2. Useful Mathematical Concepts

Just as you can follow a recipe from a cookbook without really learning to cook, it is certainly possible to study algorithms while avoiding mathematics and proofs. This may, in fact, be preferable for the practitioner who simply wants help implementing fundamental algorithms. However, our goal in this text is to learn how to cook, so to speak. A sound mathematical background is a fundamental prerequisite for success in the design and analysis of novel algorithms for novel problems.

Algorithmic study is well-suited for those who appreciate mathematics, and can help develop such an appreciation for those with limited prior mathematical exposure, since can learn much about mathematics as a side benefit of studying algorithms. We assume the reader has some background in discrete mathematics, but we stress that most of the techniques we use should be quite accessible even for those without extensive training. Algorithmic analysis mostly entails clever application of basic mathematical concepts, rather than deep results from advanced mathematics.

In this chapter, we review mathematical techniques used throughout the rest of the book. We begin with basic notation and terminology, proof techniques, and helpful "tricks". Next, we cover three topics in greater detail that we will find particularly useful: methods for solving recursively-defined expressions called *recurrences* (to analyze recursive algorithms), techniques from probability theory (to analyze randomized algorithms), and important concepts from linear algebra (to analyze some of the more advanced mathematical algorithms in this book). Finally, we highlight a few remaining bits of prerequisite knowledge that may benefit the reader from other areas of mathematical study. The reader with a relatively strong mathematical background is still advised to peruse this chapter, since it includes discussion and problems covering a number of interesting algorithmic applications.

2.1 Notation, Definitions, and Assumptions

Our notation is fairly standard and should not differ appreciably from any other mathematical text.

Sets. We expect the reader to be familiar with standard notation for manipulating sets. For example, $S \cap T$ denotes the intersection of sets S and T, and $S \cup T$ denotes their union. There are several "formulaic" ways to prove certain types of statements regarding sets. We can prove that $S = T$ by arguing that $S \subseteq T$ and also $T \subseteq S$. To show that $x \in S \cap T$, we could prove separately that $x \in S$ and $x \in T$. To argue that $S \subseteq T$, we might take an arbitrary element $x \in S$ and show that $x \in T$ as well. As a shorthand for summing over a set, we sometimes use a set-valued argument to a function:

$$f(S) = \sum_{x \in S} f(x) \quad \text{and} \quad g(S,T) = \sum_{x \in S, y \in T} g(x,y).$$

We often use interval notation to describe a contiguous set of numbers. For example, $[0,1]$ denotes the closed set of all real numbers between 0 and 1, with both endpoints included in the set, and $[5,10)$, denotes the set of all real numbers between 5 and 10, including the endpoint 5 but not 10. The *Cartesian product* of two sets, $A \times B$, is the set of all pairs (a,b) where $a \in A$ and $b \in B$. For example, $[0,1] \times [0,1]$ is the unit square in the 2D plane with corners $(0,0)$ and $(1,1)$.

Logarithms. Logs show up everywhere in the study of algorithms. As with binary search, many algorithms successively reduce a problem of size n to a smaller equivalent problem of size at most n/k (for binary search, $k = 2$). It takes $\log_k n$ such phases to reduce a problem of size n down to a trivial problem of size 1. By $\log n$ we mean $\log_2 n$ by default; we write $\ln n$ for logarithms in base e. The base of a logarithm usually doesn't matter in an asymptotic running time expression, since logs of different constant bases differ only by constant factors. The reader should be comfortable manipulating expressions involving logarithms. For example, $\log n^k = k \log n$, $2^{\log n} = n$, and $a^{\log b} = b^{\log a}$.

Sequences. We use subscripts to write a sequence of elements as $A_1, A_2, \ldots, A_n$. If the elements represent characters drawn from some alphabet, we sometimes call the sequence a *string.* The words *list* and *array* can also refer to a sequence, but we typically use these only for sequences that are specifically implemented as linked lists or arrays. We use the notation $A[1], A[2], \ldots, A[n]$ for the elements of a sequence stored in an array in memory. Our sequences typically start at index 1 instead of index 0. Finally, a *substring* or *subarray* refers to a *contiguous* block of elements $A_i \ldots A_j$ within a sequence/string/array, while a *subsequence* refers to a subset of the elements of the sequence (taken in the same order as they appear in the sequence), which may not be a single contiguous block.

Sequence S is *lexicographically smaller* than another sequence S' if $S_j < S'_j$ at the first index j where they disagree, or if there is no disagreement but S is shorter than S'. Lexicographic ordering is a natural way to order objects that have multiple components, like sequences, strings, and vectors. For text strings, lexicographic ordering corresponds to our familiar notion of alphabetic ordering.

Number Bases. We expect the reader to be familiar with the notion of different number bases. In particular, since digital computers naturally represent all numbers in binary (base 2), the reader should be comfortable with expressing numbers in binary. For example, we would write the base-10 number 19 in binary as 10011, since $19 = 16(\mathbf{1}) + 8(\mathbf{0}) + 4(\mathbf{0}) + 2(\mathbf{1}) + 1(\mathbf{1})$. In any number base, we call the leftmost digit of a number the *most significant digit* and the rightmost digit the *least significant digit.*

Factorials, Permutations, and Combinations. The number of *permutations* (different orderings) of n distinct elements is $n! = 1 \times 2 \times 3 \times \ldots \times n$. The number of different subsets of k elements we can choose from a set of n distinct elements is given by the binomial coefficient $\binom{n}{k}$, where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \leq n^k$$

The term $\binom{n}{k}$ is read "n choose k" and is sometimes written elsewhere as $C(n,k)$ or ${}_nC_k$. For example, if we have an algorithm that examines every pair of elements in an n-element array, it will examine $\binom{n}{2} = n(n-1)/2 \leq n^2$ pairs of elements.

2.1.1 Floors, Ceilings, and Other Pesky Details

In the literature on algorithms, it is customary to allow for a bit of "sloppiness" in certain aspects of our notation. This lets us avoid minor, irrelevant details in order to explain concepts more clearly at a high level. For example, we might say that binary search looks at the middle element of an array, $A[n/2]$, and then decides to search within only the first half or second half of A. If our array has an odd number of elements, then $n/2$ will not be an integer, so we should technically say $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$. Similarly, the first "half" of the array might not contain exactly half the elements. However, thanks to our use of asymptotic notation, details like this rarely make a difference in our final asymptotic running time. Hence, for simplicity we tend to say things like "$n/2$" and "half" knowing that there may actually be an implicit floor or ceiling required.

To give some more examples, suppose we say that "our algorithm partitions an input of size n into $\sqrt{n}$ blocks each of size $\sqrt{n}$" or "our algorithm runs in $\log n$ phases". In the first case, if n is not a perfect square, this implicitly means that our blocks may have size $\lceil \sqrt{n} \rceil$ or $\lfloor \sqrt{n} \rfloor$ and that the last "leftover" block may be smaller than the rest. In the second case, if n is not a power of 2 then perhaps there will actually be $\lfloor \log_2 n \rfloor$ full phases and one left-over partial phase. Presumably, these are minor details, straightforward to figure out during implementation, and irrelevant to our final asymptotic running time. If desired, we can often avoid the need for special cases like those above by adding "dummy" elements to our input (say, to increase the size to an exact power of 2). Since this usually only inflates the input size by a constant factor, it rarely changes our asymptotic running time.

2.1.2 Asymptotic Notation

Back in Section 1.4, we introduced asymptotic notation and motivated its importance in describing algorithmic running times. We learned that:

- An expression like the running time of an algorithm is said to be $O(f(n))$ if it is bounded above by some constant times $f(n)$ as n grows sufficiently large. For example, $\sqrt{n}$, n^2, and $1000n^2 + n$ are all $O(n^2)$.
- An expression is $\Omega(f(n))$ if it is bounded below by a constant times $f(n)$ for sufficiently large n. For example, $n^2 - 10n + 2$, n^3, and 2^n are all $\Omega(n^2)$.
- An expression is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$. Since it includes both an upper and lower bound, a $\Theta(\cdot)$ bound gives us a precise characteriza-

tion of the rate of growth (up to constant factors) of an algorithm's running time. For example, n^2, $3n^2$ and $7n^2 - 5n$ are all $\Theta(n^2)$.

We can round out this list with the following additional definitions:

- An expression $f(n)$ is said to be $o(g(n))$ if the limit of $f(n)/g(n)$ goes to zero as n grows sufficiently large. Here, f grows "strictly" slower than g, so this is a stronger statement than saying $f(n)$ is $O(g(n))$. For example, $3n^2$ is $o(n^3)$.
- An expression $f(n)$ is said to be $\omega(g(n))$ if the limit of $f(n)/g(n)$ goes to infinity as n grows sufficiently large. Here, f grows "strictly" faster than g, so this is a stronger statement than saying $f(n)$ is $\Omega(g(n))$. For example, $n^{2.1}$ is $\omega(n^2)$.

Informally, one should think of O, Ω, Θ, o, and ω as the equivalents of $\leq$, $\geq$, $=$, $<$, and $>$ when comparing the asymptotic growth of functions.

We often use asymptotic expressions as "placeholders". For example, we might write a polynomial as $7n^3 - 3n^2 + o(n^2)$ where the $o(n^2)$ serves as a placeholder to indicate the presence of lower order terms. Please be careful when performing algebra involving these asymptotic placeholders. If we perform $\Theta(n^2)$ iterations of an algorithm whose running time is $O(n^7)$, we can rightfully multiply these to obtain a total running time of $O(n^9)$. However, it does not make sense to divide both sides of an equation by $o(n^2)$. When in doubt, go back to the definitions above to check if a particular arithmetic manipulation makes sense.

2.1.3 Special Functions

On occasion, we will encounter two special functions that grow extremely slowly, much slower even than $\log n$. These functions are generally excellent news in terms of efficiency, since they give us very small running times even for huge values of n.

The $\log^* n$ Function. Imagine that you have a calculator with an "$f(n)$" key, so for example pressing the key three times would transform n into $f(f(f(n)))$. Assuming f is decreasing, we define $f^*(n)$ as the number of key presses needed to decrease n down to a value that is at most 1. For example, if $f(n) = n - 2$, then $f^*(n) = \lfloor n/2 \rfloor$, since $n/2$ subtractions of 2 are needed. If $f(n) = n/2$, then $f^*(n) = \lceil \log n \rceil$. This "star" operator is important when dealing with recursively-defined algorithms, since if an algorithm solves a problem of size n by recursively solving subproblems of size $f(n)$, then the algorithm will recurse for $f^*(n)$ levels. For example, each step of binary search reduces a problem of size n to a recursive subproblem of size $f(n) = n/2$, so binary search recurses to a depth of $f^*(n) = O(\log n)$.

The star operator applied to $f(n) = \log n$ gives the function $\log^* n$, one of the slowest-growing functions we occasionally encounter in the analysis of algorithms. To give you a sense of just how slowly this function grows, we have $\log^* n = 5$ if $n = 2^{65536}$, a number much larger than the number of atoms in the observable universe. For all practical purposes, we can think of $\log^* n$ as being constant, although from a theoretical standpoint we cannot simply ignore such terms since they do grow as functions of n.

Problem 8 (Practice Applying the Star Operator). If $f(n) = \sqrt{n}$, give a simple proof that $f^*(n) = \Theta(\log\log n)$. Repeated square roots never decrease below one, so please stop at 2 instead of 1 in the definition of f^* (typically any constant is fine, as long as we only care about the asymptotic form of the answer). [Solution]

Problem 9 (The Logsum Function, Binary Codes, and Unbounded Search). The function $\text{logsum}(n) = \log n + \log\log n + \log\log\log n + \ldots$ (with $\log^* n$ terms) plays an important role in several algorithmic results. For example, suppose we want to encode in a single binary stream a sequence of positive integers, with no bounds on their sizes. We could of course write each number in binary, but we also need to encode information on the number of bits we will use, since otherwise we won't know where one number stops and another starts. The clever Elias "Omega" code uses $\text{logsum}(n) + O(\log^* n)$ bits to encode n by writing n in binary after recursively encoding $\lceil \log n \rceil$, the number of bits required for n (so we encode $\lceil \log n \rceil$ in binary using $\lceil \log\log n \rceil$ bits, before which we write its length $\lceil \log\log n \rceil$ using $\lceil \log\log\log n \rceil$ bits, and so on). Inspired by this approach, consider now the related problem of unbounded search for a positive integer n, where we are given no upper bound on n. A simple approach taking at most $2\lceil \log n \rceil$ comparisons to find n is to start with $x = 1$ and successively double x until $x \geq n$ (taking $\lceil \log n \rceil$ steps), then to binary search the range $[1, x]$ for n (taking another $\lceil \log n \rceil$ steps). For a challenge, can you show how to improve on the first half of the algorithm in order to perform unbounded search in only $\text{logsum}(n) + O(\log^* n)$ comparisons? As a hint, x is just a power of two, so recursively search for $\log x$. How is the unbounded search problem essentially equivalent to the binary encoding problem above? [Solution]

The Inverse Ackermann Function. Designing a function that grows even more slowly than $\log^* n$ is as simple as just adding more stars. For example, $\log^{**} n$ is the number of times we need to iteratively apply the $\log^* n$ function to n before we reach a value no larger than 1. For $n > 4$, the value of $\log^{*\cdots*} n$ is at least 3 no matter how many stars we use, and if we add enough stars, we will eventually reach a point where $\log^{*\cdots*} n = 3$. We define the *inverse Ackermann function*, $\alpha(n)$, as one plus the minimum number of stars required to make $\log^{*\cdots*} n \leq 3$ (we add one to ensure that $\alpha(n)$ is always at least positive). On occasion, we will use a stronger two-term variant $\alpha(m, n) \leq \alpha(n)$, giving one plus the minimum number of stars required to make $\log^{*\cdots*} n \leq 3 + m/n$. One can show that $\alpha(n) = o(\log^{*\cdots*} n)$ for any constant number of stars, so inverse Ackermann running time bounds are even faster than those involving multiply-iterated logs. Almost every appearance of an inverse Ackermann running time originates from just one place — the analysis of a prominent data structure for maintaining disjoint sets, discussed in Section 4.6.

The *Ackermann function* is a famous recursively-defined function designed to grow *extremely* quickly. Many variants of the Ackermann function appear in the literature, and consequently one can also find different ways of defining inverse Ackermann functions, all of which lead to essentially the same asymptotic behavior. We don't use the Ackermann function directly in this book, so we won't say any more about it here in the main text. However, we have included further detail in the endnotes, for the interested reader.

2.1.4 Graphs and Trees

Graphs are particularly prominent objects in computer science, due to their flexibility in being able to model so many different things. Every graph consists of a set

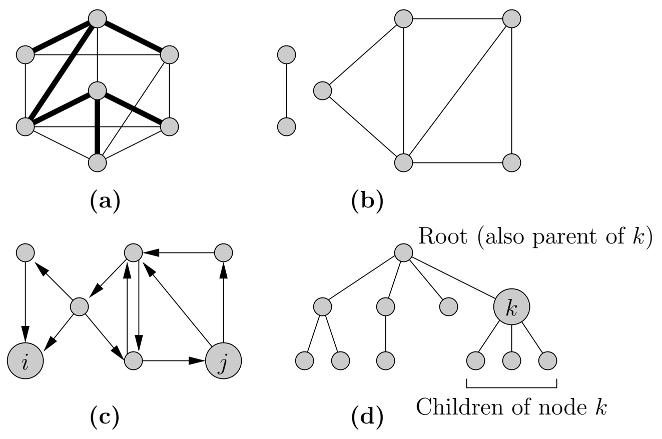


FIGURE 2.1: Examples of: (a) a connected graph with a spanning tree highlighted, (b) a graph with two connected components, (c) a directed graph where there is no directed path from i to j, but there is a directed path from j to i, (d) a rooted tree of height 2.

of *nodes* (sometimes called *vertices* or *points*) and *edges* (sometimes called *links* or *lines*), where each edge connects a pair of nodes. Depending on our physical interpretation of nodes and edges, a graph can represent virtually any kind of network (transportation, communication, social, etc.), as well as a more abstract object like a mathematical relation.

Graphs can be *undirected* or *directed*. The word "graph" technically implies an undirected graph, while "digraph" specifies a directed graph; we sometimes use the term "graph" generically to refer to both. An edge directed from node i to node j in a directed graph (sometimes called an *arc*) is written as an ordered pair (i, j), while an undirected edge between i and j in a graph is written as a set $\{i, j\}$, since this has no directional implications. The notation ij is also sometimes used for describing either a directed or undirected edge from i to j. We usually assume no edge is a *self loop* — connecting a node to itself. We generally also assume that our graphs are *simple*, meaning there is at most one edge ij for any pair of nodes (i, j). Otherwise, if multiple "parallel" edges can run between the same pair of nodes, we have a *multigraph*. The *degree* of a node is the number of edges *incident* to the node. In a directed graph, we distinguish *indegree* (number of incoming edges) and *outdegree* (number of outgoing edges). We often consider *weighted* graphs by associating numeric "weights" with edges and/or nodes. Depending on the application, these may have physical meanings as costs, values, capacities, lengths, and so on.

