

## Notes on Randomized Algorithms

Randomness can help to solve problems and is a fundamental ingredient and tool in modern complexity theory.

We assume a source of random bits. In complexity theory, we assume our source can just spit out random bits at a cost of one step per bit. Before getting into definitions let us look at some examples of randomized algorithms.

### 1 Promise problems

Our first example will consider a problem a little different from the usual language recognition one, so we begin by discussing this new class of problems. Formally, a *promise problem* is a pair  $(P, N)$  where  $P, N \subseteq \Sigma^*$  are disjoint sets. The associated problem is:

**Input:**  $x \in P \cup N$

**Question:** Is  $x \in P$ ?

The problem is specified by a set  $P$  of “positive” instances and a set  $N$  of “negative” instances. Given an instance  $x \in P \cup N$  we must determine whether it is a positive or negative instance. Note that the input is required to be in  $P \cup N$ . That’s the “promise” in the name of the problem.

An example of a promise problem is  $(L, \bar{L})$  where  $L$  is any language, and because of this, promise problems generalize problems specified by languages. Promise problems capture the fact that in practice we might know something about the distribution of input instances that enables us to preclude some instances and concentrate on others.

We can extend complexity classes to consider promise problems in the natural way. For example, a promise problem  $(P, N)$  is said to be in **pP** if there is a TM  $M$  and a polynomial  $p$  such that for all  $x \in P \cup N$

- If  $x \in P$  then  $M(x)$  accepts
- If  $x \in N$  then  $M(x)$  rejects
- $M(x)$  halts in at most  $p(|x|)$  steps.

Here **pP**, “promise-**P**”, is a complexity class containing promise problems rather than languages. It is the promise-problem analogue of **P**.

## 2 Dense-SAT

For any boolean formula  $\varphi$  with  $n$  variables we let

$$\text{SatAssg}(\varphi) = \{ x \in \{0, 1\}^n : \varphi(x) = 1 \}$$

be the set of all satisfying assignments to  $\varphi$ . Notice that

$$\varphi \in \text{SAT} \text{ iff } |\text{SatAssg}(\varphi)| \geq 1.$$

So the problem of deciding whether  $\varphi \in \text{SAT}$  is just the problem of deciding whether  $|\text{SatAssg}(\varphi)| \geq 1$  or not. We consider a variant of this problem. We let

$$\begin{aligned} P &= \{ \varphi : |\text{SatAssg}(\varphi)| \geq 2^{n-2} \text{ where } n \text{ is the number of variables in } \varphi \} \\ N &= \{ \varphi : \text{SatAssg}(\varphi) = \emptyset \}. \end{aligned}$$

In other words,  $P$  contains formulas which have lots of satisfying assignments, while  $N$  contains formulas that have no satisfying assignments, meaning are unsatisfiable. We consider the promise problem  $(P, N)$ , and call it DENSE-SAT. That is:

### DENSE-SAT

**Input:** A boolean formula  $\varphi \in P \cup N$

**Question:** Is  $\varphi$  satisfiable?

Note that this is not quite the usual kind of decision problem, because the inputs are not arbitrary formulas. Rather, the inputs are drawn from a restricted set of formulas, namely the set  $P \cup N$ . So, this decision problem does not correspond to a language, but instead is a promise problem.

We want to know whether this promise problem is in **pP**. Namely we ask if there exists a TM  $M$  and a polynomial  $p$  such that for all  $\varphi \in P \cup N$

- If  $\varphi \in P$  then  $M(\varphi)$  accepts
- If  $\varphi \in N$  then  $M(\varphi)$  rejects
- $M(\varphi)$  halts within  $p(|\varphi|)$  steps.

Notice that no requirements are made for  $\varphi \notin P \cup N$ .

However, it is not clear how we can produce a machine  $M$  as above. It is true that we have more information about  $\varphi$  than in the SAT problem, but how do we use it? Can we use it to prune a search for a truth assignment? Not clear how. If we started searching through all the  $2^n$  possible assignments to the variables of a given  $n$ -variable formula  $\varphi$ , then we may need to try  $2^n - 2^n/4 + 1 = 3/4 \cdot 2^n + 1$  assignments before we either find one that satisfies  $\varphi$  or are able to say that none such exists.

But what if we chose an assignment at random? That is:

Algorithm  $M(\varphi)$

Let  $n$  be the number of variables in  $\varphi$

Pick bits  $b_1, \dots, b_n$  at random

If  $\varphi(b_1, \dots, b_n) = 1$  then accept else reject

This is a polynomial time algorithm. We are interested in the probability that this algorithm accepts. This is

$$\mathbf{AccPr}_M(\varphi) = \Pr[M(\varphi) \text{ accepts}].$$

The probability is over the coins of the algorithm, namely the choices  $b_1, \dots, b_n$ , for each fixed  $\varphi$ . Now, the thing to notice is that:

$$\mathbf{AccPr}_M(\varphi) = \frac{|\text{SatAssg}(\varphi)|}{2^n}.$$

From the above we have:

- (1) If  $\varphi \in P$  then  $\mathbf{AccPr}_M(\varphi) \geq 2^{n-2}/2^n = 1/4$ .
- (2) If  $\varphi \in N$  then  $\mathbf{AccPr}_M(\varphi) = 0$ .

We look at this as saying that the error-probability is  $3/4$ . How good is this? Well,  $3/4$  is a lot, in the sense that being wrong 75% of the time is being wrong rather too often for comfort. What can we do? What if we chose 300 assignments at random instead of just one? That is,

Algorithm  $M'(\varphi)$

```

Let  $n$  be the number of variables in  $\varphi$ 
For  $i = 1, \dots, 300$  do
    Pick bits  $b_{i,1}, \dots, b_{i,n}$  at random
    If  $\varphi(b_{i,1}, \dots, b_{i,n}) = 1$  then accept
Reject
```

This algorithm has the property that:

- (1) If  $\varphi \in P$  then  $\mathbf{AccPr}_M(\varphi) \geq 1 - (3/4)^{300} \geq 1 - (1/2)^{100}$ .
- (2) If  $\varphi \in N$  then  $\mathbf{AccPr}_M(\varphi) = 0$ .

The error probability is  $2^{-100}$ . There is a greater chance that your machine will be hit by a meteorite than that the above test will fail. Thus, we view the above as saying that in some practical sense, we have obtained a polynomial-time algorithm for the DENSE-SAT problem. This algorithm might sometimes be wrong, but too seldom for us to worry about. This illustrates how randomness helps solve problems.

Where do coin flips come from? Any random process. We will see the model in more detail later. Let us first look at another example, less contrived than this one.

### 3 Equality testing

Party  $A$  holds a string  $a \in \{0, 1\}^n$  while party  $B$  holds a string  $b \in \{0, 1\}^n$ . They are at different locations but can communicate over a network. They want to know whether or not  $a = b$ . What is the obvious way?  $A$  sends the string  $a$  to  $B$  who checks whether  $a = b$ . But this takes  $n$  bits of communication. This is judged too much. We want to use less bandwidth.

This kind of problem might arise in the maintenance of replicated databases. Suppose we want to maintain a copy of the same database at two locations  $A$  and  $B$ . Let  $a$  be the copy at  $A$  and

<u>Algorithm for A:</u> <u>Input:</u> $a \in \{0, 1, \dots, 2^n - 1\}$ .  Pick $i$ at random from the set $\{1, \dots, n\}$ Let $f_a = a \bmod p_i$ Send $(i, f_a)$ to $B$ .	<u>Algorithm for B:</u> <u>Input:</u> $b \in \{0, 1, \dots, 2^n - 1\}$ .  Receive a pair $(i, f_a)$ from $A$ Let $f_b = b \bmod p_i$ If $f_a = f_b$ then accept else reject
---	--

Figure 1: **Randomized protocol for database equality testing**

---

$b$  the copy at  $B$ . They start out equal but then updates are made. If all goes well, updates are always communicated to both locations and the databases stay equal, but possibly some update gets reflected at one location and not the other and then we would have  $a \neq b$ . In order to make sure that this did not happen, we occasionally want to test whether or not the current copies of  $a$  and  $b$  are equal. The databases might be large, for example  $n = 10^{12}$  bits. So it is not practical to transmit the whole database.

The problem also arises in pattern matching, where  $a$  is a pattern for which we are searching in a file. We might try matching  $a$  with patterns  $b$  drawn from the file and want to be able to quickly test whether  $a = b$ .

It is possible to prove that there is no procedure which is always correct (meaning correctly determines whether or not  $a = b$ ) yet uses a total bandwidth (number of communicated bits) less than the size  $n$  of the strings  $a, b$ . However, you can do better if you are willing to live with a low (but non-zero) probability of accepting even if the databases are not equal. Let's see how.

In the sequel we view  $a, b$  as integers in the range  $0, \dots, 2^n - 1$ . In other words we view the strings as binary representations of integers.

