A MATHEMATICAL BACKGROUND

A.1 COMPLEXITY ANALYSIS AND O() NOTATION

Computer scientists are often faced with the task of comparing algorithms to see how fast they run or how much memory they require. There are two approaches to this task. The first is **benchmarking**—running the algorithms on a computer and measuring speed in seconds and memory consumption in bytes. Ultimately, this is what really matters, but a benchmark can be unsatisfactory because it is so specific: it measures the performance of a particular program written in a particular language, running on a particular computer, with a particular compiler and particular input data. From the single result that the benchmark provides, it can be difficult to predict how well the algorithm would do on a different compiler, computer, or data set. The second approach relies on a mathematical **analysis of algorithms**, independently of the particular implementation and input, as discussed below.

BENCHMARKING

ANALYSIS OF ALGORITHMS

A.1.1 Asymptotic analysis

We will consider algorithm analysis through the following example, a program to compute the sum of a sequence of numbers:

```
function SUMMATION(sequence) returns a number
   $sum \leftarrow 0$
   for $i$ = 1 to LENGTH($sequence$) do
      $sum \leftarrow sum + sequence[i]$
   return $sum$
```

The first step in the analysis is to abstract over the input, in order to find some parameter or parameters that characterize the size of the input. In this example, the input can be characterized by the length of the sequence, which we will call n. The second step is to abstract over the implementation, to find some measure that reflects the running time of the algorithm but is not tied to a particular compiler or computer. For the SUMMATION program, this could be just the number of lines of code executed, or it could be more detailed, measuring the number of additions, assignments, array references, and branches executed by the algorithm.

Either way gives us a characterization of the total number of steps taken by the algorithm as a function of the size of the input. We will call this characterization $T(n)$. If we count lines of code, we have $T(n) = 2n + 2$ for our example.

If all programs were as simple as SUMMATION, the analysis of algorithms would be a trivial field. But two problems make it more complicated. First, it is rare to find a parameter like n that completely characterizes the number of steps taken by an algorithm. Instead, the best we can usually do is compute the worst case $T_{\mathrm{worst}}(n)$ or the average case $T_{\mathrm{avg}}(n)$. Computing an average means that the analyst must assume some distribution of inputs.

The second problem is that algorithms tend to resist exact analysis. In that case, it is necessary to fall back on an approximation. We say that the SUMMATION algorithm is $O(n)$, meaning that its measure is at most a constant times n, with the possible exception of a few small values of n. More formally,

$$T(n) \text{ is } O(f(n)) \text{ if } T(n) \leq kf(n) \text{ for some } k, \text{ for all } n > n_0 .$$

ASYMPTOTIC ANALYSIS

The $O()$ notation gives us what is called an **asymptotic analysis**. We can say without question that, as n asymptotically approaches infinity, an $O(n)$ algorithm is better than an $O(n^2)$ algorithm. A single benchmark figure could not substantiate such a claim.

The $O()$ notation abstracts over constant factors, which makes it easier to use, but less precise, than the $T()$ notation. For example, an $O(n^2)$ algorithm will always be worse than an $O(n)$ in the long run, but if the two algorithms are $T(n^2 + 1)$ and $T(100n + 1000)$, then the $O(n^2)$ algorithm is actually better for $n < 110$.

Despite this drawback, asymptotic analysis is the most widely used tool for analyzing algorithms. It is precisely because the analysis abstracts over both the exact number of operations (by ignoring the constant factor k) and the exact content of the input (by considering only its size n) that the analysis becomes mathematically feasible. The $O()$ notation is a good compromise between precision and ease of analysis.

A.1.2 NP and inherently hard problems

COMPLEXITY ANALYSIS

The analysis of algorithms and the $O()$ notation allow us to talk about the efficiency of a particular algorithm. However, they have nothing to say about whether there could be a better algorithm for the problem at hand. The field of **complexity analysis** analyzes problems rather than algorithms. The first gross division is between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time, no matter what algorithm is used. The class of polynomial problems—those which can be solved in time $O(n^k)$ for some k—is called P. These are sometimes called "easy" problems, because the class contains those problems with running times like $O(\log n)$ and $O(n)$. But it also contains those with time $O(n^{1000})$, so the name "easy" should not be taken too literally.

Another important class of problems is NP, the class of nondeterministic polynomial problems. A problem is in this class if there is some algorithm that can guess a solution and then verify whether the guess is correct in polynomial time. The idea is that if you have an arbitrarily large number of processors, so that you can try all the guesses at once, or you are very lucky and always guess right the first time, then the NP problems become P problems. One of the biggest open questions in computer science is whether the class NP is equivalent to the class P when one does not have the luxury of an infinite number of processors or omniscient guessing. Most computer scientists are convinced that P $\neq$ NP; that NP problems are inherently hard and have no polynomial-time algorithms. But this has never been proven.

