41076: Methods in Quantum Computing

'Quantum Algorithms' Module

Maria Kieferova based on materials from Min-Hsiu Hsieh
Centre for Quantum Software & Information, Faculty of Engineering and Information Technology, University of Technology Sydney

Abstract

The content to be covered in these three lectures are

1. Complexity of an algorithm
2. Phase kickback
3. Hadamard transform
4. Fourier Transform Algorithm
5. Phase Estimation
6. Hamiltonian simulation and ground state preparation
7. Grover search
8. Complexity theory
9. Quantum complexity

1 What makes a good algorithm

To understand how to analyze and design quantum algorithms, we first need to talk about general concepts in algorithm design. An algorithm can be thought of as a recipe for a computer to compute a function, i.e. compute an output for each input from a set of allowed inputs. We might require that the algorithm always produces the correct output. We say that an algorithm is **sound** if it never returns a wrong solution, say an incorrectly sorted list, a wrong solution to arithmetic operations or a sub-optimal solution. An algorithm is **complete** if it answers any input. An incomplete algorithm for cat/dog classification might not provide a label if the input is neither a cat nor a dog picture.

There are notable types of algorithms that are not complete or sound:

- Many **probabilistic**/quantum algorithms give the correct answer with a high probability (say, in at least 2/3 cases) but might fail. The probability of success can be boosted by repeated runs. Examples included adiabatic phases estimation, Grover search, adiabatic algorithms (with long evolution time), random walk search ...
- **Approximation algorithms** are useful for optimization problems. Optimization problems require finding the maximum/minimum or a function on a given domain. An approximation

algorithm does not aim to achieve the best solution but produces a solution that is “close”. An example for is Goemans-Williamson algorithm that is guaranteed to achieve approximation ratio (defined as $\frac{\text{algorithm's score}}{\text{optimal score}}$) of 0.868 for MAX-CUT. Some algorithms come with rigorous guarantees for the quality of the approximation, others, such as simulated annealing, are guaranteed to produce the optimal solution if run indefinitely.

- Some algorithms perform well only on a subset of inputs. An example would be machine algorithms, for example, ML approaches to k-means clustering, that would be inefficient on general problems but perform well on problems with some structure. A formal way to restrict the set of inputs is to define a **promise problem** which guarantees additional properties of the input (as in the Deutsch-Jozsa algorithm).

Heuristic algorithms are a broad class of algorithms that trade speed for completeness, optimality, or precision that might be based on rigorous underlying theory or rule-of-thumb insights.

To analyze an algorithm, we consider the resources an algorithm needs. We call these considerations **computational complexity** The most common consideration is the time the algorithm needs to compute the output. The time is proportional to the number of operations the computer needs to make during computation. We consider how the time needed grows with the **size of the input** . This can be the number of bits of a number, the size of an array or a data set, or the number of particles in a quantum system. When talking about the dependence of the time on the size of the input, we call this measure the **time complexity** or often only **complexity**.

We sometimes consider are space needed for computation known as space complexity. Optimizing algorithms for space is needed when one is dealing with limited memory and a limited number of qubits. If we have a computational model that can execute multiple operations in parallel, such as a node with many processors/GPU-s, we might consider the depth of an algorithm instead. In practice, one might consider the dollar cost of computation which would depend on the computational time as well as the particularities of a machine used.

Let’s say that we agreed what is the metric that matters to us. How would we then evaluate the performance of our algorithms? There are several approaches

- In theoretical computer science, the most common approach is computing the **asymptotic complexity**. This is an upper bound on the amount of time one would need as a function of input size in the limit of input size going to infinity. Computing the tight bound is generally complicated but a good estimate in the big-O notation can be obtained for many algorithms.

- When an algorithm is too complicated to analyze rigorously or its performance can vary, it is common to estimate its **performance on benchmarks**. An example would be standard machine learning or SAT competition. It is important to choose benchmarks similar to the problems of practical importance and similar size. Choosing a benchmark that is too small or simple can give a false sense of success that would not generalize to larger instances.

- Argue with everyone at conferences and on Twitter that your algorithm is the best. This is common but not recommended.

Once we know the complexity of our algorithm, we can ask how good it is. The strongest type of evidence would be an argument that it is not possible to construct an asymptotically better algorithm, possibly subject to some complexity-theoretic assumptions. An example of such algorithms is $O(n \log n)$ complexity sorts and Grover search (here n refers to the number of items in an array). For these examples, the performance of the algorithms matches the theoretical lower bound on what could be possible. An argument that is still strong is showing that your algorithm is the fastest existing algorithm. Such an example is Shor's asymptotically faster factoring algorithm (polynomial vs. exponential). Since factoring has been studied for centuries and Shor's algorithm is the first efficient algorithm this constitutes a major achievement. On the other hand, the QAOA algorithm has achieved the best approximation ratio for the Max-3XOR problem, only to be outperformed a few weeks later. Of course, you might not care about having the best algorithm according to the community. You might only care about solving a particular problem, say computing properties of a complex molecule or devising a super-secret strategy for trading derivatives and you only care that your algorithm delivers a solution up to your (and your boss's standard).

1.1 Big-O notation

Perhaps the most important measure of an algorithm is how its complexity grows as the size of the input. Unfortunately, estimating the exact resources is almost always impossible - the exact number of gates will depend on the gate set physical properties of a chip. To be able to devise and compare algorithms that can run on many different chips, we focus only on the type of growth rate expressed using the big-O notation that characterizes the behavior of complexity for the limit $x \to \infty$.

Let $f(x) : \mathcal{R} \to \mathcal{R}$ be the exact complexity of an algorithm. One can write

$$f(x) = O(g(x)) \tag{1}$$

if there exist numbers $x_0 \in \mathcal{R}$ and $M \in \mathcal{R}_+$ such that

$$|f(x)| \leq Mg(x) \tag{2}$$

for all $x > x_0$. When estimating the big-O complexity, it is useful to keep in mind the following properties:

$$f_1(x) = O(g_1(x)) \text{ and } f_2(x) = O(g_2(x)) \implies f_1(x)f_2(x) = O(g_1(x)g_2(x)) \tag{3}$$

$$f_1(x) = O(g_1(x)) \implies f_1(x)f_2(x) = O(g_1(x)f_2(x)) \tag{4}$$

$$f_1(x) = O(g_1(x)) \text{ and } f_2(x) = O(g_2(x)) \implies f_1(x) + f_2(x) = O(\max(g_1(x), g_2(x))) \tag{5}$$

$$f_1(x) = O(g_1(x)) \implies \mathit{const.} \cdot f_1(x) = O(g_1(x)) \tag{6}$$

$$f_1(x) = O(g_1(x)) \implies \mathit{const.} + f_1(x) = O(g_1(x)) \tag{7}$$

Exercise 1. *Practise the big-O formalism.*

- *Is* $2^{n+1} = O(2^n)$*?*
- *Is* $2^{2n} = O(2^n)$*?*
- *Compute the asymptotic upper bound on* $2^{n+1} + 2^{2n}$.

We say that an algorithm is efficient if its complexity is polynomial in input size as opposed to exponential in any parameter. The class of all problems solvable in polynomial time on a classical computer is known as **P** and **BQP** on a quantum computer.

notation	name	example
$O(1)$	constant	computing $(-1)^x$ $x \in \mathbb{N}$
$O(\log x)$	logarithmic	binary search
$O(x)$	linear	classical unstructured search
$O(x^2)$	quadratic	bubble sort, elementary-school multiplication, Dijkstra
$O(x^c)$	polynomial	all of the above, linear programming, primality testing, shortest path on a graph
$O(2^x)$	exponential	travelling salesman, generalized checkers
$O(x!)$	factorial	brute force try all possibilities algorithm

Figure 1: Common functions characterizing asymptotic complexity.

1.2 Oracles

Apart from gates, it is possible to include oracles (sometimes also called black boxes) in computation. An oracle is an abstract operation that can perform a given computation in a unit of time. We refer to the number of queries to the oracle as the query complexity. There are several reasons why one wants to include oracles in an algorithm. An oracle can represent a separate computational primitive whose implementation can vary. An example of such oracles are functions that compute matrix elements of a Hamiltonian in Hamiltonian simulation.

An oracle can be given as a part of the definition of the problem. In this case, one can have oracular access to a function, and our goal is to determine some properties of this function as in Deutsch-Joza or Grover's algorithms. Finally, oracles are often used in the theory of computational complexity to quantify the difficulty of tasks. One can construct a hierarchy of computational difficulty with respect to more and more powerful oracles.

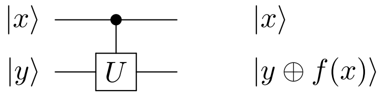


Figure 2: Reversible oracle for a function f.

2 Thinking about quantum algorithms

We often (but not always) think about quantum algorithms in terms of quantum circuits. Broadly speaking, a quantum algorithm is a description of how to build a quantum circuit that produces the desired output from the input for each size of the input. Thinking in terms of quantum circuits is often tedious for more advanced algorithms and people often describe quantum algorithms in terms of frameworks such as quantum walks or simulating sparse matrices. These concepts will be explored with more depth in UTS class Quantum algorithms.

If we are given a quantum circuit, we can easily compute its various cost metrics such as the number of operations and depth. Calculating an asymptotic complexity of a quantum algorithm is similar to calculating the complexity of classical algorithms.

