7

Probabilistic algorithms

It is sometimes useful to endow our algorithms with the ability to generate random numbers. 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 §6.9), 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.

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

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

In describing the behavior of such a **probabilistic** or **randomized algorithm** A, for any input x, we view its running time and output as random

variables, denoted $T_A(x)$ and $A(x)$, respectively. The **expected running time** of A on input x is defined as the expected value $\mathsf{E}[T_A(x)]$ of the random variable $T_A(x)$. Note that in defining expected running time, we are not considering the *input* to be drawn from some probability distribution. One could, of course, define such a notion; however, it is not always easy to come up with a distribution on the input space that reasonably models a particular real-world situation. We do not pursue this issue any more here.

We say that a probabilistic algorithm A runs in **expected polynomial time** if there exist constants c, d such that for all $n \geq 0$ and all inputs x of length n, we have $\mathsf{E}[T_A(x)] \leq n^c + d$. We say that A runs in **strict polynomial time** if there exist constants c, d such that for all n and all inputs x of length n, A *always* halts on input x within $n^c + d$, regardless of its random choices.

Defining the distributions of $T_A(x)$ and $A(x)$ is a bit tricky. Things are quite straightforward if A *always* halts on input x after a finite number of steps, regardless of the outcomes of its random choices: in this case, we can naturally view $T_A(x)$ and $A(x)$ as random variables on a uniform distribution over bit strings of some particular length—such a random bit string may be used as the source of random bits for the algorithm. However, if there is no a priori bound on the number of steps, things become more complicated: think of an algorithm that generates random bits one at a time until it generates, say, a 1 bit—just as in Example 6.29, we do not attempt to model this as a probability distribution on the uncountable set of infinite bit strings, but rather, we directly define an appropriate discrete probability distribution that models the execution of A on input x.

7.1.1 Defining the probability distribution

A warning to the reader: *the remainder of this section is a bit technical, and you might want to skip ahead to § 7.2 on first reading, if you are willing to trust your intuition regarding probabilistic algorithms.*

To motivate our definition, which may at first seem a bit strange, consider again Example 6.29. 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 σ of this special form, we assign the probability $2^{-|\sigma|}$, where $|\sigma|$ denotes the length of σ. 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, determined by the algorithm and its input.

To simplify matters, we assume that the machine produces a stream of random bits, one with every instruction executed, and if the instruction happens to be a random-bit instruction, then this is the bit it uses. For any bit string σ, we can run A on input x for up to $|\sigma|$ steps, using σ for the stream of random bits, and observe the behavior of the algorithm. The reader may wish to visualize σ as a finite path in an infinite binary tree, where we start at the root, branching to the left if the next bit in σ is a 0 bit, and branching to the right if the next bit in σ is a 1 bit. In this context, we call σ an **execution path**. Some further terminology will be helpful:

- If A halts in at most $|\sigma|$ steps, then we call σ a **complete execution path**;
- if A halts in exactly $|\sigma|$ steps, then we call σ an **exact execution path**;
- if A does not halt in fewer than $|\sigma|$ steps, then we call σ a **partial execution path**.

The sample space $\mathcal{S}$ of the probability distribution associated with A on input x consists of all *exact* execution paths. Clearly, $\mathcal{S}$ is **prefix free**; that is, no string in $\mathcal{S}$ is a proper prefix of another.

Theorem 7.1. *If $\mathcal{S}$ is a prefix-free set of bit strings, then $\sum_{\sigma\in\mathcal{S}} 2^{-|\sigma|} \leq 1$.*

Proof. We first claim that the theorem holds for any finite prefix-free set $\mathcal{S}$. We may assume that $\mathcal{S}$ is non-empty, since otherwise, the claim is trivial. We prove the claim by induction on the sum of the lengths of the elements of $\mathcal{S}$. The base case is when $\mathcal{S}$ contains just the empty string, in which case the claim is clear. If $\mathcal{S}$ contains non-empty strings, let τ be a string in $\mathcal{S}$ of maximal length, and let τ' be the prefix of length $|\tau| - 1$ of τ. Now remove from $\mathcal{S}$ all strings which have τ' as a prefix (there are either one or two such strings), and add to $\mathcal{S}$ the string τ'. It is easy to see (verify) that the resulting set $\mathcal{S}'$ is also prefix-free, and that

$$\sum_{\sigma\in\mathcal{S}} 2^{-|\sigma|} \leq \sum_{\sigma\in\mathcal{S}'} 2^{-|\sigma|}.$$

The claim now follows by induction.

For the general case, let $\sigma_1, \sigma_2, \ldots$ be a particular enumeration of $\mathcal{S}$, and consider the partial sums $S_i = \sum_{j=1}^{i} 2^{-|\sigma_j|}$ for $i = 1, 2, \ldots$. From the above claim, each of these partial sums is at most 1, from which it follows that $\lim_{i\to\infty} S_i \leq 1$. □

From the above theorem, if $\mathcal{S}$ is the sample space associated with algorithm A on input x, we have

$$S := \sum_{\sigma \in \mathcal{S}} 2^{-|\sigma|} \leq 1.$$

Assume that $S = 1$. Then we say that A **halts with probability** 1 **on input** x, and we define the distribution $\mathbf{D}_{A,x}$ associated with A on input x to be the distribution on $\mathcal{S}$ that assigns the probability $2^{-|\sigma|}$ to each bit string $\sigma \in \mathcal{S}$. We also define $T_A(x)$ and $A(x)$ as random variables on the distribution $\mathbf{D}_{A,x}$ in the natural way: for each $\sigma \in \mathcal{S}$, we define $T_A(x)$ to be $|\sigma|$ and $A(x)$ to be the output produced by A on input x using σ to drive its execution.

All of the above definitions assumed that A halts with probability 1 on input x, and indeed, we shall only be interested in algorithms that halt with probability 1 on all inputs. However, to analyze a given algorithm, we still have to prove that it halts with probability 1 on all inputs before we can use these definitions and bring to bear all the tools of discrete probability theory. To this end, it is helpful to study various finite probability distributions associated with the execution of A on input x. For every integer $k \geq 0$, let us consider the uniform distribution on bit strings of length k, and for each $j = 0, \ldots, k$, define $\mathcal{H}_j^{(k)}$ to be the event that such a random k-bit string causes A on input x to halt within j steps.

A couple of observations are in order. First, if $\mathcal{S}$ is the set of all exact execution paths for A on input x, then we have (verify)

$$\mathsf{P}[\mathcal{H}_j^{(k)}] = \sum_{\substack{\sigma \in \mathcal{S} \\ |\sigma| \leq j}} 2^{-|\sigma|}.$$

From this it follows that for all non-negative integers j, k, k' with $j \leq \min\{k, k'\}$, we have

$$\mathsf{P}[\mathcal{H}_j^{(k)}] = \mathsf{P}[\mathcal{H}_j^{(k')}].$$

Defining $H_k := \mathsf{P}[\mathcal{H}_k^{(k)}]$, it also follows that the sequence $\{H_k\}_{k \geq 0}$ is non-decreasing and bounded above by 1, and that A halts with probability 1 on input x if and only if

$$\lim_{k \to \infty} H_k = 1.$$

A simple necessary condition for halting with probability 1 on a given input is that for all partial execution paths, there exists some extension that is a complete execution path. Intuitively, if this does not hold, then with

some non-zero probability, the algorithm falls into an infinite loop. More formally, if there exists a partial execution path of length j that cannot be extended to a complete execution path, then for all $k \geq j$ we have

$$H_k \leq 1 - 2^{-j}.$$

This does not, however, guarantee halting with probability 1. A simple sufficient condition is the following:

> *There exists a bound ℓ (possibly depending on the input) such that for every partial execution path σ, there exists a complete execution path that extends σ and whose length at most $|\sigma| + \ell$.*

To see why this condition implies that A halts with probability 1, observe that if A runs for $k\ell$ steps without halting, then the probability that it does not halt within $(k+1)\ell$ steps is at most $1 - 2^{-\ell}$. More formally, let us define $\overline{H}_k := 1 - H_k$, and note that for all $k \geq 0$, we have

$$\begin{aligned}
\overline{H}_{(k+1)\ell} &= \mathsf{P}[\overline{\mathcal{H}}_{(k+1)\ell}^{((k+1)\ell)} \mid \overline{\mathcal{H}}_{k\ell}^{((k+1)\ell)}] \cdot \mathsf{P}[\overline{\mathcal{H}}_{k\ell}^{((k+1)\ell)}] \\
&\leq (1 - 2^{-\ell})\mathsf{P}[\overline{\mathcal{H}}_{k\ell}^{((k+1)\ell)}] \\
&= (1 - 2^{-\ell})\overline{H}_{k\ell},
\end{aligned}$$

and hence (by an induction argument on k), we have

$$\overline{H}_{k\ell} \leq (1 - 2^{-\ell})^k,$$

from which it follows that

$$\lim_{k \to \infty} H_k = 1.$$

It is usually fairly straightforward to verify this property for a particular algorithm "by inspection."

