Chapter 6

Randomness and Counting

I owe this almost atrocious variety to an institution which other republics do not know or which operates in them in an imperfect and secret manner: the lottery.

Jorge Luis Borges, The Lottery In Babylon

So far, our approach to computing devices was somewhat conservative: we thought of them as executing a deterministic rule. A more liberal and quite realistic approach, which is pursued in this chapter, considers computing devices that use a probabilistic rule. This relaxation has an immediate impact on the notion of efficient computation, which is consequently associated with *probabilistic* polynomial-time computations rather than with deterministic (polynomial-time) ones. We stress that the association of efficient computation with probabilistic polynomial-time computation makes sense provided that the failure probability of the latter is negligible (which means that it may be safely ignored).

The quantitative nature of the failure probability of probabilistic algorithm provides one connection between probabilistic algorithms and counting problems. The latter are indeed a new type of computational problems, and our focus is on counting efficiently recognizable objects (e.g., NP-witnesses for a given instance of set in $\mathcal{NP}$). Randomized procedures turn out to play an important role in the study of such counting problems.

Summary: Focusing on probabilistic polynomial-time algorithms, we consider various types of probabilistic failure of such algorithms (e.g., actual error versus failure to produce output). This leads to the formulation of complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. The results presented include the existence of (non-uniform) families of polynomial-size circuits that emulate probabilistic polynomial-time algorithms (i.e., $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$) and the fact that $\mathcal{BPP}$ resides in the (second level of the) Polynomial-time Hierarchy (i.e., $\mathcal{BPP} \subseteq \Sigma_2$).

We then turn to counting problems; specifically, counting the number of solutions for an instance of a search problem in $\mathcal{PC}$ (or, equivalently,

counting the number of NP-witnesses for an instance of a decision problem in $\mathcal{NP}$). We distinguish between exact counting and approximate counting (in the sense of relative approximation). In particular, while any problem in $\mathcal{PH}$ is reducible to the exact counting class $\#\mathcal{P}$, approximate counting (for $\#\mathcal{P}$) is (probabilisticly) reducible to $\mathcal{NP}$.

Additional related topics include the $\#\mathcal{P}$-completeness of various counting problems (e.g., counting the number of satisfying assignments to a given CNF formula and counting the number of perfect matchings in a given graph), the complexity of searching for *unique solutions*, and the relation between *approximate counting* and *generating random solutions* (i.e., generating almost uniformly distributed solutions).

Prerequisites: We assume basic familiarity with elementary probability theory (see Appendix D.1). In Section 6.2 we will rely extensively on formulations presented in Section 2.1 (i.e., the "NP search problem" class $\mathcal{PC}$ as well as the sets $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$, and $S_R \stackrel{\text{def}}{=} \{X : R(x) \neq \emptyset\}$ defined for every $R \in \mathcal{PC}$). In Sections 6.2.2–6.2.4 we shall extensively use various hashing functions and their properties, as presented in Appendix D.2.

6.1 Probabilistic Polynomial-Time

Considering algorithms that utilize random choices, we extend our notion of *efficient algorithms* from *deterministic* polynomial-time algorithms to *probabilistic* polynomial-time algorithms. An immediate question that arises is whether this extension buys us anything. Although randomization is known to be essential in several computational settings (e.g., cryptography (cf., Appendix C) and sampling (cf., Appendix D.3)), the question is whether randomization is useful in the context of solving decision (and search) problems. This is indeed a very good question, which is further discussed in §6.1.1.1. In fact, one of the main goals of the current section is putting this question forward. To demonstrate the potential benefit of randomized algorithms, we provide a few examples (cf., §6.1.1.2, §6.1.2.1 and §6.1.4.2).

Rigorous models of probabilistic (or randomized) algorithms are defined by natural extensions of the basic machine model. We will exemplify this approach by describing the model of probabilistic Turing machines, but we stress that (again) the specific choice of the model is immaterial (as long as it is "reasonable"). A probabilistic Turing machine is defined exactly as a non-deterministic machine (see the first item of Definition 2.7), *but the definition of its computation is fundamentally different.* Specifically, whereas Definition 2.7 refers to the question of whether or not there exists a computation of the machine that (started on a specific input) reaches a certain configuration, in case of probabilistic Turing machines we refer to *the probability that this event occurs, when at each step a choice is selected uniformly among the relevant possible choices available at this step.* That is, if the transition function of the machine maps the current state-symbol pair to several

possible triples, then in the corresponding probabilistic computation one of these triples is selected at random (with equal probability) and the next configuration is determined accordingly. These random choices may be viewed as the internal coin tosses of the machine. (Indeed, as in the case of non-deterministic machines, we may assume without loss of generality that the transition function of the machine maps each state-symbol pair to exactly two possible triples; see Exercise 2.4.)

We stress the fundamental difference between the fictitious model of a non-deterministic machine and the realistic model of a probabilistic machine. In the case of a non-deterministic machine we consider the *existence* of an adequate sequence of choices (leading to a desired outcome), and ignore the question of how these choices are actually made. In fact, the selection of such a sequence of choices is merely a mental experiment. In contrast, in the case of a probabilistic machine, at each step a real random choice is made (uniformly among a set of predetermined possibilities), and we consider the *probability* of reaching a desired outcome.

In view of the foregoing, we consider the output distribution of such a probabilistic machine on fixed inputs; that is, for a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the output distribution of M when invoked on input x, where the probability is taken uniformly over the machine's internal coin tosses. Needless to say, we will consider the probability that $M(x)$ is a "correct" answer; that is, in the case of a search problem (resp., decision problem) we will be interested in the probability that $M(x)$ is a valid solution for the instance x (resp., represents the correct decision regarding x).

The foregoing description views the internal coin tosses of the machine as taking place on-the-fly; that is, these coin tosses are performed *on-line* by the machine itself. An alternative model is one in which the sequence of coin tosses is provided by an external device, on a special "random input" tape. In such a case, we view these coin tosses as performed *off-line*. Specifically, we denote by $M'(x,r)$ the (uniquely defined) output of the residual deterministic machine M', when given the (primary) input x and random input r. Indeed, M' is a deterministic machine that takes two inputs (the first representing the actual input and the second representing the "random input"), but we consider the random variable $M(x) \stackrel{\rm def}{=} M'(x, U_{\ell(|x|)})$, where $\ell(|x|)$ denotes the number of coin tosses "expected" by $M'(x,\cdot)$.

These two perspectives on probabilistic algorithms are closely related: Clearly, the aforementioned residual deterministic machine M' yields the on-line machine M that on input x selects at random a string r of adequate length, and invokes $M'(x,r)$. On the other hand, the computation of any on-line machine M is captured by the residual machine M' that emulates the actions of $M(x)$ based on an auxiliary input r (obtained by M' and representing a possible outcome of the internal coin tosses of M). (Indeed, there is no harm in supplying more coin tosses than are actually used by M, and so the length of the aforementioned auxiliary input may be set to equal the time complexity of M.) For sake of clarity and future reference, we state the following definition.

Definition 6.1 (on-line and off-line formulations of probabilistic polynomial-time):

- *We say that M is a* on-line probabilistic polynomial-time machine *if there exists*

a polynomial p such that when invoked on any input $x \in \{0,1\}^*$, machine M always halts within at most $p(|x|)$ steps (regardless of the outcome of its internal coin tosses). *In such a case $M(x)$ is a random variable.*

- *We say that M' is a* off-line probabilistic polynomial-time machine *if there exists a polynomial p such that, for every $x \in \{0,1\}^*$ and $r \in \{0,1\}^{p(|x|)}$, when invoked on the* primary input *x and the* random-input *sequence r, machine M' halts within at most $p(|x|)$ steps. In such a case, we will consider the random variable $M'(x, U_{p(|x|)})$.*

Clearly, in the context of time-complexity, the on-line and off-line formulations are equivalent (i.e., given a on-line probabilistic polynomial-time machine we can derive a functionally equivalent off-line (probabilistic polynomial-time) machine, and vice versa). Thus, in the sequel, we will freely use whichever is more convenient.

Failure probability. A major aspect of randomized algorithms (probabilistic machines) is that they may fail (see Exercise 6.1). That is, with some specified ("failure") probability, these algorithms may fail to produce the desired output. We discuss two aspects of this failure: its *type* and its *magnitude*.

1. The type of failure is a qualitative notion. One aspect of this type is whether, in case of failure, the algorithm produces a wrong answer or merely an indication that it failed to find a correct answer. Another aspect is whether failure may occur on all instances or merely on certain types of instances. Let us clarify these aspects by considering three natural types of failure, giving rise to three different types of algorithms.

   (a) The most liberal notion of failure is the one of two-sided error. This term originates from the setting of decision problems, where it means that (in case of failure) the algorithm may err in both directions (i.e., it may rule that a yes-instance is a no-instance, and vice versa). In the case of search problems two-sided error means that, when failing, the algorithm may output a wrong answer on any input. Furthermore, the algorithm may falsely rule that the input has no solution and it may also output a wrong solution (both in case the input has a solution and in case it has no solution).

   (b) An intermediate notion of failure is the one of one-sided error. Again, the term originates from the setting of decision problems, where it means that the algorithm may err only in one direction (i.e., either on yes-instances or on no-instances). Indeed, there are two natural cases depending on whether the algorithm errs on yes-instances but not on no-instances, or the other way around. Analogous cases occur also in the setting of search problems. In one case the algorithm never outputs a wrong solution but may falsely rule that the input has no solution. In the other case the indication that an input has no solution is never wrong, but the algorithm may output a wrong solution.

(c) The most conservative notion of failure is the one of zero-sided error. In this case, the algorithm's failure amounts to indicating its failure to find an answer (by outputting a special `don't know` symbol). We stress that in this case the algorithm *never provides a wrong answer.*

Indeed, the forgoing discussion ignores the probability of failure, which is the subject of the next item.

2. The magnitude of failure is a quantitative notion. It refer to the probability that the algorithm fails, where the type of failure is fixed (e.g., as in the forgoing discussion).

   When actually using a randomized algorithm we typically wish its failure probability to be negligible, which intuitively means that the failure event is so rare that it can be ignored in practice. Formally, we say that a quantity is negligible if, as a function of the relevant parameter (e.g., the input length), this quantity vanishes faster than the reciprocal of any positive polynomial.

   For ease of presentation, we sometimes consider alternative upper-bounds on the probability of failure. These bounds are selected in a way that allows (and in fact facilitates) "error reduction" (i.e., converting a probabilistic polynomial-time algorithm that satisfies such an upper-bound into one in which the failure probability is negligible). For example, in case of two-sided error we need to be able to distinguish the correct answer from wrong answers by sampling, and in the other types of failure "hitting" a correct answer suffices.

In the following three subsections, we will discuss complexity classes corresponding to the aforementioned three types of failure. For sake of simplicity, the failure probability itself will be set to a constant that allows error reduction.

Randomized reductions. Before turning to the more detailed discussion, we note that randomized reductions play an important role in complexity theory. Such reductions can be defined analogously to the standard Cook-Reductions (resp., Karp-reductions), and again a discussion of the type and magnitude of the failure probability is in place. For clarity, we spell-out the two-sided error versions.

- In analogy to Definition 2.9, we say that a problem Π is probabilistic polynomial-time reducible to a problem Π' if there exists a probabilistic polynomial-time oracle machine M such that, for every function f that solves Π' and for every x, with probability at least $1-\mu(|x|)$, the output $M^f(x)$ is a correct solution to the instance x, where μ is a negligible function.

- In analogy to Definition 2.10, we say that a decision problem S is reducible to a decision problem S' via a randomized Karp-reduction if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $\Pr[\chi_{S'}(A(x)) = \chi_S(x)] \geq 1-\mu(|x|)$, where χ_S (resp., $\chi_{S'}$) is the characteristic function of S (resp., S') and μ is a negligible function.

These reductions preserve efficient solvability and are transitive: see Exercise 6.2.

6.1.1 Two-sided error: The complexity class BPP

In this section we consider the most liberal notion of probabilistic polynomial-time algorithms that is still meaningful. We allow the algorithm to err on each input, but require the error probability to be *negligible*. The latter requirement guarantees the usefulness of such algorithms, because in reality we may ignore the negligible error probability.

Before focusing on the decision problem setting, let us say a few words on the search problem setting (see Definition 1.1). Following the previous paragraph, we say that a probabilistic (polynomial-time) algorithm A solves the search problem of the relation R if for every $x \in S_R$ (i.e., $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\} \neq \emptyset$) it holds that $\Pr[A(x) \in R(x)] > 1 - \mu(|x|)$ and for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 1 - \mu(|x|)$, where μ is a negligible function. Note that we did not require that, when invoked on input x that has a solution (i.e., $R(x) \neq \emptyset$), the algorithm always outputs the same solution. Indeed, a stronger requirement is that for every such x there exists $y \in R(x)$ such that $\Pr[A(x) = y] > 1 - \mu(|x|)$. The latter version and quantitative relaxations of it allow for error-reduction (see Exercise 6.3).

Turning to decision problems, we consider probabilistic polynomial-time algorithms that err with negligible probability. That is, we say that a probabilistic (polynomial-time) algorithm A decides membership in S if for every x it holds that $\Pr[A(x) = \chi_S(x)] > 1 - \mu(|x|)$, where χ_S is the characteristic function of S (i.e., $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise) and μ is a negligible function. The class of decision problems that are solvable by probabilistic polynomial-time algorithms is denoted $\mathcal{BPP}$, standing for Bounded-error Probabilistic Polynomial-time. Actually, the standard definition refers to machines that err with probability at most $1/3$.

Definition 6.2 (the class $\mathcal{BPP}$): *A decision problem S is in $\mathcal{BPP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

The choice of the constant $2/3$ is immaterial, and any other constant greater than $1/2$ will do (and yields the very same class). Similarly, the complementary constant $1/3$ can be replaced by various negligible functions (while preserving the class). Both facts are special cases of the robustness of the class, which is established using the process of error reduction.

Error reduction (or confidence amplification). For $\varepsilon : \mathbb{N} \to (0, 0.5)$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε; that is, $S \in \mathcal{BPP}_\varepsilon$ if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\Pr[A(x) \neq \chi_S(x)] \leq \varepsilon(|x|)$. By definition, $\mathcal{BPP} = \mathcal{BPP}_{1/3}$. However, a wide range of other classes also equal $\mathcal{BPP}$. In particular, we mention two extreme cases:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, any error that is ("noticeably") bounded away from

1/2 (i.e., error $(1/2) - (1/\text{poly}(n))$) can be reduced to an error of 1/3.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, an error of 1/3 can be further reduced to an exponentially vanishing error.

Both facts are proved by invoking the weaker algorithm (i.e., the one having a larger error probability bound) for an adequate number of times, and ruling by majority. We stress that invoking a randomized machine several times means that the random choices made in the various invocations are independent of one another. The success probability of such a process is analyzed by applying an adequate Law of Large Numbers (see Exercise 6.4).

6.1.1.1 On the power of randomization

Let us turn back to the natural question raised at the beginning of Section 6.1; that is, *was anything gained by extending the definition of efficient computation to include also probabilistic polynomial-time ones.*

This phrasing seems too generic. We certainly gained the ability to toss coins (and generate various distributions). More concretely, randomized algorithms are essential in many settings (see, e.g., Chapter 9, Section 10.1.2, Appendix C, and Appendix D.3) and seem essential in others (see, e.g., Sections 6.2.2–6.2.4). What we mean to ask here is *whether allowing randomization increases the power of polynomial-time algorithms also in the restricted context of solving decision and search problems?*

The question is whether $\mathcal{BPP}$ extends beyond $\mathcal{P}$ (where clearly $\mathcal{P} \subseteq \mathcal{BPP}$). It is commonly conjectured that the answer is negative. Specifically, under some reasonable assumptions, it holds that $\mathcal{BPP} = \mathcal{P}$ (see Part 1 of Theorem 8.19). We note, however, that a polynomial slow-down occurs in the proof of the latter result; that is, randomized algorithms that run in time $t(\cdot)$ are emulated by deterministic algorithms that run in time $\text{poly}(t(\cdot))$. This slow-down seems inherent to the aforementioned approach (see §8.3.3.2). Furthermore, for some concrete problems (most notably primality testing (cf. §6.1.1.2)), the known probabilistic polynomial-time algorithm is significantly faster (and conceptually simpler) than the known deterministic polynomial-time algorithm. Thus, we believe that even in the context of decision problems, the notion of probabilistic polynomial-time algorithms is advantageous.

We note that the fundamental nature of $\mathcal{BPP}$ will remain intact even in the (rather unlikely) case that it turns out that randomization offers no computational advantage (i.e., even if every problem that can be decided in probabilistic polynomial-time can be decided by a deterministic algorithm of essentially the same complexity). Such a result would address a fundamental question regarding the power of randomness.[1]

[1] By analogy, establishing that $\mathcal{IP} = \mathcal{PSPACE}$ (cf. Theorem 9.4) does not diminish the importance of any of these classes, because each class models something fundamentally different.

BPP is in the Polynomial-Time Hierarchy: While it may be that $\mathcal{BPP} = \mathcal{P}$, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. The source of trouble is the two-sided error probability of $\mathcal{BPP}$, which is incompatible with the absolute rejection of no-instances required in the definition of $\mathcal{NP}$ (see Exercise 6.8). In view of this ignorance, it is interesting to note that $\mathcal{BPP}$ resides in the second level of the Polynomial-Time Hierarchy (i.e., $\mathcal{BPP} \subseteq \Sigma_2$). This is a corollary of Theorem 6.9.

Trivial derandomization. A straightforward way of eliminating randomness from an algorithm is trying all possible outcomes of its internal coin tosses, collecting the relevant statistics and deciding accordingly. This yields $\mathcal{BPP} \subseteq \mathcal{PSPACE} \subseteq \mathcal{EXP}$, which is considered the trivial derandomization of $\mathcal{BPP}$. In Section 8.3 we will consider various non-trivial derandomizations of $\mathcal{BPP}$, which are known under various intractability assumptions. The interested reader, who may be puzzled by the connection between derandomization and computational difficulty, is referred to Chapter 8.

Non-uniform derandomization. In many settings (and specifically in the context of solving search and decision problems), the power of randomization is superseded by the power of non-uniform advice. Intuitively, the non-uniform advice may specify a sequence of coin tosses that is good for all (primary) inputs of a specific length. In the context of solving search and decision problems, such an advice must be good for *each* of these inputs[2], and thus its existence is guaranteed only if the error probability is low enough (so as to support a union bound). The latter condition can be guaranteed by error-reduction, and thus we get the following result.

Theorem 6.3 *$\mathcal{BPP}$ is* (strictly) *contained in $\mathcal{P}$/*poly*.*

Proof: Recall that $\mathcal{P}$/poly contains undecidable problems (Theorem 3.7), which are certainly not in $\mathcal{BPP}$. Thus, we focus on showing that $\mathcal{BPP} \subseteq \mathcal{P}$/poly. By the discussion regarding error-reduction, for every $S \in \mathcal{BPP}$ there exists a (deterministic) polynomial-time algorithm A and a polynomial p such that for every x it holds that $\Pr[A(x, U_{p(|x|)}) \neq \chi_S(x)] < 2^{-|x|}$. Using a union bound, it follows that $\Pr_{r \in \{0,1\}^{p(n)}}[\exists x \in \{0,1\}^n \text{ s.t. } A(x,r) \neq \chi_S(x)] < 1$. Thus, for every $n \in \mathbb{N}$, there exists a string $r_n \in \{0,1\}^{p(n)}$ such that for every $x \in \{0,1\}^n$ it holds that $A(x, r_n) = \chi_S(x)$. Using such a sequence of r_n's as advice, we obtain the desired non-uniform machine (establishing $S \in \mathcal{P}$/poly). ■

Digest. The proof of Theorem 6.3 combines error-reduction with a simple application of the Probabilistic Method (cf. [10]), where the latter refers to proving the existence of an object by analyzing the probability that a random object is adequate. In this case, we sought an non-uniform advice, and proved it existence

[2]In other contexts (see, e.g., Chapters 7 and 8), it suffices to have an advice that is good on the average, where the average is taken over all relevant (primary) inputs.

by analyzing the probability that a random advice is good. The latter event was analyzed by identifying the space of advice with the set of possible sequences of internal coin tosses of a randomized algorithm.

6.1.1.2 A probabilistic polynomial-time primality test

Teaching note: Although primality has been recently shown to be in $\mathcal{P}$, we believe that the following example provides a nice illustration to the power of randomized algorithms.

We present a simple probabilistic polynomial-time algorithm for deciding whether or not a given number is a prime. The only Number Theoretic facts that we use are:

Fact 1: For every prime $p > 2$, each quadratic residue mod p has exactly two square roots mod p (and they sum-up to p).[3]

Fact 2: For every (odd and non-integer-power) composite number N, each quadratic residue mod N has at least four square roots mod N.

Our algorithm uses as a black-box an algorithm, denoted `sqrt`, that given a prime p and a quadratic residue mod p, denoted s, returns the smallest among the two modular square roots of s. There is no guarantee as to what the output is in the case that the input is not of the aforementioned form (and in particular in the case that p is not a prime). Thus, we actually present a probabilistic polynomial-time reduction of testing primality to extracting square roots modulo a prime (which is a search problem with a promise; see Section 2.4.1).

Construction 6.4 (the reduction): *On input a natural number $N > 2$ do*

1. *If N is either even or an integer-power*[4] *then* `reject`.
2. *Uniformly select $r \in \{1, ..., N-1\}$, and set $s \leftarrow r^2 \bmod N$.*
3. *Let $r' \leftarrow \texttt{sqrt}(s, N)$. If $r' \equiv \pm r \pmod N$ then* `accept` *else* `reject`.

