1 Introduction

Quantum computers are designed to use quantum mechanics to outperform any possible standard, “classical” computer based only on classical physics. In this unit we will introduce the field of quantum computation and study some of the most important ideas in this area. These include quantum computational complexity and the quantum circuit model; the famous algorithms of Shor and Grover for integer factorisation and unstructured search, and the algorithm for simulation of quantum systems; decoherence and quantum error-correction.

1.1 Complementary reading

The Quantum Information Theory unit (MATHM5610) is a prerequisite for this one, and we will aim to follow the notation from that unit where possible. The materials from that unit will therefore be very helpful.

These lecture notes have benefited significantly from the expositions in the following lecture courses, which may be of interest for background reading:

- *Quantum Computation*, Richard Jozsa, University of Cambridge  
  http://www.qi.damtp.cam.ac.uk/node/261  
  The material here on the QFT and Shor’s algorithm follows this exposition closely.

- *Quantum Algorithms*, Andrew Childs, University of Waterloo  
  http://www.cs.umd.edu/~amchilds/qa/  
  An excellent resource for more advanced topics than those covered here.

- *Theory of Quantum Information*, John Watrous, University of Waterloo  
  https://cs.uwaterloo.ca/~watrous/LectureNotes.html  
  A particularly useful resource for the theory of quantum channels.

The following books and survey papers may also be useful:

- *Quantum Computation and Quantum Information*, Nielsen and Chuang  
  Cambridge University Press, 2001  
  The Bible of quantum computing.

- *Classical and Quantum Computation*, Kitaev, Shen and Vyalyi  
  American Mathematical Society, 2002  
  A more concise introduction to many important topics in quantum computation.

- *Quantum algorithms for algebraic problems*, Childs and van Dam  
  Covers many other quantum algorithms than those discussed here.

- *Computational Complexity*, Papadimitriou  
  Addison-Wesley, 1994  
  A comprehensive introduction to classical computational complexity.
1.2 Notation
We write $[n] := \{1, \ldots, n\}$ for the integers between 1 and $n$, and $\mathbb{Z}_n$ for the group of integers modulo $n$, often just identified with the set $\{0, \ldots, n-1\}$. $\lceil x \rceil$, $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the smallest integer $y$ such that $y \geq x$, the largest integer $z$ such that $z \leq x$, and the closest integer to $x$, respectively. We use $\binom{n}{k}$ for the binomial coefficient “$n$ choose $k$”, $n!/(k!(n-k)!)$). We say a randomised or quantum algorithm is “bounded-error” if its failure probability is upper-bounded by some constant strictly less than $1/2$.

We use standard “computer science style” notation relating to asymptotic complexity:

- $f(n) = O(g(n))$ if there exist real $c > 0$ and integer $n_0 \geq 0$ such that for all $n \geq n_0$, $f(n) \leq c g(n)$.

- $f(n) = \Omega(g(n))$ if there exist real $c > 0$ and integer $n_0 \geq 0$ such that for all $n \geq n_0$, $f(n) \geq c g(n)$. Clearly, $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

$O$, $\Omega$ and $\Theta$ can be viewed as asymptotic, approximate versions of $\leq$, $\geq$ and $=$.

1.3 Change log

- v1.0: first version of notes, corresponding to first parts of 2018 unit.
2 Classical and quantum computational complexity

Computational complexity theory aims to classify different problems in terms of their difficulty, or in other words the resources required in order to solve them. Two of the most important types of resources one might study are time (the number of computational steps used by an algorithm solving a problem) and space (the amount of additional work space used by the algorithm). Classically, the formal model underpinning the notion of an “algorithm” is the Turing machine. We will not go into details about this here, instead taking the informal approach that the number of steps used by an algorithm corresponds to the number of lines of code executed when running the algorithm, and the space usage is the amount of memory (RAM) used by the algorithm. For much more on the topic, see the book *Computational Complexity* by Papadimitriou, for example.