We learned in lecture one that any classical computation can be realized reversibly using Toffoli gates. Since Toffoli gates can be (in principle) implemented on quantum computers, it means that any classical computation can be implemented on a quantum computer as a quantum algorithm. However, doing so would not produce any speedup - it would be likely slower (by a constant factor)

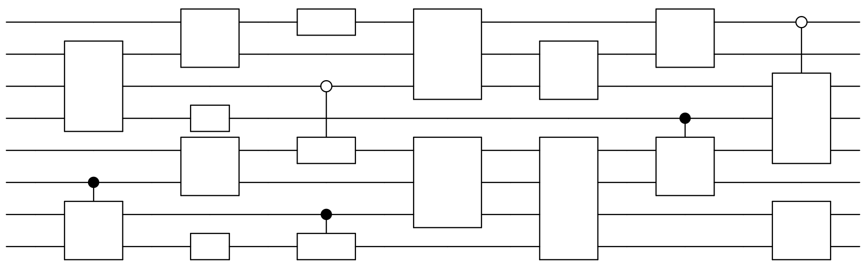


Figure 3: A quantum circuit consists of wires (qubits, represented as horizontal lines) and gates (unitary operations, represented as boxes). The space cost of this circuit is 8, the time cost is 17, and the depth is 7.

a more expensive. However, this highlights the point that an **existence of a quantum algorithm does not imply a speedup**, which refutes the common misconception that quantum computing can exponentially speed up any computation. We have examples of algorithms, for example sorting, that provably do not allow asymptotic quantum speedup.

3 Phase kickback

Exercise 2. *Compute:*

- *CNOT* $|b\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ *for* $b \in \{0,1\}$. *Hint: It will be useful to write* $1 = (-1)^0$ *and* $-1 = (-1)^1$.
- *CNOT* $\frac{|0\rangle+|1\rangle}{\sqrt{2}}\frac{|0\rangle-|1\rangle}{\sqrt{2}}$.

We can now see that if the target qubit is an eigenstate of X, CNOT will affect the control qubit instead of the target one. A phase kickback is a clever trick that extends this observation from a CNOT to an arbitrarily controlled gate. Let us take the oracle from 2 and apply it on $|y\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$. We will get:

$$U_f|x\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = \frac{U_f|x\rangle|0\rangle - U_f|x\rangle|1\rangle}{\sqrt{2}} \tag{8}$$

$$= \frac{|x\rangle|0\oplus f(x)\rangle - |x\rangle|1\oplus f(x)\rangle}{\sqrt{2}}. \tag{9}$$

Without knowing the value of x, we still know that $f(x)$ will be either 0 or 1.

Let us consider what happens for the options. For $f(x) = 0$ we get

$$\frac{|x\rangle|0\oplus 0\rangle - |x\rangle|1\oplus 0\rangle}{\sqrt{2}} = \frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{10}$$

and similarly $f(x) = 1$

$$\frac{|x\rangle|0\oplus 1\rangle - |x\rangle|1\oplus 1\rangle}{\sqrt{2}} = -\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{11}$$

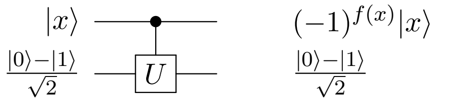

$|x\rangle$
$\frac{|0\rangle-|1\rangle}{\sqrt{2}}$
U
$(-1)^{f(x)}|x\rangle$
$\frac{|0\rangle-|1\rangle}{\sqrt{2}}$

Figure 4: Phase kickback

will result in a global phase. We can both outcomes in a compact way

$$(-1)^{f(x)}|x\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{12}$$

.

This is known as the phase-kickback trick - it allows us to turn the output of the function f into a (global) phase. To see why it can be useful, consider $|x\rangle = a|0\rangle + b|1\rangle$ to be an arbitrary superposition. Using (12) we can compute

$$U(a|0\rangle+b|1\rangle)\frac{|0\rangle-|1\rangle}{\sqrt{2}} = (-1)^{f(0)}a|0\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} + (-1)^{f(1)}b|1\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{13}$$

$$= \Big((-1)^{f(0)}a|0\rangle + (-1)^{f(1)}b|1\rangle\Big)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right). \tag{14}$$

The action of the oracle is now entirely encoded in the phase of the first qubit and there is no entanglement between registers.

An application of the phase kickback is Deutsch's algorithm.

Exercise 3 (Deutsch's problem)**.** *Consider an unknown function $f(x)$ given through a coherent oracle. Decide whether the function is constant $f(0) = f(1)$ or balanced $f(0) \neq f(1)$.*

- *Use the phase kickback trick for $x = 0$ and $x = 1$ on Deutsch's problem. What is the outcome for a constant and a balance function?*
- *Now consider $|x\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$. What gate do you need to apply on the control register to read out the solution in a computational basis?*

4 Hadamard transform

We saw that the application of Hadamard was crucial in the previous algorithm. Applying Hadamard on every qubit is known as **Hadamard transform**. In the simplest case, we can apply Hadamards on a number of qubits in each in state $|0\rangle$.

$$(H|0\rangle)^{\otimes n} = \left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)^{\otimes n} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle. \tag{15}$$

Exercise 4. *Expand $\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)^{\otimes n}$ to convince yourself that*

$$\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)^{\otimes n} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle. \tag{16}$$

How would you formally prove it?

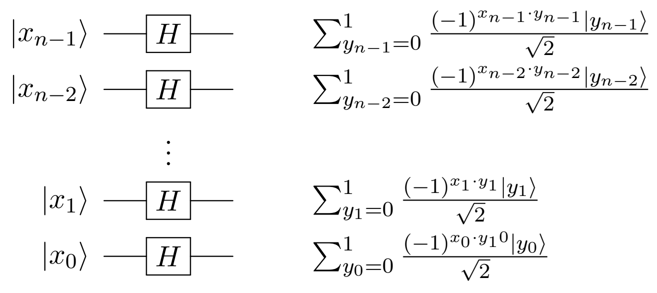
We see that applying Hadamards will create a uniform superposition over all strings. What happens when we apply Hadamard on an arbitrary input? First, notice that we can write an action of a single Hadamard as

$$H|x_i\rangle = \frac{|0\rangle + (-1)^x|1\rangle}{\sqrt{2}} = \sum_{y=0}^{1} \frac{(-1)^{x_i \cdot y}|y\rangle}{\sqrt{2}} \tag{17}$$

where $x_i \in \{0,1\}$. Let us now take $x = x_0 x_1 \dots x_{n-1}$ and apply Hadamard on this state. We obtain

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_y (-1)^{x \cdot y}|y\rangle, \tag{18}$$

where $x \cdot y = \sum_i x_i y_i$.



$|x_{n-1}\rangle$
H
$\sum_{y_{n-1}=0}^{1} \frac{(-1)^{x_{n-1} \cdot y_{n-1}}|y_{n-1}\rangle}{\sqrt{2}}$
$|x_{n-2}\rangle$
H
$\sum_{y_{n-2}=0}^{1} \frac{(-1)^{x_{n-2} \cdot y_{n-2}}|y_{n-2}\rangle}{\sqrt{2}}$
$\vdots$
$|x_1\rangle$
H
$\sum_{y_1=0}^{1} \frac{(-1)^{x_1 \cdot y_1}|y_1\rangle}{\sqrt{2}}$
$|x_0\rangle$
H
$\sum_{y_0=0}^{1} \frac{(-1)^{x_0 \cdot y_1 0}|y_0\rangle}{\sqrt{2}}$

Figure 5: Hadamard transform

Exercise 5. *Show that Hadamard transform is self-inverse, i.e. $H^{\otimes n} H^{\otimes n} = \mathbb{I}^{\otimes n}$. While this might be obvious for each qubit, it is worth expanding the expression.*

An application of phase kickback and Hadamard transform is the Deutsch-Jozsa algorithm.

Problem 6 (Deutsch-Jozsa). *Assume an oracle that implements a function $f : \{0,1\}^n \to \{0,1\}$ such that we are promised*

1. *(constant) all inputs give the same output, or*
2. *(balanced) half the inputs give '0' and the other half give '1'.*

The goal is to find out whether $f(x)$ is constant or balanced.

Before going to the quantum algorithm for solving this problem, let us first look at the possible classical solutions. In fact, the classical strategy is very simple. If we want 100% accuracy, in the worse case, one has to *query* at least $\frac{N}{2} + 1$ bits in $\boldsymbol{x}$.

It seemed quite counter-intuitive in the beginning that there is a quantum algorithm, proposed by Deutsch and Jozsa, which can produce the correct answer with just a single use of quantum oracle (i.e., quantum unitary). The quantum oracle queries the bit string $\boldsymbol{x}$ only once; hence the Deutsch-Jozsa algorithm has the exponential saving, compared with the classical strategy, in the number of queries to $\boldsymbol{x}$.

A quantum algorithm for the Deutsch-Jozsa problem expands on Deutsch's algorithm by replacing a single Hadamard with the Hadamard transform.

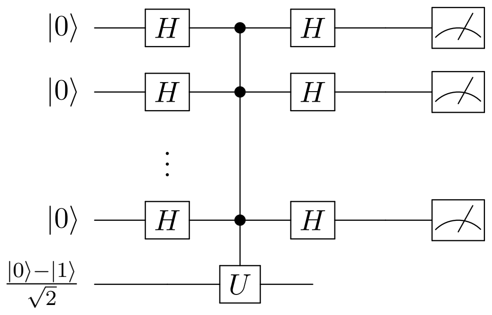


Figure 6: Deutsch-Jozsa algorithm

We will again use a quantum oracle

$$O_{\boldsymbol{x}} : |i\rangle|b\rangle \rightarrow |i\rangle \otimes |b \oplus x_i\rangle \tag{19}$$

where $i \in [N]$, $b \in \mathbb{Z}_2$ and $\oplus$ is the binary addition.