***Example* 7.1.** Consider the following algorithm:

```
repeat
    b ←_R {0, 1}
until b = 1
```

Since every loop is only a constant number of instructions, and since there is one chance to terminate with every loop iteration, the algorithm halts with probability 1. □

***Example* 7.2.** Consider the following algorithm:

$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    $s \leftarrow_R \{0,1\}^{\times i}$
until $s = 0^{\times i}$

For positive integer n, consider the probability p_n of executing at least n loop iterations (each p_n is defined using an appropriate finite probability distribution). We have

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

where we have made use of the estimate (iii) in §A1. As p_n does not tend to zero as $n \to \infty$, we may conclude that the algorithm does not halt with probability 1.

Note that every partial execution path can be extended to a complete execution path, but the length of the extension is not bounded. □

The following three exercises develop tools which simplify the analysis of probabilistic algorithms.

Exercise 7.1. Consider a probabilistic algorithm A that halts with probability 1 on input x, and consider the probability distribution $\mathbf{D}_{A,x}$ on the set $\mathcal{S}$ of exact execution paths. Let τ be a fixed, partial execution path, and let $\mathcal{B} \subseteq \mathcal{S}$ be the event that consists of all exact execution paths that extend τ. Show that $\mathsf{P}[\mathcal{B}] = 2^{-|\tau|}$.

Exercise 7.2. Consider a probabilistic algorithm A that halts with probability 1 on input x, and consider the probability distribution $\mathbf{D}_{A,x}$ on the set $\mathcal{S}$ of exact execution paths. For a bit string σ and an integer $k \geq 0$, let $\{\sigma\}_k$ denote the value of σ truncated to the first k bits. Suppose that $\mathcal{B} \subseteq \mathcal{S}$ is an event of the form

$$\mathcal{B} = \{\sigma \in \mathcal{S} : \phi(\{\sigma\}_k)\}$$

for some predicate ϕ and some integer $k \geq 0$. Intuitively, this means that $\mathcal{B}$ is completely determined by the first k bits of the execution path. Now consider the uniform distribution on $\{0,1\}^{\times k}$. Let us define an event $\mathcal{B}'$ in this distribution as follows. For $\sigma \in \{0,1\}^{\times k}$, let us run A on input x using the execution path σ for k steps or until A halts (whichever comes first). If the number of steps executed was t (where $t \leq k$), then we put σ in $\mathcal{B}'$ if and only if $\phi(\{\sigma\}_t)$. Show that the probability that the event $\mathcal{B}$ occurs

(with respect to the distribution $\mathbf{D}_{A,x}$) is the same as the probability that $\mathcal{B}'$ occurs (with respect to the uniform distribution on $\{0,1\}^{\times k}$). Hint: use Exercise 7.1.

The above exercise is very useful in simplifying the analysis of probabilistic algorithms. One can typically reduce the analysis of some event of interest into the analysis of a collection of events, each of which is determined by the first k bits of the execution path for some fixed k. The probability of an event that is determined by the first k bits of the execution path may then be calculated by analyzing the behavior of the algorithm on a random k-bit execution path.

Exercise 7.3. Suppose algorithm A calls algorithm B as a subroutine. In the probability distribution $\mathbf{D}_{A,x}$, consider a particular partial execution path τ that drives A to a point where A invokes algorithm B with a particular input y (determined by x and τ). Consider the conditional probability distribution given that τ is a prefix of A's actual execution path. We can define a random variable X on this conditional distribution whose value is the subpath traced out by the invocation of subroutine B. Show that the distribution of X is the same as $\mathbf{D}_{B,y}$. Hint: use Exercise 7.1.

The above exercise is also very useful in simplifying the analysis of probabilistic algorithms, in that it allows us to analyze a subroutine in isolation, and use the results in the analysis of an algorithm that calls that subroutine.

Exercise 7.4. Let A be a probabilistic algorithm, and for an input x and integer $k \geq 0$, consider the experiment in which we choose a random execution path of length k, and run A on input x for up to k steps using the selected execution path. If A halts within k steps, we define $A_k(x)$ to be the output produced by A, and $T_{A_k}(x)$ to be the actual number of steps executed by A; otherwise, we define $A_k(x)$ to be the distinguished value "$\perp$" and $T_{A_k}(x)$ to be k.

(a) Show that if A halts with probability 1 on input x, then for all possible outputs y,

$$\mathsf{P}[A(x) = y] = \lim_{k \to \infty} \mathsf{P}[A_k(x) = y].$$

(b) Show that if A halts with probability 1 on input x, then

$$\mathsf{E}[T_A(x)] = \lim_{k \to \infty} \mathsf{E}[T_{A_k}(x)].$$

Exercise 7.5. One can generalize the notion of a discrete, probabilistic process, as follows. Let Γ be a finite or countably infinite set. Let f be a

function mapping sequences of one or more elements of Γ to $[0,1]$, such that the following property holds:

> for all finite sequences $(\gamma_1, \ldots, \gamma_{i-1})$, where $i \geq 1$, $f(\gamma_1, \ldots, \gamma_{i-1}, \gamma)$ is non-zero for at most a finite number of $\gamma \in \Gamma$, and
>
> $$\sum_{\gamma \in \Gamma} f(\gamma_1, \ldots, \gamma_{i-1}, \gamma) = 1.$$

Now consider any prefix-free set $\mathcal{S}$ of finite sequences of elements of Γ. For $\sigma = (\gamma_1, \ldots, \gamma_n) \in \mathcal{S}$, define

$$\mathsf{P}[\sigma] := \prod_{i=1}^{n} f(\gamma_1, \ldots, \gamma_i).$$

Show that $\sum_{\sigma \in \mathcal{S}} \mathsf{P}[\sigma] \leq 1$, and hence we may define a probability distribution on $\mathcal{S}$ using the probability function $\mathsf{P}[\cdot]$ if this sum is 1. The intuition is that we are modeling a process in which we start out in the "empty" configuration; at each step, if we are in configuration $(\gamma_1, \ldots, \gamma_{i-1})$, we halt if this is a "halting" configuration, that is, an element of $\mathcal{S}$, and otherwise, we move to configuration $(\gamma_1, \ldots, \gamma_{i-1}, \gamma)$ with probability $f(\gamma_1, \ldots, \gamma_{i-1}, \gamma)$.

7.2 Approximation of functions

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 $0 \leq \epsilon < 1/2$, such that for all inputs x, $\mathsf{P}[A(x) = f(x)] \geq 1 - \epsilon$. The value ϵ is a bound on the **error probability**, which is defined as $\mathsf{P}[A(x) \neq f(x)]$.

