5 Probabilistic Analysis and Randomized Algorithms

This chapter introduces probabilistic analysis and randomized algorithms. If you are unfamiliar with the basics of probability theory, you should read Appendix C, which reviews this material. Probabilistic analysis and randomized algorithms will be revisited several times throughout this book.

5.1 The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency will send you one candidate each day. You will interview that person and then decide to either hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a large hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode. It assumes that the candidates for the office assistant job are numbered 1 through n. The procedure assumes that you are able to, after interviewing candidate i, determine if candidate i is the best candidate you have seen so far. To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
1  best ← 0        ▷ candidate 0 is a least-qualified dummy candidate
2  for i ← 1 to n
3      do interview candidate i
4         if candidate i is better than candidate best
5           then best ← i
6                hire candidate i
```

The cost model for this problem differs from the model described in Chapter 2. We are not concerned with the running time of HIRE-ASSISTANT, but instead with the cost incurred by interviewing and hiring. On the surface, analyzing the cost of this algorithm may seem very different from analyzing the running time of, say, merge sort. The analytical techniques used, however, are identical whether we are analyzing cost or running time. In either case, we are counting the number of times certain basic operations are executed.

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Let m be the number of people hired. Then the total cost associated with this algorithm is $O(nc_i + mc_h)$. No matter how many people we hire, we always interview n candidates and thus always incur the cost nc_i associated with interviewing. We therefore concentrate on analyzing mc_h, the hiring cost. This quantity varies with each run of the algorithm.

This scenario serves as a model for a common computational paradigm. It is often the case that we need to find the maximum or minimum value in a sequence by examining each element of the sequence and maintaining a current "winner." The hiring problem models how often we update our notion of which element is currently winning.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation occurs if the candidates come in increasing order of quality, in which case we hire n times, for a total hiring cost of $O(nc_h)$.

It might be reasonable to expect, however, that the candidates do not always come in increasing order of quality. In fact, we have no idea about the order in which they arrive, nor do we have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes, we use it to analyze other quantities, such as the hiring cost in

procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an expected running time. The expectation is taken over the distribution of the possible inputs. Thus we are, in effect, averaging the running time over all possible inputs.

We must be very careful in deciding on the distribution of inputs. For some problems, it is reasonable to assume something about the set of all possible inputs, and we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis.

For the hiring problem, we can assume that the applicants come in a random order. What does that mean for this problem? We assume that we can compare any two candidates and decide which one is better qualified; that is, there is a total order on the candidates. (See Appendix B for the definition of a total order.) We can therefore rank each candidate with a unique number from 1 through n, using $rank(i)$ to denote the rank of applicant i, and adopt the convention that a higher rank corresponds to a better qualified applicant. The ordered list $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ is a permutation of the list $\langle 1, 2, \ldots, n\rangle$. Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the $n!$ permutations of the numbers 1 through n. Alternatively, we say that the ranks form a ***uniform random permutation***; that is, each of the possible $n!$ permutations appears with equal probability.

Section 5.2 contains a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distribution on the inputs. In many cases, we know very little about the input distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally. Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random.

In the hiring problem, it may seem as if the candidates are being presented to us in a random order, but we have no way of knowing whether or not they really are. Thus, in order to develop a randomized algorithm for the hiring problem, we must have greater control over the order in which we interview the candidates. We will, therefore, change the model slightly. We will say that the employment agency has n candidates, and they send us a list of the candidates in advance. On each day, we choose, randomly, which candidate to interview. Although we know nothing about the candidates (besides their names), we have made a significant change.

Instead of relying on a guess that the candidates will come to us in a random order, we have instead gained control of the process and enforced a random order.

