Chapter 3

Classical Computing

In this lecture, we will introduce useful concepts from computer science and discuss the classical circuit computational model of classical computers, which we will generalise in the next lecture to define what quantum computers are.

3.1 Motivating Examples

Computers, classical or quantum, solve **computational problems**. By this we mean the problem of computing a certain function from n to m bits. Practically, all problems you can think about fit this pattern, after appropriate discretisation. The general idea is that we can always specify a problem using an integral approximation of its inputs, and giving an integer x is the same as giving its bitwise decomposition: $x = \sum_{i=0}^{n-1} x_i 2^i$, where $x_i \in \{0, 1\}$. Some examples of computational problems we shall be discussing in this course are:

- Integer factoring: find the prime factors of the integer x. For example, if $x = 15$, return $3, 5$. Here n is the number of bits of x, and m that of its prime factors. Integer factoring is an important problem since its hardness for classical computers underlies security of public-key cryptography which is behind internet transaction. Shor's algorithm is a quantum algorithm that can solve this problem efficiently, compromising in principle security of these cryptographic schemes.

- 3SAT problem: find whether there exists a p-bit string z that satisfies all clauses $C_i(z)$

$$f(x) = C_1(z) \wedge C_2(z) \wedge \cdots \wedge C_q(z)$$

C_i is the logical OR of 3 variables or their negation. For example, $p = 4, q = 2$: $C_1(z) = z_1 \vee z_2 \vee z_3$, $C_2(z) = \neg z_2 \vee \neg z_3 \vee z_4$, whose solution is for example $z_1 = 1, z_2 = z_3 = z_4 = 0$. Each C_i is represented by specifying which variables occur in it and whether they are negated or not, so it can be specified by three $p+1$-bit strings (the $+1$ for the negation). So 3SAT amounts to evaluate the Boolean function which takes as input the n-bit string describing the clauses and returns 1 if there is a satisfying assignment and 0 otherwise. 3SAT is a central problem in computer science, in particular in computational complexity, where it is known as one of the hardest problems. We will see that quantum computers are not believed to solve this efficiently either.

- Matrix function evaluation: given a $2^p \times 2^p$ matrix A, compute a matrix element of a given function of it, for example $(A^{-1})_{ij}$ or $(\exp(A))_{ij}$, where

$$\exp(A) = \mathbf{1} + A + \frac{1}{2}A^2 + \dots$$

  The problem of inverting a matrix allows one to solve linear systems $Ax = b$, which are ubiquitous in optimisation and machine learning for example, and the problem of computing the matrix exponential amounts to being able to solve differential equations of the form $\frac{\mathrm{d}}{\mathrm{d}t}x(t) = Ax(t)$. In this case, we assume that the matrix A can be specified by giving a number of bits that is a polynomial of n, for example by giving the coefficients of the matrix in a given expansion, $A = \sum_{i=1}^{n} a_i E_i$, where E_i are fixed matrices and a_i can be discretised with q bits say. The input will also include which matrix element to compute. The output of the function computed in this case is a discrete approximation to the matrix element of the matrix function to evaluate. Classical computers can solve these problems in time $(2^n)^3$, which becomes prohibitive after moderate values of n of the order of $n = 10$. Instead, we will see that under certain assumptions on the matrix, quantum computers can solve this in time polynomial in n, allowing in principle much bigger sizes to be studied.

3.2 Efficiency of Algorithms

We have seen that computational problems can then be thought of as computing certain functions that take as input an n bit-string and return another bit-string of length m. To solve a computational problem, we then want to find an **algorithm**, that computes this function in an **efficient** way.

Defining what we mean by an efficient algorithm depends on two aspects: the **computational model** we are using – for example, whether we run it on a classical or a quantum computer – and what resource we use to determine efficiency, and for definiteness you can think about the **runtime**. We also consider the worst-case scenario, namely the worst runtime over the possible inputs. Note that there are other resources, such as memory or energy, that are also useful. We call the runtime of an algorithm its time complexity.

In the theory of computational complexity, it is useful to consider how the runtime grows with the input size n rather than the specific time it takes to solve a single instance. The runtime is the sum of the time of each operation required by the algorithm. The exact time it takes to run a single operation on a computer depends on the specificity of the hardware manufacture process and we do not want to consider that: if the runtime is of the form Cn^k, the constant C in front is affected by the exact time of each operation, and so we do not care about it. The **big-O notation** helps with this: we say that a function f is $\mathcal{O}(g(n))$ if there exists an integer n_0 and a positive constant C such that for $n \geq n_0$, $|f(n)| \leq C|g(n)|$. This situation is illustrated in figure 3.1.

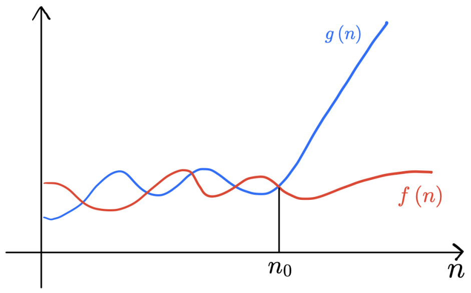


Figure 3.1: $f(n)$ being $\mathcal{O}(g(n))$ means that after some n_0, $f(n)$ is upper bounded by $Cg(n)$.

