Chapter 2

Computational Hardness

2.1 Efficient Computation and Efficient Adversaries

We start by formalizing what it means to compute a function.

Definition 19.1 (Algorithm)**.** An *algorithm* is a deterministic Turing machine whose input and output are strings over some alphabet Σ. We usually have $\Sigma = \{0, 1\}$.

Definition 19.2 (Running-time of Algorithms)**.** An algorithm $\mathcal{A}$ runs in time $T(n)$ if for all $x \in B^*$, $\mathcal{A}(x)$ halts within $T(|x|)$ steps. $\mathcal{A}$ runs in *polynomial time* (or is an *efficient* algorithm) if there exists a constant c such that $\mathcal{A}$ runs in time $T(n) = n^c$.

Definition 19.3 (Deterministic Computation)**.** Algorithm $\mathcal{A}$ is said to *compute* a function $f : \{0, 1\}^* \to \{0, 1\}^*$ if $\mathcal{A}$, on input x, outputs $f(x)$, for all $x \in B^*$.

It is possible to argue with the choice of polynomial-time as a cutoff for "efficiency", and indeed if the polynomial involved is large, computation may not be efficient in practice. There are, however, strong arguments to use the polynomial-time definition of efficiency:

1. This definition is independent of the representation of the algorithm (whether it is given as a Turing machine, a C program, etc.) as converting from one representation to another only affects the running time by a polynomial factor.

2. This definition is also closed under composition, which is desirable as it simplifies reasoning.

3. "Usually", polynomial time algorithms do turn out to be efficient ('polynomial" almost always means "cubic time or better")
4. Most "natural" functions that are not known to be polynomial-time computable require *much* more time to compute, so the separation we propose appears to have solid natural motivation.

Remark: Note that our treatment of computation is an *asymptotic* one. In practice, actual running time needs to be considered carefully, as do other "hidden" factors such as the size of the description of $\mathcal{A}$. Thus, we will need to instantiate our formulae with numerical values that make sense in practice.

Some computationally "easy" and "hard" problem

Many commonly encountered functions are computable by efficient algorithms. However, there are also functions which are known or believed to be hard.

Halting: The famous Halting problem is an example of an *uncomputable* problem: Given a description of a Turing machine M, determine whether or not M halts when run on the empty input.

Time-hierarchy: There exist functions $f : \{0,1\}^* \rightarrow \{0,1\}$ that are computable, but are not computable in polynomial time (their existence is guaranteed by the Time Hierarchy Theorem in Complexity theory).

Satisfiability: The famous SAT problem is to determine whether a Boolean formula has a satisfying assignment. SAT is *conjectured* not to be polynomial-time computable—this is the very famous $P \neq NP$ conjecture.

Randomized Computation

A natural extension of deterministic computation is to allow an algorithm to have access to a source of random coins tosses. Allowing this extra freedom is certainly plausible (as it is easy to generate such random coins in practice), and it is believed to enable more efficient algorithms for computing certain tasks. Moreover, it will be necessary for the security of the schemes that we present later. For example, as we discussed in chapter one, Kerckhoff's principle states that all algorithms in a scheme should be public. Thus, if the private key generation algorithm Gen did not use random coins, then Eve would be able to compute the same key that Alice and Bob compute. Thus, to allow for this extra reasonable (and necessary) resource, we extend the above definitions of computation as follows.

Definition 20.1 (Randomized Algorithms - Informal)**.** A *randomized algorithm* is a Turing machine equipped with an extra random tape. Each bit of the random tape is uniformly and independently chosen.

Equivalently, a randomized algorithm is a Turing Machine that has access to a "magic" randomization box (or oracle) that output a truly random bit on demand.

To define efficiency we must clarify the concept of *running time* for a randomized algorithm. There is a subtlety that arises here, as the actual run time may depend on the bit-string obtained from the random tape. We take a conservative approach and define the running time as the upper bound over all possible random sequences.

Definition 21.1 (Running-time of Randomized Algorithms)**.** A randomized Turing machine $\mathcal{A}$ runs in time $T(n)$ if for all $x \in B^*$, $\mathcal{A}(x)$ halts within $T(|x|)$ steps (independent of the content of $\mathcal{A}$'s random tape). $\mathcal{A}$ runs in *polynomial time* (or is an *efficient* randomized algorithm) if there exists a constant c such that $\mathcal{A}$ runs in time $T(n) = n^c$.

Finally, we must also extend our definition of computation to randomized algorithm. In particular, once an algorithm has a random tape, its output becomes a distribution over some set. In the case of deterministic computation, the output is a singleton set, and this is what we require here as well.

Definition 21.2. Algorithm $\mathcal{A}$ is said to *compute* a function $f : \{0,1\}^* \to \{0,1\}^*$ if $\mathcal{A}$, on input x, outputs $f(x)$ with probability 1 for all $x \in B^*$. The probability is taken over the random tape of $\mathcal{A}$.