NP-COMPLETE

Those who are interested in deciding whether P = NP look at a subclass of NP called the **NP-complete** problems. The word "complete" is used here in the sense of "most extreme" and thus refers to the hardest problems in the class NP. It has been proven that either all the NP-complete problems are in P or none of them is. This makes the class theoretically interesting, but the class is also of practical interest because many important problems are known to be NP-complete. An example is the satisfiability problem: given a sentence of propositional logic, is there an assignment of truth values to the proposition symbols of the sentence that makes it true? Unless a miracle occurs and P = NP, there can be no algorithm that solves *all* satisfiability problems in polynomial time. However, AI is more interested in whether there are algorithms that perform efficiently on *typical* problems drawn from a predetermined distribution; as we saw in Chapter 7, there are algorithms such as WALKSAT that do quite well on many problems.

CO-NP

CO-NP-COMPLETE

The class **co-NP** is the complement of NP, in the sense that, for every decision problem in NP, there is a corresponding problem in co-NP with the "yes" and "no" answers reversed. We know that P is a subset of both NP and co-NP, and it is believed that there are problems in co-NP that are not in P. The **co-NP-complete** problems are the hardest problems in co-NP.

The class #P (pronounced "sharp P") is the set of counting problems corresponding to the decision problems in NP. Decision problems have a yes-or-no answer: is there a solution to this 3-SAT formula? Counting problems have an integer answer: how many solutions are there to this 3-SAT formula? In some cases, the counting problem is much harder than the decision problem. For example, deciding whether a bipartite graph has a perfect matching can be done in time $O(VE)$ (where the graph has V vertices and E edges), but the counting problem "how many perfect matches does this bipartite graph have" is #P-complete, meaning that it is hard as any problem in #P and thus at least as hard as any NP problem.

Another class is the class of PSPACE problems—those that require a polynomial amount of space, even on a nondeterministic machine. It is believed that PSPACE-hard problems are worse than NP-complete problems, although it could turn out that NP = PSPACE, just as it could turn out that P = NP.

A.2 VECTORS, MATRICES, AND LINEAR ALGEBRA

VECTOR

Mathematicians define a **vector** as a member of a vector space, but we will use a more concrete definition: a vector is an ordered sequence of values. For example, in two-dimensional space, we have vectors such as $\mathbf{x} = \langle 3, 4 \rangle$ and $\mathbf{y} = \langle 0, 2 \rangle$. We follow the convention of boldface characters for vector names, although some authors use arrows or bars over the names: $\vec{x}$ or $\bar{y}$. The elements of a vector can be accessed using subscripts: $\mathbf{z} = \langle z_1, z_2, \ldots, z_n \rangle$. One confusing point: this book is synthesizing work from many subfields, which variously call their sequences vectors, lists, or tuples, and variously use the notations $\langle 1, 2 \rangle$, [1, 2], or (1, 2).

The two fundamental operations on vectors are vector addition and scalar multiplication. The vector addition $\mathbf{x}+\mathbf{y}$ is the elementwise sum: $\mathbf{x}+\mathbf{y} = \langle 3+0, 4+2\rangle = \langle 3, 6\rangle$. Scalar multiplication multiplies each element by a constant: $5\mathbf{x} = \langle 5 \times 3, 5 \times 4\rangle = \langle 15, 20\rangle$.

The length of a vector is denoted $|\mathbf{x}|$ and is computed by taking the square root of the sum of the squares of the elements: $|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5$. The dot product $\mathbf{x} \cdot \mathbf{y}$ (also called scalar product) of two vectors is the sum of the products of corresponding elements, that is, $\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$, or in our particular case, $\mathbf{x} \cdot \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$.

Vectors are often interpreted as directed line segments (arrows) in an n-dimensional Euclidean space. Vector addition is then equivalent to placing the tail of one vector at the head of the other, and the dot product $\mathbf{x} \cdot \mathbf{y}$ is equal to $|\mathbf{x}|\ |\mathbf{y}|\ \cos\theta$, where θ is the angle between $\mathbf{x}$ and $\mathbf{y}$.

MATRIX

A **matrix** is a rectangular array of values arranged into rows and columns. Here is a matrix $\mathbf{A}$ of size 3×4:

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} & \mathbf{A}_{2,4} \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4} \end{pmatrix}$$

The first index of $\mathbf{A}_{i,j}$ specifies the row and the second the column. In programming languages, $\mathbf{A}_{i,j}$ is often written `A[i,j]` or `A[i][j]`.