Let us see what the algorithm in 6 does step by step.

Starting with zeros on the first n registers and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ on the last qubit, we apply Hadamards on the first n qubits and end up with a uniform superposition over all strings

$$H^{\otimes n}|0\rangle^{\otimes n}\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{20}$$

Next, we apply the oracle

$$U2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\oplus f(i)\rangle-|1\oplus f(i)\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}(-1)^{f(i)}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{21}$$

where we realized the insight from the previous section that the oracle will perform a phase shift. After the second Hadamard transform, we get

$$2^{-n/2}H^{\otimes n}\sum_{i=0}^{2^n-1}(-1)^{f(i)}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n}\sum_{j=0}^{2^n-1}\sum_{i=0}^{2^n-1}(-1)^{f(i)}(-1)^{i\cdot j}|j\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{22}$$

The algorithm is concluded by measuring the first n registers. Let us now see what is the resulting state if f is constant or balanced. If f is constant, $f(i) = f$ independent of i. That would simply (22) to

$$2^{-n}(-1)^f\sum_{j=0}^{2^n-1}\sum_{i=0}^{2^n-1}(-1)^{i\cdot j}|j\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = (-1)^f|0\rangle^{\otimes n}\frac{|0\rangle-|1\rangle}{\sqrt{2}}, \tag{23}$$

because the oracle will only contribute an overall phase and Hadamard transform is self inverse. Thus, if the function is constant, we will measure all zeros. If the function is balanced, we can divide the inputs into those with output 0 and the remaining half with output 1

$$2^{-n}\left(\sum_{i,f(i)=0}(-1)^0\sum_{j=0}^{2^n-1}(-1)^{i\cdot j}|j\rangle + \sum_{i,f(i)=1}\sum_{j=0}^{2^n-1}(-1)^1(-1)^{i\cdot j}|j\rangle\right)\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{24}$$

We can see that this state has 0 overlap with $|0\rangle^{\otimes n}$. Thus, if we measure all zeros, the function is constant and otherwise, it is balanced.

Exercise 7. *Show that the output from a balanced function will be orthogonal to all the zero state.*

Before ending this section, I would like to emphasize that if we allow some small error probability in deciding whether $\boldsymbol{x}$ is constant or balanced in the classical setting, the quantum advantage of the Deutsch-Jozsa algorithm will disappear completely.

5 Quantum Fourier Transform

The discrete Fourier transform(DFT) of a set $\{x_0, \cdots, x_{N-1}\}$ of N elements is defined as

$$X_k := \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{i2\pi}{N}jk} x_j. \tag{25}$$

and its inverse

$$x_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{-\frac{i2\pi}{N}jk} X_k \tag{26}$$

Discrete Fourier transform maps discrete samples from a finite interval into the frequency domain. Sometimes, the definition of DFT and inverse DFT are reversed.

The definition of Fourier transform in Eq. (25) can be extended to the quantum setting

$$|\Psi_k\rangle := QFT|k\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{i2\pi}{N}jk} |j\rangle. \tag{27}$$

Exercise 8. *Show that QFT is unitary.*

It is crucial to note that the state $|\Psi_k\rangle$ is a product state when $N = 2^n$ and can be written as

$$\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{i2\pi jk/2^n} |j\rangle = \bigotimes_{\ell=1}^{n} \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi k/2^\ell} |1\rangle \Big),$$

.

To show that, consider the binary representation of $j \equiv (j_1, \cdots, j_n) \in [N]$, where j_1 is the most significant bit, i.e.,

$$j = j_1 2^{n-1} + j_2 2^{n-2} + \cdots + j_n 2^0, \tag{28}$$

and we write

$$j/2^n = 0.j_1 j_2 \cdots j_n = \sum_{\ell=1}^{n} j_\ell 2^{-\ell}.$$

Similarly for k,

$$k = k_1 2^{n-1} + k_2 2^{n-2} + \cdots + k_n 2^0, \tag{29}$$

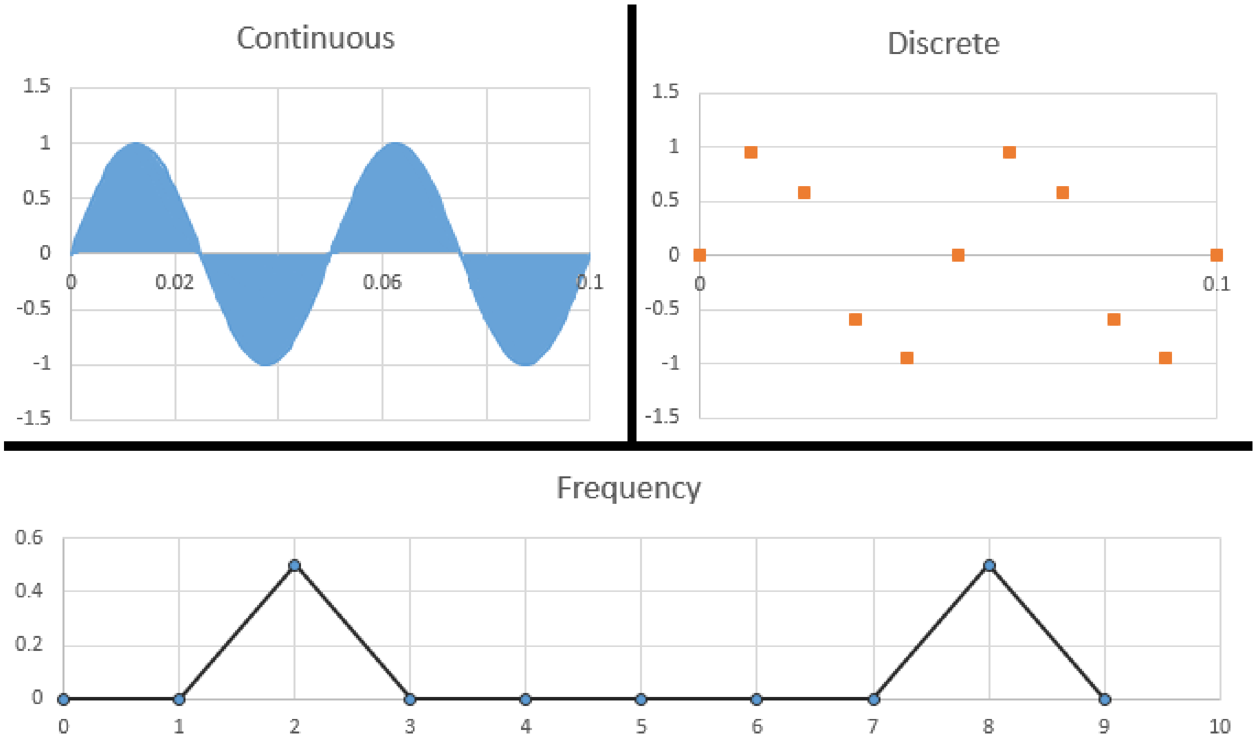


Figure 7: Discrete Fourier transform can be seen as a discretized version of Fourier transform. Applying DFT reveals dominant frequencies in the signal. Source: https://blog.endaq.com/fourier-transform-basics

Take for example, $j = 5 = (1, 0, 1)$ and $n = 3$, therefore $5/8 = 0.101$. Thus

$$\begin{aligned}
|\Psi_k\rangle &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{i2\pi jk/2^n} |j\rangle \\
&= \frac{1}{\sqrt{2^n}} \sum_{j_1=0}^{1} \cdots \sum_{j_n=0}^{1} e^{i2\pi (\sum_{\ell=1}^{n} j_\ell/2^\ell)k} |j_1, j_2, \cdots, j_n\rangle \\
&= \frac{1}{\sqrt{2^n}} \sum_{j_1=0}^{1} \cdots \sum_{j_n=0}^{1} \bigotimes_{\ell=1}^{n} e^{i2\pi j_\ell k/2^\ell} |j_\ell\rangle \\
&= \frac{1}{\sqrt{2^n}} \bigotimes_{\ell=1}^{n} \left[\sum_{j_\ell=0}^{1} e^{i2\pi j_\ell k/2^{-\ell}} |j_\ell\rangle \right] \\
&= \bigotimes_{\ell=1}^{n} \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi k/2^\ell} |1\rangle \Big), \qquad \text{(this is a product state)} \\
&:= \frac{1}{\sqrt{2^n}} \Big(|0\rangle + e^{i2\pi 0.k_1 k_2 \ldots k_n} |1\rangle \Big) \Big(|0\rangle + e^{i2\pi 0.k_2 k_3 \ldots k_n} |1\rangle \Big) \ldots \Big(|0\rangle + e^{i2\pi 0.k_n} |1\rangle \Big) \qquad (30)
\end{aligned}$$

Any digits before comma (the binary/decimal separator) in the expansion of k have no effect on the value ($e^{i2\pi m} = 1$ for $m \in \mathbb{N}$).

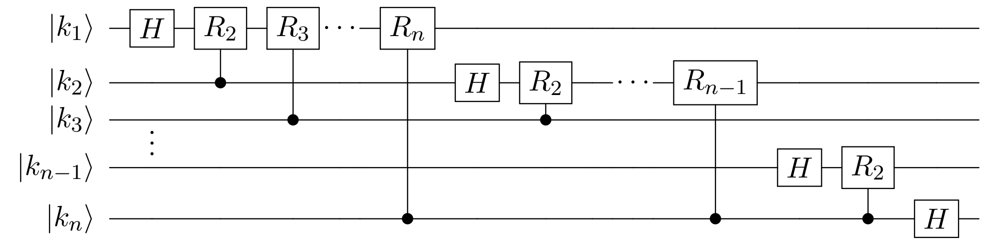


Figure 8: Circuit for quantum Fourier transform.