Thus, with randomized algorithms, we tolerate algorithms that on *rare* occasion make errors. Formally, this is a necessary relaxation because some of the algorithms that we use (e.g., primality testing) behave in such a way. In the rest of the book, however, we ignore this rare case and assume that a randomized algorithm always works correctly.

On a side note, it is worthwhile to note that a polynomial-time randomized algorithm $\mathcal{A}$ that computes a function with probability $\frac{1}{2} + \frac{1}{poly(n)}$ can be used to obtain another polynomial-time randomized machine $\mathcal{A}'$ that computes the function with probability $1 - 2^{-n}$. ($\mathcal{A}'$ simply takes multiple runs of $\mathcal{A}$ and finally outputs the most frequent output of $\mathcal{A}$. The Chernoff bound can then be used to analyze the probability with which such a "majority" rule works.)

Polynomial-time randomized algorithms will be the principal model of efficient computation considered in this course. We will refer to this class

of algorithms as *probabilistic polynomial-time Turing machine* (*p.p.t.*) or *efficient randomized algorithm* interchangeably.

Efficient Adversaries.

When modeling adversaries, we use a more relaxed notion of efficient computation. In particular, instead of requiring the adversary to be a machine with constant-sized description, we allow the size of the adversary's program to increase (polynomially) with the input length. As before, we still allow the adversary to use random coins and require that the adversary's running time is bounded by a polynomial. The primary motivation for using non-uniformity to model the adversary is to simplify definitions and proofs.

Definition 22.1 (Non-uniform p.p.t. machine)**.** A non-uniform p.p.t. machine A is a sequence of probabilistic machines $A = \{A_1, A_2, \ldots\}$ for which there exists a polynomial d such that the description size of $|A_i| < d(i)$ and the running time of A_i is also less than $d(i)$. We write $A(x)$ to denote the distribution obtained by running $A_{|x|}(x)$.

Alternatively, a non-uniform p.p.t. machine can also be defined as a *uniform* p.p.t. machine A that receives an advice string for each input length.

2.2 One-Way Functions

At a high level, there are two basic desiderata for any encryption scheme:

- it must be feasible to generate c given m and k, but
- it must be hard to recover m and k given c.

This suggests that we require functions that are easy to compute but hard to invert—*one-way functions*. Indeed, these turn out to be the most basic building block in cryptography.

There are several ways that the notion of one-wayness can be defined formally. We start with a definition that formalizes our intuition in the simplest way.

Definition 22.1. (Worst-case One-way Function). A function $f : \{0,1\}^* \to \{0,1\}$ is (worst-case) *one-way* if:

1. there exists a p.p.t. machine $\mathcal{C}$ that computes $f(x)$, and

2. there is no non-uniform p.p.t. machine $\mathcal{A}$ such that $\forall x \ \Pr[\mathcal{A}(f(x)) \in f^{-1}(f(x))] = 1$

We will see that assuming $SAT \notin BPP$, one-way functions according to the above definition must exist. In fact, these two assumptions are equivalent. Note, however, that this definition allows for certain pathological functions—e.g., those where inverting the function for *most* x values is easy, as long as every machine fails to invert $f(x)$ for infinitely many x's. It is an open question whether such functions can still be used for good encryption schemes. This observation motivates us to refine our requirements. We want functions where for a randomly chosen x, the probability that we are able to invert the function is very small. With this new definition in mind, we begin by formalizing the notion of *very small*.

Definition 23.1. A function $\varepsilon(n)$ is *negligible* if for every c, there exists some n_0 such that for all $n_0 < n$, $\epsilon(n) \leq \frac{1}{n^c}$. Intuitively, a negligible function is asymptotically smaller than the inverse of any fixed polynomial.

We say that a function $t(n)$ is *non-negligible* if there exists some constant c such that for infinitely many points $\{n_0, n_1, \ldots\}$, $t(n_i) > n_i^c$. This notion becomes important in proofs that work by contradiction.

We are now ready to present a more satisfactory definition of a one-way function.

Definition 23.2 (Strong one-way function)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$ is *strongly one-way* if it satisfies the following two conditions.

1. **Easy to compute.** There is a p.p.t. machine $\mathcal{C} : \{0,1\}^* \to \{0,1\}^*$ that computes $f(x)$ on all inputs $x \in \{0,1\}^*$.

2. **Hard to invert.** Any efficient attempt to invert f on random input will succeed with only negligible probability. Formally, for any non-uniform p.p.t. machines $\mathcal{A} : \{0,1\}^* \to \{0,1\}^*$, there exists a negligible function ϵ such that for any input length $n \in \mathbb{N}$,

$$\Pr\left[x \leftarrow \{0,1\}^n;\ y = f(x);\ \mathcal{A}(1^n, y) = x' : f(x') = y\right] \leq \epsilon(n).$$

Remark:

1. The algorithm $\mathcal{A}$ receives the additional input of 1^n; this is to allow $\mathcal{A}$ to take time polynomial in $|x|$, even if the function f should be substantially length-shrinking. In essence, we are ruling out some pathological cases where functions might be considered one-way because writing down the output of the inversion algorithm violates its time bound.

