9

Probabilistic algorithms

It is sometimes useful to endow our algorithms with the ability to generate random numbers. In fact, we have already seen two examples of how such probabilistic algorithms may be useful:

- at the end of §3.4, we saw how a probabilistic algorithm might be used to build a simple and efficient primality test; however, this test might incorrectly assert that a composite number is prime; in the next chapter, we will see how a small modification to this algorithm will ensure that the probability of making such a mistake is extremely small;
- in §4.5, we saw how a probabilistic algorithm could be used to make Fermat's two squares theorem constructive; in this case, the use of randomization never leads to incorrect results, but the running time of the algorithm was only bounded "in expectation."

We will see a number of other probabilistic algorithms in this text, and it is high time that we place them on a firm theoretical foundation. To simplify matters, we only consider algorithms that generate random bits. Where such random bits actually come from will not be of great concern to us here. In a practical implementation, one would use a pseudo-random bit generator, which should produce bits that "for all practical purposes" are "as good as random." While there is a well-developed theory of pseudo-random bit generation (some of which builds on the ideas in §8.9), we will not delve into this here. Moreover, the pseudo-random bit generators used in practice are not based on this general theory, and are much more ad hoc in design. So, although we will present a rigorous formal theory of probabilistic algorithms, the application of this theory to practice is ultimately a bit heuristic; nevertheless, experience with these algorithms has shown that the theory is a very good predictor of the real-world behavior of these algorithms.

9.1 Basic definitions

Formally speaking, we will add a new type of instruction to our random access machine (described in §3.2):

random bit This type of instruction is of the form $\gamma \leftarrow \text{RAND}$, where γ takes the same form as in arithmetic instructions. Execution of this type of instruction assigns to γ a value sampled from the uniform distribution on $\{0, 1\}$, independently from the execution of all other random-bit instructions.

Algorithms that use random-bit instructions are called **probabilistic** (or **randomized**), while those that do not are called **deterministic**.

In describing probabilistic algorithms at a high level, we shall write "$y \xleftarrow{\$} \{0, 1\}$" to denote the assignment of a random bit to the variable y, and "$y \xleftarrow{\$} \{0, 1\}^{\times \ell}$" to denote the assignment of a random bit string of length ℓ to the variable y.

To analyze the behavior of a probabilistic algorithm, we first need a probability distribution that appropriately models its execution. Once we have done this, we shall define the running time and output to be *random variables* associated with this distribution.

9.1.1 Defining the distribution

It would be desirable to define a probability distribution that could be used for all algorithms and all inputs. While this can be done in principle, it would require notions from the theory of probability more advanced than those we developed in the previous chapter. Instead, for a given probabilistic algorithm A and input x, we shall define a discrete probability distribution that models A's execution on input x. Thus, every algorithm/input pair yields a different distribution.

To motivate our definition, consider Example 8.43. We could view the sample space in that example to be the set of all bit strings consisting of zero or more 0 bits, followed by a single 1 bit, and to each such bit string ω of this special form, we assign the probability $2^{-|\omega|}$, where $|\omega|$ denotes the length of ω. The "random experiment" we have in mind is to generate random bits one at a time until one of these special "halting" strings is generated. In developing the definition of the probability distribution for a probabilistic algorithm, we simply consider more general sets of "halting" strings, as determined by the algorithm and its input.

So consider a fixed algorithm A and input x. Let λ be a finite bit string of length, say, ℓ. We can use λ to "drive" the execution of A on input x for up to ℓ execution steps, as follows: for each step $i = 1, \ldots, \ell$, if the ith instruction executed by A is $\gamma \leftarrow \text{RAND}$, the ith bit of λ is assigned to γ. In this context, we shall refer to λ as an **execution path**. The reader may wish to visualize λ as a finite path in an

infinite binary tree, where we start at the root, branching to the left if the next bit in λ is a 0 bit, and branching to the right if the next bit in λ is a 1 bit.

After using λ to drive A on input x for up to ℓ steps, we might find that the algorithm executed a *halt* instruction at some point during the execution, in which case we call λ a **complete** execution path; moreover, if this *halt* instruction was the ℓth instruction executed by A, then we call λ an **exact** execution path.

Our intent is to define the probability distribution associated with A on input x to be $\mathsf{P} : \Omega \to [0, 1]$, where the sample space Ω is the set of all *exact* execution paths, and $\mathsf{P}(\omega) := 2^{-|\omega|}$ for each $\omega \in \Omega$. However, for this to work, all the probabilities must sum to 1. The next theorem at least guarantees that these probabilities sum to at most 1. The only property of Ω that really matters in the proof of this theorem is that it is **prefix free**, which means that no exact execution path is a proper prefix of any other.

Theorem 9.1. *Let Ω be the set of all exact execution paths for A on input x. Then $\sum_{\omega\in\Omega} 2^{-|\omega|} \leq 1$.*

Proof. Let k be a non-negative integer. Let $\Omega_k \subseteq \Omega$ be the set of all exact execution paths of length at most k, and let $\alpha_k := \sum_{\omega\in\Omega_k} 2^{-|\omega|}$. We shall show below that

$$\alpha_k \leq 1. \tag{9.1}$$

From this, it will follow that

$$\sum_{\omega\in\Omega} 2^{-|\omega|} = \lim_{k\to\infty} \alpha_k \leq 1.$$

To prove the inequality (9.1), consider the set C_k of all *complete* execution paths of length *equal* to k. We claim that

$$\alpha_k = 2^{-k}|C_k|, \tag{9.2}$$

from which (9.1) follows, since clearly, $|C_k| \leq 2^k$. So now we are left to prove (9.2). Observe that by definition, each $\lambda \in C_k$ extends some $\omega \in \Omega_k$; that is, ω is a prefix of λ; moreover, ω is uniquely determined by λ, since no exact execution path is a proper prefix of any other exact execution path. Also observe that for each $\omega \in \Omega_k$, if $C_k(\omega)$ is the set of execution paths $\lambda \in C_k$ that extend ω, then $|C_k(\omega)| = 2^{k-|\omega|}$, and by the previous observation, $\{C_k(\omega)\}_{\omega\in\Omega_k}$ is a partition of C_k. Thus, we have

$$\alpha_k = \sum_{\omega\in\Omega_k} 2^{-|\omega|} = \sum_{\omega\in\Omega_k} 2^{-|\omega|} \sum_{\lambda\in C_k(\omega)} 2^{-k+|\omega|} = 2^{-k} \sum_{\omega\in\Omega_k} \sum_{\lambda\in C_k(\omega)} 1 = 2^{-k}|C_k|,$$

which proves (9.2). □

From the above theorem, if Ω is the set of all exact execution paths for A on input x, then

$$\alpha := \sum_{\omega \in \Omega} 2^{-|\omega|} \leq 1,$$

and we say that A **halts with probability** α **on input** x. If $\alpha = 1$, we define the distribution $\mathsf{P} : \Omega \to [0, 1]$ associated with A on input x, where $\mathsf{P}(\omega) := 2^{-|\omega|}$ for each $\omega \in \Omega$.

We shall mainly be interested in algorithms that halt with probability 1 on all inputs. The following four examples provide some simple criteria that guarantee this.

Example 9.1. Suppose that on input x, A always halts within a finite number of steps, regardless of its random choices. More precisely, this means that there is a bound ℓ (depending on A and x), such that all execution paths of length ℓ are complete. In this case, we say that A's running time on input x is **strictly bounded by** ℓ, and it is clear that A halts with probability 1 on input x. Moreover, one can much more simply model A's computation on input x by working with the uniform distribution on execution paths of length ℓ. □

Example 9.2. Suppose A and B are probabilistic algorithms that both halt with probability 1 on all inputs. Using A and B as subroutines, we can form their **serial composition**; that is, we can construct the algorithm

$$C(x): \qquad \text{output } B(A(x)),$$

which on input x, first runs A on input x, obtaining a value y, then runs B on input y, obtaining a value z, and finally, outputs z. We claim that C halts with probability 1 on all inputs.

For simplicity, we may assume that A places its output y in a location in memory where B expects to find its input, and that B places its output in a location in memory where C's output should go. With these assumptions, the program for C is obtained by simply concatenating the programs for A and B, making the following adjustments: every *halt* instruction in A's program is translated into an instruction that branches to the first instruction of B's program, and every target in a branch instruction in B's program is increased by the length of A's program.

Let Ω be the sample space representing A's execution on an input x. Each $\omega \in \Omega$ determines an output y, and a corresponding sample space Ω'_ω representing B's execution on input y. The sample space representing C's execution on input x is

$$\Omega'' = \{\omega\omega' : \omega \in \Omega, \omega' \in \Omega'_\omega\},$$

where $\omega\omega'$ is the concatenation of ω and ω'. We have

$$\sum_{\omega\omega' \in \Omega''} 2^{-|\omega\omega'|} = \sum_{\omega \in \Omega} 2^{-|\omega|} \sum_{\omega' \in \Omega'_\omega} 2^{-|\omega'|} = \sum_{\omega \in \Omega} 2^{-|\omega|} \cdot 1 = 1,$$

which shows that C halts with probability 1 on input x. □

Example 9.3. Suppose A, B, and C are probabilistic algorithms that halt with probability 1 on all inputs, and that A always outputs either *true* or *false*. Then we can form the **conditional construct**

$D(x)$: if $A(x)$ then output $B(x)$ else output $C(x)$.

