Chapter 5 : Randomized computation

Probabilistic or randomized computation often provides practical methods to solve certain computational problems. In order to define probabilistic complexity classes we augment Turing machines with an added capability, namely that of flipping (unbiased) coins. Based on the outcome of each coin toss, the machine makes one of two allowed next moves. In this sense randomization proposes a way to *implement* nondeterminism.

It is not immediate at the outset how such nondeterministic choices based on coin tosses could help speeding up computations. Think of a probabilistic algorithm for a language $L \in \text{NP}$, for which *many* random certificates are available. If $\alpha \in L$, trying a few random candidates may eventually lead to the finding of a certificate for α. On the other hand, if $\alpha \notin \text{NP}$, no random choices would lead to the discovery of a certificate. However, if we took a definite deterministic order of investigating certificates, we might end up generating a huge (exponential) number of certificates. This partially explains situations in which randomization may help.

5.1 Randomized complexity classes

5.1 Definition A probabilistic Turing machine is a nondeterministic Turing machine with the property that for *every* (non-halting) configuration of the machine, there exist exactly two possible next moves. We assign a probability of $1/2^k$ for a branch of computation which makes exactly k nondeterministic moves. The probability that the machine accepts an input string α is the sum of the probabilities of all accepting branches on this input, whereas the probability that the machine rejects is the sum of the probabilities of all rejecting branches. It is easy to see that these two probabilities sum up to 1. If the probabilistic Turing machine runs in time polynomially bounded by the input size (in the sense of a nondeterministic computation), we call the machine a probabilistic polynomial-time Turing machine or a PPT for short.

Note that forcing exactly two choices in each non-halting state is not a loss of generality. If an NTM N makes only one choice from some configuration, we can replace that transition by two identical transitions. If N makes t possible choices for some $t > 2$, we can replace this nondeterministic transition by a series of two-way transitions each of which guesses a bit of the choice number.

This formalism prescribes a way to implement nondeterminism — replace a nondeterministic move by a deterministic move based on the outcome of a coin toss. However, we may be unlucky to meet a bad sequence of coin tosses so as to arrive at an erroneous decision. Let L be a language having a probabilistic algorithm N. If $\alpha \in L$, we may end up in a rejecting branch and incorrectly conclude that $\alpha \notin L$. Moreover, if $\alpha \notin L$, there may be some accepting branches and an unlucky sequence of tosses may reveal such an accepting branch. For N to be useful in practice, it should run in poly-time (i.e., it should be a PPT) and the error probabilities should be small. Suppose that for $0 \leqslant \epsilon_1, \epsilon_2 \leqslant 1/2$ we have:

if $\alpha \in L$, then $\Pr(N \text{ accepts } \alpha) \geqslant 1 - \epsilon_1$, and
if $\alpha \notin L$, then $\Pr(N \text{ rejects } \alpha) \geqslant 1 - \epsilon_2$,

where these two probabilities are defined as in Definition 5.1. In this case we say that N decides L with an error probability of ϵ_1, ϵ_2. The error probabilities ϵ_1, ϵ_2 may be a function of the input length n, say 2^{-n}. In this case N accepts L with an exponentially small probability of error. For the time being, let us concentrate on some constant values of ϵ_1, ϵ_2 and define some useful randomized complexity classes. We later see how we can use these classes to obtain probabilistic poly-time algorithms with an exponentially small probability of error. Table 5.1 summarizes these classes and the corresponding values for ϵ_1, ϵ_2.

Table 5.1: Randomized complexity classes

Class	ϵ_1	ϵ_2
RP	$\leqslant 1/2$	0
coRP	0	$\leqslant 1/2$
ZPP	0	0
PP	$< 1/2$	$< 1/2$
BPP	$\leqslant 1/3$	$\leqslant 1/3$

5.2 Definition (1) [Randomized polynomial time (RP)]

We say that $L \in \mathrm{RP}$, if and only if there exists a PPT N such that

if $\alpha \in L$, then $\Pr(N \text{ accepts } \alpha) \geqslant 1/2$, and
if $\alpha \notin L$, then $\Pr(N \text{ rejects } \alpha) = 1$.

(2) [coRP]

Define $\mathrm{coRP} := \{L \mid \bar{L} \in \mathrm{RP}\}$. In other words, $L \in \mathrm{coRP}$, if and only if there exists a PPT N such that

if $\alpha \in L$, then $\Pr(N \text{ accepts } \alpha) = 1$, and
if $\alpha \notin L$, then $\Pr(N \text{ rejects } \alpha) \geqslant 1/2$.

(3) [Zero probability polynomial time (ZPP)]

$\mathrm{ZPP} := \mathrm{RP} \cap \mathrm{coRP}$.