2. As before, we must keep in mind that the above definition is *asymptotic*. To define one-way functions with concrete security, we would instead use explicit parameters that can be instantiated as desired. In such a treatment we say that a function is (t, s, ϵ)-one-way, if no $\mathcal{A}$ of size s with running time $\leq t$ will succeed with probability better than ϵ.

2.3 Hardness Amplification

A first candidate for a one-way function is the function $f_{\mathsf{mult}} : \mathbb{N}^2 \to \mathbb{N}$ defined by $f_{\mathsf{mult}}(x, y) = xy$, with $|x| = |y|$. Is this a one-way function? Clearly, by the multiplication algorithm, f_{mult} is easy to compute. But f_{mult} is not always hard to invert! If at least one of x and y is even, then their product will be even as well. This happens with probability $\frac{3}{4}$ if the input (x, y) is picked uniformly at random from $\mathbb{N}^2$. So the following attack A will succeed with probability $\frac{3}{4}$:

$$A(z) = \begin{cases} (2, \frac{z}{2}) & \text{if } z \text{ even} \\ (0, 0) & \text{otherwise.} \end{cases}$$

Something is not quite right here, since f_{mult} is conjectured to be hard to invert on *some*, but not all, inputs. Our current definition of a one-way function is too restrictive to capture this notion, so we will define a weaker variant that relaxes the hardness condition on inverting the function. This weaker version only requires that all efficient attempts at inverting will fail with some non-negligible probability.

Definition 24.1 (Weak one-way function)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$ is *weakly one-way* if it satisfies the following two conditions.

1. **Easy to compute.** (Same as that for a strong one-way function.) There is a p.p.t. machine $\mathcal{C} : \{0,1\}^* \to \{0,1\}^*$ that computes $f(x)$ on all inputs $x \in \{0,1\}^*$.

2. **Hard to invert.** Any efficient algorithm will fail to invert f on random input with non-negligible probability. More formally, for any non-uniform p.p.t. machine $\mathcal{A} : \{0,1\}^* \to \{0,1\}^*$, there exists a polynomial function $q : \mathbb{N} \to \mathbb{N}$ such that for any input length $n \in \mathbb{N}$,

$$\mathsf{Pr}\left[x \leftarrow \{0,1\}^n;\ y = f(x);\ \mathcal{A}(1^n, y) = x' : f(x') = y\right] \leq 1 - \frac{1}{q(n)}$$

It is conjectured that f_{mult} is a weak one-way function.

Hardness Amplification

Fortunately, a weak version of a one-way function is sufficient for everything because it can be used to produce a strong one-way function. The main insight is that by running a weak one-way function f with enough inputs produces a list which contains at least one member on which it is hard to invert f.

Theorem 25.1. *Given a weak one-way function $f : \{0,1\}^* \to \{0,1\}^*$, there is a fixed $m \in \mathbb{N}$, polynomial in the input length $n \in \mathbb{N}$, such that the following function $f' : (\{0,1\}^n)^m \to (\{0,1\}^n)^m$ is strongly one-way:*

$$f'(x_1, x_2, \ldots, x_m) = (f(x_1), f(x_2), \ldots, f(x_m)).$$

We prove this theorem by contradiction. We assume that f' is not strongly one-way so that there is an algorithm $\mathcal{A}'$ that inverts it with non-negligible probability. From this, we construct an algorithm $\mathcal{A}$ that inverts f with high probability.

*Proof of Theorem 25.1

Proof. Since f is weakly one-way, let $q : \mathbb{N} \to \mathbb{N}$ be a polynomial such that for any non-uniform p.p.t. algorithm $\mathcal{A}$ and any input length $n \in \mathbb{N}$,

$$\mathsf{Pr}\left[x \leftarrow \{0,1\}^n;\ y = f(x);\ \mathcal{A}(1^n, y) = x' : f(x') = y\right] \leq 1 - \frac{1}{q(n)}.$$

Define $m = 2nq(n)$.
Assume that f' as defined in the theorem is not strongly one-way. Then let $\mathcal{A}'$ be a non-uniform p.p.t. algorithm and $p' : \mathbb{N} \to \mathbb{N}$ be a polynomial such that for infinitely many input lengths $n \in \mathbb{N}$, $\mathcal{A}'$ inverts f' with probability $p'(n)$. i.e.,

$$\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f(x_i) : f'(\mathcal{A}'(y_1, y_2, \ldots, y_m)) = (y_1, y_2, \ldots, y_m)\right] > \frac{1}{p'(m)}.$$

Since m is polynomial in n, then the function $p(n) = p'(m) = p'(2nq(n))$ is also a polynomial. Rewriting the above probability, we have

$$\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f(x_i) : f'(\mathcal{A}'(y_1, y_2, \ldots, y_m)) = (y_1, y_2, \ldots, y_m)\right] > \frac{1}{p(n)}. \tag{2.1}$$

Define the algorithm $\mathcal{A}_0 : \{0,1\}^n \to \{0,1\}^n_\perp$, which will attempt to use $\mathcal{A}'$ to invert f, as follows.

