CHAPTER 2

Game-Theoretic Techniques

In this chapter we study several ideas that are basic to the design and analysis of randomized algorithms. All the topics in this chapter share a game-theoretic viewpoint, which enables us to think of a randomized algorithm as a probability distribution on deterministic algorithms. This leads to the *Yao's Minimax Principle*, which can be used to establish a lower bound on the performance of a randomized algorithm.

2.1. Game Tree Evaluation

We begin with another simple illustration of linearity of expectation, in the setting of *game tree evaluation*. This example will demonstrate a randomized algorithm whose expected running time is smaller than that of any deterministic algorithm. It will also serve as a vehicle for demonstrating a standard technique for deriving a *lower bound* on the running time of *any* randomized algorithm for a problem.

A *game tree* is a rooted tree in which internal nodes at even distance from the root are labeled MIN and internal nodes at odd distance are labeled MAX. Associated with each leaf is a real number, which we call its *value*. The *evaluation* of the game tree is the following process. Each leaf *returns* the value associated with it. Each MAX node returns the largest value returned by its children, and each MIN node returns the smallest value returned by its children. Given a tree with values at the leaves, the evaluation problem is to determine the value returned by the root.

The evaluation of game trees plays a central role in artificial intelligence, particularly in game-playing programs. The reader may readily associate the children of a node with the options available to one of the two players in a game. The leaves represent the value of the game for either player. One player seeks to maximize this value, while the other tries to minimize it. At each step, an evaluation algorithm chooses a leaf and reads its value.

We study the number of such steps taken by an algorithm for evaluating a game tree. We do not charge the algorithm for any other computation.

We will limit our discussion to the special case in which the values at the leaves are bits, 0 or 1. Thus, each MIN node can be thought of as a Boolean AND operation and each MAX node as a Boolean OR operation. This special case is of interest in its own right, having applications in mechanical theorem proving. Let $T_{d,k}$ denote a uniform tree in which the root and every internal node has d children and 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 d^{2k} leaves. An *instance* of the evaluation problem consists of the tree $T_{d,k}$ together with a Boolean value for each of the d^{2k} leaves. Given an algorithm, we study the maximum number of steps it takes to evaluate any instance of $T_{d,k}$.

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.

In Problem 2.1, the reader is asked to show that for any deterministic evaluation algorithm, there is an instance of $T_{d,k}$ that forces the algorithm to read the values on all d^{2k} leaves.

We now give a simple randomized algorithm and study the expected number of leaves it reads on any instance of $T_{d,k}$. To simplify our presentation, we restrict ourselves to the case $d = 2$. Any deterministic algorithm for this case can be made to read all $2^{2k} = 4^k$ leaves on some instance of $T_{2,k}$. Our randomized algorithm is based on 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 $3/2$.

The reader may wonder how the randomized algorithm can benefit if the AND node were to return 1, or if the OR node were to return a 0. If the two children of these nodes are leaves, then clearly both leaves must be examined. The point is that at an internal AND node in a tree returning a 1, examining the two OR children (and evaluating their sub-trees) in a random order is still beneficial. The two OR children of an AND node must also return 1, and this is the easy case for the OR nodes. Similarly, at an internal OR node returning 0, the two AND children must return 0, and this is the easy case for the AND nodes. To explain this better, we specify the complete algorithm.

To evaluate an AND node v, the algorithm chooses one of its children (a sub-tree rooted at an OR node) at random and evaluates it by recursively invoking the algorithm. If 1 is returned by the sub-tree, 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 now argue by induction on k that the expected cost of evaluating any instance of $T_{2,k}$ is at most 3^k.

The basis ($k = 1$) is an easy extension of our illustration above. Assume now that the expected cost of evaluating any instance of $T_{2,k-1}$ is at most 3^{k-1}. We establish the inductive step. Consider first a tree T whose root is an OR node, each of whose children is the root of a copy of $T_{2,k-1}$. 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 at most 1/2 both sub-trees are evaluated, incurring a net cost of at most $2 \times 3^{k-1}$. Putting these observations together, the expected cost of determining the value of T is at most

$$\frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1}. \tag{2.1}$$

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

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

Here the first term bounds the cost of evaluating both sub-trees of the OR node that returns 0; the second term accounts for the fact that with probability 1/2, an additional cost of $(3/2)3^{k-1}$ may be incurred in evaluating its sibling that returns 1.