7.2.1 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 t, times, and take the majority output as the answer. Using the Chernoff bound (Theorem 6.13), the error probability for the iterated version of A is bounded by $\exp[-(1/2-\epsilon)^2 t/2]$, and so the error probability decreases exponentially with the number of iterations. This bound is derived as follows. For $i = 1, \ldots, t$, let X_i be a random variable representing the outcome of the ith iteration of A; more precisely, $X_i = 1$ if $A(x) \neq f(x)$ on the ith iteration, and $X_i = 0$ otherwise. Let ϵ_x be the probability that $A(x) \neq f(x)$. The probability that the majority output is wrong is equal to the probability that the sample

mean of $X_1, \ldots, X_t$ exceeds the mean ϵ_x by at least $1/2 - \epsilon_x$. Part (i) of Theorem 6.13 says that this occurs with probability at most

$$\exp\left[\frac{-(1/2-\epsilon_x)^2 t}{2(1-\epsilon_x)}\right] \leq \exp\left[\frac{-(1/2-\epsilon)^2 t}{2}\right].$$

7.2.2 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 $\epsilon < 1/2$ is a bound on the error probability, and $T(n)$ is a polynomial bound on the expected running time for inputs of length n. Then A' simply runs A for at most $tT(n)$ steps, where t is any constant chosen so that $\epsilon + 1/t < 1/2$—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 $tT(n)$ steps. By Markov's inequality (Theorem 6.11), the latter probability is at most $1/t$, and hence A' approximates f as well, but with an error probability bounded by $\epsilon + 1/t$.

7.2.3 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 $\epsilon < 1/2$.
- We call a probabilistic, expected polynomial-time algorithm A a **Monte Carlo algorithm** for recognizing L if for some constant $\delta > 0$, we have:
  - for any $x \in L$, we have $\mathsf{P}[A(x) = 1] \geq \delta$, and
  - for any $x \notin L$, we have $\mathsf{P}[A(x) = 1] = 0$.

- 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 7.6. Show that any language recognized by a Las Vegas algorithm is also recognized by a Monte Carlo algorithm, and that any language recognized by a Monte Carlo algorithm is also recognized by an Atlantic City algorithm.

EXERCISE 7.7. 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 length n.

EXERCISE 7.8. 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 length n.

EXERCISE 7.9. Show that a language is recognized by a Las Vegas algorithm iff the language and its complement are recognized by Monte Carlo algorithms.

EXERCISE 7.10. 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 7.11. 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 length n, we have

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

(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.

7.3 Flipping a coin until a head appears

In this and subsequent sections of this chapter, we discuss a number of specific probabilistic algorithms.

Let us begin with the following simple algorithm (which was already presented in Example 7.1) that essentially flips a coin until a head appears:

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

Let X be a random variable that represents the number of loop iterations made by the algorithm. It should be fairly clear that X has a geometric distribution, where the associated probability of success is $1/2$ (see Example 6.30). However, let us derive this fact from more basic principles. Define random variables $B_1, B_2, \ldots$, where B_i represents the value of the bit assigned to b in the ith loop iteration, if $X \geq i$, and $\star$ otherwise. Clearly, exactly one B_i will take the value 1, in which case X takes the value i.

Evidently, for each $i \geq 1$, if the algorithm actually enters the ith loop iteration, then B_i is uniformly distributed over $\{0,1\}$, and otherwise, $B_i = \star$. That is:

$$\mathsf{P}[B_i = 0 \mid X \geq i] = 1/2, \quad \mathsf{P}[B_i = 1 \mid X \geq i] = 1/2,$$
$$\mathsf{P}[B_i = \star \mid X < i] = 1.$$

From this, we see that

$$\mathsf{P}[X \geq 1] = 1, \quad \mathsf{P}[X \geq 2] = \mathsf{P}[B_1 = 0 \mid X \geq 1]\mathsf{P}[X \geq 1] = 1/2,$$
$$\mathsf{P}[X \geq 3] = \mathsf{P}[B_2 = 0 \mid X \geq 2]\mathsf{P}[X \geq 2] = (1/2)(1/2) = 1/4,$$

and by induction on i, we see that

$$\mathsf{P}[X \geq i] = \mathsf{P}[B_{i-1} = 0 \mid X \geq i-1]\mathsf{P}[X \geq i-1] = (1/2)(1/2^{i-2}) = 1/2^{i-1},$$

from which it follows (see Exercise 6.54) that X has a geometric distribution with associated success probability $1/2$.

Now consider the expected value $\mathsf{E}[X]$. By the discussion in Example 6.35, we have $\mathsf{E}[X] = 2$. If Y denotes the total running time of the algorithm, then $Y \leq cX$ for some constant c, and hence

$$\mathsf{E}[Y] \leq c\mathsf{E}[X] = 2c,$$

and we conclude that the expected running time of the algorithm is a constant, the exact value of which depends on the details of the implementation.

[Readers who skipped §7.1.1 may also want to skip this paragraph.] As was argued in Example 7.1, the above algorithm halts with probability 1. To make the above argument completely rigorous, we should formally justify that claim that the conditional distribution of B_i, given that $X \geq i$, is uniform over $\{0,1\}$. We do not wish to assume that the values of the B_i are located at pre-determined positions of the execution path; rather, we shall employ a more generally applicable technique. For any $i \geq 1$, we shall condition on a particular partial execution path τ that drives the algorithm to the point where it is just about to sample the bit B_i, and show that in this conditional probability distribution, B_i is uniformly distributed over $\{0,1\}$. To do this rigorously in our formal framework, let us define the event $\mathcal{A}_\tau$ to be the event that τ is a prefix of the execution path. If $|\tau| = \ell$, then the events $\mathcal{A}_\tau$, $\mathcal{A}_\tau \wedge (B_i = 0)$, and $\mathcal{A}_\tau \wedge (B_i = 1)$ are determined by the first $\ell+1$ bits of the execution path. We can then consider corresponding events in a probabilistic experiment wherein we observe the behavior of the algorithm on a random $(\ell+1)$-bit execution path (see Exercise 7.2). In the latter experiment, it is clear that the conditional probability distribution of B_i, given that the first ℓ bits of the actual execution path σ agree with τ, is uniform over $\{0,1\}$, and thus, the same holds in the original probability distribution. Since this holds for all relevant τ, it follows (by a discrete version of Exercise 6.13) that it holds conditioned on $X \geq i$.

We have analyzed the above algorithm in excruciating detail. As we proceed, many of these details will be suppressed, as they can all be handled by very similar (and completely routine) arguments.

7.4 Generating a random number from a given interval

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

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

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

In the general case, we do not have a direct way to do this, since we can only directly generate random bits. However, suppose that M is a k-bit number, so that $2^{k-1} \leq M < 2^k$. Then the following algorithm does the job:

Algorithm RN:

repeat
    $s \leftarrow_R \{0,1\}^{\times k}$
    $n \leftarrow I(s)$
until $n < M$
output n

Let X denote the number of loop iterations of this algorithm, Y its running time, and N its output.

In every loop iteration, n is uniformly distributed over $\{0, \ldots, 2^k - 1\}$, and the event $n < M$ occurs with probability $M/2^k$; moreover, conditioning on the latter event, n is uniformly distributed over $\{0, \ldots, M-1\}$. It follows that X has a geometric distribution with an associated success probability $p := M/2^k \geq 1/2$, and that N is uniformly distributed over $\{0, \ldots, M-1\}$. We have $\mathsf{E}[X] = 1/p \leq 2$ (see Example 6.35) and $Y \leq ckX$ for some implementation-dependent constant c, from which it follows that

$$\mathsf{E}[Y] \leq ck\mathsf{E}[X] \leq 2ck.$$

Thus, the expected running time of Algorithm RN is $O(k)$.

Hopefully, the above argument is clear and convincing. However, as in the previous section, we can derive these results from more basic principles. Define random variables $N_1, N_2, \ldots$, where N_i represents the value of n in the ith loop iteration, if $X \geq i$, and $\star$ otherwise.

Evidently, for each $i \geq 1$, if the algorithm actually enters the ith loop iteration, then N_i is uniformly distributed over $\{0, \ldots, 2^k - 1\}$, and otherwise, $N_i = \star$. That is:

$$\mathsf{P}[N_i = j \mid X \geq i] = 1/2^k \quad (j = 0, \ldots, 2^k - 1),$$
$$\mathsf{P}[N_i = \star \mid X < i] = 1.$$

From this fact, we can derive all of the above results.

