Chapter 13

Randomized Algorithms

The idea that a process can be "random" is not a modern one; we can trace the notion far back into the history of human thought and certainly see its reflections in gambling and the insurance business, each of which reach into ancient times. Yet, while similarly intuitive subjects like geometry and logic have been treated mathematically for several thousand years, the mathematical study of probability is surprisingly young; the first known attempts to seriously formalize it came about in the 1600s. Of course, the history of computer science plays out on a much shorter time scale, and the idea of randomization has been with it since its early days.

Randomization and probabilistic analysis are themes that cut across many areas of computer science, including algorithm design, and when one thinks about random processes in the context of computation, it is usually in one of two distinct ways. One view is to consider the world as behaving randomly: One can consider traditional algorithms that confront randomly generated input. This approach is often termed *average-case analysis*, since we are studying the behavior of an algorithm on an "average" input (subject to some underlying random process), rather than a worst-case input.

A second view is to consider algorithms that behave randomly: The world provides the same worst-case input as always, but we allow our algorithm to make random decisions as it processes the input. Thus the role of randomization in this approach is purely internal to the algorithm and does not require new assumptions about the nature of the input. It is this notion of a *randomized algorithm* that we will be considering in this chapter.

Why might it be useful to design an algorithm that is allowed to make random decisions? A first answer would be to observe that by allowing randomization, we've made our underlying model more powerful. Efficient deterministic algorithms that always yield the correct answer are a special case of efficient randomized algorithms that only need to yield the correct answer with high probability; they are also a special case of randomized algorithms that are always correct, and run efficiently *in expectation*. Even in a worst-case world, an algorithm that does its own "internal" randomization may be able to offset certain worst-case phenomena. So problems that may not have been solvable by efficient deterministic algorithms may still be amenable to randomized algorithms.

But this is not the whole story, and in fact we'll be looking at randomized algorithms for a number of problems where there exist comparably efficient deterministic algorithms. Even in such situations, a randomized approach often exhibits considerable power for further reasons: It may be conceptually much simpler; or it may allow the algorithm to function while maintaining very little internal state or memory of the past. The advantages of randomization seem to increase further as one considers larger computer systems and networks, with many loosely interacting processes—in other words, a *distributed system*. Here random behavior on the part of individual processes can reduce the amount of explicit communication or synchronization that is required; it is often valuable as a tool for *symmetry-breaking* among processes, reducing the danger of contention and "hot spots." A number of our examples will come from settings like this: regulating access to a shared resource, balancing load on multiple processors, or routing packets through a network. Even a small level of comfort with randomized heuristics can give one considerable leverage in thinking about large systems.

A natural worry in approaching the topic of randomized algorithms is that it requires an extensive knowledge of probability. Of course, it's always better to know more rather than less, and some algorithms are indeed based on complex probabilistic ideas. But one further goal of this chapter is to illustrate *how little* underlying probability is really needed in order to understand many of the well-known algorithms in this area. We will see that there is a small set of useful probabilistic tools that recur frequently, and this chapter will try to develop the tools alongside the algorithms. Ultimately, facility with these tools is as valuable as an understanding of the specific algorithms themselves.

13.1 A First Application: Contention Resolution

We begin with a first application of randomized algorithms—contention resolution in a distributed system—that illustrates the general style of analysis

we will be using for many of the algorithms that follow. In particular, it is a chance to work through some basic manipulations involving *events* and their probabilities, analyzing intersections of events using *independence* as well as unions of events using a simple *Union Bound*. For the sake of completeness, we give a brief summary of these concepts in the final section of this chapter (Section 13.15).

The Problem

Suppose we have n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. We imagine time as being divided into discrete *rounds*. The database has the property that it can be accessed by at most one process in a single round; if two or more processes attempt to access it simultaneously, then all processes are "locked out" for the duration of that round. So, while each process wants to access the database as often as possible, it's pointless for all of them to try accessing it in every round; then everyone will be perpetually locked out. What's needed is a way to divide up the rounds among the processes in an equitable fashion, so that all processes get through to the database on a regular basis.

If it is easy for the processes to communicate with one another, then one can imagine all sorts of direct means for resolving the contention. But suppose that the processes can't communicate with one another at all; how then can they work out a protocol under which they manage to "take turns" in accessing the database?

Designing a Randomized Algorithm

Randomization provides a natural protocol for this problem, which we can specify simply as follows. For some number $p > 0$ that we'll determine shortly, each process will attempt to access the database in each round with probability p, independently of the decisions of the other processes. So, if exactly one process decides to make the attempt in a given round, it will succeed; if two or more try, then they will all be locked out; and if none try, then the round is in a sense "wasted." This type of strategy, in which each of a set of identical processes randomizes its behavior, is the core of the *symmetry-breaking* paradigm that we mentioned initially: If all the processes operated in lockstep, repeatedly trying to access the database at the same time, there'd be no progress; but by randomizing, they "smooth out" the contention.

Analyzing the Algorithm

As with many applications of randomization, the algorithm in this case is extremely simple to state; the interesting issue is to analyze its performance.

Defining Some Basic Events When confronted with a probabilistic system like this, a good first step is to write down some basic events and think about their probabilities. Here's a first event to consider. For a given process P_i and a given round t, let $\mathcal{A}[i,t]$ denote the event that P_i attempts to access the database in round t. We know that each process attempts an access in each round with probability p, so the probability of this event, for any i and t, is $\Pr\left[\mathcal{A}[i,t]\right] = p$. For every event, there is also a *complementary event*, indicating that the event did not occur; here we have the complementary event $\overline{\mathcal{A}[i,t]}$ that P_i does not attempt to access the database in round t, with probability

$$\Pr\left[\overline{\mathcal{A}[i,t]}\right] = 1 - \Pr\left[\mathcal{A}[i,t]\right] = 1 - p.$$

Our real concern is whether a process *succeeds* in accessing the database in a given round. Let $\mathcal{S}[i,t]$ denote this event. Clearly, the process P_i must attempt an access in round t in order to succeed. Indeed, succeeding is equivalent to the following: Process P_i attempts to access the database in round t, and each other process *does not* attempt to access the database in round t. Thus $\mathcal{S}[i,t]$ is equal to the intersection of the event $\mathcal{A}[i,t]$ with all the complementary events $\overline{\mathcal{A}[j,t]}$, for $j \neq i$:

$$\mathcal{S}[i,t] = \mathcal{A}[i,t] \cap \left(\bigcap_{j \neq i} \overline{\mathcal{A}[j,t]} \right).$$

All the events in this intersection are independent, by the definition of the contention-resolution protocol. Thus, to get the probability of $\mathcal{S}[i,t]$, we can multiply the probabilities of all the events in the intersection:

$$\Pr\left[\mathcal{S}[i,t]\right] = \Pr\left[\mathcal{A}[i,t]\right] \cdot \prod_{j \neq i} \Pr\left[\overline{\mathcal{A}[j,t]}\right] = p(1-p)^{n-1}.$$

We now have a nice, closed-form expression for the probability that P_i succeeds in accessing the database in round t; we can now ask how to set p so that this success probability is maximized. Observe first that the success probability is 0 for the extreme cases $p = 0$ and $p = 1$ (these correspond to the extreme case in which processes never bother attempting, and the opposite extreme case in which every process tries accessing the database in every round, so that everyone is locked out). The function $f(p) = p(1-p)^{n-1}$ is positive for values of p strictly between 0 and 1, and its derivative $f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$ has a single zero at the value $p = 1/n$, where the maximum is achieved. Thus we can maximize the success probability by setting $p = 1/n$. (Notice that $p = 1/n$ is a natural intuitive choice as well, if one wants exactly one process to attempt an access in any round.)

When we set $p = 1/n$, we get $\Pr\left[\mathcal{S}[i,t]\right] = \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$. It's worth getting a sense for the asymptotic value of this expression, with the help of the following extremely useful fact from basic calculus.

(13.1)

(a) *The function* $\left(1-\frac{1}{n}\right)^n$ *converges monotonically from* $\frac{1}{4}$ *up to* $\frac{1}{e}$ *as* n *increases from* 2.

(b) *The function* $\left(1-\frac{1}{n}\right)^{n-1}$ *converges monotonically from* $\frac{1}{2}$ *down to* $\frac{1}{e}$ *as* n *increases from* 2.

Using (13.1), we see that $1/(en) \leq \Pr\left[\mathcal{S}[i,t]\right] \leq 1/(2n)$, and hence $\Pr\left[\mathcal{S}[i,t]\right]$ is asymptotically equal to $\Theta(1/n)$.

Waiting for a Particular Process to Succeed Let's consider this protocol with the optimal value $p = 1/n$ for the access probability. Suppose we are interested in how long it will take process P_i to succeed in accessing the database at least once. We see from the earlier calculation that the probability of its succeeding in any one round is not very good, if n is reasonably large. How about if we consider multiple rounds?

Let $\mathcal{F}[i,t]$ denote the "failure event" that process P_i does not succeed in *any* of the rounds 1 through t. This is clearly just the intersection of the complementary events $\overline{\mathcal{S}[i,r]}$ for $r = 1, 2, \ldots, t$. Moreover, since each of these events is independent, we can compute the probability of $\mathcal{F}[i,t]$ by multiplication:

$$\Pr\left[\mathcal{F}[i,t]\right] = \Pr\left[\bigcap_{r=1}^{t} \overline{\mathcal{S}[i,r]}\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] = \left[1-\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}\right]^t.$$

This calculation does give us the value of the probability; but at this point, we're in danger of ending up with some extremely complicated-looking expressions, and so it's important to start thinking asymptotically. Recall that the probability of success was $\Theta(1/n)$ after one round; specifically, it was bounded between $1/(en)$ and $1/(2n)$. Using the expression above, we have

$$\Pr\left[\mathcal{F}[i,t]\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] \leq \left(1-\frac{1}{en}\right)^t.$$

Now we notice that if we set $t = en$, then we have an expression that can be plugged directly into (13.1). Of course en will not be an integer; so we can take $t = \lceil en \rceil$ and write

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1-\frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1-\frac{1}{en}\right)^{en} \leq \frac{1}{e}.$$

This is a very compact and useful asymptotic statement: The probability that process P_i does not succeed in any of rounds 1 through $\lceil en \rceil$ is upper-bounded by the constant e^{-1}, independent of n. Now, if we increase t by some fairly small factors, the probability that P_i does not succeed in any of rounds 1 through t drops precipitously: If we set $t = \lceil en \rceil \cdot (c \ln n)$, then we have

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^t = \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}.$$

So, asymptotically, we can view things as follows. After $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant; and between then and $\Theta(n \ln n)$, this probability drops to a quantity that is quite small, bounded by an inverse polynomial in n.

Waiting for All Processes to Get Through Finally, we're in a position to ask the question that was implicit in the overall setup: How many rounds must elapse before there's a high probability that all processes will have succeeded in accessing the database at least once?

To address this, we say that the protocol *fails* after t rounds if some process has not yet succeeded in accessing the database. Let $\mathcal{F}_t$ denote the event that the protocol fails after t rounds; the goal is to find a reasonably small value of t for which $\Pr\left[\mathcal{F}_t\right]$ is small.

The event $\mathcal{F}_t$ occurs if and only if one of the events $\mathcal{F}[i,t]$ occurs; so we can write

$$\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t].$$

Previously, we considered intersections of independent events, which were very simple to work with; here, by contrast, we have a union of events that are not independent. Probabilities of unions like this can be very hard to compute exactly, and in many settings it is enough to analyze them using a simple *Union Bound*, which says that the probability of a union of events is upper-bounded by the sum of their individual probabilities:

(13.2) (The Union Bound) *Given events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, we have*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Note that this is not an equality; but the upper bound is good enough when, as here, the union on the left-hand side represents a "bad event" that

we're trying to avoid, and we want a bound on its probability in terms of constituent "bad events" on the right-hand side.

For the case at hand, recall that $\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$, and so

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right].$$

The expression on the right-hand side is a sum of n terms, each with the same value; so to make the probability of $\mathcal{F}_t$ small, we need to make each of the terms on the right significantly smaller than $1/n$. From our earlier discussion, we see that choosing $t = \Theta(n)$ will not be good enough, since then each term on the right is only bounded by a constant. If we choose $t = \lceil en \rceil \cdot (c \ln n)$, then we have $\Pr\left[\mathcal{F}[i,t]\right] \leq n^{-c}$ for each i, which is what we want. Thus, in particular, taking $t = 2\lceil en \rceil \ln n$ gives us

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right] \leq n \cdot n^{-2} = n^{-1},$$

and so we have shown the following.

(13.3) *With probability at least $1 - n^{-1}$, all processes succeed in accessing the database at least once within $t = 2\lceil en \rceil \ln n$ rounds.*

An interesting observation here is that if we had chosen a value of t equal to $qn \ln n$ for a very small value of q (rather than the coefficient $2e$ that we actually used), then we would have gotten an upper bound for $\Pr\left[\mathcal{F}[i,t]\right]$ that was larger than n^{-1}, and hence a corresponding upper bound for the overall failure probability $\Pr\left[\mathcal{F}_t\right]$ that was larger than 1—in other words, a completely worthless bound. Yet, as we saw, by choosing larger and larger values for the coefficient q, we can drive the upper bound on $\Pr\left[\mathcal{F}_t\right]$ down to n^{-c} for any constant c we want; and this is really a very tiny upper bound. So, in a sense, all the "action" in the Union Bound takes place rapidly in the period when $t = \Theta(n \ln n)$; as we vary the hidden constant inside the $\Theta(\cdot)$, the Union Bound goes from providing no information to giving an extremely strong upper bound on the probability.

We can ask whether this is simply an artifact of using the Union Bound for our upper bound, or whether it's intrinsic to the process we're observing. Although we won't do the (somewhat messy) calculations here, one can show that when t is a small constant times $n \ln n$, there really is a sizable probability that some process has not yet succeeded in accessing the database. So a rapid falling-off in the value of $\Pr\left[\mathcal{F}_t\right]$ genuinely does happen over the range $t = \Theta(n \ln n)$. For this problem, as in many problems of this flavor, we're

really identifying the asymptotically "correct" value of t despite our use of the seemingly weak Union Bound.

13.2 Finding the Global Minimum Cut

Randomization naturally suggested itself in the previous example, since we were assuming a model with many processes that could not directly communicate. We now look at a problem on graphs for which a randomized approach comes as somewhat more of a surprise, since it is a problem for which perfectly reasonable deterministic algorithms exist as well.

The Problem

Given an undirected graph $G = (V, E)$, we define a *cut* of G to be a partition of V into two non-empty sets A and B. Earlier, when we looked at network flows, we worked with the closely related definition of an *s-t cut*: there, given a directed graph $G = (V, E)$ with distinguished source and sink nodes s and t, an *s-t cut* was defined to be a partition of V into sets A and B such that $s \in A$ and $t \in B$. Our definition now is slightly different, since the underlying graph is now undirected and there is no source or sink.

For a cut (A, B) in an undirected graph G, the *size* of (A, B) is the number of edges with one end in A and the other in B. A *global minimum cut* (or "global min-cut" for short) is a cut of minimum size. The term *global* here is meant to connote that any cut of the graph is allowed; there is no source or sink. Thus the global min-cut is a natural "robustness" parameter; it is the smallest number of edges whose deletion disconnects the graph. We first check that network flow techniques are indeed sufficient to find a global min-cut.

(13.4) *There is a polynomial-time algorithm to find a global min-cut in an undirected graph G.*

Proof. We start from the similarity between cuts in undirected graphs and *s-t* cuts in directed graphs, and with the fact that we know how to find the latter optimally.

So given an undirected graph $G = (V, E)$, we need to transform it so that there are directed edges and there is a source and sink. We first replace every undirected edge $e = (u, v) \in E$ with two oppositely oriented directed edges, $e' = (u, v)$ and $e'' = (v, u)$, each of capacity 1. Let G' denote the resulting directed graph.

Now suppose we pick two arbitrary nodes $s, t \in V$, and find the minimum *s-t* cut in G'. It is easy to check that if (A, B) is this minimum cut in G', then (A, B) is also a cut of minimum size in G *among all those that separate s from t*. But we know that the global min-cut in G must separate s from *something*,

since both sides A and B are nonempty, and s belongs to only one of them. So we fix any $s \in V$ and compute the minimum s-t cut in G' for every other node $t \in V - \{s\}$. This is $n - 1$ directed minimum-cut computations, and the best among these will be a global min-cut of G. ■

The algorithm in (13.4) gives the strong impression that finding a global min-cut in an undirected graph is in some sense a *harder* problem than finding a minimum s-t cut in a flow network, as we had to invoke a subroutine for the latter problem $n - 1$ times in our method for solving the former. But it turns out that this is just an illusion. A sequence of increasingly simple algorithms in the late 1980s and early 1990s showed that global min-cuts in undirected graphs could actually be computed just as efficiently as s-t cuts or even more so, and by techniques that didn't require augmenting paths or even a notion of flow. The high point of this line of work came with David Karger's discovery in 1992 of the Contraction Algorithm, a randomized method that is qualitatively simpler than all previous algorithms for global min-cuts. Indeed, it is sufficiently simple that, on a first impression, it is very hard to believe that it actually works.

Designing the Algorithm

Here we describe the Contraction Algorithm in its simplest form. This version, while it runs in polynomial time, is not among the most efficient algorithms for global min-cuts. However, subsequent optimizations to the algorithm have given it a much better running time.

The Contraction Algorithm works with a connected *multigraph* $G = (V, E)$; this is an undirected graph that is allowed to have multiple "parallel" edges between the same pair of nodes. It begins by choosing an edge $e = (u, v)$ of G uniformly at random and *contracting* it, as shown in Figure 13.1. This means we produce a new graph G' in which u and v have been identified into a single new node w; all other nodes keep their identity. Edges that had one end equal to u and the other equal to v are deleted from G'. Each other edge e is preserved in G', but if one of its ends was equal to u or v, then this end is updated to be equal to the new node w. Note that, even if G had at most one edge between any two nodes, G' may end up with parallel edges.

The Contraction Algorithm then continues recursively on G', choosing an edge uniformly at random and contracting it. As these recursive calls proceed, the constituent vertices of G' should be viewed as *supernodes*: Each supernode w corresponds to the subset $S(w) \subseteq V$ that has been "swallowed up" in the contractions that produced w. The algorithm terminates when it reaches a graph G' that has only two supernodes v_1 and v_2 (presumably with a number of parallel edges between them). Each of these super-nodes v_i has a corresponding subset $S(v_i) \subseteq V$ consisting of the nodes that have been

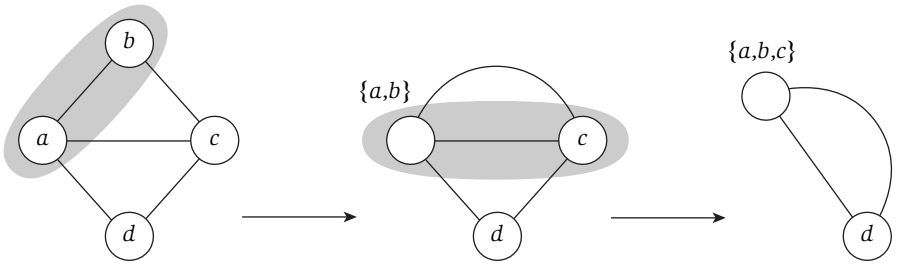

Figure 13.1 The Contraction Algorithm applied to a four-node input graph.

contracted into it, and these two sets $S(v_1)$ and $S(v_2)$ form a partition of V. We output $(S(v_1), S(v_2))$ as the cut found by the algorithm.

```
The Contraction Algorithm applied to a multigraph G = (V, E):
  For each node v, we will record
          the set S(v) of nodes that have been contracted into v
       Initially S(v) = {v} for each v
  If G has two nodes v1 and v2, then return the cut (S(v1), S(v2))
  Else choose an edge e = (u, v) of G uniformly at random
       Let G' be the graph resulting from the contraction of e,
           with a new node z_uv replacing u and v
       Define S(z_uv) = S(u) ∪ S(v)
       Apply the Contraction Algorithm recursively to G'
  Endif
```

Analyzing the Algorithm

The algorithm is making random choices, so there is some probability that it will succeed in finding a global min-cut and some probability that it won't. One might imagine at first that the probability of success is exponentially small. After all, there are exponentially many possible cuts of G; what's favoring the minimum cut in the process? But we'll show first that, in fact, the success probability is only polynomially small. It will then follow that by running the algorithm a polynomial number of times and returning the best cut found in any run, we can actually produce a global min-cut with high probability.

(13.5) *The Contraction Algorithm returns a global min-cut of G with probability at least $1/\binom{n}{2}$.*

Proof. We focus on a global min-cut (A, B) of G and suppose it has size k; in other words, there is a set F of k edges with one end in A and the other

in B. We want to give a lower bound on the probability that the Contraction Algorithm returns the cut (A, B).

Consider what could go wrong in the first step of the Contraction Algorithm: The problem would be if an edge in F were contracted. For then, a node of A and a node of B would get thrown together in the same supernode, and (A, B) could not be returned as the output of the algorithm. Conversely, if an edge not in F is contracted, then there is still a chance that (A, B) could be returned.

So what we want is an upper bound on the probability that an edge in F is contracted, and for this we need a lower bound on the size of E. Notice that if any node v had degree less than k, then the cut $(\{v\}, V - \{v\})$ would have size less than k, contradicting our assumption that (A, B) is a global min-cut. Thus every node in G has degree at least k, and so $|E| \geq \frac{1}{2}kn$. Hence the probability that an edge in F is contracted is at most

$$\frac{k}{\frac{1}{2}kn} = \frac{2}{n}.$$

Now consider the situation after j iterations, when there are $n - j$ supernodes in the current graph G', and suppose that no edge in F has been contracted yet. Every cut of G' is a cut of G, and so there are at least k edges incident to every supernode of G'. Thus G' has at least $\frac{1}{2}k(n - j)$ edges, and so the probability that an edge of F is contracted in the next iteration $j + 1$ is at most

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}.$$

The cut (A, B) will actually be returned by the algorithm if no edge of F is contracted in any of iterations $1, 2, \ldots, n - 2$. If we write $\mathcal{E}_j$ for the event that an edge of F is not contracted in iteration j, then we have shown $\Pr[\mathcal{E}_1] \geq 1 - 2/n$ and $\Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \geq 1 - 2/(n - j)$. We are interested in lower-bounding the quantity $\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-2}]$, and we can check by unwinding the formula for conditional probability that this is equal to

$$\begin{aligned}
&\Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \cdots \Pr[\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-3}] \\
&\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{n-j}\right) \cdots \left(1 - \frac{2}{3}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)} = \binom{n}{2}^{-1}. \quad \blacksquare
\end{aligned}$$

So we now know that a single run of the Contraction Algorithm fails to find a global min-cut with probability at most $(1 - 1/\binom{n}{2})$. This number is very close to 1, of course, but we can amplify our probability of success simply by repeatedly running the algorithm, with independent random choices, and taking the best cut we find. By fact (13.1), if we run the algorithm $\binom{n}{2}$ times, then the probability that we fail to find a global min-cut in any run is at most

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2}} \leq \frac{1}{e}.$$

And it's easy to drive the failure probability below $1/e$ with further repetitions: If we run the algorithm $\binom{n}{2} \ln n$ times, then the probability we fail to find a global min-cut is at most $e^{-\ln n} = 1/n$.

The overall running time required to get a high probability of success is polynomial in n, since each run of the Contraction Algorithm takes polynomial time, and we run it a polynomial number of times. Its running time will be fairly large compared with the best network flow techniques, since we perform $\Theta(n^2)$ independent runs and each takes at least $\Omega(m)$ time. We have chosen to describe this version of the Contraction Algorithm since it is the simplest and most elegant; it has been shown that some clever optimizations to the way in which multiple runs are performed can improve the running time considerably.

