measurement of the final state results in a satisfying input of f.

The quantum algorithm is the specification of the unitaries $U_0, U_1, \ldots, U_k$. The *input* to the algorithm is the black-box for f, which is inserted into the k query gates. The *quantum output* is the final state produced by the quantum circuit.

The overall approach will be as follows. For every $r \in \{0,1\}^n$, define f_r as

$$f_r(x) = \begin{cases} 1 & \text{if } x = r \\ 0 & \text{otherwise.} \end{cases} \tag{175}$$

We'll show that, for any algorithm that makes asymptotically fewer than $\sqrt{2^n}$ queries, it's success probability is very low for at least one f_r.

Assume we have some k-query algorithm, specified by $U_0, U_1, \ldots, U_k$. Suppose that we insert one of the functions f_r into the query gate.

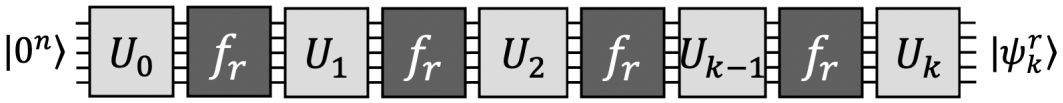

Figure 111: Quantum circuit making k queries to function f_r.

Call the final state $|\psi_k^r\rangle$. The superscript r is because this state depends on which r is selected. The subscript k is to emphasize that k queries are made.

Now consider the exact same algorithm, run with the identity operation in each query gate.

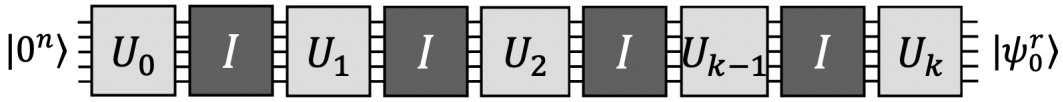

Figure 112: Quantum circuit with I substituted for the k queries.

Whenever the identity is substituted into a query gate, we'll call that a *blank query*. Let's call the final state produced by this $|\psi_0^r\rangle$. Although the superscript r is redundant for this state (because the state is not a function of r), it's harmless and it will be convenient to keep this superscript r in our notation. The circuit in figure 111 corresponds to the function f_r. The circuit in figure 112 corresponds to the zero function (the function that's zero everywhere).

What we are going to show is that, for some $r \in \{0,1\}^n$, the Euclidean distance between $|\psi_k^r\rangle$ and $|\psi_0^r\rangle$ is upper bounded as

$$\big\| |\psi_k^r\rangle - |\psi_0^r\rangle \big\| \leq \frac{2k}{\sqrt{2^n}}. \tag{176}$$

This implies that, if k is asymptotically smaller than $\sqrt{2^n}$ then the bound asymptotically approaches zero. If $\| \, |\psi_k^r\rangle - |\psi_0^r\rangle \, \|$ approaches zero then the two states are not distinguishable in the limit. This implies that the algorithm cannot distinguish between f_r and the zero function with constant probability.

Now let's consider the quantum circuits in figures 111 and 112, along with some intermediate circuits.