The implementation of Eq. (30) is given in Figure 8, where

$$R_m = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^m} \end{pmatrix}. \tag{31}$$

Let us now apply this circuit on $|k\rangle = |k_1\rangle \dots |k_n\rangle$ and see that it indeed implements QFT. The first step is Hadamard on $|k_1\rangle$. If $|k_1\rangle = |0\rangle$ on the first qubit, we will get $\frac{|0\rangle+|1\rangle}{2}$ and if $|k_1\rangle = |1\rangle$, we should get $\frac{|0\rangle-|1\rangle}{2}$. Concisely, we can write

$$|k_1\rangle \otimes \cdots \otimes |k_n\rangle \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1}|1\rangle\Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle, \tag{32}$$

because

$$e^{i2\pi 0.k_1} = e^{i2\pi \frac{k_1}{2}} = \begin{cases} -1 & \text{when } k_1 = 1 \\ 1 & \text{when } k_1 = 0 \end{cases}. \tag{33}$$

Next, we implement the first rotation. If $k_2 = 1$, we apply

$$\frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1}|1\rangle\Big)1 \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1 e^{i2\pi/2^2}}|1\rangle\Big)1 \tag{34}$$

and if k_2, the first qubit is unchanged. We can write this as

$$\frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1 k_2}|1\rangle\Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle. \tag{35}$$

verify for yourself that $j_2 = 0$, the state in Eq. (35) is the same as Eq. (32), and when $k_2 = 1$, a phase of $e^{i2\pi/2^2}$ is applied. Following the same derivation, the state on the first qubit will become

$$\frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1 k_2 \cdots k_n}|1\rangle\Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle. \tag{36}$$

On the second qubit, we get after Hadamard

$$|k_2\rangle \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_2}|1\rangle\Big)$$

and controlled rotations

$$|k_2\rangle \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_2 k_3 \dots k_n}|1\rangle\Big)$$

On all the qubits

$$2^{-n/2}\Big(|0\rangle + e^{2\pi i 0.k_n}|1\rangle\Big)\Big(|0\rangle + e^{2\pi i 0.k_{n-1}k_n}|1\rangle\Big)\dots\Big(|0\rangle + e^{2\pi i 0.k_1k_2\dots k_n}|1\rangle\Big)$$

We can see that the number of gates in Figure 8 is

$$n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}, \tag{37}$$

that is exponentially less than the classical fast Fourier transform which requires $O(n2^n)$ gates. However, these two algorithms are not directly comparable because Fast fourier transform produces a classical state and QFT a quantum one.

Exercise 9. *Show that on all zero input, the Hadamard transform and quantum Fourier transform produced the same output. Formally* $QFT|0\rangle^{\otimes n} = H^{\otimes n}|0\rangle^{\otimes n}$.

Unlike Hadamard transform, quantum Fourier transform is not self inverse. QFT^{-1} is defined as

$$|\Psi_k\rangle := U_{\mathrm{F}}^{-1}|k\rangle = \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} e^{-\frac{i2\pi}{N}jk}|j\rangle. \tag{38}$$

Inverse-QFT can be implemented by inverting the circuit, i.e. running it backwards with rotations in the negative direction. Thus, inverse-QFT has the same complexity as QFT.

Applying inverse QFT after QFT is an interesting exercise. We know that

$$QFT^{-1}QFT|k\rangle = |k\rangle \tag{39}$$

but it will allow us to showcast a useful identity.

$$QFT^{-1}QFT|k\rangle = QFT^{-1}\frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} e^{\frac{i2\pi}{N}jk}|j\rangle \tag{40}$$

$$= \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} e^{\frac{i2\pi}{N}jk}\frac{1}{\sqrt{N}}\sum_{l=0}^{N-1} e^{\frac{-i2\pi}{N}jl}|l\rangle \tag{41}$$

$$= \frac{1}{N}\sum_{l,j=0}^{N-1} e^{\frac{i2\pi}{N}j(k-l)}|l\rangle. \tag{42}$$

We will split the sum into two sums: one when $k = l$ and one when $k \neq l$

$$\frac{1}{N}\sum_{j=0}^{N-1} e^{\frac{i2\pi}{N}j(k-l)}|l\rangle = \frac{1}{N}\sum_{l,j=0}^{N-1} e^{\frac{i2\pi}{N}j(k-l)}|l\rangle\delta_{j,k} + \frac{1}{N^2}\sum_{j,l=0,l\neq k}^{N-1} e^{\frac{i2\pi}{N}j(k-l)}|l\rangle. \tag{43}$$

Here we used Kronecker delta δ_j, k which is 1 when $k = l$ and 0 otherwise. When $k = l$ we get

$$\frac{1}{N}\sum_{j,l=0}^{N-1} e^{\frac{i2\pi}{N}j(k-l)}|l\rangle\delta_{l,k} = \frac{1}{N}\sum_{j=0}^{N-1} 1|k\rangle = |k\rangle. \tag{44}$$

If $k \neq l$, we can denote $m = l - k$. The sum will cancel out because of

$$\sum_{j=0}^{N-1} e^{\frac{-mi2\pi}{N} j} = 0 \tag{45}$$

The numbers we are adding are uniformly distributed on the unit circle in a complex plane. From symmetry, their sum must be equal to 0 - I recommend writing sketching down a simple case to convince yourself about it.

We can also prove (45) by showing that the sum won't change if we multiply it by $e^{e^{\frac{-mi2\pi t}{N}}}$ for any integer t.

Thus, we showed that indeed $QFT^{-1}QFT|k\rangle = |k\rangle$.

6 Phase Estimation

In this section, we will introduce the quantum phase estimation protocol. The crucial component of the quantum phase estimation protocol is the quantum Fourier transform.

Problem 10. *Given a unitary U and its eigenvector $|\nu\rangle$, estimate the corresponding eigenvalue $\lambda = e^{i2\pi\varphi}$.*

The quantum phase estimation algorithm, illustrated in Figure 9, can estimate the value of φ to the additive error ε with high probability, using $O(\log(\frac{1}{\varepsilon}))$ qubits and $O(\frac{1}{\varepsilon})$ controlled-U operations.

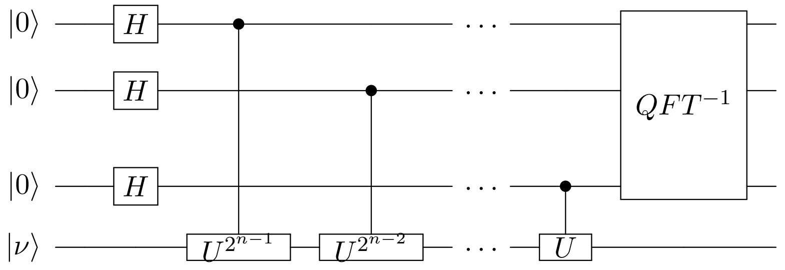


Figure 9: A circuit for phases estimation and eigenvalue estimation. The last block represents the inverse quantum Fourier transform.

For simplicity, we will assume that there exists an integer a, such that $a = \varphi N$ where $N = 2^n$. If φ cannot be exactly expressed in our binary register, the analysis is a little bit more complicated and the outcome becomes probabilistic.

After Hadamard transform step, the overall state is

$$|\Psi_{t_1}\rangle = \left(\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} |k\rangle \right) \otimes |\nu\rangle. \tag{46}$$

Since $U|\nu\rangle = e^{i2\pi\varphi}|\nu\rangle$, we have

$$U^{2^j}|\nu\rangle = e^{i2\pi 2^j \varphi}|\nu\rangle. \tag{47}$$

Each controlled unitary will act as

$$c - U^{2^j}|k_i\rangle|\nu\rangle = e^{i2\pi 2^j k_i \varphi}|\nu\rangle, \tag{48}$$

where we again think of k in its binary representation $k = k_1 2^{n-1} + k_2 2^{n-2} + \cdots + k_n 2^0$. Note that all of the controlled-U operations commute. After application of the controlled unitaries, the state will be

$$|\Psi_{t_2}\rangle = \left(\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{i2\pi ak} |k\rangle \right) \otimes |\nu\rangle \tag{49}$$

Lastly we apply the inverse quantum Fourier transform here denoted as U_F^{-1}

$$|\Psi_{t_3}\rangle = (U_{\mathrm{F}}^{-1} \otimes I)|\Psi_{t_2}\rangle = \left(\frac{1}{N} \sum_{y=0}^{N-1} \sum_{k=0}^{N-1} e^{\frac{i2\pi k}{N}(a-y)} |y\rangle \right) \otimes |\nu\rangle \tag{50}$$

$$= \left(\frac{1}{N} \sum_{k=0}^{N-1} e^{\frac{i2\pi k}{N} 0} |a\rangle \right) \otimes |\nu\rangle \tag{51}$$

$$= |a\rangle \otimes |\nu\rangle. \tag{52}$$

In the second to last step we used the identity

$$\sum_{k=0}^{N-1} e^{\frac{i2\pi k}{N}(a-y)} |y\rangle = \begin{cases} N & a = y \\ 0 & \text{otherwise} \quad (a, y \in N). \end{cases} \tag{53}$$

If we consider a case where $N\varphi$ cannot be exactly expressed in n binary places, the analysis would go in the same way up to (50) but we won't get perfect cancellation in the next step. Instead, we will go nonzero probabilities around states that are close to φN.