Further Analysis: The Number of Global Minimum Cuts

The analysis of the Contraction Algorithm provides a surprisingly simple answer to the following question: Given an undirected graph $G = (V, E)$ on n nodes, what is the maximum number of global min-cuts it can have (as a function of n)?

For a directed flow network, it's easy to see that the number of minimum s-t cuts can be exponential in n. For example, consider a directed graph with nodes $s, t, v_1, v_2, \ldots, v_n$, and unit-capacity edges (s, v_i) and (v_i, t) for each i. Then s together with any subset of $\{v_1, v_2, \ldots, v_n\}$ will constitute the source side of a minimum cut, and so there are 2^n minimum s-t cuts.

But for global min-cuts in an undirected graph, the situation looks quite different. If one spends some time trying out examples, one finds that the n-node cycle has $\binom{n}{2}$ global min-cuts (obtained by cutting any two edges), and it is not clear how to construct an undirected graph with more.

We now show how the analysis of the Contraction Algorithm settles this question immediately, establishing that the n-node cycle is indeed an extreme case.

(13.6) *An undirected graph $G = (V, E)$ on n nodes has at most $\binom{n}{2}$ global min-cuts.*

Proof. The key is that the proof of (13.5) actually established more than was claimed. Let G be a graph, and let $C_1, \ldots, C_r$ denote all its global min-cuts. Let $\mathcal{E}_i$ denote the event that C_i is returned by the Contraction Algorithm, and let $\mathcal{E} = \cup_{i=1}^{r} \mathcal{E}_i$ denote the event that the algorithm returns any global min-cut.

Then, although (13.5) simply asserts that $\Pr[\mathcal{E}] \geq 1/\binom{n}{2}$, its proof actually shows that for each i, we have $\Pr[\mathcal{E}_i] \geq 1/\binom{n}{2}$. Now each pair of events $\mathcal{E}_i$ and $\mathcal{E}_j$ are disjoint—since only one cut is returned by any given run of the algorithm—so by the Union Bound for disjoint events (13.49), we have

$$\Pr[\mathcal{E}] = \Pr\left[\cup_{i=1}^{r} \mathcal{E}_i\right] = \sum_{i=1}^{r} \Pr[\mathcal{E}_i] \geq r/\binom{n}{2}.$$

But clearly $\Pr[\mathcal{E}] \leq 1$, and so we must have $r \leq \binom{n}{2}$. ∎

13.3 Random Variables and Their Expectations

Thus far our analysis of randomized algorithms and processes has been based on identifying certain "bad events" and bounding their probabilities. This is a qualitative type of analysis, in the sense that the algorithm either succeeds or it doesn't. A more quantitative style of analysis would consider certain parameters associated with the behavior of the algorithm—for example, its running time, or the quality of the solution it produces—and seek to determine the *expected* size of these parameters over the random choices made by the algorithm. In order to make such analysis possible, we need the fundamental notion of a *random variable*.

Given a probability space, a random variable X is a function from the underlying sample space to the natural numbers, such that for each natural number j, the set $X^{-1}(j)$ of all sample points taking the value j is an event. Thus we can write $\Pr[X = j]$ as loose shorthand for $\Pr[X^{-1}(j)]$; it is because we can ask about X's probability of taking a given value that we think of it as a "random variable."

Given a random variable X, we are often interested in determining its *expectation*—the "average value" assumed by X. We define this as

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j],$$

declaring this to have the value ∞ if the sum diverges. Thus, for example, if X takes each of the values in $\{1, 2, \ldots, n\}$ with probability $1/n$, then $E[X] = 1(1/n) + 2(1/n) + \cdots + n(1/n) = \binom{n+1}{2}/n = (n+1)/2$.

Example: Waiting for a First Success

Here's a more useful example, in which we see how an appropriate random variable lets us talk about something like the "running time" of a simple random process. Suppose we have a coin that comes up `heads` with probability $p > 0$, and `tails` with probability $1 - p$. Different flips of the coin have independent outcomes. If we flip the coin until we first get a `heads`, what's the expected number of flips we will perform? To answer this, we let X denote the random variable equal to the number of flips performed. For $j > 0$, we have $\Pr[X = j] = (1-p)^{j-1}p$: in order for the process to take exactly j steps, the first $j - 1$ flips must come up `tails`, and the j^{th} must come up `heads`. Now, applying the definition, we have

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \sum_{j=1}^{\infty} j(1-p)^{j-1}p = \frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^{j}$$
$$= \frac{p}{1-p} \cdot \frac{(1-p)}{p^2} = \frac{1}{p}.$$

Thus we get the following intuitively sensible result.

(13.7) *If we repeatedly perform independent trials of an experiment, each of which succeeds with probability $p > 0$, then the expected number of trials we need to perform until the first success is $1/p$.*

Linearity of Expectation

In Sections 13.1 and 13.2, we broke events down into unions of much simpler events, and worked with the probabilities of these simpler events. This is a powerful technique when working with random variables as well, and it is based on the principle of *linearity of expectation*.

(13.8) Linearity of Expectation. *Given two random variables X and Y defined over the same probability space, we can define $X + Y$ to be the random variable equal to $X(\omega) + Y(\omega)$ on a sample point ω. For any X and Y, we have*

$$E[X + Y] = E[X] + E[Y].$$

We omit the proof, which is not difficult. Much of the power of (13.8) comes from the fact that it applies to the sum of *any* random variables; no restrictive assumptions are needed. As a result, if we need to compute the

expectation of a complicated random variable X, we can first write it as a sum of simpler random variables $X = X_1 + X_2 + \cdots + X_n$, compute each $E\left[X_i\right]$, and then determine $E[X] = \sum E\left[X_i\right]$. We now look at some examples of this principle in action.

Example: Guessing Cards

Memoryless Guessing To amaze your friends, you have them shuffle a deck of 52 cards and then turn over one card at a time. Before each card is turned over, you predict its identity. Unfortunately, you don't have any particular psychic abilities—and you're not so good at remembering what's been turned over already—so your strategy is simply to guess a card uniformly at random from the full deck each time. On how many predictions do you expect to be correct?

Let's work this out for the more general setting in which the deck has n distinct cards, using X to denote the random variable equal to the number of correct predictions. A surprisingly effortless way to compute X is to define the random variable X_i, for $i = 1, 2, \ldots, n$, to be equal to 1 if the i^{th} prediction is correct, and 0 otherwise. Notice that $X = X_1 + X_2 + \cdots + X_n$, and

$$E\left[X_i\right] = 0 \cdot \Pr\left[X_i = 0\right] + 1 \cdot \Pr\left[X_i = 1\right] = \Pr\left[X_i = 1\right] = \frac{1}{n}.$$

It's worth pausing to note a useful fact that is implicitly demonstrated by the above calculation: If Z is any random variable that only takes the values 0 or 1, then $E[Z] = \Pr[Z = 1]$.

Since $E\left[X_i\right] = \frac{1}{n}$ for each i, we have

$$E[X] = \sum_{i=1}^{n} E\left[X_i\right] = n\left(\frac{1}{n}\right) = 1.$$

Thus we have shown the following.

(13.9) *The expected number of correct predictions under the memoryless guessing strategy is* 1*, independent of* n*.*

Trying to compute $E[X]$ directly from the definition $\sum_{j=0}^{\infty} j \cdot \Pr[X = j]$ would be much more painful, since it would involve working out a much more elaborate summation. A significant amount of complexity is hidden away in the seemingly innocuous statement of (13.8).

Guessing with Memory Now let's consider a second scenario. Your psychic abilities have not developed any further since last time, but you have become very good at remembering which cards have already been turned over. Thus, when you predict the next card now, you only guess uniformly from among

the cards *not yet seen*. How many correct predictions do you expect to make with this strategy?

Again, let the random variable X_i take the value 1 if the i^{th} prediction is correct, and 0 otherwise. In order for the i^{th} prediction to be correct, you need only guess the correct one out of $n - i + 1$ remaining cards; hence

$$E\left[X_i\right] = \Pr\left[X_i = 1\right] = \frac{1}{n-i+1},$$

and so we have

$$\Pr\left[X\right] = \sum_{i=1}^{n} E\left[X_i\right] = \sum_{i=1}^{n} \frac{1}{n-i+1} = \sum_{i=1}^{n} \frac{1}{i}.$$

This last expression $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is the *harmonic number* $H(n)$, and it is something that has come up in each of the previous two chapters. In particular, we showed in Chapter 11 that $H(n)$, as a function of n, closely shadows the value $\int_1^{n+1} \frac{1}{x}\,dx = \ln(n+1)$. For our purposes here, we restate the basic bound on $H(n)$ as follows.

(13.10) $\ln(n+1) < H(n) < 1 + \ln n$, *and more loosely,* $H(n) = \Theta(\log n)$.

Thus, once you are able to remember the cards you've already seen, the expected number of correct predictions increases significantly above 1.

(13.11) *The expected number of correct predictions under the guessing strategy with memory is* $H(n) = \Theta(\log n)$.

Example: Collecting Coupons

Before moving on to more sophisticated applications, let's consider one more basic example in which linearity of expectation provides significant leverage.

Suppose that a certain brand of cereal includes a free coupon in each box. There are n different types of coupons. As a regular consumer of this brand, how many boxes do you expect to buy before finally getting a coupon of each type?

Clearly, at least n boxes are needed; but it would be sort of surprising if you actually had all n types of coupons by the time you'd bought n boxes. As you collect more and more different types, it will get less and less likely that a new box has a type of coupon you haven't seen before. Once you have $n - 1$ of the n different types, there's only a probability of $1/n$ that a new box has the missing type you need.

Here's a way to work out the expected time exactly. Let X be the random variable equal to the number of boxes you buy until you first have a coupon

of each type. As in our previous examples, this is a reasonably complicated random variable to think about, and we'd like to write it as a sum of simpler random variables. To think about this, let's consider the following natural idea: The coupon-collecting process *makes progress* whenever you buy a box of cereal containing a type of coupon you haven't seen before. Thus the goal of the process is really to make progress n times. Now, at a given point in time, what is the probability that you make progress in the next step? This depends on how many different types of coupons you already have. If you have j types, then the probability of making progress in the next step is $(n-j)/n$: Of the n types of coupons, $n-j$ allow you to make progress. Since the probability varies depending on the number of different types of coupons we have, this suggests a natural way to break down X into simpler random variables, as follows.

Let's say that the coupon-collecting process is in *phase* j when you've already collected j different types of coupons and are waiting to get a new type. When you see a new type of coupon, phase j ends and phase $j+1$ begins. Thus we start in phase 0, and the whole process is done at the end of phase $n-1$. Let X_j be the random variable equal to the number of steps you spend in phase j. Then $X = X_0 + X_1 + \cdots + X_{n-1}$, and so it is enough to work out $E[X_j]$ for each j.

(13.12) $E[X_j] = n/(n-j)$.

Proof. In each step of phase j, the phase ends immediately if and only if the coupon you get next is one of the $n-j$ types you haven't seen before. Thus, in phase j, you are really just waiting for an event of probability $(n-j)/n$ to occur, and so, by (13.7), the expected length of phase j is $E[X_j] = n/(n-j)$. ■

Using this, linearity of expectation gives us the overall expected time.

(13.13) *The expected time before all n types of coupons are collected is* $E[X] = nH(n) = \Theta(n \log n)$.

Proof. By linearity of expectation, we have

$$E[X] = \sum_{j=0}^{n-1} E[X_j] = \sum_{j=0}^{n-1} \frac{n}{n-j} = n \sum_{j=0}^{n-1} \frac{1}{n-j} = n \sum_{i=1}^{n} \frac{1}{i} = nH(n).$$

By (13.10), we know this is asymptotically equal to $\Theta(n \log n)$. ■

It is interesting to compare the dynamics of this process to one's intuitive view of it. Once $n-1$ of the n types of coupons are collected, you expect to

buy n more boxes of cereal before you see the final type. In the meantime, you keep getting coupons you've already seen before, and you might conclude that this final type is "the rare one." But in fact it's just as likely as all the others; it's simply that the final one, whichever it turns out to be, is likely to take a long time to get.

A Final Definition: Conditional Expectation

We now discuss one final, very useful notion concerning random variables that will come up in some of the subsequent analyses. Just as one can define the conditional probability of one event given another, one can analogously define the expectation of a random variable conditioned on a certain event. Suppose we have a random variable X and an event $\mathcal{E}$ of positive probability. Then we define the *conditional expectation* of X, given $\mathcal{E}$, to be the expected value of X computed only over the part of the sample space corresponding to $\mathcal{E}$. We denote this quantity by $E[X \mid \mathcal{E}]$. This simply involves replacing the probabilities $\Pr[X = j]$ in the definition of the expectation with conditional probabilities:

$$E[X \mid \mathcal{E}] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j \mid \mathcal{E}].$$

13.4 A Randomized Approximation Algorithm for MAX 3-SAT

In the previous section, we saw a number of ways in which linearity of expectation can be used to analyze a randomized process. We now describe an application of this idea to the design of an approximation algorithm. The problem we consider is a variation of the 3-SAT Problem, and we will see that one consequence of our randomized approximation algorithm is a surprisingly strong general statement about 3-SAT that on its surface seems to have nothing to do with either algorithms or randomization.

The Problem

When we studied NP-completeness, a core problem was 3-SAT: Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Intuitively, we can imagine such a problem arising in a system that tries to decide the truth or falsehood of statements about the world (the variables $\{x_i\}$), given pieces of information that relate them to one another (the clauses $\{C_j\}$). Now the world is a fairly contradictory place, and if our system gathers

enough information, it could well end up with a set of clauses that has no satisfying truth assignment. What then?

A natural approach, if we can't find a truth assignment that satisfies all clauses, is to turn the 3-SAT instance into an optimization problem: Given the set of input clauses $C_1, \ldots, C_k$, find a truth assignment that satisfies *as many as possible*. We'll call this the *Maximum 3-Satisfiability Problem* (or *MAX 3-SAT* for short). Of course, this is an NP-hard optimization problem, since it's NP-complete to decide whether the maximum number of simultaneously satisfiable clauses is equal to k. Let's see what can be said about polynomial-time approximation algorithms.

Designing and Analyzing the Algorithm

A remarkably simple randomized algorithm turns out to give a strong performance guarantee for this problem. Suppose we set each variable $x_1, \ldots, x_n$ independently to 0 or 1 with probability $\frac{1}{2}$ each. What is the expected number of clauses satisfied by such a random assignment?

Let Z denote the random variable equal to the number of satisfied clauses. As in Section 13.3, let's decompose Z into a sum of random variables that each take the value 0 or 1; specifically, let $Z_i = 1$ if the clause C_i is satisfied, and 0 otherwise. Thus $Z = Z_1 + Z_2 + \cdots + Z_k$. Now $E\left[Z_i\right]$ is equal to the probability that C_i is satisfied, and this can be computed easily as follows. In order for C_i *not* to be satisfied, each of its three variables must be assigned the value that fails to make it true; since the variables are set independently, the probability of this is $(\frac{1}{2})^3 = \frac{1}{8}$. Thus clause C_i is satisfied with probability $1 - \frac{1}{8} = \frac{7}{8}$, and so $E\left[Z_i\right] = \frac{7}{8}$.

Using linearity of expectation, we see that the expected number of satisfied clauses is $E[Z] = E\left[Z_1\right] + E\left[Z_2\right] + \cdots + E\left[Z_k\right] = \frac{7}{8}k$. Since no assignment can satisfy more than k clauses, we have the following guarantee.

(13.14) *Consider a 3-SAT formula, where each clause has three different variables. The expected number of clauses satisfied by a random assignment is within an approximation factor $\frac{7}{8}$ of optimal.*

But, if we look at what really happened in the (admittedly simple) analysis of the random assignment, it's clear that something stronger is going on. For any random variable, there must be some point at which it assumes some value at least as large as its expectation. We've shown that for every instance of 3-SAT, a random truth assignment satisfies a $\frac{7}{8}$ fraction of all clauses in expectation; so, in particular, there must *exist* a truth assignment that satisfies a number of clauses that is at least as large as this expectation.

(13.15) *For every instance of 3-SAT, there is a truth assignment that satisfies at least a $\frac{7}{8}$ fraction of all clauses.*

There is something genuinely surprising about the statement of (13.15). We have arrived at a nonobvious fact about 3-SAT—the existence of an assignment satisfying many clauses—whose statement has nothing to do with randomization; but we have done so by a randomized construction. And, in fact, the randomized construction provides what is quite possibly the simplest proof of (13.15). This is a fairly widespread principle in the area of combinatorics—namely, that one can show the existence of some structure by showing that a random construction produces it with positive probability. Constructions of this sort are said to be applications of the *probabilistic method*.

Here's a cute but minor application of (13.15): Every instance of 3-SAT with at most seven clauses is satisfiable. Why? If the instance has $k \leq 7$ clauses, then (13.15) implies that there is an assignment satisfying at least $\frac{7}{8}k$ of them. But when $k \leq 7$, it follows that $\frac{7}{8}k > k - 1$; and since the number of clauses satisfied by this assignment must be an integer, it must be equal to k. In other words, all clauses are satisfied.

Further Analysis: Waiting to Find a Good Assignment

Suppose we aren't satisfied with a "one-shot" algorithm that produces a single assignment with a large number of satisfied clauses in expectation. Rather, we'd like a randomized algorithm whose expected running time is polynomial and that is guaranteed to output a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.

A simple way to do this is to generate random truth assignments until one of them satisfies at least $\frac{7}{8}k$ clauses. We know that such an assignment exists, by (13.15); but how long will it take until we find one by random trials?

This is a natural place to apply the waiting-time bound we derived in (13.7). If we can show that the probability a random assignment satisfies at least $\frac{7}{8}k$ clauses is at least p, then the expected number of trials performed by the algorithm is $1/p$. So, in particular, we'd like to show that this quantity p is at least as large as an inverse polynomial in n and k.

For $j = 0, 1, 2, \ldots, k$, let p_j denote the probability that a random assignment satisfies exactly j clauses. So the expected number of clauses satisfied, by the definition of expectation, is equal to $\sum_{j=0}^{k} jp_j$; and by the previous analysis, this is equal to $\frac{7}{8}k$. We are interested in the quantity $p = \sum_{j \geq 7k/8} p_j$. How can we use the lower bound on the expected value to give a lower bound on this quantity?

We start by writing

$$\frac{7}{8}k = \sum_{j=0}^{k} jp_j = \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j.$$

Now let k' denote the largest natural number that is strictly smaller than $\frac{7}{8}k$. The right-hand side of the above equation only increases if we replace the terms in the first sum by $k'p_j$ and the terms in the second sum by kp_j. We also observe that $\sum_{j<7k/8} p_j = 1 - p$, and so

$$\frac{7}{8}k \leq \sum_{j<7k/8} k'p_j + \sum_{j\geq 7k/8} kp_j = k'(1-p) + kp \leq k' + kp,$$

and hence $kp \geq \frac{7}{8}k - k'$. But $\frac{7}{8}k - k' \geq \frac{1}{8}$, since k' is a natural number strictly smaller than $\frac{7}{8}$ times another natural number, and so

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8k}.$$

This was our goal—to get a lower bound on p—and so by the waiting-time bound (13.7), we see that the expected number of trials needed to find the satisfying assignment we want is at most $8k$.

(13.16) *There is a randomized algorithm with polynomial expected running time that is guaranteed to produce a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.*

13.5 Randomized Divide and Conquer: Median-Finding and Quicksort

We've seen the divide-and-conquer paradigm for designing algorithms at various earlier points in the book. Divide and conquer often works well in conjunction with randomization, and we illustrate this by giving divide-and-conquer algorithms for two fundamental problems: computing the median of n numbers, and sorting. In each case, the "divide" step is performed using randomization; consequently, we will use expectations of random variables to analyze the time spent on recursive calls.

The Problem: Finding the Median

Suppose we are given a set of n numbers $S = \{a_1, a_2, \ldots, a_n\}$. Their *median* is the number that would be in the middle position if we were to sort them. There's an annoying technical difficulty if n is even, since then there is no

"middle position"; thus we define things precisely as follows: The median of $S = \{a_1, a_2, \ldots, a_n\}$ is equal to the k^{th} largest element in S, where $k = (n+1)/2$ if n is odd, and $k = n/2$ if n is even. In what follows, we'll assume for the sake of simplicity that all the numbers are distinct. Without this assumption, the problem becomes notationally more complicated, but no new ideas are brought into play.

It is clearly easy to compute the median in time $O(n \log n)$ if we simply sort the numbers first. But if one begins thinking about the problem, it's far from clear why sorting is *necessary* for computing the median, or even why $\Omega(n \log n)$ time is necessary. In fact, we'll show how a simple randomized approach, based on divide-and-conquer, yields an expected running time of $O(n)$.

Designing the Algorithm

A Generic Algorithm Based on Splitters The first key step toward getting an expected linear running time is to move from median-finding to the more general problem of *selection*. Given a set of n numbers S and a number k between 1 and n, consider the function `Select`(S, k) that returns the k^{th} largest element in S. As special cases, `Select` includes the problem of finding the median of S via `Select`$(S, n/2)$ or `Select`$(S, (n+1)/2)$; it also includes the easier problems of finding the minimum (`Select`$(S, 1)$) and the maximum (`Select`(S, n)). Our goal is to design an algorithm that implements `Select` so that it runs in expected time $O(n)$.

The basic structure of the algorithm implementing `Select` is as follows. We choose an element $a_i \in S$, the "splitter," and form the sets $S^- = \{a_j : a_j < a_i\}$ and $S^+ = \{a_j : a_j > a_i\}$. We can then determine which of S^- or S^+ contains the k^{th} largest element, and iterate only on this one. Without specifying yet how we plan to choose the splitter, here's a more concrete description of how we form the two sets and iterate.

```
Select(S,k):
Choose a splitter a_i ∈ S
For each element a_j of S
   Put a_j in S^- if a_j < a_i
   Put a_j in S^+ if a_j > a_i
Endfor
If |S^-| = k − 1 then
   The splitter a_i was in fact the desired answer
Else if |S^-| ≥ k then
   The k^th largest element lies in S^-
   Recursively call Select(S^-, k)
```

```
Else suppose |S⁻| = ℓ < k − 1
   The k^th largest element lies in S⁺
   Recursively call Select(S⁺, k − 1 − ℓ)
Endif
```

Observe that the algorithm is always called recursively on a strictly smaller set, so it must terminate. Also, observe that if $|S| = 1$, then we must have $k = 1$, and indeed the single element in S will be returned by the algorithm. Finally, from the choice of which recursive call to make, it's clear by induction that the right answer will be returned when $|S| > 1$ as well. Thus we have the following

(13.17) *Regardless of how the splitter is chosen, the algorithm above returns the k^{th} largest element of S.*

Choosing a Good Splitter Now let's consider how the running time of `Select` depends on the way we choose the splitter. Assuming we can select a splitter in linear time, the rest of the algorithm takes linear time plus the time for the recursive call. But how is the running time of the recursive call affected by the choice of the splitter? Essentially, it's important that the splitter significantly reduce the size of the set being considered, so that we don't keep making passes through large sets of numbers many times. So a good choice of splitter should produce sets S^- and S^+ that are approximately equal in size.

For example, if we could always choose the median as the splitter, then we could show a linear bound on the running time as follows. Let cn be the running time for `Select`, not counting the time for the recursive call. Then, with medians as splitters, the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T(n/2) + cn$. This is a recurrence that we encountered at the beginning of Chapter 5, where we showed that it has the solution $T(n) = O(n)$.

