12

Randomized Algorithms*

Rajeev Motwani
Stanford University

Prabhakar Raghavan
Yahoo! Research

12.1 Introduction

A **randomized algorithm** is one that makes random choices during its execution. The behavior of such an algorithm may thus, be random even on a fixed input. The design and analysis of a randomized algorithm focuses on establishing that it is likely to behave "well" on every input; the likelihood in such a statement depends only on the probabilistic choices made by the algorithm during execution and not on any assumptions about the input. It is especially important to distinguish a randomized algorithm from the average-case analysis of algorithms, where one analyzes an algorithm assuming that its input is drawn from a fixed probability distribution. With a randomized algorithm, in contrast, no assumption is made about the input.

Two benefits of randomized algorithms have made them popular: simplicity and efficiency. For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both. In the following text, we make these notions concrete through a number of illustrative examples. We assume that the reader has had undergraduate courses in algorithms and complexity,

* Chapter coauthor Rajeev Motwani passed away in a tragic drowning accident on June 5, 2009. He was a Professor of Computer Science at Stanford University and a brilliant researcher and educator who was awarded the prestigious Gödel Prize, and was an early advisor and supporter of many companies including Google and PayPal.

and in probability theory. A comprehensive source for randomized algorithms is the book by the authors [25]. The articles by Karp [18], Maffioli et al. [22], and Welsh [45] are good surveys of randomized algorithms. The book by Mulmuley [26] focuses on randomized geometric algorithms.

Throughout this chapter we assume the RAM model of computation, in which we have a machine that can perform the following operations involving registers and main memory: input–output operations, memory–register transfers, indirect addressing, and branching and arithmetic operations. Each register or memory location may hold an integer that can be accessed as a unit, but an algorithm has no access to the representation of the number. The arithmetic instructions permitted are $+, -, \times,$ and $/$. In addition, an algorithm can compare two numbers, and evaluate the square root of a positive number. In this chapter $\mathbf{E}[X]$ will denote the expectation of a random variable X, and $\mathbf{Pr}[A]$ will denote the probability of an event A.

12.2 Sorting and Selection by Random Sampling

Some of the earliest randomized algorithms included algorithms for sorting a set (S) of numbers, and the related problem of finding the kth smallest element in S. The main idea behind these algorithms is the use of random sampling: a randomly chosen member of S is unlikely to be one of its largest or smallest elements; rather, it is likely to be "near the middle." Extending this intuition suggests that a random sample of elements from S is likely to be spread "roughly uniformly" in S. We now describe randomized algorithms for sorting and selection based on these ideas.

Algorithm RQS:

Input: A set of numbers S.

Output: The elements of S sorted in increasing order.

1. Choose an element y uniformly at random from S: every element in S has equal probability of being chosen.

2. By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.

3. Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.

Algorithm RQS is an example of a randomized algorithm—an algorithm that makes random choices during execution. It is inspired by the Quicksort algorithm due to Hoare [13], and described in [25]. We assume that the random choice in Step 1 can be made in unit time. What can we prove about the running time of RQS?

We now analyze the expected number of comparisons in an execution of RQS. Comparisons are performed in Step 2, in which we compare a randomly chosen element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of rank i (the ith smallest element) in the set S. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution, and the value 0 otherwise. Thus, the total number of comparisons is $\Sigma_{i=1}^{n} \Sigma_{j>i} X_{ij}$. By linearity of expectation the expected number of comparisons is

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i} \mathbf{E}\left[X_{ij}\right]. \tag{12.1}$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared during an execution. Then

$$\mathbf{E}\left[X_{ij}\right] = p_{ij} \times 1 + \left(1 - p_{ij}\right) \times 0 = p_{ij}. \tag{12.2}$$

To compute p_{ij} we view the execution of RQS as a binary tree T each node of which is labeled with a distinct element of S. The root of the tree is labeled with the element y chosen in Step 1, the left subtree of y contains the elements in S_1, and the right subtree of y contains the elements in S_2. The structures of the two subtrees are determined recursively by the executions of RQS on S_1 and S_2. The root y is compared to the elements in the two subtrees, but no comparison is performed between an element of the left subtree and an element of the right subtree. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

Consider the permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at distance exactly i from the root. The following two observations lead to the determination of p_{ij}.

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left subtree of $S_{(k)}$ while $S_{(j)}$ will belong to the right subtree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i, j\}$, there is an ancestor–descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by RQS.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element and hence, to appear first in π. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

It follows that $p_{ij} = 2/(j-i+1)$. By Equations 12.1 and 12.2, the expected number of comparisons is given by

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}.
\end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the nth harmonic number, defined by $H_n = \sum_{k=1}^{n} 1/k$.

THEOREM 12.1 *The expected number of comparisons in an execution of RQS is at most* $2nH_n$.

Now $H_n = \ln n + \Theta(1)$, so that the expected running time of RQS is $O(n \log n)$. Note that this expected running time holds for every input. It is an expectation that depends only on the random choices made by the algorithm, and not on any assumptions about the distribution of the input.

12.2.1 Randomized Selection

We now consider the use of random sampling for the problem of selecting the kth smallest element in a set S of n elements drawn from a totally ordered universe. We assume that the elements of S are

all distinct, although it is not very hard to modify the following analysis to allow for multisets. Let $r_S(t)$ denote the rank of an element t (the kth smallest element has rank k) and recall that $S_{(i)}$ denotes the ith smallest element of S. Thus, we seek to identify $S_{(k)}$. We extend the use of this notation to subsets of S as well. The following algorithm is adapted from one due to Floyd and Rivest [10].

Algorithm LazySelect:

Input: A set S of n elements from a totally ordered universe, and an integer k in $[1, n]$.

Output: The kth smallest element of S, $S_{(k)}$.

1. Pick $n^{3/4}$ elements from S, chosen independently and uniformly at random with replacement; call this multiset of elements R.

2. Sort R in $O\left(n^{3/4} \log n\right)$ steps using any optimal sorting algorithm.

3. Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing a and b to every element of S, determine $r_S(a)$ and $r_S(b)$.

4. if $k < n^{1/4}$, let $P = \{y \in S \mid y \leq b\}$ and $r = k$;
else if $k > n - n^{1/4}$, let $P = \{y \in S \mid y \geq a\}$ and $r = k - r_S(a) + 1$;
else if $k \in [n^{1/4}, n - n^{1/4}]$, let $P = \{y \in S \mid a \leq y \leq b\}$ and $r = k - r_S(a) + 1$;
Check whether $S_{(k)} \in P$ and $|P| \leq 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such a set P is found.

5. By sorting P in $O\left(|P| \log |P|\right)$ steps, identify P_r, which is $S_{(k)}$.

Figure 12.1 illustrates Step 3, where small elements are at the left end of the picture and large ones to the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy, since we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three **if** statements in Step 4 we execute. The sorting in Step 5 can be performed in $O\left(n^{3/4} \log n\right)$ steps.

Thus, the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. The element $S_{(k)}$ that we seek is in P, the set of elements between a and b.
2. The set P of elements is not very large, so that we can sort P inexpensively in Step 5.

As in the analysis of RQS we measure the running time of LazySelect in terms of the number of comparisons performed by it. The following theorem is established using the *Chebyshev bound* from elementary probability theory; a full proof may be found in [25].

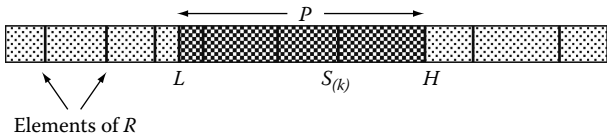

FIGURE 12.1 The LazySelect algorithm.

THEOREM 12.2 *With probability* $1 - o\left(n^{-1/4}\right)$, *LazySelect finds* $S_{(k)}$ *on the first pass through Steps 1–5, and thus, performs only* $2n + o(n)$ *comparisons.*

This adds to the significance of LazySelect: the best known deterministic selection algorithms use $3n$ comparisons in the worst case, and are quite complicated to implement.