(4) [Probabilistic polynomial time (PP)]

We say that $L \in \mathrm{PP}$, if and only if there exists a PPT N such that

if $\alpha \in L$, then $\Pr(N \text{ accepts } \alpha) > 1/2$, and
if $\alpha \notin L$, then $\Pr(N \text{ rejects } \alpha) > 1/2$.

We say that PP accepts and rejects *by majority*.

(5) [Bounded probability polynomial time (BPP)]

We say that $L \in \mathrm{BPP}$, if and only if there exists a PPT N such that

if $\alpha \in L$, then $\Pr(N \text{ accepts } \alpha) \geqslant 2/3$, and
if $\alpha \notin L$, then $\Pr(N \text{ rejects } \alpha) \geqslant 2/3$.

We say that BPP accepts and rejects *by clear majority*.

RP and coRP algorithms have *one-sided* probability of error and are often called Monte Carlo algorithms. Let L be a language decided by a randomized poly-time algorithm N. Suppose that we run N t times on some input α of length n. Assume also that the runs are independent in the sense that each coin toss N makes is modeled by an independent random variable. If $\alpha \notin L$, the machine rejects in each run. However, if $\alpha \in L$, N can still reject on every occasion, but this time with a probability $\leqslant (1-1/2)^t = 1/2^t$. Making t sufficiently large we can make this probability vanishingly small. For example, taking $t = 100$ gives an error probability of $1/2^{100}$ which is arguably much smaller than the probability of a hardware failure or even that of a meteorite hitting the computer during its computation, although theoretically, no finite value of t, however large, can give us 100% confidence about the correctness of the machine's decision.

If t is a polynomial $p(n)$ of the input size n, the error probability is exponentially small, namely $2^{-p(n)}$. Moreover, $p(n)$ runs of N on α does not alter the polynomial nature of the running time (though the degree of the polynomial increases). In other words, in the definition of RP we may take ϵ_1 to be any constant strictly between 0 and 1.

A language $L \in \mathrm{ZPP} = \mathrm{RP} \cap \mathrm{coRP}$ has two (poly-time) Monte Carlo algorithms: N that never errs when $\alpha \notin L$ and $\bar{N}$ that never errs when $\alpha \in L$. We run the two algorithms repeatedly in parallel. If N accepts α, we know *for sure* that $\alpha \in L$. (N is bound to reject any string outside α.) On the other hand, if $\bar{N}$ rejects α, it is certain that $\alpha \notin L$. However, if N rejects and $\bar{N}$ accepts, we can't definitely conclude about the membership of α in L, and this event happens with a probability $\leqslant 1/2$.

We continue the parallel simulation, until we arrive at the definitely correct answer. We may be so unlucky as to have rejection by N and acceptance by $\bar{N}$ for a huge (say, exponential) number of times. The probability that this happens for t runs is $\leqslant 1/2^t$ which decreases exponentially with t. Thus we *expect* to get the correct answer after a few (a constant number of) runs of N and $\bar{N}$. In other words, the parallel simulation halts with the correct answer in expected polynomial time. Such algorithms are called Las Vegas algorithms.

One may also view a Las Vegas algorithm for L as one that provides three possible answers: 'accept', 'reject' and 'don't know'. It never accepts a string outside L and never rejects one inside L. A 'don't know' output comes with a probability $\leqslant 1/2$.

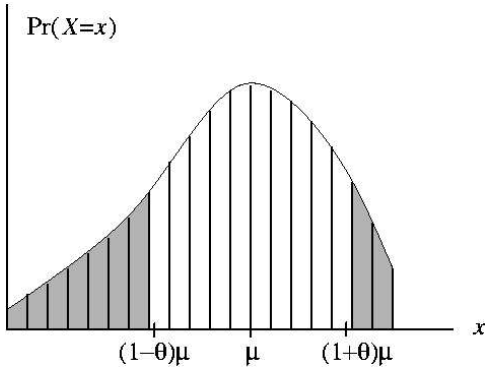
The classes PP and BPP correspond to two-sided error probabilities. These algorithms can still be effectively exploited in the following sense. Repeat an algorithm N for a language L in PP or BPP some number of times and take a decision 'by majority', i.e., if more runs accept than reject, then accept the input, whereas if more runs reject, reject the input. We are now obviously interested in assessing the merit of this decision. The following useful lemma allows us to do that.