Of course, hoping to be able to use the median as the splitter is rather circular, since the median is what we want to compute in the first place! But, in fact, one can show that any "well-centered" element can serve as a good splitter: If we had a way to choose splitters a_i such that there were at least εn elements both larger and smaller than a_i, for any fixed constant $\varepsilon > 0$, then the size of the sets in the recursive call would shrink by a factor of at least $(1 - \varepsilon)$ each time. Thus the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T((1 - \varepsilon)n) + cn$. The same argument that showed the previous recurrence had the solution $T(n) = O(n)$ can be used here: If we unroll this recurrence for any $\varepsilon > 0$, we get

$$T(n) \leq cn + (1-\varepsilon)cn + (1-\varepsilon)^2 cn + \cdots = \left[1 + (1-\varepsilon) + (1-\varepsilon)^2 + \cdots\right]$$
$$cn \leq \frac{1}{\varepsilon} \cdot cn,$$

since we have a convergent geometric series.

Indeed, the only thing to really beware of is a very "off-center" splitter. For example, if we always chose the minimum element as the splitter, then we may end up with a set in the recursive call that's only one element smaller than we had before. In this case, the running time $T(n)$ would be bounded by the recurrence $T(n) \leq T(n-1) + cn$. Unrolling this recurrence, we see that there's a problem:

$$T(n) \leq cn + c(n-1) + c(n-2) + \cdots = \frac{cn(n+1)}{2} = \Theta(n^2).$$

Random Splitters Choosing a "well-centered" splitter, in the sense we have just defined, is certainly similar in flavor to our original problem of choosing the median; but the situation is really not so bad, since *any* well-centered splitter will do.

Thus we will implement the as-yet-unspecified step of selecting a splitter using the following simple rule:

`Choose a splitter` $a_i \in S$ `uniformly at random`

The intuition here is very natural: since a fairly large fraction of the elements are reasonably well-centered, we will be likely to end up with a good splitter simply by choosing an element at random.

The analysis of the running time with a random splitter is based on this idea; we expect the size of the set under consideration to go down by a fixed constant fraction every iteration, so we should get a convergent series and hence a linear bound as previously. We now show how to make this precise.

Analyzing the Algorithm

We'll say that the algorithm is in *phase j* when the size of the set under consideration is at most $n(\frac{3}{4})^j$ but greater than $n(\frac{3}{4})^{j+1}$. Let's try to bound the expected time spent by the algorithm in phase j. In a given iteration of the algorithm, we say that an element of the set under consideration is *central* if at least a quarter of the elements are smaller than it and at least a quarter of the elements are larger than it.

Now observe that if a central element is chosen as a splitter, then at least a quarter of the set will be thrown away, the set will shrink by a factor of $\frac{3}{4}$ or better, and the current phase will come to an end. Moreover, half of all the

elements in the set are central, and so the probability that our random choice of splitter produces a central element is $\frac{1}{2}$. Hence, by our simple waiting-time bound (13.7), the expected number of iterations before a central element is found is 2; and so the expected number of iterations spent in phase j, for any j, is at most 2.

This is pretty much all we need for the analysis. Let X be a random variable equal to the number of steps taken by the algorithm. We can write it as the sum $X = X_0 + X_1 + X_2 + \cdots$, where X_j is the expected number of steps spent by the algorithm in phase j. When the algorithm is in phase j, the set has size at most $n(\frac{3}{4})^j$, and so the number of steps required for one iteration in phase j is at most $cn(\frac{3}{4})^j$ for some constant c. We have just argued that the expected number of iterations spent in phase j is at most two, and hence we have $E\left[X_j\right] \leq 2cn(\frac{3}{4})^j$. Thus we can bound the total expected running time using linearity of expectation,

$$E\left[X\right] = \sum_j E\left[X_j\right] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j \left(\frac{3}{4}\right)^j \leq 8cn,$$

since the sum $\sum_j (\frac{3}{4})^j$ is a geometric series that converges. Thus we have the following desired result.

(13.18) *The expected running time of* `Select`(n, k) *is* $O(n)$.

A Second Application: Quicksort

The randomized divide-and-conquer technique we used to find the median is also the basis of the sorting algorithm `Quicksort`. As before, we choose a splitter for the input set S, and separate S into the elements below the splitter value and those above it. The difference is that, rather than looking for the median on just one side of the splitter, we sort both sides recursively and glue the two sorted pieces together (with the splitter in between) to produce the overall output. Also, we need to explicitly include a base case for the recursive code: we only use recursion on sets of size at least 4. A complete description of `Quicksort` is as follows.

```
Quicksort(S):
If |S| ≤ 3 then
   Sort S
   Output the sorted list
Else
   Choose a splitter a_i ∈ S uniformly at random
   For each element a_j of S
```

```
        Put a_j in S^- if a_j < a_i
        Put a_j in S^+ if a_j > a_i
    Endfor
    Recursively call Quicksort(S^-) and Quicksort(S^+)
    Output the sorted set S^-, then a_i, then the sorted set S^+
Endif
```

As with median-finding, the worst-case running time of this method is not so good. If we always select the smallest element as a splitter, then the running time $T(n)$ on n-element sets satisfies the same recurrence as before: $T(n) \leq T(n-1) + cn$, and so we end up with a time bound of $T(n) = \Theta(n^2)$. In fact, this is the worst-case running time for `Quicksort`.

On the positive side, if the splitters selected happened to be the medians of the sets at each iteration, then we get the recurrence $T(n) \leq 2T(n/2) + cn$, which arose frequently in the divide-and-conquer analyses of Chapter 5; the running time in this lucky case is $O(n \log n)$.

Here we are concerned with the *expected running time*; we will show that this can be bounded by $O(n \log n)$, almost as good as in the best case when the splitters are perfectly centered. Our analysis of `Quicksort` will closely follow the analysis of median-finding. Just as in the `Select` procedure that we used for median-finding, the crucial definition is that of a *central splitter*—one that divides the set so that each side contains at least a quarter of the elements. (As we discussed earlier, it is enough for the analysis that each side contains at least some fixed constant fraction of the elements; the use of a quarter here is chosen for convenience.) The idea is that a random choice is likely to lead to a central splitter, and central splitters work well. In the case of sorting, a central splitter divides the problem into two considerably smaller subproblems.

To simplify the presentation, we will slightly modify the algorithm so that it only issues its recursive calls when it finds a central splitter. Essentially, this modified algorithm differs from `Quicksort` in that it prefers to throw away an "off-center" splitter and try again; `Quicksort`, by contrast, launches the recursive calls even with an off-center splitter, and at least benefits from the work already done in splitting S. The point is that the expected running time of this modified algorithm can be analyzed very simply, by direct analogy with our analysis for median-finding. With a bit more work, a very similar but somewhat more involved analysis can also be done for the original `Quicksort` algorithm as well; however, we will not describe this analysis here.

```
Modified Quicksort(S):
If |S| ≤ 3 then
    Sort S
```

```
   Output the sorted list
Endif
Else
   While no central splitter has been found
      Choose a splitter a_i ∈ S uniformly at random
      For each element a_j of S
         Put a_j in S^- if a_j < a_i
         Put a_j in S^+ if a_j > a_i
      Endfor
      If |S^-| ≥ |S|/4 and |S^+| ≥ |S|/4 then
         a_i is a central splitter
      Endif
   Endwhile
   Recursively call Quicksort(S^-) and Quicksort(S^+)
   Output the sorted set S^-, then a_i, then the sorted set S^+
Endif
```

Consider a subproblem for some set S. Each iteration of the `While` loop selects a possible splitter a_i and spends $O(|S|)$ time splitting the set and deciding if a_i is central. Earlier we argued that the number of iterations needed until we find a central splitter is at most 2. This gives us the following statement.

(13.19) *The expected running time for the algorithm on a set S, excluding the time spent on recursive calls, is $O(|S|)$.*

The algorithm is called recursively on multiple subproblems. We will group these subproblems by size. We'll say that the subproblem is of *type j* if the size of the set under consideration is at most $n(\frac{3}{4})^j$ but greater than $n(\frac{3}{4})^{j+1}$. By (13.19), the expected time spent on a subproblem of type j, excluding recursive calls, is $O(n(\frac{3}{4})^j)$. To bound the overall running time, we need to bound the number of subproblems for each type j. Splitting a type j subproblem via a central splitter creates two subproblems of higher type. So the subproblems of a given type j are disjoint. This gives us a bound on the number of subproblems.

(13.20) *The number of type j subproblems created by the algorithm is at most $(\frac{4}{3})^{j+1}$.*

There are at most $(\frac{4}{3})^{j+1}$ subproblems of type j, and the expected time spent on each is $O(n(\frac{3}{4})^j)$ by (13.19). Thus, by linearity of expectation, the expected time spent on subproblems of type j is $O(n)$. The number of different types is bounded by $\log_{\frac{4}{3}} n = O(\log n)$, which gives the desired bound.

(13.21) *The expected running time of* `Modified Quicksort` *is $O(n \log n)$.*

We considered this modified version of `Quicksort` to simplify the analysis. Coming back to the original `Quicksort`, our intuition suggests that the expected running time is no worse than in the modified algorithm, as accepting the noncentral splitters helps a bit with sorting, even if it does not help as much as when a central splitter is chosen. As mentioned earlier, one can in fact make this intuition precise, leading to an $O(n \log n)$ expected time bound for the original `Quicksort` algorithm; we will not go into the details of this here.

13.6 Hashing: A Randomized Implementation of Dictionaries

Randomization has also proved to be a powerful technique in the design of data structures. Here we discuss perhaps the most fundamental use of randomization in this setting, a technique called *hashing* that can be used to maintain a dynamically changing set of elements. In the next section, we will show how an application of this technique yields a very simple algorithm for a problem that we saw in Chapter 5—the problem of finding the closest pair of points in the plane.

The Problem

One of the most basic applications of data structures is to simply maintain a set of elements that changes over time. For example, such applications could include a large company maintaining the set of its current employees and contractors, a news indexing service recording the first paragraphs of news articles it has seen coming across the newswire, or a search algorithm keeping track of the small part of an exponentially large search space that it has already explored.

In all these examples, there is a *universe* U of possible elements that is extremely large: the set of all possible people, all possible paragraphs (say, up to some character length limit), or all possible solutions to a computationally hard problem. The data structure is trying to keep track of a set $S \subseteq U$ whose size is generally a negligible fraction of U, and the goal is to be able to insert and delete elements from S and quickly determine whether a given element belongs to S.

We will call a data structure that accomplishes this a *dictionary*. More precisely, a dictionary is a data structure that supports the following operations.

- `MakeDictionary`. This operation initializes a fresh dictionary that can maintain a subset S of U; the dictionary starts out empty.
- `Insert`(u) adds element $u \in U$ to the set S. In many applications, there may be some additional information that we want to associate with u

(for example, u may be the name or ID number of an employee, and we want to also store some personal information about this employee), and we will simply imagine this being stored in the dictionary as part of a record together with u. (So, in general, when we talk about the element u, we really mean u and any additional information stored with u.)

- `Delete`(u) removes element u from the set S, if it is currently present.
- `Lookup`(u) determines whether u currently belongs to S; if it does, it also retrieves any additional information stored with u.

Many of the implementations we've discussed earlier in the book involve (most of) these operations: For example, in the implementation of the BFS and DFS graph traversal algorithms, we needed to maintain the set S of nodes already visited. But there is a fundamental difference between those problems and the present setting, and that is the size of U. The universe U in BFS or DFS is the set of nodes V, which is already given explicitly as part of the input. Thus it is completely feasible in those cases to maintain a set $S \subseteq U$ as we did there: defining an array with $|U|$ positions, one for each possible element, and setting the array position for u equal to 1 if $u \in S$, and equal to 0 if $u \notin S$. This allows for insertion, deletion, and lookup of elements in constant time per operation, by simply accessing the desired array entry.

Here, by contrast, we are considering the setting in which the universe U is enormous. So we are not going to be able to use an array whose size is anywhere near that of U. The fundamental question is whether, in this case, we can still implement a dictionary to support the basic operations almost as quickly as when U was relatively small.

We now describe a randomized technique called *hashing* that addresses this question. While we will not be able to do quite as well as the case in which it is feasible to define an array over all of U, hashing will allow us to come quite close.

Designing the Data Structure

As a motivating example, let's think a bit more about the problem faced by an automated service that processes breaking news. Suppose you're receiving a steady stream of short articles from various wire services, weblog postings, and so forth, and you're storing the lead paragraph of each article (truncated to at most 1,000 characters). Because you're using many sources for the sake of full coverage, there's a lot of redundancy: the same article can show up many times.

When a new article shows up, you'd like to quickly check whether you've seen the lead paragraph before. So a dictionary is exactly what you want for this problem: The universe U is the set of all strings of length at most 1,000 (or of

length exactly 1,000, if we pad them out with blanks), and we're maintaining a set $S \subseteq U$ consisting of strings (i.e., lead paragraphs) that we've seen before.

One solution would be to keep a linked list of all paragraphs, and scan this list each time a new one arrives. But a `Lookup` operation in this case takes time proportional to $|S|$. How can we get back to something that looks like an array-based solution?

Hash Functions The basic idea of hashing is to work with an array of size $|S|$, rather than one comparable to the (astronomical) size of U.

Suppose we want to be able to store a set S of size up to n. We will set up an array H of size n to store the information, and use a function $h : U \to \{0, 1, \ldots, n-1\}$ that maps elements of U to array positions. We call such a function h a *hash function*, and the array H a *hash table*. Now, if we want to add an element u to the set S, we simply place u in position $h(u)$ of the array H. In the case of storing paragraphs of text, we can think of $h(\cdot)$ as computing some kind of numerical signature or "check-sum" of the paragraph u, and this tells us the array position at which to store u.

This would work extremely well if, for all distinct u and v in our set S, it happened to be the case that $h(u) \neq h(v)$. In such a case, we could look up u in constant time: when we check array position $H[h(u)]$, it would either be empty or would contain just u.

In general, though, we cannot expect to be this lucky: there can be distinct elements $u, v \in S$ for which $h(u) = h(v)$. We will say that these two elements *collide*, since they are mapped to the same place in H. There are a number of ways to deal with collisions. Here we will assume that each position $H[i]$ of the hash table stores a linked list of all elements $u \in S$ with $h(u) = i$. The operation `Lookup`(u) would now work as follows.

- Compute the hash function $h(u)$.
- Scan the linked list at position $H[h(u)]$ to see if u is present in this list.

Hence the time required for `Lookup`(u) is proportional to the time to compute $h(u)$, plus the length of the linked list at $H[h(u)]$. And this latter quantity, in turn, is just the number of elements in S that collide with u. The `Insert` and `Delete` operations work similarly: `Insert` adds u to the linked list at position $H[h(u)]$, and `Delete` scans this list and removes u if it is present.

So now the goal is clear: We'd like to find a hash function that "spreads out" the elements being added, so that no one entry of the hash table H contains too many elements. This is not a problem for which worst-case analysis is very informative. Indeed, suppose that $|U| \geq n^2$ (we're imagining applications where it's much larger than this). Then, for any hash function h that we choose, there will be some set S of n elements that all map to the same

position. In the worst case, we will insert all the elements of this set, and then our Lookup operations will consist of scanning a linked list of length n.

Our main goal here is to show that randomization can help significantly for this problem. As usual, we won't make any assumptions about the set of elements S being random; we will simply exploit randomization in the design of the hash function. In doing this, we won't be able to completely avoid collisions, but can make them relatively rare enough, and so the lists will be quite short.

Choosing a Good Hash Function We've seen that the efficiency of the dictionary is based on the choice of the hash function h. Typically, we will think of U as a large set of numbers, and then use an easily computable function h that maps each number $u \in U$ to some value in the smaller range of integers $\{0, 1, \ldots, n-1\}$. There are many simple ways to do this: we could use the first or last few digits of u, or simply take u modulo n. While these simple choices may work well in many situations, it is also possible to get large numbers of collisions. Indeed, a fixed choice of hash function may run into problems because of the types of elements u encountered in the application: Maybe the particular digits we use to define the hash function encode some property of u, and hence maybe only a few options are possible. Taking u modulo n can have the same problem, especially if n is a power of 2. To take a concrete example, suppose we used a hash function that took an English paragraph, used a standard character encoding scheme like ASCII to map it to a sequence of bits, and then kept only the first few bits in this sequence. We'd expect a huge number of collisions at the array entries corresponding to the bit strings that encoded common English words like *The*, while vast portions of the array can be occupied only by paragraphs that begin with strings like *qxf*, and hence will be empty.

A slightly better choice in practice is to take (u mod p) for a prime number p that is approximately equal to n. While in some applications this may yield a good hashing function, it may not work well in all applications, and some primes may work much better than others (for example, primes very close to powers of 2 may not work so well).

Since hashing has been widely used in practice for a long time, there is a lot of experience with what makes for a good hash function, and many hash functions have been proposed that tend to work well empirically. Here we would like to develop a hashing scheme where we can prove that it results in efficient dictionary operations with high probability.

The basic idea, as suggested earlier, is to use randomization in the construction of h. First let's consider an extreme version of this: for every element $u \in U$, when we go to insert u into S, we select a value $h(u)$ uniformly at

random in the set $\{0, 1, \ldots, n-1\}$, independently of all previous choices. In this case, the probability that two randomly selected values $h(u)$ and $h(v)$ are equal (and hence cause a collision) is quite small.

(13.22) *With this uniform random hashing scheme, the probability that two randomly selected values $h(u)$ and $h(v)$ collide—that is, that $h(u) = h(v)$—is exactly $1/n$.*

Proof. Of the n^2 possible choices for the pair of values $(h(u), h(v))$, all are equally likely, and exactly n of these choices results in a collision. ■

However, it will not work to use a hash function with independently random chosen values. To see why, suppose we inserted u into S, and then later want to perform either `Delete`(u) or `Lookup`(u). We immediately run into the "Where did I put it?" problem: We will need to know the random value $h(u)$ that we used, so we will need to have stored the value $h(u)$ in some form where we can quickly look it up. But this is exactly the same problem we were trying to solve in the first place.

There are two things that we can learn from (13.22). First, it provides a concrete basis for the intuition from practice that hash functions that spread things around in a "random" way can be effective at reducing collisions. Second, and more crucial for our goals here, we will be able to show how a more controlled use of randomization achieves performance as good as suggested in (13.22), but in a way that leads to an efficient dictionary implementation.

Universal Classes of Hash Functions The key idea is to choose a hash function at random not from the collection of all possible functions into $[0, n-1]$, but from a carefully selected class of functions. Each function h in our class of functions $\mathcal{H}$ will map the universe U into the set $\{0, 1, \ldots, n-1\}$, and we will design it so that it has two properties. First, we'd like it to come with the guarantee from (13.22):

- For any pair of elements $u, v \in U$, the probability that a randomly chosen $h \in \mathcal{H}$ satisfies $h(u) = h(v)$ is at most $1/n$.

We say that a class $\mathcal{H}$ of functions is *universal* if it satisfies this first property. Thus (13.22) can be viewed as saying that the class of all possible functions from U into $\{0, 1, \ldots, n-1\}$ is universal.

However, we also need $\mathcal{H}$ to satisfy a second property. We will state this slightly informally for now and make it more precise later.

- Each $h \in \mathcal{H}$ can be compactly represented and, for a given $h \in \mathcal{H}$ and $u \in U$, we can compute the value $h(u)$ efficiently.

The class of all possible functions failed to have this property: Essentially, the only way to represent an arbitrary function from U into $\{0, 1, \ldots, n-1\}$ is to write down the value it takes on every single element of U.

In the remainder of this section, we will show the surprising fact that there exist classes $\mathcal{H}$ that satisfy both of these properties. Before we do this, we first make precise the basic property we need from a universal class of hash functions. We argue that if a function h is selected at random from a universal class of hash functions, then in any set $S \subset U$ of size at most n, and any $u \in U$, the expected number of items in S that collide with u is a constant.

(13.23) *Let $\mathcal{H}$ be a universal class of hash functions mapping a universe U to the set $\{0, 1, \ldots, n-1\}$, let S be an arbitrary subset of U of size at most n, and let u be any element in U. We define X to be a random variable equal to the number of elements $s \in S$ for which $h(s) = h(u)$, for a random choice of hash function $h \in \mathcal{H}$. (Here S and u are fixed, and the randomness is in the choice of $h \in \mathcal{H}$.) Then $E[X] \leq 1$.*

Proof. For an element $s \in S$, we define a random variable X_s that is equal to 1 if $h(s) = h(u)$, and equal to 0 otherwise. We have $E\left[X_s\right] = \Pr\left[X_s = 1\right] \leq 1/n$, since the class of functions is universal.

Now $X = \sum_{s \in S} X_s$, and so, by linearity of expectation, we have

$$E[X] = \sum_{s \in S} E\left[X_s\right] \leq |S| \cdot \frac{1}{n} \leq 1. \quad \blacksquare$$

Designing a Universal Class of Hash Functions Next we will design a universal class of hash functions. We will use a prime number $p \approx n$ as the size of the hash table H. To be able to use integer arithmetic in designing our hash functions, we will identify the universe with vectors of the form $x = (x_1, x_2, \ldots x_r)$ for some integer r, where $0 \leq x_i < p$ for each i. For example, we can first identify U with integers in the range $[0, N-1]$ for some N, and then use consecutive blocks of $\lfloor \log p \rfloor$ bits of u to define the corresponding coordinates x_i. If $U \subseteq [0, N-1]$, then we will need a number of coordinates $r \approx \log N / \log n$.

Let $\mathcal{A}$ be the set of all vectors of the form $a = (a_1, \ldots, a_r)$, where a_i is an integer in the range $[0, p-1]$ for each $i = 1, \ldots, r$. For each $a \in \mathcal{A}$, we define the linear function

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p.$$

This now completes our random implementation of dictionaries. We define the family of hash functions to be $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$. To execute `MakeDictionary`, we choose a random hash function from $\mathcal{H}$; in other words, we choose a random vector from $\mathcal{A}$ (by choosing each coordinate uniformly at random), and form the function h_a. Note that in order to define $\mathcal{A}$, we need to find a prime number $p \geq n$. There are methods for generating prime numbers quickly, which we will not go into here. (In practice, this can also be accomplished using a table of known prime numbers, even for relatively large n.)

We then use this as the hash function with which to implement `Insert`, `Delete`, and `Lookup`. The family $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$ satisfies a formal version of the second property we were seeking: It has a compact representation, since by simply choosing and remembering a random $a \in \mathcal{A}$, we can compute $h_a(u)$ for all elements $u \in U$. Thus, to show that $\mathcal{H}$ leads to an efficient, hashing-based implementation of dictionaries, we just need to establish that $\mathcal{H}$ is a universal family of hash functions.

Analyzing the Data Structure

If we are using a hash function h_a from the class $\mathcal{H}$ that we've defined, then a collision $h_a(x) = h_a(y)$ defines a linear equation modulo the prime number p. In order to analyze such equations, it's useful to have the following "cancellation law."

(13.24) *For any prime p and any integer $z \neq 0 \bmod p$, and any two integers α, β, if $\alpha z = \beta z \bmod p$, then $\alpha = \beta \bmod p$.*

Proof. Suppose $\alpha z = \beta z \bmod p$. Then, by rearranging terms, we get $z(\alpha - \beta) = 0 \bmod p$, and hence $z(\alpha - \beta)$ is divisible by p. But $z \neq 0 \bmod p$, so z is not divisible by p. Since p is prime, it follows that $\alpha - \beta$ must be divisible by p; that is, $\alpha = \beta \bmod p$ as claimed. ■

We now use this to prove the main result in our analysis.

(13.25) *The class of linear functions $\mathcal{H}$ defined above is universal.*

Proof. Let $x = (x_1, x_2, \ldots x_r)$ and $y = (y_1, y_2, \ldots y_r)$ be two distinct elements of U. We need to show that the probability of $h_a(x) = h_a(y)$, for a randomly chosen $a \in A$, is at most $1/p$.