Theorem 2.1: *Given any instance of $T_{2,k}$, the expected number of steps for the above randomized algorithm is at most 3^k.*

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

	Scissors	Paper	Stone
Scissors	0	1	−1
Paper	−1	0	1
Stone	1	−1	0

Figure 2.1: Matrix for scissors-paper-stone.

2.2. The Minimax Principle

The randomized algorithm of the preceding section has an expected running time of at most $n^{0.793}$ 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 are thus seeking a lower bound on the running time of any randomized algorithm for this problem. As a first step toward this end, we introduce a standard technique for proving such lower bounds: the *minimax principle.* Indeed, it is the only known general technique for proving lower bounds on the running times of randomized algorithms. This technique only applies to algorithms that terminate in finite time on all inputs and sequences of random choices. In Section 2.2.3, we will apply this technique to the game tree evaluation problem. We begin with a review of some elementary concepts in game theory. Note that the notion of *game* theory is not directly related to the *game* tree evaluation problem studied above. Rather, the game theory studied below yields the minimax principle, a general tool, which we will then apply to randomized algorithms for the game tree evaluation problem.

2.2.1. Game Theory

Consider the following game. Roberta and Charles put their hands behind their backs and make a sign for one of the following: stone (closed fist), paper (open palm), and scissors (two fingers). They then simultaneously display their chosen sign. The winner is determined by the following rules: paper beats stone by wrapping it, scissors beats paper by cutting it, and stone beats scissors by dulling it. The loser pays \$1 to the winner, and the outcome is a draw when the two players choose the same sign. We can represent this game by the matrix in Figure 2.1. The rows of the matrix represent Roberta's choices; the columns, Charles' choices. The entries in the matrix are the amounts to be paid by Charles to Roberta.

This is an instance of a *two-person zero-sum game*, and the matrix is called the *payoff matrix.* It is called a zero-sum game because the net amount won by Roberta and Charles is always exactly zero. In general, any two-person zero-sum game can be represented by an $n \times m$ payoff matrix $\mathbf{M}$ with real entries. (Throughout this book, we use boldface to denote vectors and matrices;

	Scissors	Paper	Stone
Scissors	0	1	2
Paper	−1	0	1
Stone	−2	−1	0

Figure 2.2: Matrix for modified scissors-paper-stone.

generally, vectors will be lower-case symbols, and matrices upper-case symbols. For a vector $\boldsymbol{x}$, we denote by x_i its ith component. All vectors are column vectors unless otherwise specified.) The set of possible strategies of the row player R is in correspondence with the rows of $\boldsymbol{M}$, and likewise for the strategies of the column player C. The entry M_{ij} is the amount paid by C to R when R chooses strategy i and C chooses strategy j.

Naturally, the goal of the row (column) player is to maximize (minimize) the payoff. Assume that this is a zero-information game, in that neither player has any information about the opponent's strategy. If R chooses strategy i, then she is guaranteed a payoff of $\min_j M_{ij}$, regardless of C's strategy. An *optimal strategy* for R is an i that maximizes $\min_j M_{ij}$. Let $V_R = \max_i \min_j M_{ij}$ denote the lower bound on the value of the payoff to R when she uses an optimal strategy. An optimal strategy for C is a j that gives the best possible upper bound on the payoff from C to R. A similar argument establishes that C's optimal strategy ensures that his payoff to R is at most $V_C = \min_j \max_i M_{ij}$.

Exercise 2.1: Show that the following inequality is valid for all payoff matrices.

$$\max_i \min_j M_{ij} \leq \min_j \max_i M_{ij}.$$

In general, the inequality in Exercise 2.1 is strict; for example, in scissors-paper-stone, $V_R = -1$ and $V_C = 1$. When these two quantities are equal, the game is said to have a solution and the *value* of the game is $V = V_R = V_C$. The solution (or the *saddle-point*) is the specific choice of (optimal) strategies that lead to this payoff. For games with a solution, let ρ and γ denote optimal strategies for R and C, respectively; clearly, $V = M_{\rho\gamma}$. In general, a player could have more than one optimal strategy.