By a calculation similar to that in the previous example, it is easy to see that D halts with probability 1 on all inputs. □

Example 9.4. Suppose A and B are probabilistic algorithms that halt with probability 1 on all inputs, and that A always outputs either *true* or *false*. We can form the **iterative construct**

$C(x)$: while $A(x)$ do $x \leftarrow B(x)$
output x.

Algorithm C may or may not halt with probability 1. To analyze C, we define an infinite sequence of algorithms $\{C_n\}_{n=0}^{\infty}$; namely, we define C_0 as

$C_0(x)$: halt,

and for $n > 0$, we define C_n as

$C_n(x)$: if $A(x)$ then $C_{n-1}(B(x))$.

Essentially, C_n drives C for up to n loop iterations before halting, if necessary, in C_0. By the previous three examples, it follows by induction on n that each C_n halts with probability 1 on all inputs. Therefore, we have a well-defined probability distribution for each C_n and each input x.

Consider a fixed input x. For each $n \geq 0$, let β_n be the probability that on input x, C_n terminates by executing algorithm C_0. Intuitively, β_n is the probability that C executes at least n loop iterations; however, this probability is defined with respect to the probability distribution associated with algorithm C_n on input x. It is not hard to see that the sequence $\{\beta_n\}_{n=0}^{\infty}$ is non-increasing, and so the limit $\beta := \lim_{n\to\infty} \beta_n$ exists; moreover, C halts with probability $1 - \beta$ on input x.

On the one hand, if the loop in algorithm C is guaranteed to terminate after a finite number of iterations (as in a "for loop"), then C certainly halts with probability 1. Indeed, if on input x, there is a bound ℓ (depending on x) such that the number of loop iterations is always at most ℓ, then $\beta_{\ell+1} = \beta_{\ell+2} = \cdots = 0$. On the other hand, if on input x, C enters into a good, old-fashioned infinite loop, then C

certainly does not halt with probability 1, as $\beta_0 = \beta_1 = \cdots = 1$. Of course, there may be in-between cases, which require further analysis. □

We now illustrate the above criteria with a couple of some simple, concrete examples.

Example 9.5. Consider the following algorithm, which models an experiment in which we toss a fair coin repeatedly until it comes up *heads*:

repeat
    $b \xleftarrow{\$} \{0,1\}$
until $b = 1$

For each positive integer n, let β_n be the probability that the algorithm executes at least n loop iterations, in the sense of Example 9.4. It is not hard to see that $\beta_n = 2^{-n+1}$, and since $\beta_n \to 0$ as $n \to \infty$, the algorithm halts with probability 1, even though the loop is not guaranteed to terminate after any particular, finite number of steps. □

Example 9.6. Consider the following algorithm:

$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    $\sigma \xleftarrow{\$} \{0,1\}^{\times i}$
until $\sigma = 0^{\times i}$

For each positive integer n, let β_n be the probability that the algorithm executes at least n loop iterations, in the sense of Example 9.4. It is not hard to see that

$$\beta_n = \prod_{i=1}^{n-1}(1 - 2^{-i}) \geq \prod_{i=1}^{n-1} e^{-2^{-i+1}} = e^{-\sum_{i=0}^{n-2} 2^{-i}} \geq e^{-2},$$

where we have made use of the estimate (iii) in §A1. Therefore,

$$\lim_{n\to\infty} \beta_n \geq e^{-2} > 0,$$

and so the algorithm does not halt with probability 1, even though it never falls into an infinite loop. □

9.1.2 Defining the running time and output

Let A be a probabilistic algorithm that halts with probability 1 on a fixed input x. We may define the random variable Z that represents A's running time on input x, and the random variable Y that represents A's output on input x.

Formally, Z and Y are defined using the probability distribution on the sample space Ω, defined in §9.1.2. The sample space Ω consists of all exact execution paths for A on input x. For each $\omega \in \Omega$, $Z(\omega) := |\omega|$, and $Y(\omega)$ is the output produced by A on input x, using ω to drive its execution.

The **expected running time of A on input x** is defined to be $\mathsf{E}[Z]$. Note that in defining the expected running time, we view the input as fixed, rather than drawn from some probability distribution. Also note that the expected running time may be infinite.

We say that A runs in **expected polynomial time** if there exist constants a, b, and c, such that for all n, and for all inputs x of size n, the expected running time of A on input x is at most $an^b + c$. We say that A runs in **strict polynomial time** if there exist constants a, b, and c, such that for all n, and for all inputs x of size n, A's running time on input x is strictly bounded by $an^b + c$ (as in Example 9.1).

Example 9.7. Consider again the algorithm in Example 9.5. Let L be the random variable that represents the number of loop iterations executed by the algorithm. The distribution of L is a geometric distribution, with associated success probability $1/2$ (see Example 8.44). Therefore, $\mathsf{E}[L] = 2$ (see Example 8.46). Let Z be the random variable that represents the running time of the algorithm. We have $Z \leq cL$, for some implementation-dependent constant c. Therefore, $\mathsf{E}[Z] \leq c\,\mathsf{E}[L] = 2c$. □

Example 9.8. Consider the following probabilistic algorithm that takes as input a positive integer m. It models an experiment in which we toss a fair coin repeatedly until it comes up *heads* m times.

$k \leftarrow 0$
repeat
    $b \xleftarrow{\varphi} \{0, 1\}$
    if $b = 1$ then $k \leftarrow k + 1$
until $k = m$

Let L be the random variable that represents the number of loop iterations executed the algorithm on a fixed input m. We claim that $\mathsf{E}[L] = 2m$. To see this, define random variables $L_1, \ldots, L_m$, where L_1 is the number of loop iterations needed to get $b = 1$ for the first time, L_2 is the number of additional loop iterations needed to get $b = 1$ for the second time, and so on. Clearly, we have $L = L_1 + \cdots + L_m$, and moreover, $\mathsf{E}[L_i] = 2$ for $i = 1, \ldots, m$; therefore, by linearity of expectation, we have $\mathsf{E}[L] = \mathsf{E}[L_1] + \cdots + \mathsf{E}[L_m] = 2m$. It follows that the expected running time of this algorithm on input m is $O(m)$. □

Example 9.9. Consider the following algorithm:

$$
\begin{array}{l}
n \leftarrow 0 \\
\text{repeat} \quad n \leftarrow n+1, \quad b \xleftarrow{\$} \{0,1\} \quad \text{until } b = 1 \\
\text{repeat} \quad \sigma \xleftarrow{\$} \{0,1\}^{\times n} \quad \text{until } \sigma = 0^{\times n}
\end{array}
$$

The expected running time is infinite (even though it does halt with probability 1). To see this, define random variables L_1 and L_2, where L_1 is the number of iterations of the first loop, and L_2 is the number of iterations of the second. As in Example 9.7, the distribution of L_1 is a geometric distribution with associated success probability $1/2$, and $\mathsf{E}[L_1] = 2$. For each $k \geq 1$, the conditional distribution of L_2 given $L_1 = k$ is a geometric distribution with associated success probability $1/2^k$, and so $\mathsf{E}[L_2 \mid L_1 = k] = 2^k$. Therefore,

$$\mathsf{E}[L_2] = \sum_{k \geq 1} \mathsf{E}[L_2 \mid L_1 = k]\,\mathsf{P}[L_1 = k] = \sum_{k \geq 1} 2^k \cdot 2^{-k} = \sum_{k \geq 1} 1 = \infty. \quad \square$$

We have presented a fairly rigorous definitional framework for probabilistic algorithms, but from now on, we shall generally reason about such algorithms at a higher, and more intuitive, level. Nevertheless, all of our arguments can be translated into this rigorous framework, the details of which we leave to the interested reader. Moreover, all of the algorithms we shall present halt with probability 1 on all inputs, but we shall not go into the details of proving this (but the criteria in Examples 9.1–9.4 can be used to easily verify this).

EXERCISE 9.1. Suppose A is a probabilistic algorithm that halts with probability 1 on input x, and let $\mathsf{P} : \Omega \to [0,1]$ be the corresponding probability distribution. Let λ be an execution path of length ℓ, and assume that no proper prefix of λ is exact. Let $\mathcal{E}_\lambda := \{\omega \in \Omega : \omega \text{ extends } \lambda\}$. Show that $\mathsf{P}[\mathcal{E}_\lambda] = 2^{-\ell}$.