As for the distribution of X, it follows from a simple induction argument that $\mathsf{P}[X \geq i] = q^{i-1}$, where $q := 1 - p$; indeed, $\mathsf{P}[X \geq 1] = 1$, and for $i \geq 2$, we have

$$\mathsf{P}[X \geq i] = \mathsf{P}[N_{i-1} \geq M \mid X \geq i-1]\mathsf{P}[X \geq i-1] = q \cdot q^{i-2} = q^{i-1}.$$

It follows that X has a geometric distribution with associated success probability p (see Exercise 6.54).

As for the distribution of N, by (a discrete version of) Exercise 6.13, it suffices to show that for all $i \geq 1$, the conditional distribution of N given that

$X = i$ is uniform on $\{0, \ldots, M-1\}$. Observe that for any $j = 0, \ldots, M-1$, we have

$$\begin{aligned}\mathsf{P}[N = j \mid X = i] &= \frac{\mathsf{P}[N = j \wedge X = i]}{\mathsf{P}[X = i]} = \frac{\mathsf{P}[N_i = j \wedge X \geq i]}{\mathsf{P}[N_i < M \wedge X \geq i]} \\ &= \frac{\mathsf{P}[N_i = j \mid X \geq i]\mathsf{P}[X \geq i]}{\mathsf{P}[N_i < M \mid X \geq i]\mathsf{P}[X \geq i]} = \frac{1/2^k}{M/2^k} \\ &= 1/M.\end{aligned}$$

[Readers who skipped §7.1.1 may also want to skip this paragraph.] To make the above argument completely rigorous, we should first show that the algorithm halts with probability 1, and then show that the conditional distribution of N_i, given that $X \geq i$, is indeed uniform on $\{0, \ldots, 2^k - 1\}$, as claimed above. That the algorithm halts with probability 1 follows from the fact that in every loop iteration, there is at least one choice of s that will cause the algorithm to halt. To analyze the conditional distribution on N_i, one considers various conditional distributions, conditioning on particular partial execution paths τ that bring the computation just to the beginning of the ith loop iteration; for any particular such τ, the ith loop iteration will terminate in at most $\ell := |\tau| + ck$ steps, for some constant c. Therefore, the conditional distribution of N_i, given the partial execution path τ, can be analyzed by considering the execution of the algorithm on a random ℓ-bit execution path (see Exercise 7.2). It is then clear that the conditional distribution of N_i given the partial execution path τ is uniform over $\{0, \ldots, 2^k - 1\}$, and since this holds for all relevant τ, it follows (by a discrete version of Exercise 6.13) that the conditional distribution of N_i, given that the ith loop is entered, is uniform over $\{0, \ldots, 2^k - 1\}$.

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

$$n \leftarrow_R \{A, \ldots, B\}.$$

We also mention the following alternative approach to generating a random number from an interval. Given a positive k-bit integer M, and a parameter $t > 0$, we do the following:

Algorithm RN′:

$s \leftarrow_R \{0,1\}^{\times(k+t)}$
$n \leftarrow I(s) \bmod M$
output n

Compared with Algorithm RN, Algorithm RN′ has the advantage that

there are no loops — it 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^{-t} (see Example 6.27 in §6.8). Thus, by choosing t suitably large, we can make the output distribution "as good as uniform" for most practical purposes.

Exercise 7.12. Prove that no probabilistic algorithm that always halts in a bounded number of steps can have an output distribution that is uniform on $\{0, \ldots, M-1\}$, unless M is a power of 2.

Exercise 7.13. Let A_1 and A_2 be probabilistic algorithms such that, for any input x, the random variables $A_1(x)$ and $A_2(x)$ take on one of a finite number of values, and let δ_x be the statistical distance between $A_1(x)$ and $A_2(x)$. Let B be any probabilistic algorithm that always outputs 0 or 1. For for $i = 1, 2$, let C_i be the algorithm that given an input x, first runs A_i on that input, obtaining a value y, then it runs B on input y, obtaining a value z, which it then outputs. Show that $|\mathsf{P}[C_1(x) = 1] - \mathsf{P}[C_2(x) = 1]| \leq \delta_x$.

7.5 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 algorithm *IsPrime* that determines whether or not a given integer $n > 1$ is prime.

Now, the most naive method of testing if n is prime is to see if any of the numbers between 2 and $n-1$ divide n. Of course, one can be slightly more clever, and only perform this divisibility check for prime numbers between 2 and $\sqrt{n}$ (see Exercise 1.1). Nevertheless, such an approach does not give rise to a polynomial-time algorithm. Indeed, the design and analysis of efficient primality tests has been an active research area for many years. There is, in fact, a deterministic, polynomial-time algorithm for testing primality, which we shall discuss later, in Chapter 22. For the moment, we shall just assume we have such an algorithm, and use it as a "black box."

Our algorithm to generate a random prime between 2 and M runs as follows:

Algorithm RP:

repeat
    $n \leftarrow_R \{2, \ldots, M\}$
until *IsPrime*(n)
output n

We now wish to analyze the running time and output distribution of Algorithm RP on input M. Let $k := \mathrm{len}(M)$.

First, consider a single iteration of the main loop of Algorithm RP, viewed as a stand-alone probabilistic experiment. For any fixed prime p between 2 and M, the probability that the variable n takes the value p is precisely $1/(M-1)$. Thus, every prime is equally likely, and the probability that n is a prime is precisely $\pi(M)/(M-1)$.

Let us also consider the expected running time μ of a single loop iteration. To this end, define W_n to be the running time of algorithm *IsPrime* on input n. Also, define

$$W'_M := \frac{1}{M-1} \sum_{n=2}^{M} W_n.$$

That is, W'_M is the average value of W_n, for a random choice of $n \in \{2, \ldots, M\}$. Thus, μ is equal to W'_M, plus the expected running time of Algorithm RN, which is $O(k)$, plus any other small overhead, which is also $O(k)$. So we have $\mu \leq W'_M + O(k)$, and assuming that $W'_M = \Omega(k)$, which is perfectly reasonable, we have $\mu = O(W'_M)$.

Next, let us consider the behavior of Algorithm RP as a whole. From the above discussion, it follows that when this algorithm terminates, its output will be uniformly distributed over the set of all primes between 2 and M. If T denotes the number of loop iterations performed by the algorithm, then $\mathsf{E}[T] = (M-1)/\pi(M)$, which by Chebyshev's theorem (Theorem 5.1) is $\Theta(k)$.

So we have bounded the expected number of loop iterations. We now want to bound the expected overall running time. For $i \geq 1$, let X_i denote the amount of time (possibly zero) spent during the ith loop iteration of the algorithm, so that $X := \sum_{i \geq 1} X_i$ is the total running time of Algorithm RP. Note that

$$\begin{aligned} \mathsf{E}[X_i] &= \mathsf{E}[X_i \mid T \geq i]\mathsf{P}[T \geq i] + \mathsf{E}[X_i \mid T < i]\mathsf{P}[T < i] \\ &= \mathsf{E}[X_i \mid T \geq i]\mathsf{P}[T \geq i] \\ &= \mu \mathsf{P}[T \geq i], \end{aligned}$$

because $X_i = 0$ when $T < i$ and $\mathsf{E}[X_i \mid T \geq i]$ is by definition equal to μ. Then we have

$$\mathsf{E}[X] = \sum_{i \geq 1} \mathsf{E}[X_i] = \mu \sum_{i \geq 1} \mathsf{P}[T \geq i] = \mu \mathsf{E}[T] = O(kW'_M).$$

7.5.1 Using a probabilistic primality test

In the above analysis, we assumed that *IsPrime* was a deterministic, polynomial-time algorithm. While such an algorithm exists, there are in fact simpler and more efficient algorithms that are probabilistic. We shall discuss such an algorithm in greater depth later, in Chapter 10. This algorithm (like several other algorithms for primality testing) 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 c, where $c < 1$ is a constant. In the terminology of §7.2, such an algorithm is essentially a Monte Carlo algorithm for the language of *composite* numbers. If we want to reduce the error probability for composite inputs to some very small value ϵ, we can iterate the algorithm t times, with t chosen so that $c^t \leq \epsilon$, outputting *true* if all iterations output *true*, and outputting *false* otherwise. This yields an algorithm for primality testing that makes errors only on composite inputs, and then only with probability at most ϵ.

