*Chapter 1*

# Classical and probabilistic computation

This chapter presents classical and probabilistic computation via linear algebra in order to pave the way for quantum computation. It begins, in §1.1, with a vague presentation of several computational tasks to preview things to come. Next, in §1.2, two surprising algorithms for the simple task of multiplying integers are described: Napier's book of logs, which dramatically advanced astronomy, and the Schönhage-Strassen algorithm for multiplying polynomials and integers. The Schönhage-Strassen algorithm utilizes the fast Fourier transform (FFT) which will be essential for Shor's quantum factorizing algorithm. In §1.3, we discuss *circuits*, the basic classical model of computation. We take two steps towards quantum computation: we first introduce reversible classical computation in §1.4, because quantum computation must be reversible. Quantum algorithms will be expressed as a sequence of matrix-vector multiplications, so in §1.5, we rephrase classical computation in the language of linear algebra to facilitate comparisons. In §1.6, we discuss probabilistic computation phrased in the language of linear algebra. Finally, in §1.7, we discuss several relevant complexity classes and what is known about them.

## 1.1. Complexity of various tasks

Consider the following tasks and ask yourself how difficult they are to perform:

(1) Multiply two integers.

(2) Compute the determinant of a matrix with integer entries.

(3) Compute the permanent of a matrix with integer entries. (The permanent is the same polynomial as the determinant except that there are no minus signs.)

(4) Given a polynomial in several variables expressed as a circuit (see §1.3), determine if it is the zero polynomial.

(5) Determine if a given integer is prime.

(6) Factor an integer into prime factors.

(7) Simulate a complex system, e.g., weather or a chemical reaction.

How much work is involved in multiplying two integers depends on their size, so we are really asking how the difficulty grows as a function of their size. If we work in base 10 and count addition and multiplication of two single-digit numbers each as unit cost, then the naïve algorithm to compute $mn$ uses $(1+\lceil\log_{10}(n)\rceil)(1+\lceil\log_{10}(m)\rceil)$ single-digit multiplications and a smaller number of additions. Usually in computer science one works in base 2 and counts addition and multiplication in $\mathbb{F}_2$ (i.e., single-digit operations in base 2) as unit cost, so the size of $n$ is the number of digits in base 2, i.e., $\log(n)$, and the naïve algorithm has cost $O(\log(n)\log(m))$. Here and in general, $O(f(n))$ denotes a function bounded above by a constant times some given function $f(n)$ and the same notation is used for functions of several variables. In §1.2 we will see that already for this first task there are several surprises.

The standard formula for computing the determinant of an $n\times n$-matrix has $n!$ terms, so the work involved appears to grow exponentially with the size of the matrix. However, using Gaussian elimination (which amounts to a sequence of matrix multiplications), one can perform the task in $O(n^3)$ arithmetic operations. (One can do even better; see §1.2.4.) The formula for the permanent has the same number of terms, but no significant simplification of its computation is known, and it is conjectured [**Val79**] that there is none. This is a variant, due to L. Valiant, of the millennium prize problem P v. NP.

There is no known polynomial time deterministic algorithm for task (4). However, if one has access to randomness, one can simply evaluate the polynomial at a random point. If one obtains something nonzero, one knows the polynomial is not the zero polynomial. If one obtains zero, then with high probability (probability as close to one as desired if working over an infinite field), one concludes it is the zero polynomial. For more on this problem, see §1.7.2. Thus, at least for some problems, access to randomness may increase computing power. We say "may", as it has not been proven that there is no polynomial time deterministic algorithm. The situation with quantum computation is similar. It is believed that access to a quantum computer would increase computing power, but so far no increase in computing power has been proven.

However, it has been proven that access to *entanglement* (a quantum resource discussed in §2.6 and throughout this book) increases communication power.

Testing if a number is prime at first glance does not seem so different from finding its prime factors. We will see that the state of the art for the two questions is quite different. We have already mentioned Shor's algorithm. We describe the situation for primality in §3.4, where we discuss a famous fast algorithm that utilizes access to randomness, and the current state of the art. Here there are several surprises.

The last problem is not clearly posed, but as mentioned in the preface, it holds the best medium-term promise for quantum computers performing better than classical computers.

## 1.2. Surprising algorithms

**1.2.1. Logarithms: Fast multiplication of numbers.** Until the 1600s, whenever people attempted to make astronomical predictions[1], a difficult step was the multiplication of large numbers. In 1614 John Napier revolutionized computation by writing a book to implement a transform that swaps multiplication for addition: the logarithm. Namely the book simply lists integers and numerical approximations of their logarithms. To multiply two large integers, one looks up their logarithms, adds them, and then looks up the sum in the table of logs and records its exponentiation. (If one wants the exact answer, one must take a sufficiently good approximation that the integer closest to the exponentiation is guaranteed to be the correct one.) Johannes Kepler used Napier's book to make astronomical tables on the order of 30 times more accurate than previous tables [**Gle11**, p. 87].

Even in this old example, there is something modern to learn: if the difficult step of a calculation can be precomputed and stored in a database, it becomes essentially "free".

**Remark 1.2.1.** Exact results may be obtained at the end by rounding to the nearest integer, even though the actual output is approximate. Quantum computing will also have this feature of being approximate and if one wants an exact answer, one must take a sufficiently good approximation so that one obtains an exact result by rounding.

**Exercise 1.2.2.** Pretend you have Napier's book in hand and you want to multiply two four-digit numbers, say 8,764 and 2,366. Use your computer to compute an approximation of their logs in base 10, add the approximate logs, and then raise 10 to that power. How accurate an approximation do you need to get the first four digits of the answer correct? To get the full correct answer?

[1] The king was very interested in knowing auspicious dates for action; see [**Lyo09**].

**1.2.2. The DFT: Fast multiplication of polynomials.** The famous mathematician Kolmogorov conjectured that one could not multiply two $n$-digit numbers using less than $n^2$ one-digit multiplications. His student Karatsuba showed in fact that one can use $O(n^{1.6})$ one-digit multiplications. We now describe an even more efficient algorithm for multiplying numbers due to Schönhage and Strassen [**SS71**] that also may be used to multiply polynomials. We first present the algorithm for polynomials. A variant of this algorithm will be critical to Shor's quantum algorithm for factoring.