Figure 2.2 shows a modified version of the scissors-paper-stone game, where the amount to be paid in certain cases is changed. It is easy to verify that this game has value $V = 0$ and the solution is $\rho = 1$ and $\gamma = 1$. (Do you see why the other diagonal entries do not correspond to saddle-points?)

What happens when a game has no solution? Then there is no clear-cut optimal strategy for any player. In fact, any knowledge of the opponent's strategy can be used to improve the payoff, unlike the case of games with saddle-points. An interesting way to get around this is to introduce randomization in the choice of strategies. So far we have been talking about deterministic or *pure* strategies, but now we focus on randomized or *mixed* strategies. A mixed strategy is a probability distribution on the set of possible strategies. The row player picks a vector $\boldsymbol{p} = (p_1, \ldots, p_n)$, which is a probability distribution on the rows of $\boldsymbol{M}$, i.e., p_i is the probability that R will choose strategy i; similarly, the column player has a vector $\boldsymbol{q} = (q_1, \ldots, q_m)$, which is a probability distribution on the columns of $\boldsymbol{M}$. The payoff is now a random variable, and its expectation is given by

$$\mathrm{E}[\text{payoff}] = \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q} = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i M_{ij} q_j.$$

As before, using V_R to denote the best possible lower bound on the expected payoff to R that can be ensured by choosing a strategy $\boldsymbol{p}$, and using V_C to denote the best possible upper bound on the expected payoff by C by choosing a strategy $\boldsymbol{q}$, we obtain

$$\begin{aligned} V_R &= \max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q} \\ V_C &= \min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q}. \end{aligned}$$

Here, the min and max range over all possible distributions. The well-known Minimax Theorem of von Neumann implies that this game always has a solution and that $V_R = V_C$.

Theorem 2.2 (von Neumann's Minimax Theorem): *For any two-person zero-sum game specified by a matrix* $\boldsymbol{M}$,

$$\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q} = \min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q}.$$

In other words, the largest expected payoff that R can guarantee by choosing a mixed strategy is equal to the smallest expected payoff that C can guarantee using a mixed strategy. This common expected payoff value, called the value of the game, is denoted by V. A pair of mixed strategies $(\widehat{\boldsymbol{p}}, \widehat{\boldsymbol{q}})$ which respectively maximize the left-hand side and minimize the right-hand side of the equation in Theorem 2.2 is called a saddle-point, and the two distributions are called optimal mixed strategies.

Observe that once $\boldsymbol{p}$ is fixed, $\boldsymbol{p}^T\boldsymbol{M}\boldsymbol{q}$ is a linear function of $\boldsymbol{q}$ and is minimized by setting to 1 the q_j with the smallest coefficient in this linear function. The implications of this observation are rather interesting. If C knows the distribution $\boldsymbol{p}$ being used by R, then his optimal strategy is a pure strategy. A similar comment applies in the other direction. Also, this observation leads to a simplified version of the minimax theorem. Let $\boldsymbol{e}_k$ denote a unit vector with a 1 in the kth position and 0s elsewhere.

Theorem 2.3 (Loomis' Theorem): *For any two-person zero-sum game specified by a matrix* $\boldsymbol{M}$,

$$\max_{\boldsymbol{p}} \min_{j} \boldsymbol{p}^T\boldsymbol{M}\boldsymbol{e}_j = \min_{\boldsymbol{q}} \max_{i} \boldsymbol{e}_i^T\boldsymbol{M}\boldsymbol{q}.$$

2.2.2. Yao's Technique

We now describe the application of the above game-theoretic results to proving lower bounds on the performance of randomized algorithms. The idea is to view the algorithm designer as the column player C and the adversary choosing the input as the row player R. The columns correspond to the set of all possible algorithms; the rows correspond to the set of all possible inputs (of a fixed size). It is important to keep in mind that each column corresponds to a deterministic algorithm that always produces a correct solution. The payoff from C to R is some real-valued measure of the performance of an algorithm, such as the running time, the quality of the solution obtained, communication cost, or space. (In all the examples we will encounter in this book, the entries in the payoff matrix will be positive integers.) For the sake of concreteness, we assume in this chapter that the payoff refers to the running time, but it should be obvious that the following observations apply to any other measure. The algorithm designer would like to choose an algorithm that minimizes the payoff, while the adversary would like to maximize the payoff.

