Quantum Information Processing
Quantum Algorithms (III)

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

November 10, 2021

Abstract

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

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

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

© 2021 by Richard Cleve. All rights reserved.

Contents

1 The order-finding problem

1.1 Greatest common divisors and Euclid's algorithm

In section 2, 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 1.1 (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 1.2 (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

$$\mathbb{Z}_m^* = \big\{x \in \mathbb{Z}_m : x \text{ has a multiplicative inverse mod } m\big\}, \tag{1}$$

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

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

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

$$1, 5, 4, 20, 16, 17, 1, 5, 4, 20, 16, 17, 1, 5, 4, 20, 16, \ldots \tag{3}$$

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 1.3 (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.

1.2 The order-finding problem and its relation to factoring

Definition 1.4 (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[1] 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

$$a^r - 1 = (a^{r/2} + 1)(a^{r/2} - 1). \tag{4}$$

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

[1] 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.

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 1.5 (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 $\mathrm{ord}_m(a)$ 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

$$\gcd(a^{r/2}+1, m) \in \{p, q\}. \tag{5}$$

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 1.1. *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.

2 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 3, I will show you an efficient algorithm for factoring (using reduction to the order-finding algorithm from section 1.2).

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 $r = \mathrm{ord}_m(a)$.

2.1 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 $b \in \mathbb{Z}_m^*$,

$$U_{a,m} |b\rangle = |ab\rangle \tag{6}$$

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

$$\begin{aligned} |\psi_1\rangle &= \tfrac{1}{\sqrt{r}}\Big(|1\rangle + \omega^{-1}|a\rangle + \omega^{-2}|a^2\rangle + \cdots + \omega^{-(r-1)}|a^{r-1}\rangle\Big) && (7)\\ &= \tfrac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega^{-j}\left|a^j\right\rangle, && (8)\end{aligned}$$

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,

$$\begin{aligned} U_{a,m}|\psi_1\rangle &= \tfrac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega^{-j}\left|a^{j+1}\right\rangle && (9)\\ &= \tfrac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega\,\omega^{-(j+1)}\left|a^{j+1}\right\rangle && (10)\\ &= \omega\,|\psi_1\rangle. && (11)\end{aligned}$$

Therefore $|\psi_1\rangle$ is an eigenvalue 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 $|\psi_1\rangle$.

2.1.1 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.

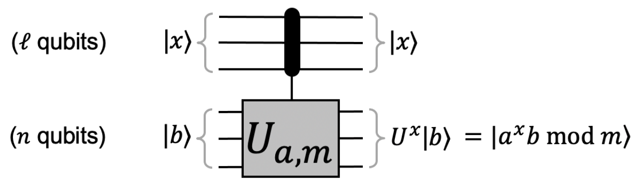


Figure 1: Multiplicity-controlled-$U_{a,m}$ gate.

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

$$|x\rangle\,(U_{m,a})^x\,|b\rangle = |x\rangle\,|a^x b\rangle \tag{12}$$

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

We can compute this efficiently[2] (with respect to ℓ and n) by using the repeated squaring trick to exponentiate a. The number of gates is $O(\ell n \log n)$.

2.1.2 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.

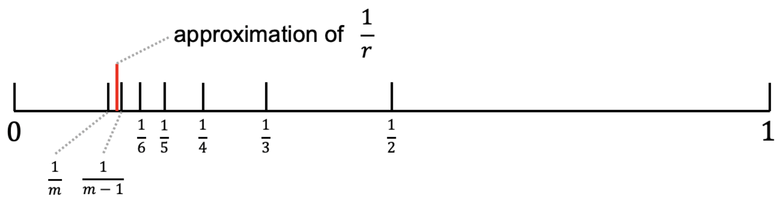


Figure 2: The positions of $\left\{\frac{1}{1},\frac{1}{2},\frac{1}{3},\ldots,\frac{1}{m}\right\}$ on the real line.

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

$$\frac{1}{m-1} - \frac{1}{m} = \frac{m-(m-1)}{m(m-1)} = \frac{1}{m(m-1)} \approx \frac{1}{m^2}. \tag{13}$$

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 $\ell = 2n+1$.

[2]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.

2.1.3 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 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.