Let $a(x), b(x)$ be polynomials of degree at most $d$. Write $a(x) = \sum_{i=0}^{d} a_i x^i$, $b(x) = \sum_{j=0}^{d} b_j x^j$. Write $\bar{a} = (a_0, \ldots, a_d)^{\mathbf{t}}$ and similarly for $\bar{b}$, where here and throughout, $()^{\mathbf{t}}$ denotes transpose. Writing $a(x)b(x) = \sum_{k=0}^{2d} c_k x^k$, one has

$$c_k = \sum_{i+j=k} a_i b_j. \tag{1.2.1}$$

One says $\bar{c}$ is the *convolution* of $\bar{a}$ and $\bar{b}$. To obtain the coefficient vector $\bar{c}$ by this standard method, one needs to perform on the order of $d^2$ arithmetic operations (i.e., $+$'s and $*$'s, where $*$ denotes multiplication). Here we treat each scalar operation as unit cost and will do so for the Schönhage-Strassen algorithm. If the cost of integer multiplication depends on the size of the integer and the sizes of the integers involved are uniformly bounded, then in both algorithms the cost will be multiplied by a constant so the ratios of the costs will be unchanged.

To express this calculation in terms of matrix-vector multiplication, note that the vector $\bar{c}$ is the product of the following $(2d+1) \times (d+1)$-matrix and column vector of height $d+1$:

$$\begin{pmatrix} a_0 & 0 & & \cdots & 0 \\ a_1 & a_0 & 0 & \cdots & 0 \\ a_2 & a_1 & a_0 & \ddots & \\ \vdots & & \vdots & & \\ a_d & a_{d-1} & & \cdots & a_0 \\ 0 & a_d & a_{d-1} & \cdots & a_1 \\ \vdots & & \ddots & & \\ 0 & \cdots & & 0 & a_d \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_d \end{pmatrix}.$$

Here we have broken the symmetry between $a(x)$ and $b(x)$. The symmetry will be restored momentarily.

In our first algorithm the input was transformed in such a way that the computation became easier and then transformed back. Here we will do something similar. We will perform a transformation that re-organizes the input data of

the two polynomials to simplify the calculation. A difference will be that instead of being free, the cost of the transform will be the bulk of the cost of the algorithm.

Up until now, we have not mentioned what the coefficients of the polynomials are: they could be integers, rational numbers, real numbers, or complex numbers. To prepare for what comes next, we will allow complex numbers. Since $\deg(ab) \leq 2d$, instead of working in the space of all polynomials in one variable, denoted $\mathbb{C}[x]$, we can work in a *quotient* of this space, namely the ring $\mathbb{C}[x]/(x^N - 1)$ of polynomials quotiented by the ideal generated by the polynomial $x^N - 1$ for any $N > 2d$. See §A.1 for basic definitions regarding rings and groups and a brief discussion of $\mathbb{C}[x]/(x^N - 1)$. Elements of this ring are equivalence classes of polynomials, where $x^N \equiv 1$ generates the equivalence relation. For the moment, to fix ideas set $N = 2d + 1$, but later we will take $N$ to be a power of two. We can then write a $(2d + 1) \times (2d + 1)$-matrix for $a(x)$ (allowing it now to have degree $2d$) as the matrix

$$
\begin{pmatrix}
a_0 & a_{2d} & a_{2d-1} & \cdots & a_3 & a_2 & a_1 \\
a_1 & a_0 & a_{2d} & \cdots & a_4 & a_3 & a_2 \\
a_2 & a_1 & a_0 & \ddots & & \vdots & \\
\vdots & & \vdots & & & & \\
a_d & a_{d-1} & & \cdots & a_{d+3} & a_{d+2} & a_{d+1} \\
a_{d+1} & a_d & a_{d-1} & \cdots & a_{d+4} & a_{d+3} & a_{d+2} \\
\vdots & & \ddots & & & & \\
a_{2d} & \cdots & & & a_2 & a_1 & a_0
\end{pmatrix} \tag{1.2.2}
$$

and similarly for $b(x)$, although we only need the first column of the product. A matrix whose entries are cyclic permutations of the entries in the first column as in (1.2.2) is called a *circulant* matrix.

If we set $a_{d+1} = \cdots = a_{2d} = 0$, the first $d + 1$ columns of this matrix are our old matrix. This looks like we are making our problem more complicated. However, now that we have a square matrix, call it $A$, we can attempt to *diagonalize* it. The attempt will succeed; see Exercise 1.2.6 below. Call the diagonalized matrix $\hat{A}$. Even better, and to restore symmetry, we will write the corresponding matrix for $b(x)$ and see that $A$ and $B$ are simultaneously diagonalizable.

**Notation.** Throughout this book $i = \sqrt{-1}$, unless it is being used as an index. Write $\omega_N = e^{\frac{2\pi i}{N}}$.

**Definition 1.2.3.** The (unnormalized) *discrete Fourier transform* is the map $\mathrm{DFT}_N : \mathbb{C}^N \to \mathbb{C}^N$ given by the matrix whose $(j, k)$-entry is

$$
(\mathrm{DFT}_N)_{jk} := \omega_N^{jk}. \tag{1.2.3}
$$

The name *discrete Fourier transform* will be explained in Remark 1.2.13.

**Remark 1.2.4.** For reasons explained in Chapter 2, often it is better to deal with the normalized matrix with entries $\frac{1}{\sqrt{N}}\omega_N^{jk}$, as then the associated map is unitary.

The matrix $\mathrm{DFT}_N$ is an example of a *Vandermonde matrix*; see Definition 3.3.9.

The next proposition shows that $\mathrm{DFT}_N$ sends the coefficient vector of a polynomial of degree at most $N-1$ to a vector with entries that are the eigenvalues of the corresponding $N \times N$ circulant matrix (1.2.2).

**Proposition 1.2.5.** *Any $N \times N$ circulant matrix is diagonalizable by conjugating the matrix by the matrix* $\mathrm{DFT}_N$ *of* (1.2.3)*. In particular, all $N \times N$ circulant matrices may be simultaneously diagonalized.*

We present two proofs:

**Pedestrian proof.**

**Exercise 1.2.6.** Show that for all $0 \leq j \leq N-1$, the vector

$$(1, \omega_N^j, \omega_N^{2j}, \ldots, \omega_N^{(N-1)j})^{\mathbf{t}}$$

is an eigenvector of (1.2.2).

The entries of the eigenvector are independent of the $a_s$, so all circulant matrices have the same eigenvectors.