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. A pure strategy for C corresponds to the choice of a deterministic algorithm, while a pure strategy for R corresponds to a specific input. Notice that an optimal pure strategy for C corresponds to an optimal deterministic algorithm, and V_C is the worst-case running time of any deterministic algorithm for the problem, which we call the deterministic complexity of the problem. (The meaning of V_R is related to the non-deterministic complexity of the problem. If the game has a solution, then the non-deterministic and deterministic complexities coincide.)

Our interest is in the interpretation of the mixed strategies for the algorithm designer and the adversary. A mixed strategy for C is a probability distribution over the space of (always correct) deterministic algorithms, so it is a Las Vegas randomized algorithm. An optimal mixed strategy for C is an optimal Las Vegas algorithm. A mixed strategy for R is a distribution over the space of all inputs.

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. This complexity is smaller than the deterministic complexity, since the algorithm knows the input distribution.

Theorem 2.3 implies that the distributional complexity equals the least possible expected running time achievable by any randomized algorithm. (We reiterate that these observations apply only to scenarios where the number of algorithms is finite.) We restate von Neumann's and Loomis's theorems in the language of algorithms as follows.

Corollary 2.4: *Let Π be a problem with a finite set $\mathcal{I}$ of input instances (of a fixed size), and a finite set of deterministic algorithms $\mathcal{A}$. For input $I \in \mathcal{I}$ and algorithm $A \in \mathcal{A}$, let $C(I, A)$ denote the running time of algorithm A on input I.*

For probability distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$, let $I_{\boldsymbol{p}}$ denote a random input chosen according to $\boldsymbol{p}$ and $A_{\boldsymbol{q}}$ denote a random algorithm chosen according to $\boldsymbol{q}$. Then,

$$\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \mathbf{E}[C(I_{\boldsymbol{p}}, A_{\boldsymbol{q}})] = \min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \mathbf{E}[C(I_{\boldsymbol{p}}, A_{\boldsymbol{q}})]$$

and

$$\max_{\boldsymbol{p}} \min_{A \in \mathcal{A}} \mathbf{E}[C(I_{\boldsymbol{p}}, A)] = \min_{\boldsymbol{q}} \max_{I \in \mathcal{I}} \mathbf{E}[C(I, A_{\boldsymbol{q}})].$$

From this corollary, we obtain the following proposition, which provides the desired lower bound technique.

Proposition 2.5 (Yao's Minimax Principle): *For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$,*

$$\min_{A \in \mathcal{A}} \mathbf{E}[C(I_{\boldsymbol{p}}, A)] \leq \max_{I \in \mathcal{I}} \mathbf{E}[C(I, A_{\boldsymbol{q}})].$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution $\boldsymbol{p}$ 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 $\boldsymbol{p}$ 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 $\boldsymbol{p}$ 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 $\boldsymbol{p}$.

The above discussion dealt only with lower bounds on the performance of Las Vegas algorithms. We conclude this section with a brief discussion of Monte Carlo algorithms with error probability $\epsilon \in [0, 1/2]$. Let us define the distributional complexity with error ϵ, denoted $\min_{A \in \mathcal{A}} \mathbf{E}[C_\epsilon(I_{\boldsymbol{p}}, A)]$, to be the minimum expected running time of any deterministic algorithm that errs with probability at most ϵ under the input distribution $\boldsymbol{p}$. Similarly, we denote by $\max_{I \in \mathcal{I}} \mathbf{E}[C_\epsilon(I, A_{\boldsymbol{q}})]$ 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 $\boldsymbol{q}$ on deterministic algorithms). Analogous to Proposition 2.5, we then have:

Proposition 2.6: *For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$ and any $\epsilon \in [0, 1/2]$,*

$$\frac{1}{2}(\min_{A \in \mathcal{A}} \mathbf{E}[C_{2\epsilon}(I_{\boldsymbol{p}}, A)]) \leq \max_{I \in \mathcal{I}} \mathbf{E}[C_\epsilon(I, A_{\boldsymbol{q}})].$$

A pointer to the source of Proposition 2.6 is given in the Notes section.