With these notions, we say that an algorithm is efficient for computing a function $f : \{0,1\}^n \to \{0,1\}^m$ if the runtime is $\mathcal{O}(p(n))$ where $p(n)$ is a **polynomial** of n. In this case, we say that the problem under consideration is easy or tractable. If there does exist any polynomial time algorithm to solve a problem, we instead say that the problem is hard or intractable. To appreciate how fast exponential functions grow, you can think that if $n > 265$, then 2^n is greater than the number of atoms in the universe. If a quantum computer has an exponential speedup with respect to the best classical algorithms, it will be able to run computations which will never be practical for classical computers.

The choice of polynomial seems arbitrary. After all, if the runtime is $\mathcal{O}(n)$ or $\mathcal{O}(n^{100})$, it does make a big difference. However, this choice is justified by two reasons. First, polynomials compose nicely: if you have a polynomial algorithm as a subroutine of another polynomial algorithm, the overall runtime is polynomial. And second, in practice, the majority of the algorithms we have discovered are polynomial with a small exponent. Proving that there is no polynomial time algorithm to solve a problem is hard, and in practice we have a lot of conjectures, which almost everyone believes are true, pointing to the existence of such hard problems. We have already discussed briefly some of the expectations with respect to the existence of efficient classical or quantum algorithms for the examples illustrated above.

3.2.1 Some complexity classes

In complexity theory, we say that a function is in the class $\mathcal{P}$ if there is a classical algorithm that computes the output in time polynomial in n for every binary string x of length n and for every n. This is the class of efficiently solvable problems. Some trivial examples are the sum of two integers, and less trivial examples are determining whether a graph has a pairing of all its vertices such that every pair is an edge of the graph (perfect matching) and determining whether an integer is prime (primality testing).

Many useful classical algorithms use randomness. For example, in machine learning, data is assumed to be drawn from a probability distribution randomly, and in Markov Chain Monte Carlo you toss a coin to decide the next state of the Markov Chain. We call these algorithms probabilistic, as often we get the right result only with a certain probability, but that is enough for many applications. A function is in the class $\mathcal{BPP}$ (bounded-error, probabilistic, polynomial time) if there exists a classical probabilistic polynomial-time algorithm A such that on every input x (independently of its size), we have that the probability of $A(x) \neq f(x)$ at most 1/3. Running this algorithm k times can make the probability of order e^{-k} – the argument is standard and uses a result called the Chernoff bound, but I omit it here due to space.

Another important class is the class $\mathcal{NP}$. It is defined for Boolean functions, namely problems where the answer is either 0 or 1. These problems are called classification or decision problems, since the task is to decide whether $f(x) = 1$, in which case we say it is in class C, or $f(x) = 0$, when x is not in

C. For example, we can consider the class C of composite numbers, those that are not prime. If we do not have an efficient algorithm for computing whether a number has this property, we can however check it if we are given a proof for it, namely a decomposition of a number into its factors. This check is efficient, since we can simply take the multiplication of its factors (which is efficient since multiplication is in $\mathcal{P}$) and see whether the outcome equals the number. $\mathcal{NP}$ is the class of such classification problems, where verifying membership $x \in C$ can be done by a polynomial time verification algorithm that uses a polynomial-size witness. The witness in the example of the composites class is the set of factors. A famous problem in $\mathcal{NP}$ is SAT, where the class is that of clauses with a satisfying assignment and the witness is a string satisfying the clauses. SAT is in fact an $\mathcal{NP}$-complete problem, namely all the problems in $\mathcal{NP}$ can be efficiently reduced to it, and is in this sense the hardest problem in $\mathcal{NP}$. Another example we have encountered is factoring integers, if we formulate the input as a triple (x, a, b) and the class is that of integers x which have a prime factor in the interval $[a, b]$. Decision problems in $\mathcal{P}$ are also in $\mathcal{NP}$ since we can simply take the verifier to be the efficient algorithm for computing membership in the class. Whether $\mathcal{P} = \mathcal{NP}$ is one of the most important open problems in computer science and beyond and is widely conjectured to be false. This conjecture implies that no efficient algorithm exists for $\mathcal{NP}$-complete problems.

Finally, we also define the class $\mathcal{PSPACE}$ of functions computable with a polynomial amount of memory but possibly exponential time, and is believed to be strictly larger than $\mathcal{NP}$. This class will be relevant to understand the relationship between classical and quantum algorithms.

