Chapter 7

Randomized Computation

"Why should we fear, when chance rules everything, And foresight of the future there is none; 'Tis best to live at random, as one can."
Sophocles, *Oedipus Rex*

"We present here the motivation and a general description of a method dealing with a class of problems in mathematical physics. The method is, essentially, a statistical approach to the study of differential equations."
N. Metropolis and S. Ulam, "The Monte Carlo Method," 1949

"We do not assume anything about the distribution of the instances of the problem to be solved. Instead we incorporate randomization into the algorithm itself... It may seem at first surprising that employing randomization leads to efficient algorithms. This claim is substantiated by two examples. The first has to do with finding the nearest pair in a set of n points in $\mathbb{R}^k$. The second example is an extremely efficient algorithm for determining whether a number is prime."
Michael Rabin, 1976

So far, we used the Turing machine (as defined in Chapter 1) as our standard model of computation. But there is one aspect of reality this model does not seem to capture: the ability to make *random choices* during the computation. (Most programming languages provide a built-in *random number generator* for this.) While scientists and philosophers may still debate if true randomness exists in the world, it definitely seems that when tossing a coin (or measuring the results of other physical experiments) we get an outcome that is sufficiently random and unpredictable for all practical purposes. Thus it makes sense to consider algorithms (and Turing machines) that can toss a coin —in other words, use a source of random bits. A moment's reflection suggests that such algorithms have been implicitly studied for a long time. Think for instance of basic procedures in classical statistics such as an opinion poll —it tries to estimate facts about a large population by taking a small random sample of the population. Similarly, randomization is also a natural tool for simulating real-world systems that are themselves probabilistic, such as nuclear fission or the stock market. Statistical ideas have also been long used in study of differential equations; see the quote by Metropolis and Ulam above. They named such algorithms *Monte Carlo methods* after the famous European gambling resort.

In the last few decades randomization was also used to give simpler or more efficient algorithms for many problems— in areas ranging from number theory to network routing— that on the face of it have nothing to do with probability. We will see some examples in this chapter. We will not address the issue of the *quality* of random number generators in this chapter, defering that discussion to Chapters 9, 20 and 21.

As complexity theorists our main interest in this chapter is to understand the power of Turing machines that can toss random coins. We give a mathematical model for *probabilistic*

computation and define in Section 7.1 the complexity class **BPP** that aims to capture the set of decision problems efficiently solvable by probabilistic algorithms.[1] Section 7.2 gives a few examples of non-trivial probabilistic algorithms, demonstrating that randomization may give added power to the algorithm designer. In fact, since random number generators are ubiquitous (leaving aside for the moment the question of how good they are) the class **BPP** (and its sister classes **RP**, **coRP** and **ZPP**) is arguably as important as **P** in capturing the notion of "efficient computation." The examples above suggest that **P** is a proper subset of **BPP**, though somewhat surprisingly, there are reasons to believe that actually **BPP** may be the same as **P**; see Chapter 20.

Our definition of a probabilistic algorithm will allow it to output a wrong answer with some small probability. At first sight, the reader might be concerned that these errors could make such algorithms impractical. However, in Section 7.4 we show how to reduce the probability of error to a minuscule quantity.

This chapter also studies the relationship between **BPP** and classes studied in earlier chapters such as $\mathbf{P}_{/\mathbf{poly}}$ and **PH**.

Many of the notions studied in the previous chapters can be extended to the probabilistic setting. For example, in Sections 7.6 and 7.7 we will describe randomized reductions and probabilistic logspace algorithms. These are probabilistic analogs of reductions and logspace algorithms studied in Chapters 2 and 4.

The role of randomness in complexity theory extends far beyond a study of randomized algorithms and classes such as **BPP**. Entire areas such as cryptography (see Chapter 9) and interactive and probabilistically checkable proofs (see Chapters 8 and 11) rely on randomness in an essential way, sometimes to prove results whose statement seemingly did not involve randomness in any way. Thus this chapter lays the groundwork for many later chapters of the book.

Throughout this chapter and the rest of the book, we will use some notions from elementary probability on finite sample spaces; see Appendix A for a quick review.

7.1 Probabilistic Turing machines

A randomized algorithm is an algorithm that may involve random choices such as initializing a variable with an integer chosen at random from some range, etc.. In practice randomized algorithms are implemented using a *random number generator*. In fact, it turns out (see Exercise 7.1) that it suffices to have a random number generator that generates *random bits*—they produce the bit 0 with probability $1/2$ and bit 1 with probability $1/2$. We will often describe such generators as tossing *fair coins*.

Just as we used standard Turing machines in Chapter 1 to model deterministic (i.e., non-probabilistic) algorithms, we model randomized algorithms using probabilistic Turing machines (PTMs) which we now define.

Definition 7.1 A probabilistic Turing machine (PTM) is a Turing machine with two transition functions δ_0, δ_1. To execute a PTM M on an input x we choose in each step with probability $1/2$ to apply the transition function δ_0 and with probability $1/2$ to apply δ_1. This choice is made independently of all previous choices.