2.2.3. Lower Bound for Game Tree Evaluation

We now apply Yao's Minimax Principle to the problem of game tree evaluation. The lower bound that results only applies to algorithms that terminate in a finite number of steps on any input and sequence of random choices. Note that a randomized algorithm for game tree evaluation can in fact 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.

Once again, we limit our attention to instances of the AND-OR tree $T_{2,k}$. While we could continue our discussion in the language of alternating levels of AND and OR nodes, the following exercise will lead to a slightly more compact representation.

Exercise 2.2: Show that the tree $T_{2,k}$ 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 $2k$. In order to prove a lower bound on the expected number of leaves evaluated by any randomized algorithm, we have to specify a distribution on instances (values for the leaves), and then prove a lower bound on the expected running time of any deterministic algorithm on such inputs. It is important to distinguish between the expected running time of the randomized algorithm (which is over the random choices made by the algorithm), and the expected running time of the deterministic algorithm when proving the lower bound (this being over the random instances). We also remind the reader that our lower bound will only apply to Las Vegas randomized algorithms that always evaluate the tree correctly.

Let $p = (3 - \sqrt{5})/2$. Each leaf of the tree is independently set to 1 with probability p. Note that if each input to a NOR node is independently 1 with probability p, then the probability that its output is 1 is the probability that both its inputs are 0, which is

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

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 sub-tree. By doing so, it can try to maximize the benefit of information obtained by inspecting leaves. An alternative view of this process is that the deterministic algorithm inspects leaves visited in a depth-first search of the tree, except of course that it ceases to visit sub-trees of a node v once the value of v has been determined. Let us call such algorithms *depth-first pruning* algorithms, referring to the order of traversal and the fact that sub-trees that supply no additional information are "pruned" away without being inspected.

Proposition 2.7: *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 $W(T)$.*

A formal proof of Proposition 2.7 by induction is omitted here and can be found in the reference given at the end of this chapter.

Proposition 2.7 tells us that for the purposes of our lower bound, we may restrict our attention to depth-first pruning algorithms. We return to a NOR tree with n leaves, each of which is set to 1 independently with probability $p = (3 - \sqrt{5})/2$. For a depth-first pruning algorithm evaluating this tree, let $W(h)$ be the expected number of leaves it inspects in determining the value of a node at distance h from the leaves. 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 sub-trees of the node, and the second term represents the work done in evaluating the other sub-tree (which will be necessary if the first sub-tree 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 2.8: *The expected running time of any randomized algorithm that always evaluates an instance of $T_{2,k}$ correctly is at least $n^{0.694}$, where $n = 2^{2k}$ is the number of leaves.*

We note that our lower bound of $n^{0.694}$ is less than the upper bound of $n^{0.793}$ that follows from Theorem 2.1. Could it be that our lower bound technique is weak? Corollary 2.4 precludes this possibility, since the identity it gives is an equality; thus for any lower bound on the expected running time there must be a distribution on the inputs such that the running time of the best deterministic algorithm matches this lower bound. One possibility 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 sub-trees of that node. Thus, to prove the best lower bound, we have to choose a distribution on the inputs that precludes the possibility 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 shows that our algorithm of Section 2.1 is optimal.

2.3. Randomness and Non-uniformity

A basic issue in the study of randomized algorithms is the extent to which randomization is necessary for solving a problem. When is it possible to remove the randomization in a randomized algorithm? The answer depends on a number of aspects of the problem being solved. The goal of this section is to show that this question is more subtle than appears at first, and touches on the issue of *uniformity* in algorithms. We now study the notion of a randomized circuit, and a general technique by which randomization can be removed in polynomial-sized randomized circuits.