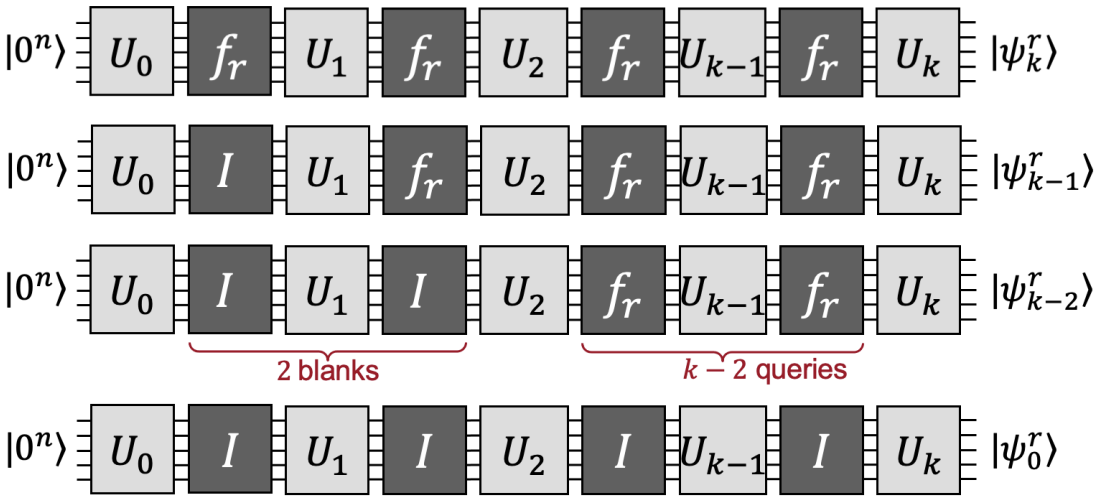

Figure 113: Quantum circuits with i blanks, followed by $k-i$ queries to f_r (for $i \in \{0, 1, \ldots, r\}$).

The circuit on top makes all k queries to f_r, and recall that we denote the final state that it produces as $|\psi_k^r\rangle$. The next circuit uses the identity in place of the *first* query and makes the remaining $k-1$ queries to f_r. Call the final state of this $|\psi_{k-1}^r\rangle$. The next circuit uses the identity in place of the first *two* queries and makes the remaining $k-2$ queries to f_r. Call the final state that this produces $|\psi_{k-2}^r\rangle$. And continue for $i = 3, \ldots, k$ with circuits using the identity in place of the first i queries and then making the remaining $k-i$ queries to f_r. Call the final states $|\psi_{k-i}^r\rangle$.

Note that each query where the identity is substituted (blanks query) reveals no information about r.

Recall that our goal is to show that the $\| \, |\psi_k^r\rangle - |\psi_0^r\rangle \, \|$ is small. Notice that we can express the difference between these states as the telescoping sum

$$|\psi_k^r\rangle - |\psi_0^r\rangle = \big(|\psi_k^r\rangle - |\psi_{k-1}^r\rangle\big) + \big(|\psi_{k-1}^r\rangle - |\psi_{k-2}^r\rangle\big) + \cdots + \big(|\psi_1^r\rangle - |\psi_0^r\rangle\big) \qquad (177)$$

which implies that

$$\big\||\psi_k^r\rangle - |\psi_0^r\rangle\big\| \leq \big\||\psi_k^r\rangle - |\psi_{k-1}^r\rangle\big\| + \big\||\psi_{k-1}^r\rangle - |\psi_{k-2}^r\rangle\big\| + \cdots + \big\||\psi_1^r\rangle - |\psi_0^r\rangle\big\|. \qquad (178)$$

Next, we'll upper bound each term $\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \|$ in Eq. 178. To understand the Euclidean distance between $|\psi_i^r\rangle$ and $|\psi_{i-1}^r\rangle$, let's look at the circuits that produce them.

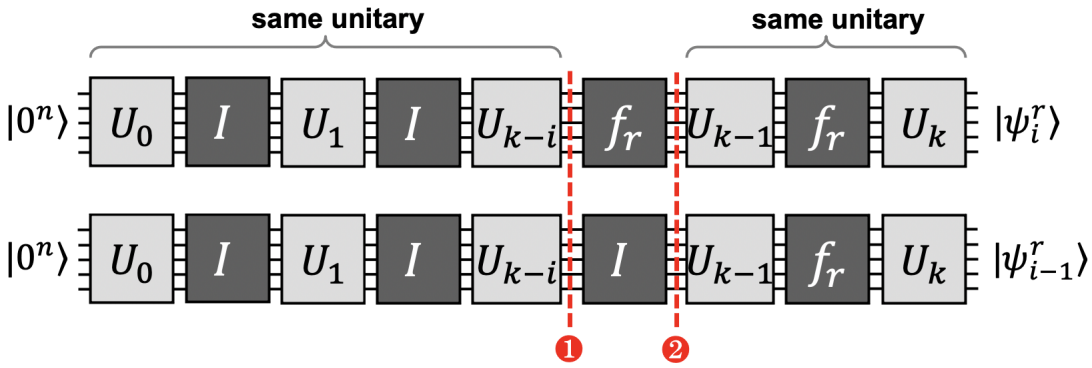

Figure 114: Circuit with $k-i$ blanks and i queries (top) and $k-i+1$ blanks and $i-1$ queries (bottom).