Rather than looking at the complexity of algorithms for solving one particular instance of a problem, the theory considers asymptotics: given a family of problems, parametrised by an instance size (usually denoted $n$), we study the resources used by the best possible algorithm for solving that family of problems. Thus the term “problem” is used henceforth as shorthand for “family of problems”. A dividing line between efficient and inefficient algorithms is provided by the notion of polynomial-time computation, where an algorithm running in time polynomial in $n$, i.e. $O(n^c)$ for some fixed $c$, is considered efficient. For example, consider the following two problems:

- Primality testing: given an integer $M$ expressed as $n$ binary digits, is it a prime number?
- Factorisation: given an integer $M$ expressed as $n$ binary digits, output the prime factors of $M$.

As the input is of size $n$, we would like to solve these problems using an algorithm which runs in time $\text{poly}(n)$ (not $\text{poly}(M)$!). No such classical algorithm is known for the factorisation problem; as we will see later, the situation is different for quantum algorithms. However, surprisingly, there is a polynomial-time classical algorithm for the tantalisingly similar problem of primality testing.

An important class of problems is known as decision problems; these are problems that have a yes-no answer. The first of the above problems is a decision problem, while the second is not. But it can be made into a decision problem without changing its underlying complexity significantly:

- Factorisation (decision variant): given integers $M$ and $K$ expressed as $n$ binary digits each, does $M$ have a prime factor smaller than $K$?

It is clear that, if we can solve the usual “search” variant of the factorisation problem, solving the decision variant is easy. Further, solving the decision variant allows us to solve the search variant of the problem using binary search. Given an integer $M$ whose prime factors we would like to determine, and an algorithm which solves the decision variant of the factorisation problem, we can use $O(\log M) = O(n)$ evaluations of this algorithm with different values of $K$ to find the smallest prime factor $F$ of $M$. (First we try $K = \lceil M/2 \rceil$, then either $K = \lceil M/4 \rceil$ or $K = \lceil 3M/4 \rceil$, etc.) The other factors can be found by dividing $M$ by $F$ and repeating. This is a simple example of a reduction: conversion of one problem into another.

A natural way to compare the complexity of problems is via the notion of complexity classes, where a complexity class is simply a set of problems. Some important classical complexity classes are:

- P: the class of decision problems which can be solved in polynomial time by a classical computer.
- **NP**: the class of decision problems such that, if the answer is “yes”, there is a proof of this fact which can be verified in polynomial time by a classical computer.
- **PSPACE**: the class of decision problems which can be solved in polynomial space by a classical computer.

Primality testing is in P, although this was shown for the first time only in 2002. The decision variant of factorisation is in NP, because given a claimed prime factor of \( M \) smaller than \( K \), it can be easily checked whether the claim is correct. However, factorisation is not known to be in P. Every problem in P is automatically in NP, because the verifier can simply ignore any claimed proof and solve the problem directly. In addition, any problem in NP is automatically in PSPACE, because one can loop over all polynomial-length proofs in polynomial space in order to determine whether the answer to a problem instance should be “yes” or “no”. Thus we have \( P \subseteq NP \subseteq PSPACE \).

A problem is said to be NP-complete if it is in NP, and every other problem in NP reduces to it in polynomial time. So NP-complete problems are, informally, the “hardest” problems in NP. These include many problems of practical importance in areas such as timetabling, resource allocation, and optimisation. One simple example is the Subset Sum problem. An instance of this problem is a sequence of integers \( x_1, \ldots, x_n \); our task, given such a sequence, is to determine whether there is a subset of the integers which sums to 0. Given such a subset, we can easily check that it sums to 0; however, finding such a subset seems to require checking exponentially many subsets.

NP stands for “nondeterministic polynomial-time”, not “non-polynomial time”. In fact, it is currently unknown whether every problem in NP can be solved in polynomial time, i.e. whether \( P = NP \). This is the famous P vs. NP question; resolving it would win you a prize of $1M from the Clay Mathematics Institute.

### 2.1 Quantum computational complexity

The basic framework in which quantum computation operates is as follows. We have a system of \( n \) qubits, and choose a basis \( \{ |0\rangle, |1\rangle \} \) for each qubit. By taking tensor products, this gives a basis of the form \( \{ |x\rangle : x \in \{0, 1\}^n \} \) for the whole system, where for conciseness we write \( |x\rangle \) for \( |x_1\rangle \otimes |x_2\rangle \otimes \cdots \otimes |x_n\rangle \). This basis is called the **computational basis**.