5.3 Lemma [The Chernoff bound] Let $X_1, \ldots, X_t$ be independent binary random variables, such that for each i we have $\Pr[X_i = 1] = p$ and $\Pr[X_i = 0] = 1 - p$. The variable $X := X_1 + \cdots + X_t$ has expected value $\mu := pt$. For any $0 \leqslant \theta \leqslant 1$ we have

$$\begin{aligned} \Pr[X - \mu \leqslant -\theta\mu] &\leqslant e^{-\theta^2\mu/3}, \text{ and} \\ \Pr[X - \mu \geqslant \theta\mu] &\leqslant e^{-\theta^2\mu/3}. \end{aligned}$$

Figure 5.1 shows the distribution for X. The shaded "tail" regions correspond to the the Chernoff probability bounds. Note that X is a discrete random variable. The drawing should not confuse you.

Figure 5.1: Explaining the Chernoff bounds



Now consider a PP or BPP algorithm N executed t times on an input α. Take X_i to be the variable that takes the value 1 if N gives the correct output during the i-th run, and that takes the value 0 if N gives the incorrect output. The probability of getting the correct output in each run is $p = 1/2 + \delta$ for some $\delta > 0$. Our decision by majority will be incorrect, if more individual decisions of N are incorrect, i.e., if $X = \sum_{i=1}^{t} X_i \leqslant t/2$, i.e., if $X - \mu \leqslant t/2 - \mu = \left(\frac{1}{2p} - 1\right)\mu = -\frac{\delta}{p}\mu$. Thus taking $\theta = \delta/p$ in Chernoff's first bound implies that the probability of error is $\leqslant e^{-\frac{\delta^2\mu}{3p^2}} = e^{-\frac{\delta^2 pt}{3p^2}} \leqslant e^{-\delta^2 t/3}$ with the last inequality following from the fact that $p \leqslant 1$.

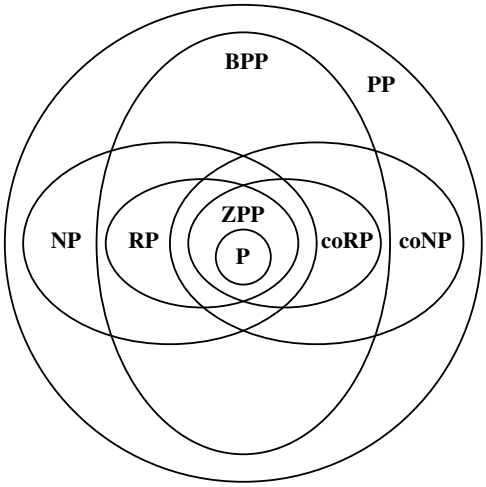
For BPP $\delta \geqslant (2/3 - 1/2) = 1/6$ and so the last probability becomes $\leqslant e^{-t/108}$, which is an exponentially small expression in the number t of trials. The conclusion holds, even if the constant $2/3$ in the definition of BPP is replaced by any other constant greater than (i.e., bounded away from) $1/2$.

With PP, however, the situation may be significantly different. We may have $\delta = 2^{-p(n)}$ for some polynomial $p(n)$ of the input size n. Such a choice for δ is perfectly consistent with the definition of PP. Now the probability of error for t trials becomes $\leqslant e^{-\frac{t/3}{2^{p(n)}}}$. But then we require an exponential (in n) number t of trials in order to make this error probability sufficiently small, i.e., we have to run N an exponential number of times in order to gain a desired level of confidence about the correctness of the decision by majority. This is not surprising, since the decision by majority becomes effective, only when there is a marked bias for correct decisions compared to incorrect decisions. The bias $2^{-p(n)}$, though positive, is too ineffective. BPP overcomes this difficulty by demanding the bias δ to be bounded away from 0, as mentioned earlier.

5.2 Relation among the randomized classes

Let us now investigate the relationships among the randomized complexity classes. Figure 5.2 describes these relations, some which are provable, others conjectured.

Figure 5.2: Relation among randomized complexity classes



First note that P sits inside every randomized class of Definition 5.2, because a poly-time deterministic algorithm may make coin tosses and ignore the outcomes of these tosses, i.e., spawn identical branches irrespective of the toss outcomes.

5.4 Proposition RP $\subseteq$ NP and so coRP $\subseteq$ coNP.

Proof Consider the computation tree on an input α of a randomized poly-time algorithm N for some $L \in$ RP. If $\alpha \in L$, at least half (and so at least one) of the branches are accepting. So N accepts α as an NTM. Conversely, if $\alpha \notin L$, the computation tree has no accepting branches, i.e., N rejects L as an NTM. Thus $L \in$ NP too. ◄

It is not known if the classes RP and coRP are equal. They are conjectured to be unequal. Their intersection is precisely the class ZPP which evidently contains P. Whether or not ZPP = P is another unknown question. Some researchers believe that P and ZPP are equal, though there does not seem to be an overwhelming evidence in favor of this conjecture.