Finally recall that an $N \times N$-matrix $A$ is diagonalizable if and only if it has $N$ linearly independent eigenvectors, and a change of basis matrix is given by an $N \times N$-matrix whose columns are linearly independent eigenvectors of $A$. □

**Geometric proof.** First observe that $a(x)b(x) = b(x)a(x)$, in both the usual multiplication of polynomials and as elements of $\mathbb{C}[x]/(x^N-1)$. The map

$$\mathbb{C}[x]/(x^N-1) \to \mathrm{Mat}_{N\times N},$$

where $a(x)$ is sent to the matrix (1.2.2) and $\mathrm{Mat}_{N\times N}$ denotes the ring of $N \times N$-matrices (endomorphisms of $\mathbb{C}^N$), is a ring homomorphism, which ensures the corresponding matrices will commute. The image is exactly the space of circulant matrices. Since the algebra $\mathbb{C}[x]/(x^N-1)$ is generated by $x$, it will be sufficient to prove the result for the matrix corresponding to multiplication by $x$. Thus it will be sufficient to show that for each $k$, the $k$-th column vector of (1.2.3) is an eigenvector for multiplication by $x$ in the ring $\mathbb{C}[x]/(x^N-1)$ with eigenvalue $\omega_N^{-k}$. To see this consider (the equivalence class of)

$$x(1 + \omega_N^j x + \omega_N^{2j} x^2 + \cdots + \omega_N^{(N-1)j} x^{N-1})$$

and notice that $\omega_N^{(N-1)j}(x^N) \equiv \omega_N^{-j}(1)$. □

Given $\hat{A}, \hat{B}$, to compute the matrix $\hat{C} := \hat{A}\hat{B}$, we only need to perform $N = O(d)$ scalar multiplications instead of $O(d^3)$.

**Exercise 1.2.7.** Verify that if $A, B$ are diagonalized by $P$, then $C = P^{-1}\hat{C}P$ is indeed the matrix corresponding to $\bar{c}$.

**Exercise 1.2.8.** Show that if two diagonalizable matrices commute, then they are simultaneously diagonalizable. ⊙

**Exercise 1.2.9.** Is the same true if one has three diagonalizable commuting matrices?

However, diagonalizing seems like a bad idea: the cost of a change of basis is worse than $O(d^2)$. Moreover, we will have to un-diagonalize $\hat{C}$ to get the coefficients of $c(x)$ (which are the elements of the first column of $C$). The punch line will be that for the highly structured matrices in our situation there exists a relatively cheap way to carry out the diagonalization.

Write $\hat{a} = \mathrm{DFT}_N \bar{a}$ (where we have padded the coefficient vector of $a(x)$ with zeros to make it have length $N$), and similarly $\hat{b} = \mathrm{DFT}_N \bar{b}$. Given $\hat{a}$ and $\hat{b}$, the vector $\hat{c}$ can be computed using $N$ entrywise scalar multiplications as $\hat{c}_k = \hat{a}_k \hat{b}_k$. Finally $\bar{c} = {\mathrm{DFT}_N}^{-1} \hat{c}$.

**Remark 1.2.10.** For those familiar with the representation theory of finite groups, the discrete Fourier transform is the change of basis matrix from the standard basis of the regular functions on the cyclic group of order $N$ to the character basis. The roots of unity appearing in the DFT matrix are the characters of the cyclic group. Using representation theory, it is straightforward to derive the DFT matrix.

Now we come to a great discovery of Gauss in 1810 [**Gau**], which was rediscovered by several people, including Danielson and Lanczos in 1942 [**DL42**] (the first published version of the discovery that we are aware of) and Cooley and Tukey in 1965 [**CT65**] (which led to its modern implementation that revolutionized signal processing): the DFT matrix factors as a product of sparse matrices. More precisely, if the matrices are $N \times N$ and $N$ is a power of two, there will be $2\lceil \log(N) \rceil$ matrices, each of which has at most $2N$ nonzero entries. The factorization is called the *fast Fourier transform* (FFT).

To get intuition as to why the DFT matrix might admit such a factorization, assume $N$ is even. The columns labeled by even values of $j$ only have even powers of $\omega_N$ appearing and consist of a doubling of the matrix for $\mathrm{DFT}_{N/2}$ with the orders re-arranged. Similarly, the entries columns indexed by odd $j$ are almost a doubling of the matrix for $\mathrm{DFT}_{N/2}$, except that in addition to the re-arrangement, the $j$-th column is also multiplied by $\omega_N^j$. This is exactly what will be exploited.

Let $\{|j\rangle \mid 0 \leq j \leq N-1\}$ denote the standard basis of $\mathbb{C}^N$. Assume $N$ is a power of 2. We have

$$\begin{aligned}
\mathrm{DFT}_N(|j\rangle) &= \sum_{k=0}^{N-1} \omega_N^{jk}|k\rangle \\
&= \sum_{k:\,\text{even}} \omega_N^{jk}|k\rangle + \sum_{k:\,\text{odd}} \omega_N^{jk}|k\rangle \\
&= \sum_{\ell=0}^{\frac{N}{2}-1} \omega_N^{2j\ell}|2\ell\rangle + \sum_{\ell=0}^{\frac{N}{2}-1} \omega_N^{j(2\ell+1)}|2\ell+1\rangle \\
&= \sum_{\ell=0}^{\frac{N}{2}-1} \omega_{N/2}^{j\ell}|2\ell\rangle + \omega_N^j \sum_{\ell=0}^{\frac{N}{2}-1} \omega_{N/2}^{j\ell}|2\ell+1\rangle.
\end{aligned}$$

We can thus compute $\mathrm{DFT}_N$ by twice applying the $\mathrm{DFT}_{N/2}$ matrix and multiplying by several sparse matrices. Let $\Delta_N$ be the diagonal matrix with entries $(1, \omega_N, \omega_N^2, \ldots, \omega_N^{\frac{N}{2}-1})$, so $-\Delta_N$ is the diagonal matrix with entries $(\omega_N^{\frac{N}{2}}, \ldots, \omega_N^{N-1})$. Let $\Pi_N$ denote the permutation matrix corresponding to the permutation $(0, 1, \ldots, N-1) \mapsto (0, 2, 4, \ldots, N-2, 1, 3, 5, \ldots, N-1)$. We have the matrix equation

$$\mathrm{DFT}_N = \begin{pmatrix} \mathrm{Id}_{N/2} & \Delta_N \\ \mathrm{Id}_{N/2} & -\Delta_N \end{pmatrix} \begin{pmatrix} \mathrm{DFT}_{N/2} & \\ & \mathrm{DFT}_{N/2} \end{pmatrix} \Pi_N. \tag{1.2.4}$$

For example,

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_2 & -\omega_4 \\ 1 & \omega_2 & 1 & \omega_2 \\ 1 & -\omega_4 & \omega_2 & \omega_4 \end{pmatrix},$$

while, when $N = 4$,

$$\begin{aligned}
&\begin{pmatrix} \mathrm{Id}_{N/2} & \Delta_N \\ \mathrm{Id}_{N/2} & -\Delta_N \end{pmatrix} \begin{pmatrix} \mathrm{DFT}_{N/2} & \\ & \mathrm{DFT}_{N/2} \end{pmatrix} \Pi_N \\
&\quad = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & \omega_4 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -\omega_4 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & \omega_2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & \omega_2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&\quad = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & -1 & -\omega_4 \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4 & -1 & \omega_4 \end{pmatrix},
\end{aligned}$$

where to get the last line and see the equality of the two expressions, recall that $\omega_2 = -1$.