The sum of two matrices is defined by adding their corresponding elements; for example $(\mathbf{A}+\mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$. (The sum is undefined if $\mathbf{A}$ and $\mathbf{B}$ have different sizes.) We can also define the multiplication of a matrix by a scalar: $(c\mathbf{A})_{i,j} = c\mathbf{A}_{i,j}$. Matrix multiplication (the product of two matrices) is more complicated. The product $\mathbf{AB}$ is defined only if $\mathbf{A}$ is of size $a \times b$ and $\mathbf{B}$ is of size $b \times c$ (i.e., the second matrix has the same number of rows as the first has columns); the result is a matrix of size $a \times c$. If the matrices are of appropriate size, then the result is

$$(\mathbf{AB})_{i,k} = \sum_j \mathbf{A}_{i,j}\mathbf{B}_{j,k}\ .$$

Matrix multiplication is not commutative, even for square matrices: $\mathbf{AB} \neq \mathbf{BA}$ in general. It is, however, associative: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Note that the dot product can be expressed in terms of a transpose and a matrix multiplication:$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\top \mathbf{y}$.

IDENTITY MATRIX
TRANSPOSE
INVERSE
SINGULAR

The **identity matrix I** has elements $\mathbf{I}_{i,j}$ equal to 1 when $i = j$ and equal to 0 otherwise. It has the property that $\mathbf{AI} = \mathbf{A}$ for all $\mathbf{A}$. The **transpose** of $\mathbf{A}$, written $\mathbf{A}^\top$ is formed by turning rows into columns and vice versa, or, more formally, by $\mathbf{A}^\top{}_{i,j} = \mathbf{A}_{j,i}$. The **inverse** of a square matrix $\mathbf{A}$ is another square matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. For a **singular** matrix, the inverse does not exist. For a nonsingular matrix, it can be computed in $O(n^3)$ time.

Matrices are used to solve systems of linear equations in $O(n^3)$ time; the time is dominated by inverting a matrix of coefficients. Consider the following set of equations, for which we want a solution in x, y, and z:

$$\begin{aligned} +2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3\ . \end{aligned}$$

We can represent this system as the matrix equation $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, where

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 8 \\ -11 \\ -3 \end{pmatrix}.$$

To solve $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ we multiply both sides by $\mathbf{A}^{-1}$, yielding $\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, which simplifies to $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. After inverting $\mathbf{A}$ and multiplying by $\mathbf{b}$, we get the answer

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}.$$

A.3 Probability Distributions

A probability is a measure over a set of events that satisfies three axioms:

1. The measure of each event is between 0 and 1. We write this as $0 \le P(X = x_i) \le 1$, where X is a random variable representing an event and x_i are the possible values of X. In general, random variables are denoted by uppercase letters and their values by lowercase letters.
2. The measure of the whole set is 1; that is, $\sum_{i=1}^{n} P(X = x_i) = 1$.
3. The probability of a union of disjoint events is the sum of the probabilities of the individual events; that is, $P(X = x_1 \vee X = x_2) = P(X = x_1) + P(X = x_2)$, where x_1 and x_2 are disjoint.

A **probabilistic model** consists of a sample space of mutually exclusive possible outcomes, together with a probability measure for each outcome. For example, in a model of the weather tomorrow, the outcomes might be *sunny, cloudy, rainy*, and *snowy*. A subset of these outcomes constitutes an event. For example, the event of precipitation is the subset consisting of $\{$*rainy, snowy*$\}$.

We use $\mathbf{P}(X)$ to denote the vector of values $\langle P(X = x_1), \ldots, P(X = x_n) \rangle$. We also use $P(x_i)$ as an abbreviation for $P(X = x_i)$ and $\sum_x P(x)$ for $\sum_{i=1}^{n} P(X = x_i)$.

The conditional probability $P(B|A)$ is defined as $P(B \cap A)/P(A)$. A and B are conditionally independent if $P(B|A) = P(B)$ (or equivalently, $P(A|B) = P(A)$). For continuous variables, there are an infinite number of values, and unless there are point spikes, the probability of any one value is 0. Therefore, we define a **probability density function**, which we also denote as $P(\cdot)$, but which has a slightly different meaning from the discrete probability function. The density function $P(x)$ for a random variable X, which might be thought of as $P(X = x)$, is intuitively defined as the ratio of the probability that X falls into an interval around x, divided by the width of the interval, as the interval width goes to zero:

PROBABILITY DENSITY FUNCTION

$$P(x) = \lim_{dx \to 0} P(x \le X \le x + dx)/dx\ .$$

The density function must be nonnegative for all x and must have

$$\int_{-\infty}^{\infty} P(x)\,dx = 1\ .$$

CUMULATIVE PROBABILITY DENSITY FUNCTION

We can also define a **cumulative probability density function** $F_X(x)$, which is the probability of a random variable being less than x:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} P(u)\,du\ .$$

Note that the probability density function has units, whereas the discrete probability function is unitless. For example, if values of X are measured in seconds, then the density is measured in Hz (i.e., 1/sec). If values of **X** are points in three-dimensional space measured in meters, then density is measured in $1/m^3$.