A quantum computation is the application of some unitary operator \( U \) to some initial state (usually \( |0\rangle^\otimes n = |0^n\rangle \), which we often just write as \( |0\rangle \)), followed by a measurement of \( k \) of the qubits in the computational basis, giving some outcome \( y \in \{0, 1\}^k \). This outcome is then the output of the computation. If the state of a quantum computer is \( \sum_{x \in \{0, 1\}^n} \alpha_x |x\rangle \) for some coefficients \( \alpha_x \), and we measure all of the qubits, the output is \( x \) with probability \( |\alpha_x|^2 \).

How are we to measure resource usage by a quantum algorithm running on a quantum computer? One framework within which to do this is the **quantum circuit model**. Although in quantum mechanics evolutions of a quantum system are described by unitary operators, not all unitary operators are equally easy to implement physically. We might imagine that, in a real quantum computing experiment, the operations that we can actually perform in the lab are small, “local” ones on just a few qubits at a time. We can build up more complicated unitary operators as products of these small, elementary operations.

Intuitively, a quantum computation running for \( T \) steps and using space \( S \) corresponds to a unitary operation on \( S \) qubits (i.e. acting on \( \mathbb{C}^{2^S} \)) expressed as a product of \( T \) elementary operations picked from some family \( \mathcal{F} \). Each elementary operation is assumed to take time \( O(1) \) and act nontrivially on \( O(1) \) qubits. That is, if \( U \) is one such elementary operation, we assume
that it can be written as \( U = U' \otimes I \), where \( U' \) acts on \( k \) qubits, for some small constant \( k \) (usually, \( k \leq 3 \)). The nontrivial parts \( U' \) of such operations are called \textit{quantum gates}, by analogy with logic gates in classical electronic circuits. The set of allowed quantum gates will depend on our physical architecture. However, it turns out that most “reasonable” sets of gates on \( O(1) \) qubits are universal, in the sense that any unitary operation on \( S \) qubits can be approximately decomposed as a product of these basic operations, acting on different qubits. A sequence of quantum gates is known as a quantum circuit.

A quantum circuit can be drawn as a diagram by associating each qubit with a horizontal “wire”, and drawing each gate as a box across the wires corresponding to the qubits on which it acts. This is easiest to illustrate with an example: the circuit

\[
\begin{array}{c}
\text{H} \\
\text{U} \\
\text{X} \\
\text{V}
\end{array}
\]

corresponds to the unitary operator \((I \otimes V)(U \otimes I)(H \otimes I \otimes X)\) on 3 qubits. Beware that a circuit is read left to right, with the starting input state on the far left, but the corresponding unitary operators act right to left! For convenience, in the diagram we have drawn multi-qubit gates as only acting on nearest-neighbour qubits, but this is not an essential restriction of the model.

The quantum circuit picture also allows us to represent initial state preparation, and final measurement of the qubits in the computational basis, as shown in this more complicated example:

\[
\begin{array}{c}
|0\rangle \\
|0\rangle \\
|0\rangle \\
|0\rangle \\
\text{H} \\
\text{U} \\
\text{Y} \\
\text{X} \\
\text{H} \\
\text{H} \\
\text{W} \\
\end{array}
\]

If we prefer, we can allow intermediate measurements during the circuit; this turns out not to change the power of the model.

Some special gates turn out to be particularly useful. You are already familiar with the Hadamard gate \( H \), which is expressed as the matrix \( \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \) (with respect to the computational basis), and the gates \( X, Y, Z \) corresponding to the Pauli operators. We can think of the \( X \) gate as implementing a NOT operation, as \( X|0\rangle = |1\rangle, X|1\rangle = |0\rangle \). Another useful type of gate will be the “controlled-G” gates. For any gate \( G \), the corresponding controlled-\( G \) gate \( CG \) uses an extra qubit to control whether the gate is applied or not. That is,

\[
CG|0\rangle|\psi\rangle = |0\rangle|\psi\rangle, \quad CG|1\rangle|\psi\rangle = |1\rangle G|\psi\rangle.
\]