The machine only outputs 1 ("Accept") or 0 ("Reject"). We denote by $M(x)$ the random variable corresponding to the value M writes at the end of this process. For a function $T : \mathbb{N} \to \mathbb{N}$, we say that M runs in $T(n)$-time if for any input x, M halts on x within $T(|x|)$ steps regardless of the random choices it makes. ♢

Recall from Section 2.1.2 that a nondeterministic TM is also a TM with two transition functions. Thus a PTM is syntactically similar. The difference is in how we interpret the working of the TM. In a PTM, each transitition is taken with probability $1/2$, so a computation that runs for time t gives rise to 2^t *branches* in the graph of all computations,

[1] **BPP** stands for "bounded-error probabilistic polynomial-time;" see chapter notes.

each of which is taken with probability $1/2^t$. Thus $\Pr[M(x) = 1]$ is simply the *fraction* of branches that end with M outputting a 1. The main difference between an NDTM and a PTM lies in how we interpret the graph of all possible computations: an NDTM is said to *accept* the input if there *exists* a branch that outputs 1, whereas in case of a PTM we consider the *fraction* of branches for which this happens. On a conceptual level, PTMs and NDTMs are very different, as PTMs, like deterministic TMs and unlike NDTMs, are intended to model realistic computation devices.

The following class **BPP** aims to capture efficient probabilistic computation. Below, for a language $L \subseteq \{0,1\}^*$ and $x \in \{0,1\}^*$, we define $L(x) = 1$ if $x \in L$ and $L(x) = 0$ otherwise.

Definition 7.2 *(The classes* **BPTIME** *and* **BPP***)*
For $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$ we say that a PTM M decides L in time $T(n)$ if for every $x \in \{0,1\}^*$, M halts in $T(|x|)$ steps regardless of its random choices, and $\Pr[M(x) = L(x)] \geq 2/3$.

We let $\mathbf{BPTIME}(T(n))$ be the class of languages decided by PTMs in $O(T(n))$ time and define $\mathbf{BPP} = \cup_c \mathbf{BPTIME}(n^c)$.

Note that the PTM in the previous definition satisfies a very strong "excluded middle" property: for every input it either accepts it with probability at least $2/3$, or rejects it with probability at least $2/3$. This property makes Definition 7.2 quite *robust*, as we will see in Section 7.4. For instance, we will see that the constant $2/3$ is arbitrary in the sense that it can be replaced with any other constant greater than half without changing the classes $\mathbf{BPTIME}(T(n))$ and **BPP**. We can also make other modifications such as allowing "unfair" coins (that output "Heads" with probability different than $1/2$) or allowing the machine to run in *expected* polynomial-time.

While Definition 7.2 allows the PTM M on input x to output a value different from $L(x)$ (i.e., output the wrong answer) with positive probability, this probability is only over the random choices that M makes in the computation. In particular, for *every* input x, $M(x)$ will output the right value $L(x)$ with probability at least $2/3$. Thus **BPP**, like **P**, is still a class capturing complexity on *worst-case* inputs.

Since a deterministic TM is a special case of a PTM (where both transition functions are equal), the class **BPP** clearly contains **P**. To study the relationship of **BPP** with other classes, it will be helpful to have the following alternative definition.

An alternative definition. As we did with **NP**, we can define **BPP** using deterministic TMs where the sequence of "coin tosses" needed for every step are provided to the TM as an additional input:

Definition 7.3 *(***BPP***, alternative definition)* A language L is in **BPP** if there exists a polynomial-time TM M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$, $\Pr_{r \in_R \{0,1\}^{p(|x|)}}[M(x,r) = L(x)] \geq 2/3$. ♢

From this definition it is clear that $\mathbf{BPP} \subseteq \mathbf{EXP}$ since in time $2^{\mathrm{poly}(n)}$ it is possible to enumerate all the possible random choices of a polynomial-time PTM. Currently researchers only know that **BPP** is sandwiched between **P** and **EXP** but are even unable to show that **BPP** is a proper subset of **NEXP**.

A central open question of complexity theory is whether or not $\mathbf{BPP} = \mathbf{P}$. Based on previous chapters, the reader would probably guess that complexity theorists believe that $\mathbf{BPP} \neq \mathbf{P}$. Not true! Many complexity theorists actually believe that $\mathbf{BPP} = \mathbf{P}$, in other words, there is a way to transform *every* probabilistic algorithm to a deterministic algorithm (one that does not toss any coins) while incurring only a polynomial slowdown. The reasons for this surprising belief are described in Chapters 19 and 20.

7.2 Some examples of PTMs

The following examples demonstrate how randomness can be a useful tool in computation. We will see many more examples in the rest of this book.