More generally, we call an algorithm ***randomized*** if its behavior is determined not only by its input but also by values produced by a ***random-number generator***. We shall assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(0, 1)$ produces 0 with probability $1/2$, and it produces 1 with probability $1/2$. A call to RANDOM$(3, 7)$ returns either 3, 4, 5, 6 or 7, each with probability $1/5$. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a $(b - a + 1)$-sided die to obtain its output. (In practice, most programming environments offer a ***pseudorandom-number generator***: a deterministic algorithm returning numbers that "look" statistically random.)a

Exercises

5.1-1

Show that the assumption that we are always able to determine which candidate is best in line 4 of procedure HIRE-ASSISTANT implies that we know a total order on the ranks of the candidates.

5.1-2 ⋆

Describe an implementation of the procedure RANDOM(a, b) that only makes calls to RANDOM$(0, 1)$. What is the expected running time of your procedure, as a function of a and b?

5.1-3 ⋆

Suppose that you want to output 0 with probability $1/2$ and 1 with probability $1/2$. At your disposal is a procedure BIASED-RANDOM, that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$, but you do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability $1/2$ and 1 with probability $1/2$. What is the expected running time of your algorithm as a function of p?

5.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we will use indicator random variables. Indicator random variables provide a convenient method

for converting between probabilities and expectations. Suppose we are given a sample space S and an event A. Then the ***indicator random variable*** $\mathrm{I}\{A\}$ associated with event A is defined as

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases} \tag{5.1}$$

As a simple example, let us determine the expected number of heads that we obtain when flipping a fair coin. Our sample space is $S = \{H, T\}$, and we define a random variable Y which takes on the values H and T, each with probability $1/2$. We can then define an indicator random variable X_H, associated with the coin coming up heads, which we can express as the event $Y = H$. This variable counts the number of heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$X_H = \mathrm{I}\{Y = H\} = \begin{cases} 1 & \text{if } Y = H \text{ ,} \\ 0 & \text{if } Y = T \text{ .} \end{cases}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{Y = H\}] \\ &= 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \text{ .} \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is $1/2$. As the following lemma shows, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Lemma 5.1
Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} \text{ ,} \end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

Although indicator random variables may seem cumbersome for an application such as counting the expected number of heads on a flip of a single coin, they are

useful for analyzing situations in which we perform repeated random trials. For example, indicator random variables give us a simple way to arrive at the result of equation (C.36). In this equation, we compute the number of heads in n coin flips by considering separately the probability of obtaining 0 heads, 1 heads, 2 heads, etc. However, the simpler method proposed in equation (C.37) actually implicitly uses indicator random variables. Making this argument more explicit, we can let X_i be the indicator random variable associated with the event in which the ith flip comes up heads. Letting Y_i be the random variable denoting the outcome of the ith flip, we have that $X_i = \mathrm{I}\{Y_i = H\}$. Let X be the random variable denoting the total number of heads in the n coin flips, so that

$$X = \sum_{i=1}^{n} X_i \ .$$

We wish to compute the expected number of heads, so we take the expectation of both sides of the above equation to obtain

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

The left side of the above equation is the expectation of the sum of n random variables. By Lemma 5.1, we can easily compute the expectation of each of the random variables. By equation (C.20)—linearity of expectation—it is easy to compute the expectation of the sum: it equals the sum of the expectations of the n random variables. Linearity of expectation makes the use of indicator random variables a powerful analytical technique; it applies even when there is dependence among the random variables. We now can easily compute the expected number of heads:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \ .
\end{aligned}$$

Thus, compared to the method used in equation (C.36), indicator random variables greatly simplify the calculation. We shall use indicator random variables throughout this book.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of times that we hire a new office assistant. In order to use a probabilistic analysis, we assume that the candidates arrive in a random order, as discussed in the previous section. (We shall see in Section 5.3 how to remove this assumption.) Let X be the random variable whose value equals the number of times we hire a new office assistant. We could then apply the definition of expected value from equation (C.19) to obtain

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

but this calculation would be cumbersome. We shall instead use indicator random variables to greatly simplify the calculation.

To use indicator random variables, instead of computing $\mathrm{E}[X]$ by defining one variable associated with the number of times we hire a new office assistant, we define n variables related to whether or not each particular candidate is hired. In particular, we let X_i be the indicator random variable associated with the event in which the ith candidate is hired. Thus,