12.3 A Simple Min-Cut Algorithm

Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be independent if the probability that they both occur is given by

$$\mathbf{Pr}\left[\mathcal{E}_1 \cap \mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_2\right]. \tag{12.3}$$

More generally when $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\mathbf{Pr}\left[\mathcal{E}_1 \cap \mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_1 \mid \mathcal{E}_2\right] \times \mathbf{Pr}\left[\mathcal{E}_2\right] = \mathbf{Pr}\left[\mathcal{E}_2 \mid \mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_1\right], \tag{12.4}$$

where $\mathbf{Pr}\left[\mathcal{E}_1 \mid \mathcal{E}_2\right]$ denotes the conditional probability of $\mathcal{E}_1$ given $\mathcal{E}_2$. When a collection of events is not independent, the probability of their intersection is given by the following generalization of Equation 12.4:

$$\mathbf{Pr}\left[\cap_{i=1}^{k}\mathcal{E}_i\right] = \mathbf{Pr}\left[\mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_2 \mid \mathcal{E}_1\right] \times \mathbf{Pr}\left[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2\right] \cdots \mathbf{Pr}\left[\mathcal{E}_k \mid \cap_{i=1}^{k-1}\mathcal{E}_i\right]. \tag{12.5}$$

Let G be a connected, undirected multigraph with n vertices. A multigraph may contain multiple edges between any pair of vertices. A cut in G is a set of edges whose removal results in G being broken into two or more components. A min-cut is a cut of minimum cardinality. We now study a simple algorithm due to Karger [15] for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end points. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Remove edges between vertices that are merged, so that there are never any self-loops. This process of merging the two end-points of an edge into a single vertex is called the contraction of that edge. With each contraction, the number of vertices of G decreases by one. Note that as long as at least two vertices remain, an edge contraction does not reduce the min-cut size in G. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut (Figure 12.2). What is the probability that this algorithm finds a min-cut?

DEFINITION 12.1 For any vertex v in a multigraph G, the *neighborhood* of G, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

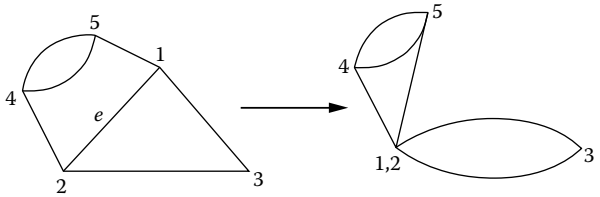

FIGURE 12.2 A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size and let C be a particular min-cut with k edges. Clearly G has at least $kn/2$ edges (otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k). We bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

For $1 \leq i \leq n-2$, let $\mathcal{E}_i$ denote the event of not picking an edge of C at the ith step. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\mathbf{Pr}[\mathcal{E}_1] \geq 1 - 2/n$. Conditioned on the occurrence of $\mathcal{E}_1$, there are at least $k(n-1)/2$ edges during the second step so that $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. Extending this calculation, $\mathbf{Pr}\left[\mathcal{E}_i \mid \cap_{j=1}^{i-1}\mathcal{E}_j\right] \geq 1 - 2/(n-i+1)$. We now recall Equation 12.5 to obtain

$$\mathbf{Pr}\left[\cap_{i=1}^{n-2}\mathcal{E}_i\right] \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

Our algorithm may err in declaring the cut it outputs to be a min-cut. But the probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$, so the probability of error is at most $1 - 2/n^2$. Repeating the aforementioned algorithm $n^2/2$ times making independent random choices each time, the probability that a min-cut is not found in any of the $n^2/2$ attempts is (by Equation 12.3) at most

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

By this process of repetition, we have managed to reduce the probability of failure from $1 - 2/n^2$ to less than $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small (the only consideration being that repetitions increase the running time). Note the extreme simplicity of this randomized min-cut algorithm. In contrast, most **deterministic algorithms** for this problem are based on network flow and are considerably more complicated.

12.3.1 Classification of Randomized Algorithms

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm always gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. Such an algorithm is called a **Las Vegas algorithm**.

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we prove that the probability of such an error is bounded. Such an algorithm is called a Monte Carlo algorithm. In Section 12.3 we observed a useful property of a **Monte Carlo algorithm:** if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. In some randomized algorithms both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. The reader is referred to [25] for a detailed discussion of these issues.

12.4 Foiling an Adversary

A common paradigm in the design of randomized algorithms is that of foiling an adversary. Whereas an adversary might succeed in defeating a deterministic algorithm with a carefully constructed "bad"

input, it is difficult for an adversary to defeat a deterministic algorithm in this fashion. Due to the random choices made by the randomized algorithm the adversary cannot, while constructing the input, predict the precise behavior of the algorithm. An alternative view of this process is to think of the randomized algorithm as first picking a series of random numbers which it then uses in the course of execution as needed. In this view, we may think of the random numbers chosen at the start as "selecting" one of a family of deterministic algorithms. In other words a randomized algorithm can be thought of as a probability distribution on deterministic algorithms. We illustrate these ideas in the setting of AND-OR tree evaluation; the following algorithm is due to Snir [39].

For our purposes an AND-OR tree is a rooted complete binary tree in which internal nodes at even distance from the root are labeled AND and internal nodes at odd distance are labeled OR. Associated with each leaf is a Boolean value. The evaluation of the game tree is the following process. Each leaf returns the value associated with it. Each OR node returns the Boolean OR of the values returned by its children, and each AND node returns the Boolean AND of the values returned by its children. At each step an evaluation algorithm chooses a leaf and reads its value. We do not charge the algorithm for any other computation. We study the number of such steps taken by an algorithm for evaluating an AND-OR tree, the worst case being taken over all assignments of Boolean values to the leaves.

Let T_k denote an AND-OR tree in which every leaf is at distance $2k$ from the root. Thus, any root-to-leaf path passes through k AND nodes (including the root itself) and k OR nodes, and there are 2^{2k} leaves. An algorithm begins by specifying a leaf whose value is to be read at the first step. Thereafter, it specifies such a leaf at each step, based on the values it has read on previous steps. In a deterministic algorithm, the choice of the next leaf to be read is a deterministic function of the values at the leaves read so far. For a randomized algorithm, this choice may be randomized. It is not hard to show that for any deterministic evaluation algorithm, there is an instance of T_k that forces the algorithm to read the values on all 2^{2k} leaves.

We now give a simple randomized algorithm and study the expected number of leaves it reads on any instance of T_k. The algorithm is motivated by the following simple observation. Consider a single AND node with two leaves. If the node were to return 0, at least one of the leaves must contain 0. A deterministic algorithm inspects the leaves in a fixed order, and an adversary can therefore always "hide" the 0 at the second of the two leaves inspected by the algorithm. Reading the leaves in a random order foils this strategy. With probability $1/2$, the algorithm chooses the hidden 0 on the first step, so its expected number of steps is $3/2$, which is better than the worst case for any deterministic algorithm. Similarly, in the case of an OR node, if it were to return a 1 then a randomized order of examining the leaves will reduce the expected number of steps to $3/2$. We now extend this intuition and specify the complete algorithm.

To evaluate an AND node v, the algorithm chooses one of its children (a subtree rooted at an OR node) at random and evaluates it by recursively invoking the algorithm. If 1 is returned by the subtree, the algorithm proceeds to evaluate the other child (again by recursive application). If 0 is returned, the algorithm returns 0 for v. To evaluate an OR node, the procedure is the same with the roles of 0 and 1 interchanged. We establish by induction on k that the expected cost of evaluating any instance of T_k is at most 3^k.

The basis ($k = 0$) is trivial. Assume now that the expected cost of evaluating any instance of T_{k-1} is at most 3^{k-1}. Consider first a tree T whose root is an OR node, each of whose children is the root of a copy of T_{k-1}. If the root of T were to evaluate to 1, at least one of its children returns 1. With probability $1/2$ this child is chosen first, incurring (by the inductive hypothesis) an expected cost of at most 3^{k-1} in evaluating T. With probability $1/2$ both subtrees are evaluated, incurring a net cost of at most $2 \times 3^{k-1}$. Thus, the expected cost of determining the value of T is

$$\leq \frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1}. \tag{12.6}$$

If on the other hand the OR were to evaluate to 0 both children must be evaluated, incurring a cost of at most $2 \times 3^{k-1}$.

Consider next the root of the tree T_k, an AND node. If it evaluates to 1, then both its subtrees rooted at OR nodes return 1. By the discussion in the previous paragraph and by linearity of expectation, the expected cost of evaluating T_k to 1 is at most $2 \times (3/2) \times 3^{k-1} = 3^k$. On the other hand, if the instance of T_k evaluates to 0, at least one of its subtrees rooted at OR nodes returns 0. With probability $1/2$ it is chosen first, and so the expected cost of evaluating T_k is at most

$$2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k.$$

THEOREM 12.3 *Given any instance of T_k, the expected number of steps for the aforementioned randomized algorithm is at most 3^k.*

Since $n = 4^k$ the expected running time of our randomized algorithm is $n^{\log_4 3}$, which we bound by $n^{0.793}$. Thus, the expected number of steps is smaller than the worst case for any deterministic algorithm. Note that this is a Las Vegas algorithm and always produces the correct answer.

12.5 The Minimax Principle and Lower Bounds

The randomized algorithm of Section 12.4 has an expected running time of $n^{0.793}$ on any uniform binary AND-OR tree with n leaves. Can we establish that no randomized algorithm can have a lower expected running time? We first introduce a standard technique due to Yao [46] for proving such lower bounds. This technique applies only to algorithms that terminate in finite time on all inputs and sequences of random choices.