GAUSSIAN DISTRIBUTION

One of the most important probability distributions is the **Gaussian distribution**, also known as the **normal distribution**. A Gaussian distribution with mean μ and standard deviation σ (and therefore variance σ^2) is defined as

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}\ ,$$

STANDARD NORMAL DISTRIBUTION

MULTIVARIATE GAUSSIAN

where x is a continuous variable ranging from $-\infty$ to $+\infty$. With mean $\mu = 0$ and variance $\sigma^2 = 1$, we get the special case of the **standard normal distribution**. For a distribution over a vector **x** in n dimensions, there is the **multivariate Gaussian** distribution:

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}} e^{-\frac{1}{2}\left((\mathbf{x}-\boldsymbol{\mu})^\top \mathbf{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right)}\ ,$$

where $\boldsymbol{\mu}$ is the mean vector and $\mathbf{\Sigma}$ is the **covariance matrix** (see below).

CUMULATIVE DISTRIBUTION

In one dimension, we can define the **cumulative distribution** function $F(x)$ as the probability that a random variable will be less than x. For the normal distribution, this is

$$F(x) = \int_{-\infty}^{x} P(z)dz = \frac{1}{2}(1 + \text{erf}(\frac{z-\mu}{\sigma\sqrt{2}}))\ ,$$

where erf(x) is the so-called **error function**, which has no closed-form representation.

CENTRAL LIMIT THEOREM

The **central limit theorem** states that the distribution formed by sampling n independent random variables and taking their mean tends to a normal distribution as n tends to infinity. This holds for almost any collection of random variables, even if they are not strictly independent, unless the variance of any finite subset of variables dominates the others.

EXPECTATION

The **expectation** of a random variable, $E(X)$, is the mean or average value, weighted by the probability of each value. For a discrete variable it is:

$$E(X) = \sum_i x_i\, P(X = x_i)\ .$$

For a continuous variable, replace the summation with an integral over the probability density function, $P(x)$:

$$E(X) = \int_{-\infty}^{\infty} xP(x)\,dx\ ,$$

ROOT MEAN SQUARE

The **root mean square**, RMS, of a set of values (often samples of a random variable) is the square root of the mean of the squares of the values,

$$RMS(x_1, \ldots, x_n) = \sqrt{\frac{x_1^2 + \ldots + x_n^2}{n}} \ .$$

COVARIANCE

The **covariance** of two random variables is the expectation of the product of their differences from their means:

$$\text{cov}(X, Y) = E((X - \mu_X)(Y - \mu_Y)) \ .$$

COVARIANCE MATRIX

The **covariance matrix**, often denoted $\mathbf{\Sigma}$, is a matrix of covariances between elements of a vector of random variables. Given $\mathbf{X} = \langle X_1, \ldots X_n \rangle^\top$, the entries of the covariance matrix are as follows:

$$\mathbf{\Sigma}_{i,j} = \text{cov}(X_i, X_j) = E((X_i - \mu_i)(X_j - \mu_j)) \ .$$

A few more miscellaneous points: we use $\log(x)$ for the natural logarithm, $\log_e(x)$. We use $\text{argmax}_x\, f(x)$ for the value of x for which $f(x)$ is maximal.

Bibliographical and Historical Notes

The $O()$ notation so widely used in computer science today was first introduced in the context of number theory by the German mathematician P. G. H. Bachmann (1894). The concept of NP-completeness was invented by Cook (1971), and the modern method for establishing a reduction from one problem to another is due to Karp (1972). Cook and Karp have both won the Turing award, the highest honor in computer science, for their work.

Classic works on the analysis and design of algorithms include those by Knuth (1973) and Aho, Hopcroft, and Ullman (1974); more recent contributions are by Tarjan (1983) and Cormen, Leiserson, and Rivest (1990). These books place an emphasis on designing and analyzing algorithms to solve tractable problems. For the theory of NP-completeness and other forms of intractability, see Garey and Johnson (1979) or Papadimitriou (1994). Good texts on probability include Chung (1979), Ross (1988), and Bertsekas and Tsitsiklis (2008).