By convention, n represents the number of nodes in a graph, and m represents the number of edges. A graph with few edges (where m is closer to n) is said to be *sparse*, while a graph with many edges (where m is closer to n^2) is said to be *dense*. Some algorithms are better suited to sparse graphs, and others dense graphs. In practice, most large graphs (e.g., the directed graph depicting the link structure of the World Wide Web) tend to be quite sparse. The running time of a graph algorithm will usually depend on both n and m. Since simple graphs satisfy

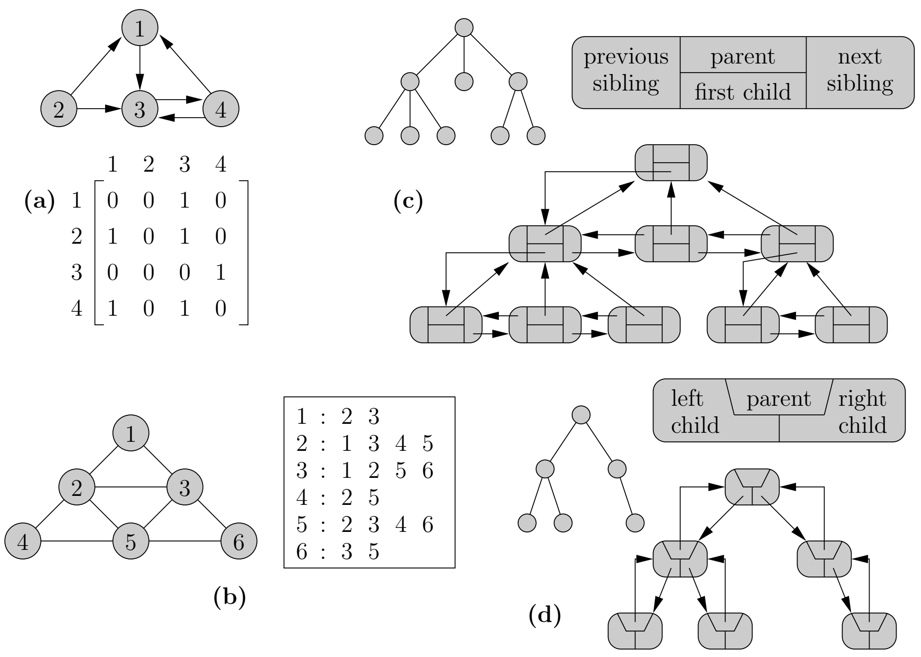
$m \leq n^2$, we can technically replace m with $O(n^2)$ to write these running times solely in terms of n. However, this is generally not a good idea, since in a sparse graph, an $O(m)$ running time is more like $O(n)$ than $O(n^2)$, and we might discourage someone from using a fast $O(m)$ algorithm if we describe its running time solely as $O(n^2)$.

Paths and Cycles. A *path* is a series of nodes connected by edges; we can regard this either as a sequence of nodes or as a sequence of edges. A path that leaves and then returns to the same node where it started is called a *cycle*. In a directed graph, paths and cycles contain edges that are consistently directed. A path is *simple* if it contains no cycles, and a cycle is simple if it contains no cycles shorter than itself. When we say "path" and "cycle" in this book, we mean a simple path or cycle. We use the term *walk* for a path that need not be simple. The *length* of a path or cycle is the number of edges it contains (or in a weighted graph, we often use instead the sum of its edge weights). A *Hamiltonian* path or cycle visits every node exactly once, and an *Eulerian* path or cycle follows every edge exactly once.

Connectivity. Nodes i and j are *adjacent* if they are directly connected by an edge, and *connected* if they are connected by a path. In a directed graph, there can possibly be an edge or path from i to j but not in the reverse direction. A graph is *connected* if every pair of nodes is connected; for example, Figure 2.1(a) is connected while Figure 2.1(b) consists of two separate *connected components*. In Chapter 15 we extend the notion of connectivity and connected components to directed graphs, as well as higher orders of connectivity (e.g., we can say that two nodes are highly connected if they are connected by a large number of edge-disjoint paths, or alternatively if we need to remove a large number of edges to separate them into distinct connected components).

Trees. A tree is a graph that is connected and *acyclic* (having no cycles). Often we designate a specific node in a tree as a *root*, allowing us to orient the tree so that every node has a well-defined *parent* (except the root) and set of *children*, as shown in Figure 2.1(d). A tree with no designated root is called a *free tree*; the word "tree" by itself can mean either a free tree or a rooted tree. We will sometimes take a free tree and "root it", designating one of its nodes as a root, after which we can then operate on it like a rooted tree. Rooted trees play a key role in most data structures, as we shall see in Chapters 4 through 9. As opposed to reality, rooted trees in computer science are usually drawn growing downward from a root at the top. Accordingly, the *depth* of a node in a rooted tree is its distance downward from the root, and the *height* of a rooted tree is the maximum depth over all nodes. Node i is an *ancestor* of node j in a rooted tree if j appears inside the *subtree* rooted at i (so i lies on the path from j up to the root). A node of degree 1 (in a rooted tree, a node with no children) is called a *leaf*. Other nodes are called *internal nodes*.

Subgraphs and Spanning Trees. A subset of the edges of a graph is called a *subgraph*. A subset of nodes *induces* a subgraph consisting of all edges with both endpoints in the subset. A particularly common subgraph in algorithmic computing is a *spanning tree* — a subset of edges that connects together all nodes and contains no cycles, as shown in Figure 2.1(a). For example, a spanning tree in a communication network gives us a minimal subset of edges within which we can still route information between all pairs of nodes. This simplifies the routing task as well, since within any tree there is a unique path joining any given pair of nodes. In the case where we want to broadcast information from one node r to all other



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$

```
1 : 2 3
2 : 1 3 4 5
3 : 1 2 5 6
4 : 2 5
5 : 2 3 4 6
6 : 3 5
```

FIGURE 2.2: Illustrations of (a) a directed graph represented by an adjacency matrix, (b) a graph represented by adjacency lists, (c) the representation of an ordered rooted tree where each node maintains a pointer to its parent, first child, and previous and next siblings, and (d) the representation of a binary tree where each node maintains a pointer to its parent, left child, and right child. Boxes from which no pointers emanate indicate "null" pointers.

nodes, we end up with a tree rooted at r. Such trees are often thought of as being "directed" outward from the root r. In Chapter 17 we will more formally study directed notions of trees, known as *branchings* and *arborescences*.

Representing a Graph in Memory. Two common ways to represent a graph are using an *adjacency matrix* or with *adjacency lists*. As shown in Figure 2.2(a), the adjacency matrix is an $n \times n$ matrix in which the (i, j) entry is 1 if there is an edge from i to j, and 0 otherwise. For an undirected graph, the matrix is symmetric since the (i, j) and (j, i) entries are equal. Adjacency matrices are well-suited for dense graphs; otherwise, they can be a liability both in terms of space and running time. A graph algorithm utilizing adjacency matrices typically cannot have a better running time than $\Theta(n^2)$ due to the need to examine the entire input matrix. Furthermore, although adjacency matrices allow us to query whether an edge (i, j) is present in only $O(1)$ time, they require $\Theta(n)$ time to enumerate the neighbors of a node, regardless of how many neighbors there are.

Figure 2.2(b) shows a graph represented by adjacency lists, where we maintain for every node i an array or linked list of the neighbors of i. In a directed graph, we usually only maintain edges directed out of i, although it is sometimes convenient to maintain a list of incoming edges as well. Adjacency lists are ideal for sparse graphs, since they require only $\Theta(m + n)$ space (linear in the size of the graph).

Sometimes we store adjacency lists in a fancier data structure such as a hash table (Chapter 7) in order to allow for fast queries to see if an edge (i, j) is present, as well as fast insertion and deletion of edges.

Representing a Tree in Memory. Since a free tree on n nodes contains only $n-1$ edges, it is typically represented just like any other sparse graph, using adjacency lists. In the event that our tree is rooted, we can augment our adjacency lists to indicate which neighbor is the parent and which are the children. Another common way to store a rooted tree in memory is shown in Figure 2.2(c), where each node maintains a pointer to its parent, first child, and previous and next siblings. Although the children of a node are stored in a particular order, it depends on our application as to whether or not this ordering is significant. *Binary* trees are particularly common among rooted trees, where each node has at most two children, each designated as a left or right child. A node can have no children, only a left child, only a right child, or both. We usually store a binary tree as shown in Figure 2.2(d), where each node points to its parent, left child, and right child.

We will eventually need quite a bit more graph terminology, but we will postpone introducing it until it is needed, when we study graphs in much greater detail.

2.2 Doing the Math

Many of our formal mathematical arguments are structured as either proofs by *induction* or by *contradiction*. We assume the reader is reasonably well-versed in both techniques. At the beginning of the next chapter, we will see how induction is commonly used to prove an algorithm is correct and analyze its running time.

Problem 10 (Proof Practice). This problem provides some fun and challenging "warm up" exercises to re-familiarize the reader with proofs by induction and by contradiction.

(a) Prove by induction that any integer of the form $4^k - 1$ must be a multiple of 3. [Solution]

(b) Prove by induction *Kraft's inequality*, which states that $\sum_i 2^{-d_i} \leq 1$ in any binary tree, where the sum is over every leaf i, and d_i is depth of leaf i (the root has depth zero). [Solution]

(c) Given a set of n points in the 2D plane, a line is *odd* if it contains an odd number of points from the set, and *even* otherwise. Argue by contradiction that among all vertical and horizontal lines, there cannot be a single unique odd line. [Solution]

(d) A *separator* in a connected graph is a set of nodes whose removal breaks the graph into two or more different connected components. Prove by contradiction that every connected graph on n nodes must have either a separator containing at least $\sqrt{n}$ nodes or a simple path containing at least $\sqrt{n}$ nodes. [Solution]

(e) In a group of n people, suppose each person has at least $n/2$ friends. Please prove by contradiction that it is possible to seat everyone around a circular table so that all adjacent pairs are friends. [Solution]

In the rest of this section, we highlight several techniques, tricks, and insights that can be helpful for the mathematical analysis of an algorithm.

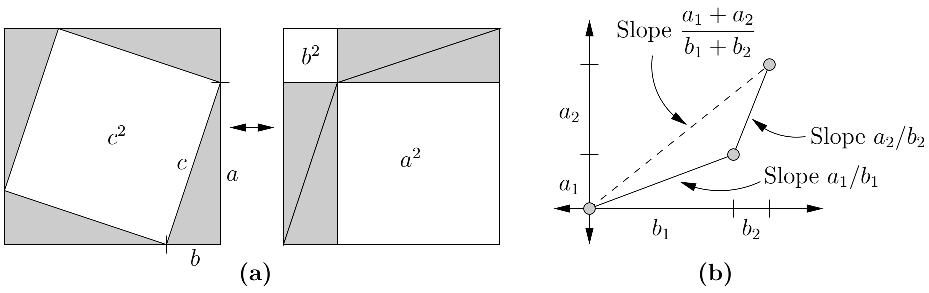


FIGURE 2.3: (a) A well-known proof by picture of the Pythagorean Theorem $a^2+b^2=c^2$, where the same area occupied by a square of side length c on the left is occupied by two squares of side lengths a and b on the right, and (b) proof by picture that the mediant of two fractions (the slope of the dashed line segment) lies between these two fractions (the slopes of the solid line segments).

Proof by Picture. Our first recommendation is to express mathematical ideas visually if at all possible, since this can be especially helpful for developing intuition for a mathematical concept. As the saying goes, "a picture is worth a thousand words". If we can translate a mathematical problem into a picture, this often provides useful structural insight, and sometimes even leads to an elegant proof. For example, in Figure 2.3(a) we see a famous visual proof of the Pythagorean Theorem, which says that the side lengths a, b, and c (the hypotenuse) of a right triangle satisfy $a^2+b^2=c^2$. The fact that our equation involves squared terms suggests that we should draw squares of side lengths a, b, and c, so we can relate their areas. In Figure 2.3(b), we see a simple proof by picture that the *mediant* of two fractions $a_1/b_1 < a_2/b_2$, defined as $(a_1+a_2)/(b_1+b_2)$, lies between the two original fractions in value (assuming the a's and b's are all positive). Here, since our problem involves ratios, it is natural to visualize these as slopes of lines.

Factorials and Stirling's Approximation. If complicated mathematical expressions arise in our analysis, it can be helpful to know how to bound certain problematic terms. For example, factorials appear often in problems such as sorting, since there are $n!$ ways to permute n elements. We often try to bound unwanted $n!$ terms with something more manageable, for example $n! \geq 2^n$ (for $n \geq 4$) or $n! \leq n^n$. However, both of these bounds are relatively weak as n grows large, since $2^n = o(n!)$ and $n! = o(n^n)$. For a better bound,

$$\log n! = \Theta(n \log n).$$

This is easy to show, since $n!$ is between $(n/2)^{n/2}$ and n^n, with the log of both being $\Theta(n \log n)$. *Stirling's approximation* says that $n!$ behaves asymptotically like $\sqrt{2\pi n}(n/e)^n$, and also provides useful bounds on the binomial coefficient $\binom{n}{k}$:

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k.$$

Many variants of Stirling's approximation exist that can provide stronger bounds on $n!$. However, these are rarely necessary for algorithmic analysis.

Convenient Exponential Bounds. A commonly-used bound involving e^x is

$$1 + x \leq e^x.$$

Looking at the graphs of $1 + x$ and e^x, we see that this inequality is only tight near $x = 0$, where equality is attained. For a stronger bound, we could take more terms from the Taylor expansion $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$ for a higher-order polynomial bound like $1 + x + x^2/2 \leq e^x$ (for $x \geq 0$). However, the simpler linear bound is usually all we need.

A useful generalization of the bound above states that for $x \geq -1$ and $y \geq 1$,

$$1 + xy \leq (1 + x)^y \leq e^{xy},$$

The right-hand bound follows directly from $1 + x \leq e^x$, and we will prove the left-hand bound two different ways over the next few pages.

To illustrate a prototypical use of this bound, many algorithms follow an "iterative refinement" strategy where they start with a sub-optimal solution and repeatedly improve it over a series of iterations until it finally becomes optimal. If we start with a solution of value 1 and make only *additive* improvements in constant increments, then our algorithm could take $\Theta(V)$ total iterations, where V is the value of an optimal solution. However, if we make *geometric* improvements (increasing the value by some constant multiplicative factor — say at least 1%), then our solution doubles every 100 iterations, since $(1 + 1/100)^{100} \geq 2$. Therefore, it only takes at most $100 \log V = O(\log V)$ iterations to reach an optimal solution. You may have seen similar uses of this bound when computing the compound interest on an investment. For example, if you earn 5% interest per year, then your investment will double at least every 20 years since $2 \leq (1 + 0.05)^{20} \leq e$ (the actual amount, approximately 2.6533, is much closer to $e \approx 2.718$ than 2, since $(1+1/n)^n$ converges to e as n grows large).

As another example, if each of n iterations of an algorithm fails independently with probability at most $1/(2n)$, then the probability the entire algorithm succeeds is at least $(1 - 1/(2n))^n \geq 1/2$ (we will see another way to reach this conclusion later when we study the "union bound" in probability theory).

The Harmonic Series. The *harmonic* series,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n},$$

appears in many of our analyses. It is useful to know that the sum of the first n terms of the harmonic series behaves like $\ln n$:

$$H_n \in [\ln n, 1 + \ln n].$$

The standard [proof] of this fact uses the clever trick of bounding a discrete summation using a continuous function. You will find many algorithms with running times containing log terms on account of the fact that $H_n = \Theta(\log n)$.

Proving Equalities Using a Pair of Inequalities. To show that $A = B$, we often prove separately that $A \leq B$ and that $B \leq A$. Although simple, this approach is used countless times in mathematical proofs. On a related note, we often show that two statements A and B are equivalent by proving separately that A implies

B, and that B implies A, and we can show that two subsets A and B are equal by proving separately that $A \subseteq B$ and that $B \subseteq A$.