The crux of the technique is to relate the running times of randomized algorithms for a problem to the running times of deterministic algorithms for the problem when faced with randomly chosen inputs. Consider a problem where the number of distinct inputs of a fixed size is finite, as is the number of distinct (deterministic, terminating, and always correct) algorithms for solving that problem. Let us define the **distributional complexity** of the problem at hand as the expected running time of the best deterministic algorithm for the worst distribution on the inputs. Thus, we envision an adversary choosing a probability distribution on the set of possible inputs, and study the best deterministic algorithm for this distribution. Let p denote a probability distribution on the set $\mathcal{I}$ of inputs. Let the random variable $C(I_{ismp}, A)$ denote the running time of deterministic algorithm $A \in \mathcal{A}$ on an input chosen according to p. Viewing a randomized algorithm as a probability distribution q on the set $\mathcal{A}$ of deterministic algorithms, we let the random variable $C(I, A_{ismq})$ denote the running time of this randomized algorithm on the worst-case input.

PROPOSITION 12.1 (Yao's minimax principle): For all distributions p over $\mathcal{I}$ and q over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_{ismp}, A)\right] \leq \max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_{ismq})\right].$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution p is a lower bound on the expected running time of the optimal (Las Vegas) randomized algorithm for Π. Thus, to prove a lower bound on the **randomized complexity** it suffices to choose any distribution p on the input and prove a lower bound on the expected running time of deterministic algorithms for that distribution. The power of this technique lies in the flexibility in the choice of p and, more importantly, the reduction to a lower bound on deterministic algorithms. It is important to remember that the deterministic algorithm "knows" the chosen distribution p.

The aforementioned discussion dealt only with lower bounds on the performance of Las Vegas algorithms. We briefly discuss Monte Carlo algorithms with error probability $\epsilon \in [0, 1/2]$. Let us define the distributional complexity with error ϵ, denoted $\min_{A \in \mathcal{A}} \mathbf{E}[C_\epsilon(I_{ismp}, A)]$, to be the minimum expected running time of any deterministic algorithm that errs with probability at most ϵ under the input distribution p. Similarly, we denote by $\max_{I \in \mathcal{I}} \mathbf{E}[C_\epsilon(I, A_{ismq})]$ the expected running time (under the worst input) of any randomized algorithm that errs with probability at most ϵ (again, the randomized algorithm is viewed as a probability distribution q on deterministic algorithms). Analogous to Proposition 12.1, we then have the following:

PROPOSITION 12.2 For all distributions p over $\mathcal{I}$ and q over $\mathcal{A}$ and any $\epsilon \in [0, 1/2]$,

$$\frac{1}{2}\left(\min_{A \in \mathcal{A}} \mathbf{E}\left[C_{2\epsilon}\left(I_{ismp}, A\right)\right]\right) \leq \max_{I \in \mathcal{I}} \mathbf{E}\left[C_\epsilon\left(I, A_{ismq}\right)\right].$$

12.5.1 Lower Bound for Game Tree Evaluation

We now apply Yao's minimax principle to the AND-OR tree evaluation problem. A randomized algorithm for AND-OR tree evaluation can be viewed as a probability distribution over deterministic algorithms, because the length of the computation as well as the number of choices at each step are both finite. We may imagine that all of these coins are tossed before the beginning of the execution.

The tree T_k is equivalent to a balanced binary tree all of whose leaves are at distance $2k$ from the root, and all of whose internal nodes compute the NOR function: a node returns the value 1 if both inputs are 0, and 0 otherwise. We proceed with the analysis of this tree of NORs of depth $2k$.

Let $p = (3 - \sqrt{5})/2$; each leaf of the tree is independently set to 1 with probability p. If each input to a NOR node is independently 1 with probability p, its output is 1 with probability

$$\left(\frac{\sqrt{5} - 1}{2}\right)^2 = \frac{3 - \sqrt{5}}{2} = p.$$

Thus, the value of every node of the NOR tree is 1 with probability p, and the value of a node is independent of the values of all the other nodes on the same level. Consider a deterministic algorithm that is evaluating a tree furnished with such random inputs; let v be a node of the tree whose value the algorithm is trying to determine. Intuitively, the algorithm should determine the value of one child of v before inspecting any leaf of the other subtree. An alternative view of this process is that the deterministic algorithm should inspect leaves visited in a depth-first search of the tree, except of course that it ceases to visit subtrees of a node v when the value of v has been determined. Let us call such an algorithm a depth-first pruning algorithm, referring to the order of traversal and the fact that subtrees that supply no additional information are "pruned" away without being inspected. The following result is due to Tarsi [41].

PROPOSITION 12.3 Let T be a NOR tree each of whose leaves is independently set to 1 with probability q for a fixed value $q \in [0, 1]$. Let $W(T)$ denote the minimum, over all deterministic algorithms, of the expected number of steps to evaluate T. Then, there is a depth-first pruning algorithm whose expected number of steps to evaluate T is $W(T)$.

Proposition 12.3 tells us that for the purposes of our lower bound, we may restrict our attention to depth-first pruning algorithms. Let $W(h)$ be the expected number of leaves inspected by a depth-first

pruning algorithm in determining the value of a node at distance h from the leaves, when each leaf is independently set to 1 with probability $(3 - \sqrt{5})/2$). Clearly

$$W(h) = W(h-1) + (1-p) \times W(h-1),$$

where the first term represents the work done in evaluating one of the subtrees of the node, and the second term represents the work done in evaluating the other subtree (which will be necessary if the first subtree returns the value 0, an event occurring with probability $1 - p$). Letting h be $\log_2 n$, and solving, we get $W(h) \geq n^{0.694}$.

THEOREM 12.4 *The expected running time of any randomized algorithm that always evaluates an instance of T_k correctly is at least $n^{0.694}$, where $n = 2^{2k}$ is the number of leaves.*

Why is our lower bound of $n^{0.694}$ less than the upper bound of $n^{0.793}$ that follows from Theorem 12.3? The reason is that we have not chosen the best possible probability distribution for the values of the leaves. Indeed, in the NOR tree if both inputs to a node are 1, no reasonable algorithm will read leaves of both subtrees of that node. Thus, to prove the best lower bound we have to choose a distribution on the inputs that precludes the event that both inputs to a node will be 1; in other words, the values of the inputs are chosen at random but not independently. This stronger (and considerably harder) analysis can in fact be used to show that the algorithm of Section 12.4 is optimal; the reader is referred to the paper of Saks and Wigderson [34] for details.

12.6 Randomized Data Structures

Recent research into data structures has strongly emphasized the use of randomized techniques to achieve increased efficiency without sacrificing simplicity of implementation. An illustrative example is the randomized data structure for dynamic dictionaries called *skip list* that is due to Pugh [28].

The dynamic dictionary problem is that of maintaining a set of keys X drawn from a totally ordered universe so as to provide efficient support of the following operations: `find`(q, X)—decide whether the query key q belongs to X and return the information associated with this key if it does indeed belong to X; `insert`(q, X)—insert the key q into the set X, unless it is already present in X; `delete`(q, X)—delete the key q from X, unless it is absent from X. The standard approach for solving this problem involves the use of a binary search tree and gives worst-case time per operation that is $O(\log n)$, where n is the size of X at the time the operation is performed. Unfortunately, achieving this time bound requires the use of complex rebalancing strategies to ensure that the search tree remains "balanced," i.e., has depth $O(\log n)$. Not only does rebalancing require more effort in terms of implementation, it also leads to significant overheads in the running time (at least in terms of the constant factors subsumed by the big-oh notation). The skip list data structure is a rather pleasant alternative that overcomes both these shortcomings.

Before getting into the details of randomized skip lists, we will develop some of the key ideas without the use of randomization. Suppose we have a totally ordered data set $X = \{x_1 < x_2 < \cdots < x_n\}$. A gradation of X is a sequence of nested subsets (called *levels*)

$$X_r \subseteq X_{r-1} \subseteq \cdots \subseteq X_2 \subseteq X_1$$

such that $X_r = \emptyset$ and $X_1 = X$. Given an ordered set X and a gradation for it, the level of any element $x \in X$ is defined as

$$L(x) = \max\{i \mid x \in X_i\},$$

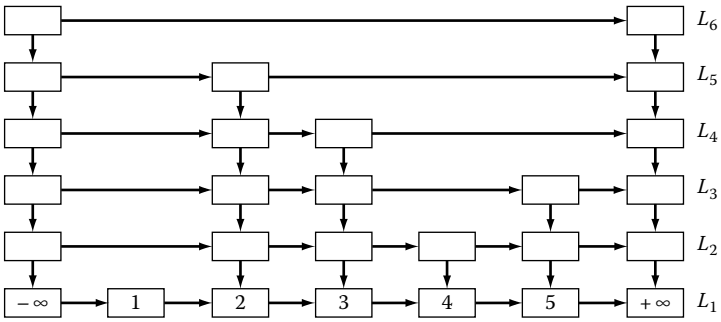

FIGURE 12.3 A skip list.

that is, $L(x)$ is the largest index i such that x belongs to the ith level of the gradation. In what follows, we will assume that two special elements $-\infty$ and $+\infty$ belong to each of the levels, where $-\infty$ is smaller than all elements in X and $+\infty$ is larger than all elements in X.