Let  $p_i$  be the  $i$ -th prime number,  $i = 1, 2, \dots$  (Thus  $p_1 = 2$  and  $p_2 = 3$  and  $p_3 = 5$  and  $p_4 = 7$  and  $p_5 = 11$  etc.) We will use the following version of the Chinese remainder theorem which we don't prove.

**Fact 3.1** Suppose  $a, b$  are integers and  $S \subseteq \{1, 2, 3, \dots\}$  is a finite set such that  $0 \leq a, b < p(S)$  where we define

$$p(S) = \prod_{i \in S} p_i .$$

Then  $a = b$  if and only if  $a \bmod p_i = b \bmod p_i$  for all  $i \in S$ . ■

Or, put another way,  $a \neq b$  if and only if there exists a value of  $i \in S$  such that  $a \bmod p_i \neq b \bmod p_i$ . The idea now is to use the values  $a \bmod p_i$  or  $b \bmod p_i$  as “fingerprints”. The protocol is illustrated in Figure 1. The value  $f_a$  is called a “fingerprint” of the database  $a$ .

Communication-complexity of protocol:

The “communication-complexity” is defined as the number of bits transmitted. The estimate of this value relies on a basic result from analytical number theory which we will not prove.

**Fact 3.2** There is a constant  $c$  such that  $p_i \leq c \cdot i \ln(i)$  for all integers  $i \geq 1$ . ■

Here  $\ln(\cdot)$  denotes the natural logarithm. Now the number of bits sent by  $A$  is

$$\begin{aligned} \lg(n) + |p_i| &\leq \lg(n) + \lg[c(n \ln(n))] \\ &= 2\lg(n) + o(\lg(n)). \end{aligned}$$

Namely about  $2\lg(n)$  bits. (Here  $\lg(\cdot)$  denote the logarithm to base two.) For  $n = 10^{12} \approx 2^{40}$  this comes to 80 bits. Big savings.

### Correctness of protocol:

Why does the protocol work? If  $a = b$  then  $B$  always accepts. If  $a \neq b$ , what happens?  $B$  might accept, but seldom, meaning with low probability. To estimate the probability we use Fact 3.1.

**Claim 3.3** Suppose  $n \geq 6$ . Let  $a, b \in \{0, \dots, 2^n - 1\}$  and  $a \neq b$ . Let

$$\text{Bad}_{a,b} = \{i : 1 \leq i \leq n \text{ and } a = b \pmod{p_i}\}.$$

Then  $|\text{Bad}_{a,b}| \leq n/2$ . ■

**Proof:** Let  $S = \text{Bad}_{a,b}$  and let  $p(S)$  be defined as in Fact 3.1. Assume towards a contradiction that  $|S| > n/2$ . Now observe that  $0 \leq a, b < p(S)$ . Why is this true? We already know that  $0 \leq a, b < 2^n$  so it would suffice to argue that  $2^n \leq p(S)$ . The latter is true because  $p_3$  onwards all  $p_i$  are more than five, so the product in question is at least  $2 \cdot 3 \cdot 5^{n/2-1} \geq (6/5) \cdot 5^{n/2} \geq 4^{n/2} = 2^n$  assuming that  $n \geq 2$  is even. If  $n$  is odd the product in question is at least  $2 \cdot 3 \cdot 5^{(n+1)/2-2}$  which is at least  $2^n$  as long as  $n \geq 7$ . Fact 3.1 now tells us that  $a \neq b$  implies that there is some  $i \in S$  such that  $a \neq b \pmod{p_i}$ . That contradicts the definition of the bad set. ■

**Claim 3.4** Suppose  $a, b \in \{0, \dots, 2^n - 1\}$  and  $a \neq b$ . Then in our protocol,  $B$  will accept with probability at most  $1/2$ . ■

**Proof:**  $B$  accepts if and only if  $i \in \text{Bad}_{a,b}$  where  $i$  is the value received from  $A$ . The probability arises due to the random choice of  $i$  made by  $A$ . The probability that  $B$  accepts is

$$\frac{|\text{Bad}_{a,b}|}{n} \leq \frac{n/2}{n} = \frac{1}{2}.$$

Here we used the fact that  $A$  chooses  $i$  at random in the set  $\{1, \dots, n\}$  and then applied the above claim. ■

To do better, repeat  $k = 100$  times. The probability of error is now  $2^{-100}$ , small enough to neglect. But still the cost is only about 8,000 bits of communication.

What makes this work? Randomness defeats any structure. No matter where is the difference in  $a, b$ , the scheme will catch it. Think of an adversary trying to make the worst possible  $a, b$ . But it can't guess the random value of  $i$ .

It turns out that any algorithm that can detect an error half the time must communicate at least  $\lg(n)$  bits. So the above algorithm is close to best possible.

Why wouldn't something simpler work, like  $A$  sends the parity of all the bits in  $a$ ? If  $a, b$  differ in exactly two places, the parity is the same. So this doesn't work at all.

## 4 Model and complexity classes

We now turn to definitions for randomized complexity classes. We have to begin with the model.

### 4.1 Model

Recall that our usual Turing Machine model includes three tapes: a read-only input tape, a work tape, and a write-only output tape. (The latter is not used when the machine is solving a language recognition problem.) A *randomized* Turing Machine has an additional read-only tape, called the *random tape*. On this appears a string  $R$ . We view  $R$  as another input to the machine; in other words a randomized machine  $M(\cdot; \cdot)$  takes as input a usual input  $x$  and a sequence of bits  $R$ .

Where is the randomness? In the choice of  $R$ , which is done externally. We assume that  $R$  is of some length  $r(n)$  where  $r$  is some known function called the *length of the random tape of  $M$* . We imagine  $R$  chosen at random from  $\{0, 1\}^{r(|x|)}$  before the computation begins. Thus to toss a coin is to read the first bit of string  $R$ . Whenever the machine needs to toss a coin, it simply reads the next bit on its random tape.

We measure the running time as a function of the length  $n$  of  $x$ , as usual. The machine is said to have running time  $t(n)$  if for all  $x, R$  it halts in at most  $t(|x|)$  steps when given inputs  $x, R$ . We say that  $M$  is polynomial-time if there exists a polynomial  $p(n)$  such that  $M$  has running time  $O(p(n))$ .

Note that if it has running time  $t(n)$  then it can't read more than  $t(n)$  bits from its random tape. Accordingly we may suppose that  $r(n) \leq t(n)$ .

On any inputs  $x, R$  the algorithm may accept or reject, and which of these happens depends on both the inputs. In particular for a fixed  $x$ , its computation and decision may be different for different values of the input  $R$ . Thus, the random choice of  $R$  leads to a probability of various outcomes for the machine. We are interested in this probability.

**Definition 4.1** Let  $M(\cdot; \cdot)$  be a randomized Turing Machine and let  $r(n)$  be the length of its random tape. For any  $x \in \Sigma^*$  we let

$$\mathbf{AccSet}_M(x) = \{ R \in \{0, 1\}^{r(|x|)} : M(x; R) \text{ accepts} \}$$

denote the set of all random tapes that lead  $M$  to accept on input  $x$ . We then let

$$\mathbf{AccPr}_M(x) = \frac{|\mathbf{AccSet}_M(x)|}{2^{r(|x|)}}.$$

This is called the *accepting probability* of  $M$  on input  $x$ . It is uniquely defined once we have fixed  $M$  and  $x$ . ■

It is important to note that the machine works, meaning computes and then accepts or rejects (possibly also outputs something) no matter what string  $R$  it is given. But the definition of the accepting probability corresponds to choosing  $R$  at random. And if you don't like to talk about probability, just think in terms of sets. The accepting probability is the ratio of the sizes of two sets. The first set is all those  $R$  on which  $M(x; R)$  accepts, and the second is the set of all possible random tapes  $R$ .

## 4.2 The class **RP**

Different complexity classes arise depending on the requirements made on the accepting probability.

**Definition 4.2** Let  $\epsilon: \mathbb{N} \rightarrow [0, 1]$  be a function that takes input an integer and returns a real number in the range from 0 to 1. Let  $M$  be a randomized TM and  $L$  a language. We say that  $M$  recognizes  $L$  with one-sided error-probability  $\epsilon$  if for every  $x \in \Sigma^*$ :

- (1) If  $x \in L$  then  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon(|x|)$ .
- (2) If  $x \notin L$  then  $\mathbf{AccPr}_M(x) = 0$ .

Language  $L$  is said to be in the complexity-class **RP** if there exists a constant (function)  $\epsilon < 1$  and a polynomial-time, randomized algorithm  $M$  such that  $M$  recognizes  $L$  with one-sided error-probability  $\epsilon$ . ■

That is, there is some separation in the probabilities. By polynomial-time we mean that the running time  $t(n)$  of  $M$  is a polynomial.