Let us analyze the behavior of Algorithm RP under the assumption that *IsPrime* is implemented by a probabilistic algorithm (such as described in the previous paragraph) with an error probability for composite inputs bounded by ϵ. Let us define W_n to be the expected running time of *IsPrime* on input n, and as before, we define

$$W'_M := \frac{1}{M-1} \sum_{n=2}^{M} W_n.$$

Thus, W'_M is the expected running time of algorithm *IsPrime*, where the average is taken with respect to randomly chosen n *and* the random choices of the algorithm itself.

Consider a single loop iteration of Algorithm RP. For any fixed prime p between 2 and M, the probability that n takes the value p is $1/(M-1)$. Thus, if the algorithm halts with a prime, every prime is equally likely. Now, the algorithm will halt if n is prime, or if n is composite and the primality test makes a mistake; therefore, the the probability that it halts at all is at least $\pi(M)/(M-1)$. So we see that the expected number of loop iterations

should be no more than in the case where we use a deterministic primality test. Using the same argument as was used before to estimate the expected total running time of Algorithm RP, we find that this is $O(kW'_M)$.

As for the probability that Algorithm RP mistakenly outputs a composite, one might be tempted to say that this probability is at most ϵ, the probability that *IsPrime* makes a mistake. However, in drawing such a conclusion, we would be committing the fallacy of Example 6.12—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.

Let us be a bit more precise. Again, consider the probability distribution defined by a single loop iteration, and let $\mathcal{A}$ be the event that *IsPrime* outputs *true*, and $\mathcal{B}$ the event that n is composite. Let $\beta := \mathsf{P}[\mathcal{B}]$ and $\alpha := \mathsf{P}[\mathcal{A} \mid \mathcal{B}]$. First, observe that, by definition, $\alpha \leq \epsilon$. Now, the probability δ that the algorithm halts and outputs a composite in this loop iteration is

$$\delta = \mathsf{P}[\mathcal{A} \wedge \mathcal{B}] = \alpha\beta.$$

The probability δ' that the algorithm halts and outputs either a prime or composite is

$$\delta' = \mathsf{P}[\mathcal{A}] = \mathsf{P}[\mathcal{A} \wedge \mathcal{B}] + \mathsf{P}[\mathcal{A} \wedge \overline{\mathcal{B}}] = \mathsf{P}[\mathcal{A} \wedge \mathcal{B}] + \mathsf{P}[\overline{\mathcal{B}}] = \alpha\beta + (1 - \beta).$$

Now consider the behavior of Algorithm RP as a whole. With T being the number of loop iterations as before, we have

$$\mathsf{E}[T] = \frac{1}{\delta'} = \frac{1}{\alpha\beta + (1 - \beta)}, \tag{7.1}$$

and hence

$$\mathsf{E}[T] \leq \frac{1}{(1 - \beta)} = \frac{M - 1}{\pi(M)} = O(k).$$

Let us now consider the probability γ that the output of Algorithm RP is composite. For $i \geq 1$, let $\mathcal{C}_i$ be the event that the algorithm halts and outputs a composite number in the ith loop iteration. The events $\mathcal{C}_i$ are pairwise disjoint, and moreover,

$$\mathsf{P}[\mathcal{C}_i] = \mathsf{P}[\mathcal{C}_i \wedge T \geq i] = \mathsf{P}[\mathcal{C}_i \mid T \geq i]\mathsf{P}[T \geq i] = \delta\mathsf{P}[T \geq i].$$

So we have

$$\gamma = \sum_{i \geq 1} \mathsf{P}[\mathcal{C}_i] = \sum_{i \geq 1} \delta\mathsf{P}[T \geq i] = \delta\mathsf{E}[T] = \frac{\alpha\beta}{\alpha\beta + (1 - \beta)}, \tag{7.2}$$

and hence

$$\gamma \leq \frac{\alpha}{(1-\beta)} \leq \frac{\epsilon}{(1-\beta)} = \epsilon \frac{M-1}{\pi(M)} = O(k\epsilon).$$

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 p is output with probability at most $1/\pi(M)$. It follows from Theorem 6.15 that $\Delta = \gamma$.

7.5.2 Generating a random k-bit prime

Instead of generating a random prime between 2 and M, we may instead want to generate a random k-bit prime, that is, a prime between 2^{k-1} and $2^k - 1$. Bertrand's postulate (Theorem 5.7) tells us that there exist such primes for every $k \geq 2$, and that in fact, there are $\Omega(2^k/k)$ such primes. Because of this, we can modify Algorithm RP, so that each candidate n is chosen at random from the interval $\{2^{k-1}, \ldots, 2^k - 1\}$, and all of the results of this section carry over essentially without change. In particular, the expected number of trials until the algorithm halts is $O(k)$, and if a probabilistic primality test as in §7.5.1 is used, with an error probability of ϵ, the probability that the output is not prime is $O(k\epsilon)$.

Exercise 7.14. Design and analyze an efficient probabilistic algorithm that takes as input an integer $M \geq 2$, and outputs a random element of $\mathbb{Z}_M^*$.

Exercise 7.15. 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 §7.4). Assume that $2\delta < \pi(M)/(M-1)$. Also, let λ 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 $k := \mathrm{len}(M)$.

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

(b) Assuming the primality test is probabilistic, with one-sided error ϵ, as in §7.5.1, show that $\lambda = O(k)$ and $\Delta = O((\delta + \epsilon)k)$.

EXERCISE 7.16. Analyze Algorithm RP assuming that the primality test is implemented by an "Atlantic City" algorithm with error probability at most ϵ.

EXERCISE 7.17. Consider the following probabilistic algorithm that takes as input a positive integer M:

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

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

The following exercises assume the reader has studied §7.1.1.

EXERCISE 7.18. Consider the following algorithm (which takes no input):

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

Show that this algorithm halts with probability 1, but that its expected running time does not exist. (Compare this algorithm with the one in Example 7.2, which does not even halt with probability 1.)

EXERCISE 7.19. Now consider the following modification to the algorithm in the previous exercise:

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

Show that this algorithm halts with probability 1, and that its expected running time exists (and is equal to some implementation-dependent constant).

7.6 Generating a random non-increasing sequence

The following algorithm, Algorithm RS, 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 takes as input an integer $M \geq 2$, and runs as follows:

Algorithm RS:

$n_0 \leftarrow M$
$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    $n_i \leftarrow_R \{1, \ldots, n_{i-1}\}$
until $n_i = 1$
$t \leftarrow i$
Output $(n_1, \ldots, n_t)$

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

7.6.1 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 chain $(n_1, \ldots, n_t)$, where $n_1 \geq n_2 \geq \cdots \geq n_{t-1} > n_t = 1$. For any such chain, we have

$$\begin{aligned} \mathsf{P}[N_1 = n_1 \wedge \cdots \wedge N_t = n_t] &= \mathsf{P}[N_1 = n_1]\mathsf{P}[N_2 = n_2 \mid N_1 = n_1] \cdots \\ &\quad \mathsf{P}[N_t = n_t \mid N_1 = n_1 \wedge \cdots \wedge N_{t-1} = n_{t-1}] \\ &= \frac{1}{M} \cdot \frac{1}{n_1} \cdot \cdots \cdot \frac{1}{n_{t-1}}. \end{aligned} \tag{7.3}$$

This completely describes the output distribution, in the sense that we have determined the probability with which each non-increasing chain 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 E_j to be the number of occurrences of j among the N_i. The E_j determine the N_i, and *vice versa*. Indeed, $E_M = e_M, \ldots, E_2 = e_2$ iff the output of the algorithm is the non-increasing chain

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

From (7.3), we can therefore directly compute

$$\mathsf{P}[E_M = e_M \wedge \ldots \wedge E_2 = e_2] = \frac{1}{M} \prod_{j=2}^{M} \frac{1}{j^{e_j}}. \tag{7.4}$$

Notice that we can write $1/M$ as a telescoping product:

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

so we can re-write (7.4) as

$$\mathsf{P}[E_M = e_M \wedge \cdots \wedge E_2 = e_2] = \prod_{j=2}^{M} j^{-e_j}(1 - 1/j). \tag{7.5}$$

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 6.1, the variables E_j are mutually independent, and for all $j = 2, \ldots, M$ and integers $e_j \geq 0$, we have

$$\mathsf{P}[E_j = e_j] = j^{-e_j}(1 - 1/j). \tag{7.6}$$

In summary, we have shown that the variables E_j are mutually independent, where for $j = 2, \ldots, M$, the variable $E_j + 1$ has a geometric distribution with an associated success probability of $1 - 1/j$.