2.2 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

$$\begin{aligned}
|\psi_2\rangle &= \frac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega^{-2j}\left|a^j\right\rangle \qquad (14)\\
\vdots \quad & \quad \vdots \\
|\psi_k\rangle &= \frac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega^{-kj}\left|a^j\right\rangle \qquad (15)\\
\vdots \quad & \quad \vdots \\
|\psi_r\rangle &= \frac{1}{\sqrt{r}}\sum_{j=0}^{r-1}\omega^{-rj}\left|a^j\right\rangle. \qquad (16)
\end{aligned}$$

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.

2.2.1 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 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.

2.2.2 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 $0.b_1 b_2 \ldots b_\ell$ 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 $0.b_1 b_2 \ldots b_\ell$ of one of the elements of

$$\left\{ \frac{k}{r} \;\middle|\; \text{where } r \in \{1, 2, \ldots, m\} \text{ and } k \in \{1, 2, \ldots, r\} \right\}. \tag{17}$$

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 3 shows all the *candidates* for $\frac{k}{r}$.

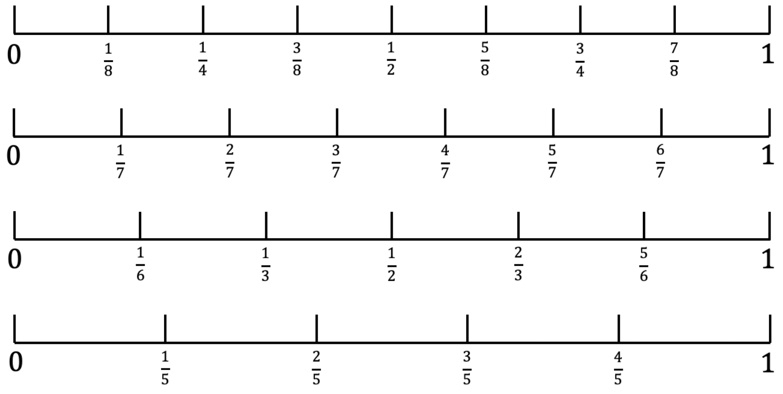


Figure 3: The set of candidates for $\frac{k}{r}$ in the $m = 8$ case.

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

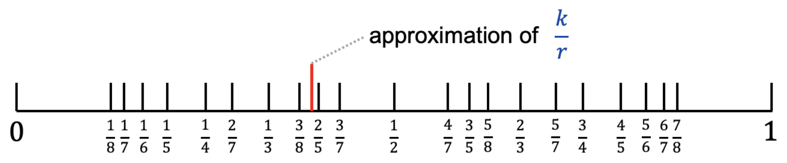

approximation of $\frac{k}{r}$
0
$\frac{1}{8}$ $\frac{1}{7}$ $\frac{1}{6}$ $\frac{1}{5}$ $\frac{1}{4}$ $\frac{2}{7}$ $\frac{1}{3}$ $\frac{3}{8}$ $\frac{2}{5}$ $\frac{3}{7}$ $\frac{1}{2}$ $\frac{4}{7}$ $\frac{3}{5}$ $\frac{5}{8}$ $\frac{2}{3}$ $\frac{5}{7}$ $\frac{3}{4}$ $\frac{4}{5}$ $\frac{5}{6}$ $\frac{6}{7}$ $\frac{7}{8}$
1

Figure 4: 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*.[3] 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*.

2.2.3 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

[3]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).

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.

In fact, we can do better than that. If we make two repetitions then there are two rational numbers $\frac{k_1}{r}$ and $\frac{k_1}{r}$ (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 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.

2.2.4 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.

2.3 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

$$\tfrac{1}{\sqrt{r}}\sum_{k=1}^{r}|\psi_k\rangle \tag{18}$$

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 2.2. So we have an efficient algorithm for order-finding using the superposition state in Eq. (18) in place of an eigenvector (it succeeds with constant probability, independent of n).