By a constant function we mean one that takes a single output value independent of the input, for example the function  $\epsilon$  defined by  $\epsilon(n) = 1/2$  for all  $n \in \mathbb{N}$ .

The machine above can be wrong on input  $x \in L$  with probability  $\epsilon(|x|)$ . (That's why  $\epsilon$  is called the error-probability). When  $\epsilon$  is a constant like  $1/2$ , this error is quite large. However it can be lowered as we now discuss.

We consider a randomized TM that, in addition to  $x$ , takes input an integer  $k$  that we refer to as the *security parameter*. This input is encoded in unary, because we want that  $M$  being polynomial-time allows it to run in time polynomial in  $k$ . Formally, the two inputs are encoded into one, so that the input to our TM is  $\langle x, 1^k \rangle$ . The machine will have error-probability  $2^{-k}$  on this input.

**Proposition 4.3** Suppose  $L \in \mathbf{RP}$ . Then there is a randomized polynomial-time TM  $M$  such that for all  $x \in \Sigma^*$  and all  $k \in \mathbb{N}$

- (1) If  $x \in L$  then  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) \geq 1 - 2^{-k}$ .
- (2) If  $x \notin L$  then  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) = 0$ .

We call  $M$  a *parameterized, randomized TM* recognizing  $L$  with one-sided error. ■

By choosing different values of  $k$ , we can obtain lower and lower error probability.

**Proof of Proposition 4.3:** Since  $L \in \mathbf{RP}$  there exists a constant  $\epsilon' < 1$  and a randomized, polynomial-time TM  $M'$  recognizing  $L$  with one-sided error-probability  $\epsilon'$ . Let  $r'(n)$  be the number of coin tosses used by  $M'$ . We design  $M$  to run  $M'$  an appropriate number of times, each time using new, independent coin tosses. The number of repetitions, denoted  $s$ , is chosen to satisfy

$$(\epsilon')^s \leq 2^{-k}. \quad (1)$$

Here is the algorithm:

Algorithm  $M(\langle x, 1^k \rangle; R)$

```

s ← ⌈k/lg(1/ε')⌉
For i = 1, ..., s do
  Let Ri be the next r(|x|) bits from the random tape R
  If M'(x; Ri) accepts then accept
EndFor
Reject

```

Since  $\epsilon'$  is a constant,  $s = O(k)$  and thus the above is a polynomial-time (randomized) TM. Let us now see why it does the job.

We observe that for any  $x \in \Sigma^*$

$$M(\langle x, 1^k \rangle; R) \text{ rejects} \quad \text{iff} \quad \forall i : M'(x; R_i) \text{ rejects} .$$

This means that for any  $x \in \Sigma^*$  and any  $k \in \mathbb{N}$

$$1 - \mathbf{AccPr}_M(\langle x, 1^k \rangle) = (1 - \mathbf{AccPr}_{M'}(x))^s . \quad (2)$$

Now suppose  $x \in L$ . We know that  $1 - \mathbf{AccPr}_{M'}(x) \leq \epsilon'$ . Equation (2) implies that

$$1 - \mathbf{AccPr}_M(\langle x, 1^k \rangle) = (1 - \mathbf{AccPr}_{M'}(x))^s \leq (\epsilon')^s \leq 2^{-k} ,$$

the last inequality being due to Equation (1). The above implies that  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) \geq 1 - 2^{-k}$  as desired.

On the other hand, suppose  $x \notin L$ . We know that  $\mathbf{AccPr}_{M'}(x) = 0$ . Then Equation (2) implies that  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) = 0$  as well.  $\blacksquare$

Now we look at relations to other classes.

**Proposition 4.4**  $\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{NP}$ .

The first inclusion above reflects a triviality we have seen again and again: extra resources don't hurt. In this case, problems in  $\mathbf{P}$  can be solved without recourse to randomness; if you have randomness, of course you can still solve them. Just ignore the random tape and do what you did before. For the second inclusion, the observation is that a certificate for membership of a string  $x$  in a language  $L \in \mathbf{RP}$  is a sequence of coin tosses that makes the randomized machine accept. The details follow.

**Proof of Proposition 4.4:** To prove the first inclusion, assume  $L \in \mathbf{P}$  and let  $M'$  be a TM that decides  $L$  in polynomial-time. To show that  $L \in \mathbf{RP}$  we need to exhibit a polynomial-time randomized machine  $M$  and a constant  $\epsilon < 1$  such that  $M$  recognizes  $L$  with one-sided error-probability  $\epsilon$ . We set  $M(x; R) = M'(x)$  for all  $R \in \{0, 1\}^{r(n)}$  and we set  $\epsilon = 0$ . (The length  $r(n)$  of the random tape of  $M$  can be set to  $r(n) = 1$  or even  $r(n) = 0$ .) Now note that if  $x \in L$  then  $\mathbf{AccPr}_M(x) = 1 \geq 1 - 0$  while if  $x \notin L$  then  $\mathbf{AccPr}_M(x) = 0$ . So the conditions of Definition 4.2 are trivially met.

For the second inclusion, assume  $L \in \mathbf{RP}$  and let  $M$  be a polynomial-time randomized machine recognizing  $L$  with one-sided error-probability  $\epsilon < 1$ . To show that  $L \in \mathbf{NP}$  we need to exhibit a verifier  $V$  for  $L$ . It works as follows:

Verifier  $V(x, R)$

If  $M(x; R)$  accepts then accept else reject

That is, the certificate provided is a string that  $V$  puts on the random tape of  $M$  before running  $M$  to see what is its outcome. We can now check that correctness of the verifier.

First assume  $x \in L$ . Then  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon$ , and since  $\epsilon < 1$  we have  $\mathbf{AccPr}_M(x) > 0$ . Now refer to Definition 4.1. It tells us that the set  $\mathbf{AccSet}_M(x)$  is not empty. (If it were empty we would have  $\mathbf{AccPr}_M(x) = 0$ .) Any member  $R$  of the above set is a valid certificate leading  $V$  to accept.

Next assume  $x \notin L$ . We claim that  $V(x, R)$  rejects no matter what  $R$  is provided. Why? We know that  $\mathbf{AccPr}_M(x) = 0$ . But this means the set  $\mathbf{AccSet}_M(x)$  is empty, or, in other words,  $M(x; R)$  rejects for all  $R$ . ■

### 4.3 The class BPP

We now consider that the algorithm might err even when  $x \notin L$ . This is called “two-sided” error-probability, while the **RP** condition was “one-sided” error-probability.

**Definition 4.5** Let  $\epsilon: \mathbb{N} \rightarrow [0, 1]$  be a function that takes input an integer and returns a real number in the range from 0 to 1. Let  $M$  be a randomized TM and  $L$  a language. We say that  $M$  recognizes  $L$  with two-sided error-probability  $\epsilon$  if for every  $x \in \Sigma^*$ :

- (1) If  $x \in L$  then  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon(|x|)$ .
- (2) If  $x \notin L$  then  $\mathbf{AccPr}_M(x) \leq \epsilon(|x|)$ .

Language  $L$  is said to be in the complexity-class **BPP** if there exists a constant (function)  $\epsilon < 1/2$  and a polynomial-time, randomized algorithm  $M$  such that  $M$  recognizes  $L$  with two-sided error-probability  $\epsilon$ . ■

Again we note that the error probabilities can be reduced. We will prove this using a probabilistic fact called the “Chernoff bound” which we state without proof. This bound is important to know in its own right, since it arises in many situations. (If you are interested in seeing a proof of the following, or want to know more about tail inequalities in general, see for example [1], but these items are not part of the syllabus of this course.)

**Fact 4.6 [Chernoff bound]** Let  $s \geq 1$  be an integer and let  $X_1, \dots, X_s$  be *independent* random variables taking values in the interval  $[0, 1]$ . Let  $X = X_1 + \dots + X_s$  and let  $a > 0$  be a real number. Then

$$\begin{aligned} \Pr[X \geq \mathbf{E}[X] + as] &\leq e^{-a^2s/2} \\ \Pr[X \leq \mathbf{E}[X] - as] &\leq e^{-a^2s/2} . \blacksquare \end{aligned}$$

Here  $\mathbf{E}[X]$  denotes the expectation of  $X$ . Note that by linearity of expectation

$$\mathbf{E}[X] = \mathbf{E}[X_1] + \dots + \mathbf{E}[X_s] .$$

**Example 4.7** Suppose we have a bin containing  $N$  balls. Of them,  $(1 - \epsilon)N$  are red and  $\epsilon N$  are blue, where  $0 \leq \epsilon < 1/2$ . We draw  $s$  balls at random (with replacement) from the bin. Let  $B$  denote the event that of these, the number of blue balls is strictly greater than  $s/2$ . We want to upper bound the probability of event  $B$ .