Now let us look at two-sided randomized classes. Running an RP or coRP algorithm twice reduces the error probability to $1/4 < 1/3$ implying that

$$\mathrm{RP} \subseteq \mathrm{BPP}, \quad \text{and} \quad \mathrm{coRP} \subseteq \mathrm{BPP}.$$

Also by definition

$$\mathrm{BPP} \subseteq \mathrm{PP}.$$

In view of the symmetry in the definitions, BPP and PP are closed under complementation, i.e.,

$$\mathrm{coBPP} = \mathrm{BPP} \quad \text{and} \quad \mathrm{coPP} = \mathrm{PP}.$$

The exact relation of BPP with NP is not known. It can, however, be shown that

$$\mathrm{BPP} \subseteq \Sigma_2\mathrm{P} \quad \text{and so} \quad \mathrm{BPP} \subseteq \Sigma_2\mathrm{P} \cap \Pi_2\mathrm{P}$$

(BPP is closed under complementation).

5.5 Proposition $\mathrm{NP} \subseteq \mathrm{PP}$.

Proof Let $L \in \mathrm{NP}$ and N an NTM that decides L in polynomial time. Without loss of generality we may assume that N is a PPT. I convert N to a PPT N' that accepts and rejects L by majority. Assume that N's running time is bounded by the polynomial $p(n)$, i.e., every branch of N's computation has a probability $\geqslant 2^{-p(n)}$. At start-up N' makes a coin toss. If the toss outcome is 'head', N' simulates N on the input α with nondeterministic moves of N replaced by probabilistic moves obtained from a fresh sequence of coin tosses. If the first toss (immediately after start) gives a 'tail', N' makes $p(n)+1$ further coin tosses. If all these $p(n)+1$ tosses give 'head', N' rejects, else it accepts.

Assume that $\alpha \in L$. Then N has at least one accepting branch. This branch has length $l \leqslant p(n)$. But then the computation tree for N' has one accepting branch of length $l+1 \leqslant p(n)+1$ via the topmost 'head'. It also has $2^{p(n)+1}-1$ accepting branches under the topmost 'tail'. Thus the probability of acceptance of α by N' is at least $\frac{1}{2^{l+1}} + \left(\frac{1}{2} - \frac{1}{2^{p(n)+2}}\right) > \frac{1}{2}$, since $l+1 < p(n)+2$. Thus N' accepts by majority.

Now assume that $\alpha \notin L$. Then each branch of computation of N is rejecting, yielding a total probability of $1/2$ under the topmost 'head'. In addition, we have exactly one rejecting branch under the topmost 'tail'. Thus the probability that N' rejects α is $\frac{1}{2} + \frac{1}{2^{p(n)+2}} > \frac{1}{2}$, i.e., N' rejects by majority too. ◀

The classes RP and BPP are not known to have complete problems. PP, however, possesses complete problem (under poly-time reductions). For example, the language

$$\mathrm{MAJSAT} := \{\langle\phi\rangle \mid \phi \text{ is a Boolean formula on } m \text{ variables and with } \geqslant 2^{m-1}+1 \text{ satisfying assignments}\}$$

is PP-complete. MAJSAT comprises those satisfiable formulas that have more than half (i.e., the majority) of the assignments satisfying.

5.3 Examples

Symbolic determinants

The determinant of an $n \times n$ matrix $A = (a_{i,j})$ is defined as $\det A := \sum_\pi \left[\sigma(\pi) \prod_{i=1}^n a_{i,\pi(i)}\right]$, where the sum runs over all permutations π of $\{1, 2, \ldots, n\}$, and where $\sigma(\pi)$ denotes the sign of the permutation π (1 or -1 depending on whether π is even or odd, i.e., on whether π can be written as a product of an even or an odd number of transpositions). When the elements of A are integers, one typically uses Gaussian elimination (a sequence of elementary row operations) to reduce the matrix to an upper-triangular form. The

determinant is then the product of the diagonal elements of the reduced matrix. Though the final answer (the determinant) is an integer, the computation involves rational arithmetic. It can be shown that the intermediate rational integers that show up in the computation are no bigger than polynomials in the input size.