We now define an ordered list data structure with respect to a gradation of the set X. The first level, X_1 is represented as an ordered linked list, and each node x in this list has a stack of $L(x) - 1$ additional nodes directly above it. Finally, we obtain the skip list with respect to the gradation of X by introducing horizontal and vertical pointers between these nodes as illustrated in Figure 12.3. The skip list in Figure 12.3 corresponds to a gradation of the data set $X = \{1, 3, 4, 7, 9\}$ consisting of the following six levels:

$$
\begin{aligned}
X_6 &= \emptyset \\
X_5 &= \{3\} \\
X_4 &= \{3, 4\} \\
X_3 &= \{3, 4, 9\} \\
X_2 &= \{3, 4, 7, 9\} \\
X_1 &= \{1, 3, 4, 7, 9\}.
\end{aligned}
$$

Observe that starting at the ith node from the bottom in the left-most column of nodes, and traversing the horizontal pointers in order yields a set of nodes corresponding to the elements of the ith level X_i.

Additionally, we will view each level i as defining a set of intervals each of which is defined as the set of elements of X spanned by a horizontal pointer at level i. The sequence of levels X_i can be viewed as successively coarser partitions of X. In Figure 12.3, the levels determine the following partitions of X into a intervals.

$$
\begin{aligned}
X_6 &= [-\infty, +\infty] \\
X_5 &= [-\infty, 3] \cup [3, +\infty] \\
X_4 &= [-\infty, 3] \cup [3, 4] \cup [4, +\infty] \\
X_3 &= [-\infty, 3] \cup [3, 4] \cup [4, 9] \cup [9, +\infty] \\
X_2 &= [-\infty, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty] \\
X_1 &= [-\infty, 1] \cup [1, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty].
\end{aligned}
$$

An alternate view of the skip list is in terms of a tree defined by the interval partition structure, as illustrated in Figure 12.4 for the aforementioned example. In this tree, each node corresponds to an interval, and the intervals at a given level are represented by nodes at the corresponding level of the

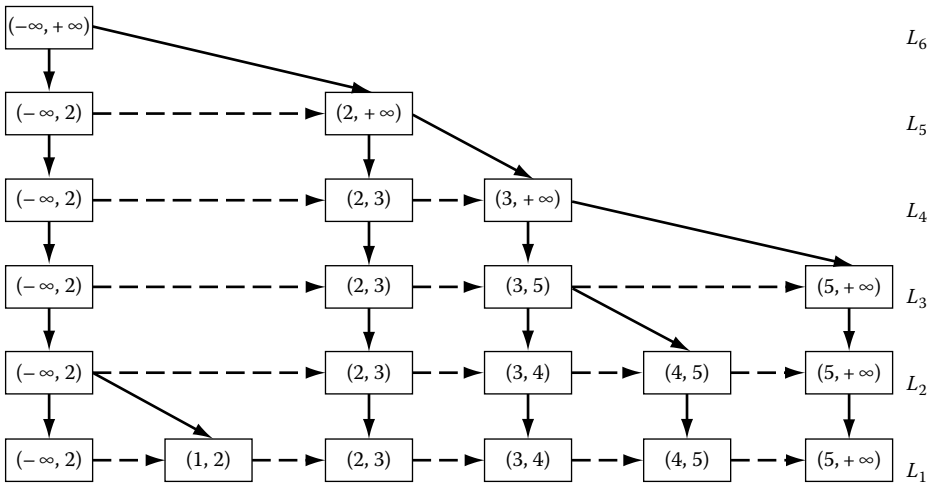

FIGURE 12.4 Tree representation of a skip list.

tree. When an interval J at level $i+1$ is a superset of an interval I at level i, then the corresponding node J has the node I as a child in this tree. Let $C(I)$ denote the number of children in the tree of a node corresponding to the interval I, i.e., it is the number of intervals from the previous level that are subintervals of I. Note that the tree is not necessarily binary since the value of $C(I)$ is arbitrary. We can view the skip list as a threaded version of this tree, where each thread is a sequence of (horizontal) pointers linking together the nodes at a level into an ordered list. In Figure 12.4, the broken lines indicate the threads, and the full lines are the actual tree pointers.

Finally, we need some notation concerning the membership of an element x in the intervals defined earlier, where x is not necessarily a member of X. For each possible x, let $I_j(x)$ be the interval at level j containing x. In the degenerate case where x lies on the boundary between two intervals, we assign it to the leftmost such interval. Observe that the nested sequence of intervals containing y,

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y),$$

corresponds to a root-leaf path in the tree corresponding to the skip list.

It remains to specify the choice of the gradation that determines the structure of a skip list. This is precisely where we introduce randomization into the structure of a skip list. The idea is to define a random gradation. Our analysis will show that with high probability, the search tree corresponding to a random skip list is "balanced," and then the dictionary operations can be efficiently implemented.

We define the random gradation for X as follows: given level X_i, the next level X_{i+1} is determined by independently choosing to retain each element $x \in X_i$ with probability $1/2$. The random selection process begins with $X_1 = X$ and terminates when for the first time the resulting level is empty. Alternatively, we may view the choice of the gradation as follows: for each $x \in X$, choose the level $L(x)$ independently from the geometric distribution with parameter $p = 1/2$ and place x in the levels $X_1, \ldots, X_{L(x)}$. We define r to be one more than the maximum of these level numbers. Such a random level is chosen for every element of X upon its insertion and remains fixed until its deletion.

We omit the proof of the following theorem bounding the space complexity of a randomized skip list. The proof is a simple exercise, and it is recommended that the reader verify this to gain some insight into the behavior of this data structure.

THEOREM 12.5 *A random skip list for a set X of size n has expected space requirement* $O(n)$.

We will go into more details about the time complexity of this data structure. The following lemma underlies the running time analysis.

LEMMA 12.1 The number of levels r in a random gradation of a set X of size n has expected value $\mathbf{E}[r] = O(\log n)$. Further, $r = O(\log n)$ with high probability.

PROOF We will prove the high probability result; the bound on the expected value follows immediately from this. Recall that the level numbers $L(x)$ for $x \in X$ are i.i.d. (independent and identically distributed) random variables distributed geometrically with parameter $p = 1/2$; notationally, we will denote these random variables by $Z_1, \ldots, Z_n$. Now, the total number of levels in the skip list can be determined as

$$r = 1 + \max_{x \in X} L(x) = 1 + \max_{1 \le i \le n} Z_i,$$

that is, as one more than the maximum of n i.i.d. geometric random variables.

For such geometric random variables with parameter p, it is easy to verify that for any positive real t, $\mathbf{Pr}[Z_i > t] \le (1-p)^t$. It follows that

$$\mathbf{Pr}\left[\max_i Z_i > t\right] \le n(1-p)^t = \frac{n}{2^t},$$

since $p = 1/2$ in this case. For any $\alpha > 1$, setting $t = \alpha \log n$, we obtain that

$$\mathbf{Pr}\left[r > \alpha \log n\right] \le \frac{1}{n^{\alpha-1}}.$$

We can now infer that the tree representing the skip list has height $O(\log n)$ with high probability. To show that the overall search time in a skip list is similarly bounded, we must first specify an efficient implementation of the `find` operation. We present the implementation of the dictionary operations in terms of the tree representation; it is fairly easy to translate this back into the skip list representation.

To implement `find`(y, X), we must walk down the path

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y).$$

For this, at level j, starting at the node $I_j(y)$, we use the vertical pointer to descend to the leftmost child of the current interval; then, via the horizontal pointers, we move rightward till the node $I_j(y)$ is reached. Note that it is easily determined whether y belongs to a given interval, or to an interval to its right. Further, in the skip list, the vertical pointers allow access only to the leftmost child of an interval, and therefore we must use the horizontal pointers to scan its children.

To determine the expected cost of a `find`(y, X) operation, we must take into account both the number of levels and the number of intervals/nodes scanned at each level. Clearly, at level j, the number of nodes visited is no more than the number of children of $I_{j+1}(y)$. It follows that the cost of `find` can be bounded by

$$O\left(\sum_{j=1}^{r} \left(1 + C\left(I_j(y)\right)\right)\right).$$

The following lemma shows that this quantity has expectation bounded by $O(\log n)$.