For  $i = 1, \dots, s$  we let  $X_i$  be the random variable taking value 1 if the  $i$ -th drawn ball is blue, and 0 otherwise. Now we note that  $X = X_1 + \dots + X_s$  counts the number of blue balls drawn. We also note that  $\mathbf{E}[X_i] = \epsilon$  for each  $i = 1, \dots, s$  and thus  $\mathbf{E}[X] = \epsilon s$  by linearity of expectation. Then

$$\begin{aligned} \Pr[B] &= \Pr\left[X > \frac{s}{2}\right] \\ &= \Pr\left[X > \epsilon s + \frac{1-2\epsilon}{2}s\right] \\ &= \Pr\left[X > \mathbf{E}[X] + \frac{1-2\epsilon}{2}s\right] \\ &\leq e^{-(1-2\epsilon)^2 s/8} . \blacksquare \end{aligned}$$

**Proposition 4.8** Suppose  $L \in \mathbf{BPP}$ . Then there is a randomized polynomial-time TM  $M$  such that for all  $x \in \Sigma^*$  and all  $k \in \mathbb{N}$

- (1) If  $x \in L$  then  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) \geq 1 - 2^{-k}$ .
- (2) If  $x \notin L$  then  $\mathbf{AccPr}_M(\langle x, 1^k \rangle) \leq 2^{-k}$ .

We call  $M$  a *parameterized randomized TM* recognizing  $L$  with two-sided error.  $\blacksquare$

**Proof of Proposition 4.8:** Since  $L \in \mathbf{BPP}$  there exists a machine  $M'$  and a constant  $0 \leq \epsilon < 1/2$  such that the conditions of Definition 4.5 are met. Suppose  $M'$  uses  $r(n)$  coin tosses. We design randomized machine  $M$  as follows:

Algorithm  $M(\langle x, 1^k \rangle; R)$

Let  $s \leftarrow \lceil 8k/(1 - 2\epsilon)^2 \rceil$

For  $i = 1, \dots, s$  do

Let  $R_i$  be the next  $r(|x|)$  bits from the random tape  $R$

Let  $d_i = 1$  if  $M'(x; R_i)$  accepts and 0 otherwise

EndFor

$d \leftarrow d_1 + d_2 + \dots + d_s$

If  $d \geq s/2$  then accept else reject

The analysis is by analogy to the above example. We let  $N = 2^{r(n)}$  and think of  $\{0, 1\}^{r(n)}$  as the set of balls. In the case that  $x \in L$  we view a ball  $R \in \{0, 1\}^{r(n)}$  as having color red if  $R \in \mathbf{AccSet}_{M'}(x)$  and blue otherwise. In the case  $x \notin L$  we view a ball  $R \in \{0, 1\}^{r(n)}$  as having color blue if  $R \in \mathbf{AccSet}_{M'}(x)$  and red otherwise. The details follow.

First suppose  $x \in L$ . For  $i = 1, \dots, s$  we let  $X_i$  be the random variable whose value is  $1 - d_i$ , where  $d_i$  is the bit computed in the algorithm above. We let  $X = X_1 + \dots + X_s$ . We note that  $\mathbf{E}[X_i] = 1 - \mathbf{AccPr}_{M'}(x) \leq \epsilon$  since  $x \in L$ . By linearity of expectation,  $\mathbf{E}[X] \leq \epsilon s$ . Then

$$1 - \mathbf{AccPr}_M(\langle x, 1^k \rangle) = \Pr\left[d < \frac{s}{2}\right]$$

$$\begin{aligned}
&= \Pr \left[ s - d > \frac{s}{2} \right] \\
&= \Pr \left[ X > \frac{s}{2} \right] \\
&= \Pr \left[ X > \epsilon s + \frac{1-2\epsilon}{2}s \right] \\
&\leq \Pr \left[ X > \mathbf{E}[X] + \frac{1-2\epsilon}{2}s \right] \\
&\leq e^{-(1-2\epsilon)^2 s/8} \\
&\leq e^{-k} \\
&\leq 2^{-k} .
\end{aligned}$$

Now suppose  $x \notin L$ . For  $i = 1, \dots, s$  we let  $X_i$  be the random variable whose value is  $d_i$ , the bit computed in the algorithm above. We let  $X = X_1 + \dots + X_s$ . We note that  $\mathbf{E}[X_i] = \mathbf{AccPr}_{M'}(x) \leq \epsilon$  since  $x \notin L$ . By linearity of expectation,  $\mathbf{E}[X] \leq \epsilon s$ . Then

$$\begin{aligned}
\mathbf{AccPr}_M(\langle x, 1^k \rangle) &= \Pr \left[ d \geq \frac{s}{2} \right] \\
&= \Pr \left[ X \geq \frac{s}{2} \right] \\
&= \Pr \left[ X \geq \epsilon s + \frac{1-2\epsilon}{2}s \right] \\
&\leq \Pr \left[ X \geq \mathbf{E}[X] + \frac{1-2\epsilon}{2}s \right] \\
&\leq e^{-(1-2\epsilon)^2 s/8} \\
&\leq e^{-k} \\
&\leq 2^{-k} .
\end{aligned}$$

This concludes the proof.  $\blacksquare$

This time, there is no obvious relation to **NP**. The few known relations of **BPP** to other complexity-classes follow.

**Proposition 4.9**  $\mathbf{RP} \subseteq \mathbf{BPP}$ .  $\blacksquare$

**Proof:** Let  $M$  be a randomized polynomial-time TM that recognizes  $L \in \mathbf{RP}$  with one-sided error-probability  $\epsilon$ . We note that as long as  $\epsilon < 1/2$ , the same machine  $M$  also recognizes  $L$  with two-sided error-probability  $\epsilon$ . (You should check this out by looking at the definitions.) We can assume  $\epsilon < 1/2$  because of Proposition 4.3.  $\blacksquare$

**Proposition 4.10**  $\mathbf{BPP} \subseteq \mathbf{PSPACE}$ .  $\blacksquare$

**Proof:** Let  $M$  be a randomized polynomial-time TM that recognizes  $L \in \mathbf{BPP}$  with two-sided error-probability  $\epsilon < 1/2$ . Let  $r(n)$  be the number of coin tosses of  $M$ . Then the following TM decides  $L$ :

Algorithm  $M'(x)$

$n \leftarrow |x|$ ;  $s \leftarrow 0$

For all  $R \in \{0, 1\}^{r(n)}$  do

```

    If  $M(x; R)$  accepts then  $s \leftarrow s + 1$ 
  EndFor
  If  $s \geq (1 - \epsilon)2^{r(n)}$  then accept else reject

```

By re-using space across executions of the For loop, this can be implemented in polynomial space. ■

#### 4.4 Closure properties

As usual there are “co” classes

**Definition 4.11**  $L$  is in **coRP** iff  $\bar{L} \in \mathbf{RP}$ .  $L$  is in **coBPP** iff  $\bar{L}$  is in **BPP**. ■

**Proposition 4.12** **BPP** is closed under complement, ie. **coBPP** = **BPP**. ■

**Proof:** Symmetry. Just reversing the answer is OK. Here are the details.

Let  $L$  be a language in **BPP**. We want to show that  $\bar{L}$  is also in **BPP**.

Since  $L \in \mathbf{BPP}$  there is a randomized, polynomial-time machine  $M$  and a constant  $0 \leq \epsilon < 1/2$  such that  $M$  recognizes  $L$  with two-sided error-probability  $\epsilon$ . Now consider the following randomized machine  $M'$ –

```

Algorithm  $M'(x; R)$ 
  If  $M(x; R)$  accepts then reject else accept

```

We claim that  $M'$  recognizes  $\bar{L}$  with two-sided error-probability  $\epsilon$ . To see this first note that on any input  $x$  (regardless of whether or not it is in  $L$ ) we have

$$\mathbf{AccPr}_{M'}(x) = 1 - \mathbf{AccPr}_M(x).$$

Now we can consider the two cases for  $x$ . If  $x \in \bar{L}$  then we know that  $\mathbf{AccPr}_M(x) \leq \epsilon$ . Hence

$$\mathbf{AccPr}_{M'}(x) = 1 - \mathbf{AccPr}_M(x) \geq 1 - \epsilon.$$

If  $x \notin \bar{L}$  then we know that  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon$ . Hence

$$\mathbf{AccPr}_{M'}(x) = 1 - \mathbf{AccPr}_M(x) \leq 1 - (1 - \epsilon) = \epsilon.$$

This shows that  $M'$  recognizes  $\bar{L}$  with two-sided error-probability  $\epsilon$ . ■

We do not know whether or not **RP** is closed under complement. However the following are pretty easy to see.

**Proposition 4.13** The following closure properties hold:

- (1) **RP** is closed under union

- (2) **RP** is closed under intersection
- (3) **BPP** is closed under union
- (4) **BPP** is closed under intersection

We will prove parts (1) and (4), leaving the other parts as exercises.

**Proof of Proposition 4.13, Part (2):** Let  $L_1, L_2 \in \mathbf{RP}$ . We want to show that  $L_1 \cap L_2 \in \mathbf{RP}$ .