(1) Input $y \in \{0,1\}^n$.
(2) Pick a random $i \leftarrow [1, m]$.
(3) For all $j \neq i$, pick a random $x_j \leftarrow \{0,1\}^n$, and let $y_j = f(x_j)$.
(4) Let $y_i = y$.
(5) Let $(z_1, z_2, \ldots, z_m) = \mathcal{A}'(y_1, y_2, \ldots, y_m)$.
(6) If $f(z_i) = y$, then output z_i; otherwise, fail and output $\perp$.

To improve our chances of inverting f, we will run $\mathcal{A}_0$ multiple times. To capture this, define the algorithm $\mathcal{A} : \{0,1\}^n \to \{0,1\}^n_\perp$ to run $\mathcal{A}_0$ with its input $2nm^2p(n)$ times, outputting the first non-$\perp$ result it receives. If all runs of $\mathcal{A}_0$ result in $\perp$, then $\mathcal{A}$ outputs $\perp$ as well.

Given this, call an element $x \in \{0,1\}^n$ "good" if $\mathcal{A}_0$ will successfully invert $f(x)$ with non-negligible probability:

$$\mathsf{Pr}\left[\mathcal{A}_0(f(x)) \neq \perp\right] \geq \frac{1}{2m^2p(n)};$$

otherwise, call x "bad."

Note that the probability that $\mathcal{A}$ fails to invert $f(x)$ on a good x is small:

$$\mathsf{Pr}\left[\mathcal{A}(f(x)) \text{ fails} \mid x \text{ good}\right] \leq \left(1 - \frac{1}{2m^2p(n)}\right)^{2m^2np(n)} \approx e^{-n}.$$

We claim that there are a significant number of good elements—enough for $\mathcal{A}$ to invert f with sufficient probability to contradict the weakly one-way assumption on f. In particular, we claim there are at least $2^n\left(1 - \frac{1}{2q(n)}\right)$ good elements in $\{0,1\}^n$. If this holds, then

$$\begin{aligned}
&\mathsf{Pr}\left[\mathcal{A}(f(x)) \text{ fails}\right] \\
&= \mathsf{Pr}\left[\mathcal{A}(f(x)) \text{ fails} \mid x \text{ good}\right] \cdot \mathsf{Pr}\left[x \text{ good}\right] + \mathsf{Pr}\left[\mathcal{A}(f(x)) \text{ fails} \mid x \text{ bad}\right] \cdot \mathsf{Pr}\left[x \text{ bad}\right] \\
&\leq \mathsf{Pr}\left[\mathcal{A}(f(x)) \text{ fails} \mid x \text{ good}\right] + \mathsf{Pr}\left[x \text{ bad}\right] \\
&\leq \left(1 - \tfrac{1}{2m^2p(n)}\right)^{2m^2np(n)} + \tfrac{1}{2q(n)} \\
&\approx e^{-n} + \tfrac{1}{2q(n)} \\
&< \tfrac{1}{q(n)}.
\end{aligned}$$

This contradicts the assumption that f is $q(n)$-weak.

It remains to be shown that there are at least $2^n\left(1 - \frac{1}{2q(n)}\right)$ good elements in $\{0,1\}^n$. Assume that there are more than $2^n\left(\frac{1}{2q(n)}\right)$ bad elements. We

will contradict fact (2.1) that with probability $\frac{1}{p(n)}$, $\mathcal{A}'$ succeeds in inverting $f'(x)$ on a random input x. To do so, we establish an upper bound on the probability by splitting it into two quantities:

$$\begin{aligned}
&\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds}\right] \\
&= \mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and some } x_i \text{ is bad}\right] \\
&\quad + \mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and all } x_i \text{ are good}\right]
\end{aligned}$$

For each $j \in [1, n]$, we have

$$\begin{aligned}
&\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and } x_j \text{ is bad}\right] \\
&\leq \mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds} \mid x_j \text{ is bad}\right] \\
&\leq m \cdot \mathsf{Pr}\left[\mathcal{A}_0(f(x_j)) \text{ succeeds} \mid x_j \text{ is bad}\right] \\
&\leq \frac{m}{2m^2 p(n)} = \frac{1}{2mp(n)}.
\end{aligned}$$