3.3 Classical Circuits

To develop a theory of computation, it is useful to model the components of a computer as mathematical functions, abstracting away the details of the implementation. We are going to discuss here a mathematical model of a classical computer, which will help us contrasting it with the model of a quantum computer we will introduce in the next lecture. It turns out that many equivalent models of classical computers exist. The most famous one is the Turing machine, but for our purposes, the most useful one is that of classical circuits. A classical circuit is made up of **wires** and **gates**. Wires carry classical information, namely bits. Gates transform that information. An example of a classical circuit is in figure 3.2. Computation time flows from left to right. Wires are represented as lines where the bit is carried

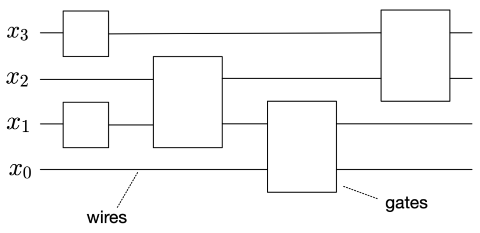


Figure 3.2: A classical circuit with $n = 4$ input bits x_0, x_1, x_2, x_3 and $m = 4$ outputs. Note that we label the bits from 0 to $n-1$ and from bottom to top.

through unchanged. Boxes represent gates that act non-trivially only on bits associated to the input wires. A gate with k input wires and k output wires is a function $g : \{0,1\}^k \to \{0,1\}^k$ and is called **reversible** if there exists an inverse gate g^{-1} such that $g(g^{-1}(x)) = x$ for all inputs x. Reversible transformation will play an important role in quantum computing, as they can be used directly as quantum gates as we shall see. Note that we can always embed a gate with a different number of inputs and outputs, such as the AND gate, which takes in two bits and returns one bit equal to 1 if and only if both inputs are 1, into a gate with the same number of inputs and inputs. Moreover, we can always make that embedding reversible. Indeed let $f : \{0,1\}^k \to \{0,1\}^\ell$ be an arbitrary function and consider the following function $R_f : \{0,1\}^{k+\ell} \to \{0,1\}^{k+\ell}$:

$$R_f : (x, y) \mapsto (x, y \oplus f(x))\,.$$

Here x has k bits and y has ℓ bits and $\oplus$ is the bitwise XOR:

$$\begin{aligned}&x_{n-1}\cdots x_1 x_0 \oplus y_{n-1}\cdots y_1 y_0 = z_{n-1}\cdots z_1 z_0\,, \quad z_i = x_i \oplus y_i\\ &0\oplus 0 = 1 \oplus 1 = 0\,, \quad 0 \oplus 1 = 1 \oplus 0 = 1\,.\end{aligned}$$

R_f is reversible since $x \oplus x = 0$ for any x, so:

$$(R_f)^2 : (x, y) \mapsto ((x, y \oplus f(x) \oplus f(x)) = (x, y)\,,$$

and allows us to compute f by simply acting on $(x, 0)$. This discussion shows that we can indeed restrict ourselves to reversible circuits to implement any (non-reversible) function, and the overhead in using R_f instead of f is polynomial, and it does not matter from the algorithmic efficiency point of view we introduced.

In fact, in the circuit model of classical computers, the number of steps is the number of **elementary gates** that are used by the algorithm. By elementary gates we mean a fixed set of gates that we can use – we will see many examples of those below – and importantly, we require that the number of inputs and outputs of every gates in this elementary set is constant as the size of the problem n grows. Thus an algorithm is efficient when run on the classical circuit model if the number of elementary gates grows only polynomially with the size of the input n. It turns out that many choices of elementary gate sets are equivalent in the sense that we can implement gates in one set in terms of gates in another set using only a number of operations that is constant in the problem input size n. In this sense, which elementary gate set used is irrelevant if we care about the asymptotic complexity. We call this set of elementary gates universal if any function can be implemented out of compositions of it – for example, AND and XOR gates form a universal set.

3.4 Matrix Representation of Classical Gates

3.4.1 Single Bit Gates

Recall that a bit is a variable with two possible values: $x \in \{0, 1\}$. We can represent it as a two-dimensional one-hot vector:

$$0 \mapsto |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} , \quad 1 \mapsto |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$|v\rangle$ is called "ket" and the notation used here and below is called Dirac notation, from Paul Dirac, the quantum physicist who introduced it. This notation is standard in quantum computing, so we are going to use it throughout the course. The ket notation should be thought to be analogous to the arrow notation of vectors $\vec{v}$. Differently from conventions with the arrow notation however, we simply write $|x\rangle$ instead of $\vec{e}_x$ to denote the one-hot vectors above.

The reason for the name ket comes from the notion of scalar product or bracket. In fact, we define bras associated with kets by

$$\langle 0| = \begin{pmatrix} 1 & 0 \end{pmatrix} , \quad \langle 1| = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

and their scalar products denoted as $\langle x|y\rangle$

$$\begin{aligned}
\langle 0|0\rangle &= \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 1\\
\langle 0|1\rangle &= \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = 0\\
\langle 1|0\rangle &= \begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 0\\
\langle 1|1\rangle &= \begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = 1\,.
\end{aligned}$$

In the language of linear algebra, $|0\rangle, |1\rangle$ form an orthonormal basis of $\mathbb{R}^2$ since $\langle x|y\rangle = \delta_{xy}$, with δ_{xy} the Kronecker delta: 1 if $x = y$ and 0 otherwise. We also denote by $|x\rangle\langle y|$ the matrices:

$$|0\rangle\langle 1| = \begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}0 & 1\end{pmatrix} = \begin{pmatrix}0 & 1\\0 & 0\end{pmatrix}, \quad |1\rangle\langle 0| = \begin{pmatrix}0 & 0\\1 & 0\end{pmatrix}$$

$$|0\rangle\langle 0| = \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}, \quad |1\rangle\langle 1| = \begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$$