A *Boolean circuit with n inputs* is a directed acyclic graph with the following properties:

1. There are n vertices of in-degree 0; these are called the *inputs* to the circuit and are labeled $x_1, x_2, \ldots, x_n$. There is one vertex with out-degree 0; this is called the *output* of the circuit.
2. Every vertex v that is not an input or the output is labeled with one Boolean function $b(v)$ from the set {AND, OR, NOT}. A vertex labeled NOT has in-degree 1.
3. Every input to the circuit is assigned a Boolean value. Under such an assignment of input values, each vertex v computes the Boolean function $b(v)$ of the values on the incoming edges, and assigns this value to its outgoing edges. The value of the output is thus a Boolean function of $x_1, x_2, \ldots, x_n$; the circuit is said to compute this function.
4. The *size* of a circuit is the number of vertices in it.

A *randomized circuit* is very similar, except that there may be more than n vertices of in-degree 0, and these are partitioned into two classes: (1) *random inputs*, each of which is assigned an independent random value from $\{0, 1\}$, and (2) the *n circuit inputs*, which are labeled $x_1, x_2, \ldots, x_n$. A randomized circuit is said to compute a function f of the inputs $x_1, x_2, \ldots, x_n$ if the following properties hold:

1. For inputs $x_1, x_2, \ldots, x_n$ for which $f(x_1, \ldots, x_n) = 0$, the output of the circuit is 0 regardless of the values of random inputs.
2. If, on the other hand, $f(x_1, \ldots, x_n) = 1$, the output of the circuit is 1 with probability at least 1/2.

Consider a Boolean function $f : \{0, 1\}^* \to \{0, 1\}$. We denote by f_n the function f restricted to inputs from $\{0, 1\}^n$. A sequence $C = C_1, C_2, \ldots$ of circuits is a *circuit family for f* if C_n has n inputs and computes $f_n(x_1, x_2, \ldots, x_n)$ at its output for all n-bit inputs $(x_1, \ldots, x_n)$. The family C is said to be *polynomial-sized* if the size of C_n is bounded above by $p(n)$ for every n, where $p(.)$ is a polynomial. A *randomized circuit family for f* is a circuit family for f that, in addition to the n inputs $x_1, \ldots, x_n$, takes m random bits $r_1, \ldots, r_m$, each of which is equiprobably 0 or 1. In addition, for every n, circuit C_n must satisfy two properties:

1. If $f_n(x_1, \ldots, x_n) = 0$, then the output of the circuit is 0 regardless of the values of random inputs $r_1, \ldots, r_m$.
2. If $f_n(x_1, \ldots, x_n) = 1$, then the output of the circuit is 1 with probability at least 1/2. In other words, at least one half of the 2^m choices of the bits $r_1, \ldots, r_m$ will result in the circuit evaluating to 1. We will refer to such m-tuples $r_1, \ldots, r_m$ as *witnesses* for $(x_1, \ldots, x_n)$, in that they testify to the correct value of $f_n(x_1, \ldots, x_n)$ when it is 1.

Theorem 2.9 below asserts that randomization can be eliminated in polynomial-sized circuits.

Theorem 2.9 (Adleman's Theorem): *If a Boolean function has a randomized, polynomial-sized circuit family, then it has a polynomial-sized circuit family.*

PROOF: The proof is by a simple counting argument.

We show how to turn a given randomized polynomial-sized circuit C_n for $f_n(x_1, \ldots, x_n)$ using random inputs $r_1, \ldots, r_m$, into a deterministic polynomial-sized circuit D_n that computes $f_n(x_1, \ldots, x_n)$.

Form a matrix $\boldsymbol{M}$ with 2^n rows, one for each possible input from $\{0, 1\}^n$. The matrix has 2^m columns, one for each of the possible m-tuples from $\{0, 1\}^m$ that the r_i can assume. The entry M_{jk} is 1 if the setting of the $r_1, \ldots, r_m$ corresponding to column k is a witness for the input $x_1, \ldots, x_n$ corresponding to row j; otherwise, the entry is 0. Eliminate all rows of $\boldsymbol{M}$ corresponding to inputs for which f_n evaluates to 0.