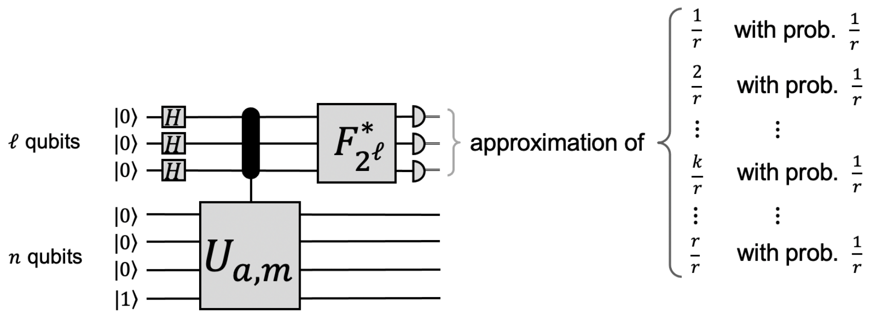
2.4 Order-finding without the requirement of an eigenvector

How can we efficiently generate the superposition state in Eq. (18)? In fact, this is extremely easy to do. To see how, expand the state as

$$\begin{aligned}
\tfrac{1}{\sqrt{r}}\sum_{k=1}^{r}|\psi_k\rangle \;=\;& \tfrac{1}{r}\Big(|1\rangle + \omega\,|a\rangle + \omega^2\,|a^2\rangle + \cdots + \omega^{r-1}|a^{r-1}\rangle\Big)\\
&+ \tfrac{1}{r}\Big(|1\rangle + \omega^2\,|a\rangle + \omega^4\,|a^2\rangle + \cdots + \omega^{2(r-1)}|a^{r-1}\rangle\Big)\\
&+ \tfrac{1}{r}\Big(|1\rangle + \omega^3\,|a\rangle + \omega^6\,|a^2\rangle + \cdots + \omega^{3(r-1)}|a^{r-1}\rangle\Big)\\
&\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\qquad\qquad \vdots\\
&+ \tfrac{1}{r}\Big(|1\rangle + \omega^r\,|a\rangle + \omega^{2r}\,|a^2\rangle + \cdots + \omega^{r(r-1)}|a^{r-1}\rangle\Big) && (19)\\
=\;& |1\rangle && (20)
\end{aligned}$$

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. (18) 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.



ℓ qubits
n qubits
$|0\rangle$
$|1\rangle$
H
$F^*_{2^\ell}$
$U_{a,m}$
approximation of
$\frac{1}{r}$ with prob. $\frac{1}{r}$
$\frac{2}{r}$ with prob. $\frac{1}{r}$
$\frac{k}{r}$ with prob. $\frac{1}{r}$
$\frac{r}{r}$ with prob. $\frac{1}{r}$

Figure 5: 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 $O(n^2 \log n)$.

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

In section 2.1.1, 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

$$|x, b\rangle \mapsto |x, a^x b\rangle, \tag{21}$$

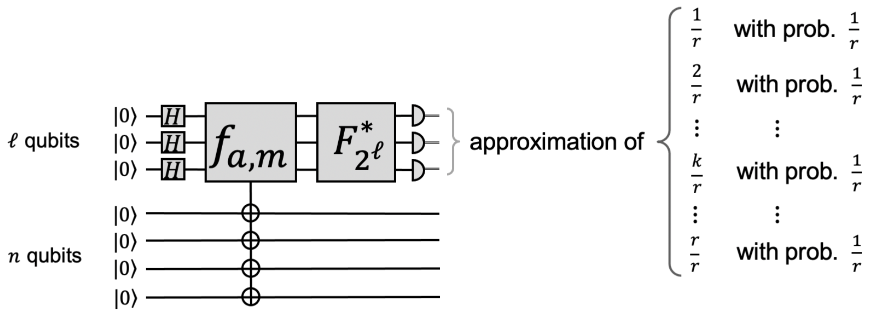
for all $x \in \{0,1\}^\ell$ and $b \in \mathbb{Z}_m^*$. But our framework for computing classical functions in superposition (in Part II: 2.6.2 & 2.6.3) is different from this. What we showed there is that we can compute an f-query

$$|x, c\rangle \mapsto |x\rangle\,|f(x) \oplus c\rangle \tag{22}$$

in terms of a classical implementation of f.

There are two remedies. One is a separate construction for efficiently computing the mapping in Eq. (21). 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. (22) *instead of* the mapping in Eq. (21).