Note that these formulas are compatible with the standard rules of matrix multiplication: in $\langle x|y\rangle$ we multiply a 1×2 by a 2×1 matrix producing a 1×1, i.e. a scalar. While in $|x\rangle\langle y|$ we multiply a 2×1 by a 1×2 matrix producing a 2×2 matrix. Also note that the notation composes as you would expect: multiplying the matrix $|x\rangle\langle y|$ by the vector $|z\rangle$ produces the new vector:

$$(|x\rangle\langle y|)\,|z\rangle = \langle y|z\rangle\,|x\rangle$$

We can add kets, bras and matrices as you would do with the arrow notation, e.g. the following is a representation of the 2×2 identity matrix:

$$|0\rangle\langle 0| + |1\rangle\langle 1| = \mathbf{1}_2$$

Finally, if one has a matrix

$$M = \begin{pmatrix}M_{00} & M_{01}\\M_{10} & M_{11}\end{pmatrix}$$

the matrix element M_{xy} can be obtained by taking the inner product $\langle x| M |y\rangle$. For example,

$$\langle 0| M |1\rangle = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = M_{01} \,.$$

With these preparations, we are now going to discuss single bit gates. There are four possible Boolean functions from bit x to bit y, represented as matrices in Dirac notation:

x	y
0	0
1	0

$$M = |0\rangle \left(\langle 0| + \langle 1|\right) ,$$

x	y
0	1
1	1

$$M = |1\rangle \left(\langle 0| + \langle 1|\right)$$

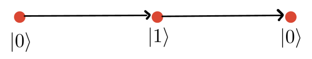
x	y
0	1
1	0

$$M = |0\rangle \langle 1| + |1\rangle \langle 0| = X \,,$$

x	y
0	0
1	1

$$M = |0\rangle \langle 0| + |1\rangle \langle 1| = \mathbf{1}_2$$

A useful mental model which connects computation with physics and underlies some of the ideas behind quantum computing, is to think about the action of the gate M as defining a deterministic dynamical system: $|x\rangle^{t+1} = M |x\rangle^t$. For example if $M = X$, the evolution can be represented as in figure 3.3.



t	0	1	2
$\lvert x\rangle^t$	$\lvert 0\rangle$	$\lvert 1\rangle$	$\lvert 0\rangle$

Figure 3.3:

Reversible functions are those that can be inverted, X (also called the NOT gate), $\mathbf{1}_2$, while the other two are not invertible. If M is invertible the dynamics generated by M can be time reversed, as in figure 3.4.

3.4.2 States of two bits

There are 4 possible states of 2 bits which we denote by $|x_1 x_0\rangle , x_0, x_1 \in \{0, 1\}$: $|00\rangle , |01\rangle , |10\rangle , |11\rangle$, and to which we assign a four-dimensional one-

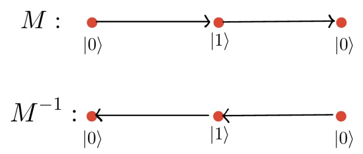


Figure 3.4:

hot vector:

$$|00\rangle \equiv |0\rangle_2 = \begin{pmatrix}1\\0\\0\\0\end{pmatrix},|01\rangle \equiv |1\rangle_2 = \begin{pmatrix}0\\1\\0\\0\end{pmatrix},|10\rangle \equiv |2\rangle_2 = \begin{pmatrix}0\\0\\1\\0\end{pmatrix},|11\rangle \equiv |3\rangle_2 = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$$

Here the notation $|x\rangle_2$ means a vector of dimension $2^2 = 4$. The identification with four-dimensional vectors can be written compactly as

$$|x_1x_0\rangle \equiv \left|2^1x_1 + 2^0x_0\right\rangle_2$$

which means that if we label the bits from right to left, the associated one-hot vectors have a 1 in the position corresponding to the integer value represented by the bit string.

Now we introduce the tensor product of two vectors by:

$$|\psi\rangle = \begin{pmatrix}\psi_1\\\psi_2\end{pmatrix},|\phi\rangle = \begin{pmatrix}\phi_1\\\phi_2\end{pmatrix}, \quad |\psi\rangle \otimes |\phi\rangle = \begin{pmatrix}\psi_1\,|\phi\rangle\\\psi_2\,|\phi\rangle\end{pmatrix} = \begin{pmatrix}\psi_1\phi_1\\\psi_1\phi_2\\\psi_2\phi_1\\\psi_2\phi_2\end{pmatrix}$$

We have that $|x_1x_0\rangle = |x_1\rangle \otimes |x_0\rangle$ as we can easily check:

$$|0\rangle \otimes |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |0\rangle \otimes |1\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix},$$

$$|1\rangle \otimes |0\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |1\rangle \otimes |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$$

In summary, we can associate to a bit string of length 2 a basis vector $|x_1x_0\rangle$ which corresponds to $|x\rangle_2$ with $x = 2^1x_1 + 2^0x_0$, and in turns we have $|x_1x_0\rangle \equiv |x_1\rangle \otimes |x_0\rangle$. The inner product of $|\psi_1\rangle \otimes |\phi_1\rangle$ with $|\psi_2\rangle \otimes |\phi_2\rangle$ is given by

$$(\langle\psi_2| \otimes \langle\phi_2|)(|\psi_1\rangle \otimes |\phi_1\rangle) = \langle\psi_2|\psi_1\rangle \langle\phi_2|\phi_1\rangle$$

It is a good exercise to verify this by comparing the l.h.s. with the r.h.s., see also exercise 3.4.9. See Appendix A for more on tensor products.

