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 \( A[i, t] \) 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 \( \Pr[A[i, t]] = p \).

For every event, there is also a **complementary event**, indicating that the event did not occur; here we have the complementary event \( \overline{A[i, t]} \) that \( P_i \) does not attempt to access the database in round \( t \), with probability

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

Our real concern is whether a process succeeds in accessing the database in a given round. Let \( S[i, t] \) 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 \( S[i, t] \) is equal to the intersection of the event \( A[i, t] \) with all the complementary events \( \overline{A[j, t]} \), for \( j \neq i \):

\[
S[i, t] = A[i, t] \cap \bigcap_{j \neq i} \overline{A[j, t]}.
\]

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

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

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 $\Pr [S[i, t]] = \frac{1}{n} \left( 1 - \frac{1}{n} \right)^{n-1}$. 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 $1/(en) \leq \Pr [S[i, t]] \leq 1/(2n)$, and hence $\Pr [S[i, t]]$ 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 $\bar{S}[i, r]$ 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:

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

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

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

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

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

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

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 $\Pr[\mathcal{F}_t]$ 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:

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

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

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

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 $t = \lfloor en\rfloor \cdot (c \ln n)$, then we have $\Pr[\mathcal{F}[i, t]] \leq n^{-c}$ for each $i$, which is what we want. Thus, in particular, taking $t = 2\lfloor en\rfloor \ln n$ gives us

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

and so we have shown the following.

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 $\Pr[\mathcal{F}[i, t]]$ that was larger than $n^{-1}$, and hence a corresponding upper bound for the overall failure probability $\Pr[\mathcal{F}_t]$ 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 $\Pr[\mathcal{F}_t]$ 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 $\Pr[\mathcal{F}_t]$ 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.

**Theorem 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
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 $v_1$ and $v_2$, then return the cut $(S(v_1), S(v_2))$

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 $S(e_{uv}) = S(u) \cup S(v)$

Apply the Contraction Algorithm recursively to $G'$

---

**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.

---

**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 $E_j$ for the event that an edge of $F$ is not contracted in iteration $j$, then we have shown $Pr[E_1] \geq 1 - 2/n$ and $Pr[E_{j+1} | E_1 \cap E_2 \cdots \cap E_j] \geq 1 - 2/(n - j)$. We are interested in lower-bounding the quantity $Pr[E_1 \cap E_2 \cdots \cap E_{n-2}]$, and we can check by unwinding the formula for conditional probability that this is equal to

$$Pr[E_1] \cdot Pr[E_2 | E_1] \cdots Pr[E_{j+1} | E_1 \cap E_2 \cdots \cap E_j] \cdots Pr[E_{n-2} | E_1 \cap E_2 \cdots \cap E_{n-3}] \geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n - 1}\right) \cdots \left(1 - \frac{2}{n - j}\right) \cdots \left(1 - \frac{2}{3}\right)$$

$$= \left(\frac{n - 2}{n}\right) \left(\frac{n - 3}{n - 1}\right) \left(\frac{n - 4}{n - 2}\right) \cdots \left(\frac{2}{4}\right) \left(\frac{1}{3}\right)$$

$$= \frac{2}{n(n - 1)} = \binom{n}{2}^{-1}.$$
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
\[
\left(1 - \frac{1}{\binom{n}{2}}\right) \leq \frac{1}{e}.
\]
And it's easy to drive the failure probability below \(1/e\) with further repetitions: if we run the algorithm \(\binom{n}{2}\) in \(n\) 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.
**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 $E_i$ denote the event that $C_i$ is returned by the Contraction Algorithm, and let $E = \bigcup_{i=1}^{r} E_i$ denote the event that the algorithm returns any global min-cut.

Then, although (13.5) simply asserts that $\Pr[E] \geq 1/(\binom{n}{2})$, its proof actually shows that for each $i$, we have $\Pr[E_i] \geq 1/(\binom{n}{2})$. Now each pair of events $E_i$ and $E_j$ 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

$$\Pr[E] = \Pr[\bigcup_{i=1}^{r} E_i] = \sum_{i=1}^{r} \Pr[E_i] \geq r / \binom{n}{2}.$$  

But clearly $\Pr[E] \leq 1$, 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 $\infty$ if the sum diverges. Thus, for example, if $X$ takes each of the values in \{1, 2, \ldots, n\} with probability $1/n$, then \[ E[X] = 1(1/n) + 2(1/n) + \cdots + n(1/n) = \left(\frac{n+1}{2}\right)/n = (n+1)/2. \]

**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.

**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**.

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
13.3 Random Variables and Their Expectations

...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 $E[X_i]$, and then determine $E[X] = \sum E[X_i]$. 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

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

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

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

Thus we have shown the following.

Trying to compute $E[X]$ directly from the definition $\sum_{j=0}^{52} j \cdot \Pr[X = j]$ 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

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

and so we have

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

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.

**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) \[ E[X_j] = \frac{n}{n-j}. \]

**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 $E[X_j] = \frac{n}{n-j}$. \[ \]

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

**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 \( E \) of positive probability. Then we define the *conditional expectation* of \( X \), given \( E \), to be the expected value of \( X \) computed only over the part of the sample space corresponding to \( E \). We denote this quantity by \( E[X \mid E] \). This simply involves replacing the probabilities \( \Pr[X = j] \) in the definition of the expectation with conditional probabilities:

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

**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 $C_1, \ldots, C_n$, 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 $p$ 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 $E[Z_i]$ 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 $E[Z_i] = \frac{7}{8}$.

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

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.
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 \textit{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.

\textbf{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 \( \frac{7}{8} \) 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 \( \sum_{j=0}^k j p_j \); and by the previous analysis, this is equal to \( \frac{7}{8} k \). We are interested in the quantity \( p = \sum_{j=27k/8} p_j \). How can we use the lower bound on the expected value to give a lower bound on this quantity?
We start by writing
\[
\frac{7}{8}k = \sum_{j=0}^{k} j p_j = \sum_{j<7k/8} j p_j + \sum_{j \geq 7k/8} j p_j.
\]
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 \(k p_j\). We also observe that \(\sum_{j<7k/8} p_j = 1 - p\), and so
\[
\frac{7}{8}k \leq \sum_{j<7k/8} k' p_j + \sum_{j \geq 7k/8} k p_j = k'(1 - p) + kp \leq k' + kp,
\]
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
\[
p \geq \frac{k' - k}{k} \geq \frac{1}{8k}.
\]
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 \(8k\).

13.5 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