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 6 instead of the circuit in figure 5.



ℓ qubits
n qubits
$|0\rangle$
H
$f_{a,m}$
$F^*_{2^\ell}$
approximation of
$\frac{1}{r}$ with prob. $\frac{1}{r}$
$\frac{2}{r}$ with prob. $\frac{1}{r}$
$\frac{k}{r}$ with prob. $\frac{1}{r}$
$\frac{r}{r}$ with prob. $\frac{1}{r}$

Figure 6: 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 5) and $f_{a,m}$-query (figure 6) is

$$\frac{1}{\sqrt{2^\ell}} \sum_{x\in\{0,1\}^\ell} |x\rangle\,|a^x\rangle\,. \tag{23}$$

3 Shor's factoring algorithm

Figure 7 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 2). The algorithm is based on the reduction of the factoring problem to the order-finding problem which is explained in section 1.2.

```
1:  choose a random $a \in \{1, 2, \ldots, m-1\}$
2:  $g \leftarrow \gcd(a, m)$
3:  if $g = 1$ then   (case where $a \in \mathbb{Z}_m^*$)
4:      $r \leftarrow \mathrm{ord}_m(a)$   (the quantum part of the algorithm)
5:      if $r$ is even then
6:          $f \leftarrow \gcd(a^{r/2} + 1, m)$
7:          if $f$ properly divides $m$ then
8:              output $f$
9:          end if
10:     end if
11: else   (case where $g > 1$, so $g$ is a proper divisor of $m$)
12:     output $g$
13: end if
```

Figure 7: 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 1.2) 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 7.

4 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.

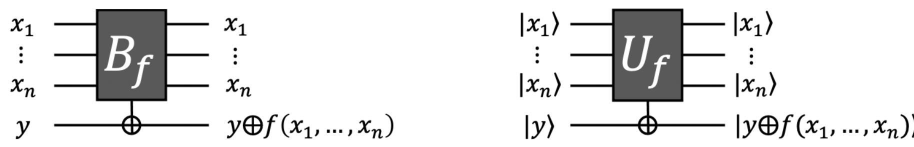


Figure 8: Classical reversible f-query (left) and quantum f-query (right).

In the notation of figure 8, $B_f : \{0,1\}^{n+1} \to \{0,1\}^{n+1}$ 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 $f : \{0,1\}^n \rightarrow \{0,1\}$ can be computed in polynomial-time, so there is a quantum circuit of size $O(n^c)$ that implements an f-query.

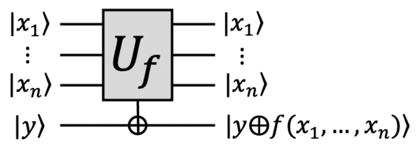


Figure 9: Implementation of a quantum f-query by a quantum circuit.

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.

4.1 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 10.

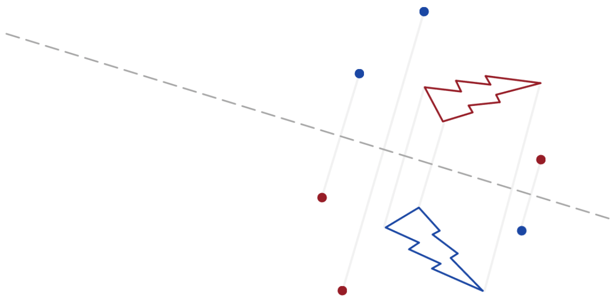


Figure 10: The reflection about the line of each red item is shown in blue.

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

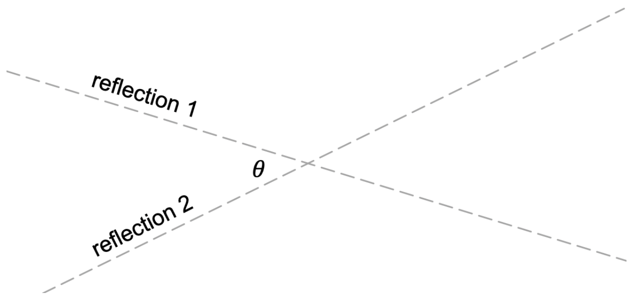


Figure 11: Two refection lines with angle θ between them.

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.