**Exercise 1.2.11.** Verify (1.2.4) directly for $\mathrm{DFT}_6$.

If we iterate (1.2.4), replacing the instances of $\mathrm{DFT}_{N/2}$ by three block matrices, then replacing the instances of $\mathrm{DFT}_{N/4}$ by three block matrices, etc., when $N = 2^k$ we obtain a product of $k$ matrices, each with at most $2^{k+1}$ nonzero entries.

The cost of matrix-vector multiplication of an $N \times N$-matrix with at most $\ell$ nonzero entries in each row is $O(\ell N)$. Our sparse matrix has 2 nonzero entries in each row (and column), so the cost of performing our DFT is $O(N \log(N))$ instead of $O(N^2)$. The inverse of the DFT has the same cost by inverting the factorization.

We conclude:

**Theorem 1.2.12** ([**SS71**], Schönhage-Strassen)**.** *Assign multiplication of complex numbers unit cost. Then two polynomials of degree at most $d$ may be multiplied with cost $O(d \log(d))$.*

**Remark 1.2.13.** You may have seen Fourier transforms of periodic functions, where convolution in the original space corresponds to multiplication in the transform space. This is the analogous transform when the group is the circle. Explicitly, write the unit circle in $\mathbb{R}^2$ as

$$S^1 = \{(\cos(\theta), \sin(\theta)) \mid \theta \in [0, 2\pi)\} \subset \mathbb{R}^2.$$

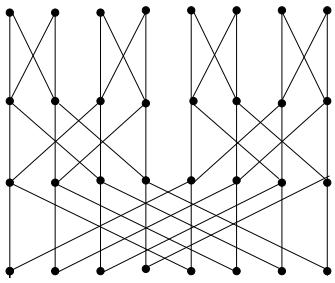

**Figure 1.2.1.** Sometimes the FFT is illustrated with this picture. The graph represents the product of three $8 \times 8$-matrices that gives $\mathrm{DFT}_8$. One travels from the top to the bottom. Vertices in each row represent indices from 1 to 8, and edge from $i$ to $j$ at level $k \in \{1, 2, 3\}$ means the $(i, j)$-th entry of the $k$-th matrix is nonzero.

Introduce complex notation $\mathbb{C} = \mathbb{R}^2$, so $S^1 = \{e^{i\theta} \mid \theta \in [0, 2\pi)\} \subset \mathbb{C}$. Then for (e.g., continuous) functions $f(\theta)$ on the unit circle, we may write

$$f(\theta) = \sum_{n=-\infty}^{\infty} c_n e^{\frac{in\theta}{2}}, \quad \text{where } c_n = \frac{1}{2\pi}\int_0^{2\pi} f(\theta)e^{-\frac{in\theta}{2}}\,d\theta.$$

Since nonzero complex numbers form a group under multiplication and since the product of elements of length one is of length one, $S^1$ is naturally a group. In signal processing, one needs to digitize (e.g., sound waves), so one approximates a periodic function by sampling it at, say, $N$ equally spaced points on the circle, e.g., the points $e^{\frac{k2\pi i}{N}}, 0 \leq k \leq N-1$. These points form a subgroup, the cyclic group of order $N$. Tracing through the calculation, the DFT is the discretization of the Fourier transform on the circle, exactly what one needs in signal processing.

**Aside 1.2.14.** For those familiar with tensors and their ranks, the structure tensor of the algebra $\mathcal{A} = \mathbb{C}[x]/(x^N - 1)$, which may be written

$$T_{\mathcal{A}} = \sum_{k,j=0}^{N-1} x^k \otimes x^j \otimes e_{k+j \bmod N},$$

where $e_j$ is the dual basis vector to $x^j$, has minimal tensor rank $N$, and the DFT is a change of basis that rewrites the structure tensor $T_{\mathcal{A}} \in \mathcal{A}^* \otimes \mathcal{A}^* \otimes \mathcal{A}$ as a sum of $N$ rank one tensors.

**Aside 1.2.15.** One might wonder if there is an even more efficient way of computing the operation $\bar{a} \mapsto \mathrm{DFT}_N \bar{a}$. This question was posed by L. Valiant in [**Val77**], where he also gave a path to resolving it (he conjectured that no significant improvement is possible). There is interesting algebraic geometry related to the question; see [**KLPSMN09**, **GHIL16**].

**1.2.3. Fast integer multiplication.** Assign multiplication and addition of single-digit integers unit cost. (We can work base in 10 or base 2; the comparison between algorithms will be the same.) If we want to multiply $m * n$ which, respectively, have $d_m, d_n$ digits, in the naïve algorithm we need to perform $d_m d_n$ single-digit multiplications and a smaller number of additions, so the complexity of multiplying two $d$-digit numbers is $O(d^2)$.

If we apply the Schönhage-Strassen algorithm with $x$ replaced by 10 if we work in base 10 or by 2 if we work in base 2, at first it looks like we can get away with $O(d\log(d))$ operations, but in intermediate steps one needs to work with complex numbers $a + bi$ where $a, b$ are not necessarily integers. One fix is to use the FFT instead on a cyclic group of prime order for a prime of the form $2^m + 1$. Another is to represent the complex numbers by sufficiently good approximations. The actual cost using the algorithm is $O(d\log(d)\log(\log(d)))$.

Schönhage-Strassen conjectured that $O(d\log(d))$ cost was possible, and over 50 years later this was proved in [**HvdH21**]. This $O(d\log(d))$ cost was conjectured to be optimal and this question is still open.

For an in-depth discussion on the use of the FFT for integer and polynomial multiplication, see [**vzGG13**].