7.2.1 Finding a median

A *median* of a set of numbers $\{a_1, \ldots, a_n\}$ is any number x such that at least $\lfloor \frac{n}{2} \rfloor$ of the a_i's are smaller or equal to x and at least $\lfloor \frac{n}{2} \rfloor$ of them are larger or equal to x. Finding a median of a given set of number is useful in many calculations. One simple way to do so is to sort the numbers and then output the $\lfloor \frac{n}{2} \rfloor$ smallest of them, but this takes $O(n \log n)$ time.[2] We now show a simple probabilistic algorithm to find the median in $O(n)$ time. There are known linear time deterministic algorithms for this problem, but the following probabilistic algorithm is still the simplest and most practical known.

Our algorithm will actually solve a more general problem: finding the k^{th} smallest number in the set for every k. It works as follows:

ALGORITHM FINDKTHELEMENT$(k, a_1, \ldots, a_n)$:

1. Pick a random $i \in [n]$ and let $x = a_i$.
2. Scan the list $\{a_1, \ldots, a_n\}$ and count the number m of a_i's such that $a_i \leq x$.
3. If $m = k$ then output x.
4. Otherwise, if $m > k$ then copy to a new list L all elements such that $a_i \leq x$ and run FINDKTHELEMENT(k, L).
5. Otherwise (if $m < k$) copy to a new list H all elements such that $a_i > x$ and run FINDKTHELEMENT$(k - m, H)$.

FINDKTHELEMENT$(k, a_1, \ldots, a_n)$ clearly outputs the k^{th} smallest element and so the only issue is analyzing its running time. Intuitively, we expect that in each recursive call the number of elements will shrink by at least $n/10$ (since in the worst case, where $k = n/2$, we expect to get a new list with roughly $\frac{3}{4}n$ elements). Thus, if $T(n)$ is the running time of the algorithm then it is given by the formula $T(n) = O(n) + T(\frac{9}{10}n)$ which implies $T(n) = O(n)$. We now prove this formally:

Claim 7.4 *For every input $k, a_1, \ldots, a_n$ to* FINDKTHELEMENT*, let $T(k, a_1, \ldots, a_n)$ be the expected number of steps the algorithm takes on this input. Let $T(n)$ be the maximum of $T(k, a_1, \ldots, a_n)$ over all length n inputs. Then $T(n) = O(n)$.* ♢

PROOF: All non-recursive operations of the algorithm can be executed in a linear number of steps: say cn for some constant c. We'll prove by induction that $T(n) \leq 10cn$. Indeed, fix some input $k, a_1, \ldots, a_n$. For every $j \in [n]$ we choose x to be the j^{th} smallest element of $a_1, \ldots, a_n$ with probability $\frac{1}{n}$ and then we perform either at most $T(j)$ steps (if $j > k$) or $T(n - j)$ steps (if $j < k$). Thus, we can see that

$$T(k, a_1, \ldots, a_n) \leq cn + \frac{1}{n}\left(\sum_{j>k} T(j) + \sum_{j<k} T(n-j)\right) .$$

Plugging in our inductive assumption that $T(j) \leq 10cj$ for $j < n$ we get

$$T(k, a_1, \ldots, a_n) \leq cn + \frac{10c}{n}\left(\sum_{j>k} j + \sum_{j<k}(n-j)\right) \leq cn + \frac{10c}{n}\left(\sum_{j>k} j + kn - \sum_{j<k} j\right) .$$

[2] We are assuming here that we can perform basic operations on each number at unit cost. To account for such operations this bound and the bound below needs to include an additional multiplicative factor of k, where k is the number of bits needed to represent each of the a_i's.

Using the fact that $\sum_{j>k} j \leq \frac{n(n-k)}{2}$ and $\sum_{j<k} j \geq \frac{k^2}{2}(1-o(1)) \geq \frac{k^2}{2.5}$ (for large enough k) we get

$$T(k,a_1,\ldots,a_n) \leq cn + \frac{10c}{n}\left(\tfrac{n(n-k)}{2} + kn - \tfrac{k^2}{2.5}\right) = cn + \frac{10c}{n}\left(\tfrac{n^2}{2} + \tfrac{kn}{2} - \tfrac{k^2}{2.5}\right) \leq cn + \frac{10c}{n}\frac{9n^2}{10} = 10cn\,,$$

where the one before last inequality can be shown by considering separately the case $k < n/2$ and the case $k \geq n/2$. ■

7.2.2 Probabilistic Primality Testing

In *primality testing* we are given an integer N and wish to determine whether or not it is prime. Algorithms for primality testing were sought after even before the advent of computers, as mathematicians needed them to test various conjectures[3]. Ideally, we want efficient algorithms that run in time polynomial in the size of N's representation, in other words, poly$(\log N)$ time. For centuries mathematicians knew of no such efficient algorithms for this problem.[4] Then in the 1970's efficient probabilistic algorithms for primality testing were discovered, giving one of the first demonstrations of the power of probabilistic algorithms. We note that in a very recent breakthrough, Agrawal, Kayal and Saxena [AKS04] gave a *deterministic* polynomial-time algorithm for primality testing.