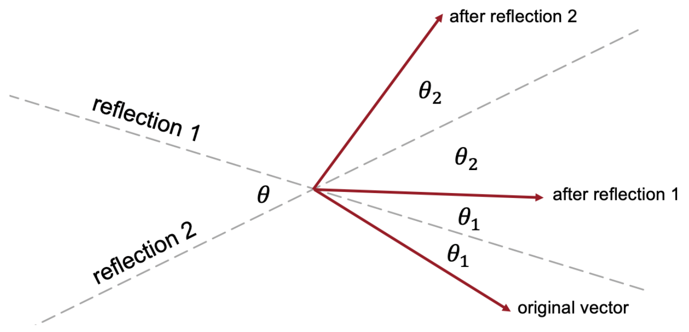


Figure 12: Effect of reflection 1 followed by reflection 2 on a vector.

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

The picture above 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 4.1. *Prove Lemma 4.1.*

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

4.2 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 $b \in \{0,1\}$,

$$U_0 |a\rangle |b\rangle = |a\rangle |b \oplus [a = 0^n]\rangle , \tag{24}$$

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

$$[a = 0^n] = \begin{cases} 1 & \text{if } a = 0^n \\ 0 & \text{if } a \neq 0^n. \end{cases} \tag{25}$$

In other words, in the computational basis, U_0 flips the target qubit if and only if the first n qubits are all $|0\rangle$. Our notation for the U_0 gate is shown in Figure 13 (which also shows one implementation in terms of an n-ary Toffoli gate).

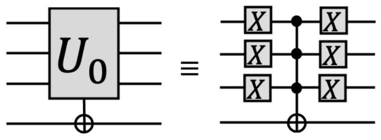


Figure 13: Notation for U_0 gate (implementable by Pauli X gates and an n-fold Toffoli gate).

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 $\otimes n$).

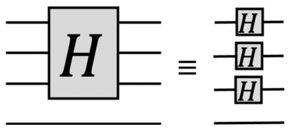


Figure 14: Notation for $H^{\otimes n}$ gate.

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 $x \in \{0,1\}^n$,

$$U_f |x\rangle |-\rangle = (-1)^{f(x)} |x\rangle |-\rangle \tag{26}$$

$$U_0 |x\rangle |-\rangle = (-1)^{[x=00\ldots0]} |x\rangle |-\rangle . \tag{27}$$

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

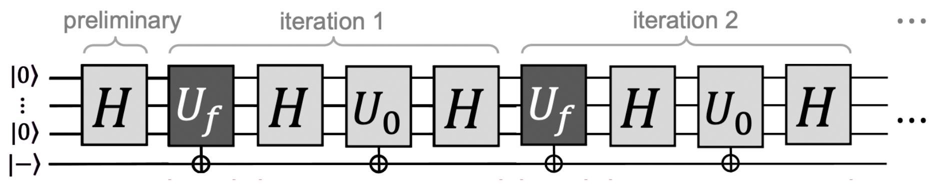


Figure 15: Quantum circuit for Grover's algorithm.

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 $H|00\dots0\rangle|-\rangle$
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 16: 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 4.1 about two reflections being a rotation). The function f partitions $\{0,1\}^n$ into two subsets, A_0 and A_1, defined as

$$A_1 = \{x \in \{0,1\}^n : f(x) = 1\} \tag{28}$$

$$A_0 = \{x \in \{0,1\}^n : f(x) = 0\}. \tag{29}$$

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

$$|A_1\rangle = \tfrac{1}{\sqrt{s_1}} \sum_{x \in A_1} |x\rangle \tag{30}$$

$$|A_0\rangle = \tfrac{1}{\sqrt{s_0}} \sum_{x \in A_0} |x\rangle \,. \tag{31}$$

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

$$\tfrac{1}{\sqrt{N}} \sum_{x \in \{0,1\}^n} |x\rangle = \sqrt{\tfrac{s_0}{N}} \Big(\tfrac{1}{\sqrt{s_0}} \sum_{x \in A_0} |x\rangle \Big) + \sqrt{\tfrac{s_1}{N}} \Big(\tfrac{1}{\sqrt{s_1}} \sum_{x \in A_1} |x\rangle \Big) \tag{32}$$

$$= \sqrt{\tfrac{s_0}{N}}\, |A_0\rangle + \sqrt{\tfrac{s_1}{N}}\, |A_1\rangle \,. \tag{33}$$

Let θ be the angle between $|A_0\rangle$ and $H|0...0\rangle$, as illustrated in figure 17.

