Chapter 19 Randomized Algorithms

Trees with snow on branches, "Half Dome, Apple Orchard, Yosemite," 1933. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Contents

An important feature of any single-person computer game is that it shouldn't be boring. That is, the user should have a new and distinct experience each time he or she plays the game. One way to achieve such a goal, of course, is to use randomization. That is, the execution of the algorithm shouldn't depend only on its inputs, but also on the use of a random-number generator to produce random numbers that are used to guide the execution of the algorithm. Thus, the structure of such algorithms depend on the outcomes of random events. In this context, the running time or correctness of an algorithm then becomes a random variable, which is averaged over the values of the random numbers the algorithm uses.

Because they are used extensively in computer games, cryptography, and computer simulations, methods that generate random numbers are built into most modern computers. Some methods, called ***pseudo-random-number generators***, generate random-like numbers deterministically, starting with an initial number called a ***seed***. Other methods use hardware devices to extract "true" random numbers from nature. In any case, we assume that our computer has access to numbers that are sufficiently random for our analysis, and we explore in this chapter some of the algorithms that can be designed using these tools.

A ***randomized algorithm*** is an algorithm whose behavior depends, in part, on the outcomes of random choices or the values of random bits. The main advantage of using randomization in algorithm design is that the results are often simple and efficient. In addition, there are some problems that need randomization for them to work effectively. For instance, consider the problem common in computer games involving playing cards—that of randomly shuffling a deck of cards so that all possible orderings are equally likely. This problem is surprisingly subtle, and there are even published stories of people who have been able to defeat online poker systems by exploiting the fact that those systems were not using a good card shuffling algorithm. So one of the problems that we address in this chapter on randomized algorithms is how to efficiently shuffle a deck of virtual cards. For example, an algorithm that doesn't generate all possible permutations with equal probability is to take an approach based on the riffle shuffle used by humans to shuffle cards. Thus, the shuffling methods we discuss in this chapter are quite different than this approach, but are just as fast.

We study a number of other interesting randomized algorithms in this chapter, as well, including finding prime numbers, which is important in cryptography, the coupon collector problem, which has applications to biodiversity studies, and methods for building search structures known as skip lists, using randomization. We assume throughout this chapter that the reader is familiar with the principles of basic probability covered in Section 1.2.4.

This is not the only place where we discuss randomized algorithms, however. We also have discussions of randimized algorithms in the context of hash tables in Chapter 6, random construction of binary search trees in Section 3.4, the randomized quick-sort algorithm in Section 8.2, the quick-select algorithm in Chapter 9, and a page-caching algorithm in Section 20.4.

19.1 Generating Random Permutations

As mentioned above, randomly shuffling cards is an important part of any computer gaming system that deals virtual cards. The problem of computing random permutations has many more applications than just computerized card games, however, including a host of other randomized algorithms, often as the very first step. In this section, we explore two random permutation algorithms, both of which run in linear time.

The input to the random permutation problem is a list, $X = (x_1, x_2, \ldots, x_n)$, of n elements, which could stand for playing cards or any other objects we want to randomly permute. The output is a reordering of the elements of X, done in a way so that all permutations of X are equally likely.

Both of the algorithms we discuss make use of a function, $\mathsf{random}(k)$, which returns an integer in the range $[0, k-1]$ chosen uniformly and independently at random. This function is based on the use of some source of random bits. So if k is a power of 2, then we simply take $\log k$ random bits from this source and interpret them as a number in the range from 0 to $k-1$, inclusive. Every number in this range is equally likely. If, on the other hand, k is not a power of 2, then we let K be the smallest power of 2 greater than k and we take $\log K$ random bits from this source. If interpreting these bits as a number gives us an integer in the range $[0, k-1]$, then we take this as our output to the $\mathsf{random}(k)$ method. Otherwise, if we get a number that is k or larger, then we discard these bits and try again. Thus, if k is not a power of 2, then the running time of the $\mathsf{random}(k)$ method is itself a random variable with expected value $O(1)$. If k is a power of 2, however, then this method always runs in $O(1)$ time.

We give our first random permutation method in Algorithm 19.2. This algorithm simply chooses a random number for each element in X and sorts the elements using these values as keys. (See Figure 19.1.) If all the keys are distinct, then the resulting permutation is generated uniformly, as we show below.

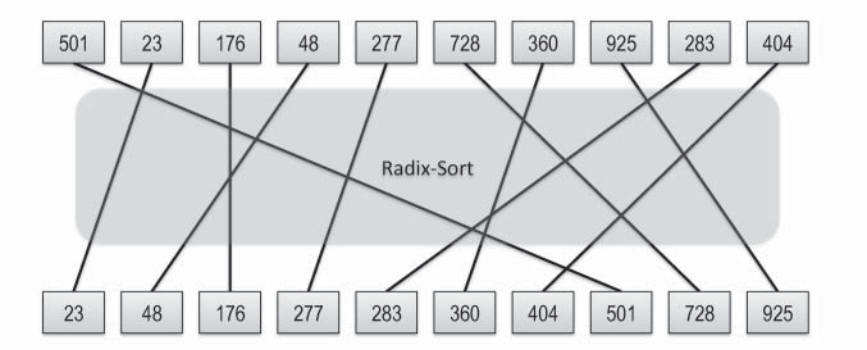

Figure 19.1: A random permutation algorithm based on sorting random numbers.

Algorithm randomSort(X):
Input: A list, X, of n elements
Output: A permutation of X so that all permutations are equally likely
Let K be the smallest power of 2 greater than or equal to n^3
for each element, x_i, in X **do**
Choose a random value, r_i, in the range $[0, K-1]$ and associate it with x_i
Sort X using the r_i values as keys via radix-sort
if all the r_i values are distinct **then**
return X according to this sorted order
else
Call randomSort(X)

Algorithm 19.2: A sorting-based algorithm for generating a random permutation.

Analyzing the Sorting-Based Random Permutation Algorithm

To see that every permutation is equally likely to be output by the randomSort method, note that each element, x_i, in X has an equal probability, $1/n$, of having its r_i value be the smallest. Thus, each element in X has equal probability of $1/n$ of being the first element in the permutation. Note, in addition, that because the algorithm will only terminate when all the elements are distinct, then there is exactly one element with smallest r_i value. Moreover, once we have removed this value, each remaining value, x_i, has equal probability of $1/(n-1)$ of holding the minimum r_i value. Thus, the probability that any element from this group will be chosen second is $1/(n-1)$. Following this reasoning to its logical conclusion, this approach implies that the permutation that is output had a probability of

$$\left(\frac{1}{n}\right) \cdot \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \cdot \left(\frac{1}{1}\right) = \frac{1}{n!}$$

of being chosen. That is, all permutations are equally likely.

In addition, because we use radix sort to sort X by r_i values, and K is $O(n^3)$, each call to the randomSort method runs in $O(n)$ time, not counting any recursive calls that are made because a duplicate r_i value is found. In any call to this method, we choose n of the r_i values, so the probability that any one of these is the same as any other is at most $n/n^3 = 1/n^2$. Therefore, since there are n values chosen, the probability that there is a duplicate found is at most $n/n^2 = 1/n$. That is, this algorithm runs in $O(n)$ time, without any retries, with probability $1 - 1/n$. In general, we say that an event involving n items holds with ***high probability*** if it occurs with probability at least $1 - 1/n$. So the sorting-based permutation algorithms runs in $O(n)$ time with high probability.

The Fisher-Yates Shuffling Algorithm

There is another shuffling algorithm, which reduces all the uncertainty in its running time to calls of the random(k) method. Thus, if we had a way of implementing this method so that it always ran in constant time, then this second algorithm would be "almost deterministic," in that it would always succeed and it would always run in $O(n)$ time. This algorithm, which is known as the ***Fisher-Yates algorithm***, assumes that the input list is given as an array; the details are given in Algorithm 19.3. (See Figure 19.4.)

Algorithm FisherYates(X):
Input: An array, X, of n elements, indexed from position 0 to $n-1$
Output: A permutation of X so that all permutations are equally likely

for $k = n-1$ **downto** 1 **do**
  Let $j \leftarrow$ random$(k+1)$ // j is a random integer in $[0,k]$
  Swap $X[k]$ and $X[j]$ // This may "swap" $X[k]$ with itself, if $j = k$
return X

Algorithm 19.3: The Fisher-Yates algorithm for generating a random permutation.

This algorithm considers the items in the array one at time from the end and swaps each element with an element in the array from that point to the beginning. Notice that each element has an equal probability, of $1/n$, of being chosen as the last element in the array X (including the element that starts out in that position). Likewise, for the elements that remain in the first $n-1$ positions, each of them has an equal probability, of $1/(n-1)$, of being the last element in that range. Following this reasoning to its logical conclusion implies that the permutation that is actually output by the Fisher-Yates algorithm had a probability of

$$\left(\frac{1}{n}\right) \cdot \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \cdot \left(\frac{1}{1}\right) = \frac{1}{n!}$$

of being the one output. That is, all permutations are equally likely. In addition, if the random(k) method runs in $O(1)$ time, then this algorithm runs in $O(n)$ time. (We revisit this issue in Section 19.5.3.)

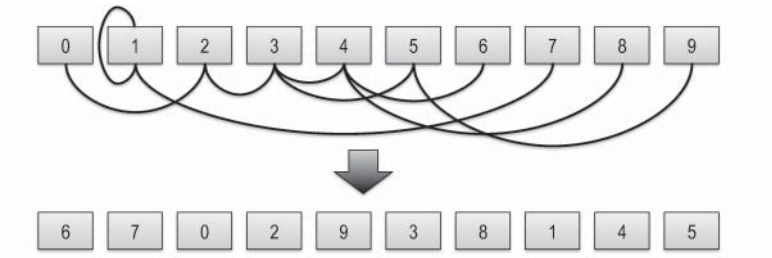

Figure 19.4: The Fisher-Yates random permutation algorithm. The arcs represent swaps, which start from 9 and go down to 1. Here, position 1 swaps with itself.

19.2 Stable Marriages and Coupon Collecting

A common question biologists often ask with respect to a given ecosystem is to determine the number of species that occupy that ecosystem. In order to get a handle on such biodiversity questions, biologists have developed various tools for sampling species. For instance, one tool that is used for this purpose is random sampling, where an ecosystem is divided into a grid, like a checkerboard, and then a careful census is prepared for a random subset of the squares in this grid. A natural issue that arises in this application, then, is to determine the right number of squares to sample in order to have a good chance of finding at least one individual from every species in the ecosystem.

One way to get a handle on this estimate is to model it as a ***coupon collector problem***, which abstracts the essence of this estimation. We imagine that there is a set, C, of n coupons and we are interested in collecting at least one of every coupon in C. We can go a ticket window once a day and request a coupon, and a clerk will choose one of the n coupons at random and give it to us. We cannot expect the clerk to remember which coupons he has given us in the past, however, so, each time he gives us a coupon, it is chosen uniformly and independently at random from among the n coupons in C. The coupon collector problem, then, is to determine the number of times we need to go to the ticket window before we have collected all n coupons. So, for instance, in the application to biodiversity estimation, each time a plant or animal is detected in a random sample is like a trip to the ticket window in the coupon collector problem.