Formally, primality testing consists of checking membership in the following language

$$\mathsf{PRIMES} = \{ \llcorner N \lrcorner : N \text{ is a prime number} \}\ .$$

We now sketch an algorithm showing that PRIMES is in **BPP** (and in fact in **coRP**; see Section 7.3). For every number N, and $A \in [N-1]$, define

$$QR_N(A) = \begin{cases} 0 & gcd(A,N) \neq 1 \\ +1 & \begin{array}{l} A \text{ is a } \textit{quadratic residue} \text{ modulo } N \\ \text{That is, } A = B^2 (\mathrm{mod} N) \text{ for some } B \text{ with } gcd(B,N)=1 \end{array} \\ -1 & \text{otherwise} \end{cases}$$

We use the following facts, all of which can be proven using elementary number theory (e.g., see [Sho05]):

- For every odd prime N and $A \in [N-1]$, $QR_N(A) = A^{(N-1)/2} \pmod N$.
- For every odd N, A define the *Jacobi symbol* $(\frac{N}{A})$ as $\prod_{i=1}^{k} QR_{P_i}(A)$ where $P_1,\ldots,P_k$ are all the (not necessarily distinct) prime factors of N (i.e., $N = \prod_{i=1}^{k} P_i$). Then, the Jacobi symbol is computable in time $O(\log A \cdot \log N)$.
- For every odd composite N, among all $A \in [N-1]$ such that $gcd(N,A)=1$, at most *half* of the A's satisfy $(\frac{N}{A}) = A^{(N-1)/2} \pmod N$.

Together these facts imply a simple algorithm for testing primality of N (which we can assume without loss of generality is odd). *Choose a random* $1 \leq A < N$. *If* $gcd(N,A) > 1$ *or* $(\frac{N}{A}) \neq A^{(N-1)/2} \pmod N$ *then output "composite", otherwise output "prime".* This algorithm will always output "prime" if N is prime, but if N is composite will output "composite" with probability at least 1/2. Of course this probability can be amplified by repeating the test a constant number of times.

Curiously, the *search* problem corresponding to primality testing —finding the factorization of a given composite number N— seems very different and much more difficult. The conjectured hardness of this problem underlies many current cryptosystems, though as we'll see in Chapter 10, it can be solved efficiently in the model of *quantum computers*.

[3] An interesting anecdote is that Gauss, even though he was very fast human computer himself, used the help of a human supercomputer —an autistic person who excelled at fast calculations— to do primality testing.

[4] In fact, in his letter to von Neumann quoted in Chapter 2, Gödel explicitly mentioned this problem as an example for an interesting problem in **NP** not known to be efficiently solvable.

7.2.3 Polynomial identity testing

We now describe a polynomial-time probabilistic algorithm for a problem that has no known efficient deterministic algorithm. The problem is the following: we are given a polynomial with integer coefficients in an implicit form, and we want to decide whether this polynomial is in fact identically zero. We assume we get the polynomial in the form of an *algebraic circuit*. This is analogous to the notion of a Boolean circuit, but instead of the operators $\wedge, \vee$ and $\neg$, we have the operators $+, -$ and $\times$ (see also Section 16.1.3). Formally, an n-variable algebraic circuit is a directed acyclic graph with the sources labeled by a variable name from the set $x_1, \ldots, x_n$, and each non-source node having in-degree two and is labeled by an operator from the set $\{+, -, \times\}$. There is a single sink in the graph which we call the *output* node. The algebraic circuit defines a polynomial from $\mathbb{Z}^n$ to $\mathbb{Z}$ by placing the inputs on the sources and computing the value of each node using the appropriate operator.[5] A simple induction shows that the circuit computes a function $f(x_1, x_2, \ldots, x_n)$ of the inputs that can be described by a multivariate polynomial in $x_1, x_2, \ldots, x_n$. We define ZEROP to be the set of algebraic circuits that compute the identically zero polynomial. Determining membership in ZEROP is also called *polynomial identity testing* since we can reduce the problem of deciding whether two circuits C, C' compute the same polynomial to ZEROP by constructing the circuit D such that $D(x_1, \ldots, x_n) = C(x_1, \ldots, x_n) - C'(x_1, \ldots, x_n)$. The polynomial identity testing problem plays an important role in complexity theory; see for instance Chapters 8, 11, and 20.