EXERCISE 9.2. Let A be a probabilistic algorithm that on a given input x, halts with probability 1, and produces an output in the set T. Let P be the corresponding probability distribution, and let Y and Z be random variables representing the output and running time, respectively. For each $k \geq 0$, let P_k be the uniform distribution on all execution paths λ of length k. We define random variables Y_k and Z_k, associated with P_k, as follows: if λ is complete, we define $Y_k(\lambda)$ to be the output produced by A, and $Z_k(\lambda)$ to be the actual number of steps executed by A; otherwise, we define $Y_k(\lambda)$ to be the special value "$\perp$" and $Z_k(\lambda)$ to be k. For each $t \in T$, let p_{tk} be the probability (relative to P_k) that $Y_k = t$, and let μ_k be the expected value (relative to P_k) of Z_k. Show that:

(a) for each $t \in T$, $\mathsf{P}[Y = t] = \lim_{k \to \infty} p_{tk}$;

(b) $\mathsf{E}[Z] = \lim_{k\to\infty} \mu_k$.

EXERCISE 9.3. Let A_1 and A_2 be probabilistic algorithms. Let B be any probabilistic algorithm that always outputs 0 or 1. For $i = 1, 2$, let A'_i be the algorithm that on input x computes and outputs $B(A_i(x))$. Fix an input x, and let Y_1 and Y_2 be random variables representing the outputs of A_1 and A_2, respectively, on input x, and let Y'_1 and Y'_2 be random variables representing the outputs of A'_1 and A'_2, respectively, on input x. Assume that the images of Y_1 and Y_2 are finite, and let $\delta := \Delta[Y_1; Y_2]$ be their statistical distance. Show that $|\mathsf{P}[Y'_1 = 1] - \mathsf{P}[Y'_2 = 1]| \leq \delta$.

9.2 Generating a random number from a given interval

Suppose we want to generate a number, uniformly at random from the interval $\{0, \ldots, m-1\}$, for a given positive integer m.

If m is a power of 2, say $m = 2^\ell$, then we can do this directly as follows: generate a random ℓ-bit string σ, and convert σ to the integer $I(\sigma)$ whose base-2 representation is σ; that is, if $\sigma = b_{\ell-1} b_{\ell-2} \cdots b_0$, where the b_i's are bits, then

$$I(\sigma) := \sum_{i=0}^{\ell-1} b_i 2^i.$$

In the general case, we do not have a direct way to do this, since we can only directly generate random bits. But the following algorithm does the job:

Algorithm RN**.** On input m, where m is a positive integer, do the following, where $\ell := \lceil \log_2 m \rceil$:

```
repeat
    σ ←$ {0,1}^{×ℓ}
    y ← I(σ)
until y < m
output y
```

Theorem 9.2. *The expected running time of Algorithm* RN *is* $O(\mathrm{len}(m))$*, and its output is uniformly distributed over* $\{0, \ldots, m-1\}$.

Proof. Note that $m \leq 2^\ell < 2m$. Let L denote the number of loop iterations of this algorithm, and Z its running time. With every loop iteration, the algorithm halts with probability $m/2^\ell$, and so the distribution of L is a geometric distribution with associated success probability $m/2^\ell > 1/2$. Therefore, $\mathsf{E}[L] = 2^\ell/m < 2$. Since $Z \leq c\,\mathrm{len}(m) \cdot L$ for some constant c, it follows that $\mathsf{E}[Z] = O(\mathrm{len}(m))$.

Next, we analyze the output distribution. Let Y denote the output of the algorithm. We want to show that Y is uniformly distributed over $\{0, \ldots, m-1\}$. This

is perhaps intuitively obvious, but let us give a rigorous justification of this claim. To do this, for $i = 1, 2, \ldots$, let Y_i denote the value of y in the ith loop iteration; for completeness, if the ith loop iteration is not executed, then we define $Y_i := \perp$. Also, for $i = 1, 2 \ldots$, let $\mathcal{H}_i$ be the event that the algorithm halts in the ith loop iteration (i.e., $\mathcal{H}_i$ is the event that $L = i$). Let $t \in \{0, \ldots, m-1\}$ be fixed.

First, by total probability (specifically, the infinite version of (8.9), discussed in §8.10.2), we have

$$\mathsf{P}[Y = t] = \sum_{i \geq 1} \mathsf{P}[(Y = t) \cap \mathcal{H}_i] = \sum_{i \geq 1} \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i]. \tag{9.3}$$

Next, observe that as each loop iteration works the same as any other, it follows that for each $i \geq 1$, we have

$$\mathsf{P}[(Y_i = t) \cap \mathcal{H}_i \mid L \geq i] = \mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1] = \mathsf{P}[Y_1 = t] = 2^{-\ell}.$$

Moreover, since $\mathcal{H}_i$ implies $L \geq i$, we have

$$\begin{aligned} \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i] &= \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i \cap (L \geq i)] \\ &= \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i \mid L \geq i]\, \mathsf{P}[L \geq i] = 2^{-\ell}\, \mathsf{P}[L \geq i], \end{aligned}$$

and so using (9.3) and the infinite version of Theorem 8.17 (discussed in §8.10.4), we have

$$\begin{aligned} \mathsf{P}[Y = t] &= \sum_{i \geq 1} \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i] = \sum_{i \geq 1} 2^{-\ell}\, \mathsf{P}[L \geq i] = 2^{-\ell} \sum_{i \geq 1} \mathsf{P}[L \geq i] \\ &= 2^{-\ell} \cdot \mathsf{E}[L] = 2^{-\ell} \cdot 2^{\ell}/m = 1/m. \end{aligned}$$

This shows that Y is uniformly distributed over $\{0, \ldots, m-1\}$. □

Of course, by adding an appropriate value to the output of Algorithm RN, we can generate random numbers uniformly in the interval $\{m_1, \ldots, m_2\}$, for any given m_1 and m_2. In what follows, we shall denote the execution of this algorithm as

$$y \xleftarrow{\$} \{m_1, \ldots, m_2\}.$$

More generally, if T is any finite, non-empty set for which we have an efficient algorithm whose output is uniformly distributed over T, we shall denote the execution of this algorithm as

$$y \xleftarrow{\$} T.$$

For example, we may write

$$y \xleftarrow{\$} \mathbb{Z}_m$$

to denote assignment to y of a randomly chosen element of $\mathbb{Z}_m$. Of course, this

is done by running Algorithm RN on input m, and viewing its output as a residue class modulo m.

We also mention the following alternative algorithm for generating an almost-random number from an interval.

Algorithm RN′**.** On input m, k, where both m and k are positive integers, do the following, where $\ell := \lceil \log_2 m \rceil$:

$\sigma \xleftarrow{R} \{0,1\}^{\times(\ell+k)}$
$y \leftarrow I(\sigma) \bmod m$
output y

Compared with Algorithm RN, Algorithm RN′ has the advantage that there are no loops—it always halts in a bounded number of steps; however, it has the disadvantage that its output is *not* uniformly distributed over the interval $\{0, \ldots, m-1\}$. Nevertheless, the statistical distance between its output distribution and the uniform distribution on $\{0, \ldots, m-1\}$ is at most 2^{-k} (see Example 8.41 in §8.8). Thus, by choosing k suitably large, we can make the output distribution "as good as uniform" for most practical purposes.

EXERCISE 9.4. Prove that if m is not a power of 2, there is no probabilistic algorithm whose running time is strictly bounded and whose output distribution is uniform on $\{0, \ldots, m-1\}$.

EXERCISE 9.5. You are to design and analyze an efficient probabilistic algorithm B that takes as input two integers n and y, with $n > 0$ and $0 \leq y \leq n$, and always outputs 0 or 1. Your algorithm should satisfy the following property. Suppose A is a probabilistic algorithm that takes two inputs, n and x, and always outputs an integer between 0 and n. Let Y be a random variable representing A's output on input n, x. Then for all inputs n, x, we should have $\mathsf{P}[B(n, A(n,x)) \text{ outputs } 1] = \mathsf{E}[Y]/n$.

9.3 The generate and test paradigm

Algorithm RN, which was discussed in §9.2, is a specific instance of a very general type of construction that may be called the "generate and test" paradigm.

Suppose we have two probabilistic algorithms, A and B, and we combine them to form a new algorithm

$C(x)$: repeat $y \leftarrow A(x)$ until $B(x, y)$
output y.

Here, we assume that $B(x, y)$ always outputs either *true* or *false*.

Our goal is to answer the following questions about C for a fixed input x:

1. Does C halt with probability 1?
2. What is the expected running time of C?
3. What is the output distribution of C?

The answer to the first question is "yes," provided (i) A halts with probability 1 on input x, (ii) for all possible outputs y of $A(x)$, B halts with probability 1 on input (x, y), and (iii) for some possible output y of $A(x)$, $B(x, y)$ outputs *true* with non-zero probability. We shall assume this from now on.