19.2.1 Analyzing the Coupon Collector Problem

Let X be a random variable representing the number of times that we need to visit the ticket window before we get all n coupons. We can write X as

$$X = X_1 + X_2 + \cdots + X_n,$$

where X_i is the number of trips we have to make to the ticket window in order to go from having $i-1$ distinct coupons to having i distinct coupons. So, for example, $X_1 = 1$, since we are guaranteed to get a distinct coupon in our first trip to the ticket window, and X_2 is the number of additional trips we must make to get our second distinct coupon. After we have gotten $i-1$ distinct coupons, our chance of getting a new one in any trip to the window is

$$p_i = \frac{n-(i-1)}{n},$$

since there are n coupons, but only $n-(i-1)$ that we don't already have at this point in time. This implies that each X_i a ***geometric random variable*** with parameter, p_i. That is, if we imagine that we have a biased coin that only comes up heads with

probability p_i, then X_i is the number of times we have to flip this coin until we get it to come up heads. The expected value, $E[X_i]$, of X_i is therefore $1/p_i$.

By the linearity of expectation,

$$\begin{aligned} E[X] &= E[X_1] + E[X_2] + \cdots + E[X_n] \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1} \\ &= n \sum_{i=1}^{n} \frac{1}{i} \\ &= n\, H_n, \end{aligned}$$

where H_n is the nth harmonic number, which, as we have observed elsewhere, can be approximated as $\ln n \leq H_n \leq \ln n + 1$. In other words, the expected number of times that we need to visit the ticket window in order to get at least one instance of each of n coupons is nH_n, which is approximately $n \ln n$.

In some cases, such as in the biodiversity application discussed above, we would like to bound the probability that we need to make significantly more than $n \ln n$ trips to the ticket window to get all n coupons. Such a bound is known as a ***tail estimate***, since it involves bounding the end or "tail" of a probability distribution. Fortunately, in the case of the coupon collector problem, coming up with such a tail bound is not that difficult.

Let us now consider that the coupons are numbered $1, 2, \ldots, n$ and let $Y_{i,t}$ be a random variable indicating the event that coupon number i was not collected even after we have made t trips to the ticket window. Thus,

$$\begin{aligned} \Pr(Y_{i,t} = 1) &= \left(1 - \frac{1}{n}\right)^t \\ &\leq e^{-t/n}, \end{aligned}$$

since $1 - x \leq e^{-x}$, for $0 < x < 1$ (see Theorem A.4). We can then bound the probability that X is more than $T = cn \ln n$, for some constant $c \geq 2$, as follows:

$$\begin{aligned} \Pr(X > T) &\leq \Pr\left(\sum_{i=1}^{n} Y_{i,T} \geq 1\right) \\ &\leq n \cdot \Pr(Y_{1,T} = 1) \\ &\leq n \cdot e^{-T/n} \\ &= n \cdot e^{-c \ln n} \\ &= n \cdot n^{-c} \\ &= n^{-c+1}. \end{aligned}$$

Thus, with high probability, we will collect all n coupons after making $cn \ln n$ trips to the ticket window, for $c \geq 2$.

19.2.2 The Stable Marriage Problem

Imagine a village consisting of n men and n women, all of whom are single, heterosexual, and interested in getting married. Every man has a list of the women ordered by his preferences, and, likewise, every woman has a list of the men ordered by her preferences. The ***stable marriage problem*** is to match up the men and women in a way that is ***stable***. Such a matching is stable if there is no unmatched man-woman pair, (x, y), such that x and y would prefer to be married to each other than to their spouses. That is, it would be unstable if x preferred y over his wife and y preferred x over her husband. (See Figure 19.5.)

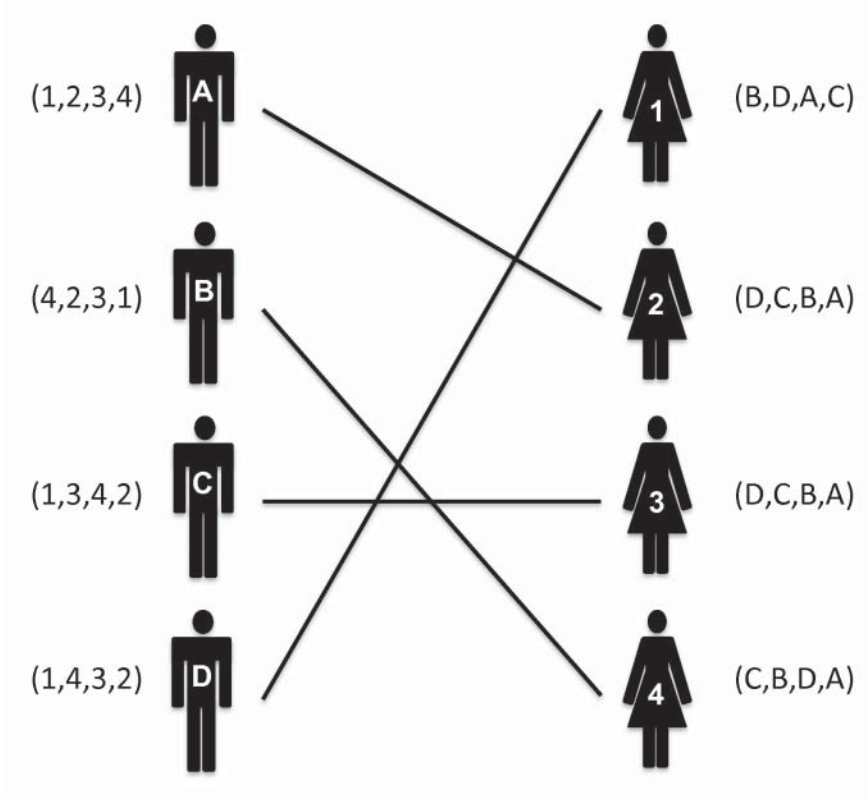

Figure 19.5: An instance of the stable marriage problem. Each man and woman is listed with his or her preference list, and the matching shown is stable. Note that even though female 2 is married to her last choice, there is no man who prefers her over his current wife.

In real life, this problem arises in the annual placement of residents in hospitals. Residents rank order the hospitals that they would like to work in, and hospitals rank the set of available residents for each of their available open slots. Then a stable "marriage" is computed between residents and available slots in hospitals.

The Proposal Algorithm

There is a simple algorithm for solving the stable marriage problem, which involves men making proposals to women in a series of rounds.

A round begins with an unmatched man making a proposal to the female highest-ranked on his list. If she is unmatched, then she accepts his proposal and the round ends. If, on the other hand, she is matched, then she accepts his proposal only if she ranks him higher than her current partner. In the case when the woman receiving the proposal is already matched, then whichever man she rejects repeats the computation for this round, making a proposal to the next woman on his list (his highest-ranked woman that he has not previously proposed to). Thus, a round continues in this way, consisting of a series of proposals and it ends when a previously unmatched woman accepts a proposal. This proposal algorithm continues performing such rounds until all the men and women are matched. (See Algorithm 19.6.)

Algorithm StableMarriage(X, Y):

Input: A set, X, of n men and their prioritized lists of women in Y, and a set, Y, of n women and their prioritized lists of men in X
Output: A stable marriage for X and Y

```
for each man, x, in X do
    Let y be x's highest-ranked woman
    Have x propose to y
    while y is matched to some man, z, such that z ≠ x do
        if y prefers x over z then
            Match x and y
            Unmatch z
            Let x ← z
        Let y be x's highest-ranked woman he has not proposed to yet
        Have x propose to y
    Match x and y
```

Algorithm 19.6: The proposal algorithm for the stable marriage problem.

To see that the matching resulting from this proposal algorithm is stable, suppose there is an unstable pair, (x, y), that are not matched by the algorithm, that is, both x and y prefer each other to the partners they end up with by following the proposal algorithm. Note that at the time x made his last proposal, to his current partner, w, he had made a proposal to every woman that he ranked higher than w. But this means that he would have made a proposal to y, which she would have accepted, because she ranks x higher than the man she ended up with. Thus, such a pair, (x, y), cannot exist.

A Worst-Case Amortized Analysis

The worst-case running time of this algorithm is $O(n^2)$. To see this fact, note that at the beginning of the algorithm, the total length of the lists of all n men is $O(n^2)$, and at each step of the algorithm, some man is using up a proposal to a woman on his list to whom he will never again propose. Thus, if we charge each entry in the list of preferences by the men 1 cyber-dollar for the work we do in having a man make a proposal, then we can pay for all the proposals using $O(n^2)$ cyber-dollars.

An Average-Case Analysis Based on Coupon Collecting

We can perform an average-case analysis of Algorithm 19.6, which has a much better performance than the worst-case bound. For the sake of this analysis, suppose the preference list for every man is an independent random permutation of the list of women. Then, each time it is a man's turn to make a proposal, he is making that proposal to a random woman he has not proposed to before.

In fact, we can simplify this analysis even further to consider a version of the algorithm where each time it is a man's turn to make a proposal, he makes his proposal to a random woman without consideration of his previous proposals. Such an algorithm is ***memoryless***, in the sense that each proposal a man makes is independent of any proposal he made earlier. Still, note that in this memoryless proposal algorithm if a man makes a proposal to a woman he has previously proposed to, she will reject his proposal, since at that point in time she will be matched to someone she prefers more than him. Thus, a bound on the running time of this memoryless algorithm gives us a bound on the original algorithm with randomly ordered preference lists, since the original version makes no more proposals than the memoryless version.

The key observation to analyze the memoryless algorithm is to focus on the women and realize that each round in this algorithm consists of a sequence of proposals to independently chosen random women until a proposal is made to an unmatched woman. That is, the memoryless algorithm is an instance of the coupon collector problem where the names of the women are the coupons. Thus, by the analysis of the coupon collector problem, the expected running time of the memoryless stable marriage algorithm is $O(n \log n)$. Therefore, we have the following theorem.

Theorem 19.1: *The expected number of proposals made in the proposal algorithm, for a set of n men and n women, is at most nH_n, assuming the preference list of each man is an independent random permutation of the list of women.*

That is, the average-case running time of the proposal algorithm for the stable marriage problem is $O(n \log n)$.

19.3 Minimum Cuts

A ***cut***, C, of a connected graph, G, is a subset of the edges of G whose removal disconnects G. That is, after removing all the edges of C, we can partition the vertices of G into two subsets, A, and B such that there are no edges between a vertex in A and a vertex in B. (See Figure 19.7.) A ***minimum cut*** of G is a cut of smallest size among all cuts of G.