The ZEROP problem is nontrivial because a very compact circuit can represent polynomials with a large number of terms. For instance, the polynomial $\prod_i (1+x_i)$ can be computed using a circuit of size $2n$ but has 2^n terms if we open all parentheses. Surprisingly, there is in fact a simple and efficient probabilistic algorithm for testing membership in ZEROP. At the heart of this algorithm is the following fact, often known as the Schwartz-Zippel Lemma, whose proof appears in Appendix A (see Lemma A.36):

Lemma 7.5 *Let $p(x_1, x_2, \ldots, x_m)$ be a non-zero polynomial of total degree at most d.*[6] *Let S be a finite set of integers. Then, if $a_1, a_2, \ldots, a_m$ are randomly chosen with replacement from S, then*

$$\Pr[p(a_1, a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}\,.$$

A size m circuit C contains at most m multiplications and so defines a polynomial of degree at most 2^m. This suggests the following simple probabilistic algorithm: choose n numbers $x_1, \ldots, x_n$ from 1 to $10 \cdot 2^m$ (this requires $O(n \cdot m)$ random bits), evaluate the circuit C on $x_1, \ldots, x_n$ to obtain an output y and then accept if $y = 0$, and reject otherwise. Clearly if $C \in$ ZEROP then we always accept. By Lemma 7.5, if $C \notin$ ZEROP then we will reject with probability at least $9/10$.

However, there is a problem with this algorithm. Since the degree of the polynomial represented by the circuit can be as high as 2^m, the output y and other intermediate values arising in the computation may be as large as $(10 \cdot 2^m)^{2^m}$ — this is a value that requires exponentially many bits just to describe!

We solve this problem using a technique called *fingerprinting*. The idea is to perform the evaluation of C on $x_1, \ldots, x_n$ modulo a number k that is chosen at random in $[2^{2m}]$. Thus, instead of computing $y = C(x_1, \ldots, x_n)$, we compute the value $y \pmod k$. Clearly, if $y = 0$ then $y \pmod k$ is also equal to 0. On the other hand, we claim that if $y \neq 0$, then with probability at least $\delta = \frac{1}{4m}$, k does not divide y— this suffices because we can repeat this procedure $O(1/\delta)$ times and accept only if the output is zero in all these repetitions. Indeed, assume that $y \neq 0$ and let $\mathcal{B} = \{p_1, \ldots, p_\ell\}$ denote the set of distinct prime factors of y. It is sufficient to show that with probability at least δ, the number k will be a prime number not in $\mathcal{B}$. Yet, by the prime number theorem, for sufficiently large m, the number

[5] We can also allow the circuit to contain constants such as $0, 1$ and other numbers, but this does not make much difference in this context.

[6] The total degree of a monomial $x_1^{e_1} \cdot x_2^{e_2} \cdots x_n^{e^m}$ is equal to $e_1 + \cdots + e_m$. The total degree of a polynomial is the largest total degree of its monomials.

of primes in $[2^{2m}]$ is at least $\frac{2^{2m}}{2m}$. Since y can have at most $\log y \leq 5m2^m = o(\frac{2^{2m}}{2m})$ prime factors, for sufficiently large m, the number of k's in $[2^{2m}]$ such that k is prime and is not in $\mathcal{B}$ is at least $\frac{2^{2m}}{4m}$, meaning that a random k will have this property with probability at least $\frac{1}{4m} = \delta$.

7.2.4 Testing for perfect matching in a bipartite graph.

Let $G = (V, E)$ be a bipartite graph with two equal parts. That is, $V = V_1 \cup V_2$ where V_1, V_2 are disjoint and of equal size, and $E \subseteq V_1 \times V_2$. A *perfect matching* in G is a subset of edges $E' \subseteq E$ such that every vertex appears exactly once in E'. Alternatively, setting $n = |V_1| = |V_2|$ and identifying both these sets with the set $[n]$, we may think of E' as a permutation $\sigma : [n] \to [n]$ mapping every $i \in [n]$ to the unique $j \in [n]$ such that $\overline{i\,j} \in E'$. Several deterministic algorithms are known for detecting if a perfect matching exists in a given graph. Here we describe a very simple randomized algorithm (due to Lovász) using the Schwartz-Zippel lemma.

For a $2n$-vertex bipartite graph $G = (V, E)$ as above, let X be an $n \times n$ matrix of real variables whose $(i, j)^{th}$ entry $X_{i,j}$ is equal to the variable $x_{i,j}$ if the edge $\overline{i\,j}$ is in E and equal to 0 otherwise. Recall that the determinant of a matrix A is defined as follows:

$$\det(A) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} A_{i,\sigma(i)}, \tag{1}$$

where S_n is the set of all permutations of $[n]$ and $sgn(\sigma)$ is the parity of the number of transposed pairs in σ (i.e., pairs $\langle i, j \rangle$ such that $i < j$ but $\sigma(i) > \sigma(j)$). Thus, $\det(X)$ is a degree n polynomial in the variables $\{x_{i,j}\}_{\overline{i\,j} \in E}$ that has a monomial for every perfect matching that exists in the graph. In other words, G has a perfect matching if and only if $\det(X)$ is not the identically zero polynomial. Now, even though $\det(X)$ may have exponentially many monomials, for every setting of values to the $x_{i,j}$ variables $\det(X)$ can be efficiently evaluated using the well known algorithm for computing determinants.