Since $x \neq y$, then there must be an index j such that $x_j \neq y_j$. We now consider the following way of choosing the random vector $a \in \mathcal{A}$. We first choose all the coordinates a_i where $i \neq j$. Then, finally, we choose coordinate a_j. We will show that regardless of how all the other coordinates a_i were

chosen, the probability of $h_a(x) = h_a(y)$, taken over the final choice of a_j, is exactly $1/p$. It will follow that the probability of $h_a(x) = h_a(y)$ over the random choice of the full vector a must be $1/p$ as well.

This conclusion is intuitively clear: If the probability is $1/p$ regardless of how we choose all other a_i, then it is $1/p$ overall. There is also a direct proof of this using conditional probabilities. Let $\mathcal{E}$ be the event that $h_a(x) = h_a(y)$, and let $\mathcal{F}_b$ be the event that all coordinates a_i (for $i \neq j$) receive a sequence of values b. We will show, below, that $\Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] = 1/p$ for all b. It then follows that $\Pr[\mathcal{E}] = \sum_b \Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] \cdot \Pr\left[\mathcal{F}_b\right] = (1/p) \sum_b \Pr\left[\mathcal{F}_b\right] = 1/p$.

So, to conclude the proof, we assume that values have been chosen arbitrarily for all other coordinates a_i, and we consider the probability of selecting a_j so that $h_a(x) = h_a(y)$. By rearranging terms, we see that $h_a(x) = h_a(y)$ if and only if

$$a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \bmod p.$$

Since the choices for all a_i ($i \neq j$) have been fixed, we can view the right-hand side as some fixed quantity m. Also, let us define $z = y_j - x_j$.

Now it is enough to show that there is exactly one value $0 \leq a_j < p$ that satisfies $a_j z = m \bmod p$; indeed, if this is the case, then there is a probability of exactly $1/p$ of choosing this value for a_j. So suppose there were two such values, a_j and a'_j. Then we would have $a_j z = a'_j z \bmod p$, and so by (13.24) we would have $a_j = a'_j \bmod p$. But we assumed that $a_j, a'_j < p$, and so in fact a_j and a'_j would be the same. It follows that there is only one a_j in this range that satisfies $a_j z = m \bmod p$.

Tracing back through the implications, this means that the probability of choosing a_j so that $h_a(x) = h_a(y)$ is $1/p$, however we set the other coordinates a_i in a; thus the probability that x and y collide is $1/p$. Thus we have shown that $\mathcal{H}$ is a universal class of hash functions. ■

13.7 Finding the Closest Pair of Points: A Randomized Approach

In Chapter 5, we used the divide-and-conquer technique to develop an $O(n \log n)$ time algorithm for the problem of finding the closest pair of points in the plane. Here we will show how to use randomization to develop a different algorithm for this problem, using an underlying dictionary data structure. We will show that this algorithm runs in $O(n)$ expected time, plus $O(n)$ expected dictionary operations.

There are several related reasons why it is useful to express the running time of our algorithm in this way, accounting for the dictionary operations

separately. We have seen in Section 13.6 that dictionaries have a very efficient implementation using hashing, so abstracting out the dictionary operations allows us to treat the hashing as a "black box" and have the algorithm inherit an overall running time from whatever performance guarantee is satisfied by this hashing procedure. A concrete payoff of this is the following. It has been shown that with the right choice of hashing procedure (more powerful, and more complicated, than what we described in Section 13.6), one can make the underlying dictionary operations run in linear expected time as well, yielding an overall expected running time of $O(n)$. Thus the randomized approach we describe here leads to an improvement over the running time of the divide-and-conquer algorithm that we saw earlier. We will talk about the ideas that lead to this $O(n)$ bound at the end of the section.

It is worth remarking at the outset that randomization shows up for two independent reasons in this algorithm: the way in which the algorithm processes the input points will have a random component, regardless of how the dictionary data structure is implemented; and when the dictionary is implemented using hashing, this introduces an additional source of randomness as part of the hash-table operations. Expressing the running time via the number of dictionary operations allows us to cleanly separate the two uses of randomness.

The Problem

Let us start by recalling the problem's (very simple) statement. We are given n points in the plane, and we wish to find the pair that is closest together. As discussed in Chapter 5, this is one of the most basic geometric *proximity* problems, a topic with a wide range of applications.

We will use the same notation as in our earlier discussion of the closest-pair problem. We will denote the set of points by $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates (x_i, y_i); and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find the pair of points p_i, p_j that minimizes $d(p_i, p_j)$.

To simplify the discussion, we will assume that the points are all in the unit square: $0 \leq x_i, y_i < 1$ for all $i = 1, \ldots, n$. This is no loss of generality: in linear time, we can rescale all the x- and y-coordinates of the points so that they lie in a unit square, and then we can translate them so that this unit square has its lower left corner at the origin.

Designing the Algorithm

The basic idea of the algorithm is very simple. We'll consider the points in random order, and maintain a current value δ for the closest pair as we process

the points in this order. When we get to a new point p, we look "in the vicinity" of p to see if any of the previously considered points are at a distance less than δ from p. If not, then the closest pair hasn't changed, and we move on to the next point in the random order. If there is a point within a distance less than δ from p, then the closest pair has changed, and we will need to update it.

The challenge in turning this into an efficient algorithm is to figure out how to implement the task of looking for points in the vicinity of p. It is here that the dictionary data structure will come into play.

We now begin making this more concrete. Let us assume for simplicity that the points in our random order are labeled $p_1, \ldots, p_n$. The algorithm proceeds in stages; during each stage, the closest pair remains constant. The first stage starts by setting $\delta = d(p_1, p_2)$, the distance of the first two points. The goal of a stage is to either verify that δ is indeed the distance of the closest pair of points, or to find a pair of points p_i, p_j with $d(p_i, p_j) < \delta$. During a stage, we'll gradually add points in the order $p_1, p_2, \ldots, p_n$. The stage terminates when we reach a point p_i so that for some $j < i$, we have $d(p_i, p_j) < \delta$. We then let δ for the next stage be the closest distance found so far: $\delta = \min_{j:j<i} d(p_i, p_j)$.

The number of stages used will depend on the random order. If we get lucky, and p_1, p_2 are the closest pair of points, then a single stage will do. It is also possible to have as many as $n - 2$ stages, if adding a new point always decreases the minimum distance. We'll show that the expected running time of the algorithm is within a constant factor of the time needed in the first, lucky case, when the original value of δ is the smallest distance.

Testing a Proposed Distance The main subroutine of the algorithm is a method to test whether the current pair of points with distance δ remains the closest pair when a new point is added and, if not, to find the new closest pair.

The idea of the verification is to subdivide the unit square (the area where the points lie) into subsquares whose sides have length $\delta/2$, as shown in Figure 13.2. Formally, there will be N^2 subsquares, where $N = \lceil 1/(2\delta) \rceil$: for $0 \le s \le N - 1$ and $1 \le t \le N - 1$, we define the subsquare S_{st} as

$$S_{st} = \{(x, y) : s\delta/2 \le x < (s + 1)\delta/2; t\delta/2 \le y < (t + 1)\delta/2\}.$$

We claim that this collection of subsquares has two nice properties for our purposes. First, any two points that lie in the same subsquare have distance less than δ. Second, and a partial converse to this, any two points that are less than δ away from each other must fall in either the same subsquare or in very close subsquares.

(13.26) *If two points p and q belong to the same subsquare S_{st}, then $d(p, q) < \delta$.*

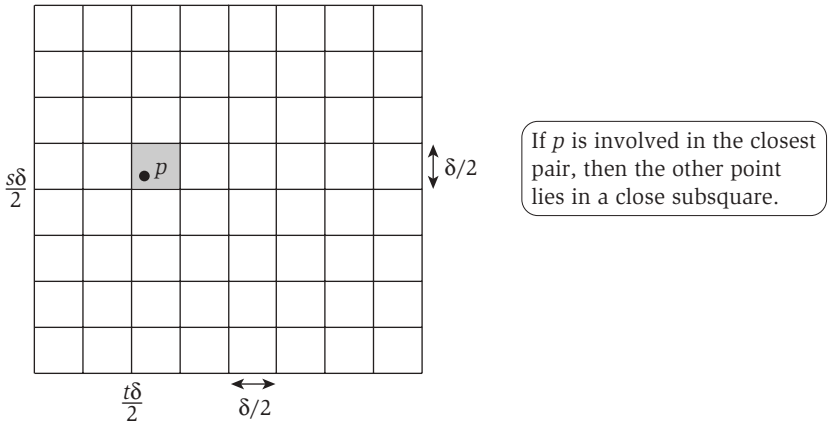

Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{st}.

Proof. If points p and q are in the same subsquare, then both coordinates of the two points differ by at most $\delta/2$, and hence $d(p,q) \leq \sqrt{(\delta/2)^2 + (\delta/2)^2)} = \delta/\sqrt{2} < \delta$, as required. ■

Next we say that subsquares S_{st} and $S_{s't'}$ are *close* if $|s - s'| \leq 2$ and $|t - t'| \leq 2$. (Note that a subsquare is close to itself.)

(13.27) *If for two points $p, q \in P$ we have $d(p,q) < \delta$, then the subsquares containing them are close.*

Proof. Consider two points $p, q \in P$ belonging to subsquares that are not close; assume $p \in S_{st}$ and $q \in S_{s't'}$, where one of s, s' or t, t' differs by more than 2. It follows that in one of their respective x- or y-coordinates, p and q differ by at least δ, and so we cannot have $d(p,q) < \delta$. ■

Note that for any subsquare S_{st}, the set of subsquares close to it form a 5×5 grid around it. Thus we conclude that there are at most 25 subsquares close to S_{st}, counting S_{st} itself. (There will be fewer than 25 if S_{st} is at the edge of the unit square containing the input points.)

Statements (13.26) and (13.27) suggest the basic outline of our algorithm. Suppose that, at some point in the algorithm, we have proceeded partway through the random order of the points and seen $P' \subseteq P$, and suppose that we know the minimum distance among points in P' to be δ. For each of the points in P', we keep track of the subsquare containing it.

Now, when the next point p is considered, we determine which of the subsquares S_{st} it belongs to. If p is going to cause the minimum distance to change, there must be some earlier point $p' \in P'$ at distance less than δ from it; and hence, by (13.27), the point p' must be in one of the 25 squares around the square S_{st} containing p. So we will simply check each of these 25 squares one by one to see if it contains a point in P'; for each point in P' that we find this way, we compute its distance to p. By (13.26), each of these subsquares contains at most one point of P', so this is at most a constant number of distance computations. (Note that we used a similar idea, via (5.10), at a crucial point in the divide-and-conquer algorithm for this problem in Chapter 5.)

A Data Structure for Maintaining the Subsquares The high-level description of the algorithm relies on being able to name a subsquare S_{st} and quickly determine which points of P, if any, are contained in it. A dictionary is a natural data structure for implementing such operations. The *universe* U of possible elements is the set of all subsquares, and the set S maintained by the data structure will be the subsquares that contain points from among the set P' that we've seen so far. Specifically, for each point $p' \in P'$ that we have seen so far, we keep the subsquare containing it in the dictionary, tagged with the index of p'. We note that $N^2 = \lceil 1/(2\delta) \rceil^2$ will, in general, be much larger than n, the number of points. Thus we are in the type of situation considered in Section 13.6 on hashing, where the universe of possible elements (the set of all subsquares) is much larger than the number of elements being indexed (the subsquares containing an input point seen thus far).

Now, when we consider the next point p in the random order, we determine the subsquare S_{st} containing it and perform a `Lookup` operation for each of the 25 subsquares close to S_{st}. For any points discovered by these `Lookup` operations, we compute the distance to p. If none of these distances are less than δ, then the closest distance hasn't changed; we insert S_{st} (tagged with p) into the dictionary and proceed to the next point.

However, if we find a point p' such that $\delta' = d(p, p') < \delta$, then we need to update our closest pair. This updating is a rather dramatic activity: Since the value of the closest pair has dropped from δ to δ', our entire collection of subsquares, and the dictionary supporting it, has become useless—it was, after all, designed only to be useful if the minimum distance was δ. We therefore invoke `MakeDictionary` to create a new, empty dictionary that will hold subsquares whose side lengths are $\delta'/2$. For each point seen thus far, we determine the subsquare containing it (in this new collection of subsquares), and we insert this subsquare into the dictionary. Having done all this, we are again ready to handle the next point in the random order.

Summary of the Algorithm We have now actually described the algorithm in full. To recap:

```
Order the points in a random sequence p_1, p_2, ..., p_n
Let δ denote the minimum distance found so far
Initialize δ = d(p_1, p_2)
Invoke MakeDictionary for storing subsquares of side length δ/2
For i = 1, 2, ..., n:
  Determine the subsquare S_st containing p_i
  Look up the 25 subsquares close to p_i
  Compute the distance from p_i to any points found in these subsquares
  If there is a point p_j (j < i) such that δ' = d(p_j, p_i) < δ then
    Delete the current dictionary
    Invoke MakeDictionary for storing subsquares of side length δ'/2
    For each of the points p_1, p_2, ..., p_i:
      Determine the subsquare of side length δ'/2 that contains it
      Insert this subsquare into the new dictionary
    Endfor
  Else
    Insert p_i into the current dictionary
  Endif
Endfor
```

Analyzing the Algorithm

There are already some things we can say about the overall running time of the algorithm. To consider a new point p_i, we need to perform only a constant number of `Lookup` operations and a constant number of distance computations. Moreover, even if we had to update the closest pair in every iteration, we'd only do n `MakeDictionary` operations.

The missing ingredient is the total expected cost, over the course of the algorithm's execution, due to reinsertions into new dictionaries when the closest pair is updated. We will consider this next. For now, we can at least summarize the current state of our knowledge as follows.

(13.28) *The algorithm correctly maintains the closest pair at all times, and it performs at most $O(n)$ distance computations, $O(n)$* `Lookup` *operations, and $O(n)$* `MakeDictionary` *operations.*

We now conclude the analysis by bounding the expected number of `Insert` operations. Trying to find a good bound on the total expected number of `Insert` operations seems a bit problematic at first: An update to the closest

pair in iteration i will result in i insertions, and so each update comes at a high cost once i gets large. Despite this, we will show the surprising fact that the expected number of insertions is only $O(n)$. The intuition here is that, even as the cost of updates becomes steeper as the iterations proceed, these updates become correspondingly less likely.

Let X be a random variable specifying the number of `Insert` operations performed; the value of this random variable is determined by the random order chosen at the outset. We are interested in bounding $E[X]$, and as usual in this type of situation, it is helpful to break X down into a sum of simpler random variables. Thus let X_i be a random variable equal to 1 if the i^{th} point in the random order causes the minimum distance to change, and equal to 0 otherwise.

Using these random variables X_i, we can write a simple formula for the total number of `Insert` operations. Each point is inserted once when it is first encountered; and i points need to be reinserted if the minimum distance changes in iteration i. Thus we have the following claim.

(13.29) *The total number of* `Insert` *operations performed by the algorithm is* $n + \sum_i iX_i$.

Now we bound the probability $\Pr[X_i = 1]$ that considering the i^{th} point causes the minimum distance to change.

(13.30) $\Pr[X_i = 1] \leq 2/i$.

Proof. Consider the first i points $p_1, p_2, \ldots, p_i$ in the random order. Assume that the minimum distance among these points is achieved by p and q. Now the point p_i can only cause the minimum distance to decrease if $p_i = p$ or $p_i = q$. Since the first i points are in a random order, any of them is equally likely to be last, so the probability that p or q is last is $2/i$. ■

Note that $2/i$ is only an upper bound in (13.30) because there could be multiple pairs among the first i points that define the same smallest distance.

By (13.29) and (13.30), we can bound the total number of `Insert` operations as

$$E[X] = n + \sum_i i \cdot E[X_i] \leq n + 2n = 3n.$$

Combining this with (13.28), we obtain the following bound on the running time of the algorithm.

(13.31) *In expectation, the randomized closest-pair algorithm requires* $O(n)$ *time plus* $O(n)$ *dictionary operations.*

Achieving Linear Expected Running Time

Up to this point, we have treated the dictionary data structure as a black box, and in (13.31) we bounded the running time of the algorithm in terms of computational time plus dictionary operations. We now want to give a bound on the actual expected running time, and so we need to analyze the work involved in performing these dictionary operations.

To implement the dictionary, we'll use a universal hashing scheme, like the one discussed in Section 13.6. Once the algorithm employs a hashing scheme, it is making use of randomness in two distinct ways: First, we randomly order the points to be added; and second, for each new minimum distance δ, we apply randomization to set up a new hash table using a universal hashing scheme.

When inserting a new point p_i, the algorithm uses the hash-table `Lookup` operation to find all nodes in the 25 subsquares close to p_i. However, if the hash table has collisions, then these 25 `Lookup` operations can involve inspecting many more than 25 nodes. Statement (13.23) from Section 13.6 shows that each such `Lookup` operation involves considering $O(1)$ previously inserted points, in expectation. It seems intuitively clear that performing $O(n)$ hash-table operations in expectation, each of which involves considering $O(1)$ elements in expectation, will result in an expected running time of $O(n)$ overall. To make this intuition precise, we need to be careful with how these two sources of randomness interact.

(13.32) *Assume we implement the randomized closest-pair algorithm using a universal hashing scheme. In expectation, the total number of points considered during the* `Lookup` *operations is bounded by* $O(n)$.

Proof. From (13.31) we know that the expected number of `Lookup` operations is $O(n)$, and from (13.23) we know that each of these `Lookup` operations involves considering only $O(1)$ points in expectation. In order to conclude that this implies the expected number of points considered is $O(n)$, we now consider the relationship between these two sources of randomness.

Let X be a random variable denoting the number of `Lookup` operations performed by the algorithm. Now the random order σ that the algorithm chooses for the points completely determines the sequence of minimum-distance values the algorithm will consider and the sequence of dictionary operations it will perform. As a result, the choice of σ determines the value of X; we let $X(\sigma)$ denote this value, and we let $\mathcal{E}_\sigma$ denote the event the algorithm chooses the random order σ. Note that the conditional expectation $E\left[X \mid \mathcal{E}_\sigma\right]$ is equal to $X(\sigma)$. Also, by (13.31), we know that $E[X] \leq c_0 n$, for some constant c_0.

Now consider this sequence of `Lookup` operations for a fixed order σ. For $i = 1, \ldots, X(\sigma)$, let Y_i be the number of points that need to be inspected during the i^{th} `Lookup` operations—namely, the number of previously inserted points that collide with the dictionary entry involved in this `Lookup` operation. We would like to bound the expected value of $\sum_{i=1}^{X(\sigma)} Y_i$, where expectation is over both the random choice of σ and the random choice of hash function.

By (13.23), we know that $E\left[Y_i \mid \mathcal{E}_\sigma\right] = O(1)$ for all σ and all values of i. It is useful to be able to refer to the constant in the expression $O(1)$ here, so we will say that $E\left[Y_i \mid \mathcal{E}_\sigma\right] \leq c_1$ for all σ and all values of i. Summing over all i, and using linearity of expectation, we get $E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \leq c_1 X(\sigma)$. Now we have

$$\begin{aligned} E\left[\sum_{i=1}^{X(\sigma)} Y_i\right] &= \sum_\sigma \Pr\left[\mathcal{E}_\sigma\right] E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \\ &\leq \sum_\sigma \Pr\left[\mathcal{E}_\sigma\right] \cdot c_1 X(\sigma) \\ &= c_1 \sum_\sigma E\left[X \mid \mathcal{E}_\sigma\right] \cdot \Pr\left[\mathcal{E}_\sigma\right] = c_1 E\left[X\right]. \end{aligned}$$

Since we know that $E[X]$ is at most $c_0 n$, the total expected number of points considered is at most $c_0 c_1 n = O(n)$, which proves the claim. ∎

Armed with this claim, we can use the universal hash functions from Section 13.6 in our closest-pair algorithm. In expectation, the algorithm will consider $O(n)$ points during the `Lookup` operations. We have to set up multiple hash tables—a new one each time the minimum distance changes—and we have to compute $O(n)$ hash-function values. All hash tables are set up for the same size, a prime $p \geq n$. We can select one prime and use the same table throughout the algorithm. Using this, we get the following bound on the running time.

(13.33) *In expectation, the algorithm uses $O(n)$ hash-function computations and $O(n)$ additional time for finding the closest pair of points.*

Note the distinction between this statement and (13.31). There we counted each dictionary operation as a single, atomic step; here, on the other hand, we've conceptually opened up the dictionary operations so as to account for the time incurred due to hash-table collisions and hash-function computations.

Finally, consider the time needed for the $O(n)$ hash-function computations. How fast is it to compute the value of a universal hash function h? The class of universal hash functions developed in Section 13.6 breaks numbers in our universe U into $r \approx \log N / \log n$ smaller numbers of size $O(\log n)$ each, and

then uses $O(r)$ arithmetic operations on these smaller numbers to compute the hash-function value. So computing the hash value of a single point involves $O(\log N / \log n)$ multiplications, on numbers of size $\log n$. This is a total of $O(n \log N / \log n)$ arithmetic operations over the course of the algorithm, more than the $O(n)$ we were hoping for.

In fact, it is possible to decrease the number of arithmetic operations to $O(n)$ by using a more sophisticated class of hash functions. There are other classes of universal hash functions where computing the hash-function value can be done by only $O(1)$ arithmetic operations (though these operations will have to be done on larger numbers, integers of size roughly $\log N$). This class of improved hash functions also comes with one extra difficulty for this application: the hashing scheme needs a prime that is bigger than the size of the universe (rather than just the size of the set of points). Now the universe in this application grows inversely with the minimum distance δ, and so, in particular, it increases every time we discover a new, smaller minimum distance. At such points, we will have to find a new prime and set up a new hash table. Although we will not go into the details of this here, it is possible to deal with these difficulties and make the algorithm achieve an expected running time of $O(n)$.

13.8 Randomized Caching

We now discuss the use of randomization for the caching problem, which we first encountered in Chapter 4. We begin by developing a class of algorithms, the *marking algorithms*, that include both deterministic and randomized approaches. After deriving a general performance guarantee that applies to all marking algorithms, we show how a stronger guarantee can be obtained for a particular marking algorithm that exploits randomization.

The Problem

We begin by recalling the *Cache Maintenance Problem* from Chapter 4. In the most basic setup, we consider a processor whose full memory has n addresses; it is also equipped with a *cache* containing k slots of memory that can be accessed very quickly. We can keep copies of k items from the full memory in the cache slots, and when a memory location is accessed, the processor will first check the cache to see if it can be quickly retrieved. We say the request is a *cache hit* if the cache contains the requested item; in this case, the access is very quick. We say the request is a *cache miss* if the requested item is not in the cache; in this case, the access takes much longer, and moreover, one of the items currently in the cache must be *evicted* to make room for the new item. (We will assume that the cache is kept full at all times.)

The goal of a Cache Maintenance Algorithm is to minimize the number of cache misses, which are the truly expensive part of the process. The sequence of memory references is not under the control of the algorithm—this is simply dictated by the application that is running—and so the job of the algorithms we consider is simply to decide on an *eviction policy*: Which item currently in the cache should be evicted on each cache miss?

In Chapter 4, we saw a greedy algorithm that is optimal for the problem: Always evict the item that will be needed the *farthest in the future*. While this algorithm is useful to have as an absolute benchmark on caching performance, it clearly cannot be implemented under real operating conditions, since we don't know ahead of time when each item will be needed next. Rather, we need to think about eviction policies that operate *online*, using only information about past requests without knowledge of the future.

The eviction policy that is typically used in practice is to evict the item that was used the least recently (i.e., whose most recent access was the longest ago in the past); this is referred to as the Least-Recently-Used, or LRU, policy. The empirical justification for LRU is that algorithms tend to have a certain locality in accessing data, generally using the same set of data frequently for a while. If a data item has not been accessed for a long time, this is a sign that it may not be accessed again for a long time.