$$X_i = \mathrm{I}\{\text{candidate } i \text{ is hired}\} = \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases} \tag{5.2}$$

and

$$X = X_1 + X_2 + \cdots + X_n . \tag{5.3}$$

By Lemma 5.1, we have that

$$\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\} ,$$

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT are executed.

Candidate i is hired, in line 5, exactly when candidate i is better than each of candidates 1 through $i - 1$. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order. Any one of these first i candidates is equally likely to be the best-qualified so far. Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i - 1$ and thus a probability of $1/i$ of being hired. By Lemma 5.1, we conclude that

$$\mathrm{E}[X_i] = 1/i . \tag{5.4}$$

Now we can compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} \qquad (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by linearity of expectation)}\\
&= \sum_{i=1}^{n} 1/i && \text{(by equation (5.4))}\\
&= \ln n + O(1) && \text{(by equation (A.7))} \ . \qquad (5.6)
\end{aligned}$$

Even though we interview n people, we only actually hire approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

Lemma 5.2
Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has a total hiring cost of $O(c_h \ln n)$.

Proof The bound follows immediately from our definition of the hiring cost and equation (5.6). ■

The expected interview cost is a significant improvement over the worst-case hiring cost of $O(nc_h)$.

Exercises

5.2-1
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you will hire exactly one time? What is the probability that you will hire exactly n times?

5.2-2
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you will hire exactly twice?

5.2-3
Use indicator random variables to compute the expected value of the sum of n dice.

5.2-4
Use indicator random variables to solve the following problem, which is known as the ***hat-check problem***. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers that get back their own hat?

5.2-5

Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

5.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm. Many times, we do not have such knowledge and no average-case analysis is possible. As mentioned in Section 5.1, we may be able to use a randomized algorithm.

For a problem such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis will guide the development of a randomized algorithm. Instead of assuming a distribution of inputs, we impose a distribution. In particular, before running the algorithm, we randomly permute the candidates in order to enforce the property that every permutation is equally likely. This modification does not change our expectation of hiring a new office assistant roughly $\ln n$ times. It means, however, that for *any* input we expect this to be the case, rather than for inputs drawn from a particular distribution.

We now explore the distinction between probabilistic analysis and randomized algorithms further. In Section 5.2, we claimed that, assuming that the candidates are presented in a random order, the expected number of times we hire a new office assistant is about $\ln n$. Note that the algorithm here is deterministic; for any particular input, the number of times a new office assistant is hired will always be the same. Furthermore, the number of times we hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates. Since this number depends only on the ranks of the candidates, we can represent a particular input by listing, in order, the ranks of the candidates, i.e., $\langle rank(1), rank(2), \ldots, rank(n) \rangle$. Given the rank list $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$, a new office assistant will always be hired 10 times, since each successive candidate is better than the previous one, and lines 5–6 will be executed in each iteration of the algorithm. Given the list of ranks $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$, a new office assistant will be hired only once, in the first iteration. Given a list of ranks $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$, a new office assistant will be hired three times, upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost of our algorithm is dependent on how many times we hire a new office assistant, we

see that there are expensive inputs, such as A_1, inexpensive inputs, such as A_2, and moderately expensive inputs, such as A_3.

Consider, on the other hand, the randomized algorithm that first permutes the candidates and then determines the best candidate. In this case, the randomization is in the algorithm, not in the input distribution. Given a particular input, say A_3 above, we cannot say how many times the maximum will be updated, because this quantity differs with each run of the algorithm. The first time we run the algorithm on A_3, it may produce the permutation A_1 and perform 10 updates, while the second time we run the algorithm, we may produce the permutation A_2 and perform only one update. The third time we run it, we may perform some other number of updates. Each time we run the algorithm, the execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, *no particular input elicits its worst-case behavior*. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.

For the hiring problem, the only change needed in the code is to randomly permute the array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  randomly permute the list of candidates
2  best ← 0        ▷ candidate 0 is a least-qualified dummy candidate
3  for i ← 1 to n
4      do interview candidate i
5         if candidate i is better than candidate best
6           then best ← i
7                hire candidate i
```