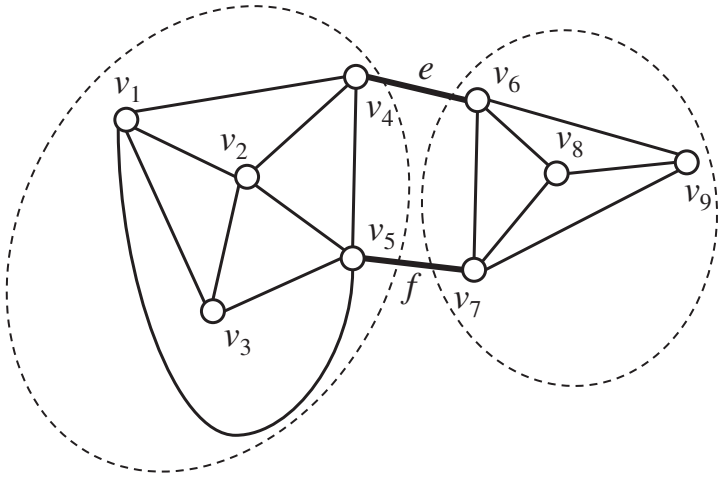

Figure 19.7: Example of a cut of a graph. The set of edges $C = \{e, f\}$, drawn with thick lines, is a cut, since removing e and f partitions the graph into two connected components. This cut is of minimum size.

In several applications, it is important to determine the size of a smallest cut of a graph. For example, in a communications network, the failures of the edges of a cut prevents the communication between the nodes on the two sides of a cut. Thus, the size of a minimum cut and the number of such cuts give an idea of the vulnerability of the network to edge failures. Small cuts are also important for the automatic classification of web content. Namely, consider a collection of web pages and model them as a graph, where vertices correspond to pages and edges to links between pages. The size of a minimum cut provides a measure of how much groups of pages have related content. Also, we can use minimum cuts to recursively partition the collection into clusters of related documents. Similar considerations can be made for minimum cuts in a social network.

As a starting point for computing minimum cuts, note that, for each vertex, v, the edges incident on v form a cut, whose size is the degree of v, $\deg(v)$, since removing all these edges separates v from the rest of the graph. Thus, the minimum degree of the vertices of a graph is an upper bound on the size of its minimum cut.

Also, recalling the notion of a biconnected graph, defined in Section 13.5, we have that a minimum cut of a biconnected graph has size at least 2.

In Section 16.1, we present a related definition of a cut in a flow network with respect to given pair of vertices, s, and t. The partition induced by such a cut has vertex s on one side and vertex t on the other size. As explored in Exercise C-19.9, one can compute a minimum cut of a graph by repeated applications of a maximum flow algorithm to a flow network derived from G.

19.3.1 Contracting Edges

In the rest of this section, we show how to compute a minimum cut by means of a randomized algorithm that is simple to implement. This algorithm repeatedly performs contraction operations on the graph. Let G be a graph with n vertices, where we allow G to have parallel edges. We denote with (v, w) any edge with endpoints v and w. The ***contraction*** of an edge e of G with endpoints u and v consists of the following steps that yield a new graph with $n - 1$ vertices, denoted G/e:

1. Remove edge e and any other edge between its endpoints, u and v.
2. Create a new vertex, w.
3. For every edge, f, incident on u, detach f from u and attach it to w. Formally speaking, let z be the other endpoint of f. Change the endpoints of f to be z and w.
4. For every edge, f, incident on v, detach f from v and attach it to w.

A series of contraction operations is shown in Figure 19.8. Note that a contraction may create parallel edges. Specifically, if vertices u and v have a common neighbor, z, the edges connecting z to u and v become parallel edges connecting z and w.

A contraction operation for an edge (u, v) has the important property of preserving the set of cuts that do not include any edge (u, v). This property is formally expressed by the following lemma.

Lemma 19.2: *Let G be a graph that may have parallel edges and let G/e be the graph resulting from G after contracting an edge, e, with endpoints u and v. A set, C, of edges is a cut of G/e if and only if C is a cut of G that does not contain any edge with endpoints u and v.*

Lemma 19.2 implies that the set of cuts of G/e is the same as the set of cuts of G that exclude any edge (u, v).

By Lemma 19.2, if we perform a series of contraction operations, we obtain a graph whose cuts are a subset of the cuts of the original graph. In particular, if we contract $n - 2$ edges, where n is the number of vertices of the graph, we obtain a graph with two vertices connected by parallel edges that form one of the cuts of the original graph. (See Figure 19.8.)

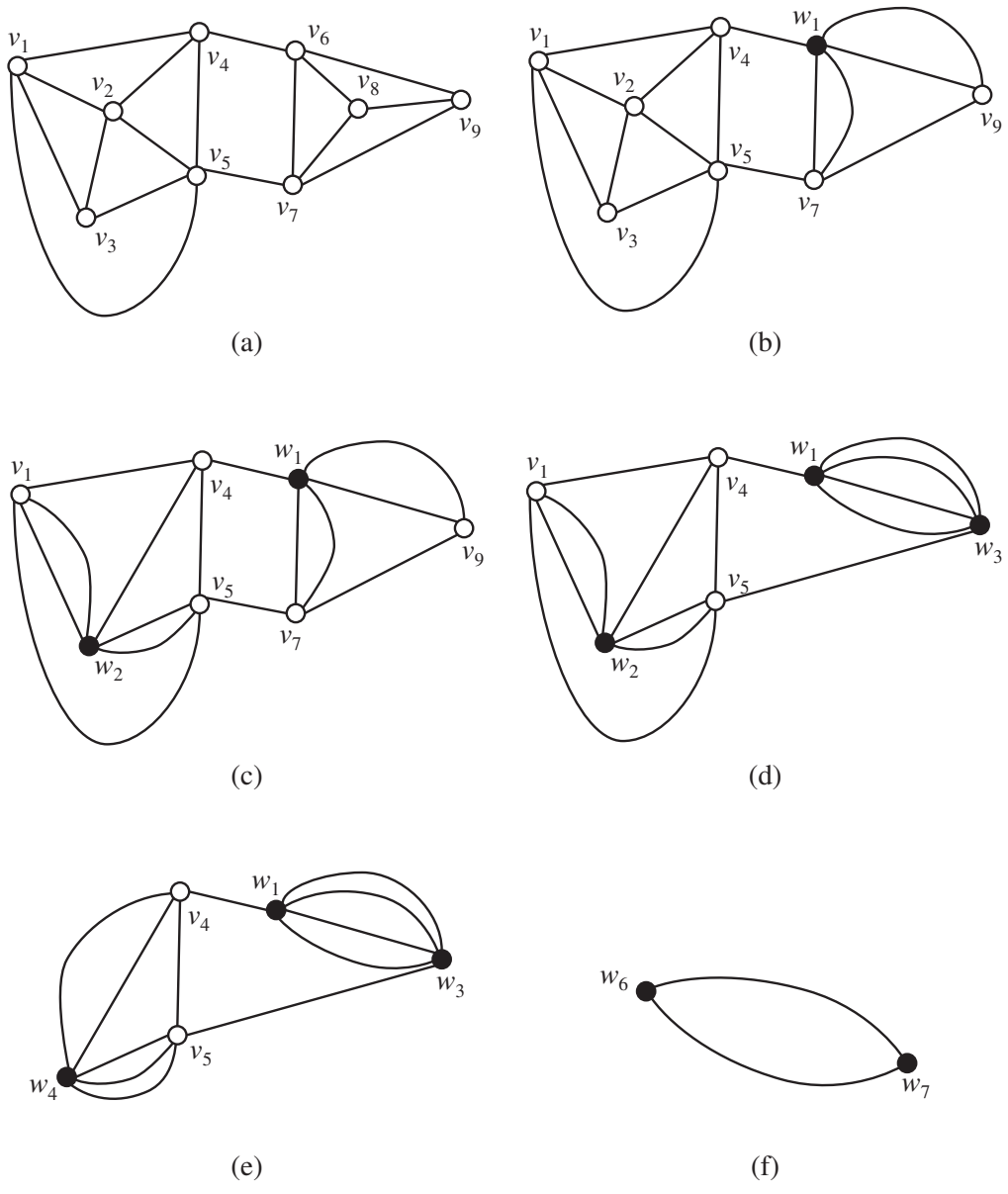

Figure 19.8: Sequence of contraction operations on a graph. (a–b) Contraction of (v_6, v_8). (b–c) Contraction of (v_2, v_3). (c–d) Contraction of (v_7, v_9). (d–e) Contraction of (v_1, w_2). (f) Final graph obtained after contracting (v_4, v_5), yielding vertex w_5, (w_4, w_5), yielding vertex w_6, and (w_1, w_2), yielding vertex w_7. The final graph has the same minimum cut as the original graph since each contraction preserves the edges of this cut.

19.3.2 Computing a Minimum Cut

Karger's algorithm for computing a minimum cut is very simple. It performs a sequence of edge contractions, selecting each time a random edge. The algorithm stops when the graph has two vertices and returns the set of edges between them. This method is summarized in Algorithm 19.9. Admittedly, the algorithm may not return a minimum cut, since, for each such cut, the algorithm may have contracted one of its edges. Nevertheless, Karger's algorithm succeeds with high probability.

Algorithm ContractGraph(G):
Input: An undirected graph, G, with n vertices
Output: A cut of G that has minimum size with probability at least $\frac{2}{n(n-1)}$
while G has more than 2 edges **do**
pick a random edge, e, of G
contract edge e
$G \leftarrow G/e$
return the edges of G

Algorithm 19.9: Single iteration of Karger's randomized algorithm for finding a minimum cut of a graph.

Let us evaluate the success probability of Algorithm 19.9, that is, the probability that the algorithm returns a minimum cut. Let G be a graph with n vertices and m edges, and let C be a given minimum cut of G. We will evaluate the probability that the algorithm returns the cut C. Since G may have other minimum cuts, this probability is a lower bound on the success probability of the algorithm.

Let G_i be the graph obtained after i contractions performed by the algorithm and let m_i be the number of edges of G_i. Assume that G_{i-1} contains all the edges of C. The probability that G_i also contains all the edges of C is equal to

$$1 - \frac{k}{m_{i-1}},$$

since we contract any given edge of C with probability $1/m_{i-1}$ and C has k edges. Thus, the probability, P, that the algorithm returns cut C is given by

$$P = \prod_{i=0,\ldots,n-3} \left(1 - \frac{k}{m_i}\right).$$

Since k is the size of the minimum cut of each graph G_i, we have that each vertex of G_i has degree at least k. Thus, we obtain the following lower bound on m_i, the number of edges of G_i:

$$m_i \geq \frac{k(n-i)}{2}, \quad \text{for } i = 0, 1, \ldots, n-3\,.$$

We can use the above inequality to derive a lower bound on P, as follows:

$$\begin{aligned}
P &= \prod_{i=0}^{n-3}\left(1-\frac{k}{m_i}\right) && (19.1)\\
&\geq \prod_{i=0}^{n-3}\left(1-\frac{2k}{k(n-i)}\right) && (19.2)\\
&= \prod_{i=0}^{n-3}\left(\frac{n-i-2}{n-i}\right) && (19.3)\\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\left(\frac{n-5}{n-3}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) && (19.4)\\
&= \frac{2}{n(n-1)} && (19.5)\\
&= \frac{1}{\binom{n}{2}} && (19.6)
\end{aligned}$$

The above analysis shows that probability that Algorithm 19.9 returns cut C is $\Omega\left(\frac{1}{n^2}\right)$, a decreasing function that tends to 0 as n grows. However, we can boost the probability by running the algorithm multiple times. In particular, if we run the algorithm for $t\binom{n}{2}$ rounds, where t is a positive integer, we have that at least one round returns cut C with probability

$$P(t) = 1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{t\binom{n}{2}}.$$

By a well-known property (Theorem A.4) of the mathematical constant e, the base of the natural logarithm, $\ln$, we obtain

$$P(t) \geq 1 - \frac{1}{e^t}.$$

In particular, if we choose $t = c\ln n$, where c is a constant, then we get a success probability, $1 - 1/n^c$, to obtain the cut C.