7 Order finding and Shor's algorithm

Order finding is an application of QFT and phase estimation that is the crucial step of Shor's algorithm. We say that r is an order of a modulo N if it is the smallest positive integer that satisfies

$$a^r = 1 (\text{mod } N). \tag{54}$$

where N is an integer.

Exercise 11. *Find the order of* 4 *modulo* 7*, i.e. find* r *such that* $4^r = 1$ *(mod* 7*).*

Generally, computing the order is difficult for large N - we do not have a classical algorithm that can perform order finding in time polynomial in the number of bits of N. In fact, factoring can be reduced into order finding. Let N be a large integer that we wish to find factors of. Assume that N is not even or of the form $n^k = N$ – these cases are easy to check and would allow us to find factors easily.

1. Randomly pick $1 < a < N$.
2. Use Euclidean algorithms to find the greatest common divisor $gcd(a, N)$. If it is larger than 1, we found a factor of N and stop.
3. Compute the period r such that $a^r = 1(\text{mod } N)$.

4. If r is odd or $a^{r/2} = N-1 (\text{mod } N)$, go back restart from step 1.
5. Otherwise, both $gcd(a^{r/2} \pm 1, N)$ give nontrivial factors of N.

The above scheme uses the Euclidean algorithm to compute the greatest common divider of two number. With the exception of step 3, all the other states can be executed efficiently on a classical computer with access to randomness.

Exercise 12. *Try this algorithm with $N = 15$ and $a = 7$.*

Shor's algorithm performs the steps above with order finding being the only quantum part of the algorithm. Order finding uses the circuit in Fig. 9 with oracle that perform modular exponentiation

$$U^s|x\rangle = |a^s x \ (mod \ N)\rangle \tag{55}$$

and we choose $|x\rangle = |1\rangle$. To ensure that U is indeed a unitary, we a and N must be coprime which we ensured in step 2.

Exercise 13. *Prove that if a operator U satisfies $U^r = I$, then the eigenvalues of U must be r-roots of 1.*

It can be shown that the eigenstates of U have the form

$$|u_k\rangle = \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} e^{-2\pi i \frac{k}{r} s} |a^s modN\rangle \tag{56}$$

with corresponding eigenvalues $e^{i2\pi \frac{k}{r}}$. If we were given $|u_k\rangle$, we would be able to learn r using phase estimation. Unfortunately, we don't know how to prepare $|u_k\rangle$ without knowing r. However, it turns out we can prepare a uniform superposition over k-s

$$\frac{1}{r} \sum_{k,s=0}^{r-1} e^{-2\pi i \frac{k}{r} s} |a^s modN\rangle. \tag{57}$$

To understand why this state is easy to prepare, let us consider $s = 0$

$$\frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i \frac{k}{r} 0} |a^0 modN\rangle = |1\rangle. \tag{58}$$

When $s \neq 0$, the terms in the sum will cancel out

$$\frac{1}{r} \sum_{s=1}^{r-1} \sum_{k,s}^{r-1} e^{-2\pi i \frac{k}{r} s} |a^s modN\rangle = \frac{1}{r} \sum_{s=1}^{r-1} 0 |a^s modN\rangle \tag{59}$$

where we used the identity (45). Thus, the state $|x\rangle = |1\rangle$ can be used in place of the eigenstate in phase estimation.

We will use controlled versions of the modular exponentiation oracle

$$|z\rangle c - U^j |x\rangle = |z\rangle |a^{z \cdot j} x(\text{ mod } N)\rangle. \tag{60}$$

We can apply U^j operators by repeated squaring.

This oracle can be implemented using modular exponentiation and each application has complexity $O(\log N^2)$. Since we need to apply it N times, the overall complexity of applying controlled-U operations is $O(\log^3 N)$. The other steps of the algorithm are the Hadamard transform with complexity $O(\log N)$ (we need $\log N$ qubit to encode N) and quantum Fourier transform with complexity $O(\log^2 N$. Thus, the overall complexity of the order-finding algorithm, as well as the quantum part of Shor's algorithm, is $O(\log^3 N)$.

Exercise 14. *A naive construction of U^j in for(60) would require j applications of U. Since $j \leq N$, this would lead to complexity $O(N)$. Why and how can we achieve complexity poly-logarithmic in N?*

Shor's algorithm is arguably the most impactful quantum algorithm we have to date. This is because a factoring underlies a common scheme for encrypting sensitive data (such as web traffic or credit card information) known as RSA (Rivest–Shamir–Adleman). The public key C for RSA is a large integer on n bits such that $C = pq$ where p, q are primes that define the private keys. One can decrypt the information if and only if they have the private keys which are kept secret. However, anyone can encrypt information using publicly available C. For classical computers, factoring C into p and q is hard, however, a quantum computer with 20 million physical qubits can break RSA in about 8 hours (given additional assumptions) [2]. A modification of Shor's algorithm can be used for computing discrete logarithms that can break other encryption schemes including elliptic curve cryptography. An area of cryptography and cybersecurity research examining (classical) encryption schemes that are not vulnerable to quantum attracts is known as *post-quantum cryptography.*

8 Hamiltonian simulation

In this section we use H to denote Hamiltonians and Had for Hadamard!

Given an initial state of a system $|\psi(0)\rangle$ and a Hamiltonian H, our goal is to simulate the time evolution $|\psi(0)\rangle \to e^{-iHt}|\psi(0)\rangle$. The goal of the Hamiltonian simulation is to design a circuit U consisting of gates and oracles that approximates the time evolution up to an error ϵ such that

$$\left\|U - e^{-iHt}\right\|_2 < \epsilon, \tag{61}$$

where $\|\cdot\|_2$ is the spectral norm.

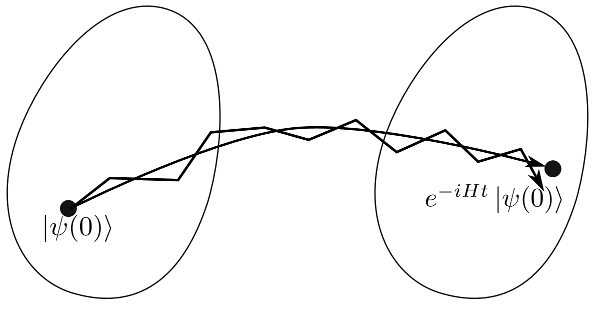


Figure 10: Hamiltonian simulation approximates the time evolution by a series of digital operations.

Quantum systems are fundamentally difficult to simulate; the dynamics of quantum systems is a BQP-hard (or BQP complete for Hamiltonians with natural restrictions). Except for a few special cases, the complexity of the best known classical algorithms grows exponentially with the number of qubits. As such, simulation of quantum dynamics is a field where quantum computers can quickly outperform classical ones. In fact, the time evolution of quantum systems was the original application for quantum computers suggested by Feynman.

In the simplest scenario, we assume that the simulated Hamiltonians are given in the form $H = \sum_{j=1}^{m} H_j$, where each H_j is sufficiently simple such that $e^{-iH_j t}$ can be implemented directly for arbitrary t. A common case is when these H_j are Paulis or local Hamiltonians.

The simplest quantum simulation algorithms rely on the Lie-Trotter formula

$$\begin{aligned}\lim_{r\to\infty}\left(e^{A/r}e^{B/r}\right)^r &= \lim_{r\to\infty}\left((1+\frac{A}{r})(1+\frac{B}{r})\right)^r && (62)\\ &= \lim_{r\to\infty}\left(1+\frac{A+B}{r}+\frac{AB}{r^2}\right)^r && (63)\\ &= \lim_{r\to\infty}\left(1+\frac{A+B}{r}\right)^r && (64)\\ &= e^{A+B} && (65)\end{aligned}$$

Since the individual terms in the Hamiltonian typically do not commute, decomposing an exponential of a sum into a finite sum of exponentials will lead to errors.

Exercise: Prove that $\left\|e^{t(A+B)} - \left(e^{At/r}e^{Bt/r}\right)^r\right\| \in \mathcal{O}\left(\frac{t^2}{r}\right)$ for $\|A\|, \|B\| \leq 1$.

Next, we can recursively that for $H = \sum_{j=1}^{m} H_j$, one can decompose the evolution with respect to H into the evolution with respect to each H_j as

$$\widetilde{U} = \left(e^{-iH_1 t/r}e^{-iH_2 t/r}\dots e^{-iH_m t/r}\right)^r + \mathcal{O}(\|H\|t^2/r). \tag{66}$$

Thus, if we are willing to tolerate error at most ϵ, we need to perform $\mathcal{O}(\frac{\|H\|t^2}{\epsilon})$ operations.

Up to now, we assumed that we know how to implement each $e^{-iH_j t}$ directly. Let us now show how to implement them in some simple cases.