LEMMA 12.2 For any y, let $I_r(y), \ldots, I_1(y)$ be the search path followed by `find`(y, X) in a random skip list for a set X of size n. Then,

$$\mathbf{E}\left[\sum_{j=1}^{r}\left(1 + C(I_j(y))\right)\right] = O(\log n).$$

PROOF We begin by showing that for any interval I in a random skip list, $\mathbf{E}[C(I)] = O(1)$. By Lemma 12.1, we are guaranteed that $r = O(\log n)$ with high probability, and so we will obtain the desired bound. It is important to note that we really do need the high probability bound on Lemma 12.1, since it is incorrect to multiply the expectation of r with that of $1 + C(I)$ (the two random variables need not be independent). However, in the approach we will use, since $r > \alpha \log n$ with probability at most $1/n^{\alpha-1}$ and $\Sigma_j(1 + C(I_j(y))) = O(n)$, it can be argued that the case $r > \alpha \log n$ does not contribute significantly to the expectation of $\Sigma_j C(I_j(y))$.

To show that the expected number of children of an interval J at level i is bounded by a constant, we will show that the expected number of siblings of J (children of its parent) is bounded by a constant; in fact, we will only bound the number of right siblings since the argument for the number of left siblings is identical. Let the intervals to the right of J be the following:

$$J_1 = [x_1, x_2]; J_2 = [x_2, x_3]; \ldots; J_k = [x_k, +\infty].$$

Since these intervals exist at level i, each of the elements $x_1, \ldots, x_k$ belong to X_i. If J has s right siblings, then it must be the case that $x_1, \ldots, x_s \notin X_{i+1}$, and $x_{s+1} \in X_{i+1}$. The latter event occurs with probability $1/2^{s+1}$ since each element of X_i is independently chosen to be in X_{i+1} with probability $1/2$. Clearly, the number of right siblings of J can be viewed as a random variable that is geometrically distributed with parameter $1/2$. It follows that the expected number of right siblings of J is at most 2.

Consider now the implementation of the `insert` and `delete` operations. In implementing the operation `insert`(y, X), we assume that a random level $L(y)$ is chosen for y as described earlier. If $L(y) > r$, then we start by creating new levels from $r + 1$ to $L(y)$ and then redefine r to be $L(y)$. This requires $O(1)$ time per level, since the new levels are all empty prior to the insertion of y. Next we perform `find`(y, X) and determine the search path $I_r(y)$, ..., $I_1(y)$, where r is updated to its new value if necessary. Given this search path, the insertion can be accomplished in time $O(L(y))$ by splitting around y the intervals $I_1(y), \ldots, I_{L(y)}(y)$ and updating the pointers as appropriate. The `delete` operation is the converse of the `insert` operation; it involves performing `find`(y, X) followed by collapsing the intervals that have y as an end-point. Both operations incur cost that is the cost of a `find` operation and additional cost proportional to $L(y)$. By Lemmas 12.1 and 12.2, we obtain the following theorem.

THEOREM 12.6 *In a random skip list for a set X of size n, the operations* `find`, `insert`, *and* `delete` *can be performed in expected time* $O(\log n)$.

12.7 Random Reordering and Linear Programming

The linear programming problem is a particularly notable example of the two main benefits of randomization—simplicity and speed. We now describe a simple algorithm for linear programming based on a paradigm for randomized algorithms known as random reordering. For many problems it is possible to design natural algorithms based on the following idea. Suppose that the

input consists of n elements. Given any subset of these n elements, there is a solution to the partial problem defined by these elements. If we start with the empty set and add the n elements of the input one at a time, maintaining a partial solution after each addition, we will obtain a solution to the entire problem when all the elements have been added. The usual difficulty with this approach is that the running time of the algorithm depends heavily on the order in which the input elements are added; for any fixed ordering, it is generally possible to force this algorithm to behave badly. The key idea behind random reordering is to add the elements in a random order. This simple device often avoids the pathological behavior that results from using a fixed order.

The linear programming problem is to find the extremum of a linear objective function of d real variables subject to a set H of n constraints that are linear functions of these variables. The intersection of the n half-spaces defined by the constraints is a polyhedron in d-dimensional space (which may be empty, or possibly unbounded). We refer to this polyhedron as the feasible region. Without loss of generality [35] we assume that the feasible region is nonempty and bounded. (Note that we are not assuming that we can test an arbitrary polyhedron for nonemptiness or boundedness; this is known to be equivalent to solving a linear program.) For a set of constraints S, let $\mathcal{B}(S)$ denote the optimum of the linear program defined by S; we seek $\mathcal{B}(S)$.

Consider the following algorithm due to Seidel [37]: add the n constraints in random order, one at a time. After adding each constraint, determine the optimum subject to the constraints added so far. This algorithm may also be viewed in the following "backward" manner, which will prove useful in the sequel.

Algorithm SLP:

Input: A set of constraints H, and the dimension d.

Output: The optimum $\mathcal{B}(H)$.

0. If there are only d constraints, output $\mathcal{B}(H) = H$.

1. Pick a random constraint $h \in H$;
Recursively find $\mathcal{B}(H \backslash \{h\})$.

2.1. if $\mathcal{B}(H \backslash \{h\})$ does not violate h, output $\mathcal{B}(H \backslash \{h\})$
to be the optimum $\mathcal{B}(H)$.

2.2. else project all the constraints of $H \backslash \{h\}$ onto h and recursively
solve this new linear programming problem of one lower
dimension.

The idea of the algorithm is simple. Either h (the constraint chosen randomly in Step 1) is redundant (in which case we execute Step 2.1), or it is not. In the latter case, we know that the vertex formed by $\mathcal{B}(H)$ must lie on the hyperplane bounding h. In this case, we project all the constraints of $H \backslash \{h\}$ onto h and solve this new linear programming problem (which has dimension $d - 1$).

The optimum $\mathcal{B}(H)$ is defined by d constraints. At the top level of recursion, the probability that a random constraint h violates $\mathcal{B}(H \backslash \{h\})$ is at most d/n. Let $T(n, d)$ denote an upper bound on the expected running time of the algorithm for any problem with n constraints in d dimensions. Then, we may write

$$T(n, d) \leq T(n-1, d) + O(d) + \frac{d}{n}\left[O(dn) + T(n-1, d-1)\right]. \tag{12.7}$$

In Equation 12.7, the first term on the right denotes the cost of recursively solving the linear program defined by the constraints in $H\backslash\{h\}$. The second accounts for the cost of checking whether h violates $\mathcal{B}(H\backslash\{h\})$. With probability d/n it does, and this is captured by the bracketed expression, whose first term counts the cost of projecting all the constraints onto h. The second counts the cost of (recursively) solving the projected problem, which has one fewer constraint and dimension. The following theorem may be verified by substitution, and proved by induction.

THEOREM 12.7 *There is a constant b such that the recurrence (Equation 12.7) satisfies the solution* $T(n, d) \leq bnd!$.

In contrast if the choice in Step 1 of SLP were not random, the recurrence (Equation 12.7) would be

$$T(n, d) \leq T(n-1, d) + O(d) + O(dn) + T(n-1, d-1), \tag{12.8}$$

whose solution contains a term that grows quadratically in n.

12.8 Algebraic Methods and Randomized Fingerprints

Some of the most notable randomized results in theoretical computer science, particularly in complexity theory, have involved a nontrivial combination of randomization and algebraic methods. In this section we describe a fundamental randomization technique based on algebraic ideas. This is the randomized fingerprinting technique, originally due to Freivalds [11], for the verification of identities involving matrices, polynomials, and integers. We also describe how this generalizes to the so-called Schwartz–Zippel technique for identities involving multivariate polynomials (independently due to Schwartz [36] and Zippel [47]; see also DeMillo and Lipton [7]). Finally, following Lovász [21], we apply the technique to the problem of detecting the existence of perfect matchings in graphs.

The fingerprinting technique has the following general form. Suppose we wish to decide the equality of two elements x and y drawn from some "large" universe U. Assuming any reasonable model of computation, this problem has a deterministic complexity $\Omega(\log |U|)$. Allowing randomization, an alternative approach is to choose a random function from U into a smaller space V such that with high probability x and y are identical if and only if their images in V are identical. These images of x and y are said to be their fingerprints, and the equality of fingerprints can be verified in time $O(\log |V|)$. Of course, for any fingerprint function the average number of elements of U mapped to an element of V is $|U|/|V|$; so, it would appear impossible to find good fingerprint functions that work for arbitrary or worst-case choices of x and y. However, as we will show in the following text, when the identity-checking is only required to be correct for x and y chosen from a small subspace S of U, particularly a subspace with some algebraic structure, it is possible to choose good fingerprint functions without any a priori knowledge of the subspace, provided the size of V is chosen to be comparable to the size of S.

Throughout this section we will be working over some unspecified field $\mathcal{F}$. Since the randomization will involve uniform sampling from a finite subset of the field, we do not even need to specify whether the field is finite or not. The reader may find it helpful in the infinite case to assume that $\mathcal{F}$ is the field $\mathcal{Q}$ of rational numbers, and in the finite case to assume that $\mathcal{F}$ is $\mathcal{Z}_p$, the field of integers modulo some prime number p.