Indeed, in the case that N is composite, the reduction invokes `sqrt` on an illegitimate input (i.e., it makes a query that violates the promise of the problem at the target of the reduction). In such a case, there is not guarantee as to what `sqrt` answers, but actually a bluntly wrong answer only plays in our favor. In general, we will show that if N is composite, then the reduction rejects with probability at least $1/2$, regardless of how `sqrt` answers. We mention that there exists a probabilistic polynomial-time algorithm for implementing `sqrt` (see Exercise 6.15).

[3] That is, for every $r \in \{1, ..., p-1\}$, the equation $x^2 \equiv r^2 \pmod p$ has two solutions modulo p (i.e., r and $p - r$).

[4] This can be checked by scanning all possible powers $e \in \{2, ..., \log_2 N\}$, and (approximately) solving the equation $x^e = N$ for each value of e (i.e., finding the smallest integer i such that $i^e \geq N$). Such a solution can be found by binary search.

Proposition 6.5 *Construction 6.4 constitutes a probabilistic polynomial-time reduction of testing primality to extracting square roots module a prime. Furthermore, if the input is a prime then the reduction always accepts, and otherwise it rejects with probability at least* $1/2$.

We stress that Proposition 6.5 refers to the reduction itself; that is, sqrt is viewed as a ("perfect") oracle that, for every prime P and quadratic residue $s \pmod P$, returns $r < s/2$ such that $r^2 \equiv s \pmod P$. Combining Proposition 6.5 with a probabilistic polynomial-time algorithm that computes sqrt with negligible error probability, we obtain that testing primality is in $\mathcal{BPP}$.

Proof: By Fact 1, on input a prime number N, Construction 6.4 always accepts (because in this case, for every $r \in \{1, ..., N-1\}$, it holds that $\texttt{sqrt}(r^2 \bmod N, N) \in \{r, N-r\}$). On the other hand, suppose that N is an odd composite that is not an integer-power. Then, by Fact 2, each quadratic residue s has at least four square roots, and each of these square roots is equally likely to be chosen at Step 2 (in other words, s yields no information regarding which of its modular square roots was selected in Step 2). Thus, for every such s, the probability that either $\texttt{sqrt}(s, N)$ or $N - \texttt{sqrt}(s, N)$ equal the root chosen in Step 2 is at most $2/4$. It follows that, on input a composite number, the reduction rejects with probability at least $1/2$. ■

Reflection: Construction 6.4 illustrates an interesting aspect of randomized algorithms (or rather reductions); that is, the ability to hide information from a subroutine. Specifically, Construction 6.4 generates a problem instance (N, s) without disclosing any additional information. Furthermore, a correct solution to this instance is likely to help the reduction; that is, a correct answer to the instance (N, s) provides probabilistic evidence regarding whether N is a prime, where the probability space refers to the missing information (regarding how s was generated).

Comment. Testing primality is actually in $\mathcal{P}$, however, the deterministic algorithm demonstrating this fact is more complex (and its analysis is even more complicated).

6.1.2 One-sided error: The complexity classes RP and coRP

In this section we consider notions of probabilistic polynomial-time algorithms having one-sided error. The notion of one-sided error refers to a natural partition of the set of instances; that is, yes-instances versus no-instances in the case of decision problems, and instances having solution versus instances having no solution in the case of search problems. We focus on decision problems, and comment that an analogous treatment can be provided for search problems (see the second paragraph of Section 6.1.1).

Definition 6.6 (the class $\mathcal{RP}$)[5]: *A decision problem S is in $\mathcal{RP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] = 1$.

The choice of the constant $1/2$ is immaterial, and any other constant greater than zero will do (and yields the very same class). Similarly, this constant can be replaced by $1-\mu(|x|)$ for various negligible functions μ (while preserving the class). Both facts are special cases of the robustness of the class (see Exercise 6.5).

Observe that $\mathcal{RP} \subseteq \mathcal{NP}$ (see Exercise 6.8) and that $\mathcal{RP} \subseteq \mathcal{BPP}$ (by the aforementioned error-reduction). Defining $\text{co}\mathcal{RP} = \{\{0,1\}^* \setminus S : S \in \mathcal{RP}\}$, note that $\text{co}\mathcal{RP}$ corresponds to the opposite direction of one-sided error probability. That is, *a decision problem S is in $\text{co}\mathcal{RP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] = 1$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] \geq 1/2$.

6.1.2.1 Testing polynomial identity

An appealing example of a one-sided error randomized algorithm refers to the problem of determining whether two polynomials are identical. For simplicity, we assume that we are given an oracle for the evaluation of each of the two polynomials. An alternative presentation that refers to polynomials that are represented by arithmetic circuits (cf. Appendix B.3) yields a standard decision problem in $\text{co}\mathcal{RP}$ (see Exercise 6.16). Either way, we refer to multi-variant polynomials and to the question of whether they are identical over any field (or, equivalently, whether they are identical over a sufficiently large finite field). Note that it suffices to consider finite fields that are larger than the degree of the two polynomials.

Construction 6.7 (Polynomial-Identity Test): *Let n be an integer and F be a finite field. Given black-box access to $p, q : \mathrm{F}^n \to \mathrm{F}$, uniformly select $r_1, ..., r_n \in \mathrm{F}$, and accept if and only if $p(r_1, ..., r_n) = q(r_1, ..., r_n)$.*

Clearly, if $p \equiv q$ then the algorithm always accepts. The following lemma implies that if p and q are different polynomials, each of total degree at most d over the finite field F, then the foregoing procedure accepts with probability at most $d/|\mathrm{F}|$.

Lemma 6.8 *Let $p : \mathrm{F}^n \to \mathrm{F}$ be a non-zero polynomial of total degree d over the finite field F. Then*

$$\Pr_{r_1,...,r_n \in \mathrm{F}}[p(r_1, ..., r_n) = 0] \leq \frac{d}{|\mathrm{F}|}.$$

Proof: The lemma is proven by induction on n. The base case of $n = 1$ follows immediately by the Fundamental Theorem of Algebra (i.e., the number of distinct

[5] The initials RP stands for Random Polynomial-time, which fails to convey the restricted type of error allowed in this class. The only nice feature of this notation is that it is reminiscent of NP, thus reflecting the fact that $\mathcal{RP}$ is a randomized polynomial-time class that is contained in $\mathcal{NP}$.

roots of a degree d univariant polynomial is at most d). In the induction step, we write p as a polynomial in its first variable. That is,

$$p(x_1, x_2, ..., x_n) = \sum_{i=0}^{d} p_i(x_2, ..., x_n) \cdot x_1^i$$

where p_i is a polynomial of total degree at most $d-i$. Let i be the largest integer for which p_i is not identically zero. Dismissing the case $i = 0$ and using the induction hypothesis, we have

$$\begin{aligned} & \mathsf{Pr}_{r_1,r_2,...,r_n}[p(r_1, r_2, ..., r_n) = 0] \\ \leq \;& \mathsf{Pr}_{r_2,...,r_n}[p_i(r_2, ..., r_n) = 0] \\ & + \mathsf{Pr}_{r_1,r_2,...,r_n}[p(r_1, r_2, ..., r_n) = 0 \,|\, p_i(r_2, ..., r_n) \neq 0] \\ \leq \;& \frac{d-i}{|\mathrm{F}|} + \frac{i}{|\mathrm{F}|} \end{aligned}$$

where the second term is bounded by fixing any sequence $r_2, ..., r_n$ for which $p_i(r_2, ..., r_n) \neq 0$ and considering the univariant polynomial $p'(x) \stackrel{\rm def}{=} p(x, r_2, ..., r_n)$ (which by hypothesis is a non-zero polynomial of degree i). ■

6.1.2.2 Relating BPP to RP

A natural question regarding probabilistic polynomial-time algorithms refers to the relation between two-sided and one-sided error probability. For example, *is $\mathcal{BPP}$ contained in $\mathcal{RP}$?* Loosely speaking, we show that $\mathcal{BPP}$ is reducible to co$\mathcal{RP}$ by *one-sided error* randomized Karp-reductions, where the actual statement refers to the promise problem versions of both classes (briefly defined in the following paragraph). Note that $\mathcal{BPP}$ is trivially reducible to co$\mathcal{RP}$ by *two-sided error* randomized Karp-reductions, whereas a deterministic reduction of $\mathcal{BPP}$ to co$\mathcal{RP}$ would imply $\mathcal{BPP} = \text{co}\mathcal{RP} = \mathcal{RP}$ (see Exercise 6.9).

First, we refer the reader to the general discussion of promise problems in Section 2.4.1. Analogously to Definition 2.30, we say that the promise problem $\Pi = (S_{\rm yes}, S_{\rm no})$ is in (the promise problem extension of) $\mathcal{BPP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S_{\rm yes}$ it holds that* $\mathsf{Pr}[A(x)=1] \geq 2/3$ *and for every $x \in S_{\rm no}$ it holds that* $\mathsf{Pr}[A(x)=0] \geq 2/3$. Similarly, Π is in co$\mathcal{RP}$ if for every $x \in S_{\rm yes}$ it holds that $\mathsf{Pr}[A(x) = 1] = 1$ and for every $x \in S_{\rm no}$ it holds that $\mathsf{Pr}[A(x)=0] \geq 1/2$. Probabilistic reductions among promise problems are defined by adapting the conventions of Section 2.4.1; specifically, queries that violate the promise at the target of the reduction may be answered arbitrarily.

Theorem 6.9 *Any problem in $\mathcal{BPP}$ is reducible by a one-sided error randomized Karp-reduction to* co$\mathcal{RP}$*, where* co$\mathcal{RP}$ (and possibly also $\mathcal{BPP}$) denotes the corresponding class of promise problems. *Specifically, the reduction always maps a* no-*instance to a* no-*instance.*

It follows that $\mathcal{BPP}$ is reducible by a one-sided error randomized Cook-reduction to $\mathcal{RP}$. Thus, using the conventions of Section 3.2.2 and referring to classes of promise problems, we may write $\mathcal{BPP} \subseteq \mathcal{RP}^{\mathcal{RP}}$. In fact, since $\mathcal{RP}^{\mathcal{RP}} \subseteq \mathcal{BPP}^{\mathcal{BPP}} = \mathcal{BPP}$, we have $\mathcal{BPP} = \mathcal{RP}^{\mathcal{RP}}$. Theorem 6.9 may be paraphrased as saying that the combination of the one-sided error probability of the reduction and the one-sided error probability of co$\mathcal{RP}$ can account for the two-sided error probability of $\mathcal{BPP}$. We warn that this statement is not a triviality like $1+1=2$, and in particular we do not know whether it holds for classes of standard decision problems (rather than for the classes of promise problems considered in Theorem 6.9).

Proof: Recall that we can easily reduce the error probability of BPP-algorithms, and derive probabilistic polynomial-time algorithms of exponentially vanishing error probability. But this does not eliminate the error (even on "one side") altogether. In general, there seems to be no hope to eliminate the error, unless we (either do something earth-shaking or) change the setting as done when allowing a one-sided error randomized reduction to a problem in co$\mathcal{RP}$. The latter setting can be viewed as a two-move randomized game (i.e., a random move by the reduction followed by a random move by the decision procedure of co$\mathcal{RP}$), and it enables applying different quantifiers to the two moves (i.e., allowing error in one direction in the first quantifier and error in the other direction in the second quantifier). In the next paragraph, which is inessential to the actual proof, we illustrate the potential power of this setting.

Teaching note: The following illustration represents an alternative way of proving Theorem 6.9. This way seems conceptual simpler but it requires a starting point (or rather an assumption) that is much harder to establish, where both comparisons are with respect to the actual proof of Theorem 6.9 (which follows the illustration).

An illustration. Suppose that for some set $S \in \mathcal{BPP}$ there exists a polynomial p' and an off-line BPP-algorithm A' such that for every x it holds that $\Pr_{r\in\{0,1\}^{2p'(|x|)}}[A'(x,r) \neq \chi_S(x)] < 2^{-(p'(|x|)+1)}$; that is, the algorithm uses $2p'(|x|)$ bits of randomness and has error probability smaller than $2^{-p'(|x|)}/2$. Note that such an algorithm cannot be obtained by standard error-reduction (see Exercise 6.10). Anyhow, such a small error probability allows a partition of the string r such that one part accounts for the entire error probability on yes-instances while the other part accounts for the error probability on no-instances. Specifically, for every $x \in S$, it holds that $\Pr_{r'\in\{0,1\}^{p'(|x|)}}[(\forall r'' \in \{0,1\}^{p'(|x|)})\, A'(x,r'r'')=1] > 1/2$, whereas for every $x \notin S$ and every $r' \in \{0,1\}^{p'(|x|)}$ it holds that $\Pr_{r''\in\{0,1\}^{p'(|x|)}}[A'(x,r'r'')=1] < 1/2$. Thus, the error on yes-instances is "pushed" to the selection of r', whereas the error on no-instances is pushed to the selection of r''. This yields a one-sided error randomized Karp-reduction that maps x to (x,r'), where r' is uniformly selected in $\{0,1\}^{p'(|x|)}$, such that deciding S is reduced to the coRP problem (regarding pairs (x,r')) that is decided by the (on-line) randomized algorithm A'' defined by $A''(x,r') \stackrel{\rm def}{=} A'(x,r'U_{p'(|x|)})$. For details, see Exercise 6.11. The actual proof, which avoids the aforementioned hypothesis, follows.

The actual starting point. Consider any BPP-problem with a characteristic function χ (which, in case of a promise problem, is a partial function, defined only over the promise). By standard error-reduction, there exists a probabilistic polynomial-time algorithm A such that for every x on which χ is defined it holds that $\Pr[A(x) \neq \chi(x)] < \mu(|x|)$, where μ is a negligible function. Looking at the corresponding off-line algorithm A' and denoting by p the polynomial that bounds the running time of A, we have

$$\Pr_{r \in \{0,1\}^{p(|x|)}}[A'(x,r) \neq \chi(x)] \;<\; \mu(|x|) \;<\; \frac{1}{2p(|x|)} \tag{6.1}$$

for all sufficiently long x's on which χ is defined. We show a randomized one-sided error Karp-reduction of χ to a promise problem in co$\mathcal{RP}$.

The main idea. As in the illustrating paragraph, the basic idea is "pushing" the error probability on yes-instances (of χ) to the reduction, while pushing the error probability on no-instances to the coRP-problem. Focusing on the case that $\chi(x) = 1$, this is achieved by augmenting the input x with a random sequence of "modifiers" that act on the random-input of algorithm A' such that for a good choice of modifiers it holds that for every $r \in \{0,1\}^{p(|x|)}$ there exists a modifier in this sequence that when applied to r yields r' that satisfies $A'(x,r') = 1$. Indeed, not all sequences of modifiers are good, but a random sequence will be good with high probability and bad sequences will be accounted for in the error probability of the reduction. On the other hand, using only modifiers that are permutations guarantees that the error probability on no-instances only increase by a factor that equals the number of modifiers we use, and this error probability will be accounted for by the error probability of the coRP-problem. Details follow.

The aforementioned modifiers are implemented by shifts (of the set of all strings by fixed offsets). Thus, we augment the input x with a random sequence of shifts, denoted $s_1, ..., s_m \in \{0,1\}^{p(|x|)}$, such that for a good choice of $(s_1, ..., s_m)$ it holds that for every $r \in \{0,1\}^{p(|x|)}$ there exists an $i \in [m]$ such that $A'(x, r \oplus s_i) = 1$. We will show that, for any yes-instance x and a suitable choice of m, with very high probability, a random sequence of shifts is good. Thus, for $A''(\langle x, s_1, ..., s_m \rangle, r) \stackrel{\text{def}}{=} \vee_{i=1}^{m} A'(x, r \oplus s_i)$, it holds that, with very high probability over the choice of $s_1, ..., s_m$, a yes-instance x is mapped to an augmented input $\langle x, s_1, ..., s_m \rangle$ that is accepted by A'' with probability 1. On the other hand, the acceptance probability of augmented no-instances (for any choice of shifts) only increases by a factor of m. In further detailing the foregoing idea, we start by explicitly stating the simple randomized mapping (to be used as a randomized Karp-reduction), and next define the target promise problem.

The randomized mapping. On input $x \in \{0,1\}^n$, we set $m = p(|x|)$, uniformly select $s_1, ..., s_m \in \{0,1\}^m$, and output the pair $(x, \overline{s})$, where $\overline{s} = (s_1, ..., s_m)$. Note that this mapping, denoted M, is easily computable by a probabilistic polynomial-time algorithm.

The promise problem. We define the following promise problem, denoted $\Pi = (\Pi_{\rm yes}, \Pi_{\rm no})$, having instances of the form $(x, \overline{s})$ such that $|\overline{s}| = p(|x|)^2$.

- The yes-instances are pairs $(x, \overline{s})$, where $\overline{s} = (s_1, ..., s_m)$ and $m = p(|x|)$, such that for every $r \in \{0,1\}^m$ there exists an i satisfying $A'(x, r \oplus s_i) = 1$.

- The no-instances are pairs $(x, \overline{s})$, where again $\overline{s} = (s_1, ..., s_m)$ and $m = p(|x|)$, such that for at least half of the possible $r \in \{0,1\}^m$, for every i it holds that $A'(x, r \oplus s_i) = 0$.

To see that Π is indeed a co$\mathcal{RP}$ promise problem, we consider the following randomized algorithm. On input $(x, (s_1, ..., s_m))$, where $m = p(|x|) = |s_1| = \cdots = |s_m|$, the algorithm uniformly selects $r \in \{0,1\}^m$, and accepts if and only if $A'(x, r \oplus s_i) = 1$ for some $i \in \{1, ..., m\}$. Indeed, yes-instances of Π are accepted with probability 1, whereas no-instances of Π are rejected with probability at least $1/2$.

Analyzing the reduction: We claim that the randomized mapping M reduces χ to Π with one-sided error. Specifically, we will prove two claims.

Claim 1: If x is a yes-instance (i.e., $\chi(x) = 1$) then $\mathsf{Pr}[M(x) \in \Pi_{\rm yes}] > 1/2$.

Claim 2: If x is a no-instance (i.e., $\chi(x) = 0$) then $\mathsf{Pr}[M(x) \in \Pi_{\rm no}] = 1$.

We start with Claim 2, which is easier to establish. Recall that $M(x) = (x, (s_1, ..., s_m))$, where $s_1, ..., s_m$ are uniformly and independently distributed in $\{0,1\}^m$. We note that (by Eq. (6.1) and $\chi(x) = 0$), for every possible choice of $s_1, ..., s_m \in \{0,1\}^m$ and every $i \in \{1, ..., m\}$, the fraction of r's that satisfy $A'(x, r \oplus s_i) = 1$ is at most $\frac{1}{2m}$. Thus, for every possible choice of $s_1, ..., s_m \in \{0,1\}^m$, for at least half of the possible $r \in \{0,1\}^m$ there exists an i such that $A'(x, r \oplus s_i) = 1$ holds. Hence, the reduction M *always* maps the no-instance x (i.e., $\chi(x) = 0$) to a no-instance of Π (i.e., an element of $\Pi_{\rm no}$).

Turning to Claim 1 (which refers to $\chi(x) = 1$), we will show shortly that in this case, with very high probability, the reduction M maps x to a yes-instance of Π. We upper-bound the probability that the reduction fails (in case $\chi(x) = 1$) as follows:

$$\begin{aligned}
\mathsf{Pr}[M(x) \notin \Pi_{\rm yes}] &= \mathsf{Pr}_{s_1,...,s_m}[\exists r \in \{0,1\}^m \text{ s.t. } (\forall i)\ A'(x, r \oplus s_i) = 0] \\
&\leq \sum_{r \in \{0,1\}^m} \mathsf{Pr}_{s_1,...,s_m}[(\forall i)\ A'(x, r \oplus s_i) = 0] \\
&= \sum_{r \in \{0,1\}^m} \prod_{i=1}^{m} \mathsf{Pr}_{s_i}[A'(x, r \oplus s_i) = 0] \\
&< 2^m \cdot \left(\frac{1}{2m}\right)^m
\end{aligned}$$

where the last inequality is due to Eq. (6.1). It follows that if $\chi(x) = 1$ then $\mathsf{Pr}[M(x) \in \Pi_{\rm yes}] \gg 1/2$. Thus, the randomized mapping M reduces χ to Π, with one-sided error on yes-instances. Recalling that $\Pi \in$ co$\mathcal{RP}$, the theorem follows. $\blacksquare$

BPP is in PH. The traditional presentation of the ideas underlying the proof of Theorem 6.9 uses them for showing that $\mathcal{BPP}$ *is in the Polynomial-time Hierarchy* (where both classes refer to standard decision problems). Specifically, to prove that $\mathcal{BPP} \subseteq \Sigma_2$ (see Definition 3.8), define the polynomial-time computable predicate $\varphi(x, \overline{s}, r) \stackrel{\text{def}}{=} \bigvee_{i=1}^{m}(A'(x, s_i \oplus r) = 1)$, and observe that

$$\chi(x) = 1 \quad \Rightarrow \quad \exists \overline{s} \, \forall r \;\; \varphi(x, \overline{s}, r) \tag{6.2}$$

$$\chi(x) = 0 \quad \Rightarrow \quad \forall \overline{s} \, \exists r \;\; \neg\varphi(x, \overline{s}, r) \tag{6.3}$$

(where Eq. (6.3) is equivalent to $\neg\exists \overline{s} \, \forall r \;\; \varphi(x, \overline{s}, r)$). Note that Claim 1 (in the proof of Theorem 6.9) establishes that *most* sequences $\overline{s}$ satisfy $\forall r \, \varphi(x, \overline{s}, r)$, whereas Eq. (6.2) only requires the existence of *at least one* such $\overline{s}$. Similarly, Claim 2 establishes that for every $\overline{s}$ *most* choices of r violate $\varphi(x, \overline{s}, r)$, whereas Eq. (6.3) only requires that for every $\overline{s}$ there exists *at least one* such r. We comment that the same proof idea yields a variety of similar statements (e.g., $\mathcal{BPP} \subseteq \mathcal{MA}$, where $\mathcal{MA}$ is a randomized version of $\mathcal{NP}$ defined in Section 9.1).[6]

6.1.3 Zero-sided error: The complexity class ZPP

We now consider probabilistic polynomial-time algorithms that never err, but may fail to provide an answer. Focusing on decision problems, the corresponding class is denoted $\mathcal{ZPP}$ (standing for Zero-error Probabilistic Polynomial-time). The standard definition of $\mathcal{ZPP}$ is in terms of machines that output $\perp$ (indicating failure) with probability at most 1/2. That is, $S \in \mathcal{ZPP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ it holds that* $\Pr[A(x) \in \{\chi_S(x), \perp\}] = 1$ *and* $\Pr[A(x) = \chi_S(x)] \geq 1/2$, *where* $\chi_S(x) = 1$ *if* $x \in S$ *and* $\chi_S(x) = 0$ *otherwise.* Again, the choice of the constant (i.e., 1/2) is immaterial, and "error-reduction" can be performed showing that algorithms that yield a meaningful answer with noticeable probability can be amplified to algorithms that fail with negligible probability (see Exercise 6.6).