In a circuit diagram, this is denoted using a filled circle on the control line:

\[
\begin{array}{c}
\text{G}
\end{array}
\]

A particularly useful such gate is controlled-NOT (CNOT), denoted \( \begin{array}{c}
\oplus
\end{array} \). Written as a matrix
with respect to the computational basis,

\[
\text{CNOT} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{pmatrix}.
\]

For any fixed gate set \(F\), some large unitary matrices cannot be decomposed efficiently in terms of gates from \(F\), in the sense that to write them as a product of gates from \(F\) requires exponentially many such gates. For a rough way of seeing this, consider the problem of producing an arbitrary quantum state of \(n\) qubits \(\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle\), in the special case where each coefficient \(\alpha_x \in \{\pm 1/2^{n/2}\}\). There are \(2^{2^n}\) such states. Any circuit on \(n\) qubits made up of \(T\) gates, each acting on \(k\) qubits, picked from a gate set of size \(G\) can be described by one of \(G \binom{n}{k} T = O(G^n k^T) = O(2^{T \log (Gk)})\) different sequences of gates, so for \(k, G = O(1)\) we need \(T \log n = \Omega(2^n)\) to be able to produce \(2^{\Omega(2^n)}\) different unitary operators, and hence \(2^{2^n}\) different states. A similar argument still works if we allow approximate computation or continuous gate sets.

In general, just as in the classical world, we look for efficient quantum circuits which use \(\text{poly}(n)\) qubits and \(\text{poly}(n)\) gates to solve a problem on an input of size \(n\). The class of decision problems which can be solved by a quantum computer, in time polynomial in the input size, with probability of failure at most 1/3, is known as BQP (“bounded-error quantum polynomial-time”). This class encapsulates the notion of efficient quantum computation. The failure probability bound of 1/3 is essentially arbitrary; it can be reduced to an arbitrarily small constant by repetition and taking the majority vote.

Observe that in the quantum circuit picture we can perform multiple operations in parallel, so we in fact have two possible ways to measure “time” complexity: circuit size (number of gates) and circuit depth (number of time steps to execute all the gates). But these can only differ by a factor of \(O(S)\), where \(S\) is the number of qubits on which the circuit operates.

### 2.2 Classical and reversible circuits

Any classical computation which maps a bit-string (element of \(\{0,1\}^n\)) to another bit-string (element of \(\{0,1\}^m\)) can be broken down into a sequence of logical operations, each of which acts on a small number of bits (e.g. AND, OR and NOT gates). Such a sequence is called a (classical) circuit\(^1\). As a first step in understanding the power of quantum computers, we would like to show that any classical circuit can be implemented as a quantum circuit, implying that quantum computation is at least as powerful as classical computation.

But there is a difficulty: in quantum mechanics, if we wish the state of our system to remain pure, the evolution that we apply has to be unitary, and hence reversible. Some classical logical operations (such as AND, written \(\wedge\)) are not reversible. However, reversible variants of these can be developed using the following trick. If we wish to compute an arbitrary classical operation \(f : \{0,1\}^n \rightarrow \{0,1\}^m\) reversibly, we attach a so-called “ancilla” register of \(m\) bits, each originally set to 0, and modify \(f\) to give a new operation \(f' : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n \times \{0,1\}^m\) which

\(^1\)Where “circuit” is understood in the sense of “electronic circuit”.

performs the map

\[ f'(x, y) = (x, y \oplus f(x)), \]

where \( \oplus \) is bitwise XOR, i.e. addition modulo 2 (so each bit of \( y \oplus f(x) \) is the sum mod 2 of the corresponding bits of \( y \) and \( f(x) \)). Then if we input \( y = 0^m \), we get \( (x, f(x)) \), from which we can extract our desired output \( f(x) \). If we perform \( f' \) twice, we get \( (x, y \oplus f(x) \oplus f(x)) = (x, y) \). So \( f \) is reversible. And any reversible function that maps bit-strings to bit-strings corresponds to a permutation matrix, which is unitary, so can be implemented as a sequence of quantum gates. If we combine many gates of this form to compute a function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \), say, we will finish with an output of the form \((\text{junk}, x, f(x))\). If we wish to remove the junk, we can simply copy the output \( f(x) \) onto a fresh ancilla bit in state 0 by applying a CNOT gate, and then repeat all the previous gates in reverse. As each is its own inverse, the final state of the computation is \((0, x, f(x))\).