Proving Inequalities by Minimization or Maximization. Suppose we want to show that $f(n) \geq g(n)$, otherwise written $f(n) - g(n) \geq 0$. If f and g happen to be easy to minimize (say, if they are continuous functions with easily computable derivatives), we can prove that $f(n) - g(n) \geq 0$ by showing that the minimum possible value of $f(n) - g(n)$ is no smaller than 0. For example, recall our earlier inequality that $1 + xy \leq (1 + x)^y$ for $x \geq -1$ and $y \geq 1$. A reasonably easy way to prove this (after having taken a multi-variable calculus course) is by showing that the function $f(x, y) = (1 + x)^y - (1 + xy)$ attains a minimum value of 0 (at $y = 1$), if we minimize over all possible choices for $x \geq -1$ and $y \geq 1$.

Minimum, Average, and Maximum. For any set of numbers $a_1 \ldots a_n$, the average value lies between the minimum and the maximum:

$$\min_{i=1\ldots n} a_i \leq \frac{1}{n}\sum_{i=1}^{n} a_i \leq \max_{i=1\ldots n} a_i.$$

This obvious yet important fact is used again and again in algorithmic analyses.

Creative Ways to Add. Many of our analyses add things up in clever ways, such as by reversing the order of a double summation (e.g., adding a table column-by-column instead of row-by-row). For example, suppose f_i denotes the number of distinct integer factors of the integer i, and that we want to know the sum $f_1 + \ldots + f_n$. Letting the "indicator function" $[x|i]$ represent the value 1 if x divides i, and 0 otherwise, an effective way to estimate the sum is

$$\sum_{i=1}^{n} f_i = \sum_{i=1}^{n}\sum_{x=1}^{n}[x|i] = \sum_{x=1}^{n}\sum_{i=1}^{n}[x|i] \approx \sum_{x=1}^{n} n/x = nH_n \approx n \ln n.$$

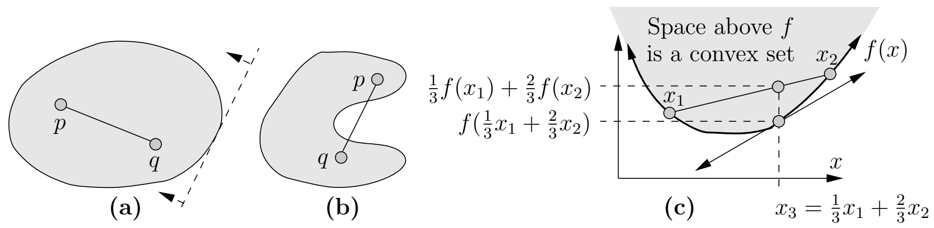
This is related to a technique called *double counting*, where we add up the same quantity two different ways to show that two expressions are equal. For example, if we sum the degrees of all nodes in a graph, we count each endpoint of each edge once. If we count all the m edges and then multiply by two, we also count each endpoint of each edge once (since each edge has two endpoints). Hence, the sum of degrees in any graph is $2m$.

Problem 11 (Nested Set Enumeration). Let S_0 be a set of n elements. It is well known that if we step through every subset $S_1 \subseteq S_0$, we will visit exactly 2^n different subsets S_1. But what if for each subset S_1 we also visit all of its subsets?

```
For every subset $S_1 \subseteq S_0$,
  For every subset $S_2 \subseteq S_1$,
    Increment counter
```

See if you can argue via a clever counting approach that the inner loop executes exactly 3^n times, and that this number becomes $(k + 1)^n$ if we generalize the code to include k nested loops. [Solution]

You probably know that summing an arithmetic series $a + (a + 1) + (a + 2) + \ldots + b$ involves taking the average term value $(a+b)/2$ times the number of terms $(b-a+1)$.



p
q
(a)
p
q
(b)
$\frac{1}{3}f(x_1) + \frac{2}{3}f(x_2)$
$f(\frac{1}{3}x_1 + \frac{2}{3}x_2)$
Space above f
is a convex set
x_1
x_2
$f(x)$
x
(c)
$x_3 = \frac{1}{3}x_1 + \frac{2}{3}x_2$

FIGURE 2.4: In (a) a convex set, any line segment connecting two points p and q in the set lies completely within the set, and also any "supporting" hyperplane tangent to the set contains the entire set on one of its sides. A non-convex set is shown in (b). As shown in (c), if f is a convex function, then the space above f is a convex set, so the line segment connecting the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ lies above f, and any tangent hyperplane (drawn here tangent at the point $(x_3, f(x_3)))$ lies below f.

For example,

$$1 + 2 + 3 + \ldots + n = n(n + 1)/2 = \Theta(n^2).$$

Series of all shapes and sizes play an important role in many of our proofs, so it is good to know how to add them all up. For example we will often encounter the geometric series

$$1 + x + x^2 + x^3 + \ldots + x^n = \frac{1 - x^{n+1}}{1 - x},$$

the infinite geometric series

$$1 + x + x^2 + x^3 + \ldots = \frac{1}{1 - x} \quad \text{(for } |x| < 1\text{)},$$

and variants of the hybrid arithmetic-geometric series

$$1 + 2x + 3x^2 + 4x^3 + \ldots = \frac{1}{(1 - x)^2} \quad \text{(for } |x| < 1\text{)}.$$

Although it is good to know formulas for adding up different types of series, it is perhaps even better to know how to re-derive these sorts of formulas quickly from scratch when needed. [Useful tricks for summing series]

Exploiting Convexity or Concavity. The reader should know what it means for a set or function to be *convex*. As shown in Figure 2.4, a set is convex if a line segment connecting any two points in the set lies entirely within the set. Intuitively, convex sets do not have any "indentations". A function is convex if the space above the function is a convex set, and *concave* if the space below the function is a convex set. Convex functions curve upward and form "bowl" shapes, while concave functions curve downward and form "hill" shapes. Convexity is particularly important in the study of optimization, since convex functions and convex constraints give us particularly well-behaved optimization problems. We will discuss convexity in this setting in far greater mathematical detail in Chapter 14.

A line segment connecting two points on a convex function will over-estimate the function, and a tangent line (plane or hyperplane, in higher dimensions) will under-estimate the function; the reverse is true for concave functions. This is useful since it allows us to bound parts of a convex or concave function with a much simpler linear function. For example, recall the inequality $1 + xy \leq (1 + x)^y$ (for $x \geq -1$ and $y \geq 1$). This follows immediately from the fact that $f(x) = (1 + x)^y$ is convex for $x \geq -1$ and $y \geq 1$, and $1+xy$ is the equation of the tangent line to this function at $x = 0$. The ability to bound a convex or concave function by a linear function is beautifully generalized by *Jensen's inequality*: given n numbers $x_1 \ldots x_n$ associated with nonnegative weights $\lambda_1 \ldots \lambda_n$ (where $\sum \lambda_i = 1$), then

$$f(\lambda_1 x_1 + \ldots + \lambda_n x_n) \leq \lambda_1 f(x_1) + \ldots + \lambda_n f(x_n)$$

if f is convex, and the reverse inequality holds if f is concave. For example, in Figure 2.4(c), you can see that $f(\frac{1}{3}x_1 + \frac{2}{3}x_2) \leq \frac{1}{3}f(x_1) + \frac{2}{3}f(x_2)$.

Problem 12 (Relating Arithmetic and Geometric Means). Using the fact that log is a concave monotonically increasing function, give a simple proof of the useful property that the geometric mean of n numbers $(x_1 x_2 \ldots x_n)^{1/n}$ is at most their arithmetic mean $(x_1 + x_2 + \ldots + x_n)/n$. [Solution]

Later in this chapter, we will learn about the expected value $\mathbf{E}[X]$ of a "random variable" X. Since expected value is nothing more than a weighted sum of values X can take (weighted by the corresponding probabilities of X taking those values), Jensen's inequality tells us the useful property that $f(\mathbf{E}[X]) \leq \mathbf{E}[f(X)]$ if f is convex, with the reverse being true if f is concave.

2.3 Recurrences

Many algorithms break a large problem into smaller subproblems of the same form, recursively solve these, and then somehow recombine their solutions to obtain a solution for the original problem. It is often convenient to express the running time of these recursive algorithms using recursively-defined expressions called *recurrences*.

Here is a simple example. Given a numeric array $A[1 \ldots n]$, the *maximum value subarray* problem asks us to find a contiguous subarray $A[i \ldots j]$ having maximum sum (we assume the array contains some negative numbers, since otherwise the problem is trivial). The obvious $\Theta(n^3)$ time "brute force" solution is to loop over all $\binom{n}{2} = \Theta(n^2)$ possible subarrays $A[i \ldots j]$, and to sum each one in $O(n)$ time. We can easily improve this to $\Theta(n^2)$ total time using *prefix sums*: first precompute an array $P[1 \ldots n]$ where $P[j] = A[1] + \ldots + A[j]$. This is easy to do in $\Theta(n)$ time by scanning through A while maintaining a running sum. After computing P, the sum of $A[i \ldots j]$ is now given in constant time by $P[j] - P[i-1]$. As it turns out, one can do much better still. Here, we describe a recursive algorithm running in $\Theta(n \log n)$ time, and in Chapter 11 we will learn an even simpler $\Theta(n)$ algorithm based on the technique of dynamic programming!

For our recursive solution, think of $A[1 \ldots n]$ as two half-sized arrays $L = A[1 \ldots n/2]$ and $R = A[n/2 + 1 \ldots n]$. The answer is then given by the best of three solutions:

- The best solution entirely within L, which we can find by recursively applying our algorithm to L,

- The best solution entirely within R, which we can find by recursively applying our algorithm to R, and
- The best solution spanning both L and R, which we find by taking the best suffix of L added to the best prefix of R. These are both easy to compute in $\Theta(n)$ time by computing suffix sums of L (scanning backward through L keeping a running sum), and also prefix sums of R (scanning forward through R keeping a running sum).

In total, our algorithm makes two recursive calls to subproblems of size $n/2$ and spends $\Theta(n)$ additional time outside these calls. If $T(n)$ denotes the running time of our algorithm on an input of size n, we can therefore write the following recursive formula for $T(n)$:

$$T(n) = 2T(n/2) + \Theta(n).$$

As a base case, since a problem of constant size can be solved in constant time we have $T(n) = O(1)$ for $n = O(1)$.

In this section, we learn how to solve common types of recurrences to produce an explicit (non-recursive) formula for $T(n)$. For instance, the solution to the example above is $T(n) = \Theta(n \log n)$.

2.3.1 "Divide and Conquer" Recurrences

If our recursive algorithm breaks a problem of size n into "multiplicatively smaller" subproblems (e.g., 2 subproblems of size $n/2$, or maybe 5 subproblems of size $n/3$ and 6 of size $n/10$), we call it a *divide and conquer* algorithm. The divide and conquer strategy (decompose large problem into subproblems, recursively solve subproblems, recombine subproblem solutions to obtain solution to initial problem) is a widely-applicable and powerful algorithm design methodology. We will continue to discuss this technique in the next chapter when we introduce several algorithms for sorting based on divide and conquer. Letting $T(n)$ denote the running time required to solve a problem of size n, divide and conquer algorithms typically give us recurrences of the form

$$T(n) = a_1T(n/b_1) + a_2T(n/b_2) + \ldots + a_kT(n/b_k) + f(n).$$

That is, we decompose a problem of size n into a_1 subproblems of size n/b_1, a_2 subproblems of size n/b_2, and so on. The function $f(n)$ represents the time spent outside our recursive calls, both initially decomposing the original problem and in later recombining the solutions to our recursively-solved subproblems.

In the discussion that follows, we assume we are dealing with recurrences like the one above where the a's and b's are constants, and where $f(n)$ is polynomially-bounded. We initially consider functions of the form $f(n) = \Theta(n^\alpha)$, and later we show how to handle slightly more complicated functions like $f(n) = \Theta(n^\alpha \log^\beta n)$ with an added logarithm term. This is broad enough to cover most common divide and conquer algorithms.

Useful Simplifications. There are several easy ways to simplify a frightening divide and conquer recurrence like

$$T(n) = 2T(n/6 + \sqrt{n} + 1) + 3T(\lfloor n/8 \rfloor) + 17n^3 \log n - 3n^2 \log^4 n + 5$$

so that it takes the form above. First, we can replace the non-recursive part $(17n^3 \log n - 3n^2 \log^4 n + 5)$ with an asymptotic placeholder $(\Theta(n^3 \log n))$, since leading constants and lower-order terms in this part won't affect the asymptotic solution. In fact, we can even drop the $\Theta()$ entirely:

$$T(n) = 2T(n/6 + \sqrt{n} + 1) + 3T(\lfloor n/8 \rfloor) + n^3 \log n.$$

Small additive offsets like "$+\sqrt{n} + 1$" inside recursive calls also have no impact on the final asymptotic solution, and can safely be ignored. In fact, this is surprisingly true even for larger additive offsets of magnitude up to $O(n/\log^2 n)$. Accordingly, we can disregard floor and ceiling functions, which are just additive offsets of at most one. After applying these changes, we have the much simpler recurrence

$$T(n) = 2T(n/6) + 3T(n/8) + n^3 \log n,$$

whose asymptotic solution is the same as that of the original recurrence.

Solving Recurrences by Expansion. A general approach for solving any recurrence is to expand it out algebraically and look for patterns. For example, take the simple recurrence $T(n) = 2T(n/2) + n$ for our maximum value subarray problem:

$$\begin{array}{rcl}
T(n) & = & 2T(n/2) + n \\
& = & 2[2T(n/4) + n/2] + n \quad \text{(expanding } T(n/2)\text{)} \\
& = & 4T(n/4) + n + n \\
& = & 4[2T(n/8) + n/4] + n + n \quad \text{(expanding } T(n/4)\text{)} \\
& = & 8T(n/8) + n + n + n \\
& \vdots & \\
& = & \underbrace{n\,T(\le 1)}_{O(1)} + \underbrace{n + n + \ldots + n}_{\log n \text{ terms}} \\
& = & \Theta(n \log n).
\end{array}$$

Tree Expansions. Suppose we arrange the terms resulting from algebraic expansion in their natural hierarchical layout as a tree, then add everything up level by level, as shown in Figure 2.5. It turns out that the level sums always give a geometric series, which we can sum with minimal effort, owing to the following insight:

- A decreasing geometric series (even an infinite one), where each term decays by a constant factor, behaves asymptotically like its first term. For example, $n^2 + (3/4)n^2 + (3/4)^2 n^2 + \ldots = \Theta(n^2)$.
- By symmetry, an increasing geometric series behaves asymptotically just like its last term.

We can tell if the geometric series obtained from our recursion tree is decreasing, increasing, or unchanging after expanding the tree out by only a single level (which is often possible to do mentally, after some practice). Knowing the nature of the series is all we need to solve the recurrence. For example, consider a recurrence with the simple form

$$T(n) = aT(n/b) + n^\alpha.$$

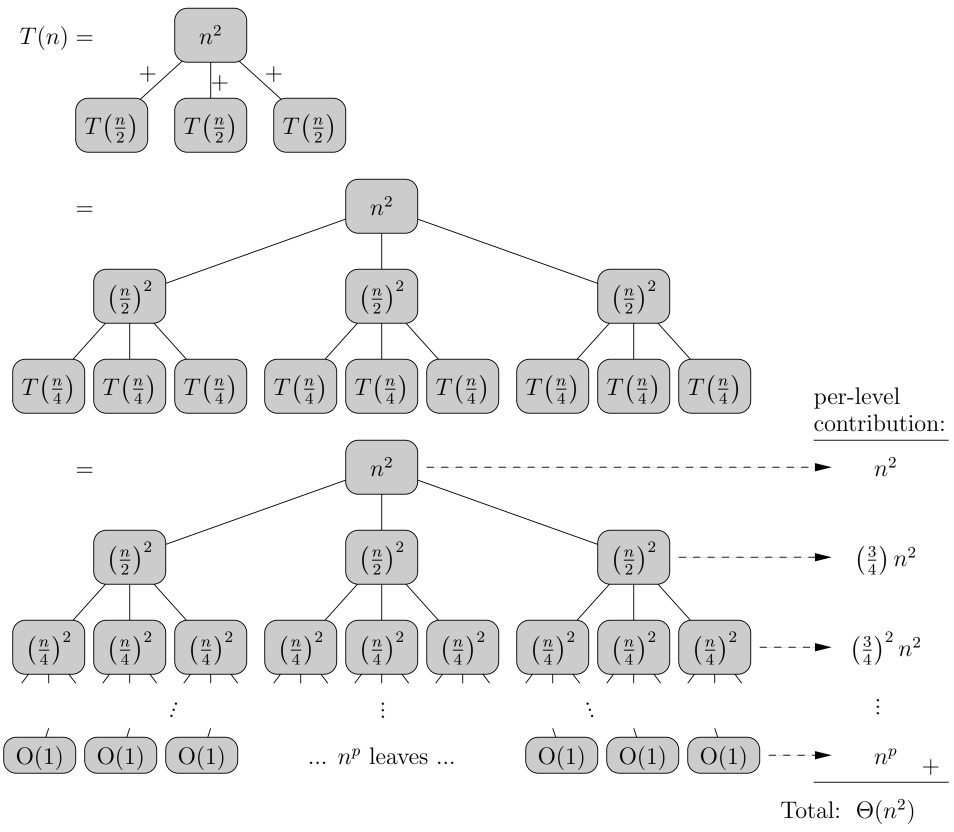


FIGURE 2.5: Expanding the recurrence $T(n) = 3T(n/2) + n^2$ out as a tree of height $\log_2 n$ with $3^{\log_2 n} = n^{\log_2 3} = n^p$ leaves (if n is not a power of 2, then these hold asymptotically rather than exactly). The sum of each level forms a decreasing geometric sequence which is asymptotically dominated by the contribution of the root, so the solution is $T(n) = \Theta(n^2)$.