Here we will evaluate the performance of different eviction policies without making any assumptions (such as locality) on the sequence of requests. To do this, we will compare the number of misses made by an eviction policy on a sequence σ with the minimum number of misses it is possible to make on σ. We will use $f(\sigma)$ to denote this latter quantity; it is the number of misses achieved by the optimal Farthest-in-Future policy. Comparing eviction policies to the optimum is very much in the spirit of providing performance guarantees for approximation algorithms, as we did in Chapter 11. Note, however, the following interesting difference: the reason the optimum was not attainable in our approximation analyses from that chapter (assuming $\mathcal{P} \neq \mathcal{NP}$) is that the algorithms were constrained to run in polynomial time; here, on the other hand, the eviction policies are constrained in their pursuit of the optimum by the fact that they do not know the requests that are coming in the future.

For eviction policies operating under this online constraint, it initially seems hopeless to say something interesting about their performance: Why couldn't we just design a request sequence that completely confounds any online eviction policy? The surprising point here is that it is in fact possible to give absolute guarantees on the performance of various online policies relative to the optimum.

We first show that the number of misses incurred by LRU, on any request sequence, can be bounded by roughly k times the optimum. We then use randomization to develop a variation on LRU that has an exponentially stronger bound on its performance: Its number of misses is never more than $O(\log k)$ times the optimum.

Designing the Class of Marking Algorithms

The bounds for both LRU and its randomized variant will follow from a general template for designing online eviction policies—a class of policies called *marking algorithms*. They are motivated by the following intuition. To do well against the benchmark of $f(\sigma)$, we need an eviction policy that is sensitive to the difference between the following two possibilities: (a) in the recent past, the request sequence has contained more than k distinct items; or (b) in the recent past, the request sequence has come exclusively from a set of at most k items. In the first case, we know that $f(\sigma)$ must be increasing, since no algorithm can handle more than k distinct items without incurring a cache miss. But, in the second case, it's possible that σ is passing through a long stretch in which an optimal algorithm need not incur any misses at all. It is here that our policy must make sure that it incurs very few misses.

Guided by these considerations, we now describe the basic outline of a marking algorithm, which prefers evicting items that don't seem to have been used in a long time. Such an algorithm operates in *phases*; the description of one phase is as follows.

```
Each memory item can be either marked or unmarked
At the beginning of the phase, all items are unmarked
On a request to item s:
   Mark s
   If s is in the cache, then evict nothing
   Else s is not in the cache:
      If all items currently in the cache are marked then
         Declare the phase over
         Processing of s is deferred to start of next phase
      Else evict an unmarked item from the cache
      Endif
   Endif
```

Note that this describes a class of algorithms, rather than a single specific algorithm, because the key step—`evict an unmarked item from the`

`cache`—does not specify which unmarked item should be selected. We will see that eviction policies with different properties and performance guarantees arise depending on how we resolve this ambiguity.

We first observe that, since a phase starts with all items unmarked, and items become marked only when accessed, the unmarked items have all been accessed less recently than the marked items. This is the sense in which a marking algorithm is trying to evict items that have not been requested recently. Also, at any point in a phase, if there are any unmarked items in the cache, then the least recently used item must be unmarked. It follows that the LRU policy evicts an unmarked item whenever one is available, and so we have the following fact.

(13.34) *The LRU policy is a marking algorithm.*

Analyzing Marking Algorithms

We now describe a method for analyzing marking algorithms, ending with a bound on performance that applies to all marking algorithms. After this, when we add randomization, we will need to strengthen this analysis.

Consider an arbitrary marking algorithm operating on a request sequence σ. For the analysis, we picture an optimal caching algorithm operating on σ alongside this marking algorithm, incurring an overall cost of $f(\sigma)$. Suppose that there are r phases in this sequence σ, as defined by the marking algorithm.

To make the analysis easier to discuss, we are going to "pad" the sequence σ both at the beginning and the end with some extra requests; these will not add any extra misses to the optimal algorithm—that is, they will not cause $f(\sigma)$ to increase—and so any bound we show on the performance of the marking algorithm relative to the optimum for this padded sequence will also apply to σ. Specifically, we imagine a "phase 0" that takes place before the first phase, in which all the items initially in the cache are requested once. This does not affect the cost of either the marking algorithm or the optimal algorithm. We also imagine that the final phase r ends with an epilogue in which every item currently in the cache of the optimal algorithm is requested twice in round-robin fashion. This does not increase $f(\sigma)$; and by the end of the second pass through these items, the marking algorithm will contain each of them in its cache, and each will be marked.

For the performance bound, we need two things: an upper bound on the number of misses incurred by the marking algorithm, and a lower bound saying that the optimum must incur at least a certain number of misses.

The division of the request sequence σ into phases turns out to be the key to doing this. First of all, here is how we can picture the history of a

phase, from the marking algorithm's point of view. At the beginning of the phase, all items are unmarked. Any item that is accessed during the phase is marked, and it then remains in the cache for the remainder of the phase. Over the course of the phase, the number of marked items grows from 0 to k, and the next phase begins with a request to a $(k+1)^{\text{st}}$ item, different from all of these marked items. We summarize some conclusions from this picture in the following claim.

(13.35) *In each phase, σ contains accesses to exactly k distinct items. The subsequent phase begins with an access to a different $(k+1)^{\text{st}}$ item.*

Since an item, once marked, remains in the cache until the end of the phase, the marking algorithm cannot incur a miss for an item more than once in a phase. Combined with (13.35), this gives us an upper bound on the number of misses incurred by the marking algorithm.

(13.36) *The marking algorithm incurs at most k misses per phase, for a total of at most kr misses over all r phases.*

As a lower bound on the optimum, we have the following fact.

(13.37) *The optimum incurs at least $r-1$ misses. In other words, $f(\sigma) \geq r-1$.*

Proof. Consider any phase but the last one, and look at the situation just after the first access (to an item s) in this phase. Currently s is in the cache maintained by the optimal algorithm, and (13.35) tells us that the remainder of the phase will involve accesses to $k-1$ other distinct items, and the first access of the next phase will involve a k^{th} other item as well. Let S be this set of k items other than s. We note that at least one of the members of S is not currently in the cache maintained by the optimal algorithm (since, with s there, it only has room for $k-1$ other items), and the optimal algorithm will incur a miss the first time this item is accessed.

What we've shown, therefore, is that for every phase $j < r$, the sequence from the second access in phase j through the first access in phase $j+1$ involves at least one miss by the optimum. This makes for a total of at least $r-1$ misses. ∎

Combining (13.36) and (13.37), we have the following performance guarantee.

(13.38) *For any marking algorithm, the number of misses it incurs on any sequence σ is at most $k \cdot f(\sigma) + k$.*

Proof. The number of misses incurred by the marking algorithm is at most

$$kr = k(r-1) + k \leq k \cdot f(\sigma) + k,$$

where the final inequality is just (13.37). ■

Note that the "$+k$" in the bound of (13.38) is just an additive constant, independent of the length of the request sequence σ, and so the key aspect of the bound is the factor of k relative to the optimum. To see that this factor of k is the best bound possible for some marking algorithms, and for LRU in particular, consider the behavior of LRU on a request sequence in which $k+1$ items are repeatedly requested in a round-robin fashion. LRU will each time evict the item that will be needed just in the next step, and hence it will incur a cache miss on each access. (It's possible to get this kind of terrible caching performance in practice for precisely such a reason: the program is executing a loop that is just slightly too big for the cache.) On the other hand, the optimal policy, evicting the page that will be requested farthest in the future, incurs a miss only every k steps, so LRU incurs a factor of k more misses than the optimal policy.

Designing a Randomized Marking Algorithm

The bad example for LRU that we just saw implies that, if we want to obtain a better bound for an online caching algorithm, we will not be able to reason about fully general marking algorithms. Rather, we will define a simple *Randomized Marking Algorithm* and show that it never incurs more than $O(\log k)$ times the number of misses of the optimal algorithm—an exponentially better bound.

Randomization is a natural choice in trying to avoid the unfortunate sequence of "wrong" choices in the bad example for LRU. To get this bad sequence, we needed to define a sequence that always evicted precisely the wrong item. By randomizing, a policy can make sure that, "on average," it is throwing out an unmarked item that will at least not be needed right away.

Specifically, where the general description of a marking contained the line

```
Else evict an unmarked item from the cache
```

without specifying how this unmarked item is to be chosen, our Randomized Marking Algorithm uses the following rule:

```
Else evict an unmarked item chosen uniformly at random
     from the cache
```

This is arguably the simplest way to incorporate randomization into the marking framework.[1]

Analyzing the Randomized Marking Algorithm

Now we'd like to get a bound for the Randomized Marking Algorithm that is stronger than (13.38); but in order to do this, we need to extend the analysis in (13.36) and (13.37) to something more subtle. This is because there are sequences σ, with r phases, where the Randomized Marking Algorithm can really be made to incur kr misses—just consider a sequence that never repeats an item. But the point is that, on such sequences, the optimum will incur many more than $r-1$ misses. We need a way to bring the upper and lower bounds closer together, based on the structure of the sequence.

This picture of a "runaway sequence" that never repeats an item is an extreme instance of the distinction we'd like to draw: It is useful to classify the unmarked items in the middle of a phase into two further categories. We call an unmarked item *fresh* if it was not marked in the previous phase either, and we call it *stale* if it was marked in the previous phase.

Recall the picture of a single phase that led to (13.35): The phase begins with all items unmarked, and it contains accesses to k distinct items, each of which goes from unmarked to marked the first time it is accessed. Among these k accesses to unmarked items in phase j, let c_j denote the number of these that are to fresh items.

To strengthen the result from (13.37), which essentially said that the optimum incurs at least one miss per phase, we provide a bound in terms of the number of fresh items in a phase.

(13.39) $f(\sigma) \geq \frac{1}{2} \sum_{j=1}^{r} c_j$.

Proof. Let $f_j(\sigma)$ denote the number of misses incurred by the optimal algorithm in phase j, so that $f(\sigma) = \sum_{j=1}^{r} f_j(\sigma)$. From (13.35), we know that in any phase j, there are requests to k distinct items. Moreover, by our definition of *fresh*, there are requests to c_{j+1} further items in phase $j+1$; so between phases j and $j+1$, there are at least $k + c_{j+1}$ distinct items requested. It follows that the optimal algorithm must incur at least c_{j+1} misses over the course of phases j

[1] It is not, however, the simplest way to incorporate randomization into a caching algorithm. We could have considered the *Purely Random Algorithm* that dispenses with the whole notion of marking, and on each cache miss selects one of its k current items for eviction uniformly at random. (Note the difference: The Randomized Marking Algorithm randomizes only over the unmarked items.) Although we won't prove this here, the Purely Random Algorithm can incur at least c times more misses than the optimum, for any constant $c < k$, and so it does not lead to an improvement over LRU.

and $j+1$, so $f_j(\sigma) + f_{j+1}(\sigma) \geq c_{j+1}$. This holds even for $j = 0$, since the optimal algorithm incurs c_1 misses in phase 1. Thus we have

$$\sum_{j=0}^{r-1} (f_j(\sigma) + f_{j+1}(\sigma)) \geq \sum_{j=0}^{r-1} c_{j+1}.$$

But the left-hand side is at most $2\sum_{j=1}^{r} f_j(\sigma) = 2f(\sigma)$, and the right-hand side is $\sum_{j=1}^{r} c_j$. ■

We now give an upper bound on the expected number of misses incurred by the Randomized Marking Algorithm, also quantified in terms of the number of fresh items in each phase. Combining these upper and lower bounds will yield the performance guarantee we're seeking. In the following statement, let M_σ denote the random variable equal to the number of cache misses incurred by the Randomized Marking Algorithm on the request sequence σ.

(13.40) *For every request sequence σ, we have $E\left[M_\sigma\right] \leq H(k) \sum_{j=1}^{r} c_j$.*

Proof. Recall that we used c_j to denote the number of requests in phase j to fresh items. There are k requests to unmarked items in a phase, and each unmarked item is either fresh or stale, so there must be $k - c_j$ requests in phase j to unmarked stale items.

Let X_j denote the number of misses incurred by the Randomized Marking Algorithm in phase j. Each request to a fresh item results in a guaranteed miss for the Randomized Marking Algorithm; since the fresh item was not marked in the previous phase, it cannot possibly be in the cache when it is requested in phase j. Thus the Randomized Marking Algorithm incurs at least c_j misses in phase j because of requests to fresh items.

Stale items, by contrast, are a more subtle matter. The phase starts with k stale items in the cache; these are the items that were unmarked *en masse* at the beginning of the phase. On a request to a stale item s, the concern is whether the Randomized Marking Algorithm evicted it earlier in the phase and now incurs a miss as it has to bring it back in. What is the probability that the i^{th} request to a stale item, say s, results in a miss? Suppose that there have been $c \leq c_j$ requests to fresh items thus far in the phase. Then the cache contains the c formerly fresh items that are now marked, $i-1$ formerly stale items that are now marked, and $k - c - i + 1$ items that are stale and not yet marked in this phase. But there are $k - i + 1$ items overall that are still stale; and since exactly $k - c - i + 1$ of them are in the cache, the remaining c of them are not. Each of the $k - i + 1$ stale items is equally likely to be no longer in the cache, and so s is not in the cache at this moment with probability $\frac{c}{k-i+1} \leq \frac{c_j}{k-i+1}$.

This is the probability of a miss on the request to s. Summing over all requests to unmarked items, we have

$$E\left[X_j\right] \leq c_j + \sum_{i=1}^{k-c_j} \frac{c_j}{k-i+1} \leq c_j \left[1 + \sum_{\ell=c_j+1}^{k} \frac{1}{\ell}\right] = c_j(1 + H(k) - H(c_j)) \leq c_j H(k).$$

Thus the total expected number of misses incurred by the Randomized Marking Algorithm is

$$E\left[M_\sigma\right] = \sum_{j=1}^{r} E\left[X_j\right] \leq H(k) \sum_{j=1}^{r} c_j. \quad \blacksquare$$

Combining (13.39) and (13.40), we immediately get the following performance guarantee.

(13.41) *The expected number of misses incurred by the Randomized Marking Algorithm is at most* $2H(k) \cdot f(\sigma) = O(\log k) \cdot f(\sigma)$.

13.9 Chernoff Bounds

In Section 13.3, we defined the expectation of a random variable formally and have worked with this definition and its consequences ever since. Intuitively, we have a sense that the value of a random variable ought to be "near" its expectation with reasonably high probability, but we have not yet explored the extent to which this is true. We now turn to some results that allow us to reach conclusions like this, and see a sampling of the applications that follow.

We say that two random variables X and Y are *independent* if, for any values i and j, the events $\Pr[X = i]$ and $\Pr[Y = j]$ are independent. This definition extends naturally to larger sets of random variables. Now consider a random variable X that is a sum of several independent 0-1-valued random variables: $X = X_1 + X_2 + \cdots + X_n$, where X_i takes the value 1 with probability p_i, and the value 0 otherwise. By linearity of expectation, we have $E[X] = \sum_{i=1}^{n} p_i$. Intuitively, the independence of the random variables $X_1, X_2, \ldots, X_n$ suggests that their fluctuations are likely to "cancel out," and so their sum X will have a value close to its expectation with high probability. This is in fact true, and we state two concrete versions of this result: one bounding the probability that X deviates above $E[X]$, the other bounding the probability that X deviates below $E[X]$. We call these results *Chernoff bounds*, after one of the probabilists who first established bounds of this form.

(13.42) *Let $X, X_1, X_2, \ldots, X_n$ be defined as above, and assume that $\mu \geq E[X]$. Then, for any $\delta > 0$, we have*

$$\Pr[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}.$$

Proof. To bound the probability that X exceeds $(1+\delta)\mu$, we go through a sequence of simple transformations. First note that, for any $t > 0$, we have $\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tX} > e^{t(1+\delta)\mu}\right]$, as the function $f(x) = e^{tx}$ is monotone in x. We will use this observation with a t that we'll select later.

Next we use some simple properties of the expectation. For a random variable Y, we have $\gamma \Pr[Y > \gamma] \leq E[Y]$, by the definition of the expectation. This allows us to bound the probability that Y exceeds γ in terms of $E[Y]$. Combining these two ideas, we get the following inequalities.

$$\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tX} > e^{t(1+\delta)\mu}\right] \leq e^{-t(1+\delta)\mu} E\left[e^{tX}\right].$$

Next we need to bound the expectation $E\left[e^{tX}\right]$. Writing X as $X = \sum_i X_i$, the expectation is $E\left[e^{tX}\right] = E\left[e^{t\sum_i X_i}\right] = E\left[\prod_i e^{tX_i}\right]$. For independent variables Y and Z, the expectation of the product YZ is $E[YZ] = E[Y]E[Z]$. The variables X_i are independent, so we get $E\left[\prod_i e^{tX_i}\right] = \prod_i E\left[e^{tX_i}\right]$.

Now, e^{tX_i} is e^t with probability p_i and $e^0 = 1$ otherwise, so its expectation can be bounded as

$$E\left[e^{tX_i}\right] = p_i e^t + (1-p_i) = 1 + p_i(e^t - 1) \leq e^{p_i(e^t-1)},$$

where the last inequality follows from the fact that $1 + \alpha \leq e^{\alpha}$ for any $\alpha \geq 0$. Combining the inequalities, we get the following bound.

$$\begin{aligned}\Pr[X > (1+\delta)\mu] &\leq e^{-t(1+\delta)\mu} E\left[e^{tX}\right] = e^{-t(1+\delta)\mu} \prod_i E\left[e^{tX_i}\right] \\ &\leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t-1)} \leq e^{-t(1+\delta)\mu} e^{\mu(e^t-1)}.\end{aligned}$$

To obtained the bound claimed by the statement, we substitute $t = \ln(1+\delta)$. ∎

Where (13.42) provided an upper bound, showing that X is not likely to deviate far above its expectation, the next statement, (13.43), provides a lower bound, showing that X is not likely to deviate far below its expectation. Note that the statements of the results are not symmetric, and this makes sense: For the upper bound, it is interesting to consider values of δ much larger than 1, while this would not make sense for the lower bound.

(13.43) *Let $X, X_1, X_2, \ldots, X_n$ and μ be defined as above. Then for any $1 > \delta > 0$, we have*

$$\Pr[X < (1-\delta)\mu] < e^{-\frac{1}{2}\mu\delta^2}.$$

The proof of (13.43) is similar to the proof of (13.42), and we do not give it here. For the applications that follow, the statements of (13.42) and (13.43), rather than the internals of their proofs, are the key things to keep in mind.

13.10 Load Balancing

In Section 13.1, we considered a distributed system in which communication among processes was difficult, and randomization to some extent replaced explicit coordination and synchronization. We now revisit this theme through another stylized example of randomization in a distributed setting.

The Problem

Suppose we have a system in which m jobs arrive in a stream and need to be processed immediately. We have a collection of n identical processors that are capable of performing the jobs; so the goal is to assign each job to a processor in a way that balances the workload evenly across the processors. If we had a central controller for the system that could receive each job and hand it off to the processors in round-robin fashion, it would be trivial to make sure that each processor received at most $\lceil m/n \rceil$ jobs—the most even balancing possible.

But suppose the system lacks the coordination or centralization to implement this. A much more lightweight approach would be to simply assign each job to one of the processors uniformly at random. Intuitively, this should also balance the jobs evenly, since each processor is equally likely to get each job. At the same time, since the assignment is completely random, one doesn't expect everything to end up perfectly balanced. So we ask: How well does this simple randomized approach work?

Although we will stick to the motivation in terms of jobs and processors here, it is worth noting that comparable issues come up in the analysis of hash functions, as we saw in Section 13.6. There, instead of assigning jobs to processors, we're assigning elements to entries in a hash table. The concern about producing an even balancing in the case of hash tables is based on wanting to keep the number of collisions at any particular entry relatively small. As a result, the analysis in this section is also relevant to the study of hashing schemes.

Analyzing a Random Allocation

We will see that the analysis of our random load balancing process depends on the relative sizes of m, the number of jobs, and n, the number of processors. We start with a particularly clean case: when $m = n$. Here it is possible for each processor to end up with exactly one job, though this is not very likely. Rather, we expect that some processors will receive no jobs and others will receive more than one. As a way of assessing the quality of this randomized load balancing heuristic, we study how heavily loaded with jobs a processor can become.

Let X_i be the random variable equal to the number of jobs assigned to processor i, for $i = 1, 2, \ldots, n$. It is easy to determine the expected value of X_i: We let Y_{ij} be the random variable equal to 1 if job j is assigned to processor i, and 0 otherwise; then $X_i = \sum_{j=1}^{n} Y_{ij}$ and $E\left[Y_{ij}\right] = 1/n$, so $E\left[X_i\right] = \sum_{j=1}^{n} E\left[Y_{ij}\right] = 1$. But our concern is with how far X_i can deviate above its expectation: What is the probability that $X_i > c$? To give an upper bound on this, we can directly apply (13.42): X_i is a sum of independent 0-1-valued random variables $\{Y_{ij}\}$; we have $\mu = 1$ and $1 + \delta = c$. Thus the following statement holds.

(13.44) $$\Pr\left[X_i > c\right] < \left(\frac{e^{c-1}}{c^c}\right).$$

In order for there to be a small probability of *any* X_i exceeding c, we will take the Union Bound over $i = 1, 2, \ldots, n$; and so we need to choose c large enough to drive $\Pr\left[X_i > c\right]$ down well below $1/n$ for each i. This requires looking at the denominator c^c in (13.44). To make this denominator large enough, we need to understand how this quantity grows with c, and we explore this by first asking the question: What is the x such that $x^x = n$?

Suppose we write $\gamma(n)$ to denote this number x. There is no closed-form expression for $\gamma(n)$, but we can determine its asymptotic value as follows. If $x^x = n$, then taking logarithms gives $x \log x = \log n$; and taking logarithms again gives $\log x + \log\log x = \log\log n$. Thus we have

$$2\log x > \log x + \log\log x = \log\log n > 766 \log x,$$

and, using this to divide through the equation $x \log x = \log n$, we get

$$\frac{1}{2}x \leq \frac{\log n}{\log\log n} \leq x = \gamma(n).$$

Thus $\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$.

Now, if we set $c = e\gamma(n)$, then by (13.44) we have

$$\Pr\left[X_i > c\right] < \left(\frac{e^{c-1}}{c^c}\right) < \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}.$$

Thus, applying the Union Bound over this upper bound for $X_1, X_2, \ldots, X_n$, we have the following.

(13.45) *With probability at least $1 - n^{-1}$, no processor receives more than $e\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$ jobs.*

With a more involved analysis, one can also show that this bound is asymptotically tight: with high probability, some processor actually receives $\Omega\left(\frac{\log n}{\log\log n}\right)$ jobs.

So, although the load on some processors will likely exceed the expectation, this deviation is only logarithmic in the number of processors.

Increasing the Number of Jobs We now use Chernoff bounds to argue that, as more jobs are introduced into the system, the loads "smooth out" rapidly, so that the number of jobs on each processor quickly become the same to within constant factors.

Specifically, if we have $m = 16n \ln n$ jobs, then the expected load per processor is $\mu = 16 \ln n$. Using (13.42), we see that the probability of any processor's load exceeding $32 \ln n$ is at most

$$\Pr\left[X_i > 2\mu\right] < \left(\frac{e}{4}\right)^{16\ln n} < \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}.$$

Also, the probability that any processor's load is below $8 \ln n$ is at most

$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}(\frac{1}{2})^2(16\ln n)} = e^{-2\ln n} = \frac{1}{n^2}.$$

Thus, applying the Union Bound, we have the following.

(13.46) *When there are n processors and $\Omega(n \log n)$ jobs, then with high probability, every processor will have a load between half and twice the average.*

13.11 Packet Routing

We now consider a more complex example of how randomization can alleviate contention in a distributed system—namely, in the context of *packet routing*.