Theorem 6.10 $\mathcal{ZPP} = \mathcal{RP} \cap \text{co}\mathcal{RP}$.

Proof Sketch: The fact that $\mathcal{ZPP} \subseteq \mathcal{RP}$ (as well as $\mathcal{ZPP} \subseteq \text{co}\mathcal{RP}$) follows by a trivial transformation of the ZPP-algorithm; that is, replacing the failure indicator $\perp$ by a "no" verdict (resp., "yes" verdict). Note that the choice of what to say in case the ZPP-algorithm fails is determined by the type of error that we are allowed.

In order to prove that $\mathcal{RP} \cap \text{co}\mathcal{RP} \subseteq \mathcal{ZPP}$ we combine the two algorithm guaranteed for a set in $\mathcal{RP} \cap \text{co}\mathcal{RP}$. The point is that we can trust the RP-algorithm (resp., coNP-algorithm) in the case that it says "yes" (resp., "no"), but not in the case that it says "no" (resp., "yes"). Thus, we invoke both algorithms,

[6] Specifically, the class $\mathcal{MA}$ is defined by allowing the verification algorithm V in Definition 2.5 to be probabilistic and err on no-instances; that is, for every $x \in S$ there exists $y \in \{0,1\}^{\text{poly}(|x|)}$ such that $\Pr[V(x, y) = 1] = 1$, whereas for every $x \notin S$ and every y it holds that $\Pr[V(x, y) = 0] \geq 1/2$. We note that $\mathcal{MA}$ can be viewed as a hybrid of the two aforementioned pairs of conditions; specifically, each problem in $\mathcal{MA}$ satisfy the conjunction of Eq. (6.2) and Claim 2. Other randomized versions of $\mathcal{NP}$ (i.e., variants of $\mathcal{MA}$) are considered in Exercise 6.12.

and output a definite answer only if we obtain an answer that we can trust (which happen with high probability). Otherwise, we output $\perp$. □

Expected polynomial-time. In some sources $\mathcal{ZPP}$ is defined in terms of randomized algorithms that run in expected polynomial-time and always output the correct answer. This definition is equivalent to the one we used (see Exercise 6.7).

6.1.4 Randomized Log-Space

In this section we discuss probabilistic polynomial-time algorithms that are further restricted such that they are allowed to use only a logarithmic amount of space.

6.1.4.1 Definitional issues

When defining space-bounded randomized algorithms, we face a problem analogous to the one discussed in the context of non-deterministic space-bounded computation (see Section 5.3). Specifically, the on-line and the off-line versions (formulated in Definition 6.1) are no longer equivalent, unless we restrict the off-line machine to access its random-input tape in a uni-directional manner. The issue is that, in the context of space-bounded computation (and unlike in the case that we only care about time-bounds), the outcome of the internal coin tosses (in the on-line model) cannot be recorded for free. Bearing in mind that, *in the current context*, we wish to model real algorithms (rather than present a fictitious model that captures a fundamental phenomena as in Section 5.3), it is clear that *using the on-line version is the natural choice.*

An additional issue that arises is the need to explicitly bound the running-time of space-bounded randomized algorithms. Recall that, without loss of generality, the number of steps taken by a space-bounded non-deterministic machine is at most exponential in its space complexity, because the shortest path between two configurations in the (directed) graph of possible configurations is upper-bounded by its size (which in turn is exponential in the space-bound). This reasoning fails in the case of randomized algorithms, because the shortest path between two configurations does not bound the expected number of random steps required for going from the first configuration to the second one. In fact, as we shall shortly see, failing to upper-bound the running time of log-space randomized algorithms seems to allow them too much power; that is, such (unrestricted) log-space randomized algorithms can emulate non-deterministic log-space computations (in exponential time). The emulation consists of repeatedly invoking the NL-machine, while using random choices in the role of the non-deterministic moves. If the input is a yes-instance then, in each attempt, with probability at least 2^{-t}, we "hit" an accepting t-step (non-deterministic) computation, where t is polynomial in the input length. Thus, the randomized machine accepts such a yes-instance after an expected number of 2^t trials. To allow for the rejection of no-instances (rather than looping infinitely in vain), we wish to implement a counter that counts till 2^t (or so) and

reject the input if this number of trials have failed. We need to implement such a counter within space $O(\log t)$ rather than t (which is easy). In fact, it suffices to have a "randomized counter" that, with high probability, counts to approximately 2^t. The implementation of such a counter is left to Exercise 6.17, and using it we may obtain a randomized algorithm that halts with high probability (on every input), always rejects a no-instance, and accepts each yes-instance with probability at least 1/2.

In light of the foregoing discussion, when defining randomized log-space algorithms we explicitly require that the algorithms halt in polynomial-time. Modulo this convention, the class $\mathcal{RL}$ (resp., $\mathcal{BPL}$) relates to $\mathcal{NL}$ analogously to the relation of $\mathcal{RP}$ (resp., $\mathcal{BPP}$) to $\mathcal{NP}$. Specificially, the probabilistic acceptance condition of $\mathcal{RL}$ (resp., $\mathcal{BPL}$) is as in the case of $\mathcal{RP}$ (resp., $\mathcal{BPP}$).

Definition 6.11 (the classes $\mathcal{RL}$ and $\mathcal{BPL}$): *We say that a randomized log-space algorithm is* admissible *if it always halts in a polynomial number of steps.*

- *A decision problem S is in $\mathcal{RL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] = 1$.
- *A decision problem S is in $\mathcal{BPL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

Clearly, $\mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{P}$ and $\mathcal{BPL} \subseteq \mathcal{P}$. Note that the classes $\mathcal{RL}$ and $\mathcal{BPL}$ remain unchanged even if we allow the algorithms to run for *expected* polynomial-time and have non-halting computations. Such algorithms can be easily transformed into admissible algorithms by truncating long computations, while using a (standard) counter (which can be implemented in logarithmic-space). Also note that error-reduction is applicable in the current setting (while essentially preserving both the time and space bounds).

6.1.4.2 The accidental tourist sees it all

An appealing example of a randomized log-space algorithm is presented next. It refers to the problem of deciding undirected connectivity, and demonstrated that this problem is in $\mathcal{RL}$. (Recall that in Section 5.2.4 we proved that this problem is actually in $\mathcal{L}$, but the algorithm and its analysis were more complicated.) Recall that Directed Connectivity is complete for $\mathcal{NL}$ (under log-space reductions). For sake of simplicity, we consider the following version of undirected connectivity, which is equivalent under log-space reductions to the version in which one needs to determine whether or not the input (undirected) graph is connected. In the current version, *the input consists of a triple (G, s, t), where G is an undirected graph, s, t are two vertices in G, and the task is to determine whether or not s and t are connected in G.*

Construction 6.12 *On input (G, s, t), the randomized algorithm starts a* poly$(|G|)$-*long random walk at vertex s, and accepts the triplet if and only if the walk passed*

through the vertex t. By a random walk we mean that at each step the algorithm selects uniformly one of the neighbors of the current vertex and moves to it.

Observe that the algorithm can be implemented in logarithmic space (because we only need to store the current vertex as well as the number of steps taken so far). Obviously, if s and t are not connected in G then the algorithm always rejects (G, s, t). Proposition 6.13 implies that undirected connectivity is indeed in $\mathcal{RL}$.

Proposition 6.13 *If s and t are connected in $G = (V, E)$ then a random walk of length $O(|V| \cdot |E|)$ starting at s passes through t with probability at least* $1/2$.

In other words, a random walk starting at s visits all vertices of the connected component of s (i.e., it sees all that there is to see).

Proof Sketch: We will actually show that if G is connected then, with probability at least $1/2$, a random walk starting at s visits all the vertices of G. For any pair of vertices (u, v), let $X_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is *first encountered*. The reader may verify that for every edge $\{u, v\} \in E$ it holds that $\mathrm{E}[X_{u,v}] \leq 2|E|$; see Exercise 6.18. Next, we let $\mathrm{cover}(G)$ denote the expected number of steps in a random walk starting at s and ending when the last of the vertices of V is encountered. Our goal is to upper-bound $\mathrm{cover}(G)$. Towards this end, we consider an arbitrary directed cyclic-tour C that visits all vertices in G, and note that

$$\mathrm{cover}(G) \;\leq\; \sum_{(u,v) \in C} \mathrm{E}[X_{u,v}] \;\leq\; |C| \cdot 2|E|.$$

In particular, selecting C as a traversal of some spanning tree of G, we conclude that $\mathrm{cover}(G) \;<\; 4 \cdot |V| \cdot |E|$. Thus, with probability at least $1/2$, a random walk of length $8 \cdot |V| \cdot |E|$ starting at s visits all vertices of G. □

6.2 Counting

We now turn to a new type of computational problems, which vastly generalize decision problems of the NP-type. We refer to counting problems, and more specifically to counting objects that can be efficiently recognized. The search and decision versions of NP provide suitable definitions of efficiently recognized objects, which in turn yield corresponding counting problems:

1. For each search problem having efficiently checkable solutions (i.e., a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ in $\mathcal{PC}$ (see Definition 2.3)), we consider the problem of counting the number of solutions for a given instance. That is, on input x, we are required to output $|\{y : (x, y) \in R\}|$.

2. For each decision problem S in $\mathcal{NP}$, and each corresponding verification procedure V (as in Definition 2.5), we consider the problem of counting the number of NP-witnesses for a given instance. That is, on input x, we are required to output $|\{y : V(x, y) = 1\}|$.

We shall consider these types of counting problems as well as relaxations (of these counting problems) that refer to approximating the said quantities (see Sections 6.2.1 and 6.2.2, respectively). Other related topics include "problems with unique solutions" (see Section 6.2.3) and "uniform generation of solutions" (see Section 6.2.4). Interestingly, randomized procedures will play an important role in the results regarding the aforementioned types of problems.

6.2.1 Exact Counting

In continuation to the foregoing discussion, we define the class of problems concerned with counting efficiently recognized objects. (Recall that $\mathcal{PC}$ denotes the class of search problems having polynomially long solutions that are efficiently checkable; see Definition 2.3.)

Definition 6.14 (counting efficiently recognized objects – $\#\mathcal{P}$): *The class $\#\mathcal{P}$ consists of all functions that count solutions to a search problem in $\mathcal{PC}$. That is, $f:\{0,1\}^* \to \mathbb{N}$ is in $\#\mathcal{P}$ if there exists $R \in \mathcal{PC}$ such that, for every x, it holds that $f(x) = |R(x)|$, where $R(x) = \{y : (x,y) \in R\}$. In this case we say that f is the* counting problem associated with *R, and denote the latter by $\#R$* (i.e., $\#R = f$).

Every decision problem in $\mathcal{NP}$ is Cook-reducible to $\#\mathcal{P}$, because every such problem can be cast as deciding membership in $S_R = \{x : |R(x)| > 0\}$ for some $R \in \mathcal{PC}$ (see Section 2.1.2). It also holds that $\mathcal{BPP}$ is Cook-reducible to $\#\mathcal{P}$ (see Exercise 6.19). The class $\#\mathcal{P}$ is sometimes defined in terms of decision problems, as is implicit in the following proposition.

Proposition 6.15 (a decisional version of $\#\mathcal{P}$): *For any $f \in \#\mathcal{P}$, deciding membership in $S_f \stackrel{\rm def}{=} \{(x,N) : f(x) \geq N\}$ is computationally equivalent to computing f.*

Actually, the claim holds for any function $f:\{0,1\}^* \to \mathbb{N}$ for which there exists a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $f(x) \leq 2^{p(|x|)}$.

Proof: Since the relation R vouching for $f \in \#\mathcal{P}$ (i.e., $f(x) = |R(x)|$) is polynomially bounded, there exists a polynomial p such that for every x it holds that $f(x) \leq 2^{p(|x|)}$. Deciding membership in S_f is easily reduced to computing f (i.e., we accept the input (x,N) if and only if $f(x) \geq N$). Computing f is reducible to deciding S_f by using a binary search (see Exercise 2.9). This relies on the fact that, on input x and oracle access to S_f, we can determine whether or not $f(x) \geq N$ by making the query (x,N). Note that we know a priori that $f(x) \in [0, 2^{p(|x|)}]$. ■

The counting class $\#\mathcal{P}$ is also related to the problem of enumerating all possible solutions to a given instance (see Exercise 6.20).

6.2.1.1 On the power of $\#\mathcal{P}$

As indicated, $\mathcal{NP} \cup \mathcal{BPP}$ is (easily) reducible to $\#\mathcal{P}$. Furthermore, as stated in Theorem 6.16, the entire Polynomial-Time Hierarchy (as defined in Section 3.2) is Cook-reducible to $\#\mathcal{P}$ (i.e., $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$). On the other hand, any problem in $\#\mathcal{P}$ is solvable in polynomial space, and so $\mathcal{P}^{\#\mathcal{P}} \subseteq \mathcal{PSPACE}$.

Theorem 6.16 *Every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$.*

We do not present a proof of Theorem 6.16 here, because the known proofs are rather technical. Furthermore, one main idea underlying these proofs appears in a more clear form in the proof of Theorem 6.29. Nevertheless, in Section F.1 we present a proof of a related result, which implies that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via *randomized* Karp-reductions.

6.2.1.2 Completeness in $\#\mathcal{P}$

The definition of $\#\mathcal{P}$-completeness is analogous to the definition of $\mathcal{NP}$-completeness. That is, *a counting problem f is $\#\mathcal{P}$-complete if $f \in \#\mathcal{P}$ and every problem in $\#\mathcal{P}$ is Cook-reducible to f.*

We claim that the counting problems associated with the NP-complete problems presented in Section 2.3.3 are all $\#\mathcal{P}$-complete. We warn that this fact is not due to the mere NP-completeness of these problems, but rather to an additional property of the reductions establishing their NP-completeness. Specifically, the Karp-reductions that were used (or variants of them) have the extra property of preserving the number of NP-witnesses (as captured by the following definition).

Definition 6.17 (parsimonious reductions): *Let $R, R' \in \mathcal{PC}$ and let g be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$, where $R(x) = \{y : (x,y) \in R\}$ and $R'(x) = \{y : (x,y) \in R'\}$. We say that g is* parsimonious (with respect to R and R') *if for every x it holds that $|R(x)| = |R'(g(x))|$. In such a case we say that g is a* parsimonious reduction of R to R'.

We stress that the condition of being parsimonious refers to the two underlying relations R and R' (and not merely to the sets S_R and $S_{R'}$). The requirement that g is a Karp-reduction is partially redundant, because if g is polynomial-time computable and for every x it holds that $|R(x)| = |R'(g(x))|$, then g constitutes a Karp-reduction of S_R to $S_{R'}$. Specifically, $|R(x)| = |R'(g(x))|$ implies that $|R(x)| > 0$ (i.e., $x \in S_R$) if and only if $|R'(g(x))| > 0$ (i.e., $g(x) \in S_{R'}$). The reader may easily verify that the Karp-reduction underlying the proof of Theorem 2.18 as well as many of the reductions used in Section 2.3.3 are parsimonious (see Exercise 2.29).

Theorem 6.18 *Let $R \in \mathcal{PC}$ and suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R. Then the counting problem associated with R is $\#\mathcal{P}$-complete.*

Proof: Clearly, the counting problem associated with R, denoted $\#R$, is in $\#\mathcal{P}$. To show that every $f' \in \#\mathcal{P}$ is reducible to f, we consider the relation $R' \in \mathcal{PC}$ that is counted by f'; that is, $\#R' = f'$. Then, by the hypothesis, there exists a parsimonious reduction g of R' to R. This reduction also reduces $\#R'$ to $\#R$; specifically, $\#R'(x) = \#R(g(x))$ for every x. ■

Corollaries. As an immediate corollary of Theorem 6.18, we get that counting the number of satisfying assignments to a given CNF formula is $\#\mathcal{P}$-complete. Similar statement hold for all the other NP-complete problems mentioned in Section 2.3.3 and in fact for all NP-complete problems listed in [81]. These corollaries follow from the fact that all known reductions among natural NP-complete problems are either parsimonious or can be easily modified to be so.

We conclude that many counting problems associated with NP-complete search problems are $\#\mathcal{P}$-complete. It turns out that also counting problems associated with efficiently solvable search problems may be $\#\mathcal{P}$-complete.

Theorem 6.19 *There exist $\#\mathcal{P}$-complete counting problems that are associated with efficiently solvable search problems. That is, there exists $R \in \mathcal{PF}$* (see Definition 2.2) *such that $\#R$ is $\#\mathcal{P}$-complete.*

Proof: Consider the relation $R_{\mathtt{dnf}}$ consisting of pairs (ϕ, τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Note that the search problem of $R_{\mathtt{dnf}}$ is easy to solve (e.g., by picking an arbitrary truth assignment that satisfies the first term in the input formula). To see that $\#R_{\mathtt{dnf}}$ is $\#\mathcal{P}$-complete consider the following reduction from $\#R_{\mathtt{SAT}}$ (which is $\#\mathcal{P}$-complete by Theorem 6.18). Given a CNF formula ϕ, transform $\neg\phi$ into a DNF formula ϕ' by applying de-Morgan's Law, and return $2^n - \#R_{\mathtt{dnf}}(\phi')$, where n denotes the number of variables in ϕ (resp., ϕ'). ■

Reflections: We note that Theorem 6.19 is not established by a parsimonious reduction (and refer the reader to more artifical $\#\mathcal{P}$-complete problems presented in Exercise 6.21). This fact should not come as a surprise because a parsimonious reduction of $\#R'$ to $\#R$ implies that $S_{R'} = \{x : \exists y \text{ s.t. } (x,y) \in R'\}$ is reducible to $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$, where in our case $S_{R'}$ is NP-Complete while $S_R \in \mathcal{P}$ (since $R \in \mathcal{PF}$). Nevertheless, the proof of Theorem 6.19 is related to the hardness of some underlying decision problem (i.e., the problem of deciding whether a given DNF formula is a tautology (i.e., whether $\#R_{\mathtt{dnf}}(\phi') = 2^n$)). But does there exist a $\#\mathcal{P}$-complete problem that is "not based on some underlying NP-complete decision problem"? Amazingly enough, the answer is positive.

Theorem 6.20 *Counting the number of perfect matchings in a bipartite graph is $\#\mathcal{P}$-complete.*[7]

Equivalently (see Exercise 6.22), the problem of computing the permanent of matrices with 0/1-entries is $\#\mathcal{P}$-complete. Recall that the permanent of an n-by-n matrix $M = (m_{i,j})$, denoted $\mathtt{perm}(M)$, equals the sum over all permutations π of $[n]$ of the products $\prod_{i=1}^{n} m_{i,\pi(i)}$. Theorem 6.20 is proven by composing the following two (many-to-one) reductions (asserted in Propositions 6.21 and 6.22, respectively) and using the fact that $\#R_{\mathtt{3SAT}}$ is $\#\mathcal{P}$-complete (see Theorem 6.18 and Exercise 2.29). Needless to say, the resulting reduction is not parsimonious.

[7] See Appendix G.1 for basic terminology regarding graphs.

Proposition 6.21 *The counting problem of* 3SAT (i.e., $\#R_{\mathtt{3SAT}}$) *is reducible to computing the permanent of integer matrices. Furthermore, there exists an* even *integer $c > 0$ and a finite set of integers I such that, on input a 3CNF formula ϕ, the reduction produces an integer matrix M_ϕ with entries in I such that* $\mathtt{perm}(M_\phi) = c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$ *where m denotes the number of clauses in ϕ.*

The original proof of Proposition 6.21 uses $c = 2^{10}$ and $I = \{-1, 0, 1, 2, 3\}$. It can be shown (see Exercise 6.23 (which relies on Theorem 6.29)) that, for every integer $n > 1$ that is relatively prime to c, computing the permanent modulo n is NP-hard (under randomized reductions). Thus, using the case of $c = 2^{10}$, this means that computing the permanent modulo n is NP-hard for any odd $n > 1$. In contrast, computing the permanent modulo 2 (which is equivalent to computing the determinant modulo 2) is easy (i.e., can be done in polynomial-time and even in $\mathcal{NC}$). Thus, assuming $\mathcal{NP} \not\subseteq \mathcal{BPP}$, Proposition 6.21 cannot hold for an odd c (because by Exercise 6.23 it would follow that computing the permanent modulo 2 is NP-Hard). We also note that, assuming $\mathcal{P} \neq \mathcal{NP}$, Proposition 6.21 cannot possibly hold for a set I containing only non-negative integers (see Exercise 6.24).