If our geometric series decreases as we scan down the tree, then the root contribution n^α is dominant, so the recurrence solves to $T(n) = \Theta(n^\alpha)$. If the series remains unchanging, then each of the $\log_b n$ levels contributes n^α, so the answer is $T(n) = \Theta(n^\alpha \log n)$. Finally, if the series increases, then the contribution from the leaves will be dominant. Each leaf contributes $O(1)$, and the number of leaves in a tree with depth $\log_b n$ and branching factor a is $a^{\log_b n} = n^{\log_b a}$. The solution is therefore $T(n) = \Theta(n^p)$, where $p = \log_b a$.

Stated more concisely, if $T(n) = aT(n/b) + n^\alpha$ (with $T(n) = O(1)$ for $n = O(1)$ as a standard base case), then

$$T(n) = \begin{cases} \Theta(n^\alpha) & \text{if } \alpha > p \quad \text{(decreasing series)} \\ \Theta(n^\alpha \log n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\ \Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} \end{cases}$$

where $p = \log_b a$. We have simplified this formula a bit by noting that we can tell if the geometric series is increasing, unchanging, or decreasing by simply comparing α and p. If you forget this, however, you can always determine the appropriate case by expanding out the tree by one level. The simple formula above is sufficient to solve the vast majority of the recurrences we will ever encounter.

Multiple-Term Recurrences. Consider now a recurrence of the form

$$T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + \ldots + a_k T(n/b_k) + n^\alpha.$$

with multiple recursive terms. We can solve this exactly as before, by expanding it into a tree and summing a geometric series. The only difference is that the tree is no longer "level" at the bottom, since n decreases at different rates down different branches. Accordingly, the level-by-level contribution starts out as a geometric series but then behaves slightly differently toward the bottom of the tree once some branches start ending. However, this only ends up changing the answer in the third case above (an increasing series), where the contribution from the leaves is dominant:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \text{if } \alpha > p \quad \text{(decreasing series)} \\ \Theta(n^\alpha \log n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\ \Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} \end{cases}$$

where p is the solution to $a_1/b_1^p + a_2/b_2^p + \ldots + a_k/b_k^p = 1$. [Proof]

For a single-term recurrence, this more complicated formula boils down to $a/b^p = 1$, the solution of which is $p = \log_b a$ as before. In the multiple-term case, however, it may not be possible to find an analytic solution for p. Therefore, while for a single-term recurrence you can easily determine the appropriate case above by directly comparing α and p, this is now more difficult, since computing the exact value of p may not be possible. There are two ways around this. The simplest is probably to expand the tree by a single level as before, since this will still reveal the nature of the geometric series. Alternatively, let $g(x) = a_1/b_1^x + a_2/b_2^x + \ldots + a_k/b_k^x$. Since g is a decreasing function and $g(p) = 1$, we know that $\alpha > p$ if $g(\alpha) < 1$ and that $\alpha < p$ if $g(\alpha) > 1$.

Extra Logarithmic Factors. Consider finally a recurrence of the form

$$T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + \ldots + a_k T(n/b_k) + n^\alpha \log^\beta n,$$

where $\beta \geq 0$. Here, we can initially ignore the extra $\log^\beta n$ term while resolving the nature of our geometric series. The term then re-appears in the solution unless we are in the increasing case where the leaves dominate:

$$T(n) = \begin{cases} \Theta(n^\alpha \log^\beta n) & \text{if } \alpha > p \quad \text{(decreasing series)} \\ \Theta(n^\alpha \log^{\beta+1} n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\ \Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} \end{cases}$$

where p is again the solution to $a_1/b_1^p + a_2/b_2^p + \ldots + a_k/b_k^p = 1$.

Problem 13 (Practice Solving Recurrences). Practice makes perfect. See if you can solve the following divide and conquer recurrences with a minimal amount of calculation. [Solutions]

(a) $T(n) = 6T(n/3) + \Theta(n^2)$.

(b) $T(n) = 5T(n/4) + \Theta(n)$.

(c) $T(n) = 3T(n/2) + \Theta(n\sqrt{n})$.

(d) $T(n) = 4T(n/2) + \Theta(n^2)$.

(e) $T(n) = T(n/6) + T(n/2) + T(n/3) + \Theta(n \log^2 n)$.

(f) $T(n) = 4T(\lfloor n/3 \rfloor) + 5T(n/2 + \log^3 n) + 2T(2n/3 + 7) + n^{62} \log^{37} n + n^{19} - 50$.

(g) $T(n) = T(\sqrt{n}) + 1$. Here, try making a change of variables (to $m = \log_2 n$) to transform this recurrence into a more familiar form. See also problem 8.

2.3.2 Linear Recurrences

In a *linear recurrence*, such as $T(n) = T(n-1) + \Theta(n)$, each recursive subproblem is "additively" smaller than the original, rather than "multiplicatively" smaller as with the divide and conquer recurrence. Linear recurrences are perhaps more often found in other fields (e.g., signal processing). In algorithms, they tend to arise in conjunction with iterative (rather than recursive) algorithms that sequentially construct a solution one step at a time. For example, the recurrence above could describe an algorithm that solves a problem of size n by first solving a subproblem of size $n-1$ (say, with all but one of its input elements present) and then doing $\Theta(n)$ work to properly adjust for the presence of the final input element.

The general form of a linear recurrence is

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \ldots + a_k T(n-k) + f(n).$$

For example, we can describe the famous sequence of *Fibonacci numbers* using the linear recurrence $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$ as base cases. We need k base cases if the recurrence contains k recursive terms.

While linear recurrences can certainly be used to model and solve for algorithmic running times, we tend not to use these in this book, because it is often easier to analyze iterative algorithms in a more direct fashion. For those who are interested, however, we include here a [short discussion] of how to solve general linear recurrences (beyond just using algebraic expansion, which often works quite effectively).

Fibonacci Numbers. Solving a linear recurrence is one of many ways to obtain an explicit formula for the nth Fibonacci number, which surprisingly always outputs an integer value:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right].$$

Since $|(1-\sqrt{5})/2| < 1$, this term decays away as n grows large and the contribution from the $(1+\sqrt{5})/2$ term dominates. Therefore, the Fibonacci numbers grow at an exponential rate:

$$F_n = \Theta(\phi^n), \text{ where } \phi = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

This will be useful to know later for the analysis of several data structures, such as AVL trees and Fibonacci heaps. The number ϕ is known as the *golden ratio*, and like π and e it plays an important role in the answer to several fundamental mathematical questions.

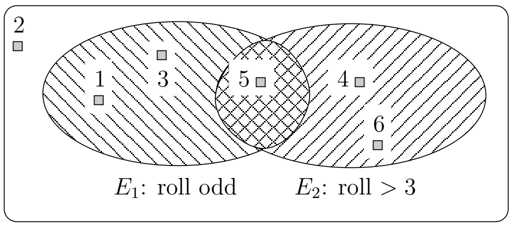


FIGURE 2.6: A Venn diagram showing two events E_1 and E_2 after rolling a 6-sided die.

2.4 Probability

Randomization often provides a way to design algorithms that are fast, simple, and elegant. To analyze them, however, we need to equip ourselves with tools from probability theory. This section summarizes most of the tools we will need.

2.4.1 The Basics

A probability space is a set of all possible fundamental *outcomes* of some random trial, each with an associated nonnegative *probability*, where these probabilities all sum to 1. An *event* E is a set of outcomes. Its probability, $\mathbf{Pr}[E]$, is the sum of the probabilities of these outcomes. For example, if our random trial involves rolling two 6-sided dice, there are 36 different possible outcomes, $(1,1),(1,2),(1,3),\ldots,(6,6)$, each occurring with probability 1/36. The event that both dice sum to 4 includes outcomes $(1,3)$, $(2,2)$, and $(3,1)$, and therefore has probability $3/36 = 1/12$. The natural way to informally interpret this probability is that if we repeatedly perform random trials (each involving a roll of two dice), on average we expect to see a sum of 4 once every 12 trials.

If all outcomes in a random experiment are equally likely, then the probability of an event E is just the number of outcomes in E divided by the total number of possible outcomes. This observation usually reduces a probability question to a combinatorics question. For example, what is the probability of seeing exactly 7 heads in 10 coin flips? There are 2^{10} different sequences of coin flips (these are our different outcomes), and $\binom{10}{7}$ of them belong to the event that we see exactly seven heads, so the probability of our event is $\binom{10}{7}/2^{10} \approx 0.117$.

Since the probabilities of all possible outcomes sum to 1, we know that

$$\mathbf{Pr}[E] = 1 - \mathbf{Pr}[\overline{E}]$$

where $\overline{E}$, the *complement* of E, is the set of all outcomes not in E (in other words, the event that E does not occur). It is often easier to directly compute $\mathbf{Pr}[\overline{E}]$ rather than $\mathbf{Pr}[E]$; for instance, the probability we see heads at least once in 10 coin flips is $1 - 1/2^{10}$, where $1/2^{10}$ is the probability of the complement, that we see all tails.

Since events are nothing more than sets of outcomes, we can use set union and intersection to express relationships between them. If E_1 and E_2 are two events, then $E_1 \cup E_2$ is the event that either one or both events occur, and $E_1 \cap E_2$ is the event that both occur.

The Union Bound. The probability that either E_1 or E_2 (or both) occurs is

$$\mathbf{Pr}[E_1 \cup E_2] = \mathbf{Pr}[E_1] + \mathbf{Pr}[E_2] - \mathbf{Pr}[E_1 \cap E_2].$$

This is obvious from the *Venn diagram* pictured in Figure 2.6, since $\mathbf{Pr}[E_1]+\mathbf{Pr}[E_2]$ counts all the outcomes in E_1 and E_2, but counts the outcomes in $E_1 \cap E_2$ twice — so we compensate by subtracting out $\mathbf{Pr}[E_1 \cap E_2]$. A more general form this idea is the *inclusion-exclusion* principle, where we compute $\mathbf{Pr}[E_1 \cup \ldots \cup E_k]$ for a union of k events by adding their individual probabilities, subtracting the probabilities of all intersecting pairs of events, adding the probabilities of all intersecting triples of events, and so on in an alternating fashion. If we only want a rough upper bound, however, we can say that

$$\mathbf{Pr}[E_1 \cup E_2 \cup \ldots \cup E_k] \leq \mathbf{Pr}[E_1] + \mathbf{Pr}[E_2] + \ldots + \mathbf{Pr}[E_k],$$

where equality holds only if all events are disjoint ($E_i \cap E_j = \emptyset$ for all pairs of events). This is known either as the *union bound* or as *Boole's inequality*, and it is an extremely useful tool in many of our analyses. It follows easily from the fact that every outcome in $E_1 \cup \ldots \cup E_k$ has its probability counted exactly once on the left-hand side, but potentially several times on the right-hand side, depending on the number of events in which the outcome is contained.

The union bound tells us that the failure probability of a complex system is at most the sum of the failure probabilities of its individual parts. A machine with 200 parts, each failing with probability at most 10^{-6}, has an overall probability of failure (i.e., of one or more parts failing) at most 200×10^{-6}. In the context of a randomized algorithm, suppose an algorithm spends $O(f(n))$ time on a single "generic" input element with probability at least $1 - 1/(2n)$, so it fails to process this element quickly enough with probability at most $1/(2n)$. Taking a union bound over all n elements, the probability of failure on any element is at most $1/2$, so consequently the algorithm runs in $O(nf(n))$ time with probability at least $1/2$.

Conditional Probability. The probability of event A given that event B occurs is written as

$$\mathbf{Pr}[A \,|\, B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}.$$

Otherwise written, $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A \,|\, B]\, \mathbf{Pr}[B]$[1]. Since by symmetry we also have $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[B \,|\, A]\, \mathbf{Pr}[A]$, we can easily derive *Bayes' rule*,

$$\mathbf{Pr}[A \,|\, B] = \frac{\mathbf{Pr}[B \,|\, A]\, \mathbf{Pr}[A]}{\mathbf{Pr}[B]},$$

which is useful for relating $\mathbf{Pr}[A \,|\, B]$ and $\mathbf{Pr}[B \,|\, A]$. We will use Bayes rule quite often when we study machine learning later in Chapter **??**.

Independence. Two events A and B are *independent* if knowledge about the occurrence of one event does not change the probability of occurrence of the other.

[1]For those familiar with calculus, conditional probabilities behave in much the same way as the chain rule for derivatives. For example, the derivative of $f(g(h(x)))$ is $f'(g(h(x)))g'(h(x))h'(x)$. Similarly, we can write $\mathbf{Pr}[A \cap B \cap C] = \mathbf{Pr}[A \,|\, B, C]\, \mathbf{Pr}[B \,|\, C]\, \mathbf{Pr}[C]$.

Formally, A and B are independent if and only if any of the following equivalent conditions are true:

- $\mathbf{Pr}[A \mid B] = \mathbf{Pr}[A]$,
- $\mathbf{Pr}[B \mid A] = \mathbf{Pr}[B]$, or
- $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$.

Independence is often used to simplify the probability of both A and B occurring, $\mathbf{Pr}[A \cap B]$, by replacing it with the product $\mathbf{Pr}[A]\,\mathbf{Pr}[B]$.

2.4.2 "With High Probability" Results

Monte Carlo randomized algorithms can give incorrect answers if they are unlucky. In order to persuade anyone to use them, we need to ensure that the probability of an incorrect answer is extremely small. We would like our algorithms to succeed *with high probability*, which usually means the following in the computing literature:

> We say a Monte Carlo algorithm with input size n is correct with high probability if it fails with probability at most $1/n^c$ for any constant $c > 0$ of our choosing. More formally, given any constant $c > 0$, we can set the hidden constant in our running time appropriately such that
>
> $$\mathbf{Pr}[\text{algorithm outputs incorrect answer}] \leq 1/n^c$$
>
> for sufficiently large values of n.

The interesting (and to some, confusing) aspect of this definition is the part where c is allowed to be any constant of our choosing. In order to claim a high probability bound, we must be able to reduce the failure probability to $1/n^2$, $1/n^{10}$, $1/n^{1000}$, or $1/n^c$ for any other constant c. The impact of our choice of c shows up only in the hidden constant in the running time of our algorithm, so it does not change the overall asymptotic running time.

To see that the definition above is a reasonable way to define "with high probability", observe that you might be happy with an algorithm that only fails at most 1% of the time, but other situations may call for an even more robust algorithm that fails at most 0.001% of the time. It isn't really satisfactory for us to pick any fixed constant threshold, since there is no obvious choice that would be universally acceptable. Instead, we use a threshold based on the input size, n. For example, we could say that a Monte Carlo algorithm is correct with high probability if it fails with probability at most $1/2^n$ on inputs of size n. This way, we are virtually guaranteed that large inputs are handled correctly. It is true that the bound is weaker for small inputs, but this is reasonable since with small inputs, there are only a limited number of random outcomes that can occur, and if even one of these is bad, then our overall probability of failure may be somewhat large. Just as the most important feature of running time is how it scales with problem size, the most meaningful definition of "with high probability" also scales with problem size. It turns out that an inverse-exponential error bound like $1/2^n$ is a bit too ambitious to hope to achieve with most problems, so the standard definition above requires a slightly weaker inverse-polynomial error bound. Of course, nobody will complain if your algorithm manages to satisfy an even stronger bound.

Boosting Success Probability Through Repetition. Suppose we have a Monte Carlo randomized algorithm that produces a correct answer with probability at least 1/2. As long as the algorithm has the ability to detect when it makes a mistake, we can boost its success probability from a constant guarantee like 1/2 to a high probability guarantee by simply running $O(\log n)$ independent trials of the algorithm. If we perform, say $c \log n$ trials, then our algorithm only fails if *every* individual trial fails, and this happens with probability at most $(1/2)^{c \log n} = 1/n^c$. Since we can select c to be any constant we like, this give us a high probability guarantee of success. If we want a failure probability of at most $1/n^{10}$, we choose $c = 10$. If we want a failure probability of at most $1/n^{100}$, we choose $c = 100$. In any case, we still perform only $O(\log n)$ iterations of our algorithm, with c disappearing as the hidden constant.