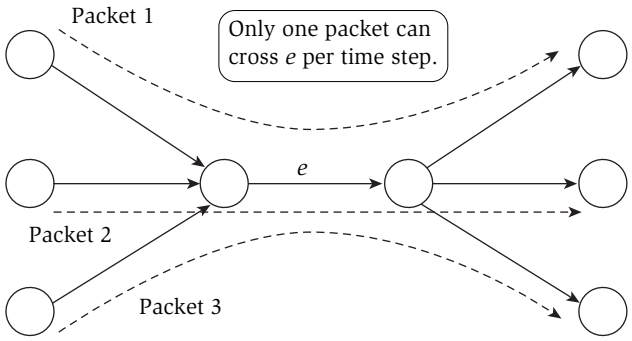

Figure 13.3 Three packets whose paths involve a shared edge e.

The Problem

Packet routing is a mechanism to support communication among nodes of a large network, which we can model as a directed graph $G = (V, E)$. If a node s wants to send data to a node t, this data is discretized into one or more *packets*, each of which is then sent over an s-t path P in the network. At any point in time, there may be many packets in the network, associated with different sources and destinations and following different paths. However, the key constraint is that a single edge e can only transmit a single packet per time step. Thus, when a packet p arrives at an edge e on its path, it may find there are several other packets already waiting to traverse e; in this case, p joins a *queue* associated with e to wait until e is ready to transmit it. In Figure 13.3, for example, three packets with different sources and destinations all want to traverse edge e; so, if they all arrive at e at the same time, some of them will be forced to wait in a queue for this edge.

Suppose we are given a network G with a set of packets that need to be sent across specified paths. We'd like to understand how many steps are necessary in order for all packets to reach their destinations. Although the paths for the packets are all specified, we face the algorithmic question of timing the movements of the packets across the edges. In particular, we must decide when to release each packet from its source, as well as a *queue management policy* for each edge e—that is, how to select the next packet for transmission from e's queue in each time step.

It's important to realize that these *packet scheduling* decisions can have a significant effect on the amount of time it takes for all the packets to reach their destinations. For example, let's consider the tree network in Figure 13.4, where there are nine packets that want to traverse the respective dotted paths up the tree. Suppose all packets are released from their sources immediately, and each edge e manages its queue by always transmitting the packet that is

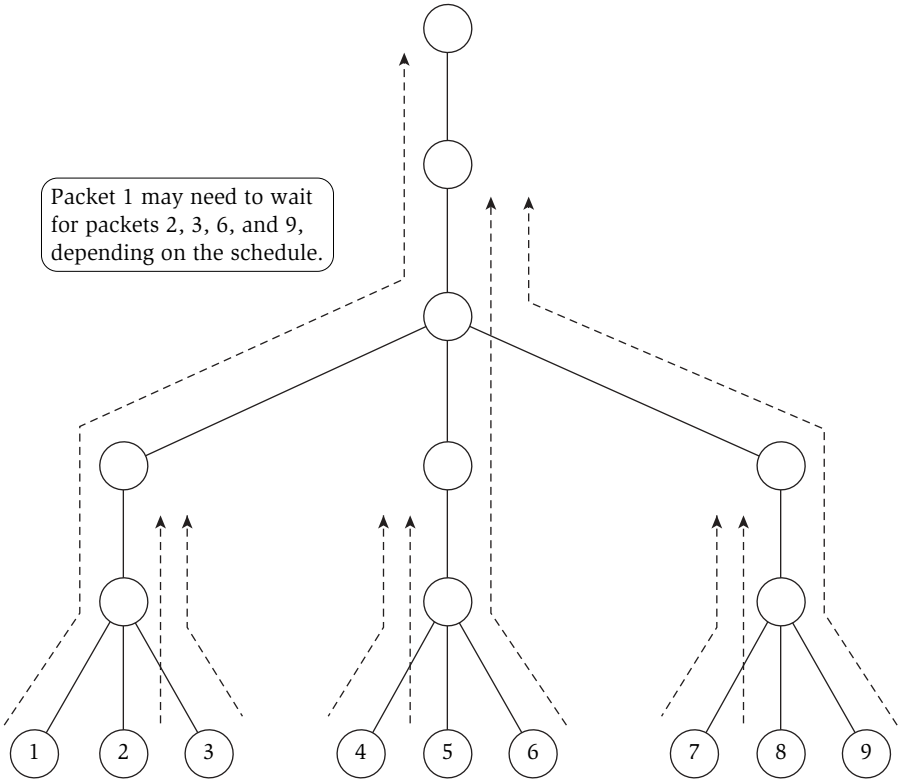

Figure 13.4 A case in which the scheduling of packets matters.

closest to its destination. In this case, packet 1 will have to wait for packets 2 and 3 at the second level of the tree; and then later it will have to wait for packets 6 and 9 at the fourth level of the tree. Thus it will take nine steps for this packet to reach its destination. On the other hand, suppose that each edge e manages its queue by always transmitting the packet that is farthest from its destination. Then packet 1 will never have to wait, and it will reach its destination in five steps; moreover, one can check that every packet will reach its destination within six steps.

There is a natural generalization of the tree network in Figure 13.4, in which the tree has height h and the nodes at every other level have k children. In this case, the queue management policy that always transmits the packet nearest its destination results in some packet requiring $\Omega(hk)$ steps to reach its destination (since the packet traveling farthest is delayed by $\Omega(k)$ steps at each of $\Omega(h)$ levels), while the policy that always transmits the packet farthest from

its destination results in all packets reaching their destinations within $O(h + k)$ steps. This can become quite a large difference as h and k grow large.

Schedules and Their Durations Let's now move from these examples to the question of scheduling packets and managing queues in an arbitrary network G. Given packets labeled $1, 2, \ldots, N$ and associated paths $P_1, P_2, \ldots, P_N$, a *packet schedule* specifies, for each edge e and each time step t, which packet will cross edge e in step t. Of course, the schedule must satisfy some basic consistency properties: at most one packet can cross any edge e in any one step; and if packet i is scheduled to cross e at step t, then e should be on the path P_i, and the earlier portions of the schedule should cause i to have already reached e. We will say that the *duration* of the schedule is the number of steps that elapse until every packet reaches its destination; the goal is to find a schedule of minimum duration.

What are the obstacles to having a schedule of low duration? One obstacle would be a very long path that some packet must traverse; clearly, the duration will be at least the length of this path. Another obstacle would be a single edge e that many packets must cross; since each of these packets must cross e in a distinct step, this also gives a lower bound on the duration. So, if we define the *dilation* d of the set of paths $\{P_1, P_2, \ldots, P_N\}$ to be the maximum length of any P_i, and the *congestion* c of the set of paths to be the maximum number that have any single edge in common, then the duration is at least $\max(c, d) = \Omega(c + d)$.

In 1988, Leighton, Maggs, and Rao proved the following striking result: Congestion and dilation are the only obstacles to finding fast schedules, in the sense that there is always a schedule of duration $O(c + d)$. While the statement of this result is very simple, it turns out to be extremely difficult to prove; and it yields only a very complicated method to actually *construct* such a schedule. So, instead of trying to prove this result, we'll analyze a simple algorithm (also proposed by Leighton, Maggs, and Rao) that can be easily implemented in a distributed setting and yields a duration that is only worse by a logarithmic factor: $O(c + d \log(mN))$, where m is the number of edges and N is the number of packets.

Designing the Algorithm

A Simple Randomized Schedule If each edge simply transmits an arbitrary waiting packet in each step, it is easy to see that the resulting schedule has duration $O(cd)$: at worst, a packet can be blocked by $c - 1$ other packets on each of the d edges in its path. To reduce this bound, we need to set things up so that each packet only waits for a much smaller number of steps over the whole trip to its destination.

The reason a bound as large as $O(cd)$ can arise is that the packets are very badly timed with respect to one another: Blocks of c of them all meet at an edge at the same time, and once this congestion has cleared, the same thing happens at the next edge. This sounds pathological, but one should remember that a very natural queue management policy caused it to happen in Figure 13.4. However, it is the case that such bad behavior relies on very unfortunate synchronization in the motion of the packets; so it is believable that, if we introduce some randomization in the timing of the packets, then this kind of behavior is unlikely to happen. The simplest idea would be just to randomly shift the times at which the packets are released from their sources. Then if there are many packets all aimed at the same edge, they are unlikely to hit it all at the same time, as the contention for edges has been "smoothed out." We now show that this kind of randomization, properly implemented, in fact works quite well.

Consider first the following algorithm, which will not quite work. It involves a parameter r whose value will be determined later.

```
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r
  i waits at its source for s time steps
  i then moves full speed ahead, one edge per time step
     until it reaches its destination
```

If the set of random delays were really chosen so that no two packets ever "collided"—reaching the same edge at the same time—then this schedule would work just as advertised; its duration would be at most r (the maximum initial delay) plus d (the maximum number of edges on any path). However, unless r is chosen to be very large, it is likely that a collision will occur somewhere in the network, and so the algorithm will probably fail: Two packets will show up at the same edge e in the same time step t, and both will be required to cross e in the next step.

Grouping Time into Blocks To get around this problem, we consider the following generalization of this strategy: rather than implementing the "full speed ahead" plan at the level of individual time steps, we implement it at the level of contiguous *blocks* of time steps.

```
For a parameter b, group intervals of b consecutive time steps
      into single blocks of time
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r
  i waits at its source for s blocks
```

```
    i then moves forward one edge per block,
       until it reaches its destination
```

This schedule will work provided that we avoid a more extreme type of collision: It should not be the case that more than b packets are supposed to show up at the same edge e at the start of the same block. If this happens, then at least one of them will not be able to cross e in the next block. However, if the initial delays smooth things out enough so that no more than b packets arrive at any edge in the same block, then the schedule will work just as intended. In this case, the duration will be at most $b(r+d)$—the maximum number of blocks, $r+d$, times the length of each block, b.

(13.47) *Let $\mathcal{E}$ denote the event that more than b packets are required to be at the same edge e at the start of the same block. If $\mathcal{E}$ does not occur, then the duration of the schedule is at most $b(r+d)$.*

Our goal is now to choose values of r and b so that both the probability $\Pr[\mathcal{E}]$ and the duration $b(r+d)$ are small quantities. This is the crux of the analysis since, if we can show this, then (13.47) gives a bound on the duration.

Analyzing the Algorithm

To give a bound on $\Pr[\mathcal{E}]$, it's useful to decompose it into a union of simpler bad events, so that we can apply the Union Bound. A natural set of bad events arises from considering each edge and each time block separately; if e is an edge, and t is a block between 1 and $r+d$, we let $\mathcal{F}_{et}$ denote the event that more than b packets are required to be at e at the start of block t. Clearly, $\mathcal{E} = \cup_{e,t} \mathcal{F}_{et}$. Moreover, if N_{et} is a random variable equal to the number of packets scheduled to be at e at the start of block t, then $\mathcal{F}_{et}$ is equivalent to the event $[N_{et} > b]$.

The next step in the analysis is to decompose the random variable N_{et} into a sum of independent 0-1-valued random variables so that we can apply a Chernoff bound. This is naturally done by defining X_{eti} to be equal to 1 if packet i is required to be at edge e at the start of block t, and equal to 0 otherwise. Then $N_{et} = \sum_i X_{eti}$; and for different values of i, the random variables X_{eti} are independent, since the packets are choosing independent delays. (Note that X_{eti} and $X_{e't'i}$, where the value of i is the same, would certainly not be independent; but our analysis does not require us to add random variables of this form together.) Notice that, of the r possible delays that packet i can choose, at most one will require it to be at e at block t; thus $E[X_{eti}] \leq 1/r$. Moreover, at most c packets have paths that include e; and if i is not one of these packets, then clearly $E[X_{eti}] = 0$. Thus we have

$$E[N_{et}] = \sum_i E[X_{eti}] \leq \frac{c}{r}.$$

We now have the setup for applying the Chernoff bound (13.42), since N_{et} is a sum of the independent 0-1-valued random variables X_{eti}. Indeed, the quantities are sort of like what they were when we analyzed the problem of throwing m jobs at random onto n processors: in that case, each constituent random variable had expectation $1/n$, the total expectation was m/n, and we needed m to be $\Omega(n \log n)$ in order for each processor load to be close to its expectation with high probability. The appropriate analogy in the case at hand is for r to play the role of n, and c to play the role of m: This makes sense symbolically, in terms of the parameters; it also accords with the picture that the packets are like the jobs, and the different time blocks of a single edge are like the different processors that can receive the jobs. This suggests that if we want the number of packets destined for a particular edge in a particular block to be close to its expectation, we should have $c = \Omega(r \log r)$.

This will work, except that we have to increase the logarithmic term a little to make sure that the Union Bound over all e and all t works out in the end. So let's set

$$r = \frac{c}{q \log(mN)},$$

where q is a constant that will be determined later.

Let's fix a choice of e and t and try to bound the probability that N_{et} exceeds a constant times $\frac{c}{r}$. We define $\mu = \frac{c}{r}$, and observe that $E\left[N_{et}\right] \leq \mu$, so we are in a position to apply the Chernoff bound (13.42). We choose $\delta = 2$, so that $(1+\delta)\mu = \frac{3c}{r} = 3q \log(mN)$, and we use this as the upper bound in the expression $\Pr\left[N_{et} > \frac{3c}{r}\right] = \Pr\left[N_{et} > (1+\delta)\mu\right]$. Now, applying (13.42), we have

$$\begin{aligned}\Pr\left[N_{et} > \frac{3c}{r}\right] &< \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} < \left[\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} = \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \\ &= \left(\frac{e}{3}\right)^{(1+\delta)\mu} = \left(\frac{e}{3}\right)^{3c/r} = \left(\frac{e}{3}\right)^{3q\log(mN)} = \frac{1}{(mN)^z},\end{aligned}$$

where z is a constant that can be made as large as we want by choosing the constant q appropriately.

We can see from this calculation that it's safe to set $b = 3c/r$; for, in this case, the event $\mathcal{F}_{et}$ that $N_{et} > b$ will have very small probability for each choice of e and t. There are m different choices for e, and $d + r$ different choice for t, where we observe that $d + r \leq d + c - 1 \leq N$. Thus we have

$$\Pr\left[\mathcal{E}\right] = \Pr\left[\bigcup_{e,t} \mathcal{F}_{et}\right] \leq \sum_{e,t} \Pr\left[\mathcal{F}_{et}\right] \leq mN \cdot \frac{1}{(mN)^z} = \frac{1}{(mN)^{z-1}},$$

which can be made as small as we want by choosing z large enough.

Our choice of the parameters b and r, combined with (13.44), now implies the following.

(13.48) *With high probability, the duration of the schedule for the packets is* $O(c + d \log (mN))$.

Proof. We have just argued that the probability of the bad event $\mathcal{E}$ is very small, at most $(mN)^{-(z-1)}$ for an arbitrarily large constant z. And provided that $\mathcal{E}$ does not happen, (13.47) tells us that the duration of the schedule is bounded by

$$b(r+d) = \frac{3c}{r}\left(r+d\right) = 3c + d \cdot \frac{3c}{r} = 3c + d(3q \log(mN)) = O(c + d \log(mN)).$$

■

13.12 Background: Some Basic Probability Definitions

For many, though certainly not all, applications of randomized algorithms, it is enough to work with probabilities defined over finite sets only; and this turns out to be much easier to think about than probabilities over arbitrary sets. So we begin by considering just this special case. We'll then end the section by revisiting all these notions in greater generality.

Finite Probability Spaces

We have an intuitive understanding of sentences like, "If a fair coin is flipped, the probability of 'heads' is 1/2." Or, "If a fair die is rolled, the probability of a '6' is 1/6." What we want to do first is to describe a mathematical framework in which we can discuss such statements precisely. The framework will work well for carefully circumscribed systems such as coin flips and rolls of dice; at the same time, we will avoid the lengthy and substantial philosophical issues raised in trying to model statements like, "The probability of rain tomorrow is 20 percent." Fortunately, most algorithmic settings are as carefully circumscribed as those of coins and dice, if perhaps somewhat larger and more complex.

To be able to compute probabilities, we introduce the notion of a *finite probability space*. (Recall that we're dealing with just the case of finite sets for now.) A finite probability space is defined by an underlying *sample space* Ω, which consists of the possible *outcomes* of the process under consideration. Each point i in the sample space also has a nonnegative *probability mass* $p(i) \geq 0$; these probability masses need only satisfy the constraint that their total sum is 1; that is, $\sum_{i \in \Omega} p(i) = 1$. We define an *event* $\mathcal{E}$ to be any subset of

Ω—an event is defined simply by the set of outcomes that constitute it—and we define the *probability* of the event to be the sum of the probability masses of all the points in $\mathcal{E}$. That is,

$$\Pr[\mathcal{E}] = \sum_{i \in \mathcal{E}} p(i).$$

In many situations that we'll consider, all points in the sample space have the same probability mass, and then the probability of an event $\mathcal{E}$ is simply its size relative to the size of Ω; that is, in this special case, $\Pr[\mathcal{E}] = |\mathcal{E}|/|\Omega|$. We use $\overline{\mathcal{E}}$ to denote the complementary event $\Omega - \mathcal{E}$; note that $\Pr\left[\overline{\mathcal{E}}\right] = 1 - \Pr[\mathcal{E}]$.

Thus the points in the sample space and their respective probability masses form a complete description of the system under consideration; it is the events—the subsets of the sample space—whose probabilities we are interested in computing. So to represent a single flip of a "fair" coin, we can define the sample space to be $\Omega = \{\texttt{heads}, \texttt{tails}\}$ and set $p(\texttt{heads}) = p(\texttt{tails}) = 1/2$. If we want to consider a biased coin in which "heads" is twice as likely as "tails," we can define the probability masses to be $p(\texttt{heads}) = 2/3$ and $p(\texttt{tails}) = 1/3$. A key thing to notice even in this simple example is that defining the probability masses is a part of defining the underlying problem; in setting up the problem, we are specifying whether the coin is fair or biased, not deriving this from some more basic data.

Here's a slightly more complex example, which we could call the *Process Naming*, or *Identifier Selection Problem*. Suppose we have n processes in a distributed system, denoted $p_1, p_2, \ldots, p_n$, and each of them chooses an identifier for itself uniformly at random from the space of all k-bit strings. Moreover, each process's choice happens concurrently with those of all the other processes, and so the outcomes of these choices are unaffected by one another. If we view each identifier as being chosen from the set $\{0, 1, 2, \ldots, 2^k - 1\}$ (by considering the numerical value of the identifier as a number in binary notation), then the sample space Ω could be represented by the set of all n-tuples of integers, with each integer between 0 and $2^k - 1$. The sample space would thus have $(2^k)^n = 2^{kn}$ points, each with probability mass 2^{-kn}.

Now suppose we are interested in the probability that processes p_1 and p_2 each choose the same name. This is an event $\mathcal{E}$, represented by the subset consisting of all n-tuples from Ω whose first two coordinates are the same. There are $2^{k(n-1)}$ such n-tuples: we can choose any value for coordinates 3 through n, then any value for coordinate 2, and then we have no freedom of choice in coordinate 1. Thus we have

$$\Pr[\mathcal{E}] = \sum_{i \in \mathcal{E}} p(i) = 2^{k(n-1)} \cdot 2^{-kn} = 2^{-k}.$$

This, of course, corresponds to the intuitive way one might work out the probability, which is to say that we can choose any identifier we want for process p_2, after which there is only 1 choice out of 2^k for process p_1 that will cause the names to agree. It's worth checking that this intuition is really just a compact description of the calculation above.

Conditional Probability and Independence

If we view the probability of an event $\mathcal{E}$, roughly, as the likelihood that $\mathcal{E}$ is going to occur, then we may also want to ask about its probability given additional information. Thus, given another event $\mathcal{F}$ of positive probability, we define the *conditional probability of $\mathcal{E}$ given $\mathcal{F}$* as

$$\Pr[\mathcal{E} \mid \mathcal{F}] = \frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]}.$$

This is the "right" definition intuitively, since it's performing the following calculation: Of the portion of the sample space that consists of $\mathcal{F}$ (the event we "know" to have occurred), what fraction is occupied by $\mathcal{E}$?

One often uses conditional probabilities to analyze $\Pr[\mathcal{E}]$ for some complicated event $\mathcal{E}$, as follows. Suppose that the events $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$ each have positive probability, and they partition the sample space; in other words, each outcome in the sample space belongs to exactly one of them, so $\sum_{j=1}^{k} \Pr[\mathcal{F}_j] = 1$. Now suppose we know these values $\Pr[\mathcal{F}_j]$, and we are also able to determine $\Pr[\mathcal{E} \mid \mathcal{F}_j]$ for each $j = 1, 2, \ldots, k$. That is, we know what the probability of $\mathcal{E}$ is if we assume that any one of the events $\mathcal{F}_j$ has occurred. Then we can compute $\Pr[\mathcal{E}]$ by the following simple formula:

$$\Pr[\mathcal{E}] = \sum_{j=1}^{k} \Pr[\mathcal{E} \mid \mathcal{F}_j] \cdot \Pr[\mathcal{F}_j].$$

To justify this formula, we can unwind the right-hand side as follows:

$$\sum_{j=1}^{k} \Pr[\mathcal{E} \mid \mathcal{F}_j] \cdot \Pr[\mathcal{F}_j] = \sum_{j=1}^{k} \frac{\Pr[\mathcal{E} \cap \mathcal{F}_j]}{\Pr[\mathcal{F}_j]} \cdot \Pr[\mathcal{F}_j] = \sum_{j=1}^{k} \Pr[\mathcal{E} \cap \mathcal{F}_j] = \Pr[\mathcal{E}].$$

Independent Events Intuitively, we say that two events are *independent* if information about the outcome of one does not affect our estimate of the likelihood of the other. One way to make this concrete would be to declare events $\mathcal{E}$ and $\mathcal{F}$ independent if $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$, and $\Pr[\mathcal{F} \mid \mathcal{E}] = \Pr[\mathcal{F}]$. (We'll assume here that both have positive probability; otherwise the notion of independence is not very interesting in any case.) Actually, if one of these two equalities holds, then the other must hold, for the following reason: If $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$, then

$$\frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]} = \Pr[\mathcal{E}],$$

and hence $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$, from which the other equality holds as well.

It turns out to be a little cleaner to adopt this equivalent formulation as our working definition of independence. Formally, we'll say that events $\mathcal{E}$ and $\mathcal{F}$ are *independent* if $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$.

This product formulation leads to the following natural generalization. We say that a collection of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ is *independent* if, for every set of indices $I \subseteq \{1, 2, \ldots, n\}$, we have

$$\Pr\left[\bigcap_{i \in I} \mathcal{E}_i\right] = \prod_{i \in I} \Pr\left[\mathcal{E}_i\right].$$

It's important to notice the following: To check if a large set of events is independent, it's not enough to check whether every pair of them is independent. For example, suppose we flip three independent fair coins: If $\mathcal{E}_i$ denotes the event that the i^{th} coin comes up `heads`, then the events $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ are independent and each has probability 1/2. Now let A denote the event that coins 1 and 2 have the same value; let B denote the event that coins 2 and 3 have the same value; and let C denote the event that coins 1 and 3 have different values. It's easy to check that each of these events has probability 1/2, and the intersection of any two has probability 1/4. Thus every pair drawn from A, B, C is independent. But the set of all three events A, B, C is not independent, since $\Pr[A \cap B \cap C] = 0$.

The Union Bound

Suppose we are given a set of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, and we are interested in the probability that *any* of them happens; that is, we are interested in the probability $\Pr\left[\cup_{i=1}^{n} \mathcal{E}_i\right]$. If the events are all pairwise disjoint from one another, then the probability mass of their union is comprised simply of the separate contributions from each event. In other words, we have the following fact.