To address the second and third questions, let us define random variables L, Z, and Y, where L is the total number of loop iterations of C, Z is the total running time of C, and Y is the output of C. We can reduce the study of L, Z, and Y to the study of a single iteration of the main loop. Instead of working with a new probability distribution that directly models a single iteration of the loop, it is more convenient to simply study the *first* iteration of the loop in C. To this end, we define random variables Z_1 and Y_1, where Z_1 is the running time of the first loop iteration of C, and Y_1 is the value assigned to y in the first loop iteration of C. Also, let $\mathcal{H}_1$ be the event that the algorithm halts in the first loop iteration, and let T be the set of possible outputs of $A(x)$. Note that by the assumption in the previous paragraph, $\mathsf{P}[\mathcal{H}_1] > 0$.

Theorem 9.3. *Under the assumptions above,*

(i) L *has a geometric distribution with associated success probability* $\mathsf{P}[\mathcal{H}_1]$, *and in particular,* $\mathsf{E}[L] = 1/\mathsf{P}[\mathcal{H}_1]$;

(ii) $\mathsf{E}[Z] = \mathsf{E}[Z_1]\,\mathsf{E}[L] = \mathsf{E}[Z_1]/\mathsf{P}[\mathcal{H}_1]$;

(iii) for every $t \in T$, $\mathsf{P}[Y = t] = \mathsf{P}[Y_1 = t \mid \mathcal{H}_1]$.

Proof. (i) is clear.

To prove (ii), for $i \geq 1$, let Z_i be the time spent by the algorithm in the ith loop iteration, so that $Z = \sum_{i \geq 1} Z_i$. Now, the conditional distribution of Z_i given $L \geq i$ is (essentially) the same as the distribution of Z_1; moreover, $Z_i = 0$ when $L < i$. Therefore, by the law of total expectation (8.24), for each $i \geq 1$, we have

$$\mathsf{E}[Z_i] = \mathsf{E}[Z_i \mid L \geq i]\,\mathsf{P}[L \geq i] + \mathsf{E}[Z_i \mid L < i]\,\mathsf{P}[L < i] = \mathsf{E}[Z_1]\,\mathsf{P}[L \geq i].$$

We may assume that $\mathsf{E}[Z_1]$ is finite, as otherwise (ii) is trivially true. By Theorem 8.40 and the infinite version of Theorem 8.17 (discussed in §8.10.4), we have

$$\mathsf{E}[Z] = \sum_{i \geq 1} \mathsf{E}[Z_i] = \sum_{i \geq 1} \mathsf{E}[Z_1]\,\mathsf{P}[L \geq i] = \mathsf{E}[Z_1] \sum_{i \geq 1} \mathsf{P}[L \geq i] = \mathsf{E}[Z_1]\,\mathsf{E}[L].$$

To prove (iii), for $i \geq 1$, let Y_i be the value assigned to y in loop iteration i, with $Y_i := \perp$ if $L < i$, and let $\mathcal{H}_i$ be the event that the algorithm halts in loop iteration i

(i.e., $\mathcal{H}_i$ is the event that $L = i$). By a calculation similar to that made in the proof of Theorem 9.2, for each $t \in T$, we have

$$\begin{aligned}\mathsf{P}[Y = t] &= \sum_{i \geq 1} \mathsf{P}[(Y = t) \cap \mathcal{H}_i] = \sum_{i \geq 1} \mathsf{P}[(Y_i = t) \cap \mathcal{H}_i \mid L \geq i]\,\mathsf{P}[L \geq i] \\ &= \mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1] \sum_{i \geq 1} \mathsf{P}[L \geq i] = \mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1] \cdot \mathsf{E}[L] \\ &= \mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1] / \mathsf{P}[\mathcal{H}_1] = \mathsf{P}[Y_1 = t \mid \mathcal{H}_1]. \quad \square\end{aligned}$$

Example 9.10. Suppose T is a finite set, and T' is a non-empty, finite subset of T. Consider the following generalization of Algorithm RN:

repeat
    $y \xleftarrow{\$} T$
until $y \in T'$
output y

Here, we assume that we have an algorithm to generate a random element of T (i.e., uniformly distributed over T), and an efficient algorithm to test for membership in T'. Let L denote the number of loop iterations, and Y the output. Also, let Y_1 be the value of y in the first iteration, and $\mathcal{H}_1$ the event that the algorithm halts in the first iteration. Since Y_1 is uniformly distributed over T, and $\mathcal{H}_1$ is the event that $Y_1 \in T'$, we have $\mathsf{P}[\mathcal{H}_1] = |T'|/|T|$. It follows that $\mathsf{E}[L] = |T|/|T'|$. As for the output, for every $t \in T$, we have

$$\mathsf{P}[Y = t] = \mathsf{P}[Y_1 = t \mid \mathcal{H}_1] = \mathsf{P}[Y_1 = t \mid Y_1 \in T'],$$

which is 0 if $t \notin T'$ and is $1/|T'|$ if $t \in T'$. It follows that Y is uniformly distributed over T'. $\square$

Example 9.11. Let us analyze the following algorithm:

repeat
    $y \xleftarrow{\$} \{1, 2, 3, 4\}$
    $z \xleftarrow{\$} \{1, \ldots, y\}$
until $z = 1$
output y

With each loop iteration, the algorithm chooses y uniformly at random, and then decides to halt with probability $1/y$. Let L denote the number of loop iterations, and Y the output. Also, let Y_1 be the value of y in the first iteration, and $\mathcal{H}_1$ the event that the algorithm halts in the first iteration. Y_1 is uniformly distributed over

$\{1, \ldots, 4\}$, and for $t = 1, \ldots, 4$, $\mathsf{P}[\mathcal{H}_1 \mid Y_1 = t] = 1/t$. Therefore,

$$\mathsf{P}[\mathcal{H}_1] = \sum_{t=1}^{4} \mathsf{P}[\mathcal{H}_1 \mid Y_1 = t]\,\mathsf{P}[Y_1 = t] = \sum_{t=1}^{4} (1/t)(1/4) = 25/48.$$

Thus, $\mathsf{E}[L] = 48/25$. For the output distribution, for $t = 1, \ldots, 4$, we have

$$\begin{aligned}\mathsf{P}[Y = t] &= \mathsf{P}[Y_1 = t \mid \mathcal{H}_1] = \mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1] / \mathsf{P}[\mathcal{H}_1] \\ &= \mathsf{P}[\mathcal{H}_1 \mid Y_1 = t]\,\mathsf{P}[Y_1 = t] / \mathsf{P}[\mathcal{H}_1] = (1/t)(1/4)(48/25) = \frac{12}{25t}.\end{aligned}$$

This example illustrates how a probabilistic test can be used to create a biased output distribution. □

EXERCISE 9.6. Design and analyze an efficient probabilistic algorithm that takes as input an integer $n \geq 2$, and outputs a random element of $\mathbb{Z}_n^*$.

EXERCISE 9.7. Consider the following probabilistic algorithm that takes as input a positive integer m:

$S \leftarrow \emptyset$
repeat
    $n \xleftarrow{\$} \{1, \ldots, m\}$, $S \leftarrow S \cup \{n\}$
until $|S| = m$

Show that the expected number of iterations of the main loop is $\sim m \log m$.

EXERCISE 9.8. Consider the following algorithm (which takes no input):

$j \leftarrow 1$
repeat
    $j \leftarrow j + 1$, $n \xleftarrow{\$} \{0, \ldots, j - 1\}$
until $n = 0$

Show that the expected running time of this algorithm is infinite (even though it does halt with probability 1).

EXERCISE 9.9. Now consider the following modification to the algorithm in the previous exercise:

$j \leftarrow 2$
repeat
    $j \leftarrow j + 1$, $n \xleftarrow{\$} \{0, \ldots, j - 1\}$
until $n = 0$ or $n = 1$

Show that the expected running time of this algorithm is finite.

EXERCISE 9.10. Consider again Algorithm RN in §9.2. On input m, this algorithm may use up to $\approx 2\ell$ random bits on average, where $\ell := \lceil \log_2 m \rceil$. Indeed, each loop iteration generates ℓ random bits, and the expected number of loop iterations will be ≈ 2 when $m \approx 2^{\ell-1}$. This exercise asks you to analyze an alternative algorithm that uses just $\ell + O(1)$ random bits on average, which may be useful in settings where random bits are a scarce resource. This algorithm runs as follows:

```
repeat
    y ← 0,  i ← 1
    while y < m and i ≤ ℓ do
(∗)         b ←$ {0,1},  y ← y + 2^{ℓ−i} b,  i ← i + 1
until y < m
output y
```

Define random variables K and Y, where K is the number of times the line marked $(*)$ is executed, and Y is the output. Show that $\mathsf{E}[K] = \ell + O(1)$ and that Y is uniformly distributed over $\{0, \ldots, m-1\}$.

EXERCISE 9.11. Let S and T be finite, non-empty sets, and let $f : S \times T \to \{-1, 0, 1\}$ be a function. Consider the following probabilistic algorithm:

```
x ←$ S, y ←$ T
if f(x, y) = 0 then
    y′ ← y
else
    y′ ←$ T
(∗) while f(x, y′) = 0 do y′ ←$ T
```

Here, we assume we have algorithms to generate random elements in S and T, and a deterministic algorithm to evaluate f. Define random variables X, Y, Y', and L, where X is the value assigned to x, Y is the value assigned to y, Y' is the *final* value assigned to y', and L is the number of times that f is evaluated at the line marked $(*)$.