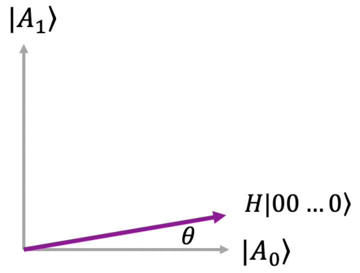


Figure 17: The state $H|0...0\rangle$ within the subspace spanned by $|A_0\rangle$ and $|A_1\rangle$.

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 4.2. *$-HU_0H$ 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

$$\begin{aligned}(-HU_0H)H|0...0\rangle &= -HU_0|0...0\rangle && (34)\\ &= -H(-|0...0\rangle) && (35)\\ &= H|0...0\rangle\,. && (36)\end{aligned}$$

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 18.

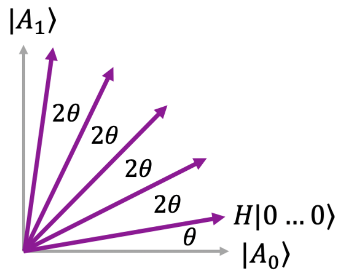


Figure 18: The state $|0...0\rangle$ rotates by angle 2θ after each iteration.

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.

4.3 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).

4.4 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 19 shows the success probability as a function of the number iterations k, for any $k \in \left\{1, 2, \ldots, \lceil \frac{\pi}{4}\sqrt{N}\,\rceil\right\}$.

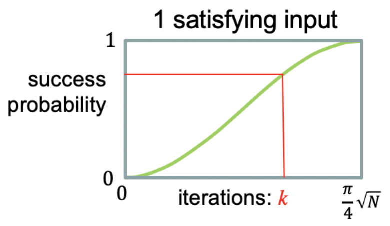


Figure 19: Success probability of measurement as a function of k, the number of iterations.

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 20 (they are sinusoidal curves corresponding to more total rotation).

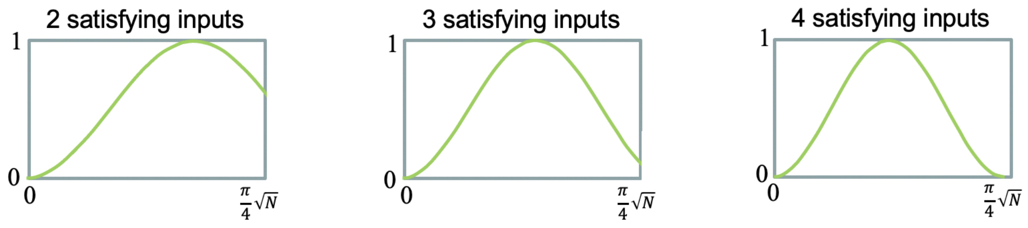


Figure 20: Success probability function in cases of 2, 3, and 4 satisfying inputs.

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.

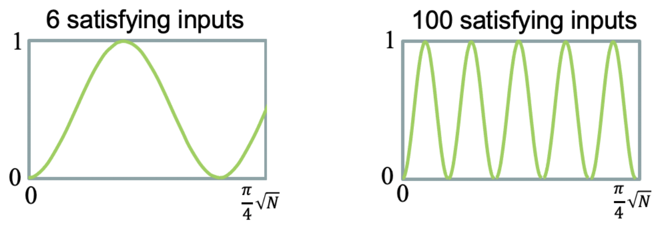


Figure 21: Success probability function in cases of 6 and 100 satisfying inputs.

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? Actually, no. It can be proven that the above query costs are the best possible, as explained in the next section.

5 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 5.1 (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 $\Omega\big(\sqrt{2^n}\,\big)$ 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).

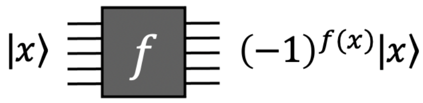


Figure 22: Query of f in the phase.