**1.2.4. Matrix multiplication.** Another surprising algorithm deals with matrix multiplication. The usual algorithm for multiplying two $n \times n$-matrices uses $O(n^3)$ arithmetic operations. Strassen [**Str69**] discovered an algorithm that uses $O(n^{2.81})$ arithmetic operations, and it has been conjectured that as $n$ grows, it becomes nearly as easy to multiply matrices as it is to add them. That is, for any $\epsilon > 0$, one can multiply matrices using $O(n^{2+\epsilon})$ arithmetic operations. By 1988 it had been shown that $O(n^{2.38})$ is possible [**CW90**], but since then progress has been limited to an improvement by 0.004 [**Sto**, **Wil**, **LG14**, **AW21**, **VXXZ23**].

## 1.3. Classical computation via circuits

Classical complexity works in binary: one deals with strings of 0's and 1's. An element of the set $\{0,1\}$ is called a *bit*: it can encode "one bit" of information. It requires $1+\lceil\log(N)\rceil$ bits to express a positive integer $N$ in binary.

**Definition 1.3.1.** A *Boolean function* is a map $f : \mathbb{F}_2^n \to \mathbb{F}_2$, or more generally a map $\mathbb{F}_2^n \to \mathbb{F}_2^m$.

We agree on some basic Boolean functions whose complexity is designated as having unit cost, e.g.,

- addition $\oplus$ (also called *XOR*) where $a \oplus b$ is addition in $\mathbb{F}_2$, i.e., $0\oplus 0 = 1\oplus 1 = 0$ and $0\oplus 1 = 1\oplus 0 = 1$,
- $\neg$ (NOT) negation, which swaps 0 and 1,
- (OR) $a \vee b$ where $0 \vee 0 = 0$ and all other $a \vee b = 1$,
- (AND = multiplication in $\mathbb{F}_2$) $a \wedge b = ab$, where $1 \wedge 1 = 1$ and all other $a \wedge b = 0$.

Such a collection is called a (*logic*) *gate set*.

**Definition 1.3.2.** A (*Boolean*) *circuit C* is a finite, directed, acyclic graph with a collection of vertices of in-degree 0, a collection of vertices of out-degree 0, and other vertices that have positive in-degrees and out-degrees. The vertices of in-degree 0 are labeled by elements of $\mathbb{F} \cup \{x_1, ..., x_n\}$, where the $x_i$ are variables, and are called *inputs*. Those of positive in-degree are labeled with Boolean functions from a gate set and are called *gates*. If the out-degree of $v$ is 0, then $v$ is called an *output gate*.

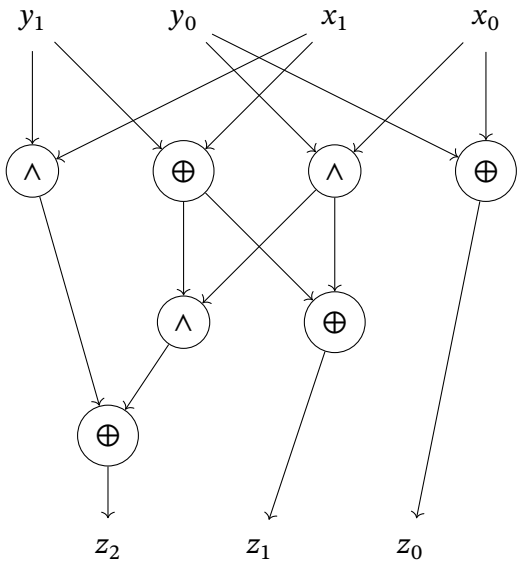


**Figure 1.3.1.** Circuit for $z = y + x$, where $y = y_1 2^1 + y_0 2^0$, etc. Output legs are the three digits of $z = z_2 2^2 + z_1 2^1 + z_0 2^0$.

**Remark 1.3.3.** See [**Fey01**, Chap. 2] for a discussion of how physical computers carry out Boolean operations.

Figure 1.3.1 depicts a Boolean circuit for the addition of two 2-digit (in binary) numbers.

**Definition 1.3.4.** A gate set is a *universal gate set* if any Boolean function can be computed with a circuit whose vertices are labeled with gate set elements.[2]

**Proposition 1.3.5.** *The gate set* $\{\neg, \wedge\}$ *is universal.*

**Exercise 1.3.6.** Give an explicit expression for $x \vee y$ and $x \oplus y$ in terms of the gate set $\{\neg, \wedge\}$.

**Exercise 1.3.7.** Prove Proposition 1.3.5 for Boolean functions that take the value 1 on exactly one input. ⊙

**Exercise 1.3.8.** Prove Proposition 1.3.5 in general. ⊙

**Exercise 1.3.9.** Write a Boolean circuit that computes $x * y$ when $x, y$ are two digits in binary.

**Exercise 1.3.10.** What is the precise number of $\oplus$ and $\vee$ needed for a circuit to compute the product of integers $M$ and $N$ with the naïve algorithm?

[2] In some of the literature a gate set is sometimes called a "basis" (despite being unrelated to bases of vector spaces), and a universal gate set is called a "complete basis".

Using the naïve algorithm, we may compute $M^k$ using a circuit of size $O(\log(M)^k)$.

**Proposition 1.3.11.** *If $a, b$ are natural numbers with $a > b$, then the division of $a$ by $b$ and its remainder may be computed with a circuit of size $O(\log(a)^2)$.*

**Proof.** Express $a$ and $b$ in binary. Elementary long division requires one to compare the size of $b$ and various numbers with $\log(b)$ digits, which costs $O(\log(b))$ each time. After the comparison, one may need to perform a subtraction, which also costs $O(\log(b))$, and there are at most $O(\log(a))$ steps. At the end, one obtains both $\lfloor \frac{a}{b} \rfloor$ and the remainder. □

**Exercise 1.3.12.** Say $M < N$ and instead of wanting to compute $M^k$, we are only interested in its remainder after division by $N$, i.e., to compute $M^k \bmod N$. Show that this can be computed with cost $O(k \log(N)^2)$ (without using Schönhage-Strassen). Moreover, if $k = 2^\ell$, then one can do the computation with cost $O(\ell \log(N)^2)$. ⊛

## 1.4. Reversible classical computation

We will see that the gates of a quantum circuit (other than the measurements) must be *reversible*; i.e., there must be another gate that takes the output of a gate and produces its input. In this section we show how classical computation may be made reversible.

In reversible computation, one deals exclusively with invertible maps $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$, even if the result of interest is just the first entry of the output.