3.4.3 Transformations of two bits

Functions of 2 bits can be written as $2^2 \times 2^2$ matrices. For example, the following function swaps the two bits and can be represented equivalently using truth tables, Dirac notation and matrix notation as follows:

x_1	x_0	y_1	y_0
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

$$M = |00\rangle\langle 00| + |10\rangle\langle 01| + |01\rangle\langle 10| + |11\rangle\langle 11| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In general, reversible transformations map distinct bit strings to distinct bit strings, so they coincide with **permutation** matrices.

As already noticed, M in the example above is the SWAP: $S_{01}|x\rangle|y\rangle = |y\rangle|x\rangle$:

$$S_{01} = |00\rangle\langle 00| + |10\rangle\langle 01| + |01\rangle\langle 10| + |11\rangle\langle 11|$$

Note $S_{01} = S_{10}$.

Another important example of reversible transformation of 2 bits is the CNOT (Controlled-NOT) C_{ij}. This is not symmetric. i is called control bit, and j the target bit, and the action of C_{10} is:

$$C_{10}|x\rangle|y\rangle = |x\rangle|y \oplus x\rangle\ , \quad C_{01}|x\rangle|y\rangle = |x \oplus y\rangle|y\rangle\ ,$$

where recall that $\oplus =$ is the XOR: $0 \oplus 0 = 0, 0 \oplus 1 = 1 \oplus 0 = 1, 1 \oplus 1 = 0$. Recall also that in our notation we label bits from right to left, so the 0-th bit is the one on the right, and the 1-st bit is the one on the left. The action of C_{ij} is that the target (j-th) bit is flipped if the control (i-th) bit is 1 and otherwise nothing happens. In Dirac and matrix notation we can write

$$C_{10} = |00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} .$$

Another example of reversible transformation of 2 bits is $A \otimes B$ where A, B are reversible 1 bit transformations, i.e. $A, B \in \{\mathbf{1}_2, X\}$. The tensor product of matrices acts on a tensor product of vectors by

$$A \otimes B\, |\psi\rangle \otimes |\phi\rangle = A\, |\psi\rangle \otimes B\, |\phi\rangle$$

so that $A \otimes B$ is associated to the $MN \times MN$ matrix:

$$A \otimes B = \begin{pmatrix} A_{00}B & \dots & A_{0,N-1}B \\ \vdots & \ddots & \vdots \\ A_{N-1,0}B & \dots & A_{N-1,N-1}B \end{pmatrix}$$

In our case, we have:

$$X \otimes \mathbf{1}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} ,$$

$$\mathbf{1}_2 \otimes X = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$X \otimes X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} .$$

You can also verify the following representations:

$$C_{10} = |0\rangle\langle 0| \otimes \mathbf{1}_2 + |1\rangle\langle 1| \otimes X\, ,$$
$$C_{01} = \mathbf{1}_2 \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1|\, .$$

3.4.4 States of n bits

For n bits we have 2^n possible bit strings:

$$|x_{n-1}\cdots x_0\rangle = |x_{n-1}\rangle \otimes \cdots \otimes |x_0\rangle \equiv |x\rangle_n \ , \quad x = \sum_{j=0}^{n-1} 2^j x_j \ ,$$

where $\{|x\rangle_n\}_{x=0}^{N-1}$ is a basis of $\mathbb{C}^N$, $N = 2^n$. The n-fold tensor product is defined recursively and using the more general formula:

$$|\psi\rangle = \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{N-1} \end{pmatrix}, |\phi\rangle = \begin{pmatrix} \phi_0 \\ \vdots \\ \phi_{M-1} \end{pmatrix} , \quad |\psi\rangle \otimes |\phi\rangle = \begin{pmatrix} \psi_0 |\phi\rangle \\ \psi_1 |\phi\rangle \\ \vdots \\ \psi_{N-1} |\phi\rangle \end{pmatrix}$$

Note that the output is a vector of length MN. For example, for $n = 3$ we have

$$|110\rangle = |1\rangle \otimes |1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |6\rangle_3 \ .$$

To ease notation we will sometimes omit the tensor product symbol, since there is no ambiguity as there is no other notion of product between kets:

$$|x_2\rangle \otimes |x_1\rangle \otimes |x_0\rangle \equiv |x_2\rangle |x_1\rangle |x_0\rangle$$

3.4.5 Transformations of n bits

Functions of n bits are represented by $2^n \times 2^n$ matrices and reversible ones are the permutations of the 2^n bit strings. For example, the SWAP of the 1-st and 3-rd bits is:

$$S_{31} |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle = |x_1\rangle |x_2\rangle |x_3\rangle |x_0\rangle$$

Another example is the CNOT C_{ij} (recall: i control, j target)

$$C_{20} |x_3\rangle |x_2\rangle |x_1\rangle |x_0\rangle = |x_3\rangle |x_2\rangle |x_1\rangle |x_0 \oplus x_2\rangle$$