Definition 4.2 tells us that there exist randomized polynomial-time TMs  $M_1, M_2$ , polynomials  $r_1, r_2$ , and constants  $\epsilon_1, \epsilon_2 < 1$  such that  $M_1$  uses  $r_1(n)$  coins and recognizes  $L_1$  with one-sided error-probability  $\epsilon_1$ , and  $M_2$  uses  $r_2(n)$  coins and recognizes  $L_2$  with one-sided error-probability  $\epsilon_2$ . We now define the following randomized TM  $M$  that takes inputs  $x$  and a random tape  $R$ :

Algorithm  $M(x; R)$

```

 $n \leftarrow |x|$ 
Let  $R_1$  be the first  $r_1(n)$  bits of  $R$ 
Let  $R_2$  be the next  $r_2(n)$  bits of  $R$ 
If ( $M_1(x; R_1)$  accepts and  $M_2(x; R_2)$  accepts) then accept else reject

```

We need to show that there exists a constant  $\epsilon < 1$  such that  $M$  recognizes  $L_1 \cap L_2$  with one-sided error-probability  $\epsilon$ . We set

$$\epsilon = 1 - (1 - \epsilon_1)(1 - \epsilon_2). \quad (3)$$

Since  $\epsilon_1, \epsilon_2$  are constants strictly less than 1, the quantity  $(1 - \epsilon_1)(1 - \epsilon_2)$  is a constant strictly greater than 0, and thus  $\epsilon$  defined above is a constant strictly less than 1. Now, we observe that for every  $x \in \Sigma^*$

$$\mathbf{AccPr}_M(x) = \mathbf{AccPr}_{M_1}(x) \cdot \mathbf{AccPr}_{M_2}(x). \quad (4)$$

To complete the proof we will now establish that for every  $x \in \Sigma^*$ :

- (1) If  $x \in L_1 \cap L_2$  then  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon$ .
- (2) If  $x \notin L_1 \cap L_2$  then  $\mathbf{AccPr}_M(x) = 0$ .

So first suppose  $x \in L_1 \cap L_2$ . In that case we know that  $\mathbf{AccPr}_{M_1}(x) \geq 1 - \epsilon_1$  and  $\mathbf{AccPr}_{M_2}(x) \geq 1 - \epsilon_2$ . So from Equation (4) and Equation (3) we get

$$\mathbf{AccPr}_M(x) = \mathbf{AccPr}_{M_1}(x) \cdot \mathbf{AccPr}_{M_2}(x) \geq (1 - \epsilon_1)(1 - \epsilon_2) = 1 - \epsilon.$$

Now suppose  $x \notin L_1 \cap L_2$ . In that case we know that either  $x \notin L_1$  or  $x \notin L_2$ . So either  $\mathbf{AccPr}_{M_1}(x) = 0$  or  $\mathbf{AccPr}_{M_2}(x) = 0$ . So from Equation (4) we get  $\mathbf{AccPr}_M(x) = 0$ . **■**

Now let us look at proving Part (4), namely that **BPP** is also closed under intersection. Let us try to proceed as above. Letting  $L_1, L_2 \in \mathbf{BPP}$ , we want to show that  $L_1 \cap L_2 \in \mathbf{BPP}$ . Definition 4.5 tells us that there exist randomized polynomial-time TMs  $M_1, M_2$ , polynomials  $r_1, r_2$ , and constants  $\epsilon_1, \epsilon_2 < 1/2$  such that  $M_1$  uses  $r_1(n)$  coins and recognizes  $L_1$  with two-sided error-probability  $\epsilon_1$ , and  $M_2$  uses  $r_2(n)$  coins and recognizes  $L_2$  with two-sided error-probability  $\epsilon_2$ . Let us now define  $M$  as above. However, we can no longer set  $\epsilon$  as in Equation (3) above. The problem is that we

need  $\epsilon$  to be strictly less than  $1/2$ , and Equation (3) does not guarantee this. For example, it could be that  $\epsilon_1 = \epsilon_2 = 3/8 < 1/2$ . In that case we get

$$\epsilon = 1 - (1 - \epsilon_1)(1 - \epsilon_2) = 1 - \frac{5}{8} \cdot \frac{5}{8} = 1 - \frac{25}{64} = \frac{38}{64} > \frac{1}{2}.$$

In fact, TM  $M$  is simply not succeeding in recognizing  $L_1 \cap L_2$  with two-sided error-probability strictly less than  $1/2$ . We must use a different TM. However, the basic strategy of  $M$  is correct; the problem is simply that the machines  $M_1, M_2$  err too often. We can correct this by appealing to Proposition 4.8 and replacing  $M_1, M_2$  with parameterized TMs that make the error low enough that the value of  $\epsilon$  given by the above equation ends up being strictly less than  $1/2$ . Let us now proceed to the actual proof showing how to do this.

**Proof of Proposition 4.13, Part (2):** Let  $L_1, L_2 \in \mathbf{BPP}$ . We want to show that  $L_1 \cap L_2 \in \mathbf{BPP}$ . Proposition 4.8 tells us that there exist parameterized, randomized polynomial-time TMs  $M_1, M_2$ , using some number of coins  $r_1, r_2$  respectively, such that  $M_1$  recognizes  $L_1$  with two-sided error and  $M_2$  recognizes  $L_2$  with two-sided error. We now define the following randomized TM  $M$  that takes inputs  $x$  and a random tape  $R$ :

Algorithm  $M(x; R)$

$k \leftarrow 2$

Let  $R_1$  be the first  $r_1(|\langle x, 1^k \rangle|)$  bits of  $R$

Let  $R_2$  be the next  $r_2(|\langle x, 1^k \rangle|)$  bits of  $R$

If  $(M_1(\langle x, 1^k \rangle; R_1)$  accepts and  $M_2(\langle x, 1^k \rangle; R_2)$  accepts) then accept else reject

The above sets the security parameter  $k$  to 2, so that the parameterized machines have error-probability  $2^{-2} = 1/4$ . Now we need to show that there exists a constant  $\epsilon < 1/2$  such that  $M$  recognizes  $L_1 \cap L_2$  with two-sided error-probability  $\epsilon$ . We set  $\epsilon = 7/16$ , which is strictly less than  $1/2$ . Now, we observe that for every  $x \in \Sigma^*$

$$\mathbf{AccPr}_M(x) = \mathbf{AccPr}_{M_1}(\langle x, 1^k \rangle) \cdot \mathbf{AccPr}_{M_2}(\langle x, 1^k \rangle). \quad (5)$$

To complete the proof we will now establish that for every  $x \in \Sigma^*$ :

(1) If  $x \in L_1 \cap L_2$  then  $\mathbf{AccPr}_M(x) \geq 1 - \epsilon = 9/16$ .

(2) If  $x \notin L_1 \cap L_2$  then  $\mathbf{AccPr}_M(x) \leq \epsilon = 7/16$ .

So first suppose  $x \in L_1 \cap L_2$ . In that case we know that  $\mathbf{AccPr}_{M_1}(\langle x, 1^k \rangle) \geq 1 - 1/4 = 3/4$  and  $\mathbf{AccPr}_{M_2}(\langle x, 1^k \rangle) \geq 1 - 1/4 = 3/4$ . So from Equation (5) we get

$$\mathbf{AccPr}_M(x) = \mathbf{AccPr}_{M_1}(\langle x, 1^k \rangle) \cdot \mathbf{AccPr}_{M_2}(\langle x, 1^k \rangle) \geq \frac{3}{4} \cdot \frac{3}{4} = \frac{9}{16}.$$

Now suppose  $x \notin L_1 \cap L_2$ . In that case we know that either  $x \notin L_1$  or  $x \notin L_2$ . So either  $\mathbf{AccPr}_{M_1}(\langle x, 1^k \rangle) \leq 1/4$  or  $\mathbf{AccPr}_{M_2}(\langle x, 1^k \rangle) \leq 1/4$ . So of the two terms in the product in Equation (4), one is always at most  $1/4$ . Since the other is at most 1, the product is at most  $1/4$ , meaning

$$\mathbf{AccPr}_M(x) = \mathbf{AccPr}_{M_1}(\langle x, 1^k \rangle) \cdot \mathbf{AccPr}_{M_2}(\langle x, 1^k \rangle) \leq \frac{1}{4} \leq \frac{7}{16}$$

as desired. ■

## 5 Primality

We now turn to another example. Recall that an integer  $N \geq 1$  is *prime* if it has exactly two divisors in the range  $1, \dots, N$ , namely 1 and  $N$ . (Note 1 is not prime but 2 is prime and 2 is the only even prime.) The language we want to look at is:

$$\text{PRIMES} = \{ \langle N \rangle : N \text{ is a prime number} \}.$$

The corresponding problem is:

### PRIMES

**Input:**  $\langle N \rangle$  where  $N \geq 1$  is an integer

**Question:** Is  $N$  prime?

We are interested in the computational complexity of this problem. Before discussing this it is worth clarifying how complexity is measured. As always, for the purpose of computation, inputs to an algorithm are encoded in binary. This means that an algorithm gets input the binary representation  $\langle N \rangle$  of an integer  $N$  and must decide whether or not  $N$  is prime. Running time of an algorithm is measured, as always, as a function of the length of the binary string that is input to the algorithm. This length is usually denoted  $n$ . When the input is an integer  $N$ , this length  $n$  is the number of bits in its binary representation, meaning the unique integer  $n$  satisfying  $2^{n-1} \leq N < 2^n$ . That is,  $n = 1 + \lfloor \lg(N) \rfloor$ , or, roughly,  $n \approx \lg(N)$ , where  $\lg(\cdot)$  denotes the logarithm to base two.