To obtain universal deterministic classical computation, it is sufficient to be able to implement the NOT and AND gates. The NOT gate is immediately reversible. Applying the above construction to AND we get the map \((x_1, x_2, y) \mapsto (x_1, x_2, y \oplus (x_1 \land x_2))\) for \( x_1, x_2, y \in \{0, 1\} \). The unitary operator which implements this is then simply the map

\[
|x_1\rangle|x_2\rangle|y\rangle \mapsto |x_1\rangle|x_2\rangle|y \oplus (x_1 \land x_2)\rangle.
\]

Written as a matrix with respect to the computational basis this is

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix},
\]

an operation known as the Toffoli gate. In a circuit diagram, the Toffoli gate is written as “controlled-controlled-NOT”, i.e.

\[
\begin{array}{c}
\begin{array}{c}
•
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
•
\end{array}
\end{array} \quad \oplus
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
•
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
•
\end{array}
\end{array}
\]
\]

Randomised classical computation can also be embedded in a quantum circuit. Imagine we have a classical computation which makes use of some random bits, each of which is 0 or 1 with equal probability. We can simulate this by applying a Hadamard gate to \(|0\rangle\) to produce the state \(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\). Then we can either measure this qubit immediately to obtain a uniformly random bit, or if we prefer, apply classical gates to it and then measure it at the end of the computation; the result is the same either way.

It is known that the Toffoli gate, together with the Hadamard gate, are even sufficient for universal quantum computation. That is, any quantum computation whatsoever can be approximately represented as a circuit of Toffoli and Hadamard gates. Another representative universal set of quantum gates is \(\{H, X, \text{CNOT}, T\}\), where \(T = \left(\begin{array}{cc}1 & 0 \\ 0 & e^{i\pi/4}\end{array}\right)\). It turns out that almost any non-trivial set of gates is universal in this sense; therefore, we generally do not worry about the details of the gate set being used.
2.3 Query complexity

While time complexity is a practically important measure of the complexity of algorithms, it suffers from the difficulty that it is very hard to prove lower bounds on it, and that technical details can sometimes obscure the key features of an algorithm. One way to sidestep this is to use a model which is less realistic, but cleaner and more mathematically tractable: the model of query complexity.

In this model, we assume we have access to an oracle, or “black box”, to which we can pass queries, and which returns answers to our queries. Our goal is to determine some property of the oracle using the minimal number of queries. On a classical computer, we can think of the oracle as a function \( f : \{0, 1\}^n \rightarrow \{0, 1\}^m \). We pass in inputs \( x \in \{0, 1\}^n \), and receive outputs \( f(x) \in \{0, 1\}^m \).

How does this fit into physical reality? We imagine we are given access to the oracle either as a physical device which we cannot open and look inside, or as a circuit which we can see, but for which it might be difficult to compute some property of the circuit. For example, even given a description of a circuit computing some function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \), it might be hard to find an input \( x \) such that \( f(x) = 1 \). Sometimes it is more natural to think of an oracle function \( f \) as a memory storing \( n \) strings of \( m \) bits each, where we can retrieve an arbitrary string at the cost of one query.

We can give a quantum computer access to a oracle using the generic reversible computation construction discussed in the previous section. That is, instead of having a function \( f : \{0, 1\}^n \rightarrow \{0, 1\}^m \), we produce a unitary operator \( O_f \) which performs the map

\[
O_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle.
\]

\( O_f \) is sometimes known as the bit oracle. If \( m = 1 \), so \( f \) returns one bit, it would also make sense to consider an oracle \( U_f \) which does not use an ancilla, but instead flips the phase of an input state \( |x\rangle \) by applying the map

\[
U_f |x\rangle = (-1)^{f(x)} |x\rangle.
\]

This variant is thus sometimes known as the phase oracle. Given access to a bit oracle, we can simulate a phase oracle by attaching an ancilla qubit in the state \( \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \):

\[
O_f |x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}(|x\rangle |f(x)\rangle - |x\rangle |f(x) \oplus 1\rangle) = (-1)^{f(x)} |x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).
\]