12.8.1 Freivalds' Technique and Matrix Product Verification

We begin by describing a fingerprinting technique for verifying matrix product identities. Currently, the fastest algorithm for matrix multiplication (due to Coppersmith and Winograd [6]) has running

time $O(n^{2.376})$, improving significantly on the obvious $O(n^3)$ time algorithm; however, the fast matrix multiplication algorithm has the disadvantage of being extremely complicated. Suppose we have an implementation of the fast matrix multiplication algorithm and, given its complex nature, are unsure of its correctness. Since program verification appears to be an intractable problem, we consider the more reasonable goal of verifying the correctness of the output produced by executing the algorithm on specific inputs. (This notion of verifying programs on specific inputs is the basic tenet of the theory of program checking recently formulated by Blum and Kannan [5].) More concretely, suppose we are given three $n \times n$ matrices X, Y, and Z over a field $\mathcal{F}$, and would like to verify that $XY = Z$. Clearly, it does not make sense to use simpler but slower matrix multiplication algorithm for the verification, as that would defeat the whole purpose of using the fast algorithm in the first place. Observe that, in fact, there is no need to recompute Z; rather, we are merely required to verify that the product of X and Y is indeed equal to Z. Freivalds' technique gives an elegant solution that leads to an $O(n^2)$ time randomized algorithm with bounded error probability.

The idea is to first pick a random vector $r \in \{0, 1\}^n$, i.e., each component of r is chosen independently and uniformly at random from the set $\{0, 1\}$ consisting of the additive and multiplicative identities of the field $\mathcal{F}$. Then, in $O(n^2)$ time, we can compute $y = Yr$, $x = Xy = XYr$, and $z = Zr$. We would like to claim that the identity $XY = Z$ can be verified by merely checking that $x = z$. Quite clearly, if $XY = Z$ then $x = z$; unfortunately, the converse is not true in general. However, given the random choice of r, we can show that for $XY \neq Z$, the probability that $x \neq z$ is at least $1/2$. Observe that the fingerprinting algorithm errs only if $XY \neq Z$ but x and z turn out to be equal, and this has a bounded probability.

THEOREM 12.8 *Let X, Y, and Z be $n \times n$ matrices over some field $\mathcal{F}$ such that $XY \neq Z$; further, let r be chosen uniformly at random from $\{0, 1\}^n$ and define $x = XYr$ and $z = Zr$. Then,*

$$\mathbf{Pr}[x = z] \leq 1/2.$$

PROOF Define $W = XY - Z$ and observe that W is not the all-zeroes matrix. Since $Wr = XYr - Zr = x - z$, the event $x = z$ is equivalent to the event that $Wr = 0$. Assume, without loss of generality, that the first row of W has a nonzero entry and that the nonzero entries in that row precede all the zero entries. Define the vector w as the first row of W, and assume that the first $k > 0$ entries in w are nonzero. Since the first component of Wr is $w^T r$, giving an upper bound on the probability that the inner product of w and r is zero will give an upper bound on the probability that $x = z$.

Observe that $w^T r = 0$ if and only if

$$r_1 = \frac{-\sum_{i=2}^{k} w_i r_i}{w_1}. \tag{12.9}$$

Suppose that while choosing the random vector r, we choose $r_2, \ldots, r_n$ before choosing r_1. After the values for $r_2, \ldots, r_n$ have been chosen, the right-hand side of Equation 12.9 is fixed at some value $v \in \mathcal{F}$. If $v \notin \{0, 1\}$, then r_1 will never equal v; conversely, if $v \in \{0, 1\}$, then the probability that $r_1 = v$ is $1/2$. Thus, the probability that $w^T r = 0$ is at most $1/2$, implying the desired result.

We have reduced the matrix multiplication verification problem to that of verifying the equality of two vectors. The reduction itself can be performed in $O(n^2)$ time and the vector equality can be checked in $O(n)$ time, giving an overall running time of $O(n^2)$ for this Monte Carlo procedure. The error probability can be reduced to $1/2^k$ via k independent iterations of the Monte Carlo algorithm. Note that there was nothing magical about choosing the components of the random

vector r from $\{0, 1\}$, since any two distinct elements of $\mathcal{F}$ would have done equally well. This suggests an alternative approach toward reducing the error probability, as follows: each component of r is chosen independently and uniformly at random from some subset $\mathcal{S}$ of the field $\mathcal{F}$; then, it is easily verified that the error probability is no more than $1/|\mathcal{S}|$.

Finally, note that Freivalds' technique can be applied to the verification of any matrix identity $A = B$. Of course, given A and B, just comparing their entries takes only $O(n^2)$ time. But there are many situations where, just as in the case of matrix product verification, computing A explicitly is either too expensive or possibly even impossible, whereas computing Ar is easy. The random fingerprint technique is an elegant solution in such settings.

12.8.2 Extension to Identities of Polynomials

The fingerprinting technique due to Freivalds is fairly general and can be applied to many different versions of the identity verification problem. We now show that it can be easily extended to identity verification for symbolic polynomials, where two polynomials $P_1(x)$ and $P_2(x)$ are deemed identical if they have identical coefficients for corresponding powers of x. Verifying integer or string equality is a special case since we can represent any string of length n as a polynomial of degree n by using the kth element in the string to determine the coefficient of the kth power of a symbolic variable.

Consider first the polynomial product verification problem: given three polynomials $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathcal{F}[x]$, we are required to verify that $P_1(x) \times P_2(x) = P_3(x)$. We will assume that $P_1(x)$ and $P_2(x)$ are of degree at most n, implying that $P_3(x)$ has degree at most $2n$. Note that degree n polynomials can be multiplied in $O(n \log n)$ time via fast Fourier transforms, and that the evaluation of a polynomial can be done in $O(n)$ time.

The randomized algorithm we present for polynomial product verification is similar to the algorithm for matrix product verification. It first fixes a set $\mathcal{S} \subseteq \mathcal{F}$ of size at least $2n + 1$ and chooses $r \in \mathcal{S}$ uniformly at random. Then, after evaluating $P_1(r)$, $P_2(r)$, and $P_3(r)$ in $O(n)$ time, the algorithm declares the identity $P_1(x)P_2(x) = P_3(x)$ to be correct if and only if $P_1(r)P_2(r) = P_3(r)$. The algorithm makes an error only in the case where the polynomial identity is false but the value of the three polynomials at r indicates otherwise. We will show that the error event has a bounded probability.

Consider the degree $2n$ polynomial $Q(x) = P_1(x)P_2(x) - P_3(x)$. The polynomial $Q(x)$ is said to be identically zero, denoted by $Q(x) \equiv 0$, if each of its coefficients equals zero. Clearly, the polynomial identity $P_1(x)P_2(x) = P_3(x)$ holds if and only if $Q(x) \equiv 0$. We need to establish that if $Q(x) \not\equiv 0$, then with high probability $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. By elementary algebra we know that $Q(x)$ has at most $2n$ distinct roots. It follows that unless $Q(x) \equiv 0$, not more that $2n$ different choices of $r \in \mathcal{S}$ will cause $Q(r)$ to evaluate to 0. Therefore, the error probability is at most $2n/|\mathcal{S}|$. The probability of error can be reduced either by using independent iterations of this algorithm, or by choosing a larger set $\mathcal{S}$. Of course, when $\mathcal{F}$ is an infinite field (e.g., the reals), the error probability can be made 0 by choosing r uniformly from the entire field $\mathcal{F}$; however, that requires an infinite number of random bits!

Note that we could also use a deterministic version of this algorithm where each choice of $r \in \mathcal{S}$ is tried once. But this involves $2n + 1$ different evaluations of each polynomial, and the best known algorithm for multiple evaluations needs $\Theta(n\log^2 n)$ time, which is more than the $O(n \log n)$ time requirement for actually performing a multiplication of the polynomials $P_1(x)$ and $P_2(x)$.

This verification technique is easily extended to a generic procedure for testing any polynomial identity of the form $P_1(x) = P_2(x)$ by converting it into the identity $Q(x) = P_1(x) - P_2(x) \equiv 0$. Of course, when P_1 and P_2 are explicitly provided, the identity can be deterministically verified in $O(n)$ time by comparing corresponding coefficients. Our randomized technique will take just as long to merely evaluate $P_1(x)$ and $P_2(x)$ at a random value. However, as in the case of verifying matrix identities, the randomized algorithm is quite useful in situations where the polynomials are implicitly

specified, e.g., when we only have a "black box" for computing the polynomials with no information about their coefficients, or when they are provided in a form where computing the actual coefficients is expensive. An example of the latter situation is provided by the following problem concerning the determinant of a symbolic matrix. In fact, the determinant problem will require a technique for the verification of polynomial identities of multivariate polynomials that we will discuss shortly.