We turn now to the analysis of the running time of the algorithm. A contraction operation can be executed in $O(n)$ time. Thus, Algorithm 19.9 (ContractGraph) takes $O(n^2)$ time. We summarize our findings with the following theorem.

Theorem 19.3: *Let G be a graph with n vertices. For any positive integer constant c, we can compute a minimum cut of G with a randomized algorithm that runs in time $O(n^4 \log n)$ and has success probability $1 - 1/n^c$.*

Proof: The algorithm consists of executing Algorithm 19.9 (ContractGraph) $c\binom{n}{2}\ln n$ times and returning the smallest of the cuts obtained. ■

19.3.3 A Faster Algorithm

We now show how to improve the running time of the above contraction approach for finding minimum cuts. By inspecting Equation 19.4, we note that the probability of avoiding cut C is at least $\frac{1}{2}$ when the algorithm has contracted the graph down to about $\frac{n}{\sqrt{2}}$ vertices. This observation and the use of recursion leads to Algorithm 19.10, due to Karger and Stein.

Algorithm RecursiveContractGraph(G, n):
  Input: An undirected graph, G, with n vertices
  Output: A cut of G that has minimum size with probability at least $\frac{1}{\log n}$
if G has at most 6 vertices **then**
  return ContractGraph$(G, 2)$
else
  $G_1 \leftarrow$ ContractGraph $\left(G, \frac{n}{\sqrt{2}}\right)$
  $C_1 \leftarrow$ RecursiveContractGraph $\left(G_1, \frac{n}{\sqrt{2}}\right)$
  $G_2 \leftarrow$ ContractGraph $\left(G, \frac{n}{\sqrt{2}}\right)$
  $C_2 \leftarrow$ RecursiveContractGraph $\left(G_2, \frac{n}{\sqrt{2}}\right)$
  if $|C_1| \leq |C_2|$ **then**
    return C_1
  else
    return C_2

Algorithm 19.10: Single iteration of the randomized algorithm by Karger and Stein for finding a minimum cut of a graph. The algorithms invokes ContractGraph(G, r), a variation of method ContractGraph(G) (Algorithm 19.9) that performs $n - r$ contractions and returns the resulting graph, which has r vertices.

Algorithm 19.10 (RecursiveContractGraph) makes two recursive calls. Also, after each call, the size of the input graph is a constant fraction, $\frac{1}{\sqrt{2}}$, of the size before the call. Thus, the execution of the algorithm can be modeled by a binary recursion tree with depth $2 \log n$.

The number of contractions performed in a call, excluding those within its recursive calls, is proportional to the number of vertices of the graph. Since each contraction takes linear time in the number of vertices, the work done within a call, excluding its recursive calls, is quadratic in the number of vertices. Hence, the total

amount of work performed at level i of the recursion tree is proportional to

$$2^i \left(\frac{n}{\sqrt{2}^i} \right)^2 = n^2.$$

We conclude that RecursiveContractGraph(G, n) runs in $O(n^2 \log n)$ time.

Let us analyze the probability that RecursiveContractGraph (Algorithm 19.10) returns a given minimum cut, C. Referring to the recursion tree, we say that a node is safe if its associated graph contains cut C. The recursive call at a safe node succeeds in returning C if one of the following events occurs:

- graph G_1 obtained from ContractGraph contains cut C and the recursive call returns C;
- graph G_2 obtained from ContractGraph contains cut C and the recursive call returns C.

Clearly, each of the above events has the same probability. However, the events are not mutually exclusive.

Recall that the height of a node of a tree is the length of the longest path from the node to a leaf. Define $P(i)$ as the probability that a safe node at height i of the recursion tree has a safe leaf in its subtree. Thus, $P(2 \log n)$ denotes the probability that algorithm RecursiveContractGraph succeeds in returning C. We know that each of graphs G_1 and G_2 returned by ContractGraph contains C with probability $\frac{1}{2}$. Thus, we can write the following recurrence relation:

$$P(i+1) \geq \frac{1}{2}P(i) + \frac{1}{2}P(i) - \left(\frac{1}{2}P(i) \right)^2 = P(i) - \frac{1}{4}P(i)^2.$$

The base case of the recurrence is for a leaf, which has height 0 and is associated with a graph with at most 6 vertices:

$$P(0) \geq \frac{1}{\binom{6}{2}} = \frac{1}{15}.$$

Analyzing the recurrence relation, one can show that $P(2 \log n)$, the success probability of the algorithm, is $\Omega\left(\frac{1}{\log n}\right)$. (See Exercise C-19.10.) Thus, we obtain the following theorem.

Theorem 19.4: *Let G be a graph with n vertices. For any positive integer constant c, we can compute a minimum cut of G with a randomized algorithm that runs in time $O(n^2 \log^3 n)$ and has success probability $1 - 1/n^c$.*

Proof: The algorithm consists of executing $c \log^2 n$ times Algorithm 19.10 (RecursiveContractGraph) and returning the smallest of the cuts obtained. ■

19.4 Finding Prime Numbers

An integer p is ***prime*** if $p \geq 2$ and its only divisors are the trivial divisors 1 and p. An integer greater than 2 that is not prime is ***composite***. So, for example, 5, 11, and 101 are prime, whereas 25 and 713 ($= 23 \cdot 31$) are composite. In this section, we discuss randomized ways of testing for primality and for then using these methods to find prime numbers. Such computations are useful, for example, in cryptography, which is discussed in Chapter 24.

In order to lay the groundwork for finding primes, a few words about the ***modulo operator*** (mod) are in order. Recall that $a \bmod n$ is the remainder of a when divided by n. That is,

$$r = a \bmod n$$

means that $r = a - \lfloor a/n \rfloor n$. In other words, there is some integer q, such that $a = qn + r$. Note, in addition, that $a \bmod n$ is always an integer in the set $\{0, 1, 2, \ldots, n-1\}$.

It is sometimes convenient to talk about ***congruence*** modulo n. If

$$a \bmod n = b \bmod n,$$

we say that a is ***congruent*** to b modulo n, which we call the ***modulus***, and we write

$$a \equiv b \pmod{n}.$$

Therefore, if $a \equiv b \bmod n$, then $a - b = kn$ for some integer k.

Having laid this groundwork, we are now ready for the first theorem of this chapter, which is known as ***Fermat's Little Theorem***.

Theorem 19.5 (*Fermat's Little Theorem*): *Let p be a prime, and let x be an integer such that $x \bmod p \neq 0$ and $0 < x < p$. Then*

$$x^{p-1} \equiv 1 \pmod{p}.$$

Proof: We know that, for $0 < x < p$, the set $\{1, 2, \ldots, p-1\}$ and the set $\{x \cdot 1, x \cdot 2, \ldots, x \cdot (p-1)\}$ contain exactly the same elements, when we do all arithmetic modulo p (see Exercise C-19.8). So when we multiply the elements of the sets together, we get the same value, namely,

$$1 \cdot 2 \cdots (p-1) = (p-1)!.$$

In other words,

$$(x \cdot 1) \cdot (x \cdot 2) \cdots (x \cdot (p-1)) \equiv (p-1)! \pmod{p}.$$

If we factor out the x terms, we get

$$x^{p-1}(p-1)! \equiv (p-1)! \pmod{p}.$$

Thus, because p is prime, we can cancel the term $(p-1)!$ from both sides, yielding $x^{p-1} \equiv 1 \bmod p$, which is the desired result. ∎

19.4.1 Primality Testing

Prime numbers play an important role in computations involving numbers, such as cryptographic computations, as we noted above. But how do we test whether a number is prime, particularly if it is large?

Testing all possible divisors of a number takes exponential time, so we need an alternative to the method many of us used to test for primality on grade-school Math assignments. Alternatively, Fermat's Little Theorem (Theorem 19.5) seems to suggest an efficient solution. Perhaps we can somehow use the equation

$$a^{p-1} \equiv 1 \bmod p$$

to form a test for p. That is, let us pick a number a, and raise it to the power $p-1$. If the result is ***not*** 1, then the number p is definitely not prime. Otherwise, there's a chance it is. Would repeating this test for various values of a prove that p is prime? Unfortunately, the answer is "no." There is a class of numbers, called ***Carmichael numbers***, that have the property that $a^{n-1} \equiv 1 \bmod n$ for all $1 \leq a \leq n-1$, but n is composite. The existence of these numbers ruins such a simple way to test for primality. Example Carmichael numbers are 561 and 1105.

Independent Repetitions: A Template for Primality Testing

While the above simple test won't work, there is a related approach that will work, by making more sophisticated use of Fermat's Little Theorem. Such a probabilistic test of primality is based on the following general approach. Let n be an odd integer that we want to test for primality, and let witness(x, n) be a Boolean function of a random variable x and n with the following properties:

1. If n is prime, then witness(x, n) is always false. So if witness(x, n) is true, then n is definitely composite.
2. If n is composite, then witness(x, n) is false with probability $q < 1$.

The function witness is said to be a ***compositeness witness function*** with error probability q, for q bounds the probability that witness will incorrectly identify a composite number as possibly prime. By repeatedly computing witness(x, n) for independent random values of the parameter x, we can determine whether n is prime with an arbitrarily small error probability. The probability that witness(x, n) would incorrectly return "false" for k independent random x's, when n is a composite number, is q^k. A template for a probabilistic primality testing algorithm based on this observation is shown in Algorithm 19.11. This algorithm assumes we have a compositeness witness function, witness, that satisfies the two conditions above. In order to turn this template into a full-blown algorithm, we need only specify the details of how to pick random numbers, x, and compute witness(x, n), the composite witness function.

Algorithm RandomizedPrimalityTesting(n, k):
Input: Odd integer $n \geq 2$ and confidence parameter k
Output: An indication of whether n is composite (which is always correct) or prime (which is incorrect with error probability 2^{-k})

```
// This method assumes we have a compositeness witness function witness(x, n)
with error probability q < 1.
t ← ⌈k/ log_2(1/q)⌉
for i ← 1 to t do
    x ← random()
    if witness(x, n) then
        return "composite"
return "prime"
```

Algorithm 19.11: A template for a probabilistic primality testing algorithm based on a compositeness witness function witness(x, n). We assume that the method random$()$ picks a value at random from the domain of the variable x.

If the method RandomizedPrimalityTesting$(n, k,$ witness$)$ returns "composite," we know with certainty that n is composite. However, if the method returns "prime," the probability that n is actually composite is no more than 2^{-k}. Indeed, suppose that n is composite but the method returns "prime." We have that the witness function witness(x, n) has evaluated to true for t random values of x. The probability of this event is q^t. From the relation between the confidence parameter k, the number of iterations t, and the error probability q of the witness function established by the first statement of the method, we have that $q^t \leq 2^{-k}$.