By definition, at least half the entries of every surviving row of $\boldsymbol{M}$ equal 1. Therefore, there must be a column with at least half its entries 1; in other words, there is an assignment of 0s and 1s to the r_i that serves as a witness to at least half of the possible inputs. Let this witness be $r_1(1), \ldots, r_m(1)$. Build a circuit T_1, which is a copy of C_n with the random inputs "hard-wired" to $r_1(1), \ldots, r_m(1)$. Delete the column in $\boldsymbol{M}$ corresponding to $r_1(1), \ldots, r_m(1)$, and all rows that had 1s in this column. Thus T_1 computes the correct value of $f_n(x_1, \ldots, x_n)$ whenever the input corresponds to one of the rows we have just eliminated.

The matrix that remains still has the property that every row has at least half its entries equal to 1, since the string $r_1(1), \ldots, r_m(1)$ was not a witness for any of these rows whereas half the entries in these rows are guaranteed to be 1s. Repeat the construction above, picking a second string $r_1(2), \ldots, r_m(2)$ that is a witness for at least half the remaining inputs and building a circuit T_2. Continuing in this manner, we will have deleted all the rows of $\boldsymbol{M}$ while building at most n circuits $T_1, \ldots, T_n$.

Now we take the OR of the outputs of the circuits $T_1, \ldots, T_n$, and this is a (deterministic) circuit whose size is $n + 1$ times that of the randomized circuit we started with. □

The technique in Theorem 2.9 is the first example we have seen of *derandomization* – where we take a randomized algorithm or computation, and diminish or entirely remove the randomness in it. This is often a useful technique for the design of deterministic algorithms. Does Theorem 2.9 mean that randomization is dispensable in all polynomial-time computations? The answer is no, and has to do with the issue of non-uniformity in computation. The deterministic circuit generated by the above process is one that works for a particular value of n. Indeed, the circuit it produces for n inputs may have very little resemblance to the circuit it produces for $n+1$ inputs, even if the original randomized circuits were similar. Any "practical" algorithm or circuit will in fact exhibit this property of similarity, which is formalized in the literature under the name *uniformity*.

Complexity theory formalizes this intuition by classifying algorithms as being *uniform* or *non-uniform* as follows. Let $a(n)$ be a function from the positive integers to strings in Σ^*. An algorithm A is said to use *advice a* if on an input of length n it is given the string $a(n)$ on a read-only tape. We say that A decides a language L with advice a if on an input x it uses the read-only string $a(|x|)$ to decide the membership of x in L. In other words, a single advice string $a(n)$ enables the algorithm A to decide the membership of x in L for all inputs x having length n. Uniform algorithms are those that use no advice strings at all, whereas non-uniform algorithms are those that use such advice. For the complexity class $\boldsymbol{P}$, we define the class $\boldsymbol{P}/\text{poly}$ to consist of all languages L that have a non-uniform polynomial-time algorithm A such that the length of the advice string $a(n)$ is bounded by a polynomial in n. Likewise, we may define the class $\boldsymbol{RP}/\text{poly}$.

Exercise 2.3: Consider any language $L \subseteq \{0,1\}^*$. We define a Boolean function f corresponding to the language L as follows. For any positive integer n, let f_n be the Boolean function such that for any $x \in \{0,1\}^n$, $f_n(x)$ assumes the value 1 if $x \in L$ and 0 otherwise. If there is a circuit family for f, we refer to it as a circuit family for L. Show that $L \in \boldsymbol{P}/\text{poly}$ if and only if it has a polynomial-sized circuit family.

In an analogous fashion, we may speak of a language L as having a randomized circuit family. Clearly, $L \in \boldsymbol{RP}/\text{poly}$ if and only if it has a randomized polynomial-sized circuit family. In the light of this discussion, we may interpret Theorem 2.9 as proving that $\boldsymbol{RP}/\text{poly} \subseteq \boldsymbol{P}/\text{poly}$. We thus have:

Corollary 2.10: *$\boldsymbol{RP} \subseteq \boldsymbol{P}/poly$.*

In summary, the removal of randomness in Theorem 2.9 only shows that this can be done in principle; it is not known how to do this in any uniform or practical way.

Notes

The material of Section 2.1 is based on a paper of Snir [381].

Most of the material in Section 2.2 is covered in textbooks on game theory. Some good sources are the books by Wang [213], Luce and Raiffa [287], and von Neumann and Morgenstern [411]. Theorem 2.2 is due to von Neumann [408], and Theorem 2.3