Consider an $n \times n$ matrix M. Recall that the determinant of the matrix M is defined as follows:

$$\det(M) = \sum_{\pi \in \mathcal{S}_n} \text{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)}, \tag{12.10}$$

where

$\mathcal{S}_n$ is the symmetric group of permutations of order n
$\text{sgn}(\pi)$ is the sign of a permutation π

(The sign function is defined to be $\text{sgn}(\pi) = (-1)^t$, where t is the number of pairwise exchanges required to convert the identity permutation into π.) Although the determinant is defined as a summation with $n!$ terms, it is easily evaluated in polynomial time provided that the matrix entries M_{ij} are explicitly specified. Consider the Vandermonde matrix $M(x_1, \ldots, x_n)$ which is defined in terms of the indeterminates $x_1, \ldots, x_n$ such that $M_{ij} = x_i^{j-1}$, i.e.,

$$M = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix}.$$

It is known that for the Vandermonde matrix, $\det(M) = \prod_{i<j}(x_i - x_j)$. Consider the problem of verifying this identity without actually devising a formal proof. Computing the determinant of a symbolic matrix is infeasible as it requires dealing with a summation over $n!$ terms. However, we can formulate the identity verification problem as the problem of verifying that the polynomial $Q(x_1, \ldots, x_n) = \det(M) - \prod_{i<j}(x_i - x_j)$ is identically zero. Based on our discussion of Freivalds' technique, it is natural to consider the substitution of random values for each x_i. Since the determinant can be computed in polynomial time for any specific assignment of values to the symbolic variables $x_1, \ldots, x_n$, it is easy to evaluate the polynomial Q for random values of the variables. The only issue is that of bounding the error probability for this randomized test.

We now extend the analysis of Freivalds' technique for univariate polynomials to the multivariate case. But first, note that in a multivariate polynomial $Q(x_1, \ldots, x_n)$, the degree of a term is the sum of the exponents of the variable powers that define it, and the total degree of Q is the maximum over all terms of the degrees of the terms.

THEOREM 12.9 *Let $Q(x_1, \ldots, x_n) \in \mathcal{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree m. Let $\mathcal{S}$ be a finite subset of the field $\mathcal{F}$, and let $r_1, \ldots, r_n$ be chosen uniformly and independently from $\mathcal{S}$. Then,*

$$\mathbf{Pr}\left[Q(r_1, \ldots, r_n) = 0 \mid Q(x_1, \ldots, x_n) \not\equiv 0\right] \leq \frac{m}{|\mathcal{S}|}.$$

PROOF We will proceed by induction on the number of variables n. The basis of the induction is the case $n = 1$, which reduces to verifying the theorem for a univariate polynomial $Q(x_1)$ of

degree m. But we have already seen for $Q(x_1) \not\equiv 0$, the probability that $Q(r_1) = 0$ is at most $m/|\mathcal{S}|$, taking care of the basis.

We now assume that the induction hypothesis holds for multivariate polynomials with at most $n-1$ variables, where $n > 1$. In the polynomial $Q(x_1, \ldots, x_n)$ we can factor out the variable x_1 and thereby express Q as

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i P_i(x_2, \ldots, x_n),$$

where $k \leq m$ is the largest exponent of x_1 in Q. Given our choice of k, the coefficient $P_k(x_2, \ldots, x_n)$ of x_1^k cannot be identically zero. Note that the total degree of P_k is at most $m-k$. Thus, by the induction hypothesis, we conclude that the probability that $P_k(r_2, \ldots, r_n) = 0$ is at most $(m-k)/|\mathcal{S}|$.

Consider now the case where $P_k(r_2, \ldots, r_n)$ is indeed not equal to 0. We define the following univariate polynomial over x_1 by substituting the random values for the other variables in Q:

$$q(x_1) = Q(x_1, r_2, r_3, \ldots, r_n) = \sum_{i=0}^{k} x_1^i P_i(r_2, \ldots, r_n).$$

Quite clearly, the resulting polynomial $q(x_1)$ has degree k and is not identically zero (since the coefficient of x_1^k is assumed to be nonzero). As in the basis case, we conclude that the probability that $q(r_1) = Q(r_1, r_2, \ldots, r_n)$ evaluates to 0 is bounded by $k/|\mathcal{S}|$.

By the preceding arguments, we have established the following two inequalities:

$$\mathbf{Pr}[P_k(r_2, \ldots, r_n) = 0] \leq \frac{m-k}{|\mathcal{S}|};$$

$$\mathbf{Pr}[Q(r_1, r_2, \ldots, r_n) = 0 \mid P_k(r_2, \ldots, r_n) \neq 0] \leq \frac{k}{|\mathcal{S}|}.$$

Using the elementary observation that for any two events $\mathcal{E}_1$ and $\mathcal{E}_2$, $\mathbf{Pr}[\mathcal{E}_1] \leq \mathbf{Pr}[\mathcal{E}_1 \mid \overline{\mathcal{E}}_2] + \mathbf{Pr}[\mathcal{E}_2]$, we obtain that the probability that $Q(r_1, r_2, \ldots, r_n) = 0$ is no more than the sum of the two probabilities on the right-hand side of the two obtained inequalities, which is $m/|\mathcal{S}|$. This implies the desired result.

This randomized verification procedure has one serious drawback: when working over large (or possibly infinite) fields, the evaluation of the polynomials could involve large intermediate values, leading to inefficient implementation. One approach to dealing with this problem in the case of integers is to perform all computations modulo some small random prime number; it can be shown that this does not have any adverse effect on the error probability.

12.8.3 Detecting Perfect Matchings in Graphs

We close by giving a surprising application of the techniques from Section 12.8.2. Let $G(U, V, E)$ be a bipartite graph with two independent sets of vertices $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, and edges E that have one end-point in each of U and V. We define a matching n G as a collection of edges $M \subseteq E$ such that each vertex is an end-point of at most one edge in M; further, a perfect matching is defined to be a matching of size n, i.e., where each vertex occurs as an end-point of exactly one edge in M. Any perfect matching M may be put into a 1-to-1 correspondence with the permutations in $\mathcal{S}_n$, where the matching corresponding to a permutation $\pi \in \mathcal{S}_n$ is given by the collection of edges $\{(u_i, v_{\pi(i)}) \mid 1 \leq i \leq n\}$. We now relate the matchings of the graph to the determinant of a matrix obtained from the graph.

THEOREM 12.10 *For any bipartite graph $G(U, V, E)$, define a corresponding $n \times n$ matrix A as follows:*

$$A_{ij} = \begin{cases} x_{ij} & (u_i, v_j) \in E \\ 0 & (u_i, v_j) \notin E \end{cases}.$$

Let the multivariate polynomial $Q(x_{11}, x_{12}, \ldots, x_{nn})$ denote the determinant $\det(A)$. *Then, G has a perfect matching if and only if $Q \not\equiv 0$.*

PROOF We may express the determinant of A as follows:

$$\det(A) = \sum_{\pi \in \mathcal{S}_n} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \ldots A_{n,\pi(n)}.$$

Note that there cannot be any cancellation of the terms in the summation since each indeterminate x_{ij} occurs at most once in A. Thus, the determinant is not identically zero if and only if there exists some permutation π for which the corresponding term in the summation is nonzero. Clearly, the term corresponding to a permutation π is nonzero if and only if $A_{i,\pi(i)} \neq 0$ for each i, $1 \leq i \leq n$; this is equivalent to the presence in G of the perfect matching corresponding to π.

The matrix of indeterminates is sometimes referred to as the Edmonds matrix of a bipartite graph. The aforementioned result can be extended to the case of nonbipartite graphs, and the corresponding matrix of indeterminates is called the Tutte matrix. Tutte [42] first pointed out the close connection between matchings in graphs and matrix determinants; the simpler relation between bipartite matchings and matrix determinants was given by Edmonds [8].

We can turn the aforementioned result into a simple randomized procedure for testing the existence of perfect matchings in a bipartite graph (due to Lovász [21]): using the algorithm from Section 12.8.2, determine whether the determinant is identically zero or not. The running time of this procedure is dominated by the cost of computing a determinant, which is essentially the same as the time required to multiply two matrices. Of course, there are algorithms for constructing a maximum matching in a graph with m edges and n vertices in time $O(m\sqrt{n})$ (see Hopcroft and Karp [14], Micali and Vazirani [23,44], and Feder and Motwani [9]). Unfortunately, the time required to compute the determinant exceeds $m\sqrt{n}$ for small m, and so the benefit in using this randomized decision procedure appears marginal at best. However, this technique was extended by Rabin and Vazirani [31,32] to obtain simple algorithms for the actual construction of maximum matchings; although their randomized algorithms for matchings are simple and elegant, they are still slower than the deterministic $O(m\sqrt{n})$ time algorithms known earlier. Perhaps more significantly, this randomized decision procedure proved to be an essential ingredient in devising fast parallel algorithms for computing maximum matchings [19,27].