This leads, in conjunction with Lemma 7.5, to Lovász's randomized algorithm: pick random values for $x_{i,j}$'s from $[2n]$, substitute them in X and compute the determinant. If the determinant is nonzero, output "accept" else output "reject." Besides its simplicity, this algorithm also has an advantage that it has an efficient parallel implementation (using the **NC** algorithm for computing determinants; see Exercise 6.16 in Chapter 6).

7.3 One-sided and "zero-sided" error: RP, coRP, ZPP

The class **BPP** captures what we call probabilistic algorithms with *two sided* error. That is, it allows an algorithm for a language L to output (with some small probability) both 0 when $x \in L$ and 1 when $x \notin L$. However, many probabilistic algorithms have the property of *one sided* error. For example if $x \notin L$ they will *never* output 1, although they may output 0 when $x \in L$. This type of behavior is captured by the class **RP** which we now define:

Definition 7.6 $\mathbf{RTIME}(T(n))$ contains every language L for which there is a is a probabilistic TM M running in $T(n)$ time such that

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq \frac{2}{3}$$
$$x \notin L \Rightarrow \Pr[(x) = 0] = 0$$

We define $\mathbf{RP} = \cup_{c>0}\mathbf{RTIME}(n^c)$. ♢

Note that $\mathbf{RP} \subseteq \mathbf{NP}$, since every accepting branch is a "certificate" that the input is in the language. In contrast, we do not know if $\mathbf{BPP} \subseteq \mathbf{NP}$. The class $\mathbf{coRP} = \{L \mid \overline{L} \in \mathbf{RP}\}$ captures one-sided error algorithms with the error in the "other direction" (i.e., may output 1 when $x \notin L$ but will never output 0 if $x \in L$).

"Zero sided" error. For a PTM M, and input x, we define the random variable $T_{M,x}$ to be the running time of M on input x. That is, $\Pr[T_{M,x} = T] = p$ if with probability p over the random choices of M on input x, it will halt within T steps. We say that M has *expected running time* $T(n)$ if the expectation $\mathsf{E}[T_{M,x}]$ is at most $T(|x|)$ for every $x \in \{0,1\}^*$. We now define PTMs that never err (also called "zero error" machines).

Definition 7.7 The class $\mathbf{ZTIME}(T(n))$ contains all the languages L for which there is an expected-time $O(T(n))$ machine M such that for every input x, whenever M halts on x, the output $M(x)$ it produces is exactly $L(x)$.

We define $\mathbf{ZPP} = \cup_{c>0}\mathbf{ZTIME}(n^c)$. ♢

The next theorem ought to be slightly surprising, since the corresponding question for nondeterminism (i.e., whether or not $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$) is open.

Theorem 7.8 $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$. ♢

We leave the proof of this theorem to the reader (see Exercise 7.6). To summarize, we have the following relations between the probabilistic complexity classes:

$$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$$
$$\mathbf{RP} \subseteq \mathbf{BPP}$$
$$\mathbf{coRP} \subseteq \mathbf{BPP}$$

7.4 The robustness of our definitions

When we defined $\mathbf{P}$ and $\mathbf{NP}$, we argued that our definitions are robust and are likely to be the same for an alien studying the same concepts in a faraway galaxy. Now we address similar issues for probabilistic computation.

7.4.1 Role of precise constants: error reduction.

The choice of the constant $2/3$ seemed pretty arbitrary. We now show that we can replace $2/3$ with any constant larger than $1/2$ and in fact even with $1/2 + n^{-c}$ for a constant $c > 0$.

Lemma 7.9 *For $c > 0$, let $\mathbf{BPP}_{1/2+n^{-c}}$ denote the class of languages L for which there is a polynomial-time PTM M satisfying $\Pr[M(x) = L(x)] \geq 1/2 + |x|^{-c}$ for every $x \in \{0,1\}^*$. Then $\mathbf{BPP}_{1/2+n^{-c}} = \mathbf{BPP}$.* ♢

Since clearly $\mathbf{BPP} \subseteq \mathbf{BPP}_{1/2+n^{-c}}$, to prove this lemma we need to show that we can transform a machine with success probability $1/2 + n^{-c}$ into a machine with success probability $2/3$. We do this by proving a much stronger result: we transform such a machine into a machine with success probability exponentially close to one!

Theorem 7.10 *(Error reduction for $\mathbf{BPP}$)*
Let $L \subseteq \{0,1\}^*$ be a language and suppose that there exists a polynomial-time PTM M such that for every $x \in \{0,1\}^*$, $\Pr[M(x) = L(x)] \geq \frac{1}{2} + |x|^{-c}$.
Then for every constant $d > 0$ there exists a polynomial-time PTM M' such that for every $x \in \{0,1\}^*$, $\Pr[M'(x) = L(x)] \geq 1 - 2^{-|x|^d}$.