We will use the following shortcut notation for a 2×2 matrix A acting on the i-th vector of an n-fold tensor product:

$$A_i = \mathbf{1}_2 \otimes \mathbf{1}_2 \otimes \cdots \otimes A \otimes \cdots \otimes \mathbf{1}_2 \,.$$

For example for $n = 3$ we will write

$$X_1 = \mathbf{1}_2 \otimes X \otimes \mathbf{1}_2 \,, \quad X_1 \left|x_2\right\rangle \left|x_1\right\rangle \left|x_0\right\rangle = \left|x_2\right\rangle \left|1 - x_1\right\rangle \left|x_0\right\rangle$$

where we wrote $1 - x_1$ as the effect of flipping the bit x_1 is to map 0 to 1 and 1 to 0, which is equivalent to map x_1 to $1 - x_1$. Note that operators acting on different bits commute no matter what they are: $A_i B_j = B_j A_i$ if $i \neq j$. For example for $n = 6$:

$$A_3 B_1 = \mathbf{1}_2 \otimes \mathbf{1}_2 \otimes A \otimes \mathbf{1}_2 \otimes B \otimes \mathbf{1}_2 = B_1 A_3$$

In a similar way we denote by A_{ij} the 4×4 matrix that acts non-trivially only on the i and j terms of an n-fold tensor product. This notation is consistent with that of S_{ij} and C_{ij} introduced above.

Exercises

Important exercises

Exercise 3.4.1. *Compute the matrix corresponding to the CNOT gate C_{01} and compare against C_{10}.*

Exercise 3.4.2. *There is nothing special about the control bit in the CNOT gate to take value one. Consider a control gate that flips the target bit if the control bit is zero. What is its matrix representation?*

Exercise 3.4.3. *The Toffoli gate is defined as $T \left|x\right\rangle \left|y\right\rangle \left|z\right\rangle = \left|x\right\rangle \left|y\right\rangle \left|z \oplus xy\right\rangle$, where xy is the AND of x and y. Compute its truth table and determine its inverse and whether it is a reversible gate.*

Exercise 3.4.4. *Verify that $S_{01} = C_{01} C_{10} C_{01}$*

Exercise 3.4.5. *Introduced $B = \left|1\right\rangle \left\langle 1\right| , \tilde{B} = \mathbf{1} - B = \left|0\right\rangle \left\langle 0\right|$, projectors onto 1 and 0 bit, prove*

(i) $BX = X\tilde{B}$

(ii) $S_{ij} = B_i B_j + \tilde{B}_i \tilde{B}_j + (X_i X_j)(B_i \tilde{B}_j + \tilde{B}_i B_j)$

(iii) $C_{ij} = \tilde{B}_i + X_j B_i$

Additional exercises

—

Exercise 3.4.6. *Compute the number of possible functions $f : \mathcal{X} \to \mathcal{Y}$ where input and output sets are finite and with dimensions $|\mathcal{X}| = N$ and $|\mathcal{Y}| = M$. What is the number of functions from n bits to n bits?*

—

Exercise 3.4.7. *Compute*

(i) $\begin{pmatrix} 1 \\ 2 \end{pmatrix} \otimes \begin{pmatrix} 3 \\ 4 \end{pmatrix}$

(ii) $\begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$

(iii) $H \otimes H |0\rangle \otimes |0\rangle$, *with* $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

—

Exercise 3.4.8. *Is $|\psi\rangle \otimes |\phi\rangle = |\phi\rangle \otimes |\psi\rangle$ for generic two-dimensional vectors $|\psi\rangle, |\phi\rangle$? If not, for which choice of vectors are they equal?*

So these are equal if $\alpha\delta = \beta\gamma$.

—

Exercise 3.4.9. *Properties of tensor product. Below we use the complex vector space $\mathbb{C}^N$. If you are not familiar with complex numbers – we will introduce them in the next lecture – you can do the exercise replacing $\mathbb{C}^N$ with $\mathbb{R}^N$.*

(i) Show the bilinearity property:

$$(\alpha |v\rangle + \alpha' |v'\rangle) \otimes (\beta |w\rangle + \beta' |w'\rangle) = \alpha\beta |v\rangle \otimes |w\rangle + \alpha\beta' |v\rangle \otimes |w'\rangle + \alpha'\beta |v'\rangle \otimes |w\rangle + \alpha'\beta' |v'\rangle \otimes |w'\rangle,$$

where $\alpha, \alpha', \beta, \beta' \in \mathbb{C}$, $|v\rangle, |v'\rangle \in \mathbb{C}^N$, $|w\rangle, |w'\rangle \in \mathbb{C}^M$.

(ii) Denote by $|i\rangle^N$ the i-th canonical basis vector of $\mathbb{C}^N$. Show that $|i\rangle^N \otimes |j\rangle^M = |iM + j\rangle^{MN}$.