(13.49) *Suppose we have events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ such that $\mathcal{E}_i \cap \mathcal{E}_j = \phi$ for each pair. Then*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] = \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

In general, a set of events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ may overlap in complex ways. In this case, the equality in (13.49) no longer holds; due to the overlaps among

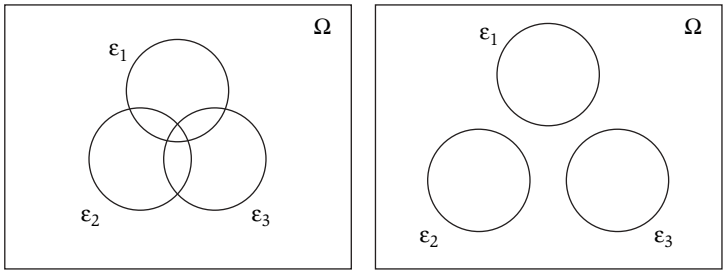

Figure 13.5 The Union Bound: The probability of a union is maximized when the events have no overlap.

events, the probability mass of a point that is counted once on the left-hand side will be counted one *or more* times on the right-hand side. (See Figure 13.5.) This means that for a general set of events, the equality in (13.49) is relaxed to an inequality; and this is the content of the Union Bound. We have stated the Union Bound as (13.2), but we state it here again for comparison with (13.49).

(13.50) (The Union Bound) *Given events* $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, *we have*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Given its innocuous appearance, the Union Bound is a surprisingly powerful tool in the analysis of randomized algorithms. It draws its power mainly from the following ubiquitous style of analyzing randomized algorithms. Given a randomized algorithm designed to produce a correct result with high probability, we first tabulate a set of "bad events" $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ with the following property: if none of these bad events occurs, then the algorithm will indeed produce the correct answer. In other words, if $\mathcal{F}$ denotes the event that the algorithm fails, then we have

$$\Pr\left[\mathcal{F}\right] \leq \Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right].$$

But it's hard to compute the probability of this union, so we apply the Union Bound to conclude that

$$\Pr\left[\mathcal{F}\right] \leq \Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Now, if in fact we have an algorithm that succeeds with very high probability, and if we've chosen our bad events carefully, then each of the probabilities $\Pr[\mathcal{E}_i]$ will be so small that even their sum—and hence our overestimate of the failure probability—will be small. This is the key: decomposing a highly complicated event, the failure of the algorithm, into a horde of simple events whose probabilities can be easily computed.

Here is a simple example to make the strategy discussed above more concrete. Recall the Process Naming Problem we discussed earlier in this section, in which each of a set of processes chooses a random identifier. Suppose that we have 1,000 processes, each choosing a 32-bit identifier, and we are concerned that two of them will end up choosing the same identifier. Can we argue that it is unlikely this will happen? To begin with, let's denote this event by $\mathcal{F}$. While it would not be overwhelmingly difficult to compute $\Pr[\mathcal{F}]$ exactly, it is much simpler to bound it as follows. The event $\mathcal{F}$ is really a union of $\binom{1000}{2}$ "atomic" events; these are the events $\mathcal{E}_{ij}$ that processes p_i and p_j choose the same identifier. It is easy to verify that indeed, $\mathcal{F} = \cup_{i<j}\mathcal{E}_{ij}$. Now, for any $i \neq j$, we have $\Pr[\mathcal{E}_{ij}] = 2^{-32}$, by the argument in one of our earlier examples. Applying the Union Bound, we have

$$\Pr[\mathcal{F}] \leq \sum_{i,j} \Pr[\mathcal{E}_{ij}] = \binom{1000}{2} \cdot 2^{-32}.$$

Now, $\binom{1000}{2}$ is at most half a million, and 2^{32} is (a little bit) more than 4 billion, so this probability is at most $\frac{.5}{4000} = .000125$.

Infinite Sample Spaces

So far we've gotten by with finite probability spaces only. Several of the sections in this chapter, however, consider situations in which a random process can run for arbitrarily long, and so cannot be well described by a sample space of finite size. As a result, we pause here to develop the notion of a probability space more generally. This will be somewhat technical, and in part we are providing it simply for the sake of completeness: Although some of our applications require infinite sample spaces, none of them really exercises the full power of the formalism we describe here.

Once we move to infinite sample spaces, more care is needed in defining a probability function. We cannot simply give each point in the sample space Ω a probability mass and then compute the probability of every set by summing. Indeed, for reasons that we will not go into here, it is easy to get into trouble if one even allows every subset of Ω to be an event whose probability can be computed. Thus a general probability space has three components:

(i) The sample space Ω.

(ii) A collection $\mathcal{S}$ of subsets of Ω; these are the only events on which we are allowed to compute probabilities.

(iii) A probability function Pr, which maps events in $\mathcal{S}$ to real numbers in $[0, 1]$.

The collection $\mathcal{S}$ of allowable events can be any family of sets that satisfies the following basic closure properties: the empty set and the full sample space Ω both belong to $\mathcal{S}$; if $\mathcal{E} \in \mathcal{S}$, then $\overline{\mathcal{E}} \in \mathcal{S}$ (closure under complement); and if $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in \mathcal{S}$, then $\cup_{i=1}^{\infty} \mathcal{E}_i \in \mathcal{S}$ (closure under countable union). The probability function Pr can be any function from $\mathcal{S}$ to $[0, 1]$ that satisfies the following basic consistency properties: $\Pr[\phi] = 0$, $\Pr[\Omega] = 1$, $\Pr[\mathcal{E}] = 1 - \Pr\left[\overline{\mathcal{E}}\right]$, and the Union Bound for disjoint events (13.49) should hold even for countable unions—if $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in \mathcal{S}$ are all pairwise disjoint, then

$$\Pr\left[\bigcup_{i=1}^{\infty} \mathcal{E}_i\right] = \sum_{i=1}^{\infty} \Pr\left[\mathcal{E}_i\right].$$

Notice how, since we are not building up Pr from the more basic notion of a probability mass anymore, (13.49) moves from being a theorem to simply a required property of Pr.

When an infinite sample space arises in our context, it's typically for the following reason: we have an algorithm that makes a sequence of random decisions, each one from a fixed finite set of possibilities; and since it may run for arbitrarily long, it may make an arbitrarily large number of decisions. Thus we consider sample spaces Ω constructed as follows. We start with a finite set of symbols $X = \{1, 2, \ldots, n\}$, and assign a *weight* $w(i)$ to each symbol $i \in X$. We then define Ω to be the set of all infinite sequences of symbols from X (with repetitions allowed). So a typical element of Ω will look like $\langle x_1, x_2, x_3, \ldots \rangle$ with each entry $x_i \in X$.

The simplest type of event we will be concerned with is as follows: it is the event that a point $\omega \in \Omega$ begins with a particular finite sequence of symbols. Thus, for a finite sequence $\sigma = x_1 x_2 \ldots x_s$ of length s, we define the *prefix event associated with* σ to be the set of all sample points of Ω whose first s entries form the sequence σ. We denote this event by $\mathcal{E}_\sigma$, and we define its probability to be $\Pr\left[\mathcal{E}_\sigma\right] = w(x_1)w(x_2) \cdots w(x_s)$.

The following fact is in no sense easy to prove.

(13.51) *There is a probability space* $(\Omega, \mathcal{S}, \Pr)$*, satisfying the required closure and consistency properties, such that* Ω *is the sample space defined above,* $\mathcal{E}_\sigma \in \mathcal{S}$ *for each finite sequence* σ*, and* $\Pr\left[\mathcal{E}_\sigma\right] = w(x_1)w(x_2)\cdots w(x_s)$.

Once we have this fact, the closure of $\mathcal{S}$ under complement and countable union, and the consistency of Pr with respect to these operations, allow us to compute probabilities of essentially any "reasonable" subset of Ω.

In our infinite sample space Ω, with events and probabilities defined as above, we encounter a phenomenon that does not naturally arise with finite sample spaces. Suppose the set X used to generate Ω is equal to $\{0, 1\}$, and $w(0) = w(1) = 1/2$. Let $\mathcal{E}$ denote the set consisting of all sequences that contain at least one entry equal to 1. (Note that $\mathcal{E}$ omits the "all-0" sequence.) We observe that $\mathcal{E}$ is an event in $\mathcal{S}$, since we can define σ_i to be the sequence of $i-1$ 0s followed by a 1, and observe that $\mathcal{E} = \cup_{i=1}^{\infty} \mathcal{E}_{\sigma_i}$. Moreover, all the events $\mathcal{E}_{\sigma_i}$ are pairwise disjoint, and so

$$\Pr[\mathcal{E}] = \sum_{i=1}^{\infty} \Pr\left[\mathcal{E}_{\sigma_i}\right] = \sum_{i=1}^{\infty} 2^{-i} = 1.$$

Here, then, is the phenomenon: It's possible for an event to have probability 1 even when it's not equal to the whole sample space Ω. Similarly, $\Pr\left[\overline{\mathcal{E}}\right] = 1 - \Pr[\mathcal{E}] = 0$, and so we see that it's possible for an event to have probability 0 even when it's not the empty set. There is nothing wrong with any of these results; in a sense, it's a necessary step if we want probabilities defined over infinite sets to make sense. It's simply that in such cases, we should be careful to distinguish between the notion that an event has probability 0 and the intuitive idea that the event "can't happen."

Solved Exercises

Solved Exercise 1

Suppose we have a collection of small, low-powered devices scattered around a building. The devices can exchange data over short distances by wireless communication, and we suppose for simplicity that each device has enough range to communicate with d other devices. Thus we can model the wireless connections among these devices as an undirected graph $G = (V, E)$ in which each node is incident to exactly d edges.

Now we'd like to give some of the nodes a stronger *uplink transmitter* that they can use to send data back to a base station. Giving such a transmitter to every node would ensure that they can all send data like this, but we can achieve this while handing out fewer transmitters. Suppose that we find a

subset S of the nodes with the property that every node in $V - S$ is adjacent to a node in S. We call such a set S a *dominating set*, since it "dominates" all other nodes in the graph. If we give uplink transmitters only to the nodes in a dominating set S, we can still extract data from all nodes: Any node $u \notin S$ can choose a neighbor $v \in S$, send its data to v, and have v relay the data back to the base station.

The issue is now to find a dominating set S of minimum possible size, since this will minimize the number of uplink transmitters we need. This is an NP-hard problem; in fact, proving this is the crux of Exercise 29 in Chapter 8. (It's also worth noting here the difference between dominating sets and vertex covers: in a dominating set, it is fine to have an edge (u, v) with neither u nor v in the set S as long as both u and v have neighbors in S. So, for example, a graph consisting of three nodes all connected by edges has a dominating set of size 1, but no vertex cover of size 1.)

Despite the NP-hardness, it's important in applications like this to find as small a dominating set as one can, even if it is not optimal. We will see here that a simple randomized strategy can be quite effective. Recall that in our graph G, each node is incident to exactly d edges. So clearly any dominating set will need to have size at least $\frac{n}{d+1}$, since each node we place in a dominating set can take care only of itself and its d neighbors. We want to show that a random selection of nodes will, in fact, get us quite close to this simple lower bound.

Specifically, show that for some constant c, a set of $\frac{cn \log n}{d+1}$ nodes chosen uniformly at random from G will be a dominating set with high probability. (In other words, this completely random set is likely to form a dominating set that is only $O(\log n)$ times larger than our simple lower bound of $\frac{n}{d+1}$.)

Solution Let $k = \frac{cn \log n}{d}$, where we will choose the constant c later, once we have a better idea of what's going on. Let $\mathcal{E}$ be the event that a random choice of k nodes is a dominating set for G. To make the analysis simpler, we will consider a model in which the nodes are selected one at a time, and the same node may be selected twice (if it happens to be picked twice by our sequence of random choices).

Now we want to show that if c (and hence k) is large enough, then $\Pr[\mathcal{E}]$ is close to 1. But $\mathcal{E}$ is a very complicated-looking event, so we begin by breaking it down into much simpler events whose probabilities we can analyze more easily.

To start with, we say that a node w *dominates* a node v if w is a neighbor of v, or $w = v$. We say that a set S dominates a node v if some element of S dominates v. (These definitions let us say that a dominating set is simply a set of nodes that dominates every node in the graph.) Let $\mathcal{D}[v, t]$ denote the

event that the t^{th} random node we choose dominates node v. The probability of this event can be determined quite easily: of the n nodes in the graph, we must choose v or one of its d neighbors, and so

$$\Pr\left[\mathcal{D}[v,t]\right] = \frac{d+1}{n}.$$

Let $\mathcal{D}_v$ denote the event that the random set consisting of all k selected nodes dominates v. Thus

$$\mathcal{D}_v = \bigcup_{t=1}^{k} \mathcal{D}[v,t].$$

For independent events, we've seen in the text that it's easier to work with intersections—where we can simply multiply out the probabilities—than with unions. So rather than thinking about $\mathcal{D}_v$, we'll consider the complementary "failure event" $\overline{\mathcal{D}_v}$, that no node in the random set dominates v. In order for no node to dominate v, each of our choices has to fail to do so, and hence we have

$$\overline{\mathcal{D}_v} = \bigcap_{t=1}^{k} \overline{\mathcal{D}[v,t]}.$$

Since the events $\overline{\mathcal{D}[v,t]}$ are independent, we can compute the probability on the right-hand side by multiplying all the individual probabilities; thus

$$\Pr\left[\overline{\mathcal{D}_v}\right] = \prod_{t=1}^{k} \Pr\left[\overline{\mathcal{D}[v,t]}\right] = \left(1 - \frac{d+1}{n}\right)^k.$$

Now, $k = \frac{cn \log n}{d+1}$, so we can write this last expression as

$$\left(1 - \frac{d+1}{n}\right)^k = \left[\left(1 - \frac{d+1}{n}\right)^{n/(d+1)}\right]^{c \log n} \leq \left(\frac{1}{e}\right)^{c \log n},$$

where the inequality follows from (13.1) that we stated earlier in the chapter.

We have not yet specified the base of the logarithm we use to define k, but it's starting to look like base e is a good choice. Using this, we can further simplify the last expression to

$$\Pr\left[\overline{\mathcal{D}_v}\right] \leq \left(\frac{1}{e}\right)^{c \ln n} = \frac{1}{n^c}.$$

We are now very close to done. We have shown that for each node v, the probability that our random set fails to dominate it is at most n^{-c}, which we can drive down to a very small quantity by making c moderately large. Now recall the original event $\mathcal{E}$, that our random set is a dominating set. This fails

to occur if and only if one of the events $\mathcal{D}_v$ fails to occur, so $\overline{\mathcal{E}} = \cup_v \overline{\mathcal{D}_v}$. Thus, by the Union Bound (13.2), we have

$$\Pr\left[\overline{\mathcal{E}}\right] \leq \sum_{v \in V} \Pr\left[\overline{\mathcal{D}_v}\right] \leq n \cdot \frac{1}{n^c} = \frac{1}{n^{c-1}}.$$

Simply choosing $c = 2$ makes this probability $\frac{1}{n}$, which is much less than 1. Thus, with high probability, the event $\mathcal{E}$ holds and our random choice of nodes is indeed a dominating set.

It's interesting to note that the probability of success, as a function of k, exhibits behavior very similar to what we saw in the contention-resolution example in Section 13.1. Setting $k = \Theta(n/d)$ is enough to guarantee that each individual node is dominated with constant probability. This, however, is not enough to get anything useful out of the Union Bound. Then, raising k by another logarithmic factor is enough to drive up the probability of dominating each node to something very close to 1, at which point the Union Bound can come into play.

Solved Exercise 2

Suppose we are given a set of n variables $x_1, x_2, \ldots, x_n$, each of which can take one of the values in the set $\{0, 1\}$. We are also given a set of k equations; the r^{th} equation has the form

$$(x_i + x_j) \bmod 2 = b_r$$

for some choice of two distinct variables x_i, x_j, and for some value b_r that is either 0 or 1. Thus each equation specifies whether the sum of two variables is even or odd.

Consider the problem of finding an assignment of values to variables that maximizes the number of equations that are satisfied (i.e., in which equality actually holds). This problem is NP-hard, though you don't have to prove this.

For example, suppose we are given the equations

$$(x_1 + x_2) \bmod 2 = 0$$

$$(x_1 + x_3) \bmod 2 = 0$$

$$(x_2 + x_4) \bmod 2 = 1$$

$$(x_3 + x_4) \bmod 2 = 0$$

over the four variables $x_1, \ldots, x_4$. Then it's possible to show that no assignment of values to variables will satisfy all equations simultaneously, but setting all variables equal to 0 satisfies three of the four equations.

(a) Let c^* denote the maximum possible number of equations that can be satisfied by an assignment of values to variables. Give a polynomial-time algorithm that produces an assignment satisfying at least $\frac{1}{2}c^*$ equations. If you want, your algorithm can be randomized; in this case, the *expected* number of equations it satisfies should be at least $\frac{1}{2}c^*$. In either case, you should prove that your algorithm has the desired performance guarantee.

(b) Suppose we drop the condition that each equation must have exactly two variables; in other words, now each equation simply specifies that the sum of an arbitrary subset of the variables, mod 2, is equal to a particular value b_r.

Again let c^* denote the maximum possible number of equations that can be satisfied by an assignment of values to variables, and give a polynomial-time algorithm that produces an assignment satisfying at least $\frac{1}{2}c^*$ equations. (As before, your algorithm can be randomized.) If you believe that your algorithm from part (a) achieves this guarantee here as well, you can state this and justify it with a proof of the performance guarantee for this more general case.

Solution Let's recall the punch line of the simple randomized algorithm for MAX 3-SAT that we saw earlier in the chapter: If you're given a constraint satisfaction problem, assigning variables at random can be a surprisingly effective way to satisfy a constant fraction of all constraints.

We now try applying this principle to the problem here, beginning with part (a). Consider the algorithm that sets each variable independently and uniformly at random. How well does this random assignment do, in expectation? As usual, we will approach this question using linearity of expectation: If X is a random variable denoting the number of satisfied equations, we'll break X up into a sum of simpler random variables.

For some r between 1 and k, let the r^{th} equation be

$$(x_i + x_j) \bmod 2 = b_r.$$

Let X_r be a random variable equal to 1 if this equation is satisfied, and 0 otherwise. $E\left[X_r\right]$ is the probability that equation r is satisfied. Of the four possible assignments to equation i, there are two that cause it to evaluate to 0 mod 2 ($x_i = x_j = 0$ and $x_i = x_j = 1$) and two that cause it to evaluate to 1 mod 2 ($x_i = 0; x_j = 1$ and $x_i = i; x_j = 0$). Thus $E\left[X_r\right] = 2/4 = 1/2$.

Now, by linearity of expectation, we have $E[X] = \sum_r E\left[X_r\right] = k/2$. Since the maximum number of satisfiable equations c^* must be at most k, we satisfy at least $c^*/2$ in expectation. Thus, as in the case of MAX 3-SAT, a simple random assignment to the variables satisfies a constant fraction of all constraints.

For part (b), let's press our luck by trying the same algorithm. Again let X_r be a random variable equal to 1 if the r^{th} equation is satisfied, and 0 otherwise; let X be the total number of satisfied equations; and let c^* be the optimum.

We want to claim that $E\left[X_r\right] = 1/2$ as before, even when there can be an arbitrary number of variables in the r^{th} equation; in other words, the probability that the equation takes the correct value mod 2 is exactly 1/2. We can't just write down all the cases the way we did for two variables per equation, so we will use an alternate argument.

In fact, there are two natural ways to prove that $E\left[X_r\right] = 1/2$. The first uses a trick that appeared in the proof of (13.25) in Section 13.6 on hashing: We consider assigning values arbitrarily to all variables but the last one in the equation, and then we randomly assign a value to the last variable x. Now, regardless of how we assign values to all other variables, there are two ways to assign a value to x, and it is easy to check that one of these ways will satisfy the equation and the other will not. Thus, regardless of the assignments to all variables other than x, the probability of setting x so as to satisfy the equation is exactly 1/2. Thus the probability the equation is satisfied by a random assignment is 1/2.

(As in the proof of (13.25), we can write this argument in terms of conditional probabilities. If $\mathcal{E}$ is the event that the equation is satisfied, and $\mathcal{F}_b$ is the event that the variables other than x receive a sequence of values b, then we have argued that $\Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] = 1/2$ for all b, and so $\Pr\left[\mathcal{E}\right] = \sum_b \Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] \cdot \Pr\left[\mathcal{F}_b\right] = (1/2) \sum_b \Pr\left[\mathcal{F}_b\right] = 1/2$.)

An alternate proof simply counts the number of ways for the r^{th} equation to have an even sum, and the number of ways for it to have an odd sum. If we can show that these two numbers are equal, then the probability that a random assignment satisfies the r^{th} equation is the probability it gives it a sum with the right even/odd parity, which is 1/2.

In fact, at a high level, this proof is essentially the same as the previous one, with the difference that we make the underlying counting problem explicit. Suppose that the r^{th} equation has t terms; then there are 2^t possible assignments to the variables in this equation. We want to claim that 2^{t-1} assignments produce an even sum, and 2^{t-1} produce an odd sum, which will show that $E\left[X_r\right] = 1/2$. We prove this by induction on t. For $t = 1$, there are just two assignments, one of each parity; and for $t = 2$, we already proved this earlier by considering all $2^2 = 4$ possible assignments. Now suppose the claim holds for an arbitrary value of $t - 1$. Then there are exactly 2^{t-1} ways to get an even sum with t variables, as follows:

- 2^{t-2} ways to get an even sum on the first $t - 1$ variables (by induction), followed by an assignment of 0 to the t^{th}, plus

- 2^{t-2} ways to get an odd sum on the first $t-1$ variables (by induction), followed by an assignment of 1 to the t^{th}.

The remaining 2^{t-1} assignments give an odd sum, and this completes the induction step.

Once we have $E\left[X_r\right] = 1/2$, we conclude as in part (a): Linearity of expectation gives us $E[X] = \sum_r E\left[X_r\right] = k/2 \geq c^*/2$.

Exercises

1. *3-Coloring* is a yes/no question, but we can phrase it as an optimization problem as follows.

Suppose we are given a graph $G = (V, E)$, and we want to color each node with one of three colors, even if we aren't necessarily able to give different colors to every pair of adjacent nodes. Rather, we say that an edge (u, v) is *satisfied* if the colors assigned to u and v are different.

Consider a 3-coloring that maximizes the number of satisfied edges, and let c^* denote this number. Give a polynomial-time algorithm that produces a 3-coloring that satisfies at least $\frac{2}{3}c^*$ edges. If you want, your algorithm can be randomized; in this case, the *expected* number of edges it satisfies should be at least $\frac{2}{3}c^*$.

2. Consider a county in which 100,000 people vote in an election. There are only two candidates on the ballot: a Democratic candidate (denoted D) and a Republican candidate (denoted R). As it happens, this county is heavily Democratic, so 80,000 people go to the polls with the intention of voting for D, and 20,000 go to the polls with the intention of voting for R.

However, the layout of the ballot is a little confusing, so each voter, independently and with probability $\frac{1}{100}$, votes for the wrong candidate—that is, the one that he or she *didn't* intend to vote for. (Remember that in this election, there are only two candidates on the ballot.)

Let X denote the random variable equal to the number of votes received by the Democratic candidate D, when the voting is conducted with this process of error. Determine the expected value of X, and give an explanation of your derivation of this value.

3. In Section 13.1, we saw a simple distributed protocol to solve a particular contention-resolution problem. Here is another setting in which randomization can help with contention resolution, through the distributed construction of an independent set.

Suppose we have a system with n processes. Certain pairs of processes are in *conflict*, meaning that they both require access to a shared resource. In a given time interval, the goal is to schedule a large subset S of the processes to run—the rest will remain idle—so that no two conflicting processes are both in the scheduled set S. We'll call such a set S *conflict-free*.