Of the gates we saw, NOT is clearly reversible as $\neg\neg x = x$. At the cost of using an extra bit, one can make addition and multiplication reversible as follows: consider the following gate, called the *Toffoli gate* Tof : $\mathbb{F}_2^3 \to \mathbb{F}_2^3$:

$$|x, y, z\rangle \mapsto |x, y, z \oplus (x \wedge y)\rangle. \tag{1.4.1}$$

If we send in $|x, y, 0\rangle$, we obtain $x \wedge y$ in the third entry, and if we send in $|x, 1, z\rangle$, we obtain $x \oplus z$. In the literature, the entries are often called *registers*, and we henceforth adopt this terminology.

**Exercise 1.4.1.** Show that Tof $\circ$ Tof $=$ Id, so Tof is indeed reversible.

**Exercise 1.4.2.** The *reversible controlled not* gate CNOT takes a pair $x, y$ and if $x = 0$, it sends $y$ to $y$, and if $x = 1$, it sends $y$ to $\neg y$, and in both cases it sends $x$ to itself. Write down CNOT as a reversible gate using our gate set and verify that it is reversible

The gate set {Tof, $\neg$} is universal and reversible, so there is no loss in computing power if one restricts to reversible classical computation.

**Exercise 1.4.3.** Show that in fact {Tof} alone is universal.

Implementing $\oplus$ or $\wedge$ with the Toffoli gate requires "workspace bits", which are bits that are not part of the input or output of the function but are needed to carry out the computation. Usually they are 0 at the beginning of a computation. To execute a Boolean function $\mathbb{F}_2^m \to \mathbb{F}_2^n$ in a Boolean circuit, one needs $\max\{m, n\}$ bits to carry it out, but with reversible computation, in general more bits are needed.

**Remark 1.4.4.** As early as 1961, researchers were concerned that as computers got more powerful, they would generate too much heat from erasing bits; see, e.g., [**Lan61**] and [**Ben03**]. One way to solve this problem would be to have reversible computation. Thus one could argue for reversible computation independent of quantum computation. See [**Fey01**, Chap. 5] for a discussion of how one may make the heat generated by a computation arbitrarily small using reversible classical computation.

## 1.5. Classical computation via linear algebra

To facilitate the comparison with quantum computing, we will encode classical computation as a sequence of matrix-vector multiplications (following [**AB09**, Exercise 10.4]). Readers not familiar with tensors should read section §A.2 before reading this section. Fix a field $\mathbb{F}$. For classical computation we could take $\mathbb{F} = \mathbb{F}_2$, but to facilitate the comparison with probabilistic computation we will take $\mathbb{F} = \mathbb{R}$. For quantum computation we will take $\mathbb{F} = \mathbb{C}$.

**1.5.1. Induced bases in tensor products of spaces.** Give $\mathbb{R}^2$ the basis $|0\rangle, |1\rangle$, which induces the basis $|i\rangle\otimes|j\rangle$, $i, j \in \{0, 1\}$ of $(\mathbb{R}^2)^{\otimes 2}$. Introduce the notation $I := (i_1, \dots, i_N)$ with $i_\alpha \in \{0, 1\}$. The basis of $\mathbb{R}^2$ also induces a basis of $(\mathbb{R}^2)^{\otimes N}$:

$$|I\rangle := |i_1\rangle\otimes\cdots\otimes|i_N\rangle, \quad i_\alpha \in \{0, 1\}, \quad 1 \leq \alpha \leq N.$$

If our bit string is 00101100, we represent it by the vector $|00101100\rangle \in \mathbb{R}^{2^8}$.

Now $(\mathbb{R}^2)^{\otimes N}$ is isomorphic as a vector space to $\mathbb{R}^{2^N}$, and sometimes it is also convenient to represent elements of $(\mathbb{R}^2)^{\otimes N}$ as column vectors, forgetting the tensor product structure. There is no canonical change of basis from the basis $|I\rangle$ to the standard basis of column vectors. We will use the lexicographic order to obtain a change of basis, so the vector

$$a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle \in \mathbb{R}^2\otimes\mathbb{R}^2 \simeq \mathbb{R}^4$$

will be mapped to the column vector

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}.$$

The linear map $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \mathrm{Id}_2$, which applies NOT on the first $\mathbb{R}^2$ and is the identity on the second $\mathbb{R}^2$, may be expressed as the $4 \times 4$-matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

**Exercise 1.5.1.** Consider $\mathbb{C}^N \otimes \mathbb{C}^M$ with basis $|js\rangle$, $0 \leq j,k \leq N-1$, $0 \leq s,t \leq M-1$ and where each index pair $(js)$ is viewed as an element of $G := \mathbb{Z}/N\mathbb{Z} \times \mathbb{Z}/M\mathbb{Z}$. If we encode the multiplication in $G$ via matrix-vector multiplication or matrix-matrix multiplication as in §1.2.2, we get a complicated matrix. However, here we may also perform the DFT for $G$ to get a diagonal matrix. What is the linear map $\mathrm{DFT}_G$ that executes this discrete Fourier transform? ⊙

**Exercise 1.5.2.** If $G = (\mathbb{Z}/2\mathbb{Z})^k$, what is the linear map/matrix $\mathrm{DFT}_G : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes k}$ that executes this transform? ⊙

**1.5.2. Reversible classical computation via linear algebra.** Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a Boolean function. We phrase the computation $x \mapsto f(x)$ as a sequence of restricted linear operations on a vector space $(\mathbb{R}^2)^{\otimes N}$, where $N = \max\{n,m\} + s$, where $s$ is the number of workspace bits needed. The restrictions will be the following:

(1) Each matrix must be invertible and take vectors representing sequences of bits, i.e., basis vectors, to basis vectors. Such matrices in $\mathrm{End}(\mathbb{R}^{2^n})$ are the *permutation matrices*. That is, each row and column has exactly one entry equal to one and all other entries zero.

(2) In order to deal with finite gate sets, require that each matrix only alters a small number of entries. Since we have a universal gate set that uses at most three entries at a time, we will assume each matrix acts on at most some $(\mathbb{R}^2)^{\otimes 3}$ and is the identity on all other factors in the tensor product.

Each map will realize some Boolean gate. For example, say we want to realize the Toffoli gate

$$|x,y,z\rangle \mapsto |x,y,z \oplus (x \wedge y)\rangle.$$

In the lexicographical basis $|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle$, of $\mathbb{R}^8$, the matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.5.1}$$

Call this matrix the *Toffoli matrix*.