(iii) Show that for $|v\rangle, |v'\rangle \in \mathbb{C}^N$ and $|w\rangle, |w'\rangle \in \mathbb{C}^M$ we have:

$$(\langle v| \otimes \langle w|)(|v'\rangle \otimes |w'\rangle) = \langle v|v'\rangle \langle w|w'\rangle.$$

Exercise 3.4.10. *Show that the Toffoli gate enables us to compute the logical AND of two bits and the logical NOT. (This shows that we can build any Boolean operation using reversible gates since all Boolean operations can be decomposed in terms of AND and NOT.)*

Exercise 3.4.6. *Compute the number of possible functions $f : \mathcal{X} \to \mathcal{Y}$ where input and output sets are finite and with dimensions $|\mathcal{X}| = N$ and $|\mathcal{Y}| = M$. What is the number of functions from n bits to n bits?*

Maybe it should all be done in the Dirac notation? Then I can do it all again.

a) $\forall y \in Y \; \exists x \in X : f(x) = y \implies \# f = N^M$

b) $|g : 2^n \to 2^n| = 2^{n^{(2^n)}} = 2^{n 2^n}$

Exercise 3.4.7. *Compute*

(i) $\begin{pmatrix} 1 \\ 2 \end{pmatrix} \otimes \begin{pmatrix} 3 \\ 4 \end{pmatrix}$

(ii) $\begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$

(iii) $H \otimes H |0\rangle \otimes |0\rangle$, *with* $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

a) $\begin{pmatrix} 1\cdot 3 \\ 1 \cdot 4 \\ 2 \cdot 3 \\ 2 \cdot 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 6 \\ 8 \end{pmatrix}$

b) $\begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix}$

c) $H \otimes H |0\rangle \otimes |0\rangle = \frac{1}{2} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} |0\rangle \otimes |0\rangle = \frac{1}{2} \cdot \left(-||- \right) \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

$= H \otimes H |0\rangle \otimes |0\rangle = H \otimes H |00\rangle = \frac{1}{2} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$

$= \frac{1}{2} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{2} \cdot \left(|00\rangle + |01\rangle + |10\rangle + |11\rangle \right)$

Exercise 3.4.8. *Is $|\psi\rangle \otimes |\phi\rangle = |\phi\rangle \otimes |\psi\rangle$ for generic two-dimensional vectors $|\psi\rangle, |\phi\rangle$? If not, for which choice of vectors are they equal?*

So these are equal if $\alpha\delta = \beta\gamma$.

$|x\rangle \otimes |y\rangle = \begin{pmatrix} x_1 |y\rangle \\ x_2 |y\rangle \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_1 y_2 \\ x_2 y_1 \\ x_2 y_2 \end{pmatrix} \overset{?}{=} \begin{pmatrix} y_1 x_1 \\ y_1 x_2 \\ y_2 x_1 \\ y_2 x_2 \end{pmatrix} = \begin{matrix} y_1 |x\rangle \\ y_2 |x\rangle \end{matrix} = |y\rangle \otimes |x\rangle$

$x_1 y_2 = y_1 x_2$
$x_2 y_1 = y_2 x_1$ $\implies x_1 y_2 = y_1 \cdot x_2$ → for all solutions under this condition

Exercise 3.4.9. *Properties of tensor product. Below we use the complex vector space $\mathbb{C}^N$. If you are not familiar with complex numbers – we will introduce them in the next lecture – you can do the exercise replacing $\mathbb{C}^N$ with $\mathbb{R}^N$.*

(i) *Show the bilinearity property:*

$$(\alpha |v\rangle + \alpha' |v'\rangle) \otimes (\beta |w\rangle + \beta' |w'\rangle) = \alpha\beta |v\rangle \otimes |w\rangle + \alpha\beta' |v\rangle \otimes |w'\rangle + \alpha'\beta |v'\rangle \otimes |w\rangle + \alpha'\beta' |v'\rangle \otimes |w'\rangle,$$

where $\alpha, \alpha', \beta, \beta' \in \mathbb{C}$, $|v\rangle, |v'\rangle \in \mathbb{C}^N$, $|w\rangle, |w'\rangle \in \mathbb{C}^M$.

(ii) *Denote by $|i\rangle^N$ the i-th canonical basis vector of $\mathbb{C}^N$. Show that $|i\rangle^N \otimes |j\rangle^M = |iM + j\rangle^{MN}$.*

(iii) *Show that for $|v\rangle, |v'\rangle \in \mathbb{C}^N$ and $|w\rangle, |w'\rangle \in \mathbb{C}^M$ we have:*

$$(\langle v| \otimes \langle w|)(|v'\rangle \otimes |w'\rangle) = \langle v|v'\rangle \langle w|w'\rangle.$$

b) $$|i\rangle^N \otimes |j\rangle^M = \begin{pmatrix} 0|j\rangle^M \\ 0|j\rangle^M \\ \vdots \\ 1|j\rangle^M \\ \vdots \\ 0|j\rangle^M \end{pmatrix}$$

$$\rightarrow 0 \cdot \begin{pmatrix} 0 \\ \vdots \\ j \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} | \\ 0 \\ | \end{pmatrix} \Big\} N$$

$$= \quad \rightarrow 1 \cdot \begin{pmatrix} 0 \\ \vdots \\ j \\ \vdots \end{pmatrix} = |j\rangle^M \Big\} \quad \vdots$$