It is important to note that running time is measured as a function of  $\lg(N)$ , not  $N$ , and in particular polynomial-time means polynomial in  $\lg(N)$ , not polynomial in  $N$ . This explains the following.

At first glance it may seem that PRIMES is in  $\mathbf{P}$ , via the naive algorithm of trying all possible divisors  $i = 2, \dots, N - 1$ . Namely consider the following algorithm that takes input an integer  $N \geq 1$ :

Algorithm Try1( $\langle N \rangle$ )

  For  $i = 2, \dots, N - 1$  do

    If  $i$  divides  $N$  then reject

  End for

  Accept

This algorithm rejects if  $N$  is composite, and accepts if  $N$  is prime, and hence decides PRIMES. However, this is an exponential time algorithm. Its running time is polynomial in  $N$ , which is exponential in  $n \approx \lg(N)$ .

One might attempt to improve this by reducing the set of  $i$  that are tested as divisors. For example, it suffices to let  $i$  range from 2 to  $\sqrt{N}$ . Also, observe that if 2 does not divide  $N$  then neither does any multiple of two, reducing by half the number of  $i$  values to consider. Similarly if 3 does not divide  $N$  we do not have to check any multiples of 3. Yet, even with these kinds of tricks, there is no known way to get the above to run in polynomial time.

## 5.1 Results on the computational complexity of primality

Here are the basic known facts:

**Theorem 5.1** PRIMES is in all the following complexity classes:

- (1) **NP**
- (2) **coNP**
- (3) **RP**
- (4) **coRP**
- (5) **P**. ■

One might note that Theorem 5.1 is a rather “redundant” statement. After all, if PRIMES is in **P**, then, since we know that **P** is a subset of all the other classes appearing in the above theorem, why explicitly state that PRIMES is in these too? There are a few reasons we do this.

One reason is historical. The primality problem has attracted a great deal of attention, and results have improved with time. The first result was that PRIMES is in **NP**. Later it was shown to be in **coRP**, and still later in **RP**. (That it is in **coNP** is trivial.) Only recently was it finally shown that PRIMES is in **P**.

Moreover, all these results represent and exemplify interesting and different techniques. The proof that PRIMES is in **NP** is one of the more non-trivial proofs of membership in **NP**, exploiting interesting aspects of the mathematics of number-theory. (As we have seen when we studied **NP**-completeness, membership of languages in **NP** is usual trivial. But certificates for primality are not obvious.) The randomized algorithms showing PRIMES is in **RP**  $\cap$  **coRP** again illustrate interesting use of number theory. The proof that PRIMES is in **P** illustrates use of different types of number theory.

Primality is an old problem that seemed for a long time to be of merely theoretical interest. Nowadays, however, there are important practical reasons to care about it. Primality surfaces in the domain of cryptography and security. Numerous public-key cryptosystems, including the popular RSA system, need to generate large prime numbers at key-generation time. The process employed is to repeatedly pick numbers at random and test whether or not they are prime, until a prime is found. Security of the cryptosystem requires that the primes be large, like 1024-bits long. (Meaning, a prime  $p \approx 2^{1024}$ .)

RSA public-key cryptography is employed in software such as SSH, which we use, for example, in the Computer Science Department at UCSD for remote login. RSA is also used in SSL, which is widely used to secure the transmission of credit card numbers in Internet purchases. In all cases, primality tests are part of the software and are run at key generation time. For these reasons, we want fast algorithms for testing primality.

However, the deterministic algorithm showing PRIMES is in **P** is slow, having running time  $O(n^8)$ . The **coRP** algorithm however is faster, with running time  $O(n^3)$ . So in practice the latter is used. This is another reason we continue to care about the **RP** and **coRP** algorithms even given that PRIMES is in **P**.

Below we will look into the proof that PRIMES is in **NP** and then look at a  $O(n^3)$  **coRP** algorithm for PRIMES. The treatment will invoke various number-theoretic facts without proof.

## 5.2 Some basic computational number theory

We list some definitions and facts, usually without proof.

If  $a, N$  are numbers then  $\gcd(a, N)$  is their greatest common divisor. Eg.  $\gcd(6, 21) = 3$ . We say that  $a, N$  are relatively prime if  $\gcd(a, N) = 1$ . Note that if  $N$  is prime and  $1 < a < N$  then  $\gcd(a, N) = 1$ .

For any integer  $N \geq 1$  we let

$$\begin{aligned}\mathbb{Z}_N^* &= \{a \in \mathbb{N} : 1 \leq a < N \text{ and } \gcd(a, N) = 1\} \\ \varphi(N) &= |\mathbb{Z}_N^*|.\end{aligned}$$

That is,  $\mathbb{Z}_N^*$  is the set of all integers in the range  $1, \dots, N - 1$  that are relatively prime to  $N$ , and  $\varphi(N)$  is the size of this set, meaning the number of integers in the range  $1, \dots, N - 1$  that are relatively prime to  $N$ . It is common to call  $|\mathbb{Z}_N^*|$  the *order* of  $\mathbb{Z}_N^*$ . We refer to  $\varphi$  as the Euler Phi function. Note that if  $p$  is prime then  $\varphi(p) = p - 1$ .

**Example 5.2** If  $N = 21$  then  $\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$ , and  $\varphi(21) = 12$ . ■

We consider the operation of multiplication modulo  $N$ , which on input  $a, b \in \mathbb{N}$  returns  $ab \bmod N$ , the remainder upon division of  $ab$  by  $N$ .

**Fact 5.3**  $\mathbb{Z}_N^*$  is a *group* under multiplication modulo  $N$ . ■

If you know what is a group from basic algebra you know what the above means. If not, here briefly is what this is saying:

- (1) For any  $a, b \in \mathbb{Z}_N^*$ , the number  $ab \bmod N$  is also in  $\mathbb{Z}_N^*$ . (For example if  $N = 21$  then  $5 * 11 \bmod 21 = 13$ , which is in  $\mathbb{Z}_{21}^*$ . Multiplying two things in  $\mathbb{Z}_N^* \bmod N$  will never yield a number not in  $\mathbb{Z}_N^*$ .)
- (2) For every  $a \in \mathbb{Z}_N^*$  there exists a unique  $b \in \mathbb{Z}_N^*$  such that  $ab \equiv ba \equiv 1 \pmod{N}$ . This  $b$  is denoted  $a^{-1}$  and called the inverse of  $a$ . (For example,  $8 * 8 \bmod 21 = 1$  so  $8^{-1} = 8$  in  $\mathbb{Z}_{21}^*$ .)

We can now raise any element of  $\mathbb{Z}_N^*$  to an integer power modulo  $N$ . Namely for any  $a \in \mathbb{Z}_N^*$  and any integer  $i \geq 1$  we have

$$a^i \bmod N = \underbrace{a \cdot a \cdots a}_{i} \bmod N.$$

The closure property of the group  $\mathbb{Z}_N^*$  implies that  $a^i \bmod N \in \mathbb{Z}_N^*$ . We adopt the convention that  $a^0 = 1$ . We also adopt the convention that  $a^{-i} = (a^{-1})^i$ , where  $a^{-1}$  denotes the multiplicative inverse of  $a$  in the group  $\mathbb{Z}_N^*$ .

**Fact 5.4** For any  $a \in \mathbb{Z}_N^*$  it is the case that  $a^{\varphi(N)} \equiv 1 \pmod{N}$ . ■

$i$	$2^i \bmod 7$	$3^i \bmod 7$
0	1	1
1	2	3
2	4	2
3	1	6
4	2	4
5	4	5
6	1	1

Figure 2: Exponentiation modulo 7 illustrating that 3 is a generator of  $\mathbb{Z}_7^*$ .

This is a corollary of a simple group-theoretic fact: any element of a group raised to the power the order (size) of the group gives you back the identity. One corollary of this is that exponents can be taken modulo  $\varphi(N)$ . That is, for any  $a \in \mathbb{Z}_N^*$  and any  $i \geq 0$  we have

$$a^i \bmod N = a^{i \bmod \varphi(N)} \bmod N .$$

We highlight the case of the above fact where  $N$  is prime:

**Fact 5.5** If  $p$  is prime then for all  $a \in \mathbb{Z}_p^*$  we have  $a^{p-1} \equiv 1 \pmod{N}$ . ■

Why? Because if  $p$  is prime then  $\varphi(p) = p - 1$ .