With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

Lemma 5.3

The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof After permuting the input array, we have achieved a situation identical to that of the probabilistic analysis of HIRE-ASSISTANT. ■

The comparison between Lemmas 5.2 and 5.3 captures the difference between probabilistic analysis and randomized algorithms. In Lemma 5.2, we make an

assumption about the input. In Lemma 5.3, we make no such assumption, although randomizing the input takes some additional time. In the remainder of this section, we discuss some issues involved in randomly permuting inputs.

Randomly permuting arrays

Many randomized algorithms randomize the input by permuting the given input array. (There are other ways to use randomization.) Here, we shall discuss two methods for doing so. We assume that we are given an array A which, without loss of generality, contains the elements 1 through n. Our goal is to produce a random permutation of the array.

One common method is to assign each element $A[i]$ of the array a random priority $P[i]$, and then sort the elements of A according to these priorities. For example if our initial array is $A = \langle 1, 2, 3, 4 \rangle$ and we choose random priorities $P = \langle 36, 3, 97, 19 \rangle$, we would produce an array $B = \langle 2, 4, 1, 3 \rangle$, since the second priority is the smallest, followed by the fourth, then the first, and finally the third. We call this procedure PERMUTE-BY-SORTING:

```
PERMUTE-BY-SORTING(A)
1  n ← length[A]
2  for i ← 1 to n
3        do P[i] = RANDOM(1, n^3)
4  sort A, using P as sort keys
5  return A
```

Line 3 chooses a random number between 1 and n^3. We use a range of 1 to n^3 to make it likely that all the priorities in P are unique. (Exercise 5.3-5 asks you to prove that the probability that all entries are unique is at least $1 - 1/n$, and Exercise 5.3-6 asks how to implement the algorithm even if two or more priorities are identical.) Let us assume that all the priorities are unique.

The time-consuming step in this procedure is the sorting in line 4. As we shall see in Chapter 8, if we use a comparison sort, sorting takes $\Omega(n \lg n)$ time. We can achieve this lower bound, since we have seen that merge sort takes $\Theta(n \lg n)$ time. (We shall see other comparison sorts that take $\Theta(n \lg n)$ time in Part II.) After sorting, if $P[i]$ is the jth smallest priority, then $A[i]$ will be in position j of the output. In this manner we obtain a permutation. It remains to prove that the procedure produces a ***uniform random permutation***, that is, that every permutation of the numbers 1 through n is equally likely to be produced.

Lemma 5.4
Procedure PERMUTE-BY-SORTING produces a uniform random permutation of the input, assuming that all priorities are distinct.

Proof We start by considering the particular permutation in which each element $A[i]$ receives the ith smallest priority. We shall show that this permutation occurs with probability exactly $1/n!$. For $i = 1, 2, \ldots, n$, let X_i be the event that element $A[i]$ receives the ith smallest priority. Then we wish to compute the probability that for all i, event X_i occurs, which is

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} \ .$$

Using Exercise C.2-6, this probability is equal to

$$\begin{aligned}&\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\} \\ &\quad \cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\} \ .\end{aligned}$$

We have that $\Pr\{X_1\} = 1/n$ because it is the probability that one priority chosen randomly out of a set of n is the smallest. Next, we observe that $\Pr\{X_2 \mid X_1\} = 1/(n-1)$ because given that element $A[1]$ has the smallest priority, each of the remaining $n-1$ elements has an equal chance of having the second smallest priority. In general, for $i = 2, 3, \ldots, n$, we have that $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$, since, given that elements $A[1]$ through $A[i-1]$ have the $i-1$ smallest priorities (in order), each of the remaining $n-(i-1)$ elements has an equal chance of having the ith smallest priority. Thus, we have

$$\begin{aligned}\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!} \ ,\end{aligned}$$

and we have shown that the probability of obtaining the identity permutation is $1/n!$.

We can extend this proof to work for any permutation of priorities. Consider any fixed permutation $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ of the set $\{1, 2, \ldots, n\}$. Let us denote by r_i the rank of the priority assigned to element $A[i]$, where the element with the jth smallest priority has rank j. If we define X_i as the event in which element $A[i]$ receives the $\sigma(i)$th smallest priority, or $r_i = \sigma(i)$, the same proof still applies. Therefore, if we calculate the probability of obtaining any particular permutation, the calculation is identical to the one above, so that the probability of obtaining this permutation is also $1/n!$. ∎

One might think that to prove that a permutation is a uniform random permutation it suffices to show that, for each element $A[i]$, the probability that it winds up in position j is $1/n$. Exercise 5.3-4 shows that this weaker condition is, in fact, insufficient.