12.9 Research Issues and Summary

Perhaps the most important research issue in the area of randomized algorithms is to prove or disprove that are problems solvable in polynomial time by either Las Vegas or Monte Carlo algorithms, but cannot be solved in polynomial time by any deterministic algorithm. Another important direction for future work is to devise high quality pseudo-random number generators, which take a small seed of truly random bits and stretch it into a much longer string that can be used as the random string to fuel randomized algorithms.

12.10 Further Information

In this section we give pointers to a plethora of randomized algorithms not covered here. The reader should also note that the examples above are but a (random!) sample of the many randomized algorithms for each of the problems considered. These algorithms have been chosen to illustrate the main ideas behind randomized algorithms, rather than to represent the state of the art for these problems. The reader interested in other algorithms for these problems is referred to the book by Motwani and Raghavan [25].

Randomized algorithms also find application in a number of other areas: in load-balancing [43], approximation algorithms and combinatorial optimization [12,17,24], graph algorithms [2,16], data structures [3], counting and enumeration [38], parallel algorithms [19,20], distributed algorithms [30], geometric algorithms [26], online algorithms [4,33], and number-theoretic algorithms [29,40] (see also [1]). The reader interested in these applications may consult these articles or the book by Motwani and Raghavan [25].

Defining Terms

Deterministic algorithm: An algorithm whose execution is completely determined by its input.

Distributional complexity: The expected running time of the best possible deterministic algorithm over the worst possible probability distribution on the inputs.

Las Vegas algorithm: A randomized algorithm that always produces correct results, with the only variation from one run to another being in its running time.

Monte Carlo algorithm: A randomized algorithm that may produce incorrect results, but with bounded error probability.

Randomized algorithm: An algorithm that makes random choices during the course of its execution.

Randomized complexity: The expected running time of the best possible randomized algorithm over the worst input.

Acknowledgment

Supported in part by NSF Grant ITR-0331640, TRUST (NSF award number CCF-0424422), and grants from Cisco, Google, KAUST, Lightspeed, and Microsoft.

References

1. Agrawal, M., Kayal, N., and Saxena, N., PRIMES is in P. *Annals of Mathematics,* 160(2), 781–793, 2004.
2. Aleliunas, R., Karp, R.M., Lipton, R.J. Lovász, L., and Rackoff, C., Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pp. 218–223, San Juan, Puerto Rico, Oct. 1979.
3. Aragon, C.R. and Seidel, R.G., Randomized search trees. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science,* pp. 540–545, Duke, NC, Oct. 1989.
4. Ben-David, S., Borodin, A., Karp, R.M., Tardos, G., and Wigderson, A., On the power of randomization in on-line algorithms. *Algorithmica,* 11(1), 2–14, 1994.
5. Blum, M. and Kannan, S., Designing programs that check their work. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing,* pp. 86–97, ACM, New York, May 1989.

6. Coppersmith, D. and Winograd, S., Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation,* 9, 251–280, 1990.
7. DeMillo, R.A. and Lipton, R.J., A probabilistic remark on algebraic program testing. *Information Processing Letters,* 7, 193–195, 1978.
8. Edmonds, J., Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards, 71B,* 4, 241–245, 1967.
9. Feder, T. and Motwani, R., Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing,* pp. 123–133, ACM, New York, 1991.
10. Floyd, R.W. and Rivest, R.L., Expected time bounds for selection. *Communications of the ACM,* 18, 165–172, 1975.
11. Freivalds, R., Probabilistic machines can use less running time. In *Information Processing 77, Proceedings of IFIP Congress 77,* pp. 839–842, Gilchrist, B., Ed., North-Holland Publishing Company Amsterdam, the Netherlands, Aug. 1977.
12. Goemans, M.X. and Williamson, D.P., 0.878-approximation algorithms for MAX-CUT and MAX-2SAT. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing,* pp. 422–431, ACM, New York, 1994.
13. Hoare, C.A.R., Quicksort. *Computer Journal,* 5, 10–15, 1962.
14. Hopcroft, J.E. and Karp, R.M., An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing,* 2, 225–231, 1973.
15. Karger, D.R., Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms,* pp. 21–30, SIAM, Philadelphia, PA, 1993.
16. Karger, D.R., Klein, P.N., and Tarjan, R.E., A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM,* 42, 321–328, 1995.
17. Karger, D., Motwani, R., and Sudan, M., Approximate graph coloring by semidefinite programming. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science,* pp. 2–13, ACM, New York, 1994.
18. Karp, R.M., An introduction to randomized algorithms. *Discrete Applied Mathematics,* 34, 165–201, 1991.
19. Karp, R.M., Upfal, E., and Wigderson, A., Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica,* 6, 35–48, 1986.
20. Karp, R.M., Upfal, E., and Wigderson, A., The complexity of parallel search. *Journal of Computer and System Sciences,* 36, 225–253, 1988.
21. Lovász, L., On determinants, matchings and random algorithms. In *Fundamentals of Computing Theory,* Budach, L., Ed., Akademia-Verlag, Berlin, Germany, 1979.
22. Maffioli, F., Speranza, M.G., and Vercellis, C., Randomized algorithms. In *Combinatorial Optimization: Annotated Bibliographies,* pp. 89–105, O'hEigertaigh, M., Lenstra, J.K., and Rinooy Kan, A.H.G., Eds., John Wiley & Sons, New York, 1985.
23. Micali, S. and Vazirani, V.V., An $O(\sqrt{|V|}|e|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science,* pp. 1 7–27, Syracuse, NY, 1980.
24. Motwani, R., Naor, J., and Raghavan, P., Randomization in approximation algorithms. In *Approximation Algorithms,* Hochbaum, D., Ed., PWS, Boston, MA, 1996.
25. Motwani, R. and Raghavan, P., *Randomized Algorithms.* Cambridge University Press, New York, 1995.
26. Mulmuley, K., *Computational Geometry: An Introduction through Randomized Algorithms.* Prentice Hall, New York, 1993.
27. Mulmuley, K., Vazirani, U.V., and Vazirani, V.V., Matching is as easy as matrix inversion. *Combinatorica,* 7, 105–113, 1987.

28. Pugh, W., Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM,* 33(6), 668–676, 1990.
29. Rabin, M.O., Probabilistic algorithm for testing primality. *Journal of Number Theory,* 12, 128–138, 1980.
30. Rabin, M.O., Randomized Byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science,* pp. 403–409, IEEE, New York, 1983.
31. Rabin, M.O. and Vazirani, V.V., Maximum matchings in general graphs through randomization. Technical Report TR-15-84, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1984.
32. Rabin, M.O. and Vazirani, V.V., Maximum matchings in general graphs through randomization. *Journal of Algorithms,* 10, 557–567, 1989.
33. Raghavan, P. and Snir, M., Memory versus randomization in on-line algorithms. *IBM Journal of Research and Development,* 38, 683–707, 1994.
34. Saks, M. and Wigderson, A., Probabilistic Boolean decision trees and the complexity of evaluating game trees. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science,* pp. 29–38, Toronto, ON, Canada, 1986.
35. Schrijver, A., *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.
36. Schwartz, J.T., Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM,* 27(4), 701–717, 1980.
37. Seidel, R.G., Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry,* 6, 423–434, 1991.
38. Sinclair, A., *Algorithms for Random Generation and Counting: A Markov Chain Approach. Progress in Theoretical Computer Science.* Birkhäuser, Boston, MA, 1992.
39. Snir, M., Lower bounds on probabilistic linear decision trees. *Theoretical Computer Science,* 38, 69–82, 1985.
40. Solovay, R. and Strassen, V., A fast Monte-Carlo test for primality. *SIAM Journal on Computing,* 6(1), 84–85, 1977. See also *SIAM Journal on Computing, 7,* 118, Feb. 1, 1978.
41. Tarsi, M., Optimal search on some game trees. *Journal of the ACM,* 30, 389–396, 1983.
42. Tutte, W.T., The factorization of linear graphs. *Journal of the London Mathematical Society,* 22, 107–111, 1947.
43. Valiant, L.G., A scheme for fast parallel communication. *SIAM Journal on Computing,* 11, 350–361, 1982.
44. Vazirani, V.V., A theory of alternating paths and blossoms for proving correctness of $O(\sqrt{V}E)$ graph maximum matching algorithms. *Combinatorica,* 14(1), 71–109, 1994.
45. Welsh, D.J.A., Randomised algorithms. *Discrete Applied Mathematics,* 5, 133–145, 1983.
46. Yao, A.C-C., Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science,* pp. 222–227, Providence, Rhode Island, 1977.
47. Zippel, R.E., Probabilistic algorithms for sparse polynomials. In *Proceedings of EUROSAM 79,* volume 72 of *Lecture Notes in Computer Science,* pp. 216–226, Marseille, France 1979.