Proposition 6.22 *Computing the permanent of integer matrices is reducible to computing the permanent of 0/1-matrices. Furthermore, the reduction maps any integer matrix A into a 0/1-matrix A'' such that the permanent of A can be easily computed from A and the permanent of A''.*

Teaching note: We do not recommend presenting the proofs of Propositions 6.21 and 6.22 in class. The high-level structure of the proof of Proposition 6.21 has the flavor of some sophisticated reductions among NP-problems, but the crucial point is the existence of adequate gadgets. We do not know of a high-level argument establishing the existence of such gadgets nor of any intuition as to why such gadgets exist.[8] Instead, the existence of such gadgets is proved by a design that is both highly non-trivial and *ad hoc* in nature. Thus, the proof of Proposition 6.21 boils down to a complicated design problem that is solved in a way that has little pedagogical value. In contrast, the proof of Proposition 6.22 uses two simple ideas that can be useful in other settings. With suitable hints, this proof can be used as a good exercise.

Proof of Proposition 6.21: We will use the correspondence between the permanent of a matrix A and the sum of the weights of the cycle covers of the weighted directed graph represented by the matrix A. A cycle cover of a graph is a collection of simple[9] *vertex-disjoint* directed cycles that covers all the graph's vertices, and its weight is the product of the weights of the corresponding edges. The SWCC of a weighted directed graph is the sum of the weights of all its cycle covers.

Given a 3CNF formula ϕ, we construct a directed weighted graph G_ϕ such that the SWCC of G_ϕ equals equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$, where c is a universal constant and m denotes the number of clauses in ϕ. We may assume, without loss of generality, that each clause of ϕ has exactly three literals (which are not necessarily distinct).

[8] Indeed, the conjecture that such gadgets exist can only be attributed to ingenuity.

[9] Here a simple cycle is a strongly connected directed graph in which each vertex has a single incoming (resp., outgoing) edge. In particular, self-loops are allowed.

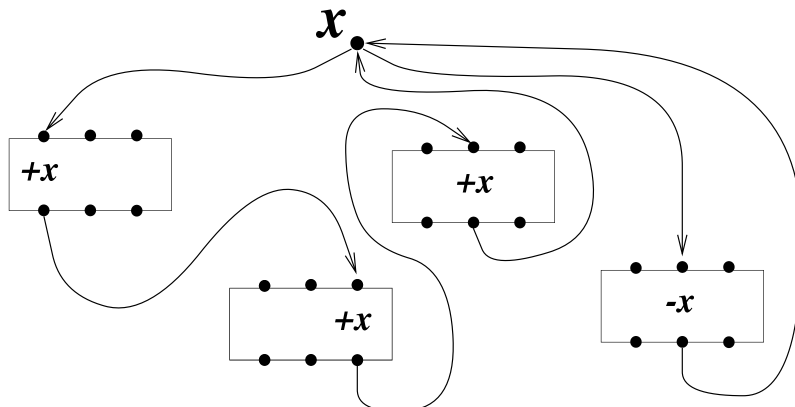
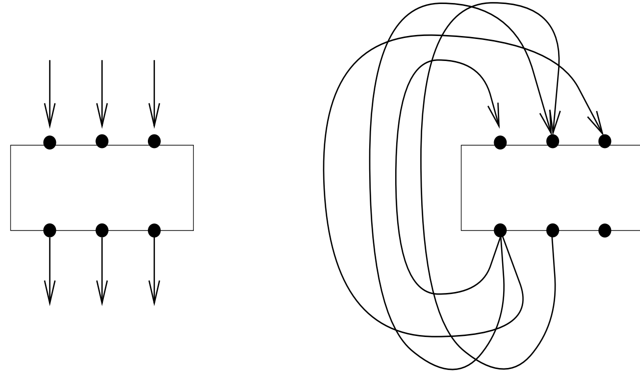


Figure 6.1: Tracks connecting gadgets for the reduction to cycle cover.

We start with a high-level description (of the construction) that refers to (clause) gadgets, each containing some internal vertices and internal (weighted) edges, which are *unspecified at this point*. In addition, each gadget has three pairs of designated vertices, one pair per each literal appearing in the clause, where one vertex in the pair is designated as an entry vertex and the other as an exit vertex. The graph G_ϕ consists of m such gadgets, one per each clause (of ϕ), and n auxiliary vertices, one per each variable (of ϕ), as well as some *additional directed edges*, each having weight 1. Specifically, for each variable, we introduce two tracks, one per each of the possible literals of this variable. The track associated with a literal consists of directed edges (each having weight 1) that form a simple "cycle" passing through the corresponding (auxiliary) vertex as well as through the designated vertices that correspond to the occurrences of this literal in the various clauses. Specifically, for each such occurrence, the track enters the corresponding clause gadget at the entry-vertex corresponding to this literal and exits at the corresponding exit-vertex. (If a literal does not appear in ϕ then the corresponding track is a self-loop on the corresponding variable.) See Figure 6.1 showing the two tracks of a variable x that occurs positively in three clauses and negatively in one clause. The entry-vertices (resp., exit-vertices) are drawn on the top (resp., bottom) part of each gadget.

For the purpose of stating the desired properties of the clause gadget, we augment the gadget by nine external edges (of weight 1), one per each pair of (not necessarily matching) entry and exit vertices such that the edge goes from the exit-vertex to the entry-vertex (see Figure 6.2). (We stress that this is an auxiliary construction that differs from and yet is related to the use of gadgets in the foregoing construction of G_ϕ.) The three edges that link the designated pairs of vertices that correspond to the three literals are called nice. We say that a collection of edges C (e.g., a collection of cycles) uses the external edges S if the intersection of C with the set of the (nine) external edges equals S. We postulate the following three properties of the clause gadget.



On the left is a gadget with the track edges adjacent to it (as in the real construction). On the right is a gadget and four out of the nine external edges (two of which are nice) used in the analysis.

Figure 6.2: External edges for the analysis of the clause gadget

1. The sum of the weights of all cycle covers (of the gadget) that do not use any external edge (i.e., use the empty set of external edges) equals zero.

2. Let $V(S)$ denote the set of vertices incident to S, and say that S is nice if it is non-empty and the vertices in $V(S)$ can be perfectly matched using nice edges.[10] Then, there exists a constant c (indeed the one postulated in the proposition's claim) such that, for any nice set S, the sum of the weights of all cycle covers that use the external edges S equals c.

3. For any non-nice set S of external edges, the sum of the weights of all cycle covers that use the external edges S equals zero.

Note that the foregoing three cases exhaust all the possible ones, and that the set of external edges used by a cycle cover must be a matching (i.e., these edges are vertex disjoint). Using the foregoing conditions, it can be shown that each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ (see Exercise 6.25). It follows that the SWCC of G_ϕ equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$.

Having established the validity of the abstract reduction, we turn to the implementation of the clause gadget. The first implementation is a *Deus ex Machina*, with a corresponding adjacency matrix depicted in Figure 6.3. Its validity (for the value $c = 12$) can be verified by computing the permanent of the corresponding sub-matrices (see analogous analysis in Exercise 6.27).

[10]Clearly, any non-empty set of nice edges is a nice set. Thus, a singleton set is nice if and only if the corresponding edge is nice. On the other hand, any set S of three (vertex-disjoint) external edges is nice, because $V(S)$ has a perfect matching using all three nice edges. Thus, the notion of nice sets is "non-trivial" only for sets of two edges. Such a set S is nice if and only if $V(S)$ consists of two pairs of corresponding designated vertices.

The gadget uses eight vertices, where the first six are the designated (entry and exit) vertices. The entry-vertex (resp., exit-vertex) associated with the $i^{\rm th}$ literal is numbered i (resp., $i+3$). The corresponding adjacency matrix follows.

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 & 0 & 1 & 1 \\ 0 & 0 & -1 & -1 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 2 & -1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Note that the edge $3 \to 6$ can be contracted, but the resulting 7-vertex graph will not be consistent with our (inessentially stringent) definition of a gadget by which the six designated vertices should be distinct.

Figure 6.3: A Deus ex Machina clause gadget for the reduction to cycle cover.

A more structured implementation of the clause gadget is depicted in Figure 6.4, which refers to a (hexagon) box to be implemented later. The box contains several vertices and weighted edges, but only two of these vertices, called terminals, are connected to the outside (and are shown in Figure 6.4). The clause gadget consists of five copies of this box, where three copies are designated for the three literals of the clause (and are marked `LB1`, `LB2`, and `LB3`), as well as additional vertices and edges shown in Figure 6.4. In particular, the clause gadget contains the six aforementioned designated vertices (i.e., a pair of entry and exit vertices per each literal), two additional vertices (shown at the two extremes of the figure), and some edges (all having weight 1). Each designated vertex has a self-loop, and is incident to a single additional edge that is outgoing (resp., incoming) in case the vertex is an entry-vertex (resp., exit-vertex) of the gadget. The two terminals of each box that is associated with some literal are connected to the corresponding pair of designated vertices (e.g., the outgoing edge of `entry1` is incident at the right terminal of the box `LB1`). Note that the five boxes reside on a directed path (going from left to right), and the only edges going in the opposite direction are those drawn below this path.

In continuation to the foregoing, we wish to state the desired properties of the box. Again, we do so by considering the augmentation of the box by external edges (of weight 1) incident at the specified vertices. In this case (see Figure 6.5), we have a pair of anti-parallel edges connecting the two terminals of the box as well as two self-loops (one on each terminal). We postulate the following three properties of the box.

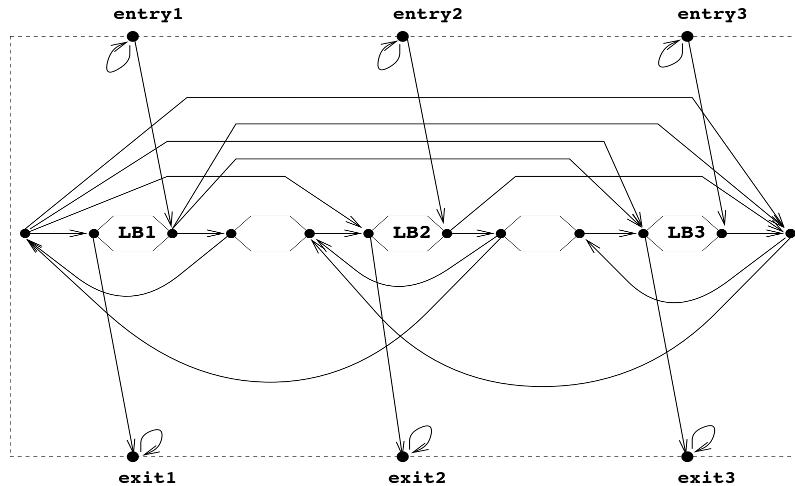
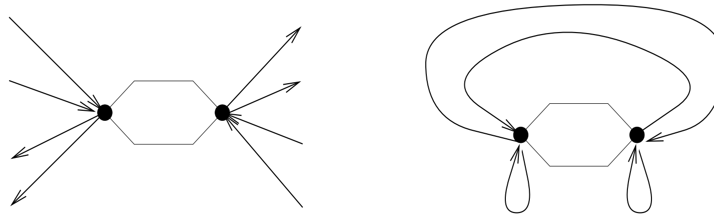


Figure 6.4: A structured clause gadget for the reduction to cycle cover.



On the left is a box with potential edges adjacent to it (as in the gadget construction). On the right is a box and the four external edges used in the analysis.

Figure 6.5: External edges for the analysis of the box

1. The sum of the weights of all cycle covers (of the box) that do not use any external edge equals zero.

2. There exists a constant b (in our case $b = 4$) such that, for each of the two anti-parallel edges, the sum of the weights of all cycle covers that use this edge equals b.

3. For any (non-empty) set S of the self-loops, the sum of the weights of all cycle covers (of the box) that use S equals zero.

Note that the foregoing three cases exhaust all the possible ones. It can be shown that the conditions regarding the box imply that the construction presented in Figure 6.4 satisfies the conditions that were postulated for the clause gadget (see Exercise 6.26). Specifically, we have $c = b^5$. As for box itself, a smaller *Deus ex*

Machina is provided by the following 4-by-4 adjacency matrix

$$\begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix} \tag{6.4}$$

where the two terminals correspond to the first and the fourth vertices. Its validity (for the value $b = 4$) can be verified by computing the permanent of the corresponding sub-matrices (see Exercise 6.27). ■

Proof of Proposition 6.22: The proof proceeds in two steps. In the first step we show that computing the permanent of *integer matrices* is reducible to computing the permanent of *non-negative matrices*. This reduction proceeds as follows. For an n-by-n integer matrix $A = (a_{i,j})_{i,j\in[n]}$, let $\|A\|_\infty = \max_{i,j}(|a_{i,j}|)$ and $Q_A = 2(n!) \cdot \|A\|_\infty^n + 1$. We note that, given A, the value Q_A can be computed in polynomial-time, and in particular $\log_2 Q_A < n^2 \log \|A\|_\infty$. Given the matrix A, the reduction constructs the non-negative matrix $A' = (a_{i,j} \bmod Q_A)_{i,j\in[n]}$ (i.e., the entries of A' are in $\{0, 1, ..., Q_A - 1\}$), queries the oracle for the permanent of A', and outputs $v \stackrel{\text{def}}{=} \texttt{perm}(A') \bmod Q_A$ if $v < Q_A/2$ and $-(Q_A - v)$ otherwise. The key observation is that

$$\texttt{perm}(A) \equiv \texttt{perm}(A') \pmod{Q_A}, \text{ while } |\texttt{perm}(A)| \le (n!) \cdot \|A\|_\infty^n < Q_A/2.$$

Thus, $\texttt{perm}(A') \bmod Q_A$ (which is in $\{0, 1, ..., Q_A - 1\}$) determines $\texttt{perm}(A)$. We note that $\texttt{perm}(A')$ is likely to be much larger than $Q_A > |\texttt{perm}(A)|$; it is merely that $\texttt{perm}(A')$ and $\texttt{perm}(A)$ are equivalent modulo Q_A.

In the second step we show that computing the permanent of non-negative matrices is reducible to computing the permanent of 0/1-matrices. In this reduction, we view the computation of the permanent as the computation of the sum of the weights of the cycle covers (SWCC) of the corresponding weighted directed graph (see proof of Proposition 6.21). Thus, we reduce the computation of the SWCC of directed graphs *with non-negative weights* to the computation of the SWCC of *unweighted directed graphs with no parallel edges* (which correspond to 0/1-matrices). The reduction is via local replacements that preserve the value of the SWCC. These local replacements combine the following two local replacements (which preserve the SWCC):

1. Replacing an edge of weight $w = \prod_{i=1}^t w_i$ by a path of length t (i.e., $t - 1$ internal nodes) with the corresponding weights $w_1, ..., w_t$, and self-loops (with weight 1) on all internal nodes.

   Note that a cycle-cover that uses the original edge corresponds to a cycle-cover that uses the entire path, whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

2. Replacing an edge of weight $w = \sum_{i=1}^t w_i$ by t parallel 2-edge paths such that the first edge on the ith path has weight w_i, the second edge has weight 1,

and the intermediate node has a self-loop (with weight 1). (Paths of length two are used because parallel edges are not allowed.)

Note that a cycle-cover that uses the original edge corresponds to a collection of cycle-covers that use one out of the t paths (and the self-loops of all other intermediate nodes), whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

In particular, writing the positive integer w, having binary expansion $\sigma_{|w|-1}\cdots\sigma_0$, as $\sum_{i:\sigma_i=1}(1+1)^i$, we may apply the additive replacement (for the sum over $\{i : \sigma_i = 1\}$), next the product replacement (for each 2^i), and finally the additive replacement (for $1+1$). Applying this process to the matrix A' obtained in the first step, we *efficiently* obtain a matrix A'' with 0/1-entries such that $\texttt{perm}(A') = \texttt{perm}(A'')$. (In particular, the dimension of A'' is polynomial in the length of the binary representation of A', which in turn is polynomial in the length of the binary representation of A.) Combining the two reductions (steps), the proposition follows. ■

6.2.2 Approximate Counting

Having seen that exact counting (for relations in $\mathcal{PC}$) seems even harder than solving the corresponding search problems, we turn to relaxations of the counting problem. Before focusing on relative approximation, we briefly consider approximation with (large) additive deviation.

Let us consider the counting problem associated with an arbitrary $R \in \mathcal{PC}$. Without loss of generality, we assume that all solutions to n-bit instances have the same length $\ell(n)$, where indeed ℓ is a polynomial. We first note that, while it may be hard to compute $\#R$, given x it is easy to approximate $\#R(x)$ up to an additive error of $0.01 \cdot 2^{\ell(|x|)}$ (by randomly sampling potential solutions for x). Indeed, such an approximation is very rough, but it is not trivial (and in fact we do not know how to obatin it deterministically). In general, we can efficiently produce at random an estimate of $\#R(x)$ that, with high probability, deviates from the correct value by at most an additive term that is related to the absolute upperbound on the number of solutions (i.e., $2^{\ell(|x|)}$).

Proposition 6.23 (approximation with large additive deviation): *Let $R \in \mathcal{PC}$ and ℓ be a polynomial such that $R \subseteq \cup_{n\in\mathbb{N}}\{0,1\}^n \times \{0,1\}^{\ell(n)}$. Then, for every polynomial p, there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$ it holds that*

$$\mathsf{Pr}[|A(x,\delta) - \#R(x)| > (1/p(|x|)) \cdot 2^{\ell(|x|)}] < \delta. \tag{6.5}$$

As usual, δ is presented to A in binary, and hence the running time of $A(x,\delta)$ is upper-bounded by $\mathrm{poly}(|x| \cdot \log(1/\delta))$.

Proof Sketch: On input x and δ, algorithm A sets $t = \Theta(p(|x|)^2 \cdot \log(1/\delta))$, selects uniformly $y_1, ..., y_t$ and outputs $2^{\ell(|x|)} \cdot |\{i : (x, y_i) \in R\}|/t$. □

Discussion. Proposition 6.23 is meaningful in the case that $\#R(x) > (1/p(|x|)) \cdot 2^{\ell(|x|)}$ holds for some x's. But otherwise, a trivial approximation (i.e., outputting the constant value zero) meets the bound of Eq. (6.5). In contrast to this notion of *additive approximation*, a *relative factor approximation* is typically more meaningful. Specifically, we will be interested in approximating $\#R(x)$ up-to a constant factor (or some other reasonable factor). In §6.2.2.1, we consider a natural $\#\mathcal{P}$-complete problem for which such a relative approximation can be obtained in probabilistic polynomial-time. We do not expect this to happen for every counting problem in $\#\mathcal{P}$, because a relative approximation allows for distinguishing instances having no solution from instances that do have solutions (i.e.,, deciding membership in S_R is reducible to a relative approximation of $\#R$). Thus, relative approximation for all $\#\mathcal{P}$ is at least as hard as deciding all problems in $\mathcal{NP}$, but in §6.2.2.2 we show that the former is not harder than the latter; that is, relative approximation for any problem in $\#\mathcal{P}$ can be obtained by a randomized Cook-reduction to $\mathcal{NP}$. Before turning to these results, let us state the underlying definition (and actually strengthen it by requiring approximation to within a factor of $1 \pm \varepsilon$, for $\varepsilon \in (0,1)$).[11]

Definition 6.24 (approximation with relative deviation): *Let* $f : \{0,1\}^* \to \mathbb{N}$ *and* $\varepsilon, \delta : \mathbb{N} \to [0,1]$. *A randomized process* Π *is called an* (ε,δ)-approximator of f *if for every* x *it holds that*

$$\Pr[|\Pi(x) - f(x)| > \varepsilon(|x|) \cdot f(x)] \; < \; \delta(|x|). \tag{6.6}$$

We say that f *is* efficiently $(1-\varepsilon)$-approximable (or just $(1-\varepsilon)$-approximable) *if there exists a probabilistic polynomial-time algorithm* A *that constitute an* $(\varepsilon, 1/3)$-*approximator of* f.

The error probability of the latter algorithm A (which has error probability $1/3$) can be reduced to δ by $O(\log(1/\delta))$ repetitions (see Exercise 6.28). Typically, the running time of A will be polynomial in $1/\varepsilon$, and ε is called the deviation parameter.

6.2.2.1 Relative approximation for $\#R_{\text{dnf}}$

In this subsection we present a natural $\#\mathcal{P}$-complete problem for which constant factor approximation can be found in probabilistic polynomial-time. Stronger results regarding unnatural $\#\mathcal{P}$-complete problems appear in Exercise 6.29.

Consider the relation R_{dnf} consisting of pairs (ϕ, τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Recall that the search problem of R_{dnf} is easy to solve and that the proof of Theorem 6.19 establishes that

[11] We refrain from formally defining an F-factor approximation, for an arbitrary F, although we shall refer to this notion in several informal discussions. There are several ways of defining the aforementioned term (and they are all equivalent when applied to our informal discussions). For example, an F-factor approximation of $\#R$ may mean that, with high probability, the output $A(x)$ satisfies $\#R(x)/F(|x|) \leq A(x) \leq F(|x|) \cdot \#R(x)$. Alternatively, we may require that $\#R(x) \leq A(x) \leq F(|x|) \cdot \#R(x)$ (or, alternatively, that $\#R(x)/F(|x|) \leq A(x) \leq \#R(x)$.