Problems arise when the elements of the matrix A are multivariate polynomials (with integer coefficients). Gaussian elimination requires arithmetic on rational functions, but the size of these intermediate rational functions (and also the final result) may blow up to occupy a space exponential in the input size. But we may restrict our attention only to the decision version whether the determinant identically vanishes (i.e., is the zero polynomial). In other words, we concentrate on the language

$$\text{SYMBOLIC-DET} := \{\langle A\rangle \mid A \text{ is a square matrix with polynomial entries and non-zero determinant}\}.$$

As discussed earlier, actually computing the determinant may take exponential space and time. In fact, no deterministic poly-time algorithms are known for SYMBOLIC-DET. We will now provide a (poly-time) Monte Carlo algorithm for this language. The algorithm is based on the following fact:

5.6 Lemma Let $f(x_1, \ldots, x_m)$ be a non-zero polynomial with $\deg_{x_i} f \leqslant d$ for every i. For an integer $B \in \mathbb{N}$ the number of roots of f in $\{0, 1, \ldots, B-1\}^m$ is $\leqslant mdB^{d-1}$. Thus a randomly chosen tuple in $\{0, 1, \ldots, B-1\}^m$ is a root of f with probability $\leqslant mdB^{d-1}/B^d = md/B$. For $B \geqslant 2md$ this probability is $\leqslant 1/2$.

Proof We go by induction on m. For $m = 1$ the result follows from the fact that a univariate polynomial of degree d (and with integer coefficients) can have at most d roots. So assume $m > 1$ and the result holds for $m-1$ variables. Substituting some value $k \in \{0, 1, \ldots, B-1\}$ for x_k in f gives us a polynomial $\phi_k(x_1, \ldots, x_{m-1}) := f(x_1, \ldots, x_{m-1}, k)$ in $m-1$ variables and with the same degree bound d. Assume that for s values of k the polynomial ϕ_k is identically zero and for the remaining $B-s$ values of k this is not zero. By induction the number of roots of f in the desired range is then $\leqslant sB^{m-1} + (B-s)(m-1)dB^{m-2}$. But ϕ_k is identically zero, if and only if $x_m - k$ divides $f(x_1, \ldots, x_m)$ (an easy check), and the polynomials $x_m - k$ are pairwise coprime for different values of k, that is, $s \leqslant d$. Also $B - s \leqslant B$. Therefore, the desired number of roots of f is $\leqslant dB^{m-1} + B(m-1)dB^{m-2} = mdB^{m-1}$. ◀

This observation leads to the Algorithm 5.1. If $\det A$ is identically zero, then any choice for $(a_1, \ldots, a_m)$ leads $\det A'$ to evaluate to 0. On the other hand, if $\det A$ is not identically zero, at least half of the choices $(a_1, \ldots, a_m)$ lead the algorithm to accept $\langle A\rangle$. It is also clear that the algorithm runs in poly-time of the input size. Thus SYMBOLIC-DET $\in$ RP.

Algorithm 5.1: A Monte Carlo algorithm for SYMBOLIC-DET

Input: $\langle A\rangle$, where A is a square matrix whose entries are multivariate polynomials with integer coefficients.

Stages:
1. Determine m and the degree bound d from A.
2. Take $B := 2md$.
3. Choose a random point $(a_1, a_2, \ldots, a_m) \in \{0, 1, \ldots, B-1\}^m$.
4. Substitute x_i by a_i in A for all i. Let A' be the resulting matrix with integer entries.
5. Compute $\det A'$ by Gaussian elimination.
6. If this determinant is 0, *reject*, else *accept*.

Primality testing

As a second example, let us look at the classical problem:

$$\text{PRIME} := \{\langle n\rangle \mid n \in \mathbb{N} \text{ is prime}\}.$$

Now we know that PRIME $\in$ P. But the best know deterministic algorithm runs in time $O(\log^{7.5} n)$ and is clearly impractical. Under Artin's conjecture (an unproven mathematical fact) the running time reduces to $O(\log^6 n)$ which continues to remain quite big. However, using randomization one can solve PRIME in time $O(\log^3 n)$. Since even integers can be quickly recognized (from its binary or decimal representation), we concentrate only on odd integers. The following is a well-known result from number theory:

5.7 Theorem [Fermat's little theorem] Let p be an odd prime and a an integer coprime to p. Then $a^{p-1} \equiv 1 \pmod{p}$.

Consider the Monte Carlo Algorithm 5.2. Stage 2 (computation of $\gcd(a, n)$) could have been omitted, since for a general n having no small prime divisors, the probability that a random a is *not* coprime to n is overwhelmingly small.

Algorithm 5.2: A Monte Carlo algorithm for primality checking

Input: $\langle n \rangle$, where n is an *odd* positive integer.

Stages:
1. Pick a random $a \in \{1, \ldots, n-1\}$.
2. If a is not coprime to n, *reject*.
3. If $a^{p-1} \equiv 1 \pmod{p}$, *accept*, else *reject*.

If n is indeed prime, then any a satisfies the congruence in Stage 3 and is accepted by the above algorithm. For most composite integers n a randomly chosen a coprime to n satisfies this congruence with probability at least $1/2$. Thus this looks like a good coRP algorithm for PRIME.