Note that the ancilla qubit is left unchanged by this operation, which is called the phase kickback trick. Also note that the effect of the phase oracle is not observable if we apply it to just one computational basis state \( |x\rangle \), but only if we apply it to a superposition:

\[
\sum_{x \in \{0, 1\}^n} \alpha_x |x\rangle \mapsto \sum_{x \in \{0, 1\}^n} (-1)^{f(x)} \alpha_x |x\rangle.
\]

Importantly, note that to implement the oracles \( O_f \) and \( U_f \) we do not need to understand any more about the inner workings of \( f \) than we do classically. That is, if we are given a classical circuit computing \( f \), we can follow a purely mechanical construction to create quantum circuits implementing \( O_f \) and \( U_f \). This is useful because \( f \) itself may have quite complicated behaviour, even if it is expressible as a small circuit, and we may not be able to understand its behaviour completely.

\(^1\text{Note that the notation used here is different to Quantum Information Theory.}\)
2.4 The Deutsch-Jozsa algorithm as a quantum circuit

Recall that the Deutsch-Jozsa algorithm can distinguish between balanced and constant functions \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) with one use of the oracle \( U_f \), whereas an exact classical algorithm for solving the same problem would require exponentially many (in \( n \)) queries to \( f \). That is, if we are promised either that \( f \) is constant or that \( |\{ x : f(x) = 0 \}| = |\{ x : f(x) = 1 \}| = 2^{n-1} \), we can determine which is the case with one quantum query.

We now verify that this algorithm can be implemented as an efficient quantum circuit on \( n \) qubits. Indeed, the circuit is very simple:

\[
\begin{align*}
|0\rangle \rightarrow H \rightarrow U_f \rightarrow \cdots \\
|0\rangle \rightarrow H \rightarrow U_f \rightarrow \cdots \\
\vdots \\
|0\rangle \rightarrow H \rightarrow U_f \\
\end{align*}
\]

The evolution of the input state throughout the circuit is

\[
|0\rangle^\otimes n \mapsto \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} \left( \frac{1}{\sqrt{2^n}} \sum_{y \in \{0,1\}^n} (-1)^{xy} |y\rangle \right)
\]

where \( x \cdot y = \sum_{i=1}^n x_i y_i \). To see the last step, observe that

\[
H^\otimes n |x\rangle = (H|x_1\rangle) \otimes (H|x_2\rangle) \otimes \cdots \otimes (H|x_n\rangle),
\]

and that \( H|x_i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{x_i}|1\rangle) \). So

\[
H^\otimes n |x\rangle = \frac{1}{\sqrt{2^n}} (|0\rangle + (-1)^{x_1}|1\rangle)(|0\rangle + (-1)^{x_2}|1\rangle) \cdots (|0\rangle + (-1)^{x_n}|1\rangle)
\]

\[
= \frac{1}{\sqrt{2^n}} \sum_{y \in \{0,1\}^n} \prod_{i=1}^n (-1)^{x_i} |y\rangle = \frac{1}{\sqrt{2^n}} \sum_{y \in \{0,1\}^n} (-1)^{\sum_{i=1}^n x_i} |y\rangle = \frac{1}{\sqrt{2^n}} \sum_{y \in \{0,1\}^n} (-1)^{xy} |y\rangle.
\]

We can rewrite the final state in the algorithm as

\[
\sum_{y \in \{0,1\}^n} \frac{1}{2^n} \left( \sum_{x \in \{0,1\}^n} (-1)^{f(x) + xy} \right) |y\rangle.
\]

Consider the case \( y = 0^n \). Then \( \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{f(x) + xy} = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} \). If \( f \) is constant, this quantity evaluates to \( \pm 1 \); if \( f \) is balanced, the sum evaluates to 0. So in the former case, the outcome \( 0^n \) is obtained by the final measurement with certainty; in the latter case, an outcome other than \( 0^n \) is obtained with certainty. So we can distinguish between the cases with certainty.

The delicate cancellation effects occurring in this algorithm are a common feature of quantum algorithms; we will see them again later in Shor’s algorithm.