The negation gate $\neg$ is obtained with the linear map $\mathbb{R}^2 \to \mathbb{R}^2$ given by the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

**Exercise 1.5.3.** Write the $8 \times 8$-matrices for $(x, y, z) \mapsto (x, y, z \oplus (x \oplus y))$ and $(x, y, z) \mapsto (x, y, z \oplus (x \vee y))$.

The computation of a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ via a reversible classical circuit has as its input the basis vector $|x, I\rangle \in \mathbb{R}^{2^{n+s}}$, where $x$ is the input and $I$ some initialization of the workspace bits that does not depend on $x$, usually just a string of zeros, and the output is of the form $|y, J\rangle$ where $y \in \mathbb{F}_2^m$ is the answer (and $J$ will in general depend on $x$).

Since the gate set used is universal, any Boolean function may be computed by a reversible classical circuit. In other words, all permutations on the first $n$ symbols may be obtained from products of Toffoli matrices. An important question discussed in §1.7 asks which functions may be computed *efficiently* with reversible classical computation.

## 1.6. Probabilistic computation via linear algebra

We now explain how to incorporate the additional resource of randomness in computation. We continue to use the language of linear algebra to express the computation.

Readers not familiar with basic definitions and notation from probability should now read §B.1.

Consider a set injection $\{0,1\}^m \to \mathbb{R}^{2^m}$, via the map $I \mapsto |I\rangle$, where $I$ is a bit string of length $m$. A probability distribution on $\{0,1\}^m$ may be encoded as a vector in $\mathbb{R}^{2^m}$: give $\mathbb{R}^2$ basis $|0\rangle, |1\rangle$ and $(\mathbb{R}^2)^{\otimes m} = \mathbb{R}^{2^m}$ basis $|I\rangle$ where $I \in \{0,1\}^m$. Assign to the distribution $p_I$ on $\{0,1\}^m$ (where $I \in \{0,1\}^m$) the vector $v = \sum_I p_I |I\rangle \in \mathbb{R}^{2^m}$.

To prepare for the analogy with quantum computation, define a *pbit*, i.e., a probabilistic bit, to be an element of the set

$$\{p_0|0\rangle + p_1|1\rangle \mid p_j \in [0,1] \text{ and } p_0 + p_1 = 1\} \subset \mathbb{R}^2.$$

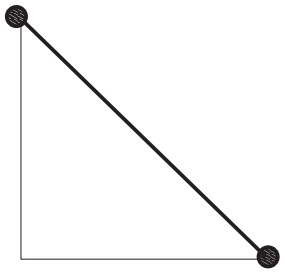

**Figure 1.6.1.** The set of pbits

In other words, the set of pbits is the set of probability distributions on $\{0,1\}$. The set of pbits is a convex set, and the basis vectors, i.e., the classical bits, are the extremal points of this convex set.

For probabilistic computation via linear algebra we require the following:

(1) Each linear map must take vectors representing probability distributions to vectors representing probability distributions. This implies the matrices are *stochastic*: the entries are nonnegative and the entries in each column sum to 1.

(2) In order to deal with finite gate sets, require that each linear map only alters a small number of entries. For simplicity assume it alters at most three entries; i.e., it acts on at most $(\mathbb{R}^2)^{\otimes 3}$ and is the identity on all other factors in the tensor product.

**Exercise 1.6.1.** Verify that the stochastic matrices are exactly those matrices that take probability distributions to probability distributions.

Since permutation matrices are stochastic, probabilistic computation includes classical computation.

To represent a fair coin flip in terms of linear algebra, introduce the matrix

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}.$$

Classical probabilistic computation cannot be made reversible, indicated by the fact that the coin flip matrix is not invertible.

For a probabilistic gate set one could take the coin flip and Toffoli matrices.

With this gate set, the matrices will actually be *doubly stochastic*: in addition to stochastic, the entries in each row will also sum to 1. The set of doubly stochastic matrices has a well-studied structure that we discuss in detail in §6.5.1.

Consider the case of a probabilistic algorithm where probability distributions with denominators that are not powers of two appear. When this probabilistic algorithm is expressed in our gate set, the computation will be *approximate*. An analogous phenomenon occurs for quantum computation.

**Exercise 1.6.2.** In probabilistic algorithms, we will want to choose an element uniformly at random from a set of $M$ elements. How can we approximately realize this choice with our probabilistic gate set with error at most some fixed $\epsilon > 0$? (We can realize it exactly when $M$ is a power of two.) ⊛

Thus with our probabilistic gate set, we can approximate any probability distribution arbitrarily closely by elements of the gate set and can perform all Boolean operations.

**Summary of probabilistic computation via linear algebra:** A probabilistic computation of $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, expressed in terms of linear algebra, starts with $|x0^{r+s}\rangle$ (or $|x0^{m-n+r+s}\rangle$ if $m > n$) where $x \in \mathbb{F}_2^n$ is the input. A sequence of elements from the gate set is applied to it, resulting in a vector $|v\rangle = \sum p_I |I\rangle$ that encodes a probability distribution on $\{0,1\}^{\max\{n,m\}+s+r}$. Then one takes the marginal distribution on the first $m$ copies of $\mathbb{R}^2$: write $I = (i_1, \ldots, i_{n+r+s}) = (J, L)$, where $J = (i_1, \ldots, i_m)$, to obtain

$$\sum_J \sum_L p_{J,L} |J\rangle \in \mathbb{R}^{2^m}.$$

Here $r$ is the number of coin flips needed and $s$ is the number of workspace bits needed. The algorithm then outputs $J \in \{0,1\}^m$ with probability $\sum_L p_{J,L}$ using some external device that chooses $J$ with probability $\sum_L p_{J,L}$.

In contrast with classical computation, we may not arrive at an arbitrary stochastic matrix via products of matrices corresponding to elements in our gate set. We may only approximate any such.

## 1.7. Brief overview of classical, probabilistic, and quantum complexity classes

**1.7.1. P/poly, P, and NP.** We said that factorization is not known to have an efficient algorithm. We now make the notion of efficient precise.

Fix a universal gate set $G$. A natural complexity measure for a Boolean function is the minimal size of a Boolean circuit that computes it. Let $F_n : \mathbb{F}_2^n \to \mathbb{F}_2$ be a sequence of functions $(F_n)$. Consider the growth with $n$ of the minimal size of a circuit needed to compute $F_n$. If it grows like a polynomial,

the sequence $(F_n)$ is said to be in the class **P**/poly with respect to $G$. One says that sequences in **P**/poly have *efficient* algorithms.