Problem 14 (Multicolored Marbles). You are given a jar containing n marbles. Either (i) all the marbles are blue, or (ii) half the marbles are blue and the other half are red. By examining only $O(\log n)$ marbles, how can we determine with high probability of correctness which scenario is reality? [Solution]

Problem 15 (Property Testing). The problem above is a trivial example from the domain of *property testing*, where we want to develop a very fast (typically sublinear time) randomized test that can distinguish two cases: whether an object has some property, or whether it is "far" from having the property. Objects close to having the property do not need to be handled correctly; for example, the fast randomized test from the solution of the problem above would be unable to readily distinguish between all marbles being blue, or all but one being blue. Another simple example, which we examine in this problem, is distinguishing whether an array $A[1 \ldots n]$ of distinct numbers is either (i) in sorted order, or (ii) far from being sorted, in that any increasing subsequence has length at most $(1-\varepsilon)n$ (remember that a *subsequence* is not necessarily contiguous within the array).

Our algorithm is simple: we call an array index i *consistent* if a binary search for $A[i]$ correctly reaches this element. If the array is sorted, then all indices are consistent. Our algorithm runs binary searches to test consistency of $O(\frac{1}{\varepsilon} \log n)$ independently-chosen random indices, so its total running time is $O(\frac{1}{\varepsilon} \log^2 n)$. We guess that the array is sorted if all are consistent, and we guess otherwise that it is far from being sorted. For a challenge, please argue that this algorithm distinguishes cases (i) and (ii) above with high probability. In case (i), we always get the right answer, so we focus on case (ii). Letting C be the set of all consistent indices in A, please show that C is an increasing subsequence. In case (ii) we must therefore assume $|C| < (1-\varepsilon)n$. Show that this means that our algorithm will pick some inconsistent index (and therefore get the right answer) with high probability. [Solution]

Problem 16 (Trends in Random Sequences). Suppose we take a sequence of n distinct numbers arranged in random order. We call a *trend* a contiguous subsequence that is either increasing or decreasing. Please argue that with high probability (at least $1 - 1/n^c$ for any constant c of our choosing), the length of the longest trend in our array will be $\Theta(\log n)$. Why does this result also show that in a random length-n binary string, the longest contiguous run of 0's or 1's must have length $\Theta(\log n)$ with high probability? Although this problem can be easily approached with Chernoff bounds (Section 2.4.4), you can also solve it directly by proving separate upper and lower bounds — i.e., by showing that the longest trend has length $O(\log n)$ and $\Omega(\log n)$ with high probability. [Solution]

Problem 17 (The Birthday Paradox). The birthday paradox is a famous mathematical puzzle stated as follows: assuming everyone has a birthday chosen at random, how many people must you gather together in a room before there is a reasonable chance (say, probability at least 1/2) that two of them share the same birthday? The problem is called a "paradox" because most people regard its answer, 23, as being surprisingly low. In general, the threshold where shared birthdays become likely is roughly where the number of people reaches the square root of the number of possible birthdays. In this problem, we investigate the mathematics behind the birthday paradox as well as some of its algorithmic implications. Along the way, we will have a chance to review many of the concepts we have covered so far in this chapter, such as independence, the union bound, and high probability guarantees.

(a) Suppose that we have n individuals, each with a birthday chosen independently at random from m possible days. Let D be the event that all n individuals have different birthdays. Please write an exact mathematical expression for $\mathbf{Pr}[D]$. Using bounds from earlier in this chapter, please show that (i) $\mathbf{Pr}[D] \geq 3/4$ if $n \leq \sqrt{m}/2$, and (ii) $\mathbf{Pr}[D] \leq 1/e$ if $n \geq 2\sqrt{m}$. For simplicity, please assume m is a perfect square. [Solution]

(b) Using a union bound over all $\binom{n}{2}$ pairs of individuals, please show that $\mathbf{Pr}[D] \geq 1/2$ if $n \leq \sqrt{m}$. This result will play an important role in the analysis of *hash table* data structures in Chapter 7, where we exploit the fact that if n elements are randomly mapped to a table of size n^2, there is at least a 1/2 probability that no two elements collide at the same location. [Solution]

(c) In a simple peer-to-peer network made up of m computers, it may be the case that the only way to find a particular object (e.g., a file) is to query computers one-by-one until we find a computer hosting the object we seek. In order to speed this process up, suppose we take each object of interest and replicate it on $\sqrt{m}$ computers throughout the network, chosen arbitrarily. To search for an object, we probe $O(\sqrt{m}\log m)$ randomly-chosen computers and succeed if one of them has the object in question. Please argue that this procedure guarantees a high probability of success. [Solution]

(d) In a lake with m fish, show how to estimate m to within some constant factor with high probability by catching, marking, and releasing only $O(\sqrt{m}\log m)$ random fish. The result of problem 32 may help. [Solution]

High Probability Guarantees for Las Vegas Algorithms. With Las Vegas randomized algorithms, the output is always correct but the running time can vary depending on luck. It may be hard to convince someone to use the algorithm unless we can persuade them that it is unlikely that the running time will be significantly larger than some target running time. In other words, we would like an algorithm with a running time guarantee that holds *with high probability.*

We say a Las Vegas algorithm with input size n runs in $O(f(n))$ time *with high probability* if it fails to run in $O(f(n))$ time with probability at most $1/n^c$, where $c > 0$ is any constant of our choosing. More formally, given any any constant $c > 0$, we can find a constant k such that

$$\mathbf{Pr}[\text{running time exceeds } kf(n)] \leq 1/n^c$$

for sufficiently large values of n.

We define a "high probability" running time bound for a Las Vegas algorithm in a very similar fashion to that of a Monte Carlo algorithm, in terms of a polynomially-small failure probability that scales with problem size[2].

For a simple example, consider using binary search within a sorted array $A[1 \dots n]$. In each iteration, we examine some "pivot" element $A[j]$ and then narrow our remaining search to either $A[1 \dots j-1]$ or $A[j+1 \dots n]$. It is natural to choose $j = n/2$, since this lets us reduce the size of the problem under consideration by a factor of 2 in each iteration and achieve a worst-case running time of $O(\log n)$. However, what if we instead choose j uniformly at random from $1 \dots n$? The resulting algorithm, which we call *randomized binary search*, is perhaps closer to what happens in practice when we want to look up a word in a dictionary. Since it is hard to open the dictionary to the page containing the exact middle word, we might instead just open to a random page. As you might suspect, randomized binary search runs in $O(\log n)$ time with high probability. If we want a guarantee that its running time will be $O(\log n)$ with probability at least $1 - 1/n^{10}$, we need only select an appropriately large hidden constant for the $O(\log n)$ running time. If we want a guarantee of at least $1 - 1/n^{100}$, we select a larger hidden constant. Regardless of the constant we choose, the running time bound is still of the form $O(\log n)$.

Randomized Reduction. In a few pages, we will learn how to use *Chernoff bounds* to prove high probability guarantees. Chernoff bounds are extremely useful and powerful tools in the analysis of randomized algorithms, but they are often somewhat complicated to apply. In order to make our proofs as easy as possible, we can derive from the Chernoff bound the following:

The Randomized Reduction Lemma. Suppose we have a problem with input size n, and we run an algorithm that in each iteration effectively reduces the size of the problem under consideration by some amount. If every iteration of our algorithm reduces the size of a problem to at most *some constant fraction of its original size* $q \in [0, 1)$ with at least *some constant probability* $p \in (0, 1]$ (where the probability at each step is independent of the others), then the algorithm requires only $O(\log n)$ iterations with high probability.

The randomized reduction lemma[3] is quite natural since it builds upon a very familiar principle: if we start with a problem of size n and apply an algorithm that, in each iteration, reduces the effective problem size to at most some constant fraction $q \in [0, 1)$ of its original size, then the algorithm will perform $O(\log n)$ iterations. The prototypical example of this principle is binary search, with $q = 1/2$. All the lemma above says is that we achieve the same logarithmic performance (with high probability) as long as there is a good chance of problem size reduction in each iteration.

[2]In fact, it is actually meaningless to define "with high probability" in this case using a fixed constant threshold as a failure probability. If an algorithm has an expected running time of $O(n^2)$ time (we will say in minute what "expected" means), then we can adjust the hidden constant in the $O(\cdot)$ notation using Markov's Inequality (also discussed shortly) to change the probability that it fails to run in $O(n^2)$ time to any constant of our choosing: 1%, 0.001%, and so on.

[3]Note that this is a name you will only find here in this book, since the lemma does not seem to appear by any common name elsewhere.

Remarkably, this lemma (which we prove later) is the only tool we need in order to argue almost all of the high probability bounds in this book! For example, with randomized binary search we can easily show that each iteration reduces our problem to at most $q = 2/3$ of its original size with probability at least $p = 1/3$: let A denote the $n/3$ smallest elements in our array, let B denote the $n/3$ middle elements, and let C denote the $n/3$ largest elements. With probability $1/3$, we choose a pivot element from B and reduce our problem to at most $2/3$ of its original size, since this eliminates either A or C from consideration (depending on the comparison with the pivot). Therefore, the randomized reduction lemma immediately tells us that randomized binary search runs in $O(\log n)$ time with high probability.

Note that the randomized reduction lemma can be applied even in situations when we aren't explicitly reducing the size of a problem. For example, we could start with a problem of unit size and in each iteration expand by a constant factor with some constant probability, stopping when we reach n. We could also identify some other parameter associated with our algorithm's state that satisfies the properties of the lemma — either shrinking or expanding by some constant factor with constant probability in each iteration.

Union Bounds and High Probability Results. Our definition of "with high probability" meshes particularly well with the union bound. In particular, if we can prove that some property holds with high probability for a generic element in our input (failure probability at most $1/n^c$), then the union bound tells us that this property also holds with high probability for all n elements in our input (with failure probability at most $n \times 1/n^c = 1/n^{c-1}$, which we can still set to any level of our choosing by selecting c appropriately). This simplifies many of our proofs by allowing us to focus on proving a high probability result for only a single element or smaller subproblem, rather than for the entire input taken as a whole. For example, in the next chapter we will show that the randomized quicksort algorithm runs in $O(n \log n)$ time with high probability by first observing that it spends $O(\log n)$ work *on any single element* with high probability (using the randomized reduction lemma), and then by applying a union bound to conclude that it spends $O(\log n)$ work per element *on all of the n input elements* also with high probability.

Non-Uniform Subproblem Sizes. If we perform a randomized binary search over only a length-k subarray of a larger length-n array, the randomized reduction lemma tells us this will run in $O(\log k)$ time, but only with high probability with respect to k (i.e., with a failure bound of the form $1/k^c$). This can make it difficult to use the union bound to aggregate n high probability results for differently-sized subproblems to obtain a global high probability bound; to do this, we would need a failure bound of the form $1/n^c$ for each subproblem. Fortunately, when we later prove the randomized reduction lemma, we will show that it actually guarantees an $O(\log n)$ running time with failure probability $1/n^c$ for any $n \geq k$. We can therefore aggregate high probability results over non-uniform subproblem sizes with little trouble — a subtle, but important point.

The following problems provide good examples of the randomized reduction lemma in action (often in conjunction with the union bound).

Problem 18 (Properties of Random Permutations). This problem and the next will give us some practice using the randomized reduction lemma.

(a) Consider a random permutation of $\{1, 2, \ldots, n\}$. Please argue that the length of its longest sequential subsequence is $O(\log n)$ with high probability. A sequential subsequence is a subsequence whose elements are increasing by 1 as we move from left to right. For example, in the ordering 7, $\underline{3}$, 1, $\underline{4}$, $\underline{5}$, 2, $\underline{6}$, the elements in the longest sequential subsequence are underlined[4]. [Solution]

(b) Suppose we step through a random permutation of n distinct numbers and count the number of times we encounter an element that is the largest element seen so far. Please show that we will encounter only $O(\log n)$ such elements with high probability. As a hint, you may want to think backward instead of forward. [Solution]

(c) Consider a random permutation of $\{1, 2, \ldots, n\}$ stored in an array $A[1 \ldots n]$. In any permutation, we can trace out the *cycle* containing an element by repeatedly moving from our current index i to the element at index $A[i]$, until we return to our starting point. Using this operation, we can decompose any permutation into a collection of disjoint cycles — this is a common alternative method for representing a permutation. Please show that a random length-n permutation decomposes into $O(\log n)$ cycles with high probability. As a hint, consider the cycle traced out starting with the first element of the array. [Solution]

Problem 19 (Load Balancing with a Random Assignment). Probability theorists are quite fond of "balls in bins" problems where we throw n balls at random into m bins (in this problem, we assume $n = m$ for simplicity). If we interpret balls as unit-sized computational jobs and bins as machines, we are constructing a random assignment of jobs to machines. This type of situation often arises in practice when we want to serve traffic for a website that is so popular that a single web server will not suffice. Here, we often locate a set of m web servers behind a "switch" that randomly assigns incoming requests (jobs) to web servers. This approach balances the load by assigning each server on average n/m jobs (exactly one job, under our assumption that $n = m$). Please show that each machine receives at most $O(\log n)$ jobs with high probability (that is, with probability at least $1 - 1/n^c$, for any constant c of our choosing), so there is little danger of substantially overloading a machine[5]. [Solution]

Problem 20 (Properties of Random Trees). There are several nice ways to build a random n-node labeled tree (with nodes numbered with labels $1 \ldots n$)[6]. In this problem, we consider the following simple method: start from node 1, and for each i from $2 \ldots n$ in sequence, attach node i to a randomly chosen node in $1 \ldots i - 1$.

(a) Please show that the maximum node degree in our random tree is $O(\log n)$ with high probability. [Solution]

[4]We discuss some related results to this problem later in the book: in problem 217 we develop an $O(n \log n)$ algorithm for finding the longest *increasing* subsequence, and in problem 364 we prove the *Erdös-Szekeres theorem*, which states that any length-n sequence must have either an increasing or decreasing subsequence consisting of at least $\sqrt{n}$ elements.

[5]Note that with a more detailed Chernoff bound analysis, this bound can be improved slightly to $O(\log n / \log \log n)$. There are also some interesting related results achievable by probing multiple bins; for example, if you place each ball in whichever of two randomly chosen bins is least full, then a much more sophisticated analysis shows that the expected maximum fullness is only $O(\log \log n)$.

[6]One elegant method is using the so-called *Prüfer* code — a method for encoding any n-node tree (with nodes labeled $1 \ldots n$) as an integer sequence of length $n - 2$ (with each element in the range $1 \ldots n$). The mapping is one-to-one (thereby providing a nice proof that there are exactly n^{n-2} labeled trees on n nodes), and can be performed in either direction — tree to sequence, or sequence to tree — in $\Theta(n)$ time. Since the number of occurrences of x in the sequence is one less than the degree of node x in the tree, problem 19 tells us that if we build a random n-node tree from a randomly-generated Prüfer code (equivalent to throwing $n - 2$ balls into n bins), its maximum degree will be $O(\log n)$ with high probability. See also problem **??** on generating a random *spanning* tree of a graph, and see the endnotes for a brief discussion of properties of random graphs in general.

(b) Please argue that the diameter (length of the longest path) of our random tree is $O(\log n)$ with high probability. [Solution]

Problem 21 (The Coupon Collector Problem). Suppose that you see a special advertisement on your box of breakfast cereal claiming that inside the box you will find one of n different types of special coupons, each equally likely. Please show that you will find at least one of each coupon type with high probability if you open $O(n \log n)$ boxes of cereal. As a hint, try to prove the slightly stronger result that by the time you have accumulated all n coupon types, you will have found only $O(\log n)$ copies of any single particular coupon type with high probability. [Solution]

Problem 22 (Randomly Spreading Information in a Distributed Network). Suppose we have a distributed network of n processors where one processor wants to broadcast a message to the others. Suppose further that we are operating under a "synchronous" model of distributed computation where time proceeds in steps according to some global clock, and in each time step a processor can exchange a single message with another processor.