Unfortunately, there exists a class of composite integers n, known as Carmichael numbers, for which the Fermat congruence holds for *every* a coprime to n. Though Carmichael numbers are not quite common, there is an infinite number of them. Moreover, there are no efficient (poly-time) algorithms known to recognize Carmichael numbers. Thus the above randomized algorithm is bound to fail for these numbers.

This problem is remedied by the following observation. An odd composite integer n has at least four square roots of 1 modulo n, i.e., at least two others than the trivial square roots ± 1). Thus we keep track of the sequence how a^{n-1} reaches 1 modulo n. If any non-trivial square-root is found on the way, n is definitely known to be composite. Consider the modified Monte Carlo Algorithm 5.3, known as the Miller-Rabin primality test.

Algorithm 5.3: Miller-Rabin primality proving algorithm

Input: $\langle n \rangle$, where n is an *odd* positive integer.

Stages:
1. Write $n - 1 = 2^s t$ for some $s, t \in \mathbb{N}$ with t odd.
2. Pick a random $a \in \{0, 1, \ldots, n-1\}$.
3. If a is not coprime to n, *reject*.
4. Compute $b_0 \equiv a^t \pmod{n}$.
5. If $b_0 \equiv 1 \pmod{n}$, *accept*.
6. For $i = 1, 2, \ldots, s$ compute $b_i \equiv b_{i-1}^2 \pmod{n}$.
7. If $b_s \not\equiv 1 \pmod{n}$, *reject*.
8. Let i be the smallest index for which $b_i \equiv 1 \pmod{n}$ and $b_{i-1} \not\equiv 1 \pmod{n}$.
9. If $b_{i-1} \equiv \pm 1 \pmod{n}$, *accept*, else *reject*.

Let me first explain what the algorithm does. Since $n - 1$ is even, we get positive values for s and t. It then selects a random base a. As in the previous algorithm, the check if $\gcd(a, n) = 1$ in Stage 3 could be avoided. The algorithm then computes the numbers $b_i \equiv a^{2^i t} \pmod{n}$ for $i = 0, 1, \ldots, s$. At the end

we have $b_s \equiv a^{2^s t} \equiv a^{n-1} \pmod{n}$. If we eventually do not reach $b_s \not\equiv 1 \pmod{n}$, n is certainly not prime by Fermat's little theorem, so the algorithm rejects. Otherwise it finds out the smallest index i for which $b_i \equiv 1 \pmod{n}$. If $i = 0$, i.e., b_i is the first element in the sequence $b_0, b_1, \ldots, b_s$, then the algorithm accepts. If $0 < i \leqslant s$, then b_{i-1} is defined and is a square root of 1 modulo n. If this root is trivial (± 1), the algorithm accepts, else it rejects.

Modulo a prime 1 has only trivial square roots. Thus if n is a prime, we cannot locate non-trivial square roots of 1. Moreover, by Fermat's little theorem, we must obtain $b_s \equiv 1 \pmod{n}$. Thus the algorithm definitely accepts n in this case.

Now assume that n is composite. If a passes the test (i.e., it gives $a^{n-1} \equiv 1 \pmod{n}$ without revealing any non-trivial square root of 1), we say that n is a strong pseudoprime to the base a. It can be shown that there are at most 1/4-th of the bases (coprime to n) to which n is a strong pseudoprime. Thus the probability that a random base a gives $a^{n-1} \equiv 1 \pmod{n}$ without divulging a non-trivial square root of 1 in the computation is at least $3/4$ (which is $\geqslant 1/2$). Thus Miller-Rabin test is a one-sided probabilistic algorithm for PRIME. One can easily check that the algorithm runs in poly-time in the input size ($\log n$). Therefore, PRIME $\in$ coRP and consequently COMPOSITE $\in$ RP.

Other problems

There exists a huge lot of other problems (function and decision) that are known to be handled effectively by randomization. Here we list a few. The details of these algorithms involve a pretty deal of mathematical gadgets and are well beyond the scope of this small survey.

- *Finding primitive roots*: Given a prime p and the prime factorization of $p - 1$ determine a primitive root modulo p (i.e., a generator for $\mathbb{Z}_p^*$.
- *Square roots modulo a prime*: Given a prime p and a quadratic residue a, compute an element b such that $b^2 \equiv a \pmod{p}$.
- *Irreducible polynomials over a finite field*: Given a finite field $\mathbb{F}_q$ (such as $\mathbb{Z}_p$ for a prime p) and a positive integer d, compute an irreducible polynomial of degree d with coefficients from $\mathbb{F}_q$.
- *RSA key inversion*: Given an RSA public-key (n, e) and the corresponding private key d, compute the factorization of n.
- *Incremental convex hull*: Given the convex hull of a set of n points $x_1, \ldots, x_n$ in $\mathbb{R}^d$ and a new point x_{n+1}, compute the convex hull of $x_1, \ldots, x_{n+1}$.
- *The min-cut problem*: Given an undirected graph G a cut is a partitioning of the vertex set $V(G)$ into two sets C and $\bar{C}$. The cut is minimum if the number of edges joining C with $\bar{C}$ is minimum. Compute a minimum cut for G.