$\#R_{\texttt{dnf}}$ is $\#\mathcal{P}$-complete (via a non-parsimonious reduction). Still there exists a probabilistic polynomial-time algorithm that provides a constant factor approximation of $\#R_{\texttt{dnf}}$. We warn that the fact that $\#R_{\texttt{dnf}}$ is $\#\mathcal{P}$-complete *via a non-parsimonious reduction* means that the constant factor approximation for $\#R_{\texttt{dnf}}$ does not seem to imply a similar approximation for all problems in $\#\mathcal{P}$. In fact, we should not expect each problem in $\#\mathcal{P}$ to have a (probabilistic) polynomial-time constant-factor approximation algorithm because this would imply $\mathcal{NP} \subseteq \mathcal{BPP}$ (since a constant factor approximation allows for distinguishing the case in which the instance has no solution from the case in which the instance has a solution).

The following algorithm is actually a deterministic reduction of the task of $(\varepsilon, 1/3)$-approximating $\#R_{\texttt{dnf}}$ to an (additive deviation) approximation of the type provided in Proposition 6.23. Consider a DNF formula $\phi = \bigvee_{i=1}^m C_i$, where each $C_i : \{0,1\}^n \to \{0,1\}$ is a conjunction. Actually, we will deal with the more general problem in which we are (implicitly) given m subsets $S_1, ..., S_m \subseteq \{0,1\}^n$ and wish to approximate $|\bigcup_i S_i|$. In our case, each S_i is the set of assignments satisfying the conjunction C_i. In general, we make two computational assumptions regarding these sets (letting efficient mean implementable in time polynomial in $n \cdot m$):

1. Given $i \in [m]$, one can efficiently determine $|S_i|$.
2. Given $i \in [m]$ and $J \subseteq [m]$, one can efficiently approximate $\mathsf{Pr}_{s \in S_i}\left[s \in \bigcup_{j \in J} S_j\right]$ up to an *additive deviation of* $1/\text{poly}(n+m)$.

These assumptions are satisfied in our setting (where $S_i = C_i^{-1}(1)$, see Exercise 6.30). Now, the key observation towards approximating $|\bigcup_{i=1}^m S_i|$ is that

$$\left|\bigcup_{i=1}^m S_i\right| = \sum_{i=1}^m \left|S_i \setminus \bigcup_{j<i} S_j\right| = \sum_{i=1}^m |S_i| \cdot \mathsf{Pr}_{s \in S_i}\left[s \notin \bigcup_{j<i} S_j\right] \tag{6.7}$$

and that the probabilities in Eq. (6.7) can be approximated by the second assumption. This leads to the following algorithm, where ε denotes the desired deviation parameter (i.e., we wish to obtain $(1 \pm \varepsilon) \cdot |\bigcup_{i=1}^m S_i|$).

Construction 6.25 *Let $\varepsilon' = \varepsilon/m$. For $i = 1$ to m do:*

1. *Using the first assumption, compute $|S_i|$.*
2. *Using the second assumption, obtain $\widetilde{p}_i = p_i \pm \varepsilon'$, where $p_i \stackrel{\text{def}}{=} \mathsf{Pr}_{s \in S_i}[s \notin \bigcup_{j<i} S_j]$. Set $a_i \stackrel{\text{def}}{=} \widetilde{p}_i \cdot |S_i|$.*

Output the sum of the a_i's.

Let $N_i = p_i \cdot |S_i|$. We are interested in the quality of the approximation to $\sum_i N_i = |\bigcup_i S_i|$ provided by $\sum_i a_i$. Using $a_i = (p_i \pm \varepsilon') \cdot |S_i| = N_i \pm \varepsilon' \cdot |S_i|$ (for all i's), we have $\sum_i a_i = \sum_i N_i \pm \varepsilon' \cdot \sum_i |S_i|$. Using $\sum_i |S_i| \leq m \cdot |\bigcup_i S_i| = m \cdot \sum_i N_i$ (and $\varepsilon = m\varepsilon'$), we get $\sum_i a_i = (1 \pm \varepsilon) \cdot \sum_i N_i$. Thus, we obtain the following result (see Exercise 6.30).

Proposition 6.26 *For every positive polynomial p, the counting problem of* $R_{\texttt{dnf}}$ *is efficiently* $(1-(1/p))$*-approximable.*

Using the reduction presented in the proof of Theorem 6.19, we conclude that the number of *un*satisfying assignments to a given CNF formula is efficiently $(1-(1/p))$-approximable. We warn, however, that the number of satisfying assignments to such a formula is *not* efficiently approximable. This concurs with the general phenomenon by which *relative approximation may be possible for one quantity, but not for the complementary quantity.* Needless to say, such a phenomenon does not occur in the context of additive-deviation approximation.

6.2.2.2 Relative approximation for $\#\mathcal{P}$

Recall that we cannot expect to efficiently approximate every $\#\mathcal{P}$ problem, where throughout the rest of this section "approximation" is used as a shorthand for "relative approximation" (as in Definition 6.24). Specifically, efficiently approximating $\#R$ yields an efficient algorithm for deciding membership in $S_R = \{x : R(x) \neq \emptyset\}$. Thus, at best we can hope that approximating $\#R$ is not harder than deciding S_R (i.e., that approximating $\#R$ is reducible in polynomial-time to S_R). This is indeed the case for every NP-complete problem (i.e., if S_R is NP-complete). More generally, we show that approximating any problem in $\#\mathcal{P}$ is reducible in probabilistic polynomial-time to $\mathcal{NP}$.

Theorem 6.27 *For every* $R \in \mathcal{PC}$ *and positive polynomial p, there exists a probabilistic polynomial-time oracle machine that when given oracle access to* $\mathcal{NP}$ *constitutes a* $(1/p,\mu)$*-approximator of* $\#R$*, where* μ *is a negligible function* (e.g., $\mu(n) = 2^{-n}$).

Recall that it suffices to provide a $(1/p,\delta)$-approximator of $\#R$, for any constant $\delta < 0.5$, because error reduction is applicable in this context (see Exercise 6.28). Also, it suffices to provide a $(1/2,\delta)$-approximator for every problem in $\#\mathcal{P}$ (see Exercise 6.31).

Proof: Given x, we show how to approximate $|R(x)|$ to within some constant factor. The desired $(1-(1/p))$-approximation can be obtained as in Exercise 6.31. We may also assume that $R(x) \neq \emptyset$, by starting with the query "is x in S_R" and halting (with output 0) if the answer is negative. Without loss of generality, we assume that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \text{poly}(|x|)$. We focus on finding some $i \in \{1,...,\ell\}$ such that $2^{i-4} \leq |R(x)| \leq 2^{i+4}$.

We proceed in iterations. For $i = 1,...,\ell+1$, we find out whether or not $|R(x)| < 2^i$. If the answer is positive then we halt with output 2^i, and otherwise we proceed to the next iteration. (Indeed, if we were able to obtain correct answers to these queries then the output 2^i would satisfy $2^{i-1} \leq |R(x)| < 2^i$.)

Needless to say, the key issue is how to check whether $|R(x)| < 2^i$. The main idea is to use a "random sieve" on the set $R(x)$ such that each element passes the sieve with probability 2^{-i}. Thus, we expect $|R(x)|/2^i$ elements of $R(x)$ to pass the sieve. Assuming that the number of elements in $R(x)$ that pass the random

sieve is indeed $\lfloor |R(x)|/2^i \rfloor$, it holds that $|R(x)| \geq 2^i$ *if and only if some element of* $R(x)$ *passes the sieve*. Assuming that the sieve can be implemented efficiently, the question of whether or not some element in $R(x)$ passed the sieve is of an "NP-type" (and thus can be referred to our NP-oracle). Combining both assumptions, we may implement the foregoing process by proceeding to the next iteration as long as some element of $R(x)$ passes the sieve. Furthermore, this implementation will provide a reasonably good approximation even if the number of elements in $R(x)$ that pass the random sieve is only approximately equal to $|R(x)|/2^i$. In fact, the level of approximation that this implementation provides is closely related to the level of approximation that is provided by the random sieve. Details follow.

Implementing a random sieve. The random sieve is implemented by using a family of hashing functions (see Appendix D.2). Specifically, in the ith iteration we use a family H_ℓ^i such that each $h \in H_\ell^i$ has a poly(ℓ)-bit long description and maps ℓ-bit long strings to i-bit long strings. Furthermore, the family is accompanied with an efficient evaluation algorithm (i.e., mapping adequate pairs (h, x) to $h(x)$) and satisfies (for every $S \subseteq \{0,1\}^\ell$)

$$\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in S : h(y) = 0^i\}| \notin (1 \pm \varepsilon) \cdot 2^{-i}|S|] \;<\; \frac{2^i}{\varepsilon^2 |S|} \tag{6.8}$$

(see Lemma D.4). *The random sieve will let y pass if and only if $h(y) = 0^i$.* Indeed, this random sieve is not as perfect as we assumed in the foregoing discussion, but Eq. (6.8) suggests that in some sense this sieve is good enough.

Implementing the queries. Recall that for some x, i and $h \in H_\ell^i$, we need to determine whether $\{y \in R(x) : h(y) = 0^i\} = \emptyset$. This type of question can be cast as membership in the set

$$S_{R,H} \stackrel{\text{def}}{=} \{(x, i, h) : \exists y \text{ s.t. } (x, y) \in R \wedge h(y) = 0^i\}. \tag{6.9}$$

Using the hypotheses that $R \in \mathcal{PC}$ and that the family of hashing functions has an efficient evaluation algorithm, it follows that $S_{R,H}$ is in $\mathcal{NP}$.

The actual procedure. On input $x \in S_R$ and oracle access to $S_{R,H}$, we proceed in iterations, starting with $i = 1$ and halting at $i = \ell$ (if not before), where ℓ denotes the length of the potential solutions for x. In the ith iteration (where $i < \ell$), we uniformly select $h \in H_\ell^i$ and query the oracle on whether or not $(x, i, h) \in S_{R,H}$. If the answer is negative then we halt with output 2^i, and otherwise we proceed to the next iteration (using $i \leftarrow i + 1$). Needless to say, if we reach the last iteration (i.e., $i = \ell$) then we just halt with output 2^ℓ.

Indeed, we have ignored the case that $x \notin S_R$, which can be easily handled by a minor modification of the foregoing procedure. Specifically, on input x, we first query S_R on x and halt with output 0 if the answer is negative. Otherwise we proceed as in the foregoing procedure.

The analysis. We upper-bound separately the probability that the procedure outputs a value that is too small and the probability that it outputs a value that is too big. In light of the foregoing discussion, we may assume that $|R(x)| > 0$, and let $i_x = \lfloor \log_2 |R(x)| \rfloor \geq 0$.

1. The probability that the procedure *halts in a specific iteration* $i < i_x$ equals $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| = 0]$, which in turn is upper-bounded by $2^i/|R(x)|$ (using Eq. (6.8) with $\varepsilon = 1$). Thus, the probability that the procedure halts *before* iteration $i_x - 3$ is upper-bounded by $\sum_{i=0}^{i_x-4} 2^i/|R(x)|$, which in turn is less than $1/8$ (because $i_x \leq \log_2 |R(x)|$). Thus, with probability at least $7/8$, the output is at least $2^{i_x-3} > |R(x)|/16$ (because $i_x > (\log_2 |R(x)|) - 1$).

2. The probability that the procedure *does not halt in iteration* $i > i_x$ equals $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \geq 1]$, which in turn is upper-bounded by $\alpha/(\alpha-1)^2$, where $\alpha = 2^i/|R(x)| > 1$ (using Eq. (6.8) with $\varepsilon = \alpha - 1$).[12] Thus, the probability that the procedure does not halt by iteration $i_x + 4$ is upper-bounded by $8/49 < 1/6$ (because $i_x > (\log_2 |R(x)|) - 1$). Thus, with probability at least $5/6$, the output is at most $2^{i_x+4} \leq 16 \cdot |R(x)|$ (because $i_x \leq \log_2 |R(x)|$).

Thus, with probability at least $(7/8)-(1/6) > 2/3$, the foregoing procedure outputs a value v such that $v/16 \leq |R(x)| < 16v$. Reducing the deviation by using the ideas presented in Exercise 6.31 (and reducing the error probability as in Exercise 6.28), the theorem follows. ■

Perspective. The key observation underlying the proof Theorem 6.27 is that, while (even with the help of an NP-oracle) we cannot directly test whether the number of solutions is greater than a given number, we can test (with the help of an NP-oracle) whether the number of solutions that "survive a random sieve" is greater than zero. If fact, we can also test whether the number of solutions that "survive a random sieve" is greater than a small number, where small means polynomial in the length of the input (see Exercise 6.33). That is, the complexity of this test is linear in the size of the threshold, and not in the length of its binary description. Indeed, in many settings it is more advantageous to use a threshold that is polynomial in some efficiency parameter (rather than using the threshold zero); examples appear in §6.2.4.2 and in [102].

6.2.3 Searching for unique solutions

A natural computational problem (regarding search problems), which arises when discussing the number of solutions, is the problem of distinguishing instances having a single solution from instances having no solution (or finding the unique solution whenever such exists). We mention that instances having a single solution facilitate numerous arguments (see, for example, Exercise 6.23 and §10.2.2.1). Formally, searching for and deciding the existence of unique solutions are defined within the framework of promise problems (see Section 2.4.1).

[12] A better bound can be obtained by using the hypothesis that, for every y, when h is uniformly selected in H_ℓ^i, the value of $h(y)$ is uniformly distributed in $\{0,1\}^i$. In this case, $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \geq 1]$ is upper-bounded by $\mathsf{E}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}|] = |R(x)|/2^i$.

Definition 6.28 (search and decision problems for unique solution instances): *The* set of instances having unique solutions *with respect to the binary relation* R *is defined as* $\texttt{US}_R \stackrel{\text{def}}{=} \{x : |R(x)| = 1\}$, *where* $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$. *As usual, we denote* $S_R = \{x : |R(x)| \geq 1\}$, *and* $\overline{S}_R \stackrel{\text{def}}{=} \{0,1\}^* \setminus S_R = \{x : |R(x)| = 0\}$.

- *The problem of* finding unique solutions for R *is defined as the* search problem R with promise $\texttt{US}_R \cup \overline{S}_R$ (see Definition 2.28).

  In continuation to Definition 2.29, the candid searching for unique solutions for R *is defined as the* search problem R with promise $\texttt{US}_R$.

- *The problem of* deciding unique solution for R *is defined as the* promise problem $(\texttt{US}_R, \overline{S}_R)$ (see Definition 2.30).

Interestingly, in many natural cases, the promise does not make any of these problems any easier than the original problem. That is, for all known NP-complete problems, the original problem is reducible in probabilistic polynomial-time to the corresponding unique instances problem.

Theorem 6.29 *Let* $R \in \mathcal{PC}$ *and suppose that every search problem in* $\mathcal{PC}$ *is parsimoniously reducible to* R. *Then solving the search problem of* R (resp., deciding membership in S_R) *is reducible in probabilistic polynomial-time to finding unique solutions for* R (resp., to the promise problem $(\texttt{US}_R, \overline{S}_R)$). *Furthermore, there exists a probabilistic polynomial-time computable mapping* M *such that for every* $x \in \overline{S}_R$ *it holds that* $\Pr[M(x) \in \overline{S}_R] = 1$, *whereas for every* $x \in S_R$ *it holds that* $\Pr[M(x) \in \texttt{US}_R] \geq 1/\text{poly}(|x|)$.

We highlight the fact that the hypothesis asserts that R is $\mathcal{PC}$-complete *via parsimonious reductions*; this hypothesis is crucial to Theorem 6.29 (see Exercise 6.34). The large (but bounded-away from 1) error probability of the randomized Karp-reduction M can be reduced by repetitions, yielding a randomized Cook-reduction with exponentially vanishing error probability. Note that the resulting reduction may make many queries that violate the promise, and still yields the correct answer (with high probability) by relying on queries that satisfy the promise. (Specifically, in the case of search problems, we avoid wrong solutions by checking each solution obtained, while in the case of decision problems we rely on the fact that for every $x \in \overline{S}_R$ it always holds that $M(x) \in \overline{S}_R$.)

Proof: As in the proof of Theorem 6.27, the idea is to apply a "random sieve" on $R(x)$, this time with the hope that a single element survives. Specifically, if we let each element passes the sieve with probability approximately $1/|R(x)|$ then with constant probability a single element survives (and we shall obtain an instance with a unique solution). Sieving will be performed by a random function selected in an adequate hashing family (see Appendix D.2). A couple of questions arise:

1. *How do we get an approximation to* $|R(x)|$? Note that we need such an approximation in order to determine the adequate hashing family. Indeed, we may just invoke Theorem 6.27, but this will not yield a many-to-one

reduction. Instead, we just select $m \in \{0, ..., \text{poly}(|x|)\}$ uniformly and note that (if $|R(x)| > 0$ then) $\Pr[m = \lceil \log_2 |R(x)| \rceil] = 1/\text{poly}(|x|)$. Next, we randomly map x to (x, m, h), where h is uniformly selected in an adequate hashing family.

2. *How does the question of whether a single element of $R(x)$ pass the random sieve translate to an instance of the unique-solution problem for R?* Recall that in the proof of Theorem 6.27 the non-emptiness of the set of element of $R(x)$ that pass the sieve (defined by h) was determined by checking membership (of (x, m, h)) in $S_{R,H} \in \mathcal{NP}$ (defined in Eq. (6.9)). Furthermore, the number of NP-witnesses for $(x, m, h) \in S_{R,H}$ equals the number of elements of $R(x)$ that pass the sieve. Using the parsimonious reduction of $S_{R,H}$ to S_R (which is guaranteed by the theorem's hypothesis), we obtained the desired instance.

Note that in case $R(x) = \emptyset$ the aforementioned mapping always generates a no-instance (of $S_{R,H}$ and thus of S_R). Details follow.

Implementation (i.e., the mapping M). As in the proof of Theorem 6.27, we assume, without loss of generality, that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \text{poly}(|x|)$. We start by uniformly selecting $m \in \{1, ..., \ell + 1\}$ and $h \in H_\ell^m$, where H_ℓ^m is a family of efficiently computable and pairwise-independent hashing functions (see Definition D.1) mapping ℓ-bit long strings to m-bit long strings. Thus, we obtain an instance (x, m, h) of $S_{R,H} \in \mathcal{NP}$ such that the set of valid solutions for (x, m, h) equals $\{y \in R(x) : h(y) = 0^m\}$. Using the parsimonious reduction g of $S_{R,H}$ to S_R, we map (x, m, h) to $g(x, m, h)$, and it holds that $|\{y \in R(x) : h(y) = 0^m\}|$ equals $|R(g(x, m, h))|$. To summarize, on input x the randomized mapping M outputs the instance $M(x) \stackrel{\text{def}}{=} g(x, m, h)$, where $m \in \{1, ..., \ell + 1\}$ and $h \in H_\ell^m$ are uniformly selected.

The analysis. Note that for any $x \in \overline{S}_R$ it holds that $\Pr[M(x) \in \overline{S}_R] = 1$. Assuming that $x \in S_R$, with probability exactly $1/(\ell + 1)$ it holds that $m = m_x$, where $m_x \stackrel{\text{def}}{=} \lceil \log_2 |R(x)| \rceil + 1$. Focusing on the case that $m = m_x$, for a uniformly selected $h \in H_\ell^m$, we shall lower-bound the probability that the set $R_h(x) \stackrel{\text{def}}{=} \{y \in R(x) : h(y) = 0^m\}$ is a singleton. First, using the Inclusion-Exclusion Principle, we lower-bound $\Pr_{h \in H_\ell^{m_x}}[|R_h(x)| > 0]$ by

$$\sum_{y \in R(x)} \Pr_{h \in H_\ell^{m_x}}[h(y) = 0^{m_x}] - \sum_{y_1 < y_2 \in R(x)} \Pr_{h \in H_\ell^{m_x}}[h(y_1) = h(y_2) = 0^{m_x}].$$

Next, we upper-bound $\Pr_{h \in H_\ell^{m_x}}[|R_h(x)| > 1]$ by

$$\sum_{y_1 < y_2 \in R(x)} \Pr_{h \in H_\ell^{m_x}}[h(y_1) = h(y_2) = 0^{m_x}].$$

Combining these two bounds, we get

$$\Pr_{h \in H_\ell^{m_x}}[|R_h(x)| = 1] \tag{6.10}$$

$$
\begin{aligned}
&= \mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|>0] \;-\; \mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|>1] \\
&\geq \sum_{y\in R(x)} \mathsf{Pr}_{h\in H_\ell^{m_x}}[h(y)=0^{m_x}] \;-\; 2\cdot \sum_{y_1<y_2\in R(x)} \mathsf{Pr}_{h\in H_\ell^{m_x}}[h(y_1)=h(y_2)=0^{m_x}] \\
&= |R(x)|\cdot 2^{-m_x} - 2\cdot \binom{|R(x)|}{2}\cdot 2^{-2m_x}
\end{aligned}
$$

where the last equality is due to the pairwise independence property. Using $2^{m_x-2} < |R(x)| \leq 2^{m_x-1}$, it follows that Eq. (6.10) is lower-bounded by 1/8. Thus, $\mathsf{Pr}[M(x)\in \mathtt{US}_R] \geq 1/8(\ell+1)$, and the theorem follows. ■

Comment. Theorem 6.29 is sometimes stated as referring to the unique solution problem of SAT. In this case and when using a specific family of pairwise independent hashing functions, the use of the parsimonious reduction can be avoided. For details see Exercise 6.35.