(a) Show that (X, Y') has the same distribution as (X, Y).

(b) Show that $\mathsf{E}[L] \leq 1$.

(c) Give an explicit example of S, T, and f, such that if the line marked $(*)$ is deleted, then $\mathsf{E}[f(X, Y)] > \mathsf{E}[f(X, Y')] = 0$.

9.4 Generating a random prime

Suppose we are given an integer $m \geq 2$, and want to generate a random prime between 2 and m. One way to proceed is simply to generate random numbers until we get a prime. This idea will work, assuming the existence of an efficient, deterministic algorithm *IsPrime* that determines whether or not a given integer is prime. We will present such an algorithm later, in Chapter 21. For the moment, we shall just assume we have such an algorithm, and use it as a "black box." Let us assume that on inputs of bit length at most ℓ, *IsPrime* runs in time at most $\tau(\ell)$. Let us also assume (quite reasonably) that $\tau(\ell) = \Omega(\ell)$.

Algorithm RP**.** On input m, where m is an integer ≥ 2, do the following:

repeat
    $n \xleftarrow{\varepsilon} \{2, \ldots, m\}$
until *IsPrime*(n)
output n

We now wish to analyze the running time and output distribution of Algorithm RP on an input m, where $\ell := \mathrm{len}(m)$. This is easily done, using the results of §9.3, and more specifically, by Example 9.10. The expected number of loop iterations performed by Algorithm RP is $(m-1)/\pi(m)$, where $\pi(m)$ is the number of primes up to m. By Chebyshev's theorem (Theorem 5.1), $\pi(m) = \Theta(m/\ell)$. It follows that the expected number of loop iterations is $\Theta(\ell)$. Furthermore, the expected running time of any one loop iteration is $O(\tau(\ell))$ (the expected running time for generating n is $O(\ell)$, and this is where we use the assumption that $\tau(\ell) = \Omega(\ell)$). It follows that the expected total running time is $O(\ell\tau(\ell))$. As for the output, it is clear that it is uniformly distributed over the set of primes up to m.

9.4.1 Using a probabilistic primality test

In the above analysis, we assumed that *IsPrime* was an efficient, deterministic algorithm. While such an algorithm exists, there are in fact simpler and far more efficient primality tests that are probabilistic. We shall discuss such an algorithm in detail in the next chapter. This algorithm (like several other probabilistic primality tests) has *one-sided error*, in the following sense: if the input n is prime, then the algorithm always outputs *true*; otherwise, if n is composite, the output may be *true* or *false*, but the probability that the output is *true* is at most ε, where ε is a very small number (the algorithm may be easily tuned to make ε quite small, e.g., 2^{-100}).

Let us analyze the behavior of Algorithm RP under the assumption that *IsPrime* is implemented by a probabilistic algorithm with an error probability for composite

inputs bounded by ε, as discussed in the previous paragraph. Let $\bar{\tau}(\ell)$ be a bound on the expected running time of this algorithm for all inputs of bit length at most ℓ. Again, we assume that $\bar{\tau}(\ell) = \Omega(\ell)$.

We use the technique developed in §9.3. Consider a fixed input m, and let $\ell := \text{len}(m)$. Let L, Z, and N be random variables representing, respectively, the number of loop iterations, the total running time, and output of Algorithm RP on input m. Also, let Z_1 be the random variable representing the running time of the first loop iteration, and let N_1 be the random variable representing the value assigned to n in the first loop iteration. Let $\mathcal{H}_1$ be the event that the algorithm halts in the first loop iteration, and let C_1 be the event that N_1 is composite.

Clearly, N_1 is uniformly distributed over $\{2, \ldots, m\}$. Also, by our assumptions about *IsPrime*, we have

$$\mathsf{E}[Z_1] = O(\bar{\tau}(\ell)),$$

and moreover, for each $j \in \{2, \ldots, m\}$, we have

$$\mathsf{P}[\mathcal{H}_1 \mid N_1 = j] \leq \varepsilon \text{ if } j \text{ is composite},$$

and

$$\mathsf{P}[\mathcal{H}_1 \mid N_1 = j] = 1 \text{ if } j \text{ is prime}.$$

In particular,

$$\mathsf{P}[\mathcal{H}_1 \mid C_1] \leq \varepsilon \text{ and } \mathsf{P}[\mathcal{H}_1 \mid \overline{C}_1] = 1.$$

It follows that

$$\begin{aligned} \mathsf{P}[\mathcal{H}_1] &= \mathsf{P}[\mathcal{H}_1 \mid C_1]\,\mathsf{P}[C_1] + \mathsf{P}[\mathcal{H}_1 \mid \overline{C}_1]\,\mathsf{P}[\overline{C}_1] \geq \mathsf{P}[\mathcal{H}_1 \mid \overline{C}_1]\,\mathsf{P}[\overline{C}_1] \\ &= \pi(m)/(m-1). \end{aligned}$$

Therefore,

$$\mathsf{E}[L] \leq (m-1)/\pi(m) = O(\ell)$$

and

$$\mathsf{E}[Z] = \mathsf{E}[L]\,\mathsf{E}[Z_1] = O(\ell\bar{\tau}(\ell)).$$

That takes care of the running time. Now consider the output. For every $j \in \{2, \ldots, m\}$, we have

$$\mathsf{P}[N = j] = \mathsf{P}[N_1 = j \mid \mathcal{H}_1].$$

If j is prime, then

$$\mathsf{P}[N = j] = \mathsf{P}[N_1 = j \mid \mathcal{H}_1] = \frac{\mathsf{P}[(N_1 = j) \cap \mathcal{H}_1]}{\mathsf{P}[\mathcal{H}_1]}$$
$$= \frac{\mathsf{P}[\mathcal{H}_1 \mid N_1 = j]\,\mathsf{P}[N_1 = j]}{\mathsf{P}[\mathcal{H}_1]} = \frac{1}{(m-1)\,\mathsf{P}[\mathcal{H}_1]}.$$

Thus, every prime is output with equal probability; however, the algorithm may also output a number that is not prime. Let us bound the probability of this event. One might be tempted to say that this happens with probability at most ε; however, in drawing such a conclusion, one would be committing the fallacy of Example 8.13—to correctly analyze the probability that Algorithm RP mistakenly outputs a composite, one must take into account the rate of incidence of the "primality disease," as well as the error rate of the test for this disease. Indeed, if C is the event that N is composite, then we have

$$\mathsf{P}[C] = \mathsf{P}[C_1 \mid \mathcal{H}_1] = \frac{\mathsf{P}[C_1 \cap \mathcal{H}_1]}{\mathsf{P}[\mathcal{H}_1]} = \frac{\mathsf{P}[\mathcal{H}_1 \mid C_1]\,\mathsf{P}[C_1]}{\mathsf{P}[\mathcal{H}_1]}$$
$$\leq \frac{\varepsilon}{\mathsf{P}[\mathcal{H}_1]} \leq \frac{\varepsilon}{\pi(m)/(m-1)} = O(\ell\varepsilon).$$

Another way of analyzing the output distribution of Algorithm RP is to consider its statistical distance Δ from the uniform distribution on the set of primes between 2 and m. As we have already argued, every prime between 2 and m is equally likely to be output, and in particular, any fixed prime is output with probability at most $1/\pi(m)$. It follows from Theorem 8.31 that $\Delta = \mathsf{P}[C] = O(\ell\varepsilon)$.

9.4.2 Generating a random ℓ-bit prime

Instead of generating a random prime between 2 and m, we may instead want to generate a random ℓ-bit prime, that is, a prime between $2^{\ell-1}$ and $2^\ell - 1$. Bertrand's postulate (Theorem 5.8) tells us that there exist such primes for every $\ell \geq 2$, and that in fact, there are $\Omega(2^\ell/\ell)$ such primes. Because of this, we can modify Algorithm RP, so that each candidate n is chosen at random from the interval $\{2^{\ell-1}, \ldots, 2^\ell - 1\}$, and all of the results for that algorithm carry over essentially without change. In particular, the expected number of trials until the algorithm halts is $O(\ell)$, and if a probabilistic primality test as in §9.4.1 is used, with an error probability of ε, the probability that the output is not prime is $O(\ell\varepsilon)$.

EXERCISE 9.12. Suppose Algorithm RP is implemented using an imperfect random number generator, so that the statistical distance between the output distribution of the random number generator and the uniform distribution on $\{2, \ldots, m\}$ is

equal to δ (e.g., Algorithm RN′ in §9.2). Assume that $2\delta < \pi(m)/(m-1)$. Also, let μ denote the expected number of iterations of the main loop of Algorithm RP, let Δ denote the statistical distance between its output distribution and the uniform distribution on the primes up to m, and let $\ell := \mathrm{len}(m)$.