Randomization may also help reduce the running time from fully exponential to subexponential. Though these algorithms do not necessarily come in the domain of PP, they are also worth studying. For example, the best know algorithms for *integer factorization* and *discrete logarithm* problems are subexponential and based on randomization techniques.

Satisfiability and randomization

Hmmm... Does randomization *always* help? Perhaps no! Let us investigate the question if SAT has a good randomized algorithm. If a satisfiable formula has many satisfying assignments, we may hope to find one such assignment soon from a sequence of randomly selected assignments (cf. MAJSAT). However, if the

formula has only few satisfying assignments, it could be difficult to locate these rare assignments by random choices only. The following algorithms attempts to improve upon an assignment so as to move *closer* to a satisfying assignment. For simplicity, we may assume that the formula is given in cnf (or, for that matter, in 3-cnf).

Input: A boolean formula $\langle\phi\rangle$ in cnf.

Stages:
1. Let $x_1, \ldots, x_m$ be all the variables of ϕ.
2. Randomly pick a truth assignment T of $x_1, \ldots, x_m$.
3. for $i = 1, \ldots, t$ repeat the following:
4. Evaluate ϕ at the current assignment T.
5. If the evaluation results in 1, *accept*.
6. Randomly select an unsatisfied clause from ϕ and a false literal in the clause.
7. Flip the truth assignment of this literal and change T accordingly.
8. No satisfying assignments have been found in t iterations, so *reject*.

How is this *random walk algorithm* expected to perform on a cnf formula? Consider the 3-cnf formula:

$$\phi(x_1, \ldots, x_m) := \left(\bigwedge_{i=1}^{m}(x_i \vee x_i \vee x_i)\right) \wedge \left(\bigwedge_{\substack{1\leqslant i,j,k\leqslant m\\ i\neq j\neq k}}(x_i \vee \overline{x_j} \vee \overline{x_k})\right).$$

There are exactly m clauses of the first type $(x_i \vee x_i \vee x_i)$ and $m(m-1)(m-2)$, i.e., $\mathrm{O}(m^3)$ clauses of the second type $(x_i \vee \overline{x_j} \vee \overline{x_k})$. The clauses of the first type indicate that ϕ has only one satisfying truth assignment, namely, $x_1 = \cdots = x_m = 1$. A randomly chosen truth assignment T is expected to have about half of the variables assigned to 0. The probability of acceptance in Stage 5 is rather slim. Now let's see how the Stages 6 and 7 update T. Stage 6 picks a random clause. Since there are many more clauses of the second type than of the first type, the chance that an unsatisfied clause of the second type is chosen is quite big. Call this clause $c := x_i \vee \overline{x_j} \vee \overline{x_k}$. Since c is false, we have $x_i = 0$ and $x_j = x_k = 1$ in T. Now the chance that Stage 7 changes the assignment of x_j or x_k from 1 to 0 is more than that it changes the assignment of x_i from 0 to 1. But then with higher probability T goes worse, i.e., more distant from the correct assignment (all true)! Consequently, after a huge number of iterations (an exponential value in m) we may remain unable to find the satisfying truth assignment.

However, it can be shown that if ϕ is any satisfiable 2-cnf formula, then the probability of meeting a satisfying truth assignment after $t = 2m^2$ iterations is at least $1/2$. Thus the above random walk algorithm solves $2\mathrm{SAT}$ in randomized poly-time (RP). But we already know that $2\mathrm{SAT} \in \mathrm{P}$ and so this algorithm is not a substantial discovery.

That the random walk algorithm didn't work favorably for SAT does not imply there do not exist better (more effective) randomized algorithms for SAT. May be there are. We don't know!!!

Exercises for Chapter 5

*** 1.** Define the class PP' to be a minor variant of PP as follows: $L \in \mathrm{PP}'$ if and only if there exists a PPT N such that if $\alpha \in L$, N accepts α with probability $> 1/2$, whereas if $\alpha \notin L$, N rejects α with probability $\geqslant 1/2$. Thus PP' accepts by majority and rejects by non-minority. Show that $\mathrm{PP}' = \mathrm{PP}$.