In the simplest case, H_j is a Pauli Z acting on the jth qubit. The Hamiltonian evolution is then a Z-rotation on the jth qubit.

$$e^{-itZ} = e^{-it}|0\rangle\langle 0| + e^{it}|1\rangle\langle 1| \tag{67}$$

Next, we show how to simulate a tensor product of Zs

$$H = Z_1 \otimes Z_2 \otimes \dots \otimes Z_n. \tag{68}$$

We first prove the following identity. For an arbitrary unitary U:

$$\begin{aligned}Ue^{-iHt}U^\dagger &= U\sum_{k=0}^{\infty}\frac{(-iHt)^k}{k!}U^\dagger && (69)\\ &= UU^\dagger - iUHU^\dagger t + i^2 UH(U^\dagger U)Ht^2U^\dagger \\ &\quad - i^3 UH(U^\dagger U)H(U^\dagger U)HU^\dagger t^2 + \dots && (70)\\ &= \sum_{k=0}^{\infty}\frac{(-iUHU^\dagger t)^k}{k!} && (71)\\ &= e^{-iUHU^\dagger t}. && (72)\end{aligned}$$

We can implement $e^{-iZ_1\otimes\cdots\otimes Z_n t}$ using the circuit in Fig. 11

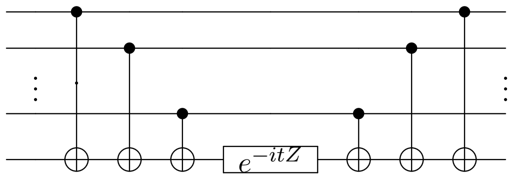


Figure 11: Simulating a tensor product of Pauli Zs.

The correctness of the circuit can be shown through induction. We already proved the first step in (67). In the inductive step, we conjugate an existing circuit with CNOTs, see Fig. 11. The top wire corresponds to the very last register in our notation. We can rewrite $CNOT e^{-itZ_1\otimes Z_2\otimes\cdots\otimes Z_{n-1}} CNOT$ as:

$$\begin{aligned}
&\big(|0\rangle\langle 0|\otimes \mathbb{I} + |1\rangle\langle 1|\otimes X\big)\big(\mathbb{I}\otimes e^{-itZ_1\otimes\cdots\otimes Z_{n-1}}\big)\big(|0\rangle\langle 0|\otimes \mathbb{I} + |1\rangle\langle 1|\otimes X\big)\\
=&|0\rangle\langle 0|\otimes e^{-itZ_1\otimes\cdots\otimes Z_{n-1}} + |1\rangle\langle 1| X e^{-itZ_1\otimes\cdots\otimes Z_{n-1}} X\otimes\\
=&e^{-itZ_1\otimes\cdots\otimes Z_{n-1}}\otimes|0\rangle\langle 0| + e^{itZ_1\otimes\cdots\otimes Z_{n-1}}\otimes|1\rangle\langle 1|\\
=&e^{-itZ_1\otimes Z_2\otimes\cdots\otimes Z_{n-1}\otimes Z_n}
\end{aligned}$$

Simulating other Paulis is possible by changing the basis. Recall that $e^{-itUHU^\dagger} = Ue^{-itH}U^\dagger$. We can then use the identities

$$X = \text{Had } Z \text{ Had} \tag{73}$$

$$Y = S^\dagger \text{Had } Z \text{ Had } S \tag{74}$$

We used Had instead of H to denote the Hadamard gate since we reserved the symbol H for Hamiltonians.

This allows us to simulate evolution according to any Pauli. For example, we can simulate the evolution according to the $H = X\otimes Y\otimes Z$ according to the circuit in Fig. 12.

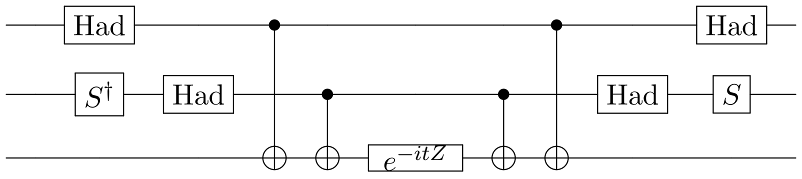


Figure 12: X and Y Paulis can be simulated by a change of basis. The circuit above depicts the simulation according to $e^{-itX\otimes Y\otimes Z}$.

Now we know how to simulate any Hamiltonian that is a sum of Paulis. The complexity of the algorithm for simulating a Hamiltonian $H=\sum_{l=0}^{L-1} P_l$ where P_ns are Paulis on at most n qubits is

$$\mathcal{O}\left(\frac{Lt^2n}{\epsilon}\right). \tag{75}$$

However, there is still a need for more efficient algorithms. First, the number of terms L needed to decompose a Hamiltonian into a sum of Paulis can be exponentially large. Second, the scaling in terms of t and ϵ is quite poor.

Other Hamiltonian simulation algorithms allow for different decomposition of Hamiltonians. A popular one is decomposing a sparse matrix into 1-sparse matrices which can be then simulated directly. Another one is decomposition into a linear combination of unitaries (LCU) which is a generalization of the Pauli decompositions.

Exercise: Let ρ, σ be density matrices, S the SWAP operator and Tr_p partial trace over the first variable. Show that

$$\mathrm{Tr}_p[e^{-iS\Delta}\rho \otimes \sigma e^{iS\Delta}] = e^{-i\rho\Delta}\sigma e^{i\rho\Delta} + \mathcal{O}(\Delta^2)$$

In these more general cases, the Hamiltonian is accessed through an oracle. One type of oracular access is particularly common when the Hamiltonian (in a computational basis) is given by a sparse matrix. We say a Hamiltonian is *row-d-sparse* if each row has at most d non-zero entries. If there is an efficient procedure to locate these entries we moreover say that the Hamiltonian is *row-computable*. In this case, one can efficiently construct oracles

$$O_{loc}|r, k\rangle = |r, k \oplus l\rangle \tag{76}$$

$$O_{val}|r, l, z\rangle = |r, l, z \oplus H_{r,l}\rangle. \tag{77}$$

Oracle O_{loc} locates the position l of the k-th non-zero element in row r. The oracle O_{val} then gives the value of the matrix element $H_{r,l}$. We compute the cost of algorithms in terms of the number of queries to these oracles.

It is possible to construct different oracles. Any Hermitian matrix can be decomposed into a sum of unitaries

$$H = \sum_{l=0}^{L-1} \alpha_l H_l, \tag{78}$$

where for each l, $\alpha_l \geq 0$ and H_l is a unitary matrix $\|H_l\| = 1$. This decomposition can be efficiently implemented for sparse Hamiltonians. The coefficients α_l and unitaries H_l can be accessed through oracles

$$O_\alpha|l, z\rangle = |l, z \oplus \alpha_l\rangle \tag{79}$$

$$O_{V_l}|l, \psi\rangle = V_l|l, \psi\rangle, \tag{80}$$

or, in some cases, described classically.

9 Grover's Algorithm

Problem 15 (The unstructured search problem)**.** *For $N = 2^n$, we are given a marked item $\boldsymbol{w} \in \mathbb{Z}_2^N$, and the goal is to locate $\boldsymbol{w}$.*

Exercise 16. *How would you solve the unstructured search classically? What is the best complexity that can be achieved?*

In the quantum setting, we are given an oracle U_{G} that allows us to distinguish whether an item is marked or not. This oracle is defined as

$$U_{\mathrm{G}}|\boldsymbol{x}\rangle = \begin{cases} -|\boldsymbol{x}\rangle, & \text{if } \boldsymbol{x} = \boldsymbol{w} \\ |\boldsymbol{x}\rangle, & \text{otherwise} \end{cases}. \tag{81}$$

In other words, Grover's oracle can add a phase to the target element $|\boldsymbol{w}\rangle$. We can see that

$$U_{\mathrm{G}} = I - 2|\boldsymbol{w}\rangle\langle\boldsymbol{w}|. \tag{82}$$

We will also define a *diffusion* operator as follows:

$$U_{\mathrm{d}} = 2|\boldsymbol{s}\rangle\langle\boldsymbol{s}| - I, \tag{83}$$

where $|s\rangle$ is the uniform superposition over all items

$$|\boldsymbol{s}\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle. \tag{84}$$

The diffusion operator can be easily implemented as follows:

$$U_{\mathrm{d}} = H^{\otimes n}\big(2|0\rangle\langle 0|^{\otimes n} - I\big)H^{\otimes n}. \tag{85}$$

The quantum circuit for the Grover's search algorithm is illustrated in Figure 13, where U_{G} and U_{d} are given in Eqs. (82) and (83), respectively.

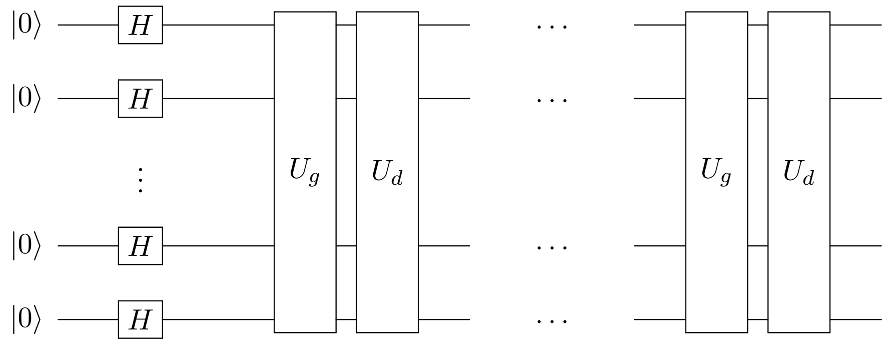


Figure 13: Grover's algorithm

As the first step, the algorithm prepares a uniform superposition state:

$$|\Psi_{t_1}\rangle = H^{\otimes n}|0\rangle^{\otimes n} = \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle\right) \equiv |\boldsymbol{s}\rangle. \tag{86}$$