6.2.4 Uniform generation of solutions

We now turn to a new type of computational problems, which may be viewed as a straining of search problems. We refer to the task of generating a uniformly distributed solution for a given instance, rather than merely finding an adequate solution. Needless to say, by definition, algorithms solving this ("uniform generation") task must be randomized. Focusing on relations in $\mathcal{PC}$ we consider two versions of the problem, which differ by the level of approximation provided for the desired (uniform) distribution.[13]

Definition 6.30 (uniform generation): *Let $R\in\mathcal{PC}$ and $S_R=\{x:|R(x)|\geq 1\}$, and let Π be a probabilistic process.*

1. *We say that Π* solves the uniform generation problem of *R if, on input $x\in S_R$, the process Π outputs either an element of $R(x)$ or a special symbol, denoted $\perp$, such that $\mathsf{Pr}[\Pi(x)\in R(x)]\geq 1/2$ and for every $y\in R(x)$ it holds that $\mathsf{Pr}[\Pi(x)=y\,|\,\Pi(x)\in R(x)]=1/|R(x)|$.*
2. *For $\varepsilon:\mathbb{N}\to[0,1]$, we say that Π* solves the $(1-\varepsilon)$-approximate uniform generation problem of *R if, on input $x\in S_R$, the distribution $\Pi(x)$ is $\varepsilon(|x|)$-close*[14] *to the uniform distribution on $R(x)$.*

In both cases, without loss of generality, we may require that if $x\notin S_R$ then $\mathsf{Pr}[\Pi(x)=\perp]=1$. More generally, we may require that Π never outputs a string not in $R(x)$.

[13]Note that a probabilistic algorithm running in strict polynomial-time is not able to output a perfectly uniform distribution on sets of certain sizes. Specifically, referring to the standard model that allows only for uniformly selected binary values, such algorithms cannot output a perfectly uniform distribution on sets having cardinality that is not a power of two.

[14]See Appendix D.1.1.

Note that the error probability of uniform generation (as in Item 1) can be made exponentially vanishing (in $|x|$) by employing error-reduction. In contrast, we are not aware of any general way of reducing the deviation of an approximate uniform generation procedure (as in Item 2).[15]

In §6.2.4.1 we show that, for many search problems, approximate uniform generation is computationally equivalent to approximate counting. In §6.2.4.2 we present a direct approach for solving the uniform generation problem of any search problem in $\mathcal{PC}$ by using an oracle to $\mathcal{NP}$.

6.2.4.1 Relation to approximate counting

We show that, for many natural search problems in $\mathcal{PC}$, the approximate counting problem associated with R is computationally equivalent to approximate uniform generation with respect to R. Specifically, we refer to search problems $R \in \mathcal{PC}$ such that $R'(x;y') \stackrel{\rm def}{=} \{y'' : (x,y'y'') \in R\}$ is *strongly parsimoniously reducible to* R, where a strongly parsimonious reduction of R' to R is a parsimonious reduction g that is coupled with an efficiently computable 1-1 mapping of pairs $(g(x),y) \in R$ to pairs $(x,h(x,y)) \in R'$ (i.e., h is efficiently computable and $h(x,\cdot)$ is a 1-1 mapping of $R(g(x))$ to $R'(x)$). Note that for many natural search problems R (e.g., the search problem of `SAT` and `Perfect Matching`), the corresponding R' is strongly parsimoniously reducible to R.

Recalling that both types of approximation problems are parameterized by the level of precision, we obtain the following quantitative form of the aforementioned equivalence.

Theorem 6.31 *Let $R \in \mathcal{PC}$ and let ℓ be a polynomial such that for every $(x,y) \in R$ it holds that $|y| \leq \ell(|x|)$. Suppose that R' is strongly parsimoniously reducible to R, where $R'(x;y') \stackrel{\rm def}{=} \{y'' : (x,y'y'') \in R\}$.*

1. From approximate counting to approximate uniform generation: *Let $\varepsilon(n) = 1/5\ell(n)$ and let $\mu : \mathbb{N} \to (0,1)$ be a function satisfying $\mu(n) \geq \exp(-\mathrm{poly}(n))$. Then, $(1-\mu)$-approximate uniform generation for R is reducible in probabilistic polynomial-time to $(1-\varepsilon)$-approximating $\#R$.*

2. From approximate uniform generation to approximate counting: *For every noticeable $\varepsilon : \mathbb{N} \to (0,1)$* (i.e., $\varepsilon(n) \geq 1/\mathrm{poly}(n)$ for every n)*, the problem of $(1-\varepsilon)$-approximating $\#R$ is reducible in probabilistic polynomial-time to $(1-\varepsilon')$-approximate uniform generation problem of R, where $\varepsilon'(n) = \varepsilon(n)/5\ell(n)$.*

In fact, Part 1 holds also in case R' is just parsimoniously reducible to R.

Note that the quality of the approximate uniform generation asserted in Part 1 (i.e., μ) is independent of the quality of the approximate counting procedure (i.e., ε) to which the former is reduced, provided that the approximate counter performs better than some threshold. On the other hand, the quality of the approximate

[15]We note that in some cases, the deviation of an approximate uniform generation procedure can be reduced. See discussion following Theorem 6.31.

counting asserted in Part 2 (i.e., ε) does depend on the quality of the approximate uniform generation (i.e., ε'), but cannot reach beyond a certain bound (i.e., noticeable relative devaition). Recall, that for problems that are NP-complete under parsimonious reductions the quality of approximate counting procedures can be improved (see Exercise 6.32). However, Theorem 6.31 is most useful when applied to problems that are not NP-complete, because for problems that are NP-complete both approximate counting and uniform generation are randomly reducible to the corresponding search problem (see Exercise 6.37).

Proof: Throughout the proof, we assume for simplicity (and in fact without loss of generality) that $R(x) \neq \emptyset$ and $R(x) \subseteq \{0,1\}^{\ell(|x|)}$.

Towards Part 1, let us first reduce the uniform generation problem of R to $\#R$ (rather than to approximating $\#R$). On input $x \in S_R$, we generate a uniformly distributed $y \in R(x)$ by randomly generating its bits one after the other. We proceed in iterations, entering the ith iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\text{def}}{=} \{y'' : (x, y'y'') \in R\}$ is not empty. With probability $|R'(x;y'1)|/|R'(x;y')|$ we set the ith bit to equal 1, and otherwise we set it to equal 0. We obtain both $|R'(x;y'1)|$ and $|R'(x;y')|$ by using a parsimonious reduction g of $R' = \{((x;y'),y'') : (x,y'y'') \in R\} \in \mathcal{PC}$ to R. That is, we obtain $|R'(x;y')|$ by querying for the value of $|R(g(x;y'))|$. Ignoring integrality issues, all this works perfectly (i.e., we generate an $\ell(n)$-bit string uniformly distributed in $R(x)$) as long as we have oracle access to $\#R$. But we only have oracle access to an approximation of $\#R$, and thus a careful modification is in place.

Let us denote the approximation oracle by A. Firstly, by adequate error reduction, we may assume that, for every x, it holds that $\mathsf{Pr}[A(x) \in (1 \pm \varepsilon(n)) \cdot \#R(x)] > 1 - \mu'(|x|)$, where $\mu'(n) = \mu(n)/\ell(n)$. In the rest of the analysis we ignore the probability that the estimate deviates from the aforementioned interval, and note that this rare event is the only source of the possible deviation of the output distribution from the uniform distribution on $R(x)$.[16] Let us assume for a moment that A is *deterministic* and that for every x and y' it holds that

$$A(g(x,y'0)) + A(g(x,y'1)) \ \leq \ A(g(x;y')). \tag{6.11}$$

We also assume that the approximation is correct at the "trivial level" (where one may just check whether or not (x,y) is in R); that is, for every $y \in \{0,1\}^{\ell(|x|)}$, it holds that

$$A(g(x;y)) = 1 \text{ if } (x,y) \in R \text{ and } A(g(x;y)) = 0 \text{ otherwise.} \tag{6.12}$$

We modify the ith iteration of the foregoing procedure such that, when entering with the $(i-1)$-bit long prefix y', we set the ith bit to $\sigma \in \{0,1\}$ with probability $A(g(x;y'\sigma))/A(g(x;y'))$ and halt (with output $\perp$) with the residual probability (i.e., $1-(A(g(x;y'0))/A(g(x;y'))) - (A(g(x;y'1))/A(g(x;y')))$). Indeed, Eq. (6.11) guarantees that the latter instruction is sound, since the two main probabilities sum-up to at most 1. If we completed the last (i.e., $\ell(|x|)^{\text{th}}$) iteration, then we

[16] The possible deviation is due to the fact that this rare event may occur with different probability in the different invocations of algorithm A.

output the $\ell(|x|)$-bit long string that was generated. Thus, as long as Eq. (6.11) holds (but regardless of other aspects of the quality of the approximation), every $y = \sigma_1 \cdots \sigma_{\ell(|x|)} \in R(x)$, is output with probability

$$\frac{A(g(x;\sigma_1))}{A(g(x;\lambda))} \cdot \frac{A(g(x;\sigma_1\sigma_2))}{A(g(x;\sigma_1))} \cdots \frac{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)}))}{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)-1}))} \qquad (6.13)$$

which, by Eq. (6.12), equals $1/A(g(x;\lambda))$. Thus, the procedure outputs each element of $R(x)$ with equal probability, and never outputs a non-$\perp$ value that is outside $R(x)$. It follows that the quality of approximation only effects the probability that the procedure outputs a non-$\perp$ value (which in turn equals $|R(x)|/A(g(x;\lambda))$). The key point is that, as long as Eq. (6.12) holds, the specific approximate values obtained by the procedure are immaterial – with the exception of $A(g(x;\lambda))$, all these values "cancel out".

We now turn to enforcing Eq. (6.11) and Eq. (6.12). We may enforce Eq. (6.12) by performing the straightforward check (of whether or not $(x,y) \in R$) rather than invoking $A(g(x,y))$.[17] As for Eq. (6.11), we enforce it artificially by using $A'(x,y') \stackrel{\text{def}}{=} (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)} \cdot A(g(x;y'))$ instead of $A(g(x;y'))$. Recalling that $A(g(x;y')) = (1 \pm \varepsilon(|xy'|)) \cdot |R'(x;y')|$, we have

$$\begin{aligned} A'(x,y') &> (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)} \cdot (1-\varepsilon(|x|)) \cdot |R'(x;y')| \\ A'(x,y'\sigma) &< (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|-1)} \cdot (1+\varepsilon(|x|)) \cdot |R'(x;y'\sigma)| \end{aligned}$$

and the claim follows using $(1-\varepsilon(|x|)) \cdot (1+\varepsilon(|x|))^3 > (1-\varepsilon(|x|))$. Note that the foregoing modification only decreases the probability of outputting a non-$\perp$ value by a factor of $(1+\varepsilon(|x|))^{3\ell(|x|)} < 2$, where the inequality is due to the setting of ε (i.e., $\varepsilon(n) = 1/5\ell(n)$). Finally, we refer to our assumption that A is deterministic. This assumption was only used in order to identify the value of $A(g(x,y'))$ obtained and used in the $(|y'|-1)^{\text{st}}$ iteration with the value of $A(g(x,y'))$ obtained and used in the $|y'|^{\text{th}}$ iteration, but the same effect can be obtained by just re-using the former value (in the $|y'|^{\text{th}}$ iteration) rather than re-invoking A in order to obtain it. Part 1 follows.

Towards Part 2, let use first reduce the task of approximating $\#R$ to the task of (exact) uniform generation for R. On input $x \in S_R$, the reduction uses the tree of possible prefixes of elements of $R(x)$ in a somewhat different manner. Again, we proceed in iterations, entering the ith iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\text{def}}{=} \{y'' : (x,y'y'') \in R\}$ is not empty. At the ith iteration we estimate the bigger among the two fractions $|R'(x;y'0)|/|R'(x;y')|$ and $|R'(x;y'1)|/|R'(x;y')|$, by uniformly sampling the uniform distribution over $R'(x;y')$. That is, taking $\text{poly}(|x|/\varepsilon'(|x|))$ uniformly distributed samples in $R'(x;y')$, we obtain with overwhelmingly high probability an approximation of these fractions up to an additive deviation of at most $\varepsilon'(|x|)/3$. This means that we obtain

[17] Alternatively, we note that since A is a $(1-\varepsilon)$-approximator for $\varepsilon < 1$ it must hold that $\#R'(z) = 0$ implies $A(z) = 0$. Also, since $\varepsilon < 1/3$, if $\#R'(z) = 1$ then $A(z) \in (2/3, 4/3)$, which may be rounded to 1.

a relative approximation up-to a factor of $1 \pm \varepsilon'(|x|)$ for the fraction (or fractions) that is (resp., are) bigger than 1/3. Indeed, we may not be able to obtain such a good relative approximation of the other fraction (in case it is very small), but this does not matter. It also does not matter that we cannot tell which is the bigger fraction among the two; it only matter that we use an approximation that indicates a quantity that is, say, bigger than 1/3. We proceed to the next iteration by augmenting y' using the bit that corresponds to such a quantity. Specifically, suppose that we obtained the approximations $a_0(y') \approx |R'(x;y'0)|/|R'(x;y')|$ and $a_1(y') \approx |R'(x;y'1)|/|R'(x;y')|$. Then we extend y' by the bit 1 if $a_1(y') > a_0(y')$ and extend y' by the bit 0 otherwise. Finally, when we reach $y = \sigma_1 \cdots \sigma_{\ell(|x|)}$ such that $(x,y) \in R$, we output

$$a_{\sigma_1}(\lambda)^{-1} \cdot a_{\sigma_2}(\sigma_1)^{-1} \cdots a_{\sigma_{\ell(|x|)}}(\sigma_1\sigma_2 \cdots \sigma_{\ell(|x|)-1})^{-1}. \tag{6.14}$$

As in Part 1, actions regarding R' (in this case uniform generation in R') are conducted via the parsimonious reduction g to R. That is, whenever we need to sample uniformly in the set $R'(x;y')$, we sample the set $R(g(x;y'))$ and recover the corresponding element of $R'(x;y')$ by using the mapping guaranteed by the hypothesis that g is strongly parsimonious. Finally, note that the deviation from uniform distribution (i.e., the fact that we can only approximately sample R) merely introduces such a deviation in each of our approximations to the relevant fractions (i.e., to a fraction bigger than 1/3). Specifically, on input x, using an oracle that provides a $(1-\varepsilon')$-approximate uniform generation for R, with overwhelmingly high probability, the output (as defined in Eq. (6.14)) is in

$$\prod_{i=1}^{\ell(|x|)} \left((1 \pm 2\varepsilon'(|x|)) \cdot \frac{|R'(x;\sigma_1 \cdots \sigma_{i-1})|}{|R'(x;\sigma_1 \cdots \sigma_i)|} \right) \tag{6.15}$$

where the error probability is due to the unlikely case that in one of the iterations our approximations deviates from the correct value by more than an additive deviation term of $\varepsilon'(n)/3$. Noting that Eq. (6.15) equals $(1 \pm 2\varepsilon'(|x|))^{\ell(|x|)} \cdot |R(x)|$ and using $(1 \pm 2\varepsilon'(|x|))^{\ell(|x|)} \subset (1 \pm \varepsilon(|x|))$, Part 2 follows, and so does the theorem. ■

6.2.4.2 A direct procedure for uniform generation

We conclude the current chapter by presenting a direct procedure for solving the uniform generation problem of any $R \in \mathcal{PC}$. This procedure uses an oracle to $\mathcal{NP}$, which is unavoidable because solving the uniform generation problem implies solving the corresponding search problem. One advantage of this procedure, over the reduction presented in §6.2.4.1, is that it solves the uniform generation problem rather than the *approximate* uniform generation problem.

We are going to use hashing again, but this time we use a family of hashing functions having a stronger "uniformity property" (see Appendix D.2.3). Specifically, we will use a family of ℓ-wise independent hashing functions mapping ℓ-bit strings to m-bit strings, where ℓ bounds the length of solutions in R, and rely on

the fact that such a family satisfies Lemma D.6. Intuitively, such functions partition $\{0,1\}^\ell$ into 2^m cells and Lemma D.6 asserts that these partitions "uniformly shatter" all sufficiently large sets. That is, for every set $S \subseteq \{0,1\}^\ell$ of size $\Omega(\ell \cdot 2^m)$ the partition induced by almost every function is such that each cell contains approximately $|S|/2^m$ elements of S. In particular, if $|S| = \Theta(\ell \cdot 2^m)$ then each cell contains $\Theta(\ell)$ elements of S.

Loosely speaking, the following procedure (for uniform generation) first selects a random hashing function and tests whether it "uniformly shatters" the target set S. If this condition holds then the procedure selects a cell at random and retrieve the elements of S residing in the chosen cell. Finally, the procedure outputs each retrieves element (in S) with a fixed probability, which is independent of the actual number of elements of S that reside in the chosen cell. This guarantees that each element $e \in S$ is output with the same probability, regardless of the number of elements of S that resides with e in the same cell.

In the following construction, we assume that on input x we also obtain a good approximation to the size of $R(x)$. This assumption can be enforced by using an approximate counting procedure as a preprocessing stage. Alternatively, the ideas presented in the following construction yield such an approximate counting procedure.

Construction 6.32 (uniform generation): *On input x and $m'_x \in \{m_x, m_x+1\}$, where $m_x \stackrel{\text{def}}{=} \lfloor \log_2 |R(x)| \rfloor$ and $R(x) \subseteq \{0,1\}^\ell$, the oracle machine proceeds as follows.*

1. Selecting a partition that "uniformly shatters" $R(x)$. *The machine sets $m = \max(0, m'_x - 6 - \log_2 \ell)$ and selects uniformly $h \in H_\ell^m$. Such a function defines a partition of $\{0,1\}^\ell$ into 2^m cells*[18]*, and the hope is that each cell contains approximately the same number of elements of $R(x)$. Next, the machine checks that this is indeed the case or rather than no cell contains more that 128ℓ elements of $R(x)$. This is done by checking whether or not $(x, h, 1^{128\ell+1})$ is in the set $S^{(1)}_{R,H}$ defined as follows*

$$\begin{aligned} S^{(1)}_{R,H} &\stackrel{\text{def}}{=} \{(x', h', 1^t) : \exists v \text{ s.t. } |\{y : (x',y) \in R \wedge h'(y) = v\}| \geq t\} \quad (6.16) \\ &= \{(x', h', 1^t) : \exists v, y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v, y_1, ..., y_t)\}, \end{aligned}$$

where $\psi^{(1)}(x', h', v, y_1, ..., y_t)$ holds if and only if $y_1 < y_2 \cdots < y_t$ and for every $j \in [t]$ it holds that $(x', y_j) \in R \wedge h'(y_j) = v$. Note that $S^{(1)}_{R,H} \in \mathcal{NP}$.

If the answer is positive (i.e., there exists a cell that contains more that 128ℓ elements of $R(x)$) *then the machine halts with output $\perp$. Otherwise, the machine continues with this choice of h. In this case, no cell contains more that 128ℓ elements of $R(x)$* (i.e., for every $v \in \{0,1\}^m$, it holds that $|\{y : (x,y) \in R \wedge h(y) = v\}| \leq 128\ell$). *We stress that this is an absolute guarantee that follows from $(x, h, 1^{128\ell+1}) \notin S^{(1)}_{R,H}$.*

[18]For sake of uniformity, we allow also the case of $m = 0$, which is rather artificial. In this case all hashing functions in H_ℓ^0 map $\{0,1\}^\ell$ to the empty string, which is viewed as 0^0, and thus define a trivial partition of $\{0,1\}^\ell$ (i.e., into a single cell).

2. Selecting a cell and determining the number of elements of $R(x)$ that are contained in it. *The machine selects uniformly* $v \in \{0,1\}^m$ *and determines* $s_v \stackrel{\rm def}{=} |\{y : (x,y) \in R \wedge h(y) = v\}|$ *by making queries to the following NP-set*

$$S^{(2)}_{R,H} \stackrel{\rm def}{=} \{(x', h', v', 1^t) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v', y_1, ..., y_t)\}. \quad (6.17)$$

Specifically, for $i = 1, ..., 128\ell$*, it checks whether* $(x, h, v, 1^i)$ *is in* $S^{(2)}_{R,H}$*, and sets* s_v *to be the largest value of* i *for which the answer is positive.*

3. Obtaining all the elements of $R(x)$ that are contained in the selected cell, and outputting one of them at random. *Using* s_v*, the procedure reconstructs the set* $S_v \stackrel{\rm def}{=} \{y : (x,y) \in R \wedge h(y) = v\}$*, by making queries to the following NP-set*

$$S^{(3)}_{R,H} \stackrel{\rm def}{=} \{(x', h', v', 1^t, j) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(3)}(x', h', v', y_1, ..., y_t, j)\}, \quad (6.18)$$

where $\psi^{(3)}(x', h', v', y_1, ..., y_t, j)$ *holds if and only if* $\psi^{(1)}(x', h', v', y_1, ..., y_t)$ *holds and the* $j^{\rm th}$ *bit of* $y_1 \cdots y_t$ *equals* 1*. Specifically, for* $j_1 = 1, ..., s_v$ *and* $j_2 = 1, ..., \ell$*, we make the query* $(x, h, v, 1^{s_v}, (j_1 - 1) \cdot \ell + j_2)$ *in order to determine the* $j_2^{\rm th}$ *bit of* y_{j_1}*. Finally, having recovered* S_v*, the procedure outputs each* $y \in S_v$ *with probability* $1/128\ell$*, and outputs* $\perp$ *otherwise (i.e., with probability* $1 - (s_v/128\ell)$*).*

Focusing on the case that $m'_x > 6 + \log_2 \ell$, we note that $m \leq m'_x - 6 - \log_2 \ell < \log_2(|R(x)|/20\ell)$. In this case, by Lemma D.6, with overwhelmingly high probability, each set $\{y : (x,y) \in R \wedge h(y) = v\}$ has cardinality $(1 \pm 0.5)|R(x)|/2^m$. Using $m'_x > (\log_2 |R(x)|) - 1$ (resp., $m'_x \leq (\log_2 |R(x)|) + 1$), it follows that $|R(x)|/2^m < 128\ell$ (resp., $|R(x)|/2^m \geq 16\ell$). Thus, Step 1 can be easily adapted to yield an approximate counting procedure for $\#R$ (see Exercise 6.36). However, our aim is to establish the following fact.