A better method for generating a random permutation is to permute the given array in place. The procedure RANDOMIZE-IN-PLACE does so in $O(n)$ time.

In iteration i, the element $A[i]$ is chosen randomly from among elements $A[i]$ through $A[n]$. Subsequent to iteration i, $A[i]$ is never altered.

```
RANDOMIZE-IN-PLACE(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(i, n)]
```

We will use a loop invariant to show that procedure RANDOMIZE-IN-PLACE produces a uniform random permutation. Given a set of n elements, a k-permutation is a sequence containing k of the n elements. (See Appendix C.) There are $n!/(n-k)!$ such possible k-permutations.

Lemma 5.5
Procedure RANDOMIZE-IN-PLACE computes a uniform random permutation.

Proof We use the following loop invariant:

> Just prior to the ith iteration of the **for** loop of lines 2–3, for each possible $(i-1)$-permutation, the subarray $A[1..i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Consider the situation just before the first loop iteration, so that $i = 1$. The loop invariant says that for each possible 0-permutation, the subarray $A[1..0]$ contains this 0-permutation with probability $(n-i+1)!/n! = n!/n! = 1$. The subarray $A[1..0]$ is an empty subarray, and a 0-permutation has no elements. Thus, $A[1..0]$ contains any 0-permutation with probability 1, and the loop invariant holds prior to the first iteration.

Maintenance: We assume that just before the $(i-1)$st iteration, each possible $(i-1)$-permutation appears in the subarray $A[1..i-1]$ with probability $(n-i+1)!/n!$, and we will show that after the ith iteration, each possible i-permutation appears in the subarray $A[1..i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration will then maintain the loop invariant.

Let us examine the ith iteration. Consider a particular i-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i \rangle$. This permutation consists of an $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ followed by the value x_i that the algorithm places in $A[i]$. Let E_1 denote the event in which the first $i-1$ iterations have created the particular $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ in $A[1..i-1]$. By the

loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$. Let E_2 be the event that ith iteration puts x_i in position $A[i]$. The i-permutation $\langle x_1, \ldots, x_i \rangle$ is formed in $A[1\,..\,i]$ precisely when both E_1 and E_2 occur, and so we wish to compute $\Pr\{E_2 \cap E_1\}$. Using equation (C.14), we have

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \ .$$

The probability $\Pr\{E_2 \mid E_1\}$ equals $1/(n-i+1)$ because in line 3 the algorithm chooses x_i randomly from the $n-i+1$ values in positions $A[i\,..\,n]$. Thus, we have

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} \ . \end{aligned}$$

Termination: At termination, $i = n+1$, and we have that the subarray $A[1\,..\,n]$ is a given n-permutation with probability $(n-n)!/n! = 1/n!$.

Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation. ■

A randomized algorithm is often the simplest and most efficient way to solve a problem. We shall use randomized algorithms occasionally throughout this book.

Exercises

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He questions whether it is true prior to the first iteration. His reasoning is that one could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.5 for your procedure.

5.3-2

Professor Kelp decides to write a procedure that will produce at random any permutation besides the identity permutation. He proposes the following procedure:

```
PERMUTE-WITHOUT-IDENTITY(A)
1  n ← length[A]
2  for i ← 1 to n − 1
3       do swap A[i] ↔ A[RANDOM(i + 1, n)]
```

Does this code do what Professor Kelp intends?

5.3-3
Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i \,..\, n]$, we swapped it with a random element from anywhere in the array:

```
PERMUTE-WITH-ALL(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(1, n)]
```

Does this code produce a uniform random permutation? Why or why not?

5.3-4
Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```
PERMUTE-BY-CYCLIC(A)
1  n ← length[A]
2  offset ← RANDOM(1, n)
3  for i ← 1 to n
4       do dest ← i + offset
5          if dest > n
6            then dest ← dest − n
7          B[dest] ← A[i]
8  return B
```

Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.

5.3-5 ⋆
Prove that in the array P in procedure PERMUTE-BY-SORTING, the probability that all elements are unique is at least $1 - 1/n$.

5.3-6
Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the case in which two or more priorities are identical. That is, your algorithm should produce a uniform random permutation, even if two or more priorities are identical.