Another, perhaps more intuitive, analysis of the joint distribution of the E_j runs as follows. Conditioning on the event $E_M = e_M, \ldots, E_{j+1} = e_{j+1}$, one sees that the value of E_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 $E_j = e_j$ 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 $j^{-e_j}(1 - 1/j)$.

7.6.2 Analysis of the running time

Let T be the random variable that takes the value t when the output is $(n_1, \ldots, n_t)$. Clearly, it is the value of T that essentially determines the running time of the algorithm.

With the random variables E_j defined as above, we see that $T = 1 + \sum_{j=2}^{M} E_j$. Moreover, for each j, $E_j + 1$ has a geometric distribution with

associated success probability $1 - 1/j$, and hence

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

Thus,

$$\mathsf{E}[T] = 1 + \sum_{j=2}^{M} \mathsf{E}[E_j] = 1 + \sum_{j=1}^{M-1} \frac{1}{j} = \int_1^M \frac{dy}{y} + O(1) \sim \log M.$$

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 $O(\mathrm{len}(M))$ steps, we expect to reach 1.

To complete the running time analysis, let us consider the total number of times X that the main loop of Algorithm RN in §7.4 is executed. For $i = 1, 2, \ldots$, let X_i denote the number of times that loop is executed in the ith loop of Algorithm RS, defining this to be zero if the ith loop is never reached. So $X = \sum_{i=1}^{\infty} X_i$. Arguing just as in §7.5, we have

$$\mathsf{E}[X] = \sum_{i \geq 1} \mathsf{E}[X_i] \leq 2 \sum_{i \geq 1} \mathsf{P}[T \geq i] = 2\mathsf{E}[T] \sim 2 \log M.$$

To finish, if Y denotes the running time of Algorithm RS on input M, then we have $Y \leq c\,\mathrm{len}(M)(X + 1)$ for some constant c, and hence $\mathsf{E}[Y] = O(\mathrm{len}(M)^2)$.

Exercise 7.20. Show that when Algorithm RS runs on input M, the expected number of (not necessarily distinct) primes in the output sequence is $\sim \log \log M$.

Exercise 7.21. For $j = 2, \ldots, M$, let $F_j := 1$ if j appears in the output of Algorithm RS on input M, and let $F_j := 0$ otherwise. Determine the joint distribution of the F_j. Using this, show that the expected number of distinct primes appearing in the output sequence is $\sim \log \log M$.

7.7 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 r uniformly distributed over the interval $\{1, \ldots, M\}$, but instead of the usual output format for such a number r, the output consists of the prime factorization of r.

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 §7.6 as a subroutine. In addition, as we did in §7.5, we shall assume the existence of a deterministic, polynomial-time primality test *IsPrime*. We denote its running time on input n by W_n, and set $W_M^* := \max\{W_n : n = 2, \ldots, M\}$.

In the analysis of the algorithm, we shall make use of Mertens' theorem, which we proved in Chapter 5 (Theorem 5.13).

On input $M \geq 2$, the algorithm to generate a random factored number $r \in \{1, \ldots, M\}$ runs as follows:

Algorithm RFN:

repeat
  Run Algorithm RS on input M, obtaining $(n_1, \ldots, n_t)$
(∗) Let $n_{i_1}, \ldots, n_{i_\ell}$ be the primes among $n_1, \ldots, n_t$, including duplicates
(∗∗) Set $r \leftarrow \prod_{j=1}^{\ell} n_{i_j}$
  If $r \leq M$ then
    $s \leftarrow_R \{1, \ldots, M\}$
    if $s \leq r$ then output $n_{i_1}, \ldots, n_{i_\ell}$ and halt
forever

Notes:

(∗) For $i = 1, \ldots, t-1$, the number n_i is tested for primality algorithm *IsPrime*.

(∗∗) 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.

Let us now analyze the running time and output distribution of Algorithm RFN on input M. Let $k := \mathrm{len}(M)$.

To analyze this algorithm, let us first consider a single iteration of the main loop as a random experiment in isolation. Let $n = 1, \ldots, M$ be a fixed integer, and let us calculate the probability that the variable r takes the particular value n in this loop iteration. Let $n = \prod_{p \leq M} p^{e_p}$ be the prime factorization of n. Then r takes the value n iff $E_p = e_p$ for all primes $p \leq M$,

which by the analysis in §7.6, happens with probability precisely

$$\prod_{p \leq M} p^{-e_p}(1 - 1/p) = \frac{U(M)}{n},$$

where

$$U(M) := \prod_{p \leq M} (1 - 1/p).$$

Now, the probability that this loop iteration produces n as output is equal to the probability that r takes the value n *and* $s \leq n$, which is

$$\frac{U(M)}{n} \cdot \frac{n}{M} = \frac{U(M)}{M}.$$

Thus, every n is equally likely, and summing over all $n = 1, \ldots, M$, we see that the probability that this loop iteration succeeds in producing *some* output is $U(M)$.

Now consider the expected running time of this loop iteration. From the analysis in §7.6, it is easy to see that this is $O(kW^*_M)$. That completes the analysis of a single loop iteration.

Finally, consider the behavior of Algorithm RFN as a whole. From our analysis of an individual loop iteration, it is clear that the output distribution of Algorithm RFN is as required, and if H denotes the number of loop iterations of the algorithm, then $\mathsf{E}[H] = U(M)^{-1}$, which by Mertens' theorem is $O(k)$. Since the expected running time of each individual loop iteration is $O(kW^*_M)$, it follows that the expected total running time is $O(k^2 W^*_M)$.

7.7.1 Using a probabilistic primality test (∗)

Analogous to the discussion in §7.5.1, 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 bounded by ϵ. Here, we assume that W_n denotes the expected running time of the primality test on input n, and set $W^*_M := \max\{W_n : n = 2, \ldots, M\}$.

The situation here is a bit more complicated than in the case of Algorithm RP, since an erroneous output of the primality test in Algorithm RFN could lead either to the algorithm halting prematurely (with a wrong output), or to the algorithm being delayed (because an opportunity to halt may be missed).

Let us first analyze in detail the behavior of a single iteration of the main

loop of Algorithm RFN. Let $\mathcal{A}$ denote the event that the primality test makes a mistake in this loop iteration, and let $\delta := \mathsf{P}[\mathcal{A}]$. If T is the number of loop iterations in a given run of Algorithm RS, it is easy to see that

$$\delta \leq \epsilon\, \mathsf{E}[T] = \epsilon\, \ell(M),$$

where

$$\ell(M) := 1 + \sum_{j=1}^{M-1} \frac{1}{j} \leq 2 + \log M.$$

Now, let $n = 1, \ldots, M$ be a fixed integer, and let us calculate the probability α_n that the correct prime factorization of n is output in this loop iteration. Let $\mathcal{B}_n$ be the event that the primes among the output of Algorithm RS multiply out to n. Then $\alpha_n = \mathsf{P}[\mathcal{B}_n \wedge \overline{\mathcal{A}}](n/M)$. Moreover, because of the mutual independence of the E_j, not only does it follow that $\mathsf{P}[\mathcal{B}_n] = U(M)/n$, but it also follows that $\mathcal{B}_n$ and $\mathcal{A}$ are independent events: to see this, note that $\mathcal{B}_n$ is determined by the variables $\{E_j : j \text{ prime}\}$, and $\mathcal{A}$ is determined by the variables $\{E_j : j \text{ composite}\}$ and the random choices of the primality test. Hence,

$$\alpha_n = \frac{U(M)}{M}(1 - \delta).$$

Thus, every n is equally likely to be output. If $\mathcal{C}$ is the event that the algorithm halts with *some* output (correct or not) in this loop iteration, then

$$\mathsf{P}[\mathcal{C}] \geq U(M)(1 - \delta), \tag{7.7}$$

and

$$\mathsf{P}[\mathcal{C} \vee \mathcal{A}] = U(M)(1 - \delta) + \delta = U(M) - \delta U(M) + \delta \geq U(M). \tag{7.8}$$

The expected running time of a single loop iteration of Algorithm RFN is also easily seen to be $O(kW_M^*)$. That completes the analysis of a single loop iteration.