Given the parameter k, which is known as a ***confidence parameter***, we can make $t = \lceil k/\log_2(1/q) \rceil$ independent repetitions of our test to get force our probability of failure to be at most 2^{-k}. For instance, taking $k = 30$, forces this failure probability to be lower than the probability that the average person will, in their lifetime, be hit by lightning. This approach of repeating independent trials of a randomized algorithm to force a failure probability down below a probability determined by a specified confidence parameter is a common pattern in the design of randomized algorithms.

The Rabin-Miller Primality Testing Algorithm

We now describe the ***Rabin-Miller algorithm*** for primality testing. It is based on Fermat's Little Theorem (Theorem 19.5) and on the following lemma.

Lemma 19.6: *Let p be a prime number greater than 2. If x is an element of Z_p such that*

$$x^2 \equiv 1 \pmod{p},$$

then either $x \equiv 1 \pmod{p}$ *or* $x \equiv -1 \pmod{p}$.

A ***nontrivial square root of the unity*** in Z_n is an integer $1 < x < n-1$ such that $x^2 \equiv 1 \pmod{n}$. Lemma 19.6 states that if n is prime, there are no nontrivial square roots of the unity in Z_n. The Rabin-Miller algorithm uses this fact to define the witness(x, n) function as shown below:

Algorithm witness(x, n):
  Write $n-1$ as $2^k m$, where m is odd.
  Compute $y \leftarrow x^m \bmod n$
  if $y \equiv 1 \pmod{n}$ **then**
    return false // n is probably prime
  for $i \leftarrow 1$ **to** $k-1$ **do**
    if $y \equiv -1 \pmod{n}$ **then**
      return false // n is probably prime
    $y \leftarrow y^2 \bmod n$
  return true // n is definitely composite

As we explore in an exercise (C-19.12), if the Rabin-Miller composite witness function returns **true**, then n is definitely composite. The error probability of the cases when Rabin-Miller composite witness algorithm returns **false** is provided by the following lemma, stated without proof.

Lemma 19.7: *Let n be a composite number. There are at most $(n-1)/4$ positive values of x in Z_n such that the Rabin-Miller compositeness witness function* witness(x, n) *returns true.*

We conclude as follows:

Theorem 19.8: *Given an odd positive integer n and a parameter $k > 0$, the Rabin-Miller algorithm determines whether n is prime, with error probability 2^{-k}, by performing $O(k \log n)$ arithmetic operations.*

Finding Prime Numbers

A primality testing algorithm can be used to select a random prime in a given range, or with a prespecified number of bits. We exploit the following result from number theory, stated without proof.

Theorem 19.9: *The number, $\pi(n)$, of primes that are less than or equal to n is $\Theta(n/\ln n)$. In fact, if $n \geq 17$, then $n/\ln n < \pi(n) < 1.26n/\ln n$.*

A consequence of Theorem 19.9 is that a random integer n is prime with probability $1/\ln n$. Thus, to find a prime with a given number b of bits, we can again use the pattern of repeated independent trials. In particular, note that if we generate a random b-bit odd number, n, then it has $\lceil \log n \rceil \geq \lceil \ln n \rceil$ bits. So n is prime with probability at least $1/b$; hence, repeating a primality test for kb such numbers gives us the following.

Theorem 19.10: *Given an integer b and a confidence parameter k, a random prime with b bits can be generated by performing $O(kb)$ primality tests, with probability at least $1 - 2^{-k}$.*

Las Vegas and Monte Carlo Algorithms

Note that in Theorem 19.10 the confidence parameter is bounding a probability that the algorithm takes longer than the stated time to run, whereas the confidence parameter in the Rabin-Miller algorithm is bounding the probability that the algorithm produces a wrong answer. These two scenarios arise so often in the context of randomized algorithms that they have names. A randomized algorithm that always succeeds in producing a correct output, but whose running time depends on random events is known as a ***Las Vegas algorithm***. A randomized algorithm that always has a deterministic running time, but whose output may be incorrect, with some probability, is known as a ***Monte Carlo algorithm***. We summarize the distinction between these two categories of randomized algorithms in Table 19.12.

	Running Time	**Correctness**
Las Vegas Algorithm	probabilistic	certain
Monte Carlo Algorithm	certain	probabilistic

Table 19.12: The difference between Las Vegas and Monte Carlo algorithms.

Note that if we have a deterministic way to test for correctness, we can always turn a Monte Carlo algorithm into a Las Vegas algorithm, as we explore in Exercise C-19.13. But without such a testing algorithm, we have no easy way of turning a Monte Carlo algorithm into a Las Vegas algorithm. Nevertheless, we do have an easy way to turn any Las Vegas algorithm into a Monte Carlo algorithm, simply by running the Las Vegas algorithm for an amount of time determined by its confidence parameter and either outputting the correct answer, if the algorithm has terminated in that time, or outputting "sorry" if it has not. Moreover, if we use a Monte Carlo algorithm as a subroutine in what would otherwise be a Las Vegas algorithm, then the resulting algorithm is Monte Carlo, using this same argument. Such is the case if we combine Theorems 19.8 and 19.10.

In addition, we say that a Monte Carlo algorithm that outputs yes-no answers has a ***one-sided error*** if, as in the Rabin-Miller algorithm, one of its outputs, "yes" or "no," is always correct. Otherwise, it has a ***two-sided error***.

19.5 Chernoff Bounds

Quite often in the analysis of randomized algorithms we are interested in proving a bound on a running time with high probability. A useful collection of tools that are frequently used to perform such analyses are the ***Chernoff bounds*** we discuss in this section. For instance, we use one in Section 6.3.3 to analyze the performance of a hash table that uses linear probing to resolve collisions.

19.5.1 Markov's Inequality

Any discussion of Chernoff bounds begins with a foundational fact of probability known as ***Markov's inequality***, which can be stated as follows.

Theorem 19.11: *Let X be a random variable such that $X \geq 0$. Then, for all $a > 0$,*

$$\Pr(X \geq a) \leq \frac{E[X]}{a}.$$

Proof: Let Y be a 0-1 indicator random variable that is 1 if $X \geq a$. Then, since $X \geq 0$,

$$Y \leq \frac{X}{a}.$$

Taking expectations of both sides, we get

$$E[Y] \leq \frac{E[X]}{a},$$

by the linearity of expectation. The theorem follows, then, by observing that

$$E[Y] = \Pr(Y = 1) = \Pr(X \geq a).$$

■

An example application of Markov's inequality is to note that at most 10% of the U.S. population can have more than 10 times the average net worth of any American.

The bounds that can be derived directly from using Markov's inequality are not always the tightest, but they are usually simple. Moreover, another nice feature of Markov's inequality is that we don't have to know anything more about X than the fact that it is nonnegative and has expected value $E[X]$. In fact, it is sufficient for us simply to have an upper bound on $E[X]$ in order to apply Markov's inequality. For example, using Markov's inequality and Theorem 19.1, we can conclude that there is at most a 10% chance that the proposal algorithm for the stable marriage problem will make more than $10nH_n$ proposals, assuming the preference list of each man is an independent random permutation of the list of women.

19.5.2 Sums of Indicator Random Variables

Let $X = X_1 + X_2 + \cdots + X_n$ be the sum of n independent 0-1 indicator random variables, such that $X_i = 1$ with probability p_i. Using the language of probability theory, X is a random variable from the ***binomial distribution***. Intuitively, X is equal to the number of heads one gets by flipping n coins such that the ith coin comes up heads with probability p_i. Define the ***mean***, or expected value of X, as

$$\mu = E[X] = \sum_{i=1}^{n} p_i.$$

Then we have the following ***Chernoff bound***.

Theorem 19.12: *For $\delta > 0$,*

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

Proof: Applying Markov's inequality, after raising both sides of the probabilistic inequality to the power e^{λ}, for $\lambda > 0$, we get

$$\begin{aligned}
\Pr(X > (1+\delta)\mu) &= \Pr(e^{\lambda X} > e^{\lambda(1+\delta)\mu}) \\
&\leq \frac{E[e^{\lambda X}]}{e^{\lambda(1+\delta)\mu}} \\
&= \frac{\prod_{i=1}^{n} E[e^{\lambda X_i}]}{e^{\lambda(1+\delta)\mu}},
\end{aligned}$$

because the X_i's are mutually independent. Note that random variable $e^{\lambda X_i}$ takes on the value e^{λ} with probability p_i, and the value 1 with probability $1 - p_i$. Thus, $E[e^{\lambda X_i}] = e^{\lambda} + 1 - p_i$. So we have

$$\begin{aligned}
\Pr(X > (1+\delta)\mu) &\leq \frac{\prod_{i=1}^{n}(e^{\lambda} + 1 - p_i)}{e^{\lambda(1+\delta)\mu}} \\
&= \frac{\prod_{i=1}^{n}(1 + p_i(e^{\lambda} - 1))}{e^{\lambda(1+\delta)\mu}}.
\end{aligned}$$

We can use the inequality $1 + x < e^{x}$ for $x = p_i(e^{\lambda} - 1)$, to show

$$\begin{aligned}
\Pr(X > (1+\delta)\mu) &\leq \frac{\prod_{i=1}^{n} e^{p_i(e^{\lambda}-1)}}{e^{\lambda(1+\delta)\mu}} \\
&= \frac{e^{(\sum_{i=1}^{n} p_i(e^{\lambda}-1))}}{e^{\lambda(1+\delta)\mu}} \\
&= \frac{e^{(e^{\lambda}-1)\mu}}{e^{\lambda(1+\delta)\mu}}.
\end{aligned}$$

Taking $\lambda = \ln(1+\delta)$, which is positive for $\delta > 0$, completes the theorem. ■

We have the following Chernoff bound as well.

Theorem 19.13: *For* $0 < \delta < 1$,

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

Proof: The proof is similar to that of Theorem 19.12, except that we rewrite the probability as $\Pr(e^{-\lambda X} > e^{-\lambda(1-\delta)\mu})$ and let $\lambda = \ln(1/(1-\delta))$. ■

Application: Processor Load Balancing

Suppose we have a set of n processors and a set of n jobs for them to perform, but no good way of assigning jobs to processors. So we just assign jobs to processors at random. What is a good high-probability upper bound on the number of jobs assigned to any processor?

We can answer this question using a Chernoff bound. Let X be a random variable representing the number of jobs assigned to processor 1. Then X can be written as

$$X = X_1 + X_2 + \cdots X_n,$$

where X_i is the 0-1 indicator random variable that job i is assigned to processor 1. Thus, $\Pr(X_i = 1) = 1/n$ and $\mu = E[X] = 1$. Since the X_i's are clearly independent, we can apply the Chernoff bound from Theorem 19.12 to get, for any integer $m > 1$,

$$\Pr(X > m) \leq \frac{e^{m-1}}{m^m}.$$

After a bit of algebra, one can show, for

$$m \geq \frac{3\ln n}{\ln\ln n},$$

and $n \geq 2^8$, that

$$\frac{e^{m-1}}{m^m} \leq \frac{1}{n^2}.$$

Thus, the probability that processor 1 has more than m jobs assigned to it by this random assignment is at most $1/n^2$. Therefore, the probability that any processor is assigned more than m jobs is at most $n/n^2 = 1/n$. In other words, the number of processors assigned to any processor is $O(\log n/\log\log n)$ with high probability.