(a) In each time step, suppose each processor that has heard the message contacts a random processor and sends it the message (note that if we are unlucky, the other processor might have already heard the message). This is called a *gossip* protocol. Using the tools from this chapter, see if you can prove that all n processors will hear the message after only $O(\log n)$ steps with high probability (with probability at least $1 - 1/n^c$ for any constant $c > 0$ of our choosing). As a hint, consider separately the number of steps until some constant fraction of the processors has initially heard the message, and then the number of steps required to spread to the remaining processors. [Solution]

(b) Suppose we implement a gossip protocol by "pulling" rather than "pushing" the message. That is, in each time step, each processor without the message contacts a random other processor and asks for a copy of the message (which it may or may not have, and even if it does have the message, it may not be able to communicate if it is already talking to a different processor, since each processor can only participate in a single communication session per step). Please try to prove that this variant also distributes a message from a single processor to all n processors in $O(\log n)$ steps with high probability. [Solution]

2.4.3 Random Variables and Linearity of Expectation

We now turn to the second major concept in our study of probability theory: *random variables.* In contrast, to a standard (non-random) variable that represents a single value, a random variable is associated with a probability distribution over values. For example, if X represents the smaller of the face values when we roll two 6-sided dice, its distribution is

$$1 : \frac{11}{36} \quad 2 : \frac{9}{36} \quad 3 : \frac{7}{36} \quad 4 : \frac{5}{36} \quad 5 : \frac{3}{36} \quad 6 : \frac{1}{36}.$$

If we actually perform a random trial by rolling two dice, then we can think of *sampling* a value according to this distribution to *instantiate* the value of X. In other words, X serves as a "placeholder" for a value between 1 and 6 that would only materialize after we perform a random trial (although we never actually replace X with a concrete value in this fashion). A random variable assigns a numeric value to every possible outcome associated with some random experiment; in the example above, the outcome $(3, 5)$ maps to the value $X = 3$.

Two random variables X and Y are *independent* if knowledge of the value of X does not change the probability distribution associated with Y, and vice versa. If X is a random variable, then expressions like "$X = v$" or "$X \geq v$" represent events (i.e., the event that X takes the value v, or the event that X takes a value at least as large as v), so it makes sense to write $\mathbf{Pr}[X = v]$ or $\mathbf{Pr}[X \geq v]$. Please refrain from writing $\mathbf{Pr}[X]$ as this makes no sense, since random variables are not events!

Expectation. The *expected value* (also called the *mean*) of a random variable X is the "center of mass" for X's probability distribution, defined as follows for a discrete[7] random variable:

$$\mathbf{E}[X] = \sum_v v\mathbf{Pr}[X = v].$$

That is, $\mathbf{E}[X]$ is the sum over all possible values v that X can take, of v weighted by the probability of X taking the value v.

Here are some examples:

- If X denotes the minimum of the values obtained by two rolls of a 6-sided die, then $\mathbf{E}[X] = 1 \cdot \frac{11}{36} + 2 \cdot \frac{9}{36} + 3 \cdot \frac{7}{36} + 4 \cdot \frac{5}{36} + 5 \cdot \frac{3}{36} + 6 \cdot \frac{1}{36} = 91/36$.
- If X denotes the number of heads we see when flipping 100 fair coins, then $\mathbf{E}[X] = 50$. If the coins are biased and show heads with probability $3/4$ and tails with probability $1/4$, then $\mathbf{E}[X] = 75$.
- If we repeatedly flip a biased coin (showing heads with probability $1/10$) and T denotes the number of flips up to and including the first time we see heads, then $\mathbf{E}[T] = 10$.
- If T denotes the amount of time we spend performing a linear search for a randomly-chosen element in an n-element array, then $\mathbf{E}[T] = \Theta(n)$.
- If T denotes the amount of time we spend performing a binary search for a randomly-chosen element in a sorted n-element array, then $\mathbf{E}[T] = \Theta(\log n)$.

The running time of a Las Vegas algorithm is a random variable. It is small if the algorithm is lucky and large if the algorithm is unlucky in its random choices. The simplest question you can generally ask about a Las Vegas algorithm is what is its expected running time. If our expected running time is, say, $O(n \log n)$, this gives a sense of what we can expect "on average" when we run the algorithm, but it still leaves open the possibility that the running time may have high variability, and may take much more than $O(n \log n)$ time with some non-negligible probability. In order to convince even the most stubborn critic that our algorithm is reliable enough to use, we may want to attempt to prove the much stronger result that it runs in $O(n \log n)$ time with high probability.

[7] A discrete random variable takes only a finite number of possible values v when restricted to any interval. In this book, essentially all of our random variables will be discrete (and mostly also integer-valued). Continuous random variables are not particularly problematic — they just require more familiarity with calculus and continuous mathematics to handle. For example, the expected value of a continuous random variable taking a value uniformly selected from the interval $[0, 10]$ would be written as the integral $\int_0^{10} \frac{x}{10} dx = 5$.

Problem 23 (Expected Versus High Probability Results). It is worth noting that while a high probability running time bound of $O(f(n))$ is generally regarded as much stronger than an expected running time bound of $O(f(n))$, neither bound technically implies the other. To illustrate this fact, please describe a hypothetical randomized algorithm whose running time is $O(f(n))$ in expectation but not $O(f(n))$ with high probability. Next, show how we could have a running time that is $O(f(n))$ with high probability, but not $O(f(n))$ in expectation. [Solution]

Linearity of Expectation. A random variable representing the running time of even a simple randomized algorithm is usually too complicated to allow computation of its expected value using the definition above. Fortunately, *linearity of expectation* lets us compute expected values of complicated random variables by decomposing them into sums of much simpler random variables. For instance, suppose $Z = 2X + 3Y$ where X and Y are simple random variables, and Z, defined in terms of X and Y, is also a random variable, albeit with a slightly more complicated distribution[8]. In this case, we can take advantage of the fact that $\mathbf{E}[\cdot]$ is a *linear* operator: if c is a constant and X and Y are random variables, then $\mathbf{E}[cX] = c\mathbf{E}[X]$ and $\mathbf{E}[X+Y] = \mathbf{E}[X]+\mathbf{E}[Y]$. It is especially noteworthy that X and Y do *not* need to be independent. We can apply linearity of expectation to *any* weighted sum of random variables, and this is what makes it such a powerful and useful principle. Applied to the example above, we have $\mathbf{E}[Z] = \mathbf{E}[2X + 3Y] = 2\mathbf{E}[X] + 3\mathbf{E}[Y]$, so $\mathbf{E}[Z]$ is easy to compute as long as we know $\mathbf{E}[X]$ and $\mathbf{E}[Y]$. Linearity of expectation is trivial to prove based on the definition of expectation. [Short proof]

Let us work through a short example just to illustrate a common usage of the technique. Suppose we want to compute $\mathbf{E}[X]$ where X denotes the number of heads we see in 100 flips of a fair coin. The random variable X has a somewhat complicated distribution, so computing $\mathbf{E}[X]$ directly would involve solving

$$\begin{aligned}\mathbf{E}[X] &= \sum_v v\mathbf{Pr}[X = v] \\ &= \sum_{v=0}^{100} v\binom{100}{v}/2^{100},\end{aligned}$$

which, while feasible, is perhaps more complicated than we might like. Instead, let us decompose X into a sum of much simpler random variables:

$$X = X_1 + X_2 + \ldots + X_{100},$$

where X_i takes the value 1 if the ith flip is heads, 0 otherwise. A 0/1-valued random variables like X_i is called an *indicator* random variable since its value indicates whether or not a certain event has occurred. The expected value of an indicator random variable is easy to compute:

$$\begin{aligned}\mathbf{E}[X_i] &= \sum_v v\mathbf{Pr}[X_i = v] \\ &= 0 \cdot \mathbf{Pr}[X_i = 0] + 1 \cdot \mathbf{Pr}[X_i = 1] \\ &= \mathbf{Pr}[X_i = 1] \\ &= 1/2.\end{aligned}$$

[8]In Chapter 23, we will see that the distribution of a sum of independent random variables is given by the *convolution* of their individual distributions.

Note that $\mathbf{E}[X_i]$ is just the probability of the event for which X_i serves as an indicator. This is true for any indicator random variable — its expected value is just the probability of its associated event. We now have

$$\mathbf{E}[X] = \mathbf{E}[X_1 + \ldots + X_{100}] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_{100}] = 50,$$

which is the result we intuitively expect. Linearity of expectation is one of the most valuable tools we have for analyzing randomized algorithms, and we will use it on many occasions. The reader is encouraged to learn it well.

Expectations of Products. Since expectations of sums behave so nicely, what happens with products? Unfortunately, $\mathbf{E}[XY] = \mathbf{E}[X]\,\mathbf{E}[Y]$ is *not* true in general. If this property does hold, then we say X and Y are *uncorrelated*. Since independent random variables are always uncorrelated, we can always decompose the expectation of a product of independent random variables into a product of expectations. Please take care not to confuse this with linearity of expectation for sums, which applies to any sum of random variables regardless of independence.

Expected Trials Until Success. If we roll a 6-sided die, then $\mathbf{Pr}[E] = 1/6$ if E is the event that we roll a 2. If we keep rolling until the event E occurs, how many trials do we expect to perform, including the final successful trial? The answer is what we might intuitively expect: 6. This is an example of a very useful principle: if we perform a sequence of independent trials, where each trial succeeds with probability p, then the expected number of trials up to and including the first success is $1/p$. Similarly, if each trial succeeds with probability *at least* p, then we expect *at most* $1/p$ trials. [Very short proof]

Problem 24 (Linearity of Expectation Practice). This problem demonstrates how we can use linearity of expectation to compute the expectation of a complicated random variable.

(a) **Exchanging Hats.** Suppose n people in a room are each wearing different hats. If they all exchange hats according to a random permutation, how many people do you expect to receive their original hat? [Solution]

(b) **Balls in Bins.** If we randomly throw n balls into m bins, what is the expected number of balls that end up in each bin? Among all $\binom{n}{2}$ pairs of balls, how many pairs do we expect to "collide" by landing in the same bin? What is the expected number of empty bins? For the last question, an approximate answer is fine. [Solution]

(c) **The Coupon Collector Problem.** Recall problem 21: inside your box of breakfast cereal you will find one of n different types of special coupons, each equally likely. Please show that you expect to open $\Theta(n \log n)$ boxes of cereal before you have collected at least one of each coupon type. As a hint, decompose the sequence of boxes we check into a series of "phases", where during one phase we have collected exactly k distinct coupons and we are opening boxes in hopes of finding a coupon of any of the remaining $n - k$ types. [Solution]

(d) **Revisiting the Randomized Reduction Lemma.** Consider a randomized algorithm whose input consists of n elements, where in each iteration of the algorithm there is at least some constant probability of reducing our problem to at most some constant fraction of its original size. In this setting, we learned how the randomized reduction lemma guarantees that we spend only $O(\log n)$ iterations with high probability. Just for fun, use linearity of expectation to show an analogous (and slightly weaker) result that we spend $O(\log n)$ iterations in expectation. [Solution]

(e) **Prefix Maxima.** Suppose in an array $A[1 \ldots n]$ that you want to compute for each index j the maximum of $A[1 \ldots j]$. This is easy to do in $\Theta(n)$ time, of course, by scanning through A and keeping a running maximum. As an exercise, however, consider the following alternative method: process the elements of A in random order. For each element $A[j]$, scan backward down to $A[1]$ keeping a running maximum. During this scan, we stop at the first element $A[i]$ previously processed, since we will have already computed and stored the maximum of $A[1 \ldots i]$, making it unnecessary to scan these elements again. What is the expected running time of this algorithm? [Solution]

Random Numbers of Random Trials and Iterated Expectation. Suppose X_1, X_2, etc., represent random variables with identical distributions, so they all have the same expectation $\mathbf{E}[X_i]$. Linearity of expectation gives the expected sum of a *fixed* number n of these variables: $\mathbf{E}[X_1 + \ldots + X_n] = n\mathbf{E}[X_i]$. However, n is itself often a random variable, for example if we are performing a random number of iterations of a randomized subroutine. In this case, intuition would suggest that $\mathbf{E}[X_1 + \ldots + X_n] = \mathbf{E}[n]\,\mathbf{E}[X_i]$, and indeed this is the case, provided there is no dependence between the X_i's and the number of trials n. This result goes by the name of *Wald's Theorem*, and it also follows from the more general *law of iterated expectations*, which can be useful in "flattening out" compositions of expectations, in this case $\mathbf{E}_n[\mathbf{E}[X_1 + \ldots + X_n \mid n]]$, where the inner expectation is conditioned on n being fixed and the outer expectation is over n. [Further details]

Problem 25 (Network Contention Resolution). Suppose n processors are all simultaneously trying to transmit information on a shared network cable, where each processor has a stream of packets to transmit. In each time step, each processor can decide if it should attempt to transmit a packet, or if it should remain idle. The goal is to have exactly one processor transmit in each step, but this can be difficult to achieve since the processors have no means of coordinating with each-other. If more than one processor attempts to transmit, the transmissions all "collide" and fail (and the processors can detect when this happens by listening to the network, so they know to retry transmission of the same packets later).

(a) If the processors all know the value of n, then let each one independently attempt transmission during each time step with probability $p = 1/n$. Please show that the expected number of time steps until a successful transmission in this case is approximately e, and (if you know calculus) please argue $1/n$ is in fact the optimal value to choose for p in order to maximize throughput. [Solution]

(b) If our processors don't know the value of n, let each processor attempt to transmit with probability $1/2$. If it decides to transmit and fails due to a collision, then it becomes "dormant" and waits until a time step occurs when it hears no other transmissions before it wakes up and starts attempting to transmit again (again with probability $1/2$ per time step). Please argue that we expect $O(\log n)$ time steps to elapse between successful transmissions. [Solution]

"Per-Element" Expected Running Time Analysis. Recall from our discussion of high probability results that the union bound allows us easily to carry over a high-probability bound for a single "generic" element in our input to all n element elements in our input. For example, if we can show that our algorithm spends $O(\log n)$ time on a generic input element with high probability, then it also spends $O(n \log n)$ total time with high probability. Linearity of expectation allows us to make a similar argument when we analyze the expected running time of an algorithm. If X

denotes the total running time of our algorithm and X_i denotes the running time spent only on element i, then it is easy to compute $\mathbf{E}[X] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n]$ by first computing the "per element" expected running times $\mathbf{E}[X_1] \ldots \mathbf{E}[X_n]$. For example, if our algorithm spends $O(\log n)$ expected time on a generic input element, then linearity of expectation tells us that it spends $O(n \log n)$ total expected time.

Common Types of Distributions. Most random variables we will encounter have probability distributions belonging to a handful of well-known classes. The simplest of these is the *Bernoulli* distribution, which takes the value 1 with probability p and 0 with probability $1 - p$. Bernoulli random variables are also called *indicator* random variables when they are used to indicate whether or not a particular event happens. For example, if we flip a biased coin that comes up heads with probability p and tails with probability $1 - p$, then we could say that our indicator variable takes the value 1 to indicate that the coin comes up heads. The expected value of this indicator variable is exactly p, the probability of its associated event.

A random variable has a *geometric* distribution if its value indicates the number of trials until a particular Bernoulli event is successful. For example, if X denotes the number of flips of our biased coin up to and including the first trial where it comes up heads, then X has a geometric distribution. It is called a "geometric" distribution since the probability of exactly k trials, $\mathbf{Pr}[X = k] = p(1-p)^{k-1}$, decays in a geometric fashion with k. As we just mentioned, a geometric distribution derived from a Bernoulli event occurring with probability p has expected value $1/p$.

Problem 26 (Simulating a Biased Coin with Unbiased Coin Flips). Suppose we would like to simulate the outcome of flipping a biased coin that comes up heads with probability p and tails with probability $1 - p$, where p could potentially even be an irrational number. All we have at our disposal is a single fair coin. Can you think of a simple method that requires only $O(1)$ expected flips of our fair coin to simulate one flip of the biased coin? As a hint, first write out p in binary[9]. [Solution]

Suppose now that we flip n biased coins, each heads with probability p. We can denote whether each individual coin comes up heads by using indicator variables $X_1 \ldots X_n$, and we can write the total number of heads as $X = X_1 + \ldots + X_n$. A random variable like X that is a sum of Bernoulli variables has a *binomial* distribution, so named because $\mathbf{Pr}[X = k]$ (the probability we see k heads in n flips) is given by the binomial coefficient formula $\binom{n}{k} p^k (1-p)^{n-k}$. Its expected value is easily given by linearity of expectation: $\mathbf{E}[X] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n] = np$. Binomial distributions tend to be tightly concentrated around their means, a property we will exploit when we introduce Chernoff bounds momentarily.

The binomial distribution is often approximated by a distribution shaped like the Taylor series expansion of e^{npx} known as the *Poisson* distribution, which is well-studied in probability theory due to its numerous properties and applications.

Problem 27 (The Odds Algorithm and Secretary Problems). The *secretary* problem is a famous probabilistic puzzle where you interview n candidates for a secretary position in random order, each with a distinct skill level that becomes known to you only

[9]This problem is closely related to the topic of *arithmetic coding*, which we briefly mention in Section 10.2.1.

at the interview. After interviewing each candidate, you can either hire them (forgoing the chance to interview any future candidates) or dismiss them in hopes of finding a better candidate later in the ordering (in which case the current candidate can never be recalled). The goal is to maximize your chances of hiring the best candidate, and it turns out we can achieve this by interviewing and dismissing the first $1/e$ fraction of all candidates, then to hire any candidate from that point on who is the best seen so far.