We next analyze the total running time of Algorithm RFN. If H is the number of loop iterations of Algorithm RFN, it follows from (7.7) that

$$\mathsf{E}[H] \leq \frac{1}{U(M)(1 - \delta)},$$

and assuming that $\epsilon\ell(M) \leq 1/2$, it follows that the expected running time of Algorithm RFN is $O(k^2 W_M^*)$.

Finally, we analyze the statistical distance Δ between the output distribution of Algorithm RFN and the uniform distribution on the numbers 1

to M, in correct factored form. Let H' denote the first loop iteration i for which the event $\mathcal{C} \vee \mathcal{A}$ occurs, meaning that the algorithm either halts or the primality test makes a mistake. Then, by (7.8), H' has a geometric distribution with an associated success probability of at least $U(M)$. Let $\mathcal{A}_i$ be the event that the primality makes a mistake *for the first time* in loop iteration i, and let $\mathcal{A}^*$ is the event that the primality test makes a mistake in any loop iteration. Observe that $\mathsf{P}[\mathcal{A}_i \mid H' \geq i] = \delta$ and $\mathsf{P}[\mathcal{A}_i \mid H' < i] = 0$, and so

$$\mathsf{P}[\mathcal{A}_i] = \mathsf{P}[\mathcal{A}_i \mid H' \geq i]\mathsf{P}[H' \geq i] = \delta \mathsf{P}[H' \geq i],$$

from which it follows that

$$\mathsf{P}[\mathcal{A}^*] = \sum_{i \geq 1} \mathsf{P}[\mathcal{A}_i] = \sum_{i \geq 1} \delta \mathsf{P}[H' \geq i] = \delta \mathsf{E}[H'] \leq \delta U(M)^{-1}.$$

Now, if γ is the probability that the output of Algorithm RFN is not in correct factored form, then

$$\gamma \leq \mathsf{P}[\mathcal{A}^*] = \delta U(M)^{-1} = O(k^2 \epsilon).$$

We have already argued that each value n between 1 and M, in correct factored form, is equally likely to be output, and in particular, each such value occurs with probability at most $1/M$. It follows from Theorem 6.15 that $\Delta = \gamma$ (verify).

Exercise 7.22. To simplify the analysis, we analyzed Algorithm RFN using the worst-case estimate W_M^* on the expected running time of the primality test. Define

$$W_M^+ := \sum_{j=2}^{M} \frac{W_j}{j-1},$$

where W_n denotes the expected running time of a probabilistic implementation of *IsPrime* on input n. Show that the expected running time of Algorithm RFN is $O(kW_M^+)$, assuming $\epsilon \ell(M) \leq 1/2$.

Exercise 7.23. Analyze Algorithm RFN assuming that the primality test is implemented by an "Atlantic City" algorithm with error probability at most ϵ.

7.8 The RSA cryptosystem

Algorithms for generating large primes, such as Algorithm RP in §7.5, have numerous applications in cryptography. One of the most well known and

important such applications is the RSA cryptosystem, named after its inventors Rivest, Shamir, and Adleman. We give a brief overview of this system here.

Suppose that Alice wants to send a secret message to Bob over an insecure network. An adversary may be able to eavesdrop on the network, and so sending the message "in the clear" is not an option. Using older, more traditional cryptographic techniques would require that Alice and Bob share a secret key between them; however, this creates the problem of securely generating such a shared secret. The RSA cryptosystem is an example of a "public key" cryptosystem. To use the system, Bob simply places a "public key" in the equivalent of an electronic telephone book, while keeping a corresponding "private key" secret. To send a secret message to Bob, Alice obtains Bob's public key from the telephone book, and uses this to encrypt her message. Upon receipt of the encrypted message, Bob uses his secret key to decrypt it, obtaining the original message.

Here is how the RSA cryptosystem works. To generate a public key/private key pair, Bob generates two very large random primes p and q. To be secure, p and q should be quite large—typically, they are chosen to be around 512 bits in length. We require that $p \neq q$, but the probability that two random 512-bit primes are equal is negligible, so this is hardly an issue. Next, Bob computes $n := pq$. Bob also selects an integer $e > 1$ such that $\gcd(e, \phi(n)) = 1$. Here, $\phi(n) = (p-1)(q-1)$. Finally, Bob computes $d := e^{-1} \bmod \phi(n)$. The public key is the pair (n, e), and the private key is the pair (n, d). The integer e is called the "encryption exponent" and d is called the "decryption exponent."

After Bob publishes his public key (n, e), Alice may send a secret message to Bob as follows. Suppose that a message is encoded in some canonical way as a number between 0 and $n-1$—we can always interpret a bit string of length less than $\mathrm{len}(n)$ as such a number. Thus, we may assume that a message is an element α of $\mathbb{Z}_n$. To encrypt the message α, Alice simply computes $\beta := \alpha^e$. The encrypted message is β. When Bob receives β, he computes $\gamma := \beta^d$, and interprets γ as a message. (Note that if Bob stores the factorization of n, then he may speed up the decryption process using the algorithm in Exercise 7.28 below.)

The most basic requirement of any encryption scheme is that decryption should "undo" encryption. In this case, this means that for all $\alpha \in \mathbb{Z}_n$, we should have

$$(\alpha^e)^d = \alpha. \tag{7.9}$$

If $\alpha \in \mathbb{Z}_n^*$, then this is clearly the case, since we have $ed = 1 + \phi(n)k$ for

some positive integer k, and hence by Euler's theorem (Theorem 2.15), we have

$$(\alpha^e)^d = \alpha^{ed} = \alpha^{1+\phi(n)k} = \alpha \cdot \alpha^{\phi(n)k} = \alpha.$$

Even if $\alpha \notin \mathbb{Z}_n^*$, equation (7.9) still holds. To see this, let $\alpha = [a]_n$, with $\gcd(a, n) \neq 1$. There are three possible cases. First, if $a \equiv 0 \pmod{n}$, then trivially, $a^{ed} \equiv 0 \pmod{n}$. Second, if $a \equiv 0 \pmod{p}$ but $a \not\equiv 0 \pmod{q}$, then trivially $a^{ed} \equiv 0 \pmod{p}$, and

$$a^{ed} \equiv a^{1+\phi(n)k} \equiv a \cdot a^{\phi(n)k} \equiv a \pmod{q},$$

where the last congruence follows from the fact that $\phi(n)k$ is a multiple of $q - 1$, which is a multiple of the multiplicative order of a modulo q (again by Euler's theorem). Thus, we have shown that $a^{ed} \equiv a \pmod{p}$ and $a^{ed} \equiv a \pmod{q}$, from which it follows that $a^{ed} \equiv a \pmod{n}$. The third case, where $a \not\equiv 0 \pmod{p}$ and $a \equiv 0 \pmod{q}$, is treated in the same way as the second. Thus, we have shown that equation (7.9) holds for all $\alpha \in \mathbb{Z}_n$.