$$= \begin{pmatrix} | \\ 0 \\ | \\ |j\rangle^M \\ | \\ 0 \\ | \end{pmatrix} \Big\} M$$

c) $$\left(\langle v| \otimes \langle w|\right) \cdot \left(|v'\rangle \otimes |w'\rangle\right) = \begin{pmatrix} v_0 \langle w| & v_1 \langle w| & \cdots & v_n \langle w| \end{pmatrix} \cdot \begin{pmatrix} v_0' |w'\rangle \\ v_1' |w'\rangle \\ \vdots \\ v_n' |w'\rangle \end{pmatrix} =$$

$$= v_0 \cdot v_0' \langle w|w'\rangle + v_1 \cdot v_1' \langle w|w'\rangle + \ldots\ldots\ldots + v_n \cdot v_n' \cdot \langle w|w'\rangle$$

$$= \left(v_0 \cdot v_0' + v_1 \cdot v_1' \cdots\right) \cdot \langle w|w'\rangle$$

$$= \langle v|v'\rangle \cdot \langle w|w'\rangle$$ $\rightarrow$ this is the desired result!

$$|x\rangle \otimes |y\rangle = \begin{pmatrix} x_1 |y\rangle \\ x_2 |y\rangle \\ \vdots \\ x |y\rangle \end{pmatrix}$$

$$\langle x| \otimes \langle y| = \begin{pmatrix} x_1 |y\rangle & x_2 |y\rangle & \ldots & x_n |y\rangle \end{pmatrix}$$

$$\langle v| = \sum_i v_i \langle i| \qquad |v'\rangle = \sum_j v_j' |j\rangle$$
$$\langle w| = \sum_h w_h \langle h| \qquad |w'\rangle = \sum_l w_l' |l\rangle$$

$$(\langle v| \otimes \langle w|) \cdot (|v'\rangle \otimes |w'\rangle) =$$

$$\left(\sum_i v_i \langle i| \cdot \sum_h w_h \langle h|\right) \cdot \left(\sum_j v_j' |j\rangle \cdot \sum_l w_l' |l\rangle\right)$$

$$\left(\sum_i \sum_h v_i w_h \cdot \langle i,h|\right) \cdot \left(\sum_j \sum_l v_j' w_l' \cdot |j,l\rangle\right)$$

$$= \sum_{i,j} \sum_{h,l} v_i w_h v_j' w_l' \langle i,h|j,l\rangle$$

Exercise 3.4.1. *Compute the matrix corresponding to the CNOT gate C_{01} and compare against C_{10}.*

Exercise 3.4.2. *There is nothing special about the control bit in the CNOT gate to take value one. Consider a control gate that flips the target bit if the control bit is zero. What is its matrix representation?*

Exercise 3.4.3. *The Toffoli gate is defined as $T|x\rangle|y\rangle|z\rangle = |x\rangle|y\rangle|z \oplus xy\rangle$, where xy is the AND of x and y. Compute its truth table and determine its inverse and whether it is a reversible gate.*

Exercise 3.4.4. *Verify that $S_{01} = C_{01}C_{10}C_{01}$*

Exercise 3.4.5. *Introduced $B = |1\rangle\langle 1|, \tilde{B} = \mathbf{1} - B = |0\rangle\langle 0|$, projectors onto 1 and 0 bit, prove*

(i) $BX = X\tilde{B}$

(ii) $S_{ij} = B_iB_j + \tilde{B}_i\tilde{B}_j + (X_iX_j)(B_i\tilde{B}_j + \tilde{B}_iB_j)$

(iii) $C_{ij} = \tilde{B}_i + X_jB_i$

Further reading

A nice conceptual reference on computational complexity is [3] and you can see [1] for a more in-depth discussion of classical computing and its relation to quantum computing. You can refer to [2, Sec. 1.1 - 1.4] and [4, Sec. 2.7] for more on the matrix representation of classical computing and tensor products.

Summary

In this chapter, we learnt that

- A computational problem is modelled mathematically as computing a function from n to m bits, e.g. the problem of factoring integers or finding a satisfying assignment to a Boolean formula.
- The efficiency of an algorithm depends on the computational model used to run it. An algorithm is efficient if its runtime grows as $\mathcal{O}(p(n))$ where $p(n)$ is a polynomial of the input size of the problem n.
- A classical circuit is a model of a classical computer that has wires and gates.
- We can associate one-hot vectors to bit strings and matrices to gates. The states and gates of many bits are described by the tensor product.
- Important reversible classical gates are the NOT gate (also called the X gate), the CNOT gate, and the SWAP gate.

References

[1] A.Y. Kitaev, A. Shen, and M.N. Vyalyi. *Classical and Quantum Computation*. Graduate studies in mathematics. American Mathematical Society, 2002. ISBN: 9780821832295. URL: https://books.google.co.uk/books?id=qYHTvHPvmG8C.

[2] N David Mermin. *Quantum computer science: an introduction*. Cambridge University Press, 2007.

[3] A. Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019. ISBN: 9780691189130. URL: https://books.google.co.uk/books?id=-WCqDwAAQBAJ.

[4] Noson S Yanofsky and Mirco A Mannucci. *Quantum computing for computer scientists*. Cambridge University Press, 2008.