A remarkably simple algorithm, known as the *odds algorithm*, solves not only this problem but a generalization that has several other applications. In a sequence of n independent random trials, let p_i be the probability that trial i "succeeds", let $q_i = 1 - p_i$, and let $r_i = p_i/q_i$ be the odds of success (the ratio of probability for versus against). Our goal is to observe the outcome of each successive trial (success or not), choosing a point at which to stop that maximizes our probability of stopping on the *last* successful event. For the secretary problem, $p_i = 1/i$, since success corresponds to interviewing a candidate who is the best seen so far. However, this scenario applies to many other problems, such as betting on the last time a stock price will jump up, or guessing when to stop at an open parking space in hopes of parking at the closest available space to your place of work.

Letting $R_i = r_i + r_{i+1} + \ldots + r_n$ and $Q_i = q_i q_{i+1} \ldots q_n$, let t be the largest index[10] at which $R_t \geq 1$ (or we set $t = 1$ if no such index exists). Please show that an optimal strategy is to stop at the first successful event at or beyond index t, and that this gives a probability of $R_t Q_t$ of stopping on the last successful event. [Solution]

2.4.4 Tail Inequalities

In addition to computing the expected value of a random variable, we are often interested in the probability that the random variable deviates significantly from its expectation. For example, we will feel much safer using a randomized algorithm whose expected running time is $\Theta(n^2)$ if we know that its running time tends to fall consistently around $\Theta(n^2)$ and not vary too widely. There are several common methods for bounding the probability of drawing a sample from the "tail" of a distribution (the part of the distribution that is far away from the expected value).

Markov's Inequality. If X is any *nonnegative* random variable, then

$$\mathbf{Pr}[X \geq k\mathbf{E}[X]] \leq \frac{1}{k}.$$

This is known as *Markov's inequality.* It tells us that if our algorithm runs in 100 expected units of time, then the probability it runs longer than 400 units of time is at most 1/4 and the probability it runs longer than 900 units of time is at most 1/9. The inequality is "tight" in the sense that we cannot hope for a better bound if all we know about X is its expected value. We can typically only prove fairly weak results using Markov's inequality, but in many cases we will find it sufficient for the task at hand. [A simple proof of Markov's inequality]

Problem 28 (From Expected to High Probability Guarantees). Given any Las Vegas algorithm with expected running time $O(f(n))$, how can we construct a Las Vegas algorithm whose running time is $O(f(n) \log n)$ with high probability? See if you

[10]If every trial has success probability p, our knowledge of geometric random variables tells us to expect $1/p$ trials until the *first* success. Equivalently, we keep a running sum of success probabilities and stop when this reaches 1. Note that the odds algorithm has a pleasantly symmetric form for guessing the *last* success, where we sum the odds in reverse order until we reach a sum of 1.

can find a solution that does not require knowledge of the hidden constant in the original expected running time guarantee. [Solution]

Problem 29 (Randomized Reduction with Reduction in Expectation). Please prove that an algorithm will take $O(\log n)$ iterations both in expectation and with high probability if the expected size of our problem instance after each iteration is at most a constant factor $q \in [0, 1)$ times its current value (and satisfaction of this property in each iteration is independent of other iterations). [Solution]

Chebyshev's Inequality. The *variance* of a random variable X is the expected squared deviation from X's mean:

$$\mathbf{Var}[X] = \mathbf{E}[(X - \mathbf{E}[X])^2].$$

Applying Markov's inequality to the random variable $(X - \mathbf{E}[X])^2$, we obtain

$$\mathbf{Pr}[(X - \mathbf{E}[X])^2 \geq k^2\mathbf{Var}[X]] \leq 1/k^2,$$

which is usually written as

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq k\mathbf{Std}[X]] \leq 1/k^2,$$

where $\mathbf{Std}[X] = \sqrt{\mathbf{Var}[X]}$ is the *standard deviation* of X (we take the square root of the variance to reduce it back to a quantity on the same scale as X's original distribution). The inequality above is called *Chebyshev's inequality*, and it is a useful tool for analyzing a random variable whose variance is easy to compute. For example, it tells us that the probability X falls more than 2 standard deviations away from $\mathbf{E}[X]$ is at most 1/4, and the probability that X deviates from $\mathbf{E}[X]$ by more than 5 standard deviations is at most 1/25.

Chebyshev's inequality is stated most simply in terms of the *z-score* of a sample of a random variable X, defined as $(X - \mathbf{E}[X])/\mathbf{Std}[X]$, telling us how many standard deviations X lies above or below its mean. Chebyshev's inequality tells us that there is at most a $1/k^2$ probability that the z-score of a random variable has absolute value k or larger.

We don't use Chebyshev's inequality too much in this book, so we will spare the reader a lengthy discussion of properties of variances and standard deviation. We will limit our discussion to three useful facts:

- $\mathbf{Var}[X] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2$ is another common formula used to compute variance; it is easily obtained by expanding out $\mathbf{E}[(X - \mathbf{E}[X])^2]$.
- If k is a constant and X is a random variable, then $\mathbf{Var}[kX] = k^2\mathbf{Var}[X]$.
- If X and Y are uncorrelated random variables (which is true if they are independent), then $\mathbf{Var}[X + Y] = \mathbf{Var}[X] + \mathbf{Var}[Y]$.

Problem 30 (Polling and Monte Carlo Estimation). Here we illustrate a typical use of Chebyshev's inequality. Suppose we wish to estimate the value of π by randomly sampling points from a unit square in which a circle is inscribed. As we sample more and more points, the fraction of the points landing inside the circle should approach the ratio of the area of the circle to that of the square, $\pi/4$. Suppose we want to compute an estimate of π in this fashion that is accurate to within an additive error of at most

ε with 99% probability. According to Chebyshev's inequality, approximately how many points do we need to sample? Similarly, suppose we are conducting a poll to estimate what fraction of the population wants to vote for candidate A in an election. According to Chebyshev's inequality, approximately how many people must we sample at random before we have an estimate that is accurate to within an additive error of ε with 99% probability. [Solution]

Chernoff Bounds. Some of the most powerful bounds in our arsenal are the *Chernoff bounds*, which tell us that a binomial random variable tends to be very tightly concentrated near its expected value. For example, we can write the number of heads in n coin flips as $X = X_1 + \ldots + X_n$ where each X_i is an independent indicator random variable that takes the value 1 if coin flip i is heads, 0 otherwise. Here, we anticipate that the number of heads will almost always end up very close to $\mathbf{E}[X] = n/2$.

There are many different forms of Chernoff bounds, depending on whether we want a bound on deviation below or above $\mathbf{E}[X]$, and whether we want a bound on the probability of absolute or relative deviation from $\mathbf{E}[X]$. Letting $X = X_1 + \ldots + X_n$ be a sum of independent indicator random variables, we have

1. $\mathbf{Pr}[X \leq \mathbf{E}[X] - \varepsilon] \leq e^{-2\varepsilon^2/n}$
2. $\mathbf{Pr}[X \geq \mathbf{E}[X] + \varepsilon] \leq e^{-2\varepsilon^2/n}$
3. $\mathbf{Pr}[X \leq (1-\varepsilon)\mathbf{E}[X]] \leq e^{-\varepsilon^2\mathbf{E}[x]/2}$
4. $\mathbf{Pr}[X \geq (1+\varepsilon)\mathbf{E}[X]] \leq \begin{cases} e^{-\varepsilon^2\mathbf{E}[x]/4} & \text{if } \varepsilon \leq 2e-1 \\ 2^{-(\varepsilon+1)\mathbf{E}[x]} & \text{if } \varepsilon \geq 2e-1 \end{cases}$

As an example, what is the probability we see more than 75 heads in 100 coin flips? According to the second bound above, $\mathbf{Pr}[X \geq 75] \leq e^{-25^2/50} = e^{-12.5} < 0.000004$, which is quite unlikely! [For the interested reader, a proof of our Chernoff bounds]

Since the full-fledged Chernoff bounds above can sometimes be cumbersome to use, that is why we earlier developed the randomized reduction lemma as a simpler interface to the Chernoff bounds. Using Chernoff bounds, we can now give a short [proof] of the randomized reduction lemma.

Problem 31 (Boosting Success Probabilities With Two-Sided Errors). Suppose we have a Monte Carlo algorithm for solving a decision problem (so the output is either "yes" or "no"). We say that our algorithm has only a *one-sided error* if it correctly outputs "no" for all "no" inputs, but for a "yes" input it might erroneously output "no" with constant probability $p < 1$ (or vice versa: it could answer "yes" inputs correctly and make mistakes on "no" inputs). In this case, we have shown earlier that by running $O(\log n)$ independent trials of the algorithm we can obtain a high probability bound of success (at least $1 - 1/n^c$ for any constant $c > 0$ of our choosing). Consider now the slightly trickier case of *two-sided error*, where our algorithm might mistakes on both "yes" and "no" inputs. Please show that if the probability of making such a mistake is at most some constant $p < 1/2$, then we can again use $O(\log n)$ independent trials of our algorithm to obtain an answer that is correct with high probability. [Solution]

Problem 32 (Improving Robustness by Taking the Median). Suppose you have an algorithm whose output is accurate to within some $1 \pm \varepsilon$ factor with constant probability strictly greater than 1/2. Please show that by taking the median of $O(\log n)$ invocations of the algorithm, we get an answer that is accurate to within a $1 \pm \varepsilon$ factor with high probability. How is this result related to that of the preceding problem? [Solution]

2.5 Linear Algebra

Most of the serious "number crunching" going on in the world involves linear algebra in some way or another. Linear systems and linear optimization play such a crucial role in such a broad range of computational applications that any reader pursuing a career in computation is highly encouraged to develop some level of familiarity with at least basic linear algebra. In recent years, an increasing number of elegant connections have been made between linear algebra and other seemingly unrelated algorithmic problems, particularly graph problems. For example, we will use linear algebra techniques later in the book to compute shortest paths, transitive closures, graph clusters, linear orderings, and matchings, and to count directed walks in a DAG, spanning trees, arborescences, and directed Eulerian tours. Prior knowledge of linear algebra is helpful for several parts of this book (and in general, linear algebra coursework is recommended to any serious student of computing). Here, we review some of the fundamentals.

Our notation and terminology is fairly standard. All vectors are column vectors; we can express the vector x as a row vector by writing x^T, where the superscript T denotes the transpose operation. For a vector x with n components, we refer to these components individually as either $x(1) \ldots x(n)$ or $x_1 \ldots x_n$. We assume the reader is familiar with the *dot product* (or *inner product*) of two vectors,

$$x \cdot y = x^\mathsf{T} y = \sum_{i=1}^{n} x_i y_i,$$

as well as its geometric interpretation: if y is a unit vector, then $x \cdot y$ gives the length of the projection of x in the direction of y (we elaborate more on this in Section 21.1.3 when we study computational geometry). Accordingly, two vectors x and y are perpendicular, or *orthogonal*, if $x \cdot y = 0$. If both x and y have unit length, then $x \cdot y \in [-1, +1]$ also has an interpretation as the *correlation* of x and y, which is large if x_i tends to be large when y_i is large and vice versa (for example, if $x_1 \ldots x_n$ and $y_1 \ldots y_n$ represent time series data at n values of time, $x \cdot y$ measures the extent to which they follow similarly-shaped trajectories over time). This value also represents the cosine of the angle formed by x and y, which is $+1$ if x and y point in the same direction, zero if they are orthogonal, and -1 if they point in completely opposite directions.

When we study computational geometry we will define and make extensive use of the *cross product* of a pair of three-dimensional vectors. When we write $x \leq y$ for two length-n vectors x and y, this means that $x_i \leq y_i$ for each component $i = 1 \ldots n$.

Independence and Rank. A set of vectors is linearly independent if we cannot express one of them as a weighted sum of the others. The *rank* of a set of vectors is the maximum number of linearly independent vectors in the set, and accordingly the rank of a matrix A is the maximum number of linearly independent rows of A (the same number tells us the maximum number of linearly independent columns in A). A linear algebra enthusiast would also describe the rank of a set of vectors as the dimensionality of the *subspace* spanned by all of their weighted combinations. A matrix is *singular* if its rows or columns are not linearly independent, or equivalently if its rank is less than the number or rows or columns.

Matrix-Vector and Matrix-Matrix Multiplication. It is assumed the reader knows how to multiply a matrix by a vector and a matrix by a matrix: The matrix-vector product Ax gives us a vector whose entries are the dot products between x and rows of A. Equivalently, it gives us a sum of A's columns, each weighted by the components of x. The matrix product AB gives us a matrix in which the (i, j) entry is the dot product of the ith row of A and the jth column of B. Matrix multiplication is associative ($A(BC) = (AB)C$), and distributive ($A(B + C) = AB + AC$), but not necessarily commutative (in general, $AB \neq BA$).

The *inverse* of an $n \times n$ matrix A, denoted A^{-1}, is the unique $n \times n$ matrix such that $AA^{-1} = A^{-1}A = I$, where I is the $n \times n$ *identity* matrix (all zeros, with ones down the diagonal). Every $n \times n$ matrix has an inverse unless it is singular. In Chapter 24, we will learn how to define an object called the *pseudo-inverse*, that plays a similar role to the inverse of either a singular or non-square matrix.

Problem 33 (Using Matrix Multiplication to Count Directed Walks). This problem illustrates just one use of linear algebra to solve a simple graph problem. Let A be the adjacency matrix of a directed graph, so $A_{ij} = 1$ if there is a directed edge from node i to node j, and $A_{ij} = 0$ otherwise. Prove, using induction on k, that the (i, j) entry of A^k tells us the number of different directed walks from i to j that are exactly k edges in length. [Solution]

Orthonormal Bases. In n-dimensional space, a set of mutually orthogonal unit vectors $v_1 \ldots v_n$ is called an *orthonormal basis*, and it defines what we could consider as a new set of "coordinate axes" into which we can project any point in space (i.e., the coordinates of point p along these new axes will be $p \cdot v_1 \ldots p \cdot v_n$). We say an $n \times n$ matrix A is orthonormal if it has orthonormal rows or columns. Orthonormal matrices have many nice properties. For example, the length of a vector does not change when we multiply it by an orthonormal matrix, and if A is orthonormal then A^{-1} is simply $A^\intercal$, the transpose of A. In Chapter 24, we will learn several approaches for constructing an orthonormal basis for the subspace spanned by a set of vectors.

Special Classes of Matrices. Orthonormal matrices are one nice class of matrix we will often encounter. Others include *symmetric* matrices (square matrices A for which $A = A^\intercal$), *symmetric positive semi-definite* matrices (symmetric matrices A for which $x^\intercal Ax \geq 0$ for any vector x), *symmetric positive definite matrices* (similarly defined, only $x^\intercal Ax > 0$ for all $x \neq 0$), *permutation* matrices (the identity matrix with its rows or columns re-ordered), and *Markov* matrices (nonnegative matrices whose rows each sum to 1). We will say more about each of these classes later in the book.

Norms and Distances. The length of a vector x is given by

$$||x|| = \sqrt{x^\intercal x} = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2},$$

and accordingly the distance between two n-dimensional points x and y is given by the length of the vector joining them:

$$||x - y|| = \sqrt{(x-y)^\intercal(x-y)} = \sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2}.$$

There are several other common *norms* we can use to measure lengths of vectors, and hence also distance. Above we have used the common L_2, or *Euclidean* norm $||x||_2$ (which we usually write as just $||x||$ without the subscript 2). Other common vector norms[11] are the L_1 and L_∞ norms,

$$||x||_1 = |x_1| + |x_2| + \ldots + |x_n|, \qquad ||x||_\infty = \max_i |x_i|.$$

When used to measure distance, the L_1 distance between x and y,

$$||x - y||_1 = |x_1 - y_1| + |x_2 - y_2| + \ldots + |x_n - y_n|,$$

is sometimes called the “Manhattan” distance between x and y since in 2 dimensions $||x - y||_1$ corresponds to the distance one would along a grid of city streets from point x to point y. The L_1 and L_∞ distances are sometimes nice to use since they avoid producing irrational numbers, whereas the L_2 (Euclidean) distance measure involves a square root.

2.6 Other Useful Mathematical Topics

The study of algorithms gives us an opportunity to learn useful ideas and techniques from many different areas of mathematics. In this section, we briefly highlight some of the principal remaining mathematical topics not yet discussed above that we will encounter throughout the rest of the book.