**Exercise 1.7.1.** Show that membership in **P**/poly with respect to $G$ is independent of the choice of the finite universal gate set $G$.

Thus one just says that the sequence $(F_n)$ is in the class **P**/poly.

The complexity class **P** is the standard model for feasible computations, and it is unfortunate that **P**/poly, with a simple description, has a different definition than **P**. The definition of **P**/poly is less restrictive, but not so much less, and it can sometimes be used as a substitute for **P**; see [**AB09**, Chap. 6]. It is known that $\mathbf{P} \subsetneq \mathbf{P}/\text{poly}$; however, membership in **P**/poly for many problems of interest is not known.

The class **P** is usually defined in terms of a different model of computation, namely *Turing machines*. We will avoid defining them and assume the reader has at least a passing familiarity with them. A sequence $(F_n)$ is in **P** if it is in **P**/poly and there exists a Turing Machine TM such that the circuits $C_n$ computing $F_n$ are constructed by TM in time poly$(n)$; see [**KSV02**, Thm. 2.3]. Sometimes one takes the more restrictive definition of *efficient* that requires membership in **P**.

The complexity class **NP** essentially consists of problems whose proposed solutions can be verified quickly, i.e., in polynomial time. A famous problem in **NP** is the traveling salesman problem: if someone claims to have a route to visit 30 cities traveling less than 2,000 miles, it is easy to verify the claim by examining the route, but the only known way of *finding* such a route is essentially by a brute-force search. Another problem in **NP** is "SAT": one is handed a Boolean circuit and wants to know if it ever outputs 1. (If it does, to convince you it does, someone just needs to hand you an input that works, and you can quickly check if it outputs 1.) SAT and the traveling salesman problems are **NP**-*complete*, which means one could define **NP** to be the collection of problems that can be *reduced* (in polynomial time) to SAT; see, e.g., [**AB09**, Chap. 2]. In other words, there exists a polynomial time algorithm for SAT if and only if $\mathbf{P} = \mathbf{NP}$.

A problem like factoring asks for something beyond a $\{0,1\}$ output and the corresponding complexity classes have slightly different names, but since we will not be discussing them further and since the corresponding issues are the same, we leave it to the curious reader to work through the zoo of complexity classes; see, e.g., `https://complexityzoo.net/Complexity_Zoo`.

**1.7.2. BPP.** It might increase our computational power if we exploit randomness. (Assuming we have a method to generate random numbers — more on this in Remark 4.5.14.) For example, if someone hands you a complicated expression for a polynomial, it can be very difficult to determine if the polynomial

is just the zero polynomial in disguise. If we test the polynomial at a point and its evaluation is nonzero, then we know it is not the zero polynomial. If it does evaluate to zero, then we have no information. For a polynomial of degree $d$ in one variable, it is sufficient to test $d+1$ distinct points, but as the number of variables grows, the number of points one needs to check grows exponentially. However, if we are allowed to test at a random point and it evaluates to zero, then with high probability the polynomial is the zero polynomial. Even over finite fields, we can make this probability as high as we want by testing on several random points.

These observations motivate the class **BPP** (short for "bounded-error probabilistic polynomial time"), where one works with a Turing machine with access to randomness, and instead of asking for a correct yes or no answer on any input in polynomial time, one asks for a correct yes/no answer with probability strictly greater than $\frac{2}{3}$ on any input in polynomial time. (Thus if one runs the program enough times, one can get a correct answer on any input with probability as high as one wants.) See, e.g., [**KSV02**, §1.4] for the precise definition.

**Remark 1.7.2.** It is subtle to know if a given sequence of numbers is random. For example, the first digit of the number $2^n$ is far from a random element of $\{1,\dots,9\}$; see [**Ad89**, §16, Ex. 4]. Fortunately, for most situations, *pseudo-random* numbers suffice; see [**AB09**, §9.2.3].

**Aside 1.7.3.** If we are given certain types of additional information about a set of polynomials that we want to test, then one can test if a polynomial in the set is zero by testing a reasonable number of points. This subject, called PIT (polynomial identity testing), is an active area of research; see [**AB09**, §7.2.3]. For a geometric perspective see [**Lan17**, §7.7].

**1.7.3. What is known.** $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{P}/\mathrm{Poly} = \mathbf{BPP}/\mathrm{Poly}$.

The inclusion $\mathbf{BPP} \subseteq \mathbf{P}/\mathrm{Poly}$ is Adelman's theorem [**Adl78**]. The key observation is that "off-line" computations are not counted in the complexity assessment for $\mathbf{P}/\mathrm{Poly}$. So one can create, for any given $n$, a library of "random" $a$'s to test on.

Does randomness really help? At the moment, we don't know. See [**AB09**, Chap. 20] for a discussion, where they state (p. 126) "most complexity theorists actually believe that $\mathbf{BPP} = \mathbf{P}$."

**1.7.4. BQP.** We are not yet in a position to define the class **BQP**. It will be the quantum analog of **BPP**, the decision problems that can be solved efficiently, with high probability, on a quantum computer. Pbits will be replaced by *qubits*, which are unit vectors in $\mathbb{C}^2$ subject to an equivalence relation. The matrices will be allowed to have complex entries; they will be required to be *unitary* (see

§2.1) instead of stochastic. There is no generally used analog of **P** for a quantum computer as in general answers have a nonzero probability of being incorrect.

In a similar manner to how a finite probabilistic gate set can only approximate arbitrary distributions, a finite "universal quantum gate set" will only be able to approximate any unitary map arbitrarily closely by elements of the gate set.

**1.7.5. The Church-Turing theses.** The Church-Turing thesis (made explicitly by Church in [**Chu36**]) is:

*Any algorithm can be realized by a Turing machine.*

So far there has been no challenge to this thesis — e.g., any computation that can be done on a quantum computer can be done with a sufficiently large Turing machine. (Since we do not have a precise definition of what is allowable as an "algorithm", it would be impossible to affirmatively prove the thesis.)

The quantitative (sometimes called *strong*) Church-Turing thesis [**VSD86**] is:

*Any algorithmic process can be simulated efficiently by a Turing machine*

or

*Any algorithmic process can be simulated efficiently by a probabilistic Turing machine.*

If one believes factoring is not in **P**, then Shor's algorithm challenges this thesis. On the other hand, there are experts who think that factoring could be in **P**, because unlike, say, SAT or the traveling salesman problem, the problem is highly structured.