PROOF: The machine M' simply does the following: *for every input $x \in \{0,1\}^*$, run $M(x)$ for $k = 8|x|^{2c+d}$ times obtaining k outputs $y_1, \ldots, y_k \in \{0,1\}$. If the majority of these outputs is 1 then output 1, otherwise output 0.*

To analyze this machine, define for every $i \in [k]$ the random variable X_i to equal 1 if $y_i = L(x)$ and to equal 0 otherwise. Note that $X_1, \ldots, X_k$ are independent Boolean random variables with $\mathsf{E}[X_i] = \Pr[X_i = 1] \geq p$ for $p = 1/2 + |x|^{-c}$. The Chernoff bound (see Corollary A.15) implies that for δ sufficiently small:

$$\Pr\Big[\big|\sum_{i=1}^{k} X_i - pk\big| > \delta pk\Big] < e^{-\frac{\delta^2}{4}pk}.$$

In our case $p = 1/2+|x|^{-c}$ and setting $\delta = |x|^{-c}/2$ guarantees that if $\sum_{i=1}^{k} X_i \geq pk-\delta pk$ then we will output the right answer. Hence, the probability we output a wrong answer is bounded by

$$e^{-\frac{1}{4|x|^{2c}}\frac{1}{2}8|x|^{2c+d}} \leq 2^{-|x|^d}. \blacksquare$$

A similar (and even easier to prove) result holds for the one-sided error classes **RP** and **coRP**; see Exercise 7.4. In that case we can even change the constant $2/3$ to values smaller than $1/2$.

These error reduction results imply that we can take a probabilistic algorithm that succeeds with quite modest probability and transform it into an algorithm that succeeds with overwhelming probability. In fact, even for moderate values of n an error probability that is of the order of 2^{-n} is so small that for all practical purposes, probabilistic algorithms are just as good as deterministic algorithms.

Randomness-efficient repetitions. The proof of Theorem 7.10 uses $O(k)$ independent repetitions to transform an algorithm with success probability $2/3$ into an algorithm with success probability $1 - 2^{-k}$. Thus, if the original used m random coins then the new algorithm will use $O(km)$ coins. Surprisingly, we can do better: there is a transformation that only uses $O(m + k)$ random coins to achieve the same error reduction. This transformation will be described in Chapter 21 (Section 21.2.5).

7.4.2 Expected running time versus worst-case running time.

When defining $\mathbf{RTIME}(T(n))$ and $\mathbf{BPTIME}(T(n))$ we required the machine to halt in $T(n)$ time regardless of its random choices. We could have used *expected* running time instead, as in the definition of **ZPP** (Definition 7.7). It turns out this yields an equivalent definition: we can transform a PTM M whose expected running time is $T(n)$ to a PTM M' that always halts after at most $100T(n)$ steps by simply adding a counter and halting with an arbitrary output after too many steps have gone by. By Markov's inequality (see Lemma A.7), the probability that M runs for more than $100T(n)$ steps is at most $1/100$ and so this will change the acceptance probability by at most $1/100$.

7.4.3 Allowing more general random choices than a fair random coin.

One could conceive of real-life computers that have a "coin" that comes up heads with probability ρ that is not $1/2$. We call such a coin a ρ-coin. Indeed it is conceivable that for a random source based upon quantum mechanics, ρ is an irrational number, such as $1/e$. Could such a coin give probabilistic algorithms new power? The following claim shows that it will not, at least if ρ is efficiently computable. (The exercises show that if ρ is not efficiently computable, then a ρ-coin can indeed provide additional power.)

Lemma 7.12 *A coin with* $\Pr[Heads] = \rho$ *can be simulated by a PTM in expected time* $O(1)$ *provided the* i*th bit of* ρ *is computable in* $\mathrm{poly}(i)$ *time.* $\diamondsuit$

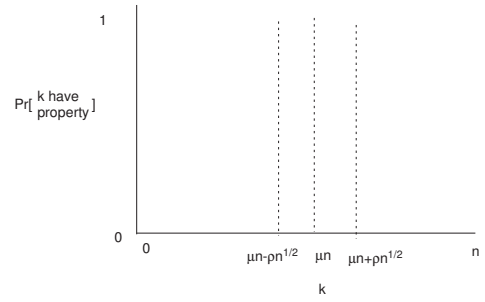
Proof: Let the binary expansion of ρ be $0.p_1p_2p_3\ldots$. The PTM generates a sequence of random bits $b_1, b_2, \ldots$, one by one, where b_i is generated at step i. If $b_i < p_i$ then the machine

Note 7.11 *(The Chernoff Bound and Statistical Estimation)*

The Chernoff bound is extensively used (sometimes under different names) in many areas of computer science and other sciences. A typical scenario is the following: there is a universe $\mathcal{U}$ of objects, a fraction μ of them have a certain property, and we wish to estimate μ. For example, in the proof of Theorem 7.10 the universe was the set of 2^m possible coin tosses of some probabilistic algorithm and we wanted to know how many of them make the algorithm accept its input. As another example, $\mathcal{U}$ can be the set of all the citizens of the United States, and we wish to find out how many of them own a dog.