Suppose  $g \in \mathbb{Z}_N^*$  and consider the sequence  $g^0, g^1, g^2, \dots$  all taken modulo  $N$ . The closure property of the group  $\mathbb{Z}_N^*$  tells us that all the points in this sequence are in  $\mathbb{Z}_N^*$ . Since the latter is a finite set, the sequence has to wrap around at some point. We denote by  $\mathbf{ord}_N(g)$  the smallest integer  $m \geq 1$  such that  $g^m \bmod N = 1$ . We say that  $g$  is a *generator* if  $\mathbf{ord}_N(g) = \varphi(N)$ . When  $g$  is a generator we have

$$\{ g^i \bmod N : i = 0, \dots, \varphi(N) - 1 \} = \mathbb{Z}_N^* .$$

We say that the group  $\mathbb{Z}_N^*$  is *cyclic* if it has a generator. For any  $a \in \mathbb{Z}_N^*$ , the unique  $i$  such that  $g^i \bmod N = a$  and  $0 \leq i < \varphi(N)$  is called the *discrete logarithm* of  $a$  to base  $g$  modulo  $N$ . Here is another basic group-theoretic fact:

**Fact 5.6** Suppose  $g \in \mathbb{Z}_N^*$ . Then  $\mathbf{ord}_N(g)$  divides  $\varphi(N)$ . ■

Figure 2 illustrates exponentiation modulo 7. It shows that  $\mathbf{ord}_7(2) = 3$  and  $\mathbf{ord}_7(3) = 6$ . Since  $\varphi(7) = 6$  this means that 3 is a generator of  $\mathbb{Z}_7^*$ , meaning  $\mathbb{Z}_7^*$  is cyclic.

**Fact 5.7** If  $p$  is prime then  $\mathbb{Z}_p^*$  is cyclic. ■

The converse, however is not true. For example,  $\mathbb{Z}_9^*$  is cyclic but 9 is not prime. Primality can however be characterized as follows.

**Fact 5.8** Let  $N \geq 2$  be an integer and let  $q_1^{\alpha_1} \dots q_k^{\alpha_k}$  be the prime factorization of  $N - 1$ . Then the following are equivalent:

- (1)  $N$  is prime
- (2)  $\exists g \in \mathbb{Z}_N^* [\text{ord}_N(g) = N - 1]$
- (3)  $\exists g \in \mathbb{Z}_N^* [g^{N-1} \bmod N = 1 \text{ and } \forall i : g^{(N-1)/q_i} \bmod N \neq 1]$  ■

When we say that the prime factorization of an integer  $M$  is  $q_1^{\alpha_1} \dots q_k^{\alpha_k}$  we mean that  $q_1 < q_2 < \dots < q_k$  are primes and  $\alpha_1, \dots, \alpha_k \geq 1$  are integers such that  $M = q_1^{\alpha_1} \dots q_k^{\alpha_k}$ . Any positive integer has a unique prime factorization. We stress that above the prime factorization is that of  $N - 1$ , not that of  $N$ .

The second condition above is saying that  $\mathbb{Z}_N^*$  has order  $N - 1$  and  $g$  is a generator of  $\mathbb{Z}_N^*$ , meaning  $\mathbb{Z}_N^*$  is cyclic. The third condition provides a way to test the second condition, reducing it to testing  $k + 1$  modular relations. It turns out that this test can be performed efficiently given the prime factorization of  $N - 1$  by using the following.

**Fact 5.9** The following are polynomial-time computable functions, where  $N, i \geq 1$  are integers and  $a, b \in \mathbb{Z}$ :

- (1)  $(a, b) \mapsto \text{gcd}(a, b)$
- (2)  $(a, N) \mapsto a \bmod N$
- (3)  $(a, N) \mapsto a^{-1} \bmod N$ , for  $a$  such that  $\text{gcd}(a, N) = 1$
- (4)  $(N, a, b) \mapsto ab \bmod N$
- (5)  $(N, a, i) \mapsto a^i \bmod N$  ■

The gcd is computed via Euclid's classical algorithm, and an extension of this algorithm enables the computation of multiplicative inverses. For the multiplication, use the standard high school algorithm. For exponentiation, repeated multiplication leads to an exponential time algorithm; you have to be more clever. The method used is called repeated squaring.

We now have enough in hand to prove that PRIMES is in NP.

### 5.3 PRIMES is in NP

We need to specify a verifier for PRIMES. Let us begin by discussing the form of a certificate (witness) proving primality of  $N$ . The certificate provided by the prover (alien) would contain  $g$  and  $(q_1, \alpha_1), \dots, (q_k, \alpha_k)$  where  $g$  is an element of  $\mathbb{Z}_N^*$  of order  $N - 1$  and the prime factorization of  $N - 1$  is  $q_1^{\alpha_1} \dots q_k^{\alpha_k}$ . The verifier can check the conditions listed in Fact 5.8 above, using the polynomial-time computability of the functions listed in Fact 5.9. However, it must also check that  $q_1^{\alpha_1} \dots q_k^{\alpha_k}$  really is the prime factorization of  $N - 1$ . This is done by having the certificate for  $N$  include a certificate  $\text{cert}_i$  proving  $q_i$  is prime for each  $i = 1, \dots, k$ . These certificates are determined and verified recursively. The recursion bottoms out at the prime 2.

**Example 5.10** Let  $N = 31$  which is prime. The prime factorization of  $N - 1 = 30$  is  $2^1 \cdot 3^1 \cdot 5^1$ . The number 12 turns out to be a generator of the cyclic group  $\mathbb{Z}_{31}^*$ . Accordingly a certificate proving

Verifier  $V_{\text{PRIMES}}(N, \text{cert})$

- If  $N = 2$  then accept
- If  $N$  is even then reject
- Parse cert as  $(g, (q_1, \alpha_1, \text{cert}_1), \dots, (q_k, \alpha_k, \text{cert}_k))$
- If all the following hold then accept else reject:
  1.  $1 \leq g < N - 1$  and  $\gcd(g, N) = 1$
  2.  $g^{N-1} \bmod N = 1$
  3. For all  $i$ :  $g^{(N-1)/q_i} \bmod N \neq 1$
  4. For all  $i$ :  $\alpha_i \geq 1$
  5.  $N - 1 = q_1^{\alpha_1} \cdots q_k^{\alpha_k}$
  6. For all  $i$ :  $V_{\text{PRIMES}}(q_i, \text{cert}_i)$  accepts

Figure 3: Verifier for PRIMES, showing PRIMES is in **NP**.

primality of 31 could have the form

$$\text{cert} = (12, (2, 1, \varepsilon), (3, 1, \text{cert}_2), (5, 1, \text{cert}_3)) .$$

Here  $\text{cert}_2$  proves primality of 3 and  $\text{cert}_3$  proves primality of 5. Since  $2 = 2^1$  is the prime factorization of  $3 - 1 = 2$  and 2 is a generator of  $\mathbb{Z}_3^*$  we can set

$$\text{cert}_2 = (2, (2, 1, \varepsilon)) .$$

Since  $4 = 2^2$  is the prime factorization of  $5 - 1 = 4$  and 3 is a generator of  $\mathbb{Z}_5^*$  we can set

$$\text{cert}_3 = (3, (2, 2, \varepsilon)) .$$

In summary

$$\text{cert} = (12, (2, 1, \varepsilon), (3, 1, (2, (2, 1, \varepsilon))), (5, 1, (3, (2, 2, \varepsilon))))$$

is a certificate proving primality of 31. Verification would involve checking that  $12 \in \mathbb{Z}_{31}^*$  and

$$12^{30} \bmod 31 = 1 \text{ and } 12^{15} \bmod 31 \neq 1 \text{ and } 12^{10} \bmod 31 \neq 1 \text{ and } 12^6 \bmod 31 \neq 1 .$$

Then, one would check  $30 = 2^1 \cdot 3^1 \cdot 5^1$ , and finally verify the sub-certificates recursively. ■

A verifier  $V_{\text{PRIMES}}$  for PRIMES is depicted in Figure 3. It is a recursive algorithm. We need to argue that it is correct and also that it is computable in time polynomial in the length of its first input  $N$ .