So taking a union bound, we have

$$\begin{aligned}
&\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and some } x_i \text{ is bad}\right] \\
&\leq \textstyle\sum_j \mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and } x_j \text{ is bad}\right] \\
&\leq \frac{m}{2mp(n)} = \frac{1}{2p(n)}.
\end{aligned}$$

Also,

$$\begin{aligned}
&\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds and all } x_i \text{ are good}\right] \\
&\leq \mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n : \text{all } x_i \text{ are good}\right] \\
&< \left(1 - \frac{1}{2q(n)}\right)^m = \left(1 - \frac{1}{2q(n)}\right)^{2nq(n)} \approx e^{-n}.
\end{aligned}$$

Hence, $\mathsf{Pr}\left[x_i \leftarrow \{0,1\}^n; y_i = f'(x_i) : \mathcal{A}'(\vec{y}) \text{ succeeds}\right] < \frac{1}{2p(n)} + e^{-n} < \frac{1}{p(n)}$, thus contradicting (2.1). □

This theorem indicates that the existence of weak one-way functions is equivalent to that of strong one-way functions.

2.4 Collections of One-Way Functions

In the last two sections, we have come to suitable definitions for strong and weak one-way functions. These two definitions are concise and elegant, and can nonetheless be used to construct generic schemes and protocols. However, the definitions are more suited for research in complexity-theoretic aspects of cryptography.

For more practical cryptography, we introduce a slightly more flexible definition that combines the practicality of a weak OWF with the security of

a strong OWF. In particular, instead of requiring the function to be one-way on a randomly chosen string, we define a domain and a domain sampler for *hard-to-invert* instances. Because the inspiration behind this definition comes from "candidate one-way functions," we also introduce the concept of a *collection* of functions; one function per input size.

Definition 28.1. (Collection of OWFs). A *collection of one-way functions* is a family $\mathcal{F} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}_{i \in I}$ satisfying the following conditions:

1. It is easy to sample a function, i.e. there exists a PPT Gen such that $Gen(1^n)$ outputs some $i \in I$.
2. It is easy to sample a given domain, i.e. there exists a PPT that on input i returns a uniformly random element of $\mathcal{D}_i$
3. It is easy to evaluate, i.e. there exists a PPT that on input $i, x \in \mathcal{D}_i$ computes $f_i(x)$.
4. It is hard to invert, i.e. for any PPT $\mathcal{A}$ there exists a negligible function ϵ such that

$$\mathsf{Pr}\left[i \leftarrow Gen; x \leftarrow \mathcal{D}_i; y \leftarrow f_i(x); z = \mathcal{A}(1^n, i, y) : f(z) = y\right] \leq \epsilon(n)$$

Despite our various relaxations, the existence of a collection of one-way functions is equivalent to the existence of a strong one-way function.

Theorem 28.1. *There exists a collection of one-way functions if and only if there exists a single strong one-way function*

Proof idea: If we have a single one-way function f, then we can choose our index set ot be the singleton set $I = \{0\}$, choose $\mathcal{D}_0 = \mathbb{N}$, and $f_0 = f$.

The difficult direction is to construct a single one-way function given a collection $\mathcal{F}$. The trick is to define $g(r_1, r_2)$ to be $i, f_i(x)$ where i is generated using r_1 as the random bits and x is sampled from $\mathcal{D}_i$ using r_2 as the random bits. The fact that g is a strong one-way function is left as an excercise. □

2.5 Basic Computational Number Theory

Before we can study candidate collections of one-way functions, it serves us to review some basic algorithms and concepts in number theory and group theory.

Modular Arithmetic

We state the following basic facts about modular arithmetic:

Claim 28.1. *For $N > 0$ and $a, b \in \mathbb{Z}$,*

1. $(a \bmod N) + (b \bmod N) \bmod N = (a + b) \bmod N$
2. $(a \bmod N)(b \bmod N) \bmod N = ab \bmod N$

Euclid's algorithm

Euclid's algorithm appears in text around 300B.C.; it is therefore well-studied. Given two numbers a and b such that $a \geq b$, Euclid's algorithm computes the greatest common divisor of a and b, denoted $\gcd(a, b)$. It is not at all obvious how this value can be efficiently computed, without say, the factorization of both numbers. Euclid's insight was to notice that any divisor of a and b will also be a divisor of b and $a - b$. The latter is both easy to compute and a *smaller* problem than the original one. The algorithm has since been updated to use $a \bmod b$ in place of $a - b$ to improve efficiency. An elegant version of the algorithm which we present here also computes values x, y such that $ax + by = \gcd(a, b)$.