19.5.3 Sums of Geometric Random Variables

Let Y be a random variable that is the sum of n independent geometric random variables with parameter p. That is,

$$Y = Y_1 + Y_2 + \cdots + Y_n,$$

where each Y_i is a number of times to flip a coin, which comes up heads with probability p, until getting an outcome of heads. Using the language of probability theory, Y is a random variable from the ***negative binomial distribution***. In this case,

$$E[Y_i] = 1/p, \quad \text{for each } i = 1, 2, \ldots, n;$$

hence, $E[Y] = \alpha n$, where $\alpha = 1/p$.

The Chernoff bound for Y that we derive in this section concerns the characterization of the probability that Y exceeds its mean, which is $\alpha n = n/p$. In particular, we are interested in the probability,

$$\Pr(Y > (\alpha + t)n),$$

where $t > 0$. That is, using the coin flipping metaphor, we are interested in the probability that we have to flip a coin more than $(\alpha + t)n$ times to get n heads.

Relating the Binomial and Negative Binomial Distributions

In order to bound this probability, let us consider another random variable, X, which is the sum of $(\alpha + t)n$ independent indicator random variables, each of which is 1 with probability p. That is, X is the number of heads we get from flipping $(\alpha + t)n$ coins, each of which comes up heads with probability p. This random variable, which comes from the binomial distribution, can help us bound the above probability for Y, which comes from the negative binomial distribution, because

$$\Pr(Y > (\alpha + t)n) \quad = \quad \Pr(X < n).$$

In other words, the probability that it takes more than $(\alpha + t)n$ coin tosses to get n heads is equal to the probability that we get fewer than n heads in exactly $(\alpha + t)n$ coin tosses.

Note that X is the same kind of random variable we discussed in the previous subsection. Thus, we can use Theorem 19.13 to analyze it. In this case, note that if we let $\mu = E[X]$, then

$$\begin{aligned} \mu &= p(\alpha + t)n \\ &= p((1/p) + t)n \\ &= (1 + tp)n. \end{aligned}$$

We use the above fact in the following application of Theorem 19.13.

Lemma 19.14: *For $t > 0$,*

$$\Pr(Y > (\alpha + t)n) \leq e^{-tpn}(1 + tp)^n.$$

Proof: We already observed that $\Pr(Y > (\alpha + t)n) = \Pr(X < n)$, so, in order to apply Theorem 19.13, we need to bound the probability that X is less than $(1 - \delta)\mu$, where this quantity is equal to n. That is, we have

$$(1 - \delta)\mu = (1 - \delta)(1 + tp)n = n.$$

Hence,

$$1 - \delta = \frac{1}{1 + tp} \quad \text{and} \quad -\delta = \frac{-tp}{1 + tp}.$$

Thus, by Theorem 19.13,

$$\begin{aligned} \Pr(X < (1 - \delta)\mu) &< \left[\frac{e^{-tp/(1+tp)}}{[1/(1 + tp)]^{1/(1+tp)}}\right]^{(1+tp)n} \\ &= e^{-tpn}(1 + tp)^n. \end{aligned}$$

■

This lemma allows us to then derive the following Chernoff bound, in the spirit of Theorem 19.12.

Theorem 19.15: *Let $Y = Y_1 + Y_2 + \cdots + Y_n$ be the sum of n independent geometric random variables with parameter p. Then, for $\alpha = 1/p$ and $t \geq \alpha$,*

$$\Pr(Y > (\alpha + t)n) \leq e^{-tpn/5}.$$

Proof: By Lemma 19.14,

$$\Pr(Y > (\alpha + t)n) \leq e^{-tpn}\,(1 + tp)^n\,.$$

Unfortunately, if we use the inequality, $1 + x \leq e^x$, with $x = tp$, to bound the righthand term in the above equation, then we get a useless result. So, instead, we use a better approximation, namely, that, if $x \geq 1$, then $1 + x < e^{x/(1+1/4)}$. Thus, for $t \geq \alpha$,

$$\begin{aligned} \Pr(Y > (\alpha + t)n) &\leq e^{-tpn} \cdot e^{4tpn/5} \\ &= e^{-tpn/5}. \end{aligned}$$

■

A More Realistic Analysis of Fisher-Yates Random Shuffling

In the analysis given earlier for the Fisher-Yates shuffling algorithm to generate a random permutation, we assumed that the random(k) method, which returns a random integer in the range $[0, k-1]$, always runs in $O(1)$ time. If we are using a random-number generator based on the use of unbiased random bits, however, this is not a realistic assumption.

In a system based on using random bits, we would most naturally implement the random(k) method by generating $\lceil \log k \rceil$ random bits, interpreting these bits as an unsigned integer, and then repeating this operation until we got an integer in the range $[0, k-1]$. Thus, counting each iteration in such an algorithm as a "step," we could conservatively say that the random(k) method runs in 1 step with probability $1/2$, 2 steps with probability $1/2^2$, and, in general, in i steps with probability $1/2^i$. That is, its running time is a geometric random variable with parameter

$$p = \frac{1}{2}.$$

Under this more realistic assumption, the running time of the Fisher-Yates random permutation algorithm is proportional to the sum,

$$Y = Y_1 + Y_2 + \cdots Y_{n-1},$$

where each Y_i is the number of steps performed in the ith call to the random method. In other words, Y is the sum of $n-1$ independent geometric random variables with parameter $p = 1/2$, since the running times of the calls to the random(k) method are independent. Thus, focusing on the steps used in calls to this method, the expected running time of the Fisher-Yates algorithm is

$$E[Y] = 2(n-1).$$

Given this information, we can use the Chernoff bound for a sum of independent geometric random variables given in Theorem 19.15, with $\alpha = 2$, to bound the probability that the Fisher-Yates algorithm takes more than $4n$ steps as follows:

$$\Pr(Y > 4n) \leq e^{-n/5}.$$

Therefore, under this more realistic analysis, the Fisher-Yates algorithm runs in $O(n)$ time with high probability.

19.6 Skip Lists

An interesting data structure for efficiently realizing the ordered set of items is the ***skip list***. This data structure makes random choices in arranging the items in such a way that search and update times are logarithmic ***on average***. A ***skip list*** S for an ordered dictionary, D, of key-value pairs consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. Each list S_i stores a subset of the items of D sorted by a nondecreasing key plus items with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in D and $+\infty$ is larger than every possible key that can be inserted in D. In addition, the lists in S satisfy the following (see Figure 19.13):

- S_0 contains every item in D (plus special items with keys $-\infty$ and $+\infty$).
- For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in list S_{i-1}.
- S_h contains only $-\infty$ and $+\infty$.

It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to h as the ***height*** of skip list S.

Intuitively, the lists are set up so that S_{i+1} contains essentially every other item in S_i. The items in S_{i+1} are chosen at random from the items in S_i by picking each item from S_i to also be in S_{i+1} with probability $1/2$. That is, in essence, we "flip a coin" for each item in S_i and place that item in S_{i+1} if the coin comes up "heads." Thus, we expect S_1 to have about $n/2$ items, S_2 to have about $n/4$ items, and, in general, S_i to have about $n/2^i$ items. In other words, we expect the height h of S to be about $\log n$. We view a skip list as a two-dimensional collection of nodes arranged horizontally into ***levels*** and vertically into ***towers***. Each level is a list S_i and each tower contains nodes storing the same item across consecutive lists.

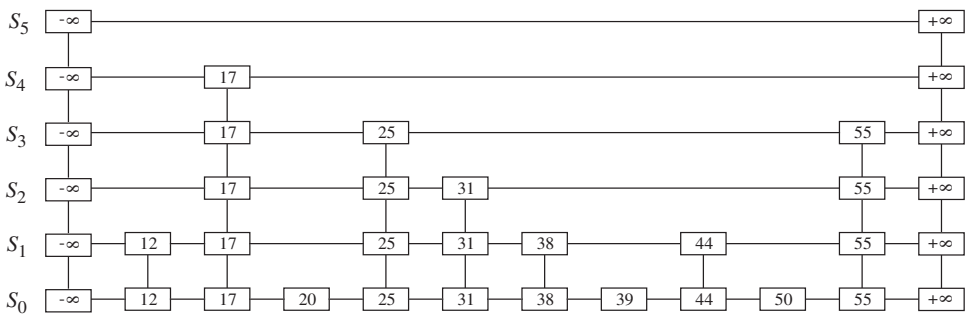

Figure 19.13: Example of a skip list.

19.6.1 Searching

The nodes in a skip list can be traversed using the following operations:

after(p): Return the node following p on the same level.
before(p): Return the node preceding p on the same level.
below(p): Return the node below p in the same tower.
above(p): Return the node above p in the same tower.

We assume that the above operations return **null** if the node requested does not exist. Searching in a skip list is based on the method shown in Algorithm 19.14.

Algorithm SkipSearch(k):
Input: A search key k
Output: Node in S whose item has the largest key less than or equal to k

Let p be the topmost, left node of S (which should have at least 2 levels).
while below(p) $\neq$ **null** **do**
  $p \leftarrow$ below(p) // drop down
  while key(after(p)) $\leq k$ **do**
    Let $p \leftarrow$ after(p) // scan forward
return p.

Algorithm 19.14: Algorithm for searching in a skip list S.

Note that we begin the SkipSearch method by setting p to the topmost, left node in the skip list S, and repeating the following steps (see Figure 19.15):

1. If S.below(p) is null, then the search terminates—we are ***at the bottom*** and have located the largest item in S with key less than or equal to the search key k. Otherwise, we ***drop down*** to the next lower level in the present tower.
2. Starting at node p, we move p forward until it is at the right-most node on the present level such that key(p) $\leq k$. We call this the ***scan forward*** step. Note that such a node always exists, since each level contains the special keys $+\infty$ and $-\infty$. In fact, after we perform the scan forward for this level, p may remain where it started. In any case, we then repeat the previous step.

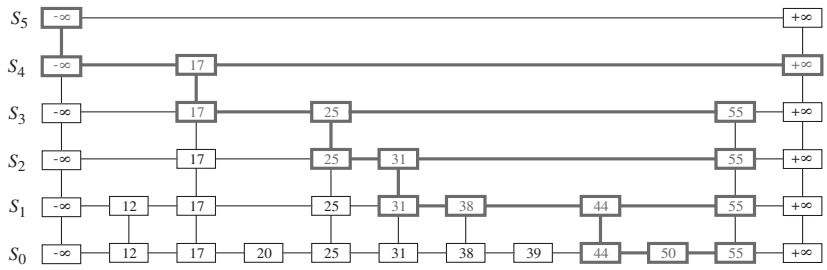

Figure 19.15: Example of a search in a skip list. The positions visited and the links traversed when searching (unsuccessfully) for key 52 are drawn with thick lines.

19.6.2 Update Operations

Given the SkipSearch method, it is easy to implement find(k)—we simply perform $p \leftarrow$ SkipSearch(k) and test whether key$(p) = k$. As it turns out, the expected running time of the SkipSearch algorithm is $O(\log n)$. We postpone this analysis, however, until after we discuss the update methods for skip lists.