(a) Assuming the primality test is deterministic, show that $\mu = O(\ell)$ and $\Delta = O(\delta\ell)$.

(b) Assuming the primality test is probabilistic, with one-sided error ε, as in §9.4.1, show that $\mu = O(\ell)$ and $\Delta = O((\delta + \varepsilon)\ell)$.

9.5 Generating a random non-increasing sequence

The following algorithm will be used in the next section as a fundamental subroutine in a beautiful algorithm (Algorithm RFN) that generates random numbers in *factored form*.

Algorithm RS. On input m, where m is an integer ≥ 2, do the following:

$n_0 \leftarrow m$
$k \leftarrow 0$
repeat
    $k \leftarrow k + 1$
    $n_k \xleftarrow{\$} \{1, \ldots, n_{k-1}\}$
until $n_k = 1$
output $(n_1, \ldots, n_k)$

We analyze first the output distribution, and then the running time.

9.5.1 Analysis of the output distribution

Let $N_1, N_2, \ldots$ be random variables denoting the choices of $n_1, n_2, \ldots$ (for completeness, define $N_i := 1$ if loop i is never entered).

A particular output of the algorithm is a non-increasing sequence $(j_1, \ldots, j_h)$, where $j_1 \geq j_2 \geq \cdots \geq j_{h-1} > j_h = 1$. For any such sequence, we have

$$\mathsf{P}\Big[\bigcap_{v=1}^{h}(N_v = j_v)\Big] = \mathsf{P}[N_1 = j_1] \cdot \prod_{v=2}^{h} \mathsf{P}\Big[N_v = j_v \mid \bigcap_{w<v}(N_w = j_w)\Big]$$
$$= \frac{1}{m} \cdot \frac{1}{j_1} \cdots \frac{1}{j_{h-1}}. \tag{9.4}$$

This completely describes the output distribution, in the sense that we have determined the probability with which each non-increasing sequence appears as an output. However, there is another way to characterize the output distribution

that is significantly more useful. For $j = 2, \ldots, m$, define the random variable O_j to be the number of occurrences of the integer j in the output sequence. The O_j's determine the N_i's, and *vice versa*. Indeed, $O_m = e_m, \ldots, O_2 = e_2$ if and only if the output of the algorithm is the sequence

$$(\underbrace{m, \ldots, m}_{e_m \text{ times}}, \underbrace{m-1, \ldots, m-1}_{e_{m-1} \text{ times}}, \ldots, \underbrace{2, \ldots, 2}_{e_2 \text{ times}}, 1).$$

From (9.4), we can therefore directly compute

$$\mathsf{P}\Big[\bigcap_{j=2}^{m}(O_j = e_j)\Big] = \frac{1}{m}\prod_{j=2}^{m}\frac{1}{j^{e_j}}. \tag{9.5}$$

Moreover, we can write $1/m$ as a telescoping product,

$$\frac{1}{m} = \frac{m-1}{m} \cdot \frac{m-2}{m-1} \cdot \cdots \cdot \frac{2}{3} \cdot \frac{1}{2} = \prod_{j=2}^{m}(1 - 1/j),$$

and so re-write (9.5) as

$$\mathsf{P}\Big[\bigcap_{j=2}^{m}(O_j = e_j)\Big] = \prod_{j=2}^{m} j^{-e_j}(1 - 1/j). \tag{9.6}$$

Notice that for $j = 2, \ldots, m$,

$$\sum_{e_j \geq 0} j^{-e_j}(1 - 1/j) = 1,$$

and so by (a discrete version of) Theorem 8.7, the family of random variables $\{O_j\}_{j=2}^{m}$ is mutually independent, and for each $j = 2, \ldots, m$ and each integer $e_j \geq 0$, we have

$$\mathsf{P}[O_j = e_j] = j^{-e_j}(1 - 1/j). \tag{9.7}$$

In summary, we have shown:

> *that the family $\{O_j\}_{j=2}^{m}$ is mutually independent, where for each $j = 2, \ldots, m$, the variable $O_j + 1$ has a geometric distribution with an associated success probability of $1 - 1/j$.*

Another, perhaps more intuitive, analysis of the distribution of the O_j's runs as follows. Conditioning on the event $O_m = e_m, \ldots, O_{j+1} = e_{j+1}$, one sees that the value of O_j is the number of times the value j appears in the sequence $N_i, N_{i+1}, \ldots$, where $i = e_m + \cdots + e_{j+1} + 1$; moreover, in this conditional probability distribution, it is not too hard to convince oneself that N_i is uniformly distributed over $\{1, \ldots, j\}$. Hence the probability that $O_j = e_j$ in this conditional probability distribution is the

probability of getting a run of exactly e_j copies of the value j in an experiment in which we successively choose numbers between 1 and j at random, and this latter probability is clearly $j^{-e_j}(1 - 1/j)$.

9.5.2 Analysis of the running time

Let $\ell := \mathrm{len}(m)$, and let K be the random variable that represents the number of loop iterations performed by the algorithm. With the random variables $O_2, \ldots, O_m$ defined as above, we can write $K = 1 + \sum_{j=2}^{m} O_j$. Moreover, for each j, $O_j + 1$ has a geometric distribution with associated success probability $1 - 1/j$, and hence

$$\mathsf{E}[O_j] = \frac{1}{1 - 1/j} - 1 = \frac{1}{j - 1}.$$

Thus,

$$\mathsf{E}[K] = 1 + \sum_{j=2}^{m} \mathsf{E}[O_j] = 1 + \sum_{j=1}^{m-1} \frac{1}{j} \leq 2 + \int_1^m \frac{dy}{y} = \log m + 2,$$

where we have estimated the sum by an integral (see §A5).

Intuitively, this is roughly as we would expect, since with probability $1/2$, each successive n_i is at most one half as large as its predecessor, and so after $O(\ell)$ steps, we expect to reach 1.

Let Z be the total running time of the algorithm. We may bound $\mathsf{E}[Z]$ using essentially the same argument that was used in the proof of Theorem 9.3. First, write $Z = \sum_{i \geq 1} Z_i$, where Z_i is the time spent in the ith loop iteration. Each loop iteration, if executed at all, runs in expected time $O(\ell)$. That is, there exists a constant c, such that for each $i \geq 1$,

$$\mathsf{E}[Z_i \mid K \geq i] \leq c\ell \quad \text{and} \quad \mathsf{E}[Z_i \mid K < i] = 0.$$

Thus,

$$\mathsf{E}[Z_i] = \mathsf{E}[Z_i \mid K \geq i]\,\mathsf{P}[K \geq i] + \mathsf{E}[Z_i \mid K < i]\,\mathsf{P}[K < i] \leq c\ell\,\mathsf{P}[K \geq i],$$

and so

$$\mathsf{E}[Z] = \sum_{i \geq 1} \mathsf{E}[Z_i] \leq c\ell \sum_{i \geq 1} \mathsf{P}[K \geq i] = c\ell\,\mathsf{E}[K] = O(\ell^2).$$

In summary, we have shown:

the expected running time of Algorithm RS *on ℓ-bit inputs is $O(\ell^2)$.*

EXERCISE 9.13. Show that when Algorithm RS runs on input m, the expected number of (not necessarily distinct) primes in the output sequence is $\sim \log \log m$.

9.6 Generating a random factored number

We now present an efficient algorithm that generates a random factored number. That is, on input $m \geq 2$, the algorithm generates a number y uniformly distributed over the interval $\{1, \ldots, m\}$, but instead of the usual output format for such a number y, the output consists of the prime factorization of y.

As far as anyone knows, there are no efficient algorithms for factoring large numbers, despite years of active research in search of such an algorithm. So our algorithm to generate a random factored number will *not* work by generating a random number and then factoring it.

Our algorithm will use Algorithm RS in §9.5 as a subroutine. In addition, as we did in §9.4, we shall assume the existence of an efficient, deterministic primality test *IsPrime*. In the analysis of the algorithm, we shall make use of Mertens' theorem, which we proved in Chapter 5 (Theorem 5.13).

Algorithm RFN**.** On input m, where m is an integer ≥ 2, do the following:

```
    repeat
        run Algorithm RS on input m, obtaining (n_1, ..., n_k)
(*)     let (p_1, ..., p_r) be the subsequence of primes in (n_1, ..., n_k)
(**)    y ← p_1 ··· p_r
        if y ≤ m then
            x ←$ {1, ..., m}
            if x ≤ y then output (p_1, ..., p_r) and halt
    forever
```

Notes:

(*) For $i = 1, \ldots, k-1$, the number n_i is tested for primality using algorithm *IsPrime*. The sequence $(n_1, \ldots, n_k)$ may contain duplicates, and if these are prime, they are appear in $(p_1, \ldots, p_r)$ with the same multiplicity.

(**) We assume that the product is computed by a simple iterative procedure that halts as soon as the partial product exceeds m. This ensures that the time spent forming the product is always $O(\mathrm{len}(m)^2)$, which simplifies the analysis.