Algorithm 1: ExtendedEuclid(a, b)

Input: (a, b) s.t $a > b \geq 0$
Output: (x, y) s.t. $ax + by = \gcd(a, b)$

```
if a mod b = 0 then
    Return (0, 1)
else
    (x, y) ← ExtendedEuclid (b, a mod b)
    Return (y, x − y(⌊a/b⌋))
```

Note: by polynomial time we always mean polynomial in the size of the input, that is $poly(\log a + \log b)$

Proof. On input $a > b \geq 0$, we aim to prove that Algorithm 1 returns (x, y) such that $ax + by = \gcd(a, b) = d$ via induction. First, let us argue that the procedure terminates in polynomial time. The original analysis by Lamé is slightly better; for us the following suffices since each recursive call involves only a constant number of divisions and subtraction operations.

Claim 29.1. *If $a > b \geq 0$ and $a < 2^n$, then ExtendedEuclid(a, b) makes at most $2n$ recursive calls.*

Proof. By inspection, if $n \leq 2$, the procedure returns after at most 2 recursive calls. Assume for $a < 2^n$. Now consider an instance with $a < 2^{n+1}$. We identify two cases.

1. If $b < 2^n$, then the next recursive call on $(b, a \bmod b)$ meets the inductive hypothesis and makes at most $2n$ recursive calls. Thus, the total number of recursive calls is less than $2n + 1 < 2(n + 1)$.

2. If $b > 2^n$, we can only guarantee that first argument of the next recursive call on $(b, a \bmod b)$ is upper-bounded by 2^{n+1} since $a > b$. Thus, the problem is no "smaller" on face. However, we can show that the second argument will be small enough to satisfy the prior case:

$$\begin{aligned} a \bmod b &= a - \lfloor a/b \rfloor \cdot b \\ &< 2^{n+1} - b \\ &< 2^{n+1} - 2^n = 2^n \end{aligned}$$

   Thus, 2 recursive calls are required to reduce to the inductive case for a total of $2 + 2n = 2(n + 1)$ calls.

□

Now for correctness, suppose that b divided a evenly (i.e., $a \bmod b = 0$). Then we have $\gcd(a, b) = b$, and the algorithm returns $(0, 1)$ which is correct by inspection. By the inductive hypothesis, assume that the recursive call returns (x, y) such that

$$bx + (a \bmod b)y = \gcd(b, a \bmod b)$$

First, we claim that

Claim 30.1. $\gcd(a, b) = \gcd(b, a \bmod b)$

Proof. Divide a by b and write the result as $a = qb + r$. Rearrange to get $r = a - qb$.

Observe that if d is a divisor of a and b (i.e. $a = a'd$ and $b = b'd$ for $a', b' \in \mathbb{Z}$) then d is also a divisor of r since $r = (a'd) - q(b'd) = d(a' - qb')$. Since this holds for all divisors of a and b, it follows that $\gcd(a, b) = \gcd(b, r)$. □

Thus, we can write

$$bx + (a \bmod b)y = d$$

and by adding 0 to the right, and regrouping, we get

$$\begin{aligned} d &= bx - b(\lfloor a/b \rfloor)y + (a \bmod b)y + b(\lfloor a/b \rfloor)y \\ &= b(x - (\lfloor a/b \rfloor)y) + ay \end{aligned}$$

which shows that the return value $(y, x - (\lfloor a/b \rfloor)y)$ is correct. □

The assumption that the inputs are such that $a > b$ is without loss of generality since otherwise the first recursive call swaps the order of the inputs.

Exponentiation modulo N

Given a, x, N, we now demonstrate how to efficiently compute $a^x \bmod N$. Recall that by *efficient*, we require the computation to take polynomial time in the size of the representation of a, x, N. Since inputs are given in binary notation, this requires our procedure to run in time $poly(\log(a), \log(x), \log(N))$.

The key idea is to rewrite x in binary as $x = 2^\ell x_\ell + 2^{\ell-1}x_{\ell-1} + \cdots + 2x_1 + x_0$ where $x_i \in \{0, 1\}$ so that

$$a^x \bmod N = a^{2^\ell x_\ell + 2^{\ell-1}x_{\ell-1} + \cdots + 2^1 x_1 + x_0} \bmod N$$

We show this can be further simplified as

$$a^x \bmod n = \prod_{i=0}^{\ell} x_i a^{2^i} \bmod N$$

Algorithm 2: ModularExponentiation(a, x, N)

Input: $a, x \in [1, N]$

```
r ← 1
while x > 0 do
    if x is odd then
        r ← r · a mod N
    x ← ⌊x/2⌋
    a ← a^2 mod N
Return r
```

Theorem 31.1. *On input* (a, x, N) *where* $a, x \in [1, N]$*, Algorithm 2 computes* $a^x \bmod N$ *in time* $O(\log^3(N))$.

Proof. By the basic facts of modulo arithmetic from Claim [**?**], we can rewrite $a^x \bmod N$ as $\prod_i x_i a^{2^i} \bmod N$.