One can picture this process in terms of a graph $G = (V, E)$ with a node representing each process and an edge joining pairs of processes that are in conflict. It is easy to check that a set of processes S is conflict-free if and only if it forms an independent set in G. This suggests that finding a maximum-size conflict-free set S, for an arbitrary conflict G, will be difficult (since the general Independent Set Problem is reducible to this problem). Nevertheless, we can still look for heuristics that find a reasonably large conflict-free set. Moreover, we'd like a simple method for achieving this without centralized control: Each process should communicate with only a small number of other processes and then decide whether or not it should belong to the set S.

We will suppose for purposes of this question that each node has exactly d neighbors in the graph G. (That is, each process is in conflict with exactly d other processes.)

(a) Consider the following simple protocol.

> *Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability $\frac{1}{2}$ and sets x_i to 0 with probability $\frac{1}{2}$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.*

Prove that the set S resulting from the execution of this protocol is conflict-free. Also, give a formula for the expected size of S in terms of n (the number of processes) and d (the number of conflicts per process).

(b) The choice of the probability $\frac{1}{2}$ in the protocol above was fairly arbitrary, and it's not clear that it should give the best system performance. A more general specification of the protocol would replace the probability $\frac{1}{2}$ by a parameter p between 0 and 1, as follows.

> *Each process P_i independently picks a random value x_i; it sets x_i to 1 with probability p and sets x_i to 0 with probability $1 - p$. It then decides to enter the set S if and only if it chooses the value 1, and each of the processes with which it is in conflict chooses the value 0.*

In terms of the parameters of the graph G, give a value of p so that the expected size of the resulting set S is as large as possible. Give a formula for the expected size of S when p is set to this optimal value.

4. A number of *peer-to-peer systems* on the Internet are based on *overlay networks*. Rather than using the physical Internet topology as the network on which to perform computation, these systems run protocols by which nodes choose collections of virtual "neighbors" so as to define a higher-level graph whose structure may bear little or no relation to the underlying physical network. Such an overlay network is then used for sharing data and services, and it can be extremely flexible compared with a physical network, which is hard to modify in real time to adapt to changing conditions.

Peer-to-peer networks tend to grow through the arrival of new participants, who join by linking into the existing structure. This growth process has an intrinsic effect on the characteristics of the overall network. Recently, people have investigated simple abstract models for network growth that might provide insight into the way such processes behave, at a qualitative level, in real networks.

Here's a simple example of such a model. The system begins with a single node v_1. Nodes then join one at a time; as each node joins, it executes a protocol whereby it forms a directed link to a single other node chosen uniformly at random from those already in the system. More concretely, if the system already contains nodes $v_1, v_2, \ldots, v_{k-1}$ and node v_k wishes to join, it randomly selects one of $v_1, v_2, \ldots, v_{k-1}$ and links to this node.

Suppose we run this process until we have a system consisting of nodes $v_1, v_2, \ldots, v_n$; the random process described above will produce a directed network in which each node other than v_1 has exactly one outgoing edge. On the other hand, a node may have multiple incoming links, or none at all. The incoming links to a node v_j reflect all the other nodes whose access into the system is via v_j; so if v_j has many incoming links, this can place a large load on it. To keep the system load-balanced, then, we'd like all nodes to have a roughly comparable number of incoming links. That's unlikely to happen here, however, since nodes that join earlier in the process are likely to have more incoming links than nodes that join later. Let's try to quantify this imbalance as follows.

(a) Given the random process described above, what is the expected number of incoming links to node v_j in the resulting network? Give an exact formula in terms of n and j, and also try to express this quantity

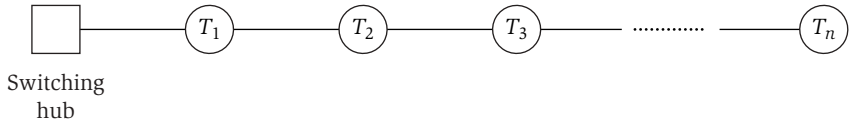

Figure 13.6 Towns $T_1, T_2 \ldots, T_n$ need to decide how to share the cost of the cable.

asymptotically (via an expression without large summations) using $\Theta(\cdot)$ notation.

(b) Part (a) makes precise a sense in which the nodes that arrive early carry an "unfair" share of the connections in the network. Another way to quantify the imbalance is to observe that, in a run of this random process, we expect many nodes to end up with no incoming links.

Give a formula for the expected number of nodes with no incoming links in a network grown randomly according to this model.

5. Out in a rural part of the county somewhere, n small towns have decided to get connected to a large Internet switching hub via a high-volume fiber-optic cable. The towns are labeled $T_1, T_2, \ldots, T_n$, and they are all arranged on a single long highway, so that town T_i is i miles from the switching hub (See Figure 13.6).

Now this cable is quite expensive; it costs k dollars per mile, resulting in an overall cost of kn dollars for the whole cable. The towns get together and discuss how to divide up the cost of the cable.

First, one of the towns way out at the far end of the highway makes the following proposal.

Proposal A. *Divide the cost evenly among all towns, so each pays k dollars.*

There's some sense in which Proposal A is fair, since it's as if each town is paying for the mile of cable directly leading up to it.

But one of the towns very close to the switching hub objects, pointing out that the faraway towns are actually benefiting from a large section of the cable, whereas the close-in towns only benefit from a short section of it. So they make the following counterproposal.

Proposal B. *Divide the cost so that the contribution of town T_i is proportional to i, its distance from the switching hub.*

One of the other towns very close to the switching hub points out that there's another way to do a nonproportional division that is also

natural. This is based on conceptually dividing the cable into n equal-length "edges" $e_1, \ldots, e_n$, where the first edge e_1 runs from the switching hub to T_1, and the i^{th} edge e_i ($i > 1$) runs from T_{i-1} to T_i. Now we observe that, while all the towns benefit from e_1, only the last town benefits from e_n. So they suggest

> **Proposal C.** *Divide the cost separately for each edge e_i. The cost of e_i should be shared equally by the towns $T_i, T_{i+1}, \ldots, T_n$, since these are the towns "downstream" of e_i.*

So now the towns have many different options; which is the fairest? To resolve this, they turn to the work of Lloyd Shapley, one of the most famous mathematical economists of the 20th century. He proposed what is now called the *Shapley value* as a general mechanism for sharing costs or benefits among several parties. It can be viewed as determining the "marginal contribution" of each party, *assuming the parties arrive in a random order.*

Here's how it would work concretely in our setting. Consider an ordering $\mathcal{O}$ of the towns, and suppose that the towns "arrive" in this order. The *marginal cost of town T_i in order $\mathcal{O}$* is determined as follows. If T_i is first in the order $\mathcal{O}$, then T_i pays ki, the cost of running the cable all the way from the switching hub to T_i. Otherwise, look at the set of towns that come before T_i in the order $\mathcal{O}$, and let T_j be the farthest among these towns from the switching hub. When T_i arrives, we assume the cable already reaches out to T_j but no farther. So if $j > i$ (T_j is farther out than T_i), then the marginal cost of T_i is 0, since the cable already runs past T_i on its way out to T_j. On the other hand, if $j < i$, then the marginal cost of T_i is $k(i - j)$: the cost of extending the cable from T_j out to T_i.

(For example, suppose $n = 3$ and the towns arrive in the order T_1, T_3, T_2. First T_1 pays k when it arrives. Then, when T_3 arrives, it only has to pay $2k$ to extend the cable from T_1. Finally, when T_2 arrives, it doesn't have to pay anything since the cable already runs past it out to T_3.)

Now, let X_i be the random variable equal to the marginal cost of town T_i when the order $\mathcal{O}$ is selected uniformly at random from all permutations of the towns. Under the rules of the Shapley value, the amount that T_i should contribute to the overall cost of the cable is the expected value of X_i.

The question is: Which of the three proposals above, if any, gives the same division of costs as the Shapley value cost-sharing mechanism? Give a proof for your answer.

6. One of the (many) hard problems that arises in genome mapping can be formulated in the following abstract way. We are given a set of n *markers* $\{\mu_1, \ldots, \mu_n\}$—these are positions on a chromosome that we are trying to map—and our goal is to output a linear ordering of these markers. The output should be consistent with a set of k *constraints*, each specified by a triple (μ_i, μ_j, μ_k), requiring that μ_j lie *between* μ_i and μ_k in the total ordering that we produce. (Note that this constraint does not specify which of μ_i or μ_k should come first in the ordering, only that μ_j should come between them.)

Now it is not always possible to satisfy all constraints simultaneously, so we wish to produce an ordering that satisfies as many as possible. Unfortunately, deciding whether there is an ordering that satisfies at least k' of the k constraints is an NP-complete problem (you don't have to prove this.)

Give a constant $\alpha > 0$ (independent of n) and an algorithm with the following property. If it is possible to satisfy k^* of the constraints, then the algorithm produces an ordering of markers satisfying at least αk^* of the constraints. Your algorithm may be randomized; in this case it should produce an ordering for which the *expected* number of satisfied constraints is at least αk^*.

7. In Section 13.4, we designed an approximation algorithm to within a factor of $7/8$ for the MAX 3-SAT Problem, where we assumed that each clause has terms associated with three different variables. In this problem, we will consider the analogous MAX SAT Problem: Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, and all the variables in a single clause are distinct, but otherwise we do not make any assumptions on the length of the clauses: There may be clauses that have a lot of variables, and others may have just a single variable.

(a) First consider the randomized approximation algorithm we used for MAX 3-SAT, setting each variable independently to *true* or *false* with probability 1/2 each. Show that the expected number of clauses satisfied by this random assignment is at least $k/2$, that is, at least half of the clauses are satisfied in expectation. Give an example to show that there are MAX SAT instances such that no assignment satisfies more than half of the clauses.

(b) If we have a clause that consists only of a single term (e.g., a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: We need to set the corresponding variable in the appropriate

way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction.

Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the randomized procedure above to improve the approximation factor from 1/2 to at least .6. That is, change the algorithm so that the expected number of clauses satisfied by the process is at least $.6k$.

(c) Give a randomized polynomial-time algorithm for the general MAX SAT Problem, so that the expected number of clauses satisfied by the algorithm is at least a .6 fraction of the maximum possible.

(Note that, by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a .6 fraction *of the maximum that can be satisfied by an optimal assignment*.)

8. Let $G = (V, E)$ be an undirected graph with n nodes and m edges. For a subset $X \subseteq V$, we use $G[X]$ to denote the subgraph *induced* on X—that is, the graph whose node set is X and whose edge set consists of all edges of G for which both ends lie in X.

We are given a natural number $k \leq n$ and are interested in finding a set of k nodes that induces a "dense" subgraph of G; we'll phrase this concretely as follows. Give a polynomial-time algorithm that produces, for a given natural number $k \leq n$, a set $X \subseteq V$ of k nodes with the property that the induced subgraph $G[X]$ has at least $\frac{mk(k-1)}{n(n-1)}$ edges.

You may give either (a) a deterministic algorithm, or (b) a randomized algorithm that has an expected running time that is polynomial, and that only outputs correct answers.

9. Suppose you're designing strategies for selling items on a popular auction Web site. Unlike other auction sites, this one uses a *one-pass auction*, in which each bid must be immediately (and irrevocably) accepted or refused. Specifically, the site works as follows.

- First a seller puts up an item for sale.
- Then buyers appear in sequence.
- When buyer i appears, he or she makes a bid $b_i > 0$.
- The seller must decide immediately whether to accept the bid or not. If the seller accepts the bid, the item is sold and all future buyers are

turned away. If the seller rejects the bid, buyer i departs and the bid is withdrawn; and only then does the seller see any future buyers.

Suppose an item is offered for sale, and there are n buyers, each with a distinct bid. Suppose further that the buyers appear in a random order, and that the seller knows the number n of buyers. We'd like to design a strategy whereby the seller has a reasonable chance of accepting the highest of the n bids. By a *strategy*, we mean a rule by which the seller decides whether to accept each presented bid, based only on the value of n and the sequence of bids seen so far.

For example, the seller could always accept the first bid presented. This results in the seller accepting the highest of the n bids with probability only $1/n$, since it requires the highest bid to be the first one presented.

Give a strategy under which the seller accepts the highest of the n bids with probability at least $1/4$, regardless of the value of n. (For simplicity, you may assume that n is an even number.) Prove that your strategy achieves this probabilistic guarantee.

10. Consider a very simple online auction system that works as follows. There are n *bidding agents*; agent i has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidding agents appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.)

What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

Example. Suppose $b_1 = 20$, $b_2 = 25$, and $b_3 = 10$, and the bidders arrive in the order $1, 3, 2$. Then b^* is updated for 1 and 2, but not for 3.

11. *Load balancing algorithms* for parallel or distributed systems seek to spread out collections of computing jobs over multiple machines. In this way, no one machine becomes a "hot spot." If some kind of central coordination is possible, then the load can potentially be spread out almost perfectly. But what if the jobs are coming from diverse sources that can't coordinate? As we saw in Section 13.10, one option is to assign them to machines at random and hope that this randomization will work to prevent imbalances. Clearly, this won't generally work as well as a perfectly centralized solution, but it can be quite effective. Here we try analyzing some variations and extensions on the simple load balancing heuristic we considered in Section 13.10.

Suppose you have k machines, and k jobs show up for processing. Each job is assigned to one of the k machines independently at random (with each machine equally likely).

(a) Let $N(k)$ be the expected number of machines that do not receive any jobs, so that $N(k)/k$ is the expected fraction of machines with nothing to do. What is the value of the limit $\lim_{k\to\infty} N(k)/k$? Give a proof of your answer.

(b) Suppose that machines are not able to queue up excess jobs, so if the random assignment of jobs to machines sends more than one job to a machine M, then M will do the first of the jobs it receives and reject the rest. Let $R(k)$ be the expected number of rejected jobs; so $R(k)/k$ is the expected fraction of rejected jobs. What is $\lim_{k\to\infty} R(k)/k$? Give a proof of your answer.

(c) Now assume that machines have slightly larger buffers; each machine M will do the first two jobs it receives, and reject any additional jobs. Let $R_2(k)$ denote the expected number of rejected jobs under this rule. What is $\lim_{k\to\infty} R_2(k)/k$? Give a proof of your answer.

12. Consider the following analogue of Karger's algorithm for finding minimum s-t cuts. We will contract edges iteratively using the following randomized procedure. In a given iteration, let s and t denote the possibly contracted nodes that contain the original nodes s and t, respectively. To make sure that s and t do not get contracted, at each iteration we delete any edges connecting s and t and select a random edge to contract among the remaining edges. Give an example to show that the probability that this method finds a minimum s-t cut can be exponentially small.

13. Consider a balls-and-bins experiment with $2n$ balls but only two bins. As usual, each ball independently selects one of the two bins, both bins equally likely. The expected number of balls in each bin is n. In this problem, we explore the question of how big their difference is likely to be. Let X_1 and X_2 denote the number of balls in the two bins, respectively. (X_1 and X_2 are random variables.) Prove that for any $\varepsilon > 0$ there is a constant $c > 0$ such that the probability $\Pr\left[X_1 - X_2 > c\sqrt{n}\right] \leq \varepsilon$.

14. Some people designing parallel physical simulations come to you with the following problem. They have a set P of k *basic processes* and want to assign each process to run on one of two machines, M_1 and M_2. They are then going to run a sequence of n *jobs*, $J_1, \ldots, J_n$. Each job J_i is represented by a set $P_i \subseteq P$ of exactly $2n$ basic processes which must be running (each on its assigned machine) while the job is processed. An assignment of basic processes to machines will be called *perfectly balanced* if, for

each job J_i, exactly n of the basic processes associated with J_i have been assigned to each of the two machines. An assignment of basic processes to machines will be called *nearly balanced* if, for each job J_i, no more than $\frac{4}{3}n$ of the basic processes associated with J_i have been assigned to the same machine.

(a) Show that for arbitrarily large values of n, there exist sequences of jobs $J_1, \ldots, J_n$ for which no perfectly balanced assignment exists.

(b) Suppose that $n \geq 200$. Give an algorithm that takes an arbitrary sequence of jobs $J_1, \ldots, J_n$ and produces a nearly balanced assignment of basic processes to machines. Your algorithm may be randomized, in which case its expected running time should be polynomial, and it should always produce the correct answer.

15. Suppose you are presented with a very large set S of real numbers, and you'd like to approximate the median of these numbers by sampling. You may assume all the numbers in S are distinct. Let $n = |S|$; we will say that a number x is an *ε-approximate median* of S if at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are less than x, and at least $(\frac{1}{2} - \varepsilon)n$ numbers in S are greater than x.

Consider an algorithm that works as follows. You select a subset $S' \subseteq S$ uniformly at random, compute the median of S', and return this as an approximate median of S. Show that there is an absolute constant c, independent of n, so that if you apply this algorithm with a sample S' of size c, then with probability at least .99, the number returned will be a (.05)-approximate median of S. (You may consider either the version of the algorithm that constructs S' by sampling with replacement, so that an element of S can be selected multiple times, or one without replacement.)

16. Consider the following (partially specified) method for transmitting a message securely between a sender and a receiver. The message will be represented as a string of bits. Let $\Sigma = \{0, 1\}$, and let Σ^* denote the set of all strings of 0 or more bits (e.g., $0, 00, 1110001 \in \Sigma^*$). The "empty string," with no bits, will be denoted $\lambda \in \Sigma^*$.

The sender and receiver share a secret function $f : \Sigma^* \times \Sigma \to \Sigma$. That is, f takes a word and a bit, and returns a bit. When the receiver gets a sequence of bits $\alpha \in \Sigma^*$, he or she runs the following method to decipher it.

Let $\alpha = \alpha_1\alpha_2 \cdots \alpha_n$, where n is the number of bits in α
The goal is to produce an n-bit deciphered message,
$\beta = \beta_1\beta_2 \cdots \beta_n$
Set $\beta_1 = f(\lambda, \alpha_1)$

```
For i = 2, 3, 4, ..., n
  Set β_i = f(β_1 β_2 ··· β_{i-1}, α_i)
  Endfor
Output β
```

One could view this is as a type of "stream cipher with feedback." One problem with this approach is that, if any bit α_i gets corrupted in transmission, it will corrupt the computed value of β_j for all $j \geq i$.

We consider the following problem. A sender S wants to transmit the same (plain-text) message β to each of k receivers $R_1, \ldots, R_k$. With each one, he shares a different secret function $f^{\langle i \rangle}$. Thus he sends a different encrypted message $\alpha^{\langle i \rangle}$ to each receiver, so that $\alpha^{\langle i \rangle}$ decrypts to β when the above algorithm is run with the function $f^{\langle i \rangle}$.

Unfortunately, the communication channels are very noisy, so each of the n bits in each of the k transmissions is *independently* corrupted (i.e., flipped to its complement) with probability $1/4$. Thus no single receiver on his or her own is likely to be able to decrypt the message correctly. Show, however, that if k is large enough as a function of n, then the k receivers can jointly reconstruct the plain-text message in the following way. They get together, and without revealing any of the $\alpha^{\langle i \rangle}$ or the $f^{\langle i \rangle}$, they interactively run an algorithm that will produce the correct β with probability at least $9/10$. (How large do you need k to be in your algorithm?)

17. Consider the following simple model of gambling in the presence of bad odds. At the beginning, your net profit is 0. You play for a sequence of n rounds; and in each round, your net profit increases by 1 with probability $1/3$, and decreases by 1 with probability $2/3$.

Show that the expected number of steps in which your net profit is positive can be upper-bounded by an absolute constant, independent of the value of n.

18. In this problem, we will consider the following simple randomized algorithm for the Vertex Cover Algorithm.

```
Start with S = ∅
While S is not a vertex cover,
   Select an edge e not covered by S
   Select one end of e at random (each end equally likely)
   Add the selected node to S
Endwhile
```

We will be interested in the expected cost of a vertex cover selected by this algorithm.

(a) Is this algorithm a c-approximation algorithm for the Minimum Weight Vertex Cover Problem for some constant c? Prove your answer.

(b) Is this algorithm a c-approximation algorithm for the Minimum Cardinality Vertex Cover Problem for some constant c? Prove your answer.

(Hint: For an edge, let p_e denote the probability that edge e is selected as an uncovered edge in this algorithm. Can you express the expected value of the solution in terms of these probabilities? To bound the value of an optimal solution in terms of the p_e probabilities, try to bound the sum of the probabilities for the edges incident to a given vertex v, namely, $\sum_{e \text{ incident to } v} p_e$.)

Notes and Further Reading

The use of randomization in algorithms is an active research area; the books by Motwani and Raghavan (1995) and Mitzenmacher and Upfal (2005) are devoted to this topic. As the contents of this chapter make clear, the types of probabilistic arguments used in the study of basic randomized algorithms often have a discrete, combinatorial flavor; one can get background in this style of probabilistic analysis from the book by Feller (1957).

The use of randomization for contention resolution is common in many systems and networking applications. Ethernet-style shared communication media, for example, use randomized *backoff* protocols to reduce the number of collisions among different senders; see the book by Bertsekas and Gallager (1992) for a discussion of this topic.

The randomized algorithm for the Minimum-Cut Problem described in the text is due to Karger, and after further optimizations due to Karger and Stein (1996), it has become one of the most efficient approaches to the minimum cut problem. A number of further extensions and applications of the algorithm appear in Karger's (1995) Ph.D. thesis.

The approximation algorithm for MAX 3-SAT is due to Johnson (1974), in a paper that contains a number of early approximation algorithms for NP-hard problems. The surprising punch line to that section—that every instance of 3-SAT has an assignment satisfying at least 7/8 of the clauses—is an example of the *probabilistic method*, whereby a combinatorial structure with a desired property is shown to exist simply by arguing that a random structure has the property with positive probability. This has grown into a highly refined

technique in the area of combinatorics; the book by Alon and Spencer (2000) covers a wide range of its applications.

Hashing is a topic that remains the subject of extensive study, in both theoretical and applied settings, and there are many variants of the basic method. The approach we focus on in Section 13.6 is due to Carter and Wegman (1979). The use of randomization for finding the closest pair of points in the plane was originally proposed by Rabin (1976), in an influential early paper that exposed the power of randomization in many algorithmic settings. The algorithm we describe in this chapter was developed by Golin et al. (1995). The technique used there to bound the number of dictionary operations, in which one sums the expected work over all stages of the random order, is sometimes referred to as *backwards analysis*; this was originally proposed by Chew (1985) for a related geometric problem, and a number of further applications of backwards analysis are described in the survey by Seidel (1993).

The performance guarantee for the LRU caching algorithm is due to Sleator and Tarjan (1985), and the bound for the Randomized Marking algorithm is due to Fiat, Karp, Luby, McGeoch, Sleator, and Young (1991). More generally, the paper by Sleator and Tarjan highlighted the notion of *online algorithms*, which must process input without knowledge of the future; caching is one of the fundamental applications that call for such algorithms. The book by Borodin and El-Yaniv (1998) is devoted to the topic of online algorithms and includes many further results on caching in particular.

There are many ways to formulate bounds of the type in Section 13.9, showing that a sum of 0-1-valued independent random variables is unlikely to deviate far from its mean. Results of this flavor are generally called *Chernoff bounds*, or *Chernoff-Hoeffding* bounds, after the work of Chernoff (1952) and Hoeffding (1963). The books by Alon and Spencer (1992), Motwani and Raghavan (1995), and Mitzenmacher and Upfal (2005) discuss these kinds of bounds in more detail and provide further applications.

The results for packet routing in terms of congestion and dilation are due to Leighton, Maggs, and Rao (1994). Routing is another area in which randomization can be effective at reducing contention and hot spots; the book by Leighton (1992) covers many further applications of this principle.

Notes on the Exercises Exercise 6 is based on a result of Benny Chor and Madhu Sudan; Exercise 9 is a version of the *Secretary Problem*, whose popularization is often credited to Martin Gardner.