We now analyze the running time and output distribution of Algorithm RFN on input m, using the generate-and-test paradigm discussed in §9.3; here, the "generate" part consists of the first two lines of the loop body, which generates the sequence $(p_1, \ldots, p_r)$, while the "test" part consists of the last four lines of the loop body.

Let $\ell := \mathrm{len}(m)$. We assume that each call to *IsPrime* takes time at most $\tau(\ell)$, and for simplicity, we assume $\tau(\ell) = \Omega(\ell)$.

Let K_1 be the value of k in the first loop iteration, Z_1 be the running time of

the first loop iteration, Y_1 be the value of y in the first loop iteration, and $\mathcal{H}_1$ be the event that the algorithm halts in the first loop iteration. Also, let Z be the total running time of the algorithm, and let Y be the value of y in the last loop iteration (i.e., the number whose factorization is output).

We begin with three preliminary calculations.

First, let $t = 1, \ldots, m$ be a fixed integer, and let us calculate the probability that $Y_1 = t$. Suppose $t = \prod_{p \le m} p^{e_p}$ is the prime factorization of t. Let $O_2, \ldots, O_m$ be random variables as defined in §9.5, so that O_j represents the number of occurrences of j in the output sequence of the first invocation of Algorithm RS. Then $Y_1 = t$ if and only if $O_p = e_p$ for all primes $p \le m$, and so by the analysis in §9.5, we have

$$\mathsf{P}[Y_1 = t] = \prod_{p \le m} p^{-e_p}(1 - 1/p) = \frac{g(m)}{t},$$

where

$$g(m) := \prod_{p \le m}(1 - 1/p).$$

Second, we calculate $\mathsf{P}[\mathcal{H}_1]$. Observe that for $t = 1, \ldots, m$, we have

$$\mathsf{P}[\mathcal{H}_1 \mid Y_1 = t] = t/m,$$

and so

$$\mathsf{P}[\mathcal{H}_1] = \sum_{t=1}^{m} \mathsf{P}[\mathcal{H}_1 \mid Y_1 = t]\,\mathsf{P}[Y_1 = t] = \sum_{t=1}^{m} \frac{t}{m}\frac{g(m)}{t} = g(m).$$

Third, let $t = 1, \ldots, m$ be a fixed integer, and let us calculate the conditional probability that $Y_1 = t$ given $\mathcal{H}_1$. We have

$$\begin{aligned}\mathsf{P}[Y_1 = t \mid \mathcal{H}_1] &= \frac{\mathsf{P}[(Y_1 = t) \cap \mathcal{H}_1]}{\mathsf{P}[\mathcal{H}_1]} = \frac{\mathsf{P}[\mathcal{H}_1 \mid Y_1 = t]\,\mathsf{P}[Y_1 = t]}{\mathsf{P}[\mathcal{H}_1]} \\ &= \frac{(t/m)(g(m)/t)}{g(m)} = \frac{1}{m}.\end{aligned}$$

We may now easily analyze the output distribution of Algorithm RFN. By Theorem 9.3, for each $t = 1, \ldots, m$, we have

$$\mathsf{P}[Y = t] = \mathsf{P}[Y_1 = t \mid \mathcal{H}_1] = \frac{1}{m},$$

which shows that the output is indeed uniformly distributed over all integers in $\{1, \ldots, m\}$, represented in factored form.

Finally, we analyze the expected running time of Algorithm RFN. It is easy to

see that $\mathsf{E}[Z_1] = O(\mathsf{E}[K_1]\tau(\ell) + \ell^2)$, and by the analysis in §9.5, we know that $\mathsf{E}[K_1] = O(\ell)$, and hence $\mathsf{E}[Z_1] = O(\ell\tau(\ell))$. By Theorem 9.3, we have

$$\mathsf{E}[Z] = \mathsf{E}[Z_1]/\,\mathsf{P}[\mathcal{H}_1] = \mathsf{E}[Z_1]g(m)^{-1}.$$

By Mertens' theorem, $g(m)^{-1} = O(\ell)$. We conclude that

$$\mathsf{E}[Z] = O(\ell^2\tau(\ell)).$$

That is, the expected running time of Algorithm RFN is $O(\ell^2\tau(\ell))$.

9.6.1 Using a probabilistic primality test (∗)

Analogous to the discussion in §9.4.1, we can analyze the behavior of Algorithm RFN under the assumption that *IsPrime* is a probabilistic algorithm which may erroneously indicate that a composite number is prime with probability at most ε. Let $\ell := \mathrm{len}(m)$, and as we did in §9.4.1, let $\bar{\tau}(\ell)$ be a bound on the expected running time of *IsPrime* for all inputs of bit length at most ℓ (and again, assume $\bar{\tau}(\ell) = \Omega(\ell)$).

The random variables K_1, Z_1, Y_1, Z, Y and the event $\mathcal{H}_1$ are defined as above. Let us also define $\mathcal{F}_1$ to be the event that the primality test makes a mistake in the first loop iteration, and $\mathcal{F}$ to be the event that the output of the algorithm is not a list of primes. Let $\delta := \mathsf{P}[\mathcal{F}_1]$.

Again, we begin with three preliminary calculations.

First, let $t = 1, \ldots, m$ be fixed and let us calculate $\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1]$. To do this, define the random variable Y_1' to be the product of the actual primes among the output of the first invocation of Algorithm RS (because the primality test may err, Y_1 may contain additional factors). Evidently, the events $(Y_1 = t) \cap \overline{\mathcal{F}}_1$ and $(Y_1' = t) \cap \overline{\mathcal{F}}_1$ are the same. Moreover, we claim that the events $Y_1' = t$ and $\overline{\mathcal{F}}_1$ are independent. To see this, recall that the family $\{O_j\}_{j=2}^m$ is mutually independent, and also observe that the event $Y_1' = t$ depends only on the random variables O_j, where j is prime, while the event $\overline{\mathcal{F}}_1$ depends only on the random variables O_j, where j is composite, along with the execution paths of *IsPrime* on corresponding inputs. Thus, by a calculation analogous to one we made above,

$$\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1] = \mathsf{P}[Y_1' = t]\,\mathsf{P}[\overline{\mathcal{F}}_1] = \frac{g(m)}{t}(1 - \delta).$$

Second, we calculate $\mathsf{P}[\mathcal{H}_1 \cap \overline{\mathcal{F}}_1]$. Observe that for $t = 1, \ldots, m$, we have

$$\mathsf{P}[\mathcal{H}_1 \mid (Y_1 = t) \cap \overline{\mathcal{F}}_1] = t/m,$$

and so

$$\begin{aligned}\mathsf{P}[\mathcal{H}_1 \cap \overline{\mathcal{F}}_1] &= \sum_{t=1}^{m} \mathsf{P}[\mathcal{H}_1 \cap (Y_1 = t) \cap \overline{\mathcal{F}}_1] \\ &= \sum_{t=1}^{m} \mathsf{P}[\mathcal{H}_1 \mid (Y_1 = t) \cap \overline{\mathcal{F}}_1]\,\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1] \\ &= \sum_{t=1}^{m} \frac{t}{m}\frac{g(m)}{t}(1-\delta) = g(m)(1-\delta).\end{aligned}$$

Third, let $t = 1, \ldots, m$ be a fixed integer, and let us calculate the conditional probability that $(Y_1 = t) \cap \overline{\mathcal{F}}_1$ given $\mathcal{H}_1$. We have

$$\begin{aligned}\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1 \mid \mathcal{H}_1] &= \frac{\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1 \cap \mathcal{H}_1]}{\mathsf{P}[\mathcal{H}_1]} \\ &= \frac{\mathsf{P}[\mathcal{H}_1 \mid (Y_1 = t) \cap \overline{\mathcal{F}}_1]\,\mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1]}{\mathsf{P}[\mathcal{H}_1]} \\ &= \frac{(t/m)((1-\delta)g(m)/t)}{\mathsf{P}[\mathcal{H}_1]} = \frac{g(m)(1-\delta)}{m\,\mathsf{P}[\mathcal{H}_1]}.\end{aligned}$$

We may now easily analyze the output distribution of Algorithm RFN. By Theorem 9.3, for each $t = 1, \ldots, m$, we have

$$\mathsf{P}[(Y = t) \cap \overline{\mathcal{F}}] = \mathsf{P}[(Y_1 = t) \cap \overline{\mathcal{F}}_1 \mid \mathcal{H}_1] = \frac{g(m)(1-\delta)}{m\,\mathsf{P}[\mathcal{H}_1]}.$$

Thus, every integer between 1 and m is equally likely to be output by Algorithm RFN in correct factored form.