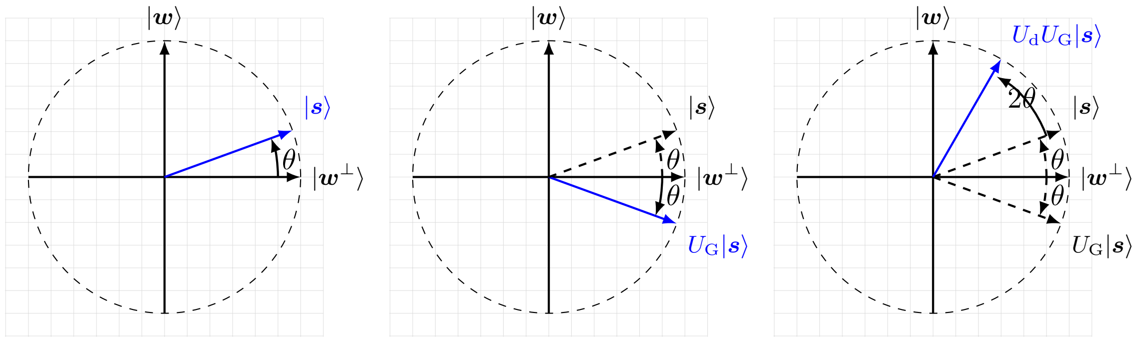
At the next step, the protocol employs Grover's oracle U_{G}:

$$\begin{aligned} |\Psi_{t_2}\rangle &= U_{\mathrm{G}}|\Psi_{t_1}\rangle \\ &= |\boldsymbol{s}\rangle - \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle, \end{aligned} \tag{87}$$

because $\langle \boldsymbol{s}|\boldsymbol{w}\rangle = \frac{1}{\sqrt{N}}$. Then, the protocol employs the diffusion oracle $U_{\rm d}$:

$$
\begin{aligned}
|\Psi_{t_3}\rangle &= U_{\rm d}|\Psi_{t_2}\rangle && (88)\\
&= (2|\boldsymbol{s}\rangle\langle\boldsymbol{s}| - I)\left(|\boldsymbol{s}\rangle - \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle\right) && (89)\\
&= \left(\frac{N-4}{N}|\boldsymbol{s}\rangle\right) + \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle. && (90)
\end{aligned}
$$

We say that "one iteration" of Grover's search algorithm consists of the employment of $U_{\rm G}$ followed by $U_{\rm d}$. In the following, we aim to show that (i) the protocol can find the target $\boldsymbol{w}$ with $O(\sqrt{N})$ iterations with successful probability ≈ 1 and (ii) this is optimal given access to quantum computers.


$|\boldsymbol{w}\rangle$
$|\boldsymbol{s}\rangle$
θ
$|\boldsymbol{w}^{\perp}\rangle$
$|\boldsymbol{w}\rangle$
$|\boldsymbol{s}\rangle$
θ
θ
$|\boldsymbol{w}^{\perp}\rangle$
$U_{\rm G}|\boldsymbol{s}\rangle$
$|\boldsymbol{w}\rangle$
$U_{\rm d}U_{\rm G}|\boldsymbol{s}\rangle$
2θ
$|\boldsymbol{s}\rangle$
θ
$|\boldsymbol{w}^{\perp}\rangle$
θ
$U_{\rm G}|\boldsymbol{s}\rangle$

(a) Step t_1 **(b)** Step t_2 **(c)** Step t_3

Figure 14: Geometric illustration of Grover's search procedure

The easiest way to prove statement (i) is by the following geometric argument in Figure 14. Denote

$$|\boldsymbol{w}^{\perp}\rangle = \frac{1}{\sqrt{N-1}} \sum_{\boldsymbol{x}\neq\boldsymbol{w}} |\boldsymbol{x}\rangle.$$

One can see that the uniform superposition state $|\boldsymbol{s}\rangle$ at the step t_1 can be decomposed into

$$|\boldsymbol{s}\rangle = \sqrt{\frac{1}{N}}|\boldsymbol{w}\rangle + \sqrt{\frac{N-1}{N}}|\boldsymbol{w}^{\perp}\rangle, \tag{91}$$

and the angle θ in Figure 14a corresponds to

$$\sin\theta = \sqrt{\frac{1}{N}}, \qquad \cos\theta = \sqrt{\frac{N-1}{N}}. \tag{92}$$

After the second step, the application of $U_{\rm G}$ leads to

$$
\begin{aligned}
U_{\rm G}|\boldsymbol{s}\rangle &= -\sqrt{\frac{1}{N}}|\boldsymbol{w}\rangle + \sqrt{\frac{N-1}{N}}|\boldsymbol{w}^{\perp}\rangle\\
&= -\sin\theta|\boldsymbol{w}\rangle + \cos\theta|\boldsymbol{w}^{\perp}\rangle. && (93)
\end{aligned}
$$

Geometrically, the oracle U_G reflects the vector $|\boldsymbol{s}\rangle$ along the axis $|\boldsymbol{w}^\perp\rangle$ in Figure 14b. Finally, application of U_d at the third step t_3 to Eq. (93) is equivalent to reflect the state $U_\mathrm{G}|\boldsymbol{s}\rangle$ along the axis $|\boldsymbol{s}\rangle$. Therefore, the application of $U_\mathrm{d}U_\mathrm{G}$ yields

$$U_\mathrm{d}U_\mathrm{G}|\boldsymbol{s}\rangle = \sin 3\theta|\boldsymbol{w}\rangle + \cos 3\theta|\boldsymbol{w}^\perp\rangle. \tag{94}$$

By induction, after k iterations, we have

$$(U_\mathrm{d}U_\mathrm{G})^k|\boldsymbol{s}\rangle = \sin(2k+1)\theta|\boldsymbol{w}\rangle + \cos(2k+1)\theta|\boldsymbol{w}^\perp\rangle. \tag{95}$$

If we measure after k iterations, the probability of obtaining the target element $\boldsymbol{w}$ is

$$p_k := \Pr\{\boldsymbol{w} \text{ appears}\} = \sin((2k+1)\theta)^2. \tag{96}$$

If we choose $k = \frac{\pi}{4\theta} - \frac{1}{2}$, then the Grover's algorithm will produce the state $|\boldsymbol{w}\rangle$ with certainty because $p_k = 1$. However $k = \frac{\pi}{4\theta} - \frac{1}{2}$ will unlikely be an integer, but we can still show that if $\tilde{k}$ is an integer closest to k and $1 \ll N$, then the failure probability decays proportional to N:

$$\begin{aligned} 1 - p_{\tilde{k}} &= \cos((2\tilde{k}+1)\theta)^2 \\ &= \cos((2k+1)\theta + 2(\tilde{k}-k)\theta)^2 \\ &= \cos(\pi/2 + 2(\tilde{k}-k)\theta)^2 \\ &= \sin(2(\tilde{k}-k)\theta)^2 \\ &\leq \sin(\theta)^2 \\ &= \frac{1}{N}, \end{aligned} \tag{97}$$

where the first inequality follows because $|k - \tilde{k}| \leq 1/2$. Since $\arcsin\theta \geq \theta$, then

$$\tilde{k} \leq \frac{\pi}{4\theta} = \frac{\pi}{4}\sqrt{N}. \tag{98}$$

Hence, we can see that the number of iterations is $O(\sqrt{N})$.

Grover's algorithm comes with two important caveats. First, we need to know how many items are marked. Previously we assumed that only one item is marked but if there were two marked items, Grover's algorithm would overrate and decrease the probability of outputting the right item. There are versions of Grover that can work with unknown number of marked items but they are technically more difficult.

The second caveat is the construction of the oracle. In many cases, implementation of the oracle is very costly and it can easily wipe out all the speedup. Such an example would be using Grover search as a database search, where the oracle requires $\Omega(N)$ steps, leading to a quantum algorithm that is slower than a classical one. Thus, Grover's algorithm is not suitable for a database search.

A better way of thinking about Grover's algorithm is in terms of the preimage of function. Assume a function $f : A \to B$ can is easy to compute. For instance, for a given graph we can compute the length for any given path quite easily. However, finding a path corresponding to a fixed length l is an exponentially (in the number of vertices) difficult problem. One can implement the function f on a quantum computer and say that x is marked if $f(x) = l$. Using Grover search, we can then obtain a quadratic speedup for finding paths of a fixed length compared to a naive algorithm that would simply check all possible paths.

10 Complexity theory

At this point, we saw examples of what quantum computers can and cannot do. Instead of studying problems one by one, it is useful to characterize them more broadly which is the goal of complexity theory.

In classical complexity theory, we can define classes of problems that can be solved on a deterministic Turing machine within a certain time constraint. The most common examples are the class **P**, problems that can be solved in polynomial time, and **EXP**, problems that can be solved in exponential time. Clearly, **P** $\subseteq$ **EXP** In fact, **P** is a strict subset of **EXP**. Similarly, we can restrict the computational space (i.e. tape) that the deterministic Turing machine can access and define **PSPACE** as the class of problems that can be solved on a deterministic Turing machine in a polynomial amount of space. We can show that **P** $\subseteq$ **PSPACE** and **PSPACE** $\subseteq$ **EXP** but it is not known if these classes are equal (it is generally believed that they are not).

We can also look at what problems can be solved on a non-deterministic Turing machine - a TM whose transition function allows multiple possible actions for any configuration. We define the class **NP** as the set of problems solvable by a non-deterministic TM in a polynomial time. Since a nondeterministic TMs include deterministic TMs, $P \subseteq NP$. While non-deterministic computers only exist as an abstract concept, we can equivalently define **NP** as the class of problems to which, given a proposed solution, we can verify in polynomial time if the solution is correct or not.