2. Define the class PP'' to be another *minor* variant of PP as follows: $L \in \mathrm{PP}''$ if and only if there exists a PPT N such that if $\alpha \in L$, N accepts α with probability $\geqslant 1/2$, whereas if $\alpha \notin L$, N rejects α with probability $\geqslant 1/2$. Thus PP'' both accepts and rejects by non-minority. Show that $\mathrm{PP}'' = \Sigma^*$.

*** 3.** Let $k \in \mathbb{N}$ be a constant. Define the class PP_k as follows: $L \in \mathrm{PP}_k$ if and only if there exists a PPT N such that whenever $\alpha \in L$, N accepts α with probability $> 2^{-k}$, whereas for $\alpha \notin L$, N rejects α with probability $\geqslant 1 - 2^{-k}$. (In other words, N accepts α if and only if more than a 2^{-k} fraction of branches of N on α are accepting. Note that $\mathrm{PP}' = \mathrm{PP}_1$.) Prove that $\mathrm{PP}_k = \mathrm{PP}$.

4. Define the class BPP' as follows: $L \in \mathrm{BPP}'$ if and only if there exists a polynomial $p(n)$ and a PPT N, such that whenever $\alpha \in L$, N accepts α with probability $\geqslant \frac{1}{2} + \frac{1}{p(n)}$, and whenever $\alpha \notin L$, N rejects α with probability $\geqslant \frac{1}{2} + \frac{1}{p(n)}$, where $n = |\alpha|$. Argue that $\mathrm{BPP}' = \mathrm{BPP}$.

5. Prove that $\mathrm{PP} \subseteq \mathrm{PSPACE}$.

6. Prove the following conditional statements:

(a) If $\mathrm{P} = \mathrm{NP}$, then $\mathrm{P} = \mathrm{BPP}$. (**Hint:** Polynomial hierarchy.)

(b) If $\mathrm{NP} \subseteq \mathrm{coRP}$, then $\mathrm{ZPP} = \mathrm{NP}$.

*** (c)** If $\mathrm{NP} \subseteq \mathrm{BPP}$, then $\mathrm{RP} = \mathrm{NP}$. (**Hint:** Compute a (probably) satisfying assignment of a Boolean formula using a BPP algorithm for SAT.)

7. Prove that RP and BPP are closed under union and intersection.

8. [Probabilistic space-bounded Turing machines] In the text we have studied only probabilistic time-bounded Turing machines. Now we define probabilistic space classes. Let $f(n) \geqslant \log n$

(a) Define the class $\mathrm{RSPACE}(f(n))$ as follows: $L \in \mathrm{RSPACE}(f(n))$ if and only if there exists an $\mathrm{O}(f(n))$-space and $2^{\mathrm{O}(f(n))}$-time probabilistic Turing machine N with the property that if $\alpha \in L$, then N accepts α with probability $\geqslant 1/2$, whereas if $\alpha \notin L$, then N rejects L with probability 1. In particular, define $\mathrm{RPSPACE} := \bigcup_{k \in \mathbb{N}} \mathrm{RSPACE}(n^k)$ and $\mathrm{RL} := \mathrm{RSPACE}(\log n)$. Show that

$$\mathrm{SPACE}(f(n)) \subseteq \mathrm{RSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n)).$$

In particular, $\mathrm{L} \subseteq \mathrm{RL} \subseteq \mathrm{NL}$ and $\mathrm{RPSPACE} = \mathrm{PSPACE}$.

**** (b)** Define the class $\mathrm{RSPACE}'(f(n))$ as in Part (a) except that the restriction $2^{\mathrm{O}(f(n))}$-time on N is now removed. Show that $\mathrm{RSPACE}'(f(n)) = \mathrm{NSPACE}(f(n))$. In particular, $\mathrm{RL}' = \mathrm{NL}$.

9. In this exercise we deal with an example of a problem in RL. Consider the language:

$$\mathrm{UPATH} := \{\langle G, s, t\rangle \mid G \text{ is an } \textit{undirected} \text{ graph with an } s,t\text{-path}\}.$$

The corresponding directed version PATH is known to be NL-complete and so is not in RL, unless $\mathrm{RL} = \mathrm{NL}$, a relation which appears unlikely. However, $\mathrm{UPATH} \in \mathrm{RL}$ as the remaining part of this exercise suggests.

A *random walk* in an undirected graph G is a sequence of steps with each step being a movement from a vertex to a randomly chosen neighbor. Assume that G has an s,t-path. It can be shown (for example, see Papadimitriou) that a random walk starting at s and consisting of $8|V(G)||E(G)|$ (which is $\mathrm{O}(m^3)$ where $m = |V(G)|$) steps visits t with probability $\geqslant 1/2$. Design an RL algorithm for UPATH based on this result.