Notice that both computations are identical for the first $k-i$ blank queries. We can write the state at stage ❶ (for both circuits) as

$$\sum_{x\in\{0,1\}^n} \alpha_{i,x} |x\rangle , \tag{179}$$

where of course the state is a unit vector in Euclidean norm (a.k.a, the 2-norm)

$$\sum_{x\in\{0,1\}^n} |\alpha_{i,x}|^2 = 1. \tag{180}$$

These states make sense for each $i \in \{1, \ldots, k\}$, and they are independent of r.

It's interesting to consider what happens to the state at the next step. For the bottom computation, nothing happens to the state, since the identity is performed in the query. For the top computation, a query to f_r is performed. Since f_r takes value 1 only at the point r, the f_r-query in the phase negates the amplitude of $|r\rangle$ and has no effect on the other amplitudes of the state. Therefore, the state of the top computation at stage ❷ is

$$\sum_{x\in\{0,1\}^n} (-1)^{[x=r]} \alpha_{i,x} |x\rangle . \tag{181}$$

The difference between the state in Eq. (179) and the state in Eq. (181) is only in the amplitude of $|r\rangle$, which is negated in Eq. (181). So, at stage ❷ the Euclidean distance between the states of the two circuits is $|\alpha_{r,i} - (-\alpha_{r,i})| = 2|\alpha_{r,i}|$.

Now let's look at what the rest of the two circuits in figure 114 do. The remaining steps for each circuit are the same unitary operation. Since unitary operations preserve Euclidean distances, this means that

$$\left\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \right\| = 2|\alpha_{i,r}|. \tag{182}$$

And this holds for all $i \in \{1, \ldots, k\}$.

Now consider the average Euclidean distance between $|\psi_k^r\rangle$ and $|\psi_0^r\rangle$, averaged over all n-bit strings r. We can upper bound this as

$$\frac{1}{2^n} \sum_{r \in \{0,1\}^n} \left\| |\psi_k^r\rangle - |\psi_0^r\rangle \right\| \leq \frac{1}{2^n} \sum_{r \in \{0,1\}^n} \left(\sum_{i=1}^{k} \left\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \right\| \right) \quad \text{telescoping sum} \tag{183}$$

$$= \frac{1}{2^n} \sum_{r \in \{0,1\}^n} \left(\sum_{i=1}^{k} 2|\alpha_{i,r}| \right) \quad \text{by Eq. (182)} \tag{184}$$

$$= \frac{1}{2^n} \sum_{i=1}^{k} 2 \left(\sum_{r \in \{0,1\}^n} |\alpha_{i,r}| \right) \quad \text{reordering sums} \tag{185}$$

$$\leq \frac{1}{2^n} \sum_{i=1}^{k} 2 \left(\sqrt{2^n} \right) \quad \text{Cauchy-Schwarz} \tag{186}$$

$$= \frac{2k}{\sqrt{2^n}}. \tag{187}$$

In Eq. (186) we are using the the Cauchy-Schwarz inequality, which implies that, for any vector whose 2-norm is 1, its maximum possible 1-norm is the square root of the dimension of the space, which in this case is $\sqrt{2^n}$.

Since an average of Euclidean distances is upper bounded by $2k/\sqrt{2^n}$, there must exist an r for which the Euclidean distance upper bounded by this amount.

Since the denominator is $\sqrt{2^n}$ and the numerator is $2k$, it must be the case that k is proportional to $\sqrt{2^n}$, in order for this Euclidean distance to be a constant. And the Euclidean must be lower bounded by a constant if the algorithm distinguishes between f_r and the zero function with probability $\frac{3}{4}$. This completes the proof that the number of queries k much be order $\sqrt{2^n}$.