Insertion

The insertion algorithm for skip lists uses randomization to decide how many nodes for the new item (k, e) should be added to the skip list. We begin the insertion of a new item (k, e) into a skip list by performing a SkipSearch(k) operation. This gives us the position p of the bottom-level item with the largest key less than or equal to k (note that p may be the node of the special item with key $-\infty$). We then insert (k, e) in this bottom-level list immediately after p. After inserting the new item at this level, we call a method random() that returns a random bit that is 1 (which stands for "heads") or 0 (which stands for "tails"), each with probability $1/2$. If the bit comes up tails, then we stop here. If the bit comes up heads, on the other hand, then we backtrack to the previous (next higher) level and insert (k, e) in this level at the appropriate position. We again generate a random bit, and if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new item (k, e) in lists until we finally get a flip that comes up tails. We link together all the nodes for the new item, (k, e), created in this process to create the ***tower*** for (k, e). We give the pseudocode for this insertion algorithm for a skip list S in Algorithm 19.16 and we illustrate this algorithm in Figure 19.17. Our insertion algorithm uses an operation insertAfterAbove$(p, q, (k, e))$ that inserts a node storing the item (k, e) after p (on the same level as p) and above the node q, returning the node r for the new item (and setting internal references so that after, before, above, and below methods will work correctly for p, q, and r).

Algorithm SkipInsert(k, e):
Input: Item (k, e)
Output: None

```
p ← SkipSearch(k)
q ← insertAfterAbove(p, null, (k, e))        // we are at the bottom level
while random() = 1 do
    while above(p) = null do
        p ← before(p)        // scan backward
    p ← above(p)        // jump up to higher level
    q ← insertAfterAbove(p, q, (k, e))        // insert new item
```

Algorithm 19.16: Insertion in a skip list, assuming random() returns a random number between 0 and 1, and we never insert past the top level.

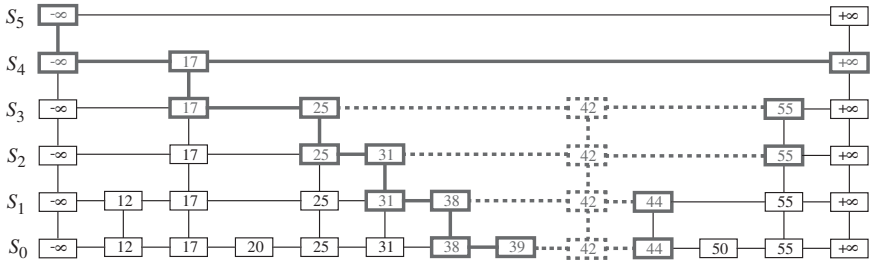

Figure 19.17: Insertion of an element with key 42 into the skip list of Figure 19.13. The nodes visited and the links traversed are drawn with thick lines. The nodes inserted to hold the new item are drawn with dashed lines.

Removal

Like the search and insertion algorithms, the removal algorithm for a skip list S is quite simple. In fact, it is even easier than the insertion algorithm. Namely, to perform a remove(k) operation, we begin by performing a search for the given key k. If a node p with key k is not found, then we return the ***null*** element. Otherwise, if a node p with key k is found (on the bottom level), then we remove all the nodes above p, which are easily accessed by using above operations to climb up the tower of this item in S starting at p. The removal algorithm is illustrated in Figure 19.18 and a detailed description of it is left as an exercise (R-19.14). As we show in the next subsection, the running time for removal in a skip list is expected to be $O(\log n)$.

Before we give this analysis, however, there are some minor improvements to the skip list data structure we would like to discuss. First, we don't need to store references to items at the levels above 0, because all that is needed at these levels are references to keys. Second, we don't actually need the above method. In fact, we don't need the before method either. We can perform item insertion and removal in strictly a top-down, scan-forward fashion, thus saving space for "up" and "prev" references. We explore the details of this optimization in an exercise (C-19.17).

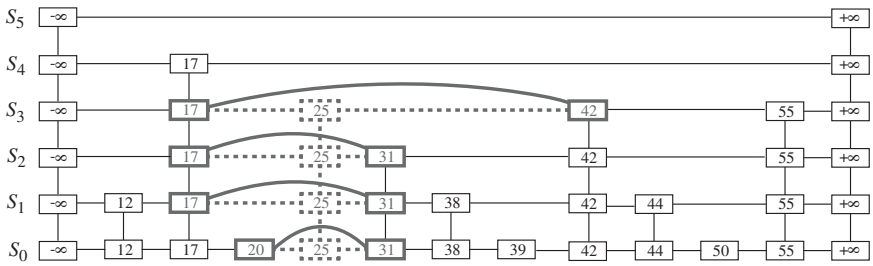

Figure 19.18: Removal of the item with key 25 from the skip list of Figure 19.17. The positions visited and the links traversed after the initial search are drawn with thick lines. The positions removed are drawn with dashed lines.

Maintaining the Topmost Level

A skip list S must maintain a reference to the topmost, left position in S as an instance variable, and must have a policy for any insertion that wishes to continue inserting a new item past the top level of S. There are two possible courses of action we can take, both of which have their merits.

One possibility is to restrict the top level, h, to be kept at some fixed value that is a function of n, the number of elements currently in the dictionary (from the analysis we will see that $h = \max\{10, 2\lceil \log n \rceil\}$ is a reasonable choice, and picking $h = 3\lceil \log n \rceil$ is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new item once we reach the topmost level (unless $\lceil \log n \rceil < \lceil \log(n+1) \rceil$, in which case we can now go at least one more level, since the bound on the height is increasing).

The other possibility is to let an insertion continue inserting a new element as long it keeps returning heads from the random-number generator. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than $O(\log n)$ is very low, so this design choice should also work.

However, either choice will still result in our being able to perform element search, insertion, and removal in expected $O(\log n)$ time, which we will show in the next section.

19.6.3 A Probabilistic Analysis of Skip Lists

As we have shown above, skip lists provide a simple implementation of an ordered dictionary. In terms of worst-case performance, however, skip lists are not a superior data structure. In fact, if we don't officially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an infinite loop (it is not actually an infinite loop, however, since the probability of having a fair coin repeatedly come up heads forever is 0). Moreover, we cannot infinitely add elements to a list without eventually running out of memory. In any case, if we terminate item insertion at the highest level h, then the ***worst-case*** running time for performing the find, insert, and remove operations in a skip list S with n items and height h is $O(n + h)$. This worst-case performance occurs when the tower of every item reaches level $h-1$, where h is the height of S. However, this event has very low probability. Judging from this worst case, we might conclude that the skip list structure is strictly inferior to the other dictionary implementations discussed earlier in this chapter. But this would not be a fair analysis, for this worst-case behavior is a gross overestimate. Because the insertion step involves randomization, a more honest analysis of skip lists involves a bit of probability.

Let us begin with the expected value of the height h of S (assuming that we do not terminate insertions early). The probability that a given item is stored in a position at level i is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is $1/2^i$. Hence, the probability, P_i, that level i has at least one item is at most $P_i \leq n/2^i$, for the probability that any one of n different events occurs is, at most, the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one item, that is, it is no more than P_i. This means that h is larger than, say, $3 \log n$ with probability at most

$$P_{3\log n} \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}.$$

More generally, given a constant $c > 1$, h is larger than $c \log n$ with probability at most $1/n^{c-1}$. That is, the probability that h is smaller than or equal to $c \log n$ is at least $1 - 1/n^{c-1}$. Thus, with high probability, the height h of S is $O(\log n)$.

Consider the running time of a search in skip list S, and recall that such a search involves two nested while-loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is $O(\log n)$ with high probability, the number of drop-down steps is $O(\log n)$ with high probability.

So we have yet to bound the number of scan-forward steps we make. Let n_i be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level $i + 1$. If any of these items were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in n_i is $1/2$. Therefore, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. That is, each n_i is a geometric random variable with parameter $1/2$. Thus, $E[n_i] = 2$; hence, the expected amount of time spent scanning forward at any level i is $O(1)$. Since S has $O(\log n)$ levels with high probability, a search in S takes expected time $O(\log n)$. By a similar analysis, we can show that the expected running time of an insertion or a removal is $O(\log n)$. Moreover, by applying the Chernoff bound for a sum of independent geometric random variables, given in Theorem 19.15, we can show that searches and updates in a skip list run in $O(\log n)$ time with high probability.

Finally, let us turn to the space requirement of a skip list S. As we observed above, the expected number of items at level i is $n/2^i$, which means that the expected total number of items in S is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n.$$

Hence, the expected space requirement of S is $O(n)$.

19.7 Exercises

Reinforcement

R-19.1 Give a variation of Algorithm 19.2 (randomSort) that runs in $O(n)$ time with probability $1 - 1/n^4$.

R-19.2 Suppose two teams, the Anteaters and the Bears, have a long rivalry in basketball. Suppose further that in any given game, the Anteaters will beat the Bears with probability $2/3$, independent of any other games that they play. Give a bound on the probability that, in spite of this, the Bears will win a majority of n games that they play.

R-19.3 Suppose a certain birth defect occurs independently at random with probability $p = 0.02$ in any live birth. Use a Chernoff bound to bound the probability that more than 4% of the 1 million children born in a given large city have this birth defect.

R-19.4 Suppose a builder, named Bob, wants to hammer in 20 nails into a piece of wood. Bob is very strong and can hammer down a nail in a single blow if he hits the nail square on its head. But Bob is also a little near-sighted and, in any given swing of his hammer, he only hits any given nail square on its head with probability $p = 1/3$ and misses it completely with probability $1 - p$. Derive a bound on the probability that it takes Bob more than 120 swings to hammer down all 20 nails.

R-19.5 Suppose A is an array of n bits, half of which are 0's and half of which are 1's. But the bits in A can be in any order, so that the worst-case performance of any deterministic algorithm for finding a 1 in A is $\Theta(n)$. Give a Las Vegas algorithm that finds a 1 in A in $O(\log n)$ time with high probability.

R-19.6 Give a Monte Carlo algorithm for the previous problem that examines at most $\lceil \log n \rceil$ entries in A and succeeds in finding a 1 in A with high probability.

R-19.7 Suppose that a well-known collector, Kivas Fajo, is trying to collect each of 50 coupons, as in the coupon collector problem. Derive good upper and lower bounds on the expected number of times that Kivas has to visit the ticket window to get all 50 coupons.

R-19.8 Consider the cycle graph, C_n, consisting of vertices $v_0, v_1, \ldots, v_{n-1}$ and edges $(v_i, v_{i+1 \bmod n})$, for $i = 0, \ldots, n-1$. Clearly, the size of a minimum cut of C_n is 2. Determine the number of minimum cuts of C_n.

R-19.9 Derive the running time of Algorithm 19.10 (RecursiveContractGraph) using a recurrence relation.

R-19.10 Show that a graph with n vertices has at most $\binom{n}{2}$ minimum cuts.

R-19.11 Given a parameter, k, suppose we wish to find a number, p, that is prime with probability 2^{-k}. What is the asymptotic number of arithmetic operations needed?

R-19.12 Suppose we have a six-sided die, which we roll n times, and let X denote the number of times we role a 1.