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.

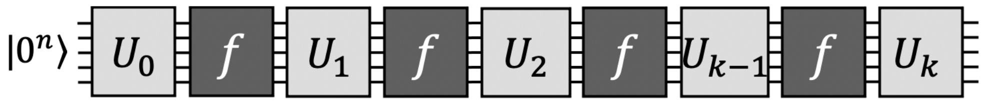


Figure 23: Form of a k query quantum circuit.

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 $U_0, U_1, \dots, U_k$. 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

$$f_r(x) = \begin{cases} 1 & \text{if } x = r \\ 0 & \text{otherwise.} \end{cases} \tag{37}$$

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 $U_0, U_1, \dots, U_k$. Suppose that we insert one of the functions f_r into the query gate.

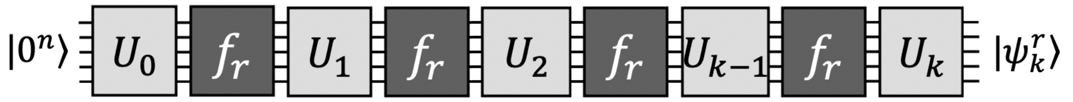


Figure 24: Quantum circuit making k queries to function f_r.

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.

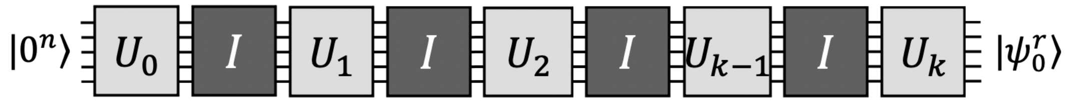


Figure 25: Quantum circuit with I substituted for the k queries.

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 24 corresponds to the function f_r. The circuit in figure 25 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

$$\big\| |\psi_k^r\rangle - |\psi_0^r\rangle \big\| \leq \frac{2k}{\sqrt{2^n}}. \tag{38}$$

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 24 and 25, along with some intermediate circuits.

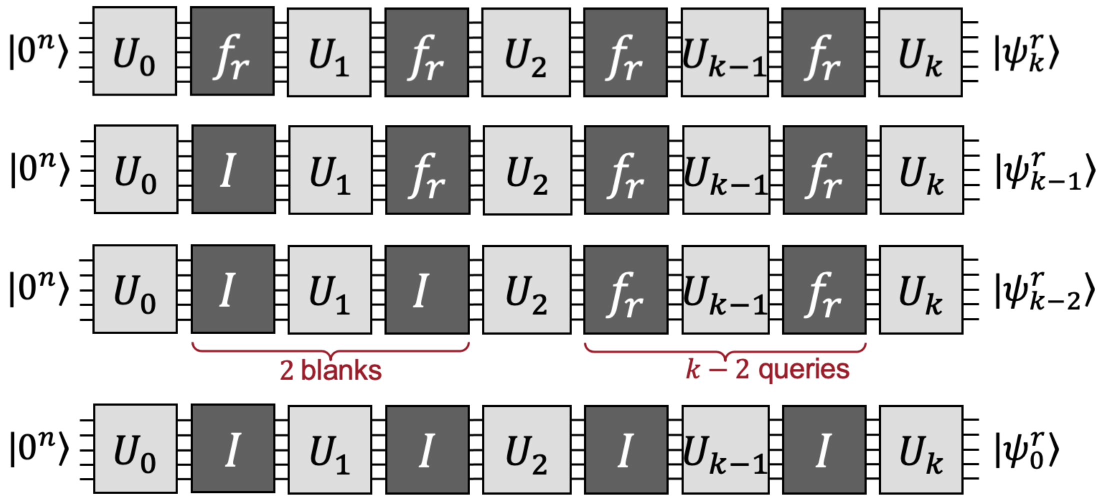


Figure 26: Quantum circuits with i blanks, followed by $k-i$ queries to f_r (for $i \in \{0, 1, \ldots, r\}$).