Since multiplying (and squaring) modulo N take time $\log^2(N)$, each iteration of the loop requires $O(\log^2(N))$ time. Because $x < N$, and each iteration divides x by two, the loop runs at most $\log N$ times which establishes a running time of $O(\log^3(N))$. □

Later, after we have introduced Euler's theorem, we present a similar algorithm for modular exponentiation which removes the restriction that $x < N$. In order to discuss this, we must introduce the notion of Groups.

Groups

Definition 32.1. A group G is a set of elements G with a binary operator $\oplus : G \times G \to G$ that satisfies the following axioms:

1. Closure: For all $a, b \in G$, $a \oplus b \in G$,
2. Identity: There is an element i in G such that for all $a \in G$, $i \oplus a = a \oplus i = a$. This element i is called the *identity* element.
3. Associativity: For all a, b and c in G, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
4. Inverse: For all $a \in G$, there is an element $b \in G$ such that $a \oplus b = b \oplus a = i$ where i is the *identity*.

Example: The Additive Group Mod N

There are many natural groups which we have already implicitly worked with. The additive group modulo N is denoted $(\mathbb{Z}_N, +)$ where $Z_N = \{0, 1, ..N-1\}$ and $+$ is addition modulo N. It is straightforward to verify the four properties for this set and operation.

Example: The Multiplicative Group Mod N

The multiplicative group modulo $N > 0$ is denoted $(\mathbb{Z}_N^*, \times)$, where $Z_N^* = \{x \in [1, N-1] \mid \gcd(x, N) = 1\}$ and $\times$ is multiplication modulo N.

Theorem 32.1. $(\mathbb{Z}_N^*, \times)$ *is a group*

Proof. It is easy to see that 1 is the identity in this group and that $(a * b) * c = a * (b * c)$ for $a, b, c \in \mathbb{Z}_N^*$. However, we must verify that the group is closed and that each element has an inverse.

Closure For the sake of contradiction, suppose there are two elements $a, b \in \mathbb{Z}_N^*$ such that $ab \notin \mathbb{Z}_N^*$. This implies that $\gcd(a, N) = 1$, $\gcd(b, N) = 1$, but that $\gcd(ab, N) = d > 1$. The latter condition implies that d has a non-trivial factor that divides both ab and N. Thus, d must also divide either a or b (verify as an exercise), which contradicts the assumption that $\gcd(a, N) = 1$ or $\gcd(b, N) = 1$.

Inverse Consider an element $a \in \mathbb{Z}_N^*$. Since $\gcd(a, N) = 1$, we can use Euclid's algorithm on (a, N) to compute values (x, y) such that $ax + Ny = 1$. Notice, this directly produces a value x such that $ax = 1 \bmod N$. Thus, every element $a \in \mathbb{Z}_N^*$ has an inverse which can be efficiently computed. □

Remark: The groups $(\mathbb{Z}_N, +)$ and $(\mathbb{Z}_N^*, \times)$ are also *abelian* or commutative groups in which $a \oplus b = b \oplus a$.

Definition 33.1. A *prime* is a positive integer that is divisible by only 1 and itself.

The number of unique elements in $\mathbb{Z}_N^*$ (often referred to as the *order* of the group) is denoted by the the Euler Totient function $\Phi(N)$.

$$\begin{array}{ll} \Phi(p) = p - 1 & \text{if } p \text{ is prime} \\ \Phi(N) = (p-1)(q-1) & \text{if } N = pq \text{ and } p, q \text{ are primes} \end{array}$$

The first case follows because all elements less than p will be relatively prime to p. The second case requires some simple counting (show this).

The structure of these multiplicative groups results in some very special properties which we can exploit throughout this course. One of the first properties is the following identity first proven by Euler in 1736.

Theorem 33.1 (Euler). $\forall a \in \mathbb{Z}_N^*, a^{\Phi(N)} \equiv 1 \bmod N$

Proof. Consider the set $A = \{ax \mid x \in \mathbb{Z}_N^*\}$. Since $\mathbb{Z}_N^*$ is a group, it follows that $A \subseteq \mathbb{Z}_N^*$ since every element $ax \in \mathbb{Z}_N^*$. Now suppose that $|A| < |\mathbb{Z}_N^*|$. By the pidgeonhole principle, this implies that there exist two group element $i, j \in \mathbb{Z}_N^*$ such that $i \neq j$ but $ai = aj$. Since $a \in \mathbb{Z}_N^*$, there exists an inverse a^{-1}

such that $aa^{-1} = 1$. Multiplying on both sides we have $a^{-1}ai = a^{-1}aj \implies i = j$ which is a contradiction. Thus, $|A| = |\mathbb{Z}_N^*|$ which implies that $A = \mathbb{Z}_N^*$.