(a) What is $E[X]$?

(b) Show that $X < n/3$ with high probability.

R-19.13 Draw an example skip list resulting from performing the following sequence of operations on the skip list in Figure 19.18: remove(38), insert(8,x), insert(24,y), remove(55). Assume the coin flips for the first insertion yield two heads followed by tails, and those for the second insertion yield three heads followed by tails.

R-19.14 Give a pseudocode description of the remove dictionary operation, assuming the dictionary is implemented by a skip-list structure.

Creativity

C-19.1 Suppose that Bob wants a constant-time method for implementing the random(k) method, which returns a random integer in the range $[0, k-1]$. Bob has a source of unbiased bits, so to implement random(k), he samples $\lceil \log k \rceil$ of these bits, interprets them as an unsigned integer, K, and returns the value $K \bmod k$. Show that Bob's algorithm does not return every integer in the range $[0, k-1]$ with equal probability.

C-19.2 Design a variation of Algorithm 19.2 (randomSort) that inserts the pairs (r_i, x_i) into a balanced tree and calls itself recursively when r_i is found to be equal to one of the previously generated random values. Give pseudocode for this variation of the algorithm and analyze its running time. Also, discuss its advantages and disadvantages with respect to the original algorithm.

C-19.3 Design a variation of Algorithm 19.2 (randomSort) that begins by generating distinct random values, r_i ($i = 1, \cdots, n$) and then sorts the pairs (r_i, x_i) . Give pseudocode for this variation of the algorithm and analyze its running time. Also, discuss its advantages and disadvantages with respect to the original algorithm.

C-19.4 Consider a modification of the Fisher-Yates random shuffling algorithm where we replace the call to random($k+1$) with random(n), and take the for-loop down to 0, so that the algorithm now swaps each element with another element in the array, with each cell in the array having an equal likelihood of being the swap location. Show that this algorithm does not generate every permutation with equal probability.

Hint: Consider the case when $n = 3$.

C-19.5 Suppose you have a collection, S, of n distinct items and you wish to select a random sample of these items of size exactly $\lceil n^{1/2} \rceil$. Describe an efficient method for selecting such a sample so that each element in S has an equal probability of being included in the sample.

C-19.6 Suppose you have a collection, S, of n distinct items and you create a random sample, R, of S, as follows: For each x in S, select it to belong to R independently with probability $1/n^{1/2}$. Derive bounds on the probability that the number of items in R is more than $2n^{1/2}$ or less than $n^{1/2}/2$.

C-19.7 Suppose that there is a collection of $3n$ distinct coupons, n of which are colored red and $2n$ of which are colored blue. Suppose that each time you go to a ticket window to get a coupon, the clerk first randomly decides, with probability $1/2$, whether he will give you a red coupon or blue coupon and then he chooses a coupon uniformly at random from among the coupons that are that color. What is the expected number of times that you must visit the ticket window to get all $3n$ coupons?

C-19.8 Show that if we do all arithmetic modulo a prime number, p, then, for any integer $x > 0$,

$$\{ix \bmod p\colon\ i = 0, 1, \ldots, p-1\} = \{i\colon\ i = 0, 1, \ldots, p-1\}.$$

Hint: Use the fact that if p is prime, then every nonzero integer less than p has a multiplicative inverse when we do arithmetic modulo p.

C-19.9 Give an algorithm that computes a minimum cut of a graph with n vertices by $O(n)$ applications of a maximum flow algorithm to a flow network derived from G.

C-19.10 Let $P(x)$ be a probability function that satisfies the recurrence $P(i+1) \geq P(i) - \frac{1}{4}P(i)^2$, with $P(0)$ a constant. Show that $P(2\log n)$ is $\Omega\left(\frac{1}{\log n}\right)$.

C-19.11 Give a randomized algorithm that computes all minimum cuts of a graph with high probability.

C-19.12 Show that if the compositeness witness function, $\mathsf{witness}(x, n)$, of the Rabin-Miller algorithm returns **true**, then the number n is composite.

C-19.13 Suppose we have a Monte Carlo algorithm, A, and a deterministic algorithm, B, for testing if the output of A is correct. How can we use A and B to construct a Las Vegas algorithm? Also, if A succeeds with probability $1/2$, and both A and B run in $O(n)$ time, what is the expected running time of the Las Vegas algorithm that is produced?

C-19.14 Suppose $X_1, X_2, \ldots, X_n$ is a set of mutually independent indicator random variables, such that each X_i is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let μ' denote an upper bound on the mean of X, that is, $E[X] = \mu \leq \mu'$. Show that, for $\delta > 0$,

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

C-19.15 Let $Y = Y_1 + Y_2 + \cdots + Y_n$ be the sum of n independent geometric random variables with parameter p. Show that, for $\alpha = 1/p$,

$$\Pr(Y < 0.25\,\alpha\,n) \leq 0.75^n.$$

C-19.16 Describe how to perform an operation, RangeSearch(k_1, k_2), which returns all the items with keys in the range $[k_1, k_2]$, in an ordered dictionary implemented with a skip list, and show that it runs in expected time $O(\log n + s)$, where n is the number of elements in the skip list and s is the number of items returned.

C-19.17 Show that the methods above(p) and before(p) are not actually needed to efficiently implement a dictionary using a skip list. That is, we can implement item insertion and removal in a skip list using a strictly top-down, scan-forward approach, without ever using the above or before methods.

C-19.18 Show that the randomized quick-sort algorithm runs in $O(n \log n)$ time with high probability.

Applications

A-19.1 A renowned food critic, Anton Ego, will enjoy a meal only if it is the highest-quality meal he has ever eaten up to that point in his life. Assuming that the qualities of the n meals he eats in his life are distinct and come in a random order over the course of his life, what is the expected number of times that Anton Ego will enjoy a meal in his life?

A-19.2 In the Mega Millions lottery game, a player picks five ***lucky*** numbers, in the range from 1 to 56, and one additional ***Mega*** number, in the range from 1 to 46. In order to win the jackpot, a player must match all six numbers. If there is no jackpot winner for a given drawing, then the jackpot is rolled into the next drawing. Suppose that every time a lottery ticket is sold it is chosen as an independent random pick of five lucky numbers and a Mega number. What is the expected number of Mega Millions lottery tickets that must be sold for a given drawing to guarantee with 100% certainty that there is a winner?

A-19.3 In a famous experiment, Stanley Milgram told a group of people in Kansas and Nebraska to each send a postcard to a lawyer in Boston, but they had to do it by forwarding it to someone that they knew, who had to forward it to someone that they knew, and so on. Most of the postcards that were successfully forwarded made it in 6 hops, which gave rise to the saying that everyone in America is separated by "six degrees of separation." The idea behind this experiment is also behind a technique, called ***probabilistic packet marking***, for doing traceback during a distributed denial-of-service attack, where a website is bombarded by connection requests. In implementing the probabilistic packet marking strategy, a router, R, will, with some probability, $p \leq 1/2$, replace some seldom-used parts of a packet it is processing with the IP address for R, to enable tracing back the attack to the sender. It is as if, in the Milgram experiment, there is just one sender, who is mailing multiple postcards, and each person forwarding a postcard would, with probability, p, erase the return address and replace it with his own. Suppose

that an attacker is sending a large number of packets in a denial-of-service attack to some recipient, and every one of the d routers in the path from the sender to the recipient is performing probabilistic packet marking.

(a) What is the probability that the router farthest from the recipient will mark a packet and this mark will survive all the way to the recipient?

(b) Derive a good upper bound on the expected number of packets that the recipient needs to collect to identify all the routers along the path from the sender to the recipient.

A-19.4 The Massachusetts state lottery game, Cash WinFall, used to have a way that anyone with enough money and time could stand a good chance of getting rich, and it is reported that an MIT computer scientist did just that. In this game, a player picks 6 numbers from the range from 1 to 46. If he matches all 6, then he could win as much as $2 million, but the odds of that payout don't justify a bet, so let us ignore the possibility of winning this jackpot. Nevertheless, there were times when matching just 5 of the 6 numbers in a $2 lottery ticket would pay $100,000. Suppose in this scenario that you were able to bet $600,000.

(a) What is the expected amount that you would win?

(b) Derive a bound on the probability that you would lose $300,000 or more in this scenario, that is, that you would have 3 or fewer of the 5 of the 6 winning tickets.

A-19.5 There is a probabilistic data structure often used for representing sets in networking and computer security applications, which is known as the ***Bloom filter***. This data structure represents a set, S, using of an array, A, of m bits and a collection of k hash functions, $h_1, \ldots, h_k$. Initially, all the bits in A are 0. To add an element, x, to the set, S, we assign each of the bits, $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$, to 1. To test if an element belongs to S, we check if all these bits are equal to 1. If so, then we say that x is a member of S and, if not, then we say that x is not a member of S. Note that this algorithm has a one-sided error, since it is always correct when it says that x is not in S, but there is a chance that has a ***false positive***, saying that x belongs to S when really it doesn't. Assuming that each hash function maps any element, x, to k distinct random locations in A, and we have inserted $n < kn/2$ elements into S, derive a bound on the probability that a Bloom filter returns a false positive response.

A-19.6 There is a classic surprising fact from probability, known as the ***birthday paradox***, which states that in a room of at least 23 people there is better than a 50-50 chance that two of them have the same birthday. This fact is surprising to some, because there are 366 possible birthdays, which is much larger than 23. While this is surprising, there is an interesting security application of the analysis that goes into the birthday paradox. Suppose a company is installing a keypad on its entry door and will be assigning each employee with an independently chosen 8-digit PIN to use when they enter the building. So there are 100 million possible PINs, but let us use n to denote this number of possible PINs and m to denote the number of employees.

(a) Imagine that we assign PINs to employees sequentially, one employee at a time. What is the probability, p_i, that the ith employee has a distinct PIN given that the $i-1$ PINs given before are distinct?

(b) Note that the probability the PINs are all distinct is the product of the p_i's, for $i = 1, 2, \ldots, m$. Use the fact that $1 - x \leq e^{-x}$, for $0 < x < 1$, to show that this product is bounded by $e^{-m^2/2n}$.

(c) Given the above bound, how many PINs does the system need to produce so that the probability that two employees have the same PIN is at least $1/2$?

Chapter Notes

The Rabin-Miller primality testing algorithm is presented in [177]. Random shuffling is discussed by Fisher and Yates [71], Durstenfeld [59], and Knuth [131]. Arkin *et al.* [13] describe how they were able to exploit a poorly designed random shuffling algorithm to defeat online poker systems. The stable marriage problem was first studied by Gale and Shapley [78], and they present a proposal-based algorithm for solving it. Our analysis of the stable marriage problem is based on unpublished course notes by John Canny. The randomized processor load balancing application we mention for Chernoff bounds is from in a paper by Raab and Steger [176]. The randomized minimum cut algorithm based on contractions was introduced by Karger [118] and improved by Karger and Stein [120]. Skip lists were introduced by Pugh [175]. Our analysis of skip lists is a simplification of a presentation given in the book by Motwani and Raghavan [162]. In addition, there are also other more in-depth analyses of skip lists [126, 168, 172], as well as a binary-tree analogue due to Seidel and Aragon [191].