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

$$|\psi_k^r\rangle - |\psi_0^r\rangle = \big(|\psi_k^r\rangle - |\psi_{k-1}^r\rangle\big) + \big(|\psi_{k-1}^r\rangle - |\psi_{k-2}^r\rangle\big) + \cdots + \big(|\psi_1^r\rangle - |\psi_0^r\rangle\big) \tag{39}$$

which implies that

$$\big\||\psi_k^r\rangle - |\psi_0^r\rangle\big\| \leq \big\||\psi_k^r\rangle - |\psi_{k-1}^r\rangle\big\| + \big\||\psi_{k-1}^r\rangle - |\psi_{k-2}^r\rangle\big\| + \cdots + \big\||\psi_1^r\rangle - |\psi_0^r\rangle\big\|. \tag{40}$$

Next, we'll upper bound each term $\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \|$ in Eq. 40. 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.

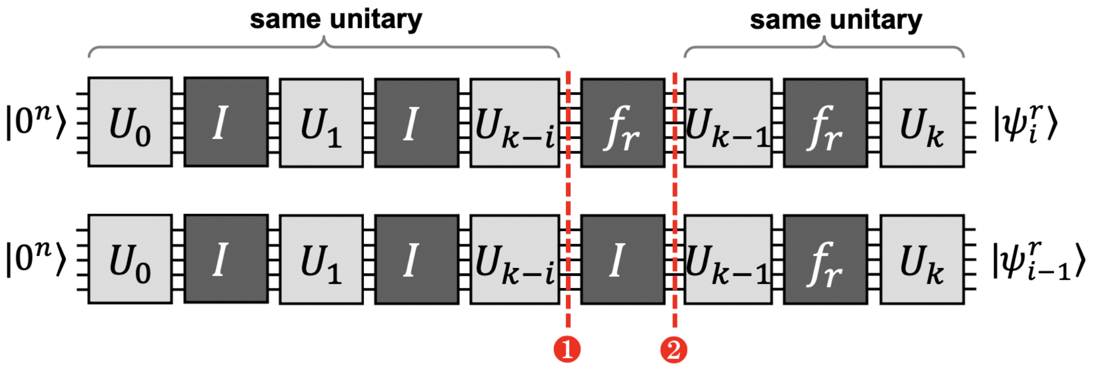


Figure 27: Circuit with $k-i$ blanks and i queries (top) and $k-i+1$ blanks and $i-1$ queries (bottom).

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

$$\sum_{x\in\{0,1\}^n} \alpha_{i,x} |x\rangle\,, \tag{41}$$

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

$$\sum_{x\in\{0,1\}^n} |\alpha_{i,x}|^2 = 1. \tag{42}$$

These states make sense for each $i \in \{1, \dots, k\}$, 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

$$\sum_{x\in\{0,1\}^n} (-1)^{[x=r]}\alpha_{i,x} |x\rangle\,. \tag{43}$$

The difference between the state in Eq. (41) and the state in Eq. (43) is only in the amplitude of $|r\rangle$, which is negated in Eq. (43). 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 27 do. The remaining steps for each circuit are the same unitary operation. Since unitary operations preserve Euclidean distances, this means that

$$\big\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \big\| = 2|\alpha_{i,r}|. \tag{44}$$

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

$$\begin{aligned}
\frac{1}{2^n} \sum_{r\in\{0,1\}^n} \big\| |\psi_k^r\rangle - |\psi_0^r\rangle \big\| &\leq \frac{1}{2^n} \sum_{r\in\{0,1\}^n} \left(\sum_{i=1}^{k} \big\| |\psi_i^r\rangle - |\psi_{i-1}^r\rangle \big\| \right) && \text{telescoping sum} && (45)\\
&= \frac{1}{2^n} \sum_{r\in\{0,1\}^n} \left(\sum_{i=1}^{k} 2|\alpha_{i,r}| \right) && \text{by Eq. (44)} && (46)\\
&= \frac{1}{2^n} \sum_{i=1}^{k} 2 \left(\sum_{r\in\{0,1\}^n} |\alpha_{i,r}| \right) && \text{reordering sums} && (47)\\
&\leq \frac{1}{2^n} \sum_{i=1}^{k} 2 \left(\sqrt{2^n} \right) && \text{Cauchy-Schwarz} && (48)\\
&= \frac{2k}{\sqrt{2^n}}. && && (49)
\end{aligned}$$

In Eq. (48) 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 $\sqrt{2^n}$.