Correct means that if  $N$  is prime then there is a certificate cert such that  $V_{\text{PRIMES}}(N, \text{cert})$  accepts, and, on the other hand, if  $N$  is composite, then for any cert, the computation  $V_{\text{PRIMES}}(N, \text{cert})$  rejects. For the first part, a certificate for prime  $N$  could be recursively created by the following process:

MakeCert( $N$ )

- If  $N = 2$  then return  $\varepsilon$
- Let  $g$  be a generator of  $\mathbb{Z}_N^*$
- Let  $q_1^{\alpha_1} \cdots q_k^{\alpha_k}$  be the prime factorization of  $N - 1$

$a$	$a^2 \bmod 21$
1	1
2	4
4	16
5	4
8	1
10	16
11	16
13	1
16	4
17	16
19	4
20	1

Figure 4: Results of squaring elements in  $Z_{21}^*$ . The elements in the second column are the squares modulo 21.

```

For  $i = 1, \dots, k$  do  $\text{cert}_i \leftarrow \text{MakeCert}(q_i)$  EndFor
Return  $(g, (q_1, \alpha_1, \text{cert}_1), \dots, (q_k, \alpha_k, \text{cert}_k))$ 

```

Note that the process is not deterministic. It chooses a generator in an unspecified way. That's ok, since it is being run by an all powerful alien.

If  $N$  is not prime, Fact 5.8 together with the description of the verifier tell us that there is no cert that will make the verifier accept input  $N, \text{cert}$ .

With regard to running time, we argue that the total number of times the verifier algorithm is invoked (due to recursive calls) when started on input  $N, \text{cert}$  is at most  $n = |N|$ , the length of the binary representation of  $N$ . The overhead in each call is  $\text{poly}(n)$  due to Fact 5.9 so this implies that the verifier runs in  $\text{poly}(n)$  time. (Actually we have to be a bit careful about the implementation. For example, in parsing  $\text{cert}$ , the verifier should not continue to read its length is above some suitable  $\text{poly}(n)$  amount.)

To substantiate the claim about the recursion, consider a tree whose nodes are labeled by integers. The root is labeled by the input  $N$  whose primality is being tested. A node has label  $M$  has  $k$  children labeled  $q_1, \dots, q_k$  respectively, where  $q_1^{\alpha_1} \cdots q_k^{\alpha_k}$  is the prime factorization of  $M - 1$ . A node with label 2 is a leaf. The number of recursive calls in the computation is the number of nodes in the tree. Let  $L$  denote the number of leaves. Note that the label of a node is always greater than the product of the labels of its children. Since leaves are labeled with 2, a consequence is that  $2^L \leq N$ , meaning the number of leaves is at most  $|N|$ . It follows that the number of nodes in the tree is  $O(|N|)$ , which is what we wanted to show.

## 5.4 More computational number theory

Towards showing PRIMES is in **coRP** we look at some more number theory.

Let us say that  $a \in \mathbb{Z}_N^*$  is a *quadratic residue* or *square* if there is a  $b \in \mathbb{Z}_N^*$  such that  $b^2 \equiv a \pmod{N}$ . Not all numbers in  $\mathbb{Z}_N^*$  are quadratic residues. For example, Figure 4 shows the squares of all elements in  $\mathbb{Z}_{21}^*$ . We see that the quadratic residues in  $\mathbb{Z}_{21}^*$  are 1, 4, 16. Notice a square can have more than one square root and they all have the same number of square roots. Notice that the square roots occur in pair of the form  $x, -x$ ; for example,  $-1 \equiv 20 \pmod{21}$  so 20 is a square root of 1, and  $-4 \equiv 17 \pmod{21}$  so 17 is a square root of  $4^2 = 16$ .

If  $p$  is prime and  $x \in \mathbb{Z}_N^*$  we let  $J_p(x) = 1$  if  $x$  is a square and  $-1$  otherwise. We call this the Legendre symbol. Note it is defined only for primes  $p$ .

**Fact 5.11** If  $p$  is prime then  $a^{(p-1)/2} \pmod{p} \equiv J_p(a) \pmod{p}$  for all  $a \in \mathbb{Z}_N^*$ . ■

Fact 5.9 and the above imply that one can compute the Legendre symbol in polynomial time given  $p, a$ .

Now, we extend the Legendre symbol to non-primes in a very simple way. If  $N$  is an arbitrary integer, let  $N = q_1^{\alpha_1} \cdots q_k^{\alpha_k}$  denote its prime factorization. We define the Jacobi symbol of  $x \in \mathbb{Z}_N^*$  by

$$J_N(x) = \prod_{i=1}^k J_{p_i}(x)^{\alpha_i} .$$

As an example, let us compute  $J_{21}(17)$ . First we write  $21 = 3 * 7$ . Now note that  $17 \pmod{3} = 2$  and  $17 \pmod{7} = 3$ . But:

$$(1) \quad J_3(2) = -1 \text{ since } 2 \text{ is a non-square mod } 3$$

$$(2) \quad J_7(3) = -1 \text{ since } 3 \text{ is a non-square mod } 7$$

So  $J_{21}(17) = 1$ . Note however 17 is not a square, so unlike the Legendre symbol, the Jacobi symbol doesn't tell us anything about squareness.

**Fact 5.12** There is a polynomial time algorithm that given inputs  $N, x$  with  $x \in \mathbb{Z}_N^*$  outputs  $J_N(x)$ . ■

Note that the algorithm is *not* given the prime factorization of  $N$ . So it is not immediately obvious why it can easily compute the Jacobi symbol, given that our definition of the latter was in terms of the prime factorization of  $N$ . Nonetheless the fact is true.

From the above we know that if  $N$  is prime then  $J_N(x) \equiv a^{(N-1)/2} \pmod{N}$  for all  $x \in \mathbb{Z}_N^*$ . We now claim that if  $N$  is composite, this relation fails to hold for at least half the possible values of  $x$ . This will lead to the desired algorithm.

**Fact 5.13** Let  $N \geq 2$  be an integer and let  $S_N = \{a \in \mathbb{Z}_N^* : a^{(N-1)/2} \equiv J_N(a) \pmod{N}\}$ . Then:

$$(1) \quad \text{If } N \text{ is prime then } S_N = \mathbb{Z}_N^*$$

$$(2) \quad \text{If } N \text{ is not prime then } |S_N| \leq |\mathbb{Z}_N^*|/2. \quad \blacksquare$$

## 5.5 PRIMES is in coRP

We seek a randomized, polynomial-time algorithm (TM)  $M$  such that for any integer  $N$

- (1) If  $N$  is prime then  $\mathbf{AccPr}_M(N) = 1$
- (2) If  $N$  is composite then  $\mathbf{AccPr}_M(N) \leq 1/2$ .

A natural starting point is Fact 5.5, which suggests the following test. On input  $N$ , pick  $a$  at random from  $\mathbb{Z}_N^*$  and compute  $a^{N-1} \bmod N$ . If this value is 1 then accept else reject. The probability that this test accepts is  $|T_N|/|\mathbb{Z}_N^*|$  where  $T_N$  is the set of all  $a \in \mathbb{Z}_N^*$  having the property that  $a^{N-1} \bmod N = 1$ . When  $N$  is prime we know that  $T_N = \mathbb{Z}_N^*$  so the test always accepts, as desired. However, it turns out there are composite numbers  $N$ , called Carmichael numbers, having the property that  $T_N = \mathbb{Z}_N^*$ , meaning the test fails the second desired condition above.

In practice, Carmichael numbers seem to be rare, and this test actually performs very well, but we want a test with guaranteed correctness as given by the conditions listed above. The test, below, is based on Fact 5.13.

Algorithm  $M(N)$

1. Choose at random  $a \in \{1, \dots, N-1\}$
2. If  $\gcd(a, N) \neq 1$  then reject
3. Else
4.     Let  $\delta = a^{(N-1)/2} \bmod N$
5.     Let  $\epsilon = J_N(a)$
6.     If  $\delta \equiv \epsilon \pmod{N}$
7.         then accept else reject

We know that the gcd, Jacobi symbol and the modular exponentiation function are all polynomial time computable, and thus the test runs in polynomial time. We now verify the correctness by showing that the two conditions above are true.

First assume  $N$  is prime. The gcd at line 2 then must be one, so the algorithm does not reject, and moreover we know that  $a \in \mathbb{Z}_N^*$ . Fact 5.13 then says that the test at line 6 is true regardless of the value of  $a$ . So the algorithm accepts for all choices of  $a$ , meaning with probability one.

Now assume  $N$  is composite. If  $\gcd(a, N) \neq 1$  then the algorithm correctly rejects. If it did not reject, we know that  $a \in \mathbb{Z}_N^*$ , and since  $a$  was initially chosen at random in the range  $1, \dots, N-1$  it is a random element of  $\mathbb{Z}_N^*$ . Fact 5.13 then says that the test at line 6 is true with probability at most  $1/2$ .

Finally, let us note that the running time of the above algorithm is  $O(n^3)$ . This is so because modular exponentiation takes cubic time and the Jacobi symbol can also be computed in cubic time.

## References

- [1] M. BELLARE. Tail Inequalities. Under Miscellaneous notes, at <http://www-cse.ucsd.edu/users/mihir/courses.html>.