For example, consider the (decision version of the) traveling salesman problem (TSP): Given a list of cities, the distances between each pair of cities, and a number c, is there a route that visits each city exactly once and returns to the origin city that is shorter than c? We do not have an algorithm that can in general solve TSP in polynomial time and most scientists strongly believe that such an algorithm does not exist. However, if someone proposes a route, it is easy to verify if this route visits each city exactly once and returns to the origin city and if it is shorter than c. Thus, the problem belongs to **NP**. There is a subset of problems in **NP** that are **NP**-complete. **NP**-complete can be used to simulate every other problem in **NP**, in this sense, they are the hardest problems in the class. In other words, an algorithm for an **NP**-complete problem can be used with a polynomial overhead to solve any other problem in **NP**. Showing that a problem is **NP**-complete is very strong evidence of it not being solvable on a classical (or even quantum) computer, however, we can still find solve small enough instances, some problems in special cases and often get good enough approximations.

Besides deterministic and non-deterministic TMs, we also have probabilistic TMs (PTM). A PTM can "flip a coin" at each step and then decide on the move based on the result of the coin flip (in addition to its state and symbol on the tape). Thus, the outcome of a PTM is probabilistic. The class of problems that can be solvable on PTM is polynomial time is **BPP**. It is not known if **BPP**=**P** but this might be the case because a pseudo-random generator can often replace true randomness in algorithms.

Many problems about the relationship between complexity classes are still open, most famously whether **P**=**NP** (most believe that this is not true). The relationship between **NP** and **BPP** is unknown and it is conjectured that these classes are incomparable.

We can now define the class of problems that can be efficiently solved on a quantum computer, **BQP**:

Definition 1. *Let $A = (A_{yes}, A_{no})$ be a promise problem and let $c, s : \mathbb{N} \to [0, 1]$ be functions. Then*

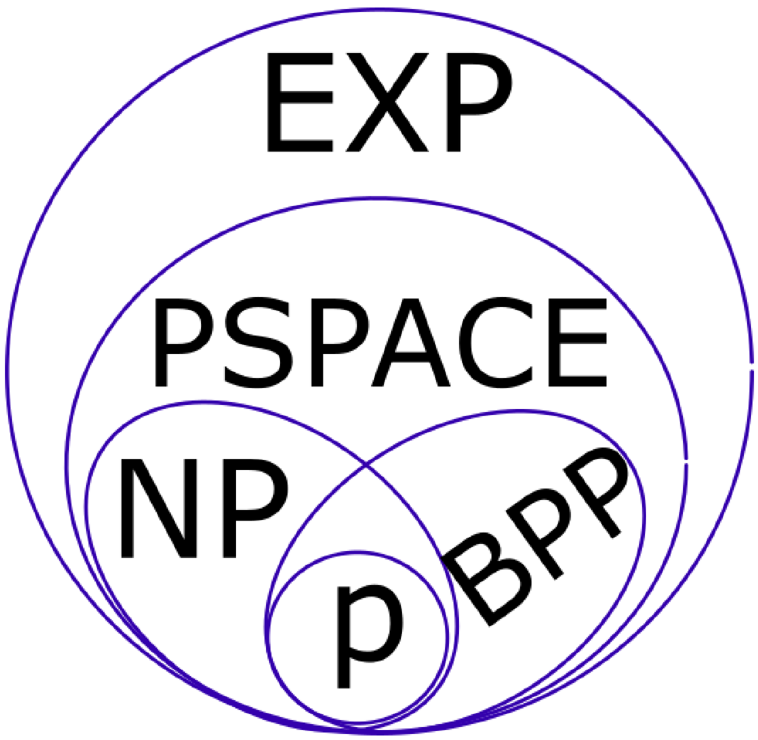


Figure 15: Selected complexity classes and the relationships between them. Some subsets might not be strict.

*$A \in$ **BQP**(c,s) if and only if there exists a polynomial-time uniform family of quantum circuits $Q_n : n \in N$, where Q_n takes n qubits as input and outputs 1 bit, such that*
if $x \in A_{yes}$ then $Pr[Q_{|x|}(x) = 1] \geq c(|x|)$, and
if $x \in A_{no}$ then $Pr[Q_{|x|}(x) = 1] \leq s(|x|)$.
The class **BQP** *is defined as* ***BQP*** *=* ***BQP***$(2/3, 1/3)$.

Class **P** is trivially included in BQP because a quantum computer can always simulate a classical computer by only working in a computational basis. We believe that **BQP** is larger than P, i.e. there exist computers that are intractable classically but solvable on a quantum computer. It is also trivial to see that **BQP** $\subseteq$ **EXP** by writing out all the superpositions and simulating a quantum computer classically using circuits. It is possible to show that **BQP** $\subseteq$ **PSPSACE** and even **BQP** $\subseteq$ **PP**.

The "analog" of class NP is a class **QMA**. These are problems when we can verify if a solution is correct but we might not be able to always find it. **QMA**-complete problems, similarly to **NP**-complete problems are problems that are hard even for quantum computers.

QMA stands for quantum Merlin Arthur. In this setting, Merlin is a powerful result that presents Arthur with a solution to his computational problem. Arthur has access to a quantum computer and if the solution was correct, he has to accept it with a high probability and if Merlin was trying to cheat him, he needs to reject the solution with a high probability.

Definition 2. *Let $A = (A_{yes}, A_{no})$ be a promise problem and let $c, s : N \to [0,1]$ be functions. Then $A \in$ **QMA**(c,s) if and only if there exists a polynomial-time uniform family of quantum circuits $\{Q_n : n \in \mathbb{N}\}$, where Q_n takes $p(n)$ qubits as input for some polynomial p and outputs 1 bit, such that*

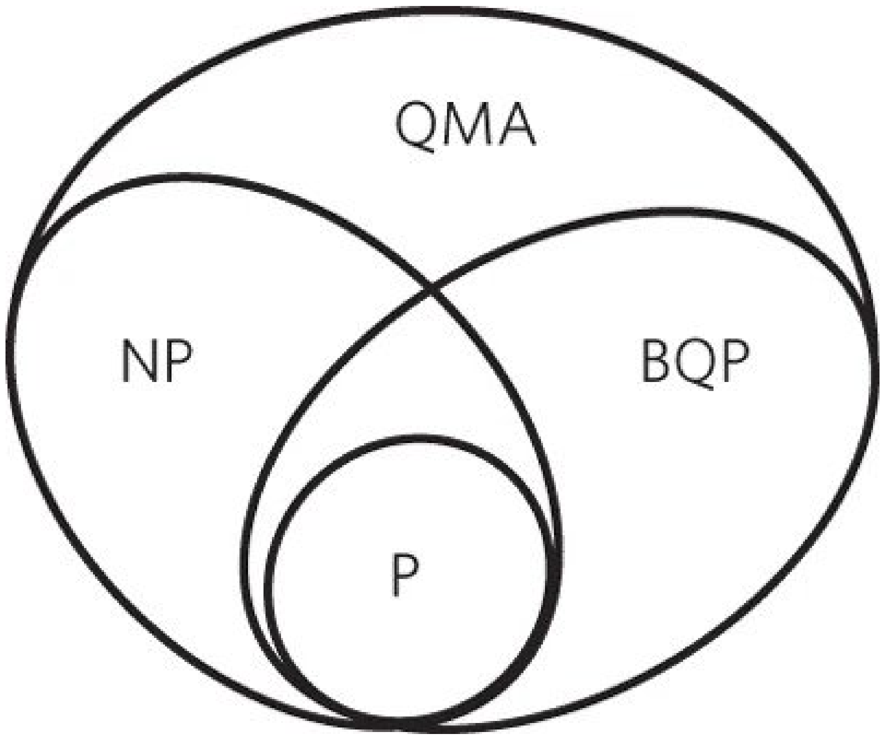


Figure 16: Quantum complexity classes in relation to P and Np. Source: Schuch and Verstraete

- *(Completeness) if $x \in A_{yes}$ then there exists an $p(n)$-qubit state $|\psi\rangle$ such that $Pr\big[Q_n(x, |\psi\rangle) = 1\big] \geq c(n)$), and*
- *(Soundness) if $x \in A_{no}$ then for all $p(n)$-qubit state $|\psi\rangle$, $Pr\big[Q_n(x, |\psi\rangle) = 1\big] \leq s(n)$.*

The class QMA is defined as QMA(2/3,1/3).

The most famous example of an **QMA** complete problem is the problem of computing the ground state of a local Hamiltonian.

Definition 3. *A k-local Hamiltonian H is a summation $H = \sum_{j=1}^{m} H_j$ of local terms H_j acting on at most k qubits (out of n qubits). The k-local Hamiltonian problem is the promise problem with*

Input: (H, a, b) where H is a k-local Hamiltonian, a, b are real numbers such that $b - a \geq 1/poly(n)$,

Yes instances: The smallest eigenvalue of H is at most a,

No instances: The smallest eigenvalue of H is at least b.

For $k \geq 2$, the local Hamiltonian problem is **QMA complete**.

11 Additional reading

UTS courses Quantum Software and Quantum Algorithms both build on the topics from this lecture. Introduction to algorithms [1] is a wonderful introduction textbook to classical algorithms and data structures. A lot of the explanations here are based on [3]. Lecture notes on quantum algorithms from Andrew Childs (available online) cover quantum algorithms in a lot more detail.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2009.

[2] Craig Gidney and Martin Ekerå, *How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits*, Quantum **5** (April 2021), 433.

[3] Phillip Kaye, Raymond Laflamme, Michele Mosca, et al., *An introduction to quantum computing*, Oxford University Press on Demand, 2007.