Proposition 6.33 *Construction 6.32 solves the uniform generation problem of* R.

Proof: By Lemma D.6 (and the setting of m), with overwhelmingly high probability, a uniformly selected $h \in H^m_\ell$ partitions $R(x)$ into 2^m cells, each containing at most 128ℓ elements. The key observation, stated in Step 1, is that if the procedure does not halt in Step 1 then it is indeed the case that h induces such a partition. The fact that these cells may contain a different number of elements is immaterial, because each element is output with the same probability (i.e., $1/128\ell$). What matters is that the average number of elements in the cells is sufficiently large, because this average number determines the probability that the procedure outputs an element of $R(x)$ (rather than $\perp$). Specifically, the latter probability equals the aforementioned average number (which equals $|R(x)|/2^m$) divided by 128ℓ. Using $m \leq \max(0, 1 + \log_2(2|R(x)|) - 6 - \log_2 \ell)$, we have $|R(x)|/2^m \geq \min(|R(x)|, 16\ell)$, which means that the procedure outputs some element of $R(x)$ with probability at least $\min((|R(x)|/128\ell), (1/8))$. ■

Technical comments. We can easily improve the performance of Construction 6.32 by dealing separately with the case $m = 0$. In such a case, Step 3 can be simplified and improved by uniformly selecting and outputting an element of S_λ (which equals $R(x)$). Under this modification, the procedure outputs some element of $R(x)$ with probability at least $1/8$. In any case, recall that the probability that a uniform generation procedure outputs $\perp$ can be deceased by repeated invocations.

Chapter Notes

One key aspect of randomized procedures is their success probability, which is obviously a quantitative notion. This aspect provides a clear connection between probabilistic polynomial-time algorithms considered in Section 6.1 and the counting problems considered in Section 6.2 (see also Exercise 6.19). More appealing connections between randomized procedures and counting problems (e.g., the application of randomization in approximate counting) are presented in Section 6.2. These connections justify the presentation of these two topics in the same chapter.

Randomized algorithms

Making people take an unconventional step requires compelling reasons, and indeed the study of randomized algorithms was motivated by a few compelling examples. Ironically, the appeal of the two most famous examples (discussed next) has been somewhat diminished due to subsequent finding, but the fundamental questions that emerged remain fascinating regardless of the status of these two examples. These questions refer to the power of randomization in various computational settings, and in particular in the context of decision and search problems. We shall return to these questions after briefly reviewing the story of the aforementioned examples.

The first example: primality testing. For more than two decades, primality testing was the archetypical example of the usefulness of randomization in the context of efficient algorithms. The celebrated algorithms of Solovay and Strassen [203] and of Rabin [177], proposed in the late 1970's, established that deciding primality is in co$\mathcal{RP}$ (i.e., these tests always recognize correctly prime numbers, but they may err on composite inputs). (The approach of Construction 6.4, which only establishes that deciding primality is in $\mathcal{BPP}$, is commonly attributed to M. Blum.) In the late 1980's, Adleman and Huang [2] proved that deciding primality is in $\mathcal{RP}$ (and thus in $\mathcal{ZPP}$). In the early 2000's, Agrawal, Kayal, and Saxena [3] showed that deciding primality is actually in $\mathcal{P}$. One should note, however, that strong evidence to the fact that deciding primality is in $\mathcal{P}$ was actually available from the start: we refer to Miller's deterministic algorithm [159], which relies on the Extended Riemann Hypothesis.

The second example: undirected connectivity. Another celebrated example to the power of randomization, specifically in the context of log-space computations, was provided by testing undirected connectivity. The random-walk algorithm presented in Construction 6.12 is due to Aleliunas, Karp, Lipton, Lovász, and Rackoff [5]. Recall that a deterministic log-space algorithm was found twenty-five years later (see Section 5.2.4 or [183]).

Other randomized algorithms. In addition to the two foregoing examples, several other appealing randomized algorithms are known. Confining ourselves to the context of search and decision problems, we mention the algorithms for finding perfect matchings and minimum cuts in graphs (see, e.g., [86, Apdx. B.1] or [161, Sec. 12.4&10.2]), and note the prominent role of randomization in computational number theory (see, e.g., [22] or [161, Chap. 14]). We mention that randomized algorithms are more abundant in the context of approximation problems (let alone in other computational settings (cf., e.g., Chapter 9, Appendix C, and Appendix D.3). For a general textbook on randomized algorithms, we refer the interested reader to [161].

While it can be shown that randomization is essential in several important computational settings (cf., e.g., Chapter 9, Appendix C, and Appendix D.3), a fundamental question is whether randomization is essential in the context of search and decision problems. The prevailing conjecture is that randomization is of *limited help* in the context of time-bounded and space-bounded algorithms. For example, it is conjectured that $\mathcal{BPP} = \mathcal{P}$ and $\mathcal{BPL} = \mathcal{L}$. Note that such conjectures do not rule out the possibility that randomization is helpful also in these contexts, it merely says that this help is limited. For example, it may be the case that any quadratic-time randomized algorithm can be emulated by a cubic time deterministic algorithm, but not by a quadratic time deterministic algorithm.

On the study of $\mathcal{BPP}$. The conjecture $\mathcal{BPP} = \mathcal{P}$ is referred to as a full derandomization of $\mathcal{BPP}$, and can be shown to hold under some reasonable intractability assumptions. This result (and related ones) will be presented in Section 8.3. In the current chapter, we only presented uncoditional results regarding $\mathcal{BPP}$ like $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$ and $\mathcal{BPP} \subseteq \mathcal{PH}$. Our presentation of Theorem 6.9 follows the proof idea of Lautemann [144]. A different proof technique, which yields a weaker result but found more applications (see, e.g., Theorem 6.27 and [107]), was presented (independently) by Sipser [199].

On the role of promise problems. In addition to their use in the formulation of Theorem 6.9, promise problems allow for establishing time hierarchy theorems (as in §4.2.1.1) for randomized computation (see Exercise 6.14). We mention that such results are not known for the corresponding classes of standard decision problems. The technical difficulty is that we do not know how to enumerate probabilistic machines that utilize a non-trivial probabilistic decision rule.

On the feasibility of randomized computation. Different perspectives on this question are offered by Chapter 8 and Appendix D.4. Specifically, as advocated in Chapter 8, generating uniformly distributed bit sequences is not really necessary for implementing randomized algorithms; it suffices to generate sequences that look as if they are uniformly distributed. In many cases this leads to reducing the number of coin tosses in such implementations, and at times even to a full (efficient) derandomization (see Sections 8.3 and 8.4). A less radical approach is presented in Appendix D.4, which deals with the task of extracting almost uniformly distributed bit sequences from sources of weak randomness. Needless to say, these two approaches are complimentary and can be combined.

Counting problems

The counting class $\#\mathcal{P}$ was introduced by Valiant [220], who proved that computing the permanent of 0/1-matrices is $\#\mathcal{P}$-complete (i.e., Theorem 6.20). Interestingly, like in the case of Cook's introduction of NP-completeness [55], Valiant's motivation was determining the complexity of a specific problem (i.e., the permanent).

Our presentation of Theorem 6.20 is based both on Valiant's paper [220] and on subsequent studies (most notably [29]). Specifically, the high-level structure of the reduction presented in Proposition 6.21 as well as the "structured" design of the clause gadget is taken from [220], whereas the Deus Ex Machina gadget presented in Figure 6.3 is based on [29]. The proof of Proposition 6.22 is also based on [29] (with some variants). Turning back to the design of clause gadgets we regret not being able to cite and/or use a systematic study of this design problem.

As noted in the main text, we decided not to present a proof of Toda's Theorem [212], which asserts that every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$ (i.e., Theorem 6.16). A proof of a related result appears in Section F.1 (implying that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via probabilistic polynomial-time reductions). Alternative proofs can be found in [130, 204, 212].

Approximate counting and related problems. The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [206], following an idea of Sipser [199]. Our exposition, however, follows further developments in the area. The randomized reduction of $\mathcal{NP}$ to problems of unique solutions was discovered by Valiant and Vazirani [222]. Again, our exposition is a bit different.

The connection between approximate counting and uniform generation (presented in §6.2.4.1) was discovered by Jerrum, Valiant, and Vazirani [128], and turned out to be very useful in the design of algorithms (e.g., in the "Markov Chain approach" (see [161, Sec. 11.3.1])). The direct procedure for uniform generation (presented in §6.2.4.2) is taken from [26].

In continuation to §6.2.2.1, which is based on [133], we refer the interested reader to [127], which presents a probabilistic polynomial-time algorithm for approximating the permanent of non-negative matrices. This fascinating algorithm is based on the fact that knowing (approximately) certain parameters of a non-negative matrix M allows to approximate the same parameters for a matrix M', provided that M and M' are sufficiently similar. Specifically, M and M' may differ only

on a single entry, and the ratio of the corresponding values must be sufficiently close to one. Needless to say, the actual observation (is not generic but rather) refers to specific parameters of the matrix, which include its permanent. Thus, given a matrix M for which we need to approximate the permanent, we consider a sequence of matrices $M_0, ..., M_t \approx M$ such that M_0 is the all 1's matrix (for which it is easy to evaluate the said parameters), and each M_{i+1} is obtained from M_i by reducing some adequate entry by a factor sufficiently close to one. This process of (polynomially many) gradual changes, allows to transform the dummy matrix M_0 into a matrix M_t that is very close to M (and hence has a permanent that is very close to the permanent of M). Thus, approximately obtaining the parameters of M_t allows to approximate the permanent of M.

Finally, we note that Section 10.1.1 provides a treatment of a different type of approximation problems. Specifically, when given an instance x (for a search problem R), rather than seeking an approximation of the number of solutions (i.e., $\#R(x)$), one seeks an approximation of the value of the best solution (i.e., best $y \in R(x)$), where the value of a solution is defined by an auxiliary function.

Exercises

Exercise 6.1 Show that if a search (resp., decision) problem can be solved by a probabilistic polynomial-time algorithm having zero failure probability, then the problem can be solve by a deterministic polynomial-time algorithm.

(Hint: replace the internal coin tosses by a fixed outcome that is easy to generate deterministically (e.g., the all-zero sequence).)

Exercise 6.2 (randomized reductions) In continuation to the definitions presented at the beginning of Section 6.1, prove the following:

1. If a problem Π is probabilistic polynomial-time reducible to a problem that is solvable in probabilistic polynomial-time then Π is solvable in probabilistic polynomial-time, where by solving we mean solving correctly except with negligible probability.

   Warning: Recall that in the case that Π' is a search problem, we required that on input x the solver provides a correct solution with probability at least $1 - \mu(|x|)$, but we did not require that it always returns the same solution.

   (Hint: without loss of generality, the reduction does not make the same query twice.)

2. Prove that probabilistic polynomial-time reductions are transitive.

3. Prove that randomized Karp-reductions are transitive and that they yield a special case of probabilistic polynomial-time reductions.

Define one-sided error and zero-sided error randomized (Karp and Cook) reductions, and consider the foregoing items when applied to them. Note that the implications for the case of one-sided error are somewhat subtle.

Exercise 6.3 (on the definition of probabilistically solving a search problem) In continuation to the discussion at the beginning of Section 6.1.1, suppose that for some probabilistic polynomial-time algorithm A and a positive polynomial p the following holds: for every $x \in S_R \stackrel{\rm def}{=} \{z : R(z) \neq \emptyset\}$ there exists $y \in R(x)$ such that $\Pr[A(x) = y] > 0.5 + (1/p(|x|))$, whereas for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 0.5 + (1/p(|x|))$.

1. Show that there exists a probabilistic polynomial-time algorithm that solves the search problem of R with negligible error probability.

   (Hint: See Exercise 6.4 for a related procedure.)

2. Reflect on the need to require that one (correct) solution occurs with probability greater than $0.5+(1/p(|x|))$. Specifically, what can we do if it is only guaranteed that for every $x \in S_R$ it holds that $\Pr[A(x) \in R(x)] > 0.5 + (1/p(|x|))$ (and for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 0.5 + (1/p(|x|)))$?

Note that R is not necessarily in $\mathcal{PC}$. Indeed, in the case that $R \in \mathcal{PC}$ we can eliminate the error probability for every $x \notin S_R$, and perform error-reduction as in $\mathcal{RP}$.

Exercise 6.4 (error-reduction for $\mathcal{BPP}$) For $\varepsilon : \mathbb{N} \to [0, 1]$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε. Prove the following two claims:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}$ equals $\mathcal{BPP}_\varepsilon$.

Formulate a corresponding version for the setting of search problem. Specifically, for every input that has a solution, consider the probability that a specific solution is output.

Guideline: Given an algorithm A for the syntactically weaker class, consider an algorithm A' that on input x invokes A on x for $t(|x|)$ times, and rules by majority. For Part 1 set $t(n) = O(p(n)^2)$ and apply Chebyshev's Inequality. For Part 2 set $t(n) = O(p(n))$ and apply the Chernoff Bound.

Exercise 6.5 (error-reduction for $\mathcal{RP}$) For $\rho : \mathbb{N} \to [0, 1]$, we define the class of decision problem $\mathcal{RP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\Pr[A(x) = 1] \geq \rho(|x|)$ and for every $x \notin S$ it holds that $\Pr[A(x) = 0] = 1$. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{RP}_{1/p}$ equals $\mathcal{RP}$.

2. For every positive polynomial p, the class $\mathcal{RP}$ equals $\mathcal{RP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

(Hint: The one-sided error allows using an "or-rule" (rather than a "majority-rule") for the decision.)

Exercise 6.6 (error-reduction for $\mathcal{ZPP}$) For $\rho : \mathbb{N} \to [0,1]$, we define the class of decision problem $\mathcal{ZPP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\mathsf{Pr}[A(x) = \chi_S(x)] \geq \rho(|x|)$ and $\mathsf{Pr}[A(x) \in \{\chi_S(x), \perp\}] = 1$, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{ZPP}_{1/p}$ equals $\mathcal{ZPP}$.
2. For every positive polynomial p, the class $\mathcal{ZPP}$ equals $\mathcal{ZPP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

Exercise 6.7 (an alternative definition of $\mathcal{ZPP}$) We say that the decision problem S is solvable in expected probabilistic polynomial-time if there exists a randomized algorithm A and a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $\mathsf{Pr}[A(x) = \chi_S(x)] = 1$ and the expected number of steps taken by $A(x)$ is at most $p(|x|)$. Prove that $S \in \mathcal{ZPP}$ if and only if S is solvable in expected probabilistic polynomial-time.

Guideline: Repeatedly invoking a ZPP algorithm until it yields an output other than $\perp$, results in an expected probabilistic polynomial-time solver. On the other hand, truncating runs of an expected probabilistic polynomial-time algorithm once they exceed twice the expected number of steps (and outputting $\perp$ on such runs), we obtain a ZPP algorithm.

Exercise 6.8 Prove that for every $S \in \mathcal{NP}$ there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\mathsf{Pr}[A(x) = 1] > 0$ and for every $x \notin S$ it holds that $\mathsf{Pr}[A(x) = 0] = 1$. That is, A has error probability at most $1 - \exp(-\text{poly}(|x|))$ on yes-instances but never errs on no-instances. Thus, $\mathcal{NP}$ may be fictitiously viewed as having a huge one-sided error probability.

Exercise 6.9 Let $\mathcal{BPP}$ and co$\mathcal{RP}$ be classes of promise problems (as in Theorem 6.9).

1. Prove that every problem in $\mathcal{BPP}$ is reducible to the set $\{1\} \in \mathcal{P}$ by a *two-sided error* randomized Karp-reduction.

   (Hint: Such a reduction may effectively decide membership in any set in $\mathcal{BPP}$.)

2. Prove that if a set S is Karp-reducible to $\mathcal{RP}$ (resp., co$\mathcal{RP}$) via a deterministic reduction then $S \in \mathcal{RP}$ (resp., $S \in$ co$\mathcal{RP}$).

Exercise 6.10 (randomness-efficient error-reductions) Note that standard error-reduction (as in Exercise 6.4) yields error probability δ at the cost of increasing the randomness complexity by a *factor* of $O(\log(1/\delta))$. Using the randomness-efficient error-reductions outlined in §D.4.1.3, show that error probability δ can be obtained at the cost of increasing the randomness complexity by a constant factor and an *additive term* of $1.5\log_2(1/\delta)$. Note that this allows satisfying the hypothesis made in the illustrative paragraph of the proof of Theorem 6.9.

Exercise 6.11 In continuation to the illustrative paragraph in the proof of Theorem 6.9, consider the promise problem $\Pi' = (\Pi'_{\rm yes}, \Pi'_{\rm no})$ such that $\Pi'_{\rm yes} = \{(x,r') : |r'|=p'(|x|) \wedge (\forall r'' \in \{0,1\}^{|r'|})\, A'(x,r'r'')=1\}$ and $\Pi'_{\rm no} = \{(x,r') : x \notin S\}$. Recall that for every x it holds that $\Pr_{r\in\{0,1\}^{2p'(|x|)}}[A'(x,r)\neq\chi_S(x)] < 2^{-(p'(|x|)+1)}$.

1. Show that mapping x to (x,r'), where r' is uniformly distributed in $\{0,1\}^{p'(|x|)}$, constitutes a one-sided error randomized Karp-reduction of S to Π'.

2. Show that Π' is in the promise problem class co$\mathcal{RP}$.

Exercise 6.12 (randomized versions of $\mathcal{NP}$) In continuation to Footnote 6, consider the following two variants of $\mathcal{MA}$ (which we consider the main randomized version of $\mathcal{NP}$).

1. $S \in \mathcal{MA}^{(1)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $\Pr[V(x,y)=1] \geq 1/2$, whereas for every $x \notin S$ and every y it holds that $\Pr[V(x,y)=0] = 1$.

2. $S \in \mathcal{MA}^{(2)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $\Pr[V(x,y)=1] \geq 2/3$, whereas for every $x \notin S$ and every y it holds that $\Pr[V(x,y)=0] \geq 2/3$.

Prove that $\mathcal{MA}^{(1)} = \mathcal{NP}$ whereas $\mathcal{MA}^{(2)} = \mathcal{MA}$.

Guideline: For the first part, note that a sequence of internal coin tosses that makes V accept (x,y) can be incorporated into y itself (yielding a standard NP-witness). For the second part, apply the ideas underlying the proof of Theorem 6.9, and note that an adequate sequence shifts (to be used by the verifier) can be incorporated in the single message sent by the prover.

Exercise 6.13 ($\mathcal{BPP} \subseteq \mathcal{ZPP}^{\mathcal{NP}}$) In continuation to the proof of Theorem 6.9, present a zero-error randomized reduction of $\mathcal{BPP}$ to $\mathcal{NP}$, where all classes are the standard classes of decision problems.

Exercise 6.14 (time hierarchy theorems for promise problem versions of BPTIME**)** Fixing a model of computation, let BPTIME(t) denote the class of promise problems that are solvable by a randomized algorithm of time complexity t that has a two-sided error probability at most 1/3. (The common definition refers only to decision problems.) Formulate and prove results analogous to Theorem 4.3 and Corollary 4.4.

Guideline: Analogously to the proof of Theorem 4.3, we construct a Boolean function f by associating with each admissible machine M an input x_M, and making sure that $\Pr[f(x_M) \neq M'(x)] \geq 2/3$, where $M'(x)$ denotes the emulation of $M(x)$ suspended after $t_1(|x|)$ steps. The key point is that f is a partial function (corresponding to a promise problem) that is defined only for machines (called admissible) that have two-sided error at most 1/3 (on every input). This restriction allows for a randomized computation of f with two-sided error probability at most 1/3 (on each input on which f is defined).

Exercise 6.15 (extracting square roots modulo a prime) Using the following guidelines, present a probabilistic polynomial-time algorithm that, on input a prime P and a quadratic residue $s \pmod P$, returns r such that $r^2 \equiv s \pmod P$.

1. Prove that if $P \equiv 3 \pmod 4$ then $s^{(P+1)/4} \bmod P$ is a square root of the quadratic residue $s \pmod P$.

2. Note that the procedure suggested in Item 1 relies on the ability to find an *odd* integer e such that $s^e \equiv 1 \pmod P$. Indeed, once such e is found, we may output $s^{(e+1)/2} \bmod P$. (In Item 1, we used $e = (P-1)/2$, which is odd since $P \equiv 3 \pmod 4$.)

   Show that it suffices to find an *odd* integer e together with a residue t and an *even* integer e' such that $s^e t^{e'} \equiv 1 \pmod P$, because $s \equiv s^{e+1} t^{e'} \equiv (s^{(e+1)/2} t^{e'/2})^2$.

3. Given a prime $P \equiv 1 \pmod 4$, a quadratic residue s, and any quadratic non-residue t (i.e., residue t such that $t^{(P-1)/2} \equiv -1 \pmod P$), show that e and e' as in Item 2 can be efficiently found.[19]

4. Prove that, for a prime P, with probability 1/2 a uniformly chosen $t \in \{1, ..., P\}$ satisfies $t^{(P-1)/2} \equiv -1 \pmod P$.

Note that randomization is used only in the last item, which in turn is used only for $P \equiv 1 \pmod 4$.

Exercise 6.16 Referring to the definition of arithmetic circuits (cf. Appendix B.3), show that the following decision problem is in co$\mathcal{RP}$: *Given a pair of circuits (C_1, C_2) of depth d over a field that has more than 2^{d+1} elements, determine whether the circuits compute the same polynomial.*

Guideline: Note that each of these circuits computes a polynomial of degree at most 2^d.