Metrics. A *metric* is a distance function between pairs of elements of some set (e.g., points in space, nodes in a graph) that is nonnegative, symmetric, and that obeys the *triangle inequality.* A distance function d satisfies the triangle inequality if $d(x,z) \leq d(x,y) + d(y,z)$ for all ordered triples (x,y,z). In other words, the “direct” distance from x to z is never more than the distance from x to z when traveling through an intermediate point y (this is the same notion as the triangle inequality you learned in elementary geometry, which states that one side of a triangle can be no larger than the sum of the other two sides). The L_2 (Euclidean), L_1 (Manhattan), and L_∞ distance functions are all metrics. We will see several other examples throughout this book. In Chapter **??** we will see a version of Euclidean distance scaled by the shape of an underlying point set known as *Mahalanobis* distance. When we study graphs in more detail, we will learn that both *shortest path*

[11]In general, the length of a vector x as measured with the L_p norm is given by $||x||_p = (|x_1|^p + |x_2|^p + \ldots + |x_n|^p)^{1/p}$. Although $p = 1$, $p = 2$, and $p = +\infty$ are probably the most common values for p, in Section 7.5.8 we will see that $p = 0$ gives a meaningful result (or more precisely, the limit as $p \to 0$), with $||x||_0$ telling us the number of nonzero elements in x. Related to these p-norms is the notion of the p-mean of a vector x of nonnegative real numbers, defined as $[\frac{1}{n}(x_1^p + x_2^p + \ldots + x_n^p)]^{1/p}$. For $p = 1$, this gives the familiar arithmetic mean (average), for $p = 0$ (in the limit), it is the geometric mean, for $p = -1$ it is the “harmonic” mean, for $p = -\infty$ (in the limit) it is the minimum of the x_i's, and for $p = +\infty$ (in the limit) it is the maximum of the x_i's. It is quite nice that one formula captures essentially all common types of “means”, and since the formula increases with p this also gives a convient way to establish that, say, geometric mean is at most arithmetic mean.

distance and also *commute time* distance form metrics over the set of all nodes in a graph.

Problem 34 (Finding a Central Element). Consider a set of n elements $x_1 \ldots x_n$ belonging to a larger set U on which we have defined some distance metric d. For example, $x_1 \ldots x_n$ could be points in k-dimensional space with $d(x, y)$ giving the standard Euclidean distance between points x and y. We wish to find a "central" element $x^* \in U$ whose average distance to our n points $D(x^*) = \frac{1}{n}\sum_i d(x^*, x_i)$ is minimal (this is equivalent to finding a point x^* minimizing the total distance $\sum_i d(x^*, x_i)$ summed over all points). In many cases, this can be rather difficult; for example, finding a consensus ranking under the inversion distance metric (Problem 62) and finding a consensus string under the edit distance metric (Section 9.5.3) are examples of NP-hard optimization problems that fit into this framework. In fact, even for n points in the 2-dimensional plane and Euclidean distances, it is known that there is no explicit formula for x^*. Please show that if we simply choose one of our n input points x_i as our answer (whichever one minimizes $D(x_i)$), then we achieve a 2-approximation. Along the way, please argue that if we pick one of our input points x_i uniformly at random, then $\mathbf{E}[D(x_i)] \leq 2D(x^*)$. [Solution]

Continuous Mathematics; Analysis and Calculus. We encounter problems of a continuous nature quite often in practice. To give a few examples, we may want to solve a nonlinear system of equations, find the roots of a polynomial, fit a polynomial or exponential curve to a set of points, minimize a linear or nonlinear function over some set of constraints, approximate the area bounded by a nonlinear curve, or study the evolution over time of a system of differential equations. In this book, the reader will occasionally benefit from knowledge of derivatives (gradients, in higher dimensions), and how to approximate a function locally in terms of a low-degree polynomial — i.e., the first few terms of its Taylor expansion. These techniques will be particularly useful in Chapter 14 when we study algorithms for optimization of continuous functions.

Combinatorics and Counting. Many algorithmic problems come directly from combinatorics, involving counting, enumerating, or randomly sampling discrete objects of a particular type. For example, in Section **??** we study the problem of counting the number of spanning trees in a graph — a famous problem with an elegant solution. For many problems, we can ask not only for a single solution but also for an algorithm that counts or enumerates all solutions. Since there may be exponentially many, we typically want to count them in polynomial time[12] and enumerate them in polynomial time per solution.

Algorithms deal with many types of "combinatorial objects" (e.g., sequences, trees, graphs) that often end up closely related or even equivalent to each-other, allowing algorithms designed for one type of object to be easily adapted for others. Examples are shown in Figure 2.7; those shown in parts (b) and (c) are just a small subset of dozens of objects with a "nested parenthesis" structure, countable by the famous *Catalan numbers*. [Further details on equivalence and structure of these objects]

[12] There is one slight nuisance we should bear in mind for counting problems, particularly those in which the answer can be exponentially large. The RAM model of computation usually only permits us $O(\log n)$ bits in a word, allowing us to count only "polynomially" high — up to n^c for some constant c. If we need to count to some "exponentially" high number like 2^n, this requires at least n bits. We can resolve this issue either by relaxing the model of computation or by penalizing our running times to account for arithmetic on large numbers.

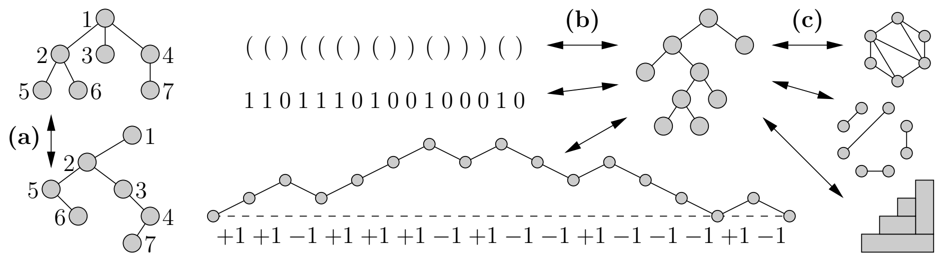


FIGURE 2.7: Equivalent combinatorial objects: (a) forests of rooted ordered trees on n nodes are equivalent to binary trees on n nodes by re-interpreting each node's "first child" and "next sibling" pointers as "left child" and "right child"; (b) a "full" binary tree on n nodes (each node being a leaf or having two children) can be traversed (as in Figure 6.3(a)) to obtain an equivalent sequence of $2n-2$ balanced parentheses, or similarly a binary sequence of length $2n-2$ where each prefix has no more 0s than 1s (known as *Dyck sequences*), or a sequence of +1s and -1s of length $2n-2$ where every prefix has a nonnegative sum, equivalent to a "landscape" that never crosses underground — here, following an edge downward emits (, 1, or +1, and upward emits), 0, or -1. Part (c) shows equivalence between full binary trees on $2n+1$ nodes, triangulations of convex polygons with $n+2$ sides, non-crossing perfect matchings in convex polygons with $2n$ vertices, and staircase partitions built from n rectangles.

Problem 35 (Indexing and Sampling Combinatorial Objects). If we are considering a large set of k possible objects (e.g., permutations, combinations, graphs, trees, etc.), it is sometimes helpful to find a convenient mapping between these objects and the integers $0 \ldots k-1$. After doing this, we can easily enumerate all k objects by stepping through $i = 0 \ldots k-1$ and asking our mapping to produce the ith object. It is also easy to randomly sample an object by applying the mapping to a random value of i chosen uniformly between 0 and $k-1$. For simplicity, since k might be exponentially large, please feel welcome to ignore the issue of word size for parts (a) and (b) below.

(a) Suppose the objects we want to index are the $n!$ different permutations of an array containing n distinct elements. Give a $\Theta(n)$ algorithm that constructs the ith such permutation in lexicographic order given any $i \in \{0, 1, \ldots, n!-1\}$, and give a $\Theta(n)$ algorithm that performs the inverse mapping as well, taking an ordering and producing i as output. [Solution]

(b) Consider the same problem for combinations, rather than permutations. That is, we want to index all k-element subsets of an array containing n distinct elements. Show how to map between a subset (stored in an array of size k) and an index $i \in \{0, 1, \ldots, \binom{n}{k} - 1\}$ in $O(k)$ time. [Solution]

(c) To generate a random k-element subset of an n-element array, consider the following approach: for each i from $n-k+1$ up to n, choose a random index $j \in \{1, \ldots, i\}$. If the jth element is not in our subset already, add it, otherwise add the ith element instead. We can implement this method in $O(k)$ expected time with hashing (Chapter 7). Please show that it does indeed produce a k-element subset chosen uniformly at random. [Solution]

Information Theory and Entropy. A nonnegative vector x with components summing to one can be regarded as a probability distribution. The *entropy* of this vector, defined by

$$H(x) = \mathbf{E}[-\log_2 x_i] = -\sum_{i=1}^{n} x_i \log x_i$$

is maximized at $H(x) = \log_2 n$ for a vector $x = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ representing a uniform distribution, and minimized at $H(x) = 0$ for a vector like $x = (0, 0, 0, 1, 0)$ with all its mass isolated in a single component. Entropy reflects the amount of disorder or uncertainty inherent in a probability distribution, or alternatively the expected amount of "surprise" inherent in drawing a sample from it, measured in the expected number of bits required to describe the sample. For example, a sample from the distribution described by $x = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ is equally likely to be any of its n components, so we need $\log_2 n$ bits to describe the result, whereas we need zero bits to describe the completely predictable result from sampling from $x = (0, 0, 0, 1, 0)$. Entropy is an important concept in data compression, since data described by a low-entropy distribution can be compressed more easily (e.g., English text, with its non-uniform letter frequencies, can be more easily compressed than a text with roughly equal frequencies of 'A' through 'Z'). We elaborate on connections with data compression in Section 10.2.1. Entropy also provides a natural formula for measuring the extent to which the mass within a vector is globally spread out in a somewhat uniform fashion versus locally clumped within a few components.

If two random variables X and Y are not independent, then revealing the instantiation of X lowers the uncertainty (entropy) for Y's distribution. The amount of reduction in entropy for Y is called the *mutual information* between X and Y; we get the same number if we measure the reduction in X's entropy as a consequence of revealing Y. This is a good measure of dependency between X and Y.

Modular Arithmetic, Algebraic Fields. It is good to be comfortable with arithmetic on integers modulo some integer n. Here we identify a number only with its remainder when divided by n, so for example 8 and 1 are the same number in arithmetic modulo 7. We write this fact as $8 \equiv 1 \pmod{7}$, which reads "8 is congruent to 1, modulo 7". After every arithmetic operation, we "reduce" the result by taking its remainder modulo n so it drops back into the range $0 \ldots n-1$. For example, in modulo 7 arithmetic, we obtain 2 when we add 4 and 5, when we subtract 6 from 1, and when we multiply 5 and 6. We often use arithmetic modulo n to ensure the numbers during a computation don't grow too large.

In an equation like $x + a \equiv b \pmod{n}$, we can solve for x by adding $-a$ to both sides, just like with ordinary arithmetic. Moreover, if $n = p$ where p is a prime number, then remarkably division is also possible! That is, if $ax \equiv b \pmod{p}$ and $a \neq 0$, then there is a unique multiplicative inverse a^{-1} satisfying $aa^{-1} \equiv 1 \pmod{p}$ that we can multiply by both sides to solve for x. For example, if $3x \equiv 5 \pmod{7}$, then we can multiply both sides by 5 (the multiplicative inverse of 3, when working modulo 7) to obtain $x \equiv 4 \pmod{7}$. We will use the existence of unique multiplicative inverses modulo a prime on several occasions in this book, notably when we study hashing in Chapter 7. To compute multiplicative inverses modulo a prime efficiently, we can use an extended version of *Euclid's algorithm*, described in Section **??**.

Since arithmetic modulo a prime supports addition, subtraction, multiplication, and division, as well as a few extra properties to which we are accustomed in normal arithmetic (e.g., addition and multiplication are commutative and associative, multiplication distributes over addition, etc.), this system of arithmetic is known as an algebraic *field*. Many mathematical algorithms that use only addition, subtraction, multiplication, and division (for example solving a system of linear equations or interpolating a polynomial) work perfectly well in any field, so just as you can use them for problems involving real numbers, you can also use them to solve problems involving arithmetic modulo a prime. In fact, one can more generally build a finite field not just of size p over integers modulo a prime p, but of any size p^k, where p is prime and k is a nonnegative integer (interestingly, these are the only possible valid sizes for finite fields). We discuss these more general finite fields, known as *Galois* fields, in slightly more detail in Section 23.1.3.

Polynomial Root Bounds. Consider a degree-n polynomial $A(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ that is not identically zero. As a consequence of the *fundamental theorem of algebra*, we know that A can have at most n *roots* (values of x for which $A(x) = 0$). This holds in any field, including arithmetic modulo a prime p. Since this system of arithmetic only involves the p different numbers $0, 1, \ldots, p-1$, another way to state our polynomial root bound is by saying that for any degree-n polynomial $A(x)$ that is not identically zero, at most an n/p fraction of all possible settings for x will result in $A(x) \equiv 0 \pmod{p}$. As it turns out, this bound generalizes to *multivariate*[13] polynomials as well: if $A(x_1, \ldots, x_k)$ is a multivariate polynomial of degree n (not identically zero), then at most an n/p fraction of all possible variable settings for $x_1 \ldots x_k$ will result in $A(x_1, \ldots, x_k) \equiv 0 \pmod{p}$ [Short proof]. This bound typically comes into play in randomized algorithms, where we can say that a random choice for $x_1 \ldots x_k$ is unlikely to be a root of A as long as p is sufficiently large (usually choosing a root corresponds to failure, so we want to minimize the chance this happens). We will use this idea in Chapter 20 to develop simple randomized algorithms for a variety of matching problems.

Problem 36 (Secret Sharing). This problem illustrates a simple yet powerful result that makes use of polynomial root bounds.

(a) Show that the n coefficients of a degree-$(n-1)$ polynomial are uniquely determined if we specify the value of the polynomial at n or more different points, even if we are performing arithmetic modulo a prime p. [Solution]

(b) Suppose n people want to share a secret (some integer a in the range $0 \ldots p-1$, where p is prime). For extra security, they want to distribute information about the secret among themselves so that it takes at least k people cooperating together to determine the secret. That is, any subset of k or more people should be able to determine the secret, and any subset of $k-1$ or fewer people should not be able to learn anything about the secret. How can we use the preceding fact to accomplish this? [Solution]

Problem 37 (Verifying Matrix Multiplication). This problem serves as a nice conclusion for our chapter since it demonstrates an elegant combination of many of the techniques we have talked about so far. Suppose we have two $n \times n$ matrices A and B that we would like to multiply to obtain an $n \times n$ product matrix C. Matrix multiplication, as we will learn in Chapter 24, takes a substantial amount of time — $O(n^3)$ time to multiply

[13] A multivariate polynomial is a polynomial involving several variables, such as $A(x_1, x_2, x_3) = 7x_1x_2^2 + 3x_1^2x_2^3x_3 - 2x_2x_3$. Here, the degree of a term is the sum of the exponents of the variables in the term (e.g., the degree of the term $3x_1^2x_2^3x_3$ is $2+3+1=6$), and the overall degree of a polynomial is the maximum term degree.

matrices in the usual straightforward fashion, and $O(n^{2.3727})$ time using sophisticated and very complicated techniques. By contrast, we can multiply a matrix by a vector in only $O(n^2)$ time. This allows us to develop an remarkably simple randomized algorithm, initially due to the Latvian mathematician Rusins Freivalds, that *verifies* the result of a matrix multiplication faster than we know how to multiply matrices in the first place. The idea is simple, given three matrices A, B, and C as input, we want to check whether $AB = C$. To do this, we pick a random vector x whose n components are all either 0 or 1 each with probability 1/2, and we compare ABx to Cx. Note that this takes only $O(n^2)$ time, since $ABx = A(Bx)$, so we can first multiply B times x and then multiply the resulting vector by A. If $ABx \neq Cx$, we know for a fact that $AB \neq C$. However, if $ABx = Cx$ we may suspect that $AB = C$ but there is some chance we are making a mistake. This approach also gives us a preview of the use of "hashing" (Chapter 7) to compare large complicated objects by first mapping them down to simpler objects.

Please show that $\mathbf{Pr}[ABx = Cx \mid AB \neq C] \leq 1/2$, so our algorithm has a probability of at most 1/2 of mistakenly claiming that $AB = C$ when actually $AB \neq C$. As a hint, you may first want to consider the simpler problem of showing that $\mathbf{Pr}[a \cdot x = b \cdot x \mid a \neq b] \leq 1/2$ if a and b are length-n vectors and x is a randomly-chosen length-n vector whose components are each independently set to 0 or 1 with probability 1/2. Show also how you can achieve a high probability bound for correctness by running multiple times, or by using integers modulo a large prime p instead of just zeros and ones. [Solution]