A natural approach for computing the fraction μ is to *sample* n members of the universe independently at random, find out the number k of the sample's members that have the property, and then guess that μ is k/n. Of course, a guess based upon a small sample is unlikely to produce the exact answer. For instance, the true fraction of dog owners may be 10% but in a sample of size say 1000 we may find that only 99 (i.e., 9.9%) people are dog owners. So we set our goal only to *estimate* the true fraction μ up to an *error* of $\pm\epsilon$ for some $\epsilon > 0$. Despite allowing ourselves this error margin, we may get really unlucky and our sample may turn out to be really unrepresentative—e.g., there is a non-zero probability that the entire sample of 1000 consists of dog owners. So we allow a small *probability of failure* δ that our estimate will not lie in the interval $[\mu - \epsilon, \mu + \epsilon]$. The natural question is *how many samples do we need in order to estimate μ up to an error of $\pm\epsilon$ with probability at least $1 - \delta$?* The Chernoff bound tells us that (considering μ as a constant) this number is $O(\log(1/\delta)/\epsilon^2)$.

Setting $\rho = \log(1/\delta)$, this implies that the probability that k is $\rho\sqrt{n}$ far from μn decays *exponentially* with ρ. That is, this probability has the famous "bell curve" shape:



We will use this exponential decay phenomena many times in this book. See for instance the proof of Theorem 7.14, showing that $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

outputs "heads" and stops; if $b_i > p_i$ the machine outputs "tails" and halts; otherwise the machine goes to step $i+1$. Clearly, the machine reaches step $i+1$ iff $b_j = p_j$ for all $j \leq i$, which happens with probability $1/2^i$. Thus the probability of "heads" is $\sum_i p_i \frac{1}{2^i}$, which is exactly ρ. Furthermore, the expected running time is $\sum_i i^c \cdot \frac{1}{2^i}$. For every constant c this infinite sum is bounded by another constant (see Exercise 7.2). ■

Conversely, probabilistic algorithms that only have access to ρ-coins do not have less power than standard probabilistic algorithms:

Lemma 7.13 *(von-Neumann [vN51]) A coin with $\Pr[Heads] = 1/2$ can be simulated by a probabilistic TM with access to a stream of ρ-biased coins in expected time $O(\frac{1}{\rho(1-\rho)})$.* ♢

PROOF: We construct a TM M that given the ability to toss ρ-coins, outputs a 1/2-coin. The machine M tosses pairs of coins until the first time it gets a pair containing two different results (i.e., "Heads-Tails" or "Tails-Heads"). Then, if the first of these two results is "Heads" it outputs "Heads" and otherwise it outputs "Tails".

The probability that a pair of coins comes up "Head-Tails" is $\rho(1-\rho)$, while the probability it comes up "Tails-Heads" is $(1-\rho)\rho = \rho(1-\rho)$. Hence, in each step M halts with probability $2\rho(1-\rho)$, and conditioned on M halting in a particular step, the outputs "Heads" and "Tails" are equiprobable (i.e., M's output is a fair coin). Note that we did not need to know ρ to run this simulation. ■

Weak random sources. Physicists (and philosophers) are still not completely certain that randomness exists in the world, and even if it does, it is not clear that our computers have access to an endless stream of independent coins. Conceivably, it may be the case that we only have access to a source of *imperfect* randomness, that although unpredictable, does not consist of independent coins. As we will see in Chapter 21, we do know how to simulate probabilistic algorithms designed for perfect independent 1/2-coins even using such a weak random source.

7.5 Relationship between BPP and other classes

Below we will show that $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$. Thus $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$. Furthermore, we show that $\mathbf{BPP} \subseteq \mathbf{\Sigma}_2^p \cap \mathbf{\Pi}_2^p$ and so if $\mathbf{NP} = \mathbf{P}$ then $\mathbf{BPP} = \mathbf{P}$. Of course, since we do not believe $\mathbf{P} = \mathbf{NP}$, this still leaves open the possibility that $\mathbf{P} \neq \mathbf{BPP}$. However, as mentioned above (and will be elaborated in Chapters 19 and 20) we can show that if certain plausible complexity-theoretic conjectures are true then $\mathbf{BPP} = \mathbf{P}$. Thus we suspect that **BPP** is the same as **P** and hence (by the Time Hierarchy theorem) **BPP** is a proper subset of, say, $\mathbf{DTIME}(n^{\log n})$. Yet currently researchers are not even able to show that **BPP** is a proper subset of **NEXP**.

7.5.1 $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$

Now we show that all **BPP** languages have polynomial sized circuits. Together with Theorem 6.19 this shows that unless the polynomial-hierarchy collapses, 3SAT cannot be solved in probabilistic polynomial time.

Theorem 7.14 *([Adl78])*
$\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

PROOF: Suppose $L \in \mathbf{BPP}$, then by the alternative definition of **BPP** and the error reduction procedure of Theorem 7.10, there exists a TM M that on inputs of size n uses m