Of course, the interesting question about the RSA cryptosystem is whether or not it really is secure. Now, if an adversary, given only the public key (n, e), were able to factor n, then he could easily compute the decryption exponent d. It is widely believed that factoring n is computationally infeasible, for sufficiently large n, and so this line of attack is ineffective, barring a breakthrough in factorization algorithms. However, there may be other possible lines of attack. For example, it is natural to ask whether one can compute the decryption exponent without having to go to the trouble of factoring n. It turns out that the answer to this question is no: if one could compute the decryption exponent d, then $ed - 1$ would be a multiple of $\phi(n)$, and as we shall see later in §10.6, given any multiple of $\phi(n)$, we can easily factor n.

Thus, computing the encryption exponent is equivalent to factoring n, and so this line of attack is also ineffective. But there still could be other lines of attack. For example, even if we assume that factoring large numbers is infeasible, this is not enough to guarantee that for a given encrypted message β, the adversary is unable to compute β^d (although nobody actually knows how to do this without first factoring n).

The reader should be warned that the proper notion of security for an encryption scheme is quite subtle, and a detailed discussion of this is well beyond the scope of this text. Indeed, the simple version of RSA presented here suffers from a number of security problems (because of this, actual implementations of public-key encryption schemes based on RSA are somewhat more complicated). We mention one such problem here (others are examined

in some of the exercises below). Suppose an eavesdropping adversary knows that Alice will send one of a few, known, candidate messages. For example, an adversary may know that Alice's message is either "let's meet today" or "let's meet tomorrow." In this case, the adversary can encrypt for himself all of the candidate messages, intercept Alice's actual encrypted message, and then by simply comparing encryptions, the adversary can determine which particular message Alice encrypted. This type of attack works simply because the encryption algorithm is deterministic, and in fact, any deterministic encryption algorithm will be vulnerable to this type of attack. To avoid this type of attack, one must use a *probabilistic* encryption algorithm. In the case of the RSA cryptosystem, this is often achieved by padding the message with some random bits before encrypting it.

Exercise 7.24. Alice submits a bid to an auction, and so that other bidders cannot see her bid, she encrypts it under the public key of the auction service. Suppose that the auction service provides a public key for an RSA encryption scheme, with a modulus n. Assume that bids are encoded simply as integers between 0 and $n - 1$ prior to encryption. Also, assume that Alice submits a bid that is a "round number," which in this case means that her bid is a number that is divisible by 10. Show how an eavesdropper can submit an encryption of a bid that exceeds Alice's bid by 10%, without even knowing what Alice's bid is. In particular, your attack should work even if the space of possible bids is very large.

Exercise 7.25. To speed up RSA encryption, one may choose a very small encryption exponent. This exercise develops a "small encryption exponent attack" on RSA. Suppose Bob, Bill, and Betty have RSA public keys with moduli n_1, n_2, and n_3, and all three use encryption exponent 3. Assume that n_1, n_2, n_3 are pairwise relatively prime. Suppose that Alice sends an encryption of the same message to Bob, Bill, and Betty — that is, Alice encodes her message as an integer a, with $0 \leq a < \min\{n_1, n_2, n_3\}$, and computes the three encrypted messages $\beta_i := [a^3]_{n_i}$, for $i = 1, \ldots, 3$. Show how to recover Alice's message from these three encrypted messages.

Exercise 7.26. To speed up RSA decryption, one might choose a small decryption exponent, and then derive the encryption exponent from this. This exercise develops a "small decryption exponent attack" on RSA. Suppose $n = pq$, where p and q are distinct primes with $\mathrm{len}(p) = \mathrm{len}(q)$. Let d and e be integers such that $1 < d < \phi(n)$, $1 < e < \phi(n)$, and $de \equiv 1 \pmod{\phi(n)}$.

Further, assume that

$$4d < n^{1/4}.$$

Show how to efficiently compute d, given n and e. Hint: since $de \equiv 1 \pmod{\phi(n)}$, it follows that $de = 1 + k\phi(n)$ for an integer k with $0 < k < d$; let $r := kn - de$, and show that $|r| < n^{3/4}$; next, show how to recover d (along with r and k) using Theorem 4.6.

Exercise 7.27. Suppose there is a probabilistic algorithm A that takes as input an integer n of the form $n = pq$, where p and q are distinct primes. The algorithm also takes as input an integer $e > 1$, with $\gcd(e, \phi(n)) = 1$, and an element $\beta \in \mathbb{Z}_n^*$. It outputs either "failure," or $\alpha \in \mathbb{Z}_n^*$ such that $\alpha^e = \beta$. Furthermore, assume that A runs in strict polynomial time, and that for all n and e of the above form, and for randomly chosen $\beta \in \mathbb{Z}_n^*$, A succeeds in finding α as above with probability $\epsilon(n, e)$. Here, the probability is taken over the random choice of β, as well as the random choices made during the execution of A. Show how to use A to construct another probabilistic algorithm A' that takes as input n and e as above, as well as $\beta \in \mathbb{Z}_n^*$, runs in expected polynomial time, and that satisfies the following property:

> if $\epsilon(n, e) \geq 0.001$, then *for all* $\beta \in \mathbb{Z}_n^*$, A' finds $\alpha \in \mathbb{Z}_n^*$ with $\alpha^e = \beta$ with probability at least 0.999.

The algorithm A' in the above exercise is an example of what is called a **random self-reduction**, that is, an algorithm that reduces the task of solving an arbitrary instance of a given problem to that of solving a random instance of the problem. Intuitively, the fact that a problem is random self-reducible in this sense means that the problem is no harder in "the worst case" than in "the average case."

Exercise 7.28. This exercise develops an algorithm for speeding up RSA decryption. Suppose that we are given two distinct ℓ-bit primes, p and q, an element $\beta \in \mathbb{Z}_n$, where $n := pq$, and an integer d, where $1 < d < \phi(n)$. Using the algorithm from Exercise 3.26, we can compute β^d at a cost of essentially 2ℓ squarings in $\mathbb{Z}_n$. Show how this can be improved, making use of the factorization of n, so that the total cost is essentially that of ℓ squarings in $\mathbb{Z}_p$ and ℓ squarings in $\mathbb{Z}_q$, leading to a roughly four-fold speed-up in the running time.

7.9 Notes

See Luby [59] for an exposition of the theory of pseudo-random bit generation.

Our approach in §7.1 to defining the probability distribution associated with the execution of a probabilistic algorithm is a bit unusual (indeed, it is a bit unusual among papers and textbooks on the subject to even bother to formally define much of anything). There are alternative approaches. One approach is to define the output distribution and expected running time of an algorithm on a given input directly, using the identities in Exercise 7.4, and avoid the construction of an underlying probability distribution. However, without such a probability distribution, we would have very few tools at our disposal to analyze the output distribution and running time of particular algorithms. Another approach (which we dismissed with little justification early on in §7.1) is to attempt to define a distribution that models an infinite random bit string. One way to do this is to identify an infinite bit string with the real number in the unit interval $[0, 1]$ obtained by interpreting the bit string as a number written in base 2, and then use continuous probability theory (which we have not developed here, but which is covered in a standard undergraduate course on probability theory), applied to the uniform distribution on $[0, 1]$. There are a couple of problems with this approach. First, the above identification of bit strings with numbers is not quite one-to-one. Second, when one tries to define the notion of expected running time, numerous technical problems arise; in particular, the usual definition of an expected value in terms of an integral would require us to integrate functions that are not Riemann integrable. To properly deal with all of these issues, one would have to develop a good deal of measure theory (σ-algebras, Lesbegue integration, and so on), at the level normally covered in a graduate-level course on probability or measure theory.

The algorithm presented here for generating a random factored number is due to Kalai [50], 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 $O(k)$ primality tests, as opposed to the $O(k^2)$ primality tests used by Kalai's algorithm.

The RSA cryptosystem was invented by Rivest, Shamir, and Adleman [78]. There is a vast literature on cryptography. One starting point is the book by Menezes, van Oorschot, and Vanstone [62]. The attack in Exercise 7.26 is due to Wiener [104]; this attack was recently strengthened by Boneh and Durfee [19].