Let us also calculate an upper bound on the probability $\mathsf{P}[\mathcal{F}]$ that Algorithm RFN outputs an integer that is not in correct factored form. Making use of Exercise 8.1, we have

$$\mathsf{P}[\mathcal{F}_1 \mid \mathcal{H}_1] = \frac{\mathsf{P}[\mathcal{F}_1 \cap \mathcal{H}_1]}{\mathsf{P}[\mathcal{H}_1]} \leq \frac{\mathsf{P}[\mathcal{F}_1]}{\mathsf{P}[\mathcal{F}_1 \cup \mathcal{H}_1]}.$$

Moreover,

$$\begin{aligned}\mathsf{P}[\mathcal{F}_1 \cup \mathcal{H}_1] &= \mathsf{P}[\mathcal{F}_1] + \mathsf{P}[\mathcal{H}_1 \cap \overline{\mathcal{F}}_1] = \delta + g(m)(1-\delta) \\ &\geq g(m)\delta + g(m)(1-\delta) = g(m).\end{aligned}$$

By Theorem 9.3, it follows that

$$\mathsf{P}[\mathcal{F}] = \mathsf{P}[\mathcal{F}_1 \mid \mathcal{H}_1] \leq \delta/g(m).$$

Now, the reader may verify that

$$\delta \leq \varepsilon \cdot (\mathsf{E}[K_1] - 1),$$

and by our calculations in §9.5, $\mathsf{E}[K_1] \leq \log m + 2$. Thus,

$$\delta \leq \varepsilon \cdot (\log m + 1),$$

and so by Mertens' theorem,

$$\mathsf{P}[\mathcal{F}] = O(\ell^2 \varepsilon).$$

We may also analyze the statistical distance Δ between the output distribution of Algorithm RFN and the uniform distribution on $\{1, \ldots, m\}$ (in factored form). It follows from Theorem 8.31 that $\Delta = \mathsf{P}[\mathcal{F}] \leq \delta / g(m) = O(\ell^2 \varepsilon)$.

Finally, we analyze the expected running time of Algorithm RFN. We have

$$\mathsf{P}[\mathcal{H}_1] \geq \mathsf{P}[\mathcal{H}_1 \cap \overline{\mathcal{F}}_1] = g(m)(1 - \delta).$$

We leave it to the reader to verify that $\mathsf{E}[Z_1] = O(\ell \bar{\tau}(\ell))$, from which it follows by Theorem 9.3 that

$$\mathsf{E}[Z] = \mathsf{E}[Z_1] / \mathsf{P}[\mathcal{H}_1] = O(\ell^2 \bar{\tau}(\ell) / (1 - \delta)).$$

If ε is moderately small, so that $\varepsilon(\log m + 1) \leq 1/2$, and hence $\delta \leq 1/2$, then

$$\mathsf{E}[Z] = O(\ell^2 \bar{\tau}(\ell)).$$

9.7 Some complexity theory

We close this chapter with a few observations about probabilistic algorithms from a more "complexity theoretic" point of view.

Suppose f is a function mapping bit strings to bit strings. We may have an algorithm A that **approximately computes** f in the following sense: there exists a constant ε, with $0 \leq \varepsilon < 1/2$, such that for all inputs x, $A(x)$ outputs $f(x)$ with probability at least $1 - \varepsilon$. The value ε is a bound on the **error probability**, which is defined as the probability that $A(x)$ does not output $f(x)$.

9.7.1 Reducing the error probability

There is a standard "trick" by which one can make the error probability very small; namely, run A on input x some number, say k, times, and take the majority output as the answer. Suppose $\varepsilon < 1/2$ is a bound on the error probability. Using the Chernoff bound (Theorem 8.24), the error probability for the iterated version of A is bounded by

$$\exp[-(1/2 - \varepsilon)^2 k/2], \tag{9.8}$$

and so the error probability decreases exponentially with the number of iterations. This bound is derived as follows. For $i = 1, \ldots, k$, let X_i be the indicator variable

for the event that the ith iteration of $A(x)$ does not output $f(x)$. The expected value of the sample mean $\overline{X} := \frac{1}{k}\sum_{i=1}^{k} X_i$ is at most ε, and if the majority output of the iterated algorithm is wrong (or indeed, if there is no majority), then $\overline{X}$ exceeds its expectation by at least $1/2 - \varepsilon$. The bound (9.8) follows immediately from part (i) of Theorem 8.24.

9.7.2 Strict polynomial time

If we have an algorithm A that runs in expected polynomial time, and which approximately computes a function f, then we can easily turn it into a new algorithm A' that runs in *strict* polynomial time, and also approximates f, as follows. Suppose that $\varepsilon < 1/2$ is a bound on the error probability, and $Q(n)$ is a polynomial bound on the expected running time for inputs of size n. Then A' simply runs A for at most $kQ(n)$ steps, where k is any constant chosen so that $\varepsilon + 1/k < 1/2$—if A does not halt within this time bound, then A' simply halts with an arbitrary output. The probability that A' errs is at most the probability that A errs plus the probability that A runs for more than $kQ(n)$ steps. By Markov's inequality (Theorem 8.22), the latter probability is at most $1/k$, and hence A' approximates f as well, but with an error probability bounded by $\varepsilon + 1/k$.

9.7.3 Language recognition

An important special case of approximately computing a function is when the output of the function f is either 0 or 1 (or equivalently, *false* or *true*). In this case, f may be viewed as the characteristic function of the language $L := \{x : f(x) = 1\}$. (It is the tradition of computational complexity theory to call sets of bit strings "languages.") There are several "flavors" of probabilistic algorithms for approximately computing the characteristic function f of a language L that are traditionally considered—for the purposes of these definitions, we may restrict ourselves to algorithms that output either 0 or 1:

- We call a probabilistic, expected polynomial-time algorithm an **Atlantic City algorithm** for recognizing L if it approximately computes f with error probability bounded by a constant $\varepsilon < 1/2$.
- We call a probabilistic, expected polynomial-time algorithm A a **Monte Carlo algorithm** for recognizing L if for some constant $\delta > 0$, we have:
  - $\mathsf{P}[A(x) \text{ outputs } 1] \geq \delta$ for all $x \in L$;
  - $\mathsf{P}[A(x) \text{ outputs } 1] = 0$ for all $x \notin L$.
- We call a probabilistic, expected polynomial-time algorithm a **Las Vegas algorithm** for recognizing L if it computes f correctly on all inputs x.

One also says an Atlantic City algorithm has **two-sided error**, a Monte Carlo algorithm has **one-sided error**, and a Las Vegas algorithm has **zero-sided error**.

EXERCISE 9.14. Show that every language recognized by a Las Vegas algorithm is also recognized by a Monte Carlo algorithm, and that every language recognized by a Monte Carlo algorithm is also recognized by an Atlantic City algorithm.

EXERCISE 9.15. Show that if L is recognized by an Atlantic City algorithm that runs in expected polynomial time, then it is recognized by an Atlantic City algorithm that runs in strict polynomial time, and whose error probability is at most 2^{-n} on inputs of size n.

EXERCISE 9.16. Show that if L is recognized by a Monte Carlo algorithm that runs in expected polynomial time, then it is recognized by a Monte Carlo algorithm that runs in strict polynomial time, and whose error probability is at most 2^{-n} on inputs of size n.

EXERCISE 9.17. Show that a language is recognized by a Las Vegas algorithm if and only if the language and its complement are recognized by Monte Carlo algorithms.

EXERCISE 9.18. Show that if L is recognized by a Las Vegas algorithm that runs in strict polynomial time, then L may be recognized in deterministic polynomial time.

EXERCISE 9.19. Suppose that for a given language L, there exists a probabilistic algorithm A that runs in expected polynomial time, and always outputs either 0 or 1. Further suppose that for some constants α and c, where

- α is a rational number with $0 \leq \alpha < 1$, and
- c is a positive integer,

and for all sufficiently large n, and all inputs x of size n, we have

- if $x \notin L$, then $\mathsf{P}[A(x) \text{ outputs } 1] \leq \alpha$, and
- if $x \in L$, then $\mathsf{P}[A(x) \text{ outputs } 1] \geq \alpha + 1/n^c$.

(a) Show that there exists an Atlantic City algorithm for L.

(b) Show that if $\alpha = 0$, then there exists a Monte Carlo algorithm for L.

9.8 Notes

Our approach in §9.1 to defining the probability distribution associated with the execution of a probabilistic algorithm is not the only possible one. For example,

one could define the output distribution and expected running time of an algorithm on a given input directly, using the identities in Exercise 9.2, and avoid the construction of an underlying probability distribution altogether; however, we would then have very few tools at our disposal to analyze the behavior of an algorithm. Yet another approach is to define a distribution that models an infinite random bit string. This can be done, but requires more advanced notions from probability theory than those that have been covered in this text.

The algorithm presented in §9.6 for generating a random factored number is due to Kalai [52], although the analysis presented here is a bit different, and our analysis using a probabilistic primality test is new. Kalai's algorithm is significantly simpler, though less efficient, than an earlier algorithm due to Bach [9], which uses an expected number of $O(\ell)$ primality tests, as opposed to the $O(\ell^2)$ primality tests used by Kalai's algorithm.

See Luby [63] for an exposition of the theory of pseudo-random bit generation.