Exercise 6.17 (small-space randomized step-counter) As defined in Exercise 4.4, a step-counter is an algorithm that halts after issuing a number of "signals" as specified in its input, where these signals are defined as entering (and leaving) a designated state (of the algorithm). Recall that a step-counter may be run in parallel to another procedure in order to suspend the execution after a desired number of steps (of the other procedure) has elapsed. Note that there exists a simple deterministic machine that, on input n, halts after issuing n signals while using $O(1) + \log_2 n$ space (and $\widetilde{O}(n)$ time). The goal of this exercise is presenting a (randomized) step-counter that allows for many more signals while using the same amount of space. Specifically, present a (randomized) algorithm that, on input

[19] Write $(P-1)/2 = (2j+1) \cdot 2^{i_0}$, and note that $s^{(2j+1) \cdot 2^{i_0}} \equiv 1 \pmod P$. Assuming that for some $i' > i > 0$ and j' it holds that $s^{(2j+1) \cdot 2^i} t^{(2j'+1) \cdot 2^{i'}} \equiv 1 \pmod P$, show how to find $i'' > i-1$ and j'' such that $s^{(2j+1) \cdot 2^{i-1}} t^{(2j''+1) \cdot 2^{i''}} \equiv 1 \pmod P$. (Extra hint: $s^{(2j+1) \cdot 2^{i-1}} t^{(2j'+1) \cdot 2^{i'-1}} \equiv \pm 1 \pmod P$ and $t^{(2j+1) \cdot 2^{i_0}} \equiv -1 \pmod P$.) Thus, starting with $i = i_0$, we reach $i = 1$, at which point we have what we need.

n, uses $O(1) + \log_2 n$ space (and $\widetilde{O}(2^n)$ time) and halts after issuing an expected number of 2^n signals. Furthermore, prove that, with probability at least $1-2^{-k+1}$, this step-counter halts after issuing a number of signals that is between 2^{n-k} and 2^{n+k}.

Guideline: Repeat the following experiment till reaching success. Each trial consists of uniformly selecting n bits (i.e., tossing n unbiased coins), and is deemed successful if all bits turn out to equal the value 1 (i.e., all outcomes equal HEAD). Note that such a trial can be implemented by using space $O(1) + \log_2 n$ (mainly for implementing a standard counter for determining the number of bits). Thus, each trial is successful with probability 2^{-n}, and the expected number of trials is 2^n.

Exercise 6.18 (analysis of random walks on arbitrary undirected graphs) In order to complete the proof of Proposition 6.13, prove that if $\{u,v\}$ is an edge of the graph $G=(V,E)$ then $\mathrm{E}[X_{u,v}] \leq 2|E|$. Recall that, for a fixed graph, $X_{u,v}$ is a random variable representing the number of steps taken in a random walk that starts at the vertex u until the vertex v is first encountered.

Guideline: Let $Z_{u,v}(n)$ be a random variable counting the number of *minimal* paths from u to v that appear along a random walk of length n, where the walk starts at the stationary vertex distribution (which is well-defined assuming the graph is not bipartite, which in turn may be enforced by adding a self-loop). On one hand, $\mathsf{E}[X_{u,v} + X_{v,u}] = \lim_{n\to\infty}(n/\mathsf{E}[Z_{u,v}(n)])$, due to the memoryless property of the walk. On the other hand, letting $\chi_{v,u}(i) \stackrel{\rm def}{=} 1$ if the edge $\{u,v\}$ was traversed from v to u in the $i^{\rm th}$ step of such a random walk and $\chi_{v,u}(i) \stackrel{\rm def}{=} 0$ otherwise, we have $\sum_{i=1}^{n}\chi_{v,u}(i) \leq Z_{u,v}(n)+1$ and $\mathrm{E}[\chi_{v,u}(i)] = 1/2|E|$ (because, in each step, each directed edge appears on the walk with equal probability). It follows that $\mathrm{E}[X_{u,v}] < 2|E|$.

Exercise 6.19 (the class $\mathcal{PP} \supseteq \mathcal{BPP}$ and its relation to $\#\mathcal{P}$) In contrast to $\mathcal{BPP}$, which refers to useful probabilistic polynomial-time algorithms, the class $\mathcal{PP}$ does not capture such algorithms but is rather closely related to $\#\mathcal{P}$. A decision problem S is in $\mathcal{PP}$ if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $x \in S$ if and only if $\mathsf{Pr}[A(x)=1] > 1/2$. Note that $\mathcal{BPP} \subseteq \mathcal{PP}$. Prove that $\mathcal{PP}$ is Cook-reducible to $\#\mathcal{P}$ and vise versa.

Guideline: For $S \in \mathcal{PP}$ (by virtue of the algorithm A), consider the relation R such that $(x,r) \in R$ if and only if A accepts the input x when using the random-input $r \in \{0,1\}^{p(|x|)}$, where p is a suitable polynomial. Thus, $x \in S$ if and only if $|R(x)| > 2^{p(|x|)-1}$, which in turn can de determined by querying the counting function of R. To reduce $f \in \#\mathcal{P}$ to $\mathcal{PP}$, consider the relation $R \in \mathcal{PC}$ that is counted by f (i.e., $f(x) = |R(x)|$) and the decision problem S_f as defined in Proposition 6.15. Let p be the polynomial specifying the length of solutions for R (i.e., $(x,y) \in R$ implies $|y| = p(|x|)$), and consider the algorithm A' that on input (x,N) proceeds as follows: With probability 1/2, it uniformly selects $y \in \{0,1\}^{p(|x|)}$ and accepts if and only if $(x,y) \in R$, and otherwise (i.e., in the other case) it accepts with probability $\frac{2^{p(|x|)}-N+0.5}{2^{p(|x|)}}$. Prove that $(x,N) \in S_f$ if and only if $\mathsf{Pr}[A'(x)=1] > 1/2$.

Exercise 6.20 (enumeration problems) For any binary relation R, define the enumeration problem of R as a function $f_R : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \cup \{\perp\}$ such that $f_R(x,i)$ equals the ith element in $|R(x)|$ if $|R(x)| \geq i$ and $f_R(x,i) = \perp$ otherwise. The above definition refers to the standard lexicographic order on strings, but any other efficient order of strings will do.[20]

1. Prove that, for any polynomially bounded R, computing $\#R$ is reducible to computing f_R.

2. Prove that, for any $R \in \mathcal{PC}$, computing f_R is reducible to some problem in $\#\mathcal{P}$.

   Guideline: Consider the binary relation $R' = \{(\langle x,b\rangle, y) : (x,y) \in R \wedge y \leq b\}$, and show that f_R is reducible to $\#R'$. (Extra hint: Note that $f_R(x,i) = y$ if and only if $|R'(\langle x,y\rangle)| = i$ and for every $y' < y$ it holds that $|R'(\langle x,y'\rangle)| < i$.)

Exercise 6.21 (artificial $\#\mathcal{P}$-complete problems) Show that there exists a relation $R \in \mathcal{PC}$ such that $\#R$ is $\#\mathcal{P}$-complete and $S_R = \{0,1\}^*$. Furthermore, prove that for every $R \in \mathcal{PC}$ there exists $R' \in \mathcal{PF}$ such that for every x it holds that $\#R'(x) = \#R(x) + 1$. Note that Theorem 6.19 follows by starting with a relation that satisfies Theorem 6.18.

Exercise 6.22 (computing the permanent of integer matrices) Prove that computing the permanent of matrices with 0/1-entries is computationally equivalent to computing the number of perfect matchings in bipartite graphs.
(Hint: Given a bipartite graph $G = ((X,Y),E)$, consider the matrix M representing the edges between X and Y (i.e., the (i,j)-entry in M is 1 if the ith vertex of X is connected to the jth entry of Y), and note that only perfect matchings in G contribute to the permanent of M.)

Exercise 6.23 (computing the permanent modulo 3) Combining Proposition 6.21 and Theorem 6.29, prove that for every integer $n > 1$ that is relatively prime to c, computing the permanent modulo n is NP-hard under randomized reductions.[21] Since Proposition 6.21 holds for $c = 2^{10}$, hardness holds for every odd integer $n > 1$.

Guideline: Apply the reduction of Proposition 6.21 to the promise problem of deciding whether a 3CNF formula has a unique satisfiable assignment or is unsatisfiable. Use the fact that n does not divide any power of c.

Exercise 6.24 (negative values in Proposition 6.21) Assuming $\mathcal{P} \neq \mathcal{NP}$, prove that Proposition 6.21 cannot hold for a set I containing only non-negative integers. Note that the claim holds even if the set I is not finite (and even if I is the set of all non-negative integers).

---

[20] An order of strings is a 1-1 and onto mapping μ from the natural numbers to the set of all strings. Such order is called efficient if both μ and its inverse are efficiently computable. The standard lexicographic order satisfies $\mu(i) = y$ if the (compact) binary expansion of i equals $1y$; that is $\mu(1) = \lambda$, $\mu(2) = 0$, $\mu(3) = 1$, $\mu(4) = 00$, etc.

[21] Actually, a sufficient condition is that n does not divide any power of c. Thus (referring to $c = 2^{10}$), hardness holds for every integer $n > 1$ that is not a power of 2. On the other hand, for any fixed $n = 2^e$, the permanent modulo n can be computed in polynomial-time [220, Thm. 3].

Guideline: A reduction as in Proposition 6.21 yields a Karp-reduction of 3SAT to deciding whether the permanent of a matrix with entries in I is non-zero. Note that the permanent of a *non-negative* matrix is non-zero if and only if the corresponding bipartite graph has a perfect matching.

Exercise 6.25 (high-level analysis of the permanent reduction) Establish the correctness of the high-level reduction presented in the proof of Proposition 6.21. That is, show that if the clause gadget satisfies the three conditions postulated in the said proof, then each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ whereas unsatisfying assignments have no contribution.

Guideline: Cluster the cycle covers of G_ϕ according to the set of track edges that they use (i.e., the edges of the cycle cover that belong to the various tracks). (Note the correspondence between these edges and the external edges used in the definition of the gadget's properties.) Using the postulated conditions (regarding the clause gadget) prove that, for each such set T of track edges, if the sum of the weights of all cycle covers that use the track edges T is non-zero then the following hold:

1. The intersection of T with the set of track edges incident at each specific clause gadget is non-empty. Furthermore, if this set contains an incoming edge (resp., outgoing edge) of some entry-vertex (resp., exit-vertex) then it also contains an outgoing edge (resp., incoming edge) of the corresponding exit-vertex (resp., entry-vertex).
2. If T contains an edge that belongs to some track then it contains all edges of this track. It follows that, for each variable x, the set T contains the edges of a single track associated with x.
3. The tracks "picked" by T correspond to a single truth assignment to the variables of ϕ, and this assignment satisfies ϕ (because, for each clause, T contains an external edge that corresponds to a literal that satisfies this clause).

It follows that each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ.

Exercise 6.26 (analysis of the implementation of the clause gadget) Establish the correctness of the implementation of the clause gadget presented in the proof of Proposition 6.21. That is, show that if the box satisfy the three conditions postulated in the said proof, then the clause gadget of Figure 6.4 satisfies the conditions postulated for it.

Guideline: Cluster the cycle covers of a gadget according to the set of non-box edges that they use, where non-box edges are the edges shown in Figure 6.4. Using the postulated conditions (regarding the box) prove that, for each set S of non-box edges, if the sum of the weights of all cycle covers that use the non-box edges S is non-zero then the following hold:

1. The intersection of S with the set of edges incident at each box must contain two (non-selfloop) edges, one incident at each of the box's terminals. Needless to say, one edge is incoming and the other outgoing. Referring to the six edges that connects one of the six designated vertices (of the gadget) with the corresponding box terminals as connectives, note that if S contains a connective incident at the terminal of some box then it must also contain the connective incident at the other terminal. In such a case, we say that this box is picked by S,

2. Each of the three (literal-designated) boxes that is not picked by S is "traversed" from left to right (i.e., the cycle cover contains an incoming edge of the left terminal and an outgoing edge of the right terminal). Thus, the set S must contain a connective, because otherwise no directed cycle may cover the leftmost vertex shown in Figure 6.4. That is, S must pick some box.
3. The set S is fully determined by the non-empty set of boxes that it picks.

The postulated properties of the clause gadget follow, with $c = b^5$.

Exercise 6.27 (analysis of the design of a box for the clause gadget) Prove that the 4-by-4 matrix presented in Eq. (6.4) satisfies the properties postulated for the "box" used in the second part of the proof of Proposition 6.21. In particular:

1. Show a correspondence between the conditions required of the box and conditions regarding the value of the permanent of certain sub-matrices of the adjacency matrix of the graph.

   (Hint: For example, show that the first condition correspond to requiring that the value of the permanent of the entire matrix equals zero. The second condition refers to sub-matrices obtained by omitting either the first row and fourth column or the fourth row and first column.)

2. Verify that the matrix in Eq. (6.4) satisfies the aforementioned conditions (regarding the value of the permanent of certain sub-matrices).

Prove that no 3-by-3 matrix (and thus also no 2-by-2 matrix) can satisfy the aforementioned conditions.

Exercise 6.28 (error reduction for approximate counting) Show that the error probability δ in Section 6.24 can be reduced from $1/3$ (or even $(1/2)+(1/\text{poly}(|x|))$ to $\exp(-\text{poly}(|x|))$.

Guideline: Invoke the weaker procedure for an adequate number of times and take the *median* value among the values obtained in these invocations.

Exercise 6.29 (strong approximation for some $\#\mathcal{P}$-complete problems) Show that there exists $\#\mathcal{P}$-complete problems (albeit unnatural ones) for which an $(\varepsilon, 0)$-approximation can be found by a (deterministic) polynomial-time algorithm. Furthermore, the running-time depends polynomially on $1/\varepsilon$.

Guideline: Combine any $\#\mathcal{P}$-complete problem referring to some $R_1 \in \mathcal{PC}$ with a trivial counting problem (e.g., the counting problem associated with the trivial relation $R_2 = \cup_{n\in\mathbb{N}}\{(x,y) : x, y \in \{0,1\}^n\}$). Show that, without loss of generality, it holds that $\#R_1(x) \leq 2^{|x|/2}$. Prove that the counting problem of $R = \{(x, 1y) : (x,y) \in R_1\} \cup \{(x, 0y) : (x,y) \in R_2\}$ is $\#\mathcal{P}$-complete (by reducing from $\#R_1$). Present a deterministic algorithm that, on input x and $\varepsilon > 0$, outputs an $(\varepsilon, 0)$-approximation of $\#R(x)$ in time $\text{poly}(|x|/\varepsilon)$ (Extra hint: distinguish between $\varepsilon \geq 2^{-|x|/2}$ and $\varepsilon < 2^{-|x|/2}$).

Exercise 6.30 (relative approximation for DNF satisfaction) Referring to the text of §6.2.2.1, prove the following claims.

1. Both assumptions regarding the general setting hold in case $S_i = C_i^{-1}(1)$, where $C_i^{-1}(1)$ denotes the set of truth assignments that satisfy the conjunction C_i.

   Guideline: In establishing the second assumption note that it reduces to the conjunction of the following two assumptions:

   (a) Given i, one can efficiently generate a uniformly distributed element of S_i. Actually, generating a distribution that is almost uniform over S_i suffices.

   (b) Given i and x, one can efficiently determine whether $x \in S_i$.

2. Prove Proposition 6.26, relating to details such as the error probability in an implementation of Construction 6.25.

3. Note that Construction 6.25 does not require exact computation of $|S_i|$. Analyze the output distribution in the case that we can only approximate $|S_i|$ up-to a factor of $1 \pm \varepsilon'$.

Exercise 6.31 (reducing the relative deviation in approximate counting) Prove that, for any $R \in \mathcal{PC}$ and every polynomial p and constant $\delta < 0.5$, there exists $R' \in \mathcal{PC}$ such that $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R'$. Furthermore, for any $F(n) = \exp(\text{poly}(n))$, prove that there exists $R'' \in \mathcal{PC}$ such that $(1/p, \delta)$-approximation for $\#R$ is reducible to approximating $\#R''$ to within a factor of F with error probability δ.

Guideline (for the main part): For $t(n) = \Theta(p(n))$, define R' such that $(y_1, ..., y_{t(|x|)}) \in R'(x)$ if and only if $(\forall i)\,(x, y_i) \in R$. Note that $|R(x)| = |R'(x)|^{1/t(|x|)}$, and thus if $a = (1 \pm (1/2)) \cdot |R'(x)|$ then $a^{1/t(|x|)} = (1 \pm (1/2))^{1/t(|x|)} \cdot |R(x)|$.

Exercise 6.32 (deviation reduction in approximate counting, cont.) In continuation to Exercise 6.31, prove that if R is NP-complete via parsimonious reductions then, for every positive polynomial p and constant $\delta < 0.5$, the problem of $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R$.
(Hint: Compose the reduction (to the problem of $(1/2, \delta)$-approximation for $\#R'$) provided in Exercise 6.31 with the parsimonious reduction of $\#R'$ to $\#R$.)
Prove that, for every function F' such that $F'(n) = \exp(n^{o(1)})$, we can also reduce the aforementioned problems to the problem of approximating $\#R$ to within a factor of F' with error probability δ.

Guideline: Using R'' as in Exercise 6.31, we encounter a technical difficulty. The issue is that the composition of the ("amplifying") reduction of $\#R$ to $\#R''$ with the parsimonious reduction of $\#R''$ to $\#R$ may increase the length of the instance. Indeed, the length of the new instance is polynomial in the length of the original instance, but this polynomial may depend on R'', which in turn depends on F'. Thus, we cannot use $F'(n) = \exp(n^{1/O(1)})$ but $F'(n) = \exp(n^{o(1)})$ is fine.

Exercise 6.33 Referring to the procedure in the proof Theorem 6.27, show how to use an NP-oracle in order to determine whether the number of solutions that "pass a random sieve" is greater than t. You are allowed queries of length polynomial in the length of x, h and in the size of t.

(Hint: Consider the set $S'_{R,H} \stackrel{\rm def}{=} \{(x,i,h,1^t) : \exists y_1,...,y_t \text{ s.t. } \psi'(x,h,y_1,...,y_t)\}$, where $\psi'(x,h,y_1,...,y_t)$ holds if and only if the y_j are different and for every j it holds that $(x,y_j)\in R \wedge h(y_j)=0^i$.)

Exercise 6.34 (parsimonious reductions and Theorem 6.29) Demonstrate the importance of parsimonious reductions in Theorem 6.29 by proving the following:

1. There exists a search problem $R \in \mathcal{PC}$ such that every problem in $\mathcal{PC}$ is reducible to R (by a non-parsimonious reduction) and still the the promise problem $(\texttt{US}_R, \overline{S}_R)$ is decidable in polynomial-time.

   Guideline: Consider the following artificial witness relation R for SAT in which $(\phi, \sigma\tau) \in R$ if $\sigma \in \{0,1\}$ and τ satisfies ϕ. Note that the standard witness relation of SAT is reducible to R, but this reduction is not parsimonious. Also note that $\texttt{US}_R = \emptyset$ and thus $(\texttt{US}_R, \overline{S}_R)$ is trivial.

2. There exists a search problem $R \in \mathcal{PC}$ such that $\#R$ is $\#\mathcal{P}$-complete and still the the promise problem $(\texttt{US}_R, \overline{S}_R)$ is decidable in polynomial-time.

   Guideline: Just use the relation suggested in the guideline to Part 1. An alternative proof relies on Theorem 6.20 and on the fact that it is easy to decide $(\texttt{US}_R, \overline{S}_R)$ when R is the corresponding perfect matching relation (by computing the determinant).

Exercise 6.35 Prove that SAT is randomly reducible to deciding unique solution for SAT, *without using the fact that* SAT *is NP-complete via parsimonious reductions.*

Guideline: Follow the proof of Theorem 6.29, while using the family of pairwise independent hashing functions provided in Construction D.3 (or in Eq. (8.18)). Note that, in this case, the condition $(\tau \in R_{\texttt{SAT}}(\phi)) \wedge (h(\tau)=0^i)$ can be directly encoded as a CNF formula. That is, consider the formula ϕ_h such that $\phi_h(z) \stackrel{\rm def}{=} \phi(z) \wedge (h(z)=0^i)$, and note that $h(z)=0^i$ can be written as the conjunction of i clauses, where each clause is a CNF that is logically equivalent to the parity of some of the bits of z (where the identity of these bits is determined by h).

Exercise 6.36 (an alternative procedure for approximate counting) Adapt Step 1 of Construction 6.32 so to obtain an approximate counting procedure for $\#R$.

Guideline: For $m = 0,1,...\ell$, the procedure invokes Step 1 of Construction 6.32 until a negative answer is obtained, and outputs 2^m for the current value of m. For $|R(x)| > 128\ell$, this yields a constant factor approximation of $|R(x)|$. In fact, we can obtain a better estimate by making additional queries at iteration m (i.e., queries of the form $(x,h,1^i)$ for $i = 16\ell, ..., 128\ell$). The case $|R(x)| \leq 128\ell$ can be treated by using Step 2 of Construction 6.32, in which case we obtain an exact count.

Exercise 6.37 Let R be an arbitrary $\mathcal{PC}$-complete search problem. Show that approximate counting and uniform generation for R can be randomly reduced to deciding membership in S_R, where by approximate counting we mean a $(1-(1/p)$-approximation for any polynomial p.

Guideline: Note that Construction 6.32 yields such procedures (see also Exercise 6.36), except that they make oracle calls to some other set in $\mathcal{NP}$. Using the NP-completeness of S_R, we are done.