Because the group is abelian (i.e., commutative), we can take products and substitute the defintion of A to get

$$\prod_{x \in \mathbb{Z}_N^*} x = \prod_{y \in A} y = \prod_{x \in \mathbb{Z}^* n} ax$$

The product further simplifies as

$$\prod_{x \in \mathbb{Z}_N^*} x = a^{\Phi(n)} \prod_{x \in \mathbb{Z}_N^*} x$$

Finally, since the closure property guarantees that $\prod_{x \in \mathbb{Z}_N^*} x \in \mathbb{Z}_N^*$ and since the inverse property guarantees that this element has an inverse, we can multiply the inverse on both sides to obtain

$$1 = a^{\Phi(n)}$$

□

Corollary 34.1 (Fermat's little theorem)**.** $\forall a \in \mathbb{Z}_p^*, a^{p-1} \equiv 1 \bmod p$

N)

Corollary 34.2. $a^x \bmod N = a^{x \bmod \Phi(N)} \bmod N$. *Thus, given* $\Phi(N)$*, the operation* $a^x \bmod N$ *can be computed efficiently in* $\mathbb{Z}_N$ *for any* x*.*

Thus, we can efficiently compute $a^{2^{2^x}} \bmod N$.

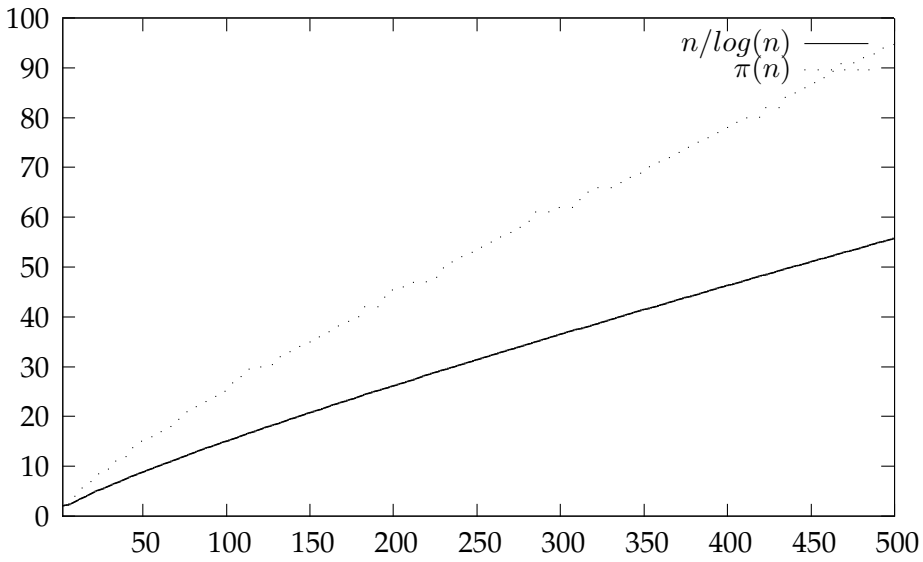
There are many primes

The problem of characterizing the set of prime numbers has been considered since antiquity. Early on, it was noted that there are an infinite number of primes. However, merely having an infinite number of them is not reassuring, since perhaps they are distributed in such a haphazard way as to make finding them extremely difficult. An empirical way to approach the problem is to define the function

$$\pi(x) = \text{number of primes} \leq x$$

and graph it for reasonable values of x:

By empirically fitting this curve, one might guess that $\pi(x) \approx x/\log x$. In fact, at age 15, Gauss made exactly this conjecture. Since then, many people



have answered the question with increasing precision; notable are Chebyshev's theorem (upon which our argument below is based), and the famous *Prime Number Theorem* which establishes that $\pi(N)$ approaches $\frac{N}{\log N}$ as N grows to infinity. Here, we will prove a much simpler theorem which only lower-bounds $\pi(x)$:

Theorem 35.1 (Chebyshev)**.** *For $x > 1$, $\pi(x) > \frac{x}{\log x}$*

Proof. Consider the value

$$X = \frac{2x!}{(x!)^2} = \left(\frac{x+x}{x}\right)\left(\frac{x+(x-1)}{(x-1)}\right)\cdots\left(\frac{x+2}{2}\right)\left(\frac{x+1}{1}\right)$$

Observe that $X > 2^x$ (since each term is greater than 2) and that the largest prime dividing X is at most $2x$ (since the largest numerator in the product is $2x$). By these facts and unique factorization, we can write

$$X = \prod_{p<2x} p^{\nu_p(X)} > 2^x$$

where the product is over primes p less than $2x$ and $\nu_p(X)$ denotes the integral power of p in the factorization of X. Taking logs on both sides, we have

$$\sum_{p<2x} \nu_p(X) \log p > x$$

We now employ the following claim proven below.

Claim 35.1. $\nu_p(X) < \frac{\log 2x}{\log p}$

Substituting this claim, we have

$$\sum_{p<2x} \left(\frac{\log 2x}{\log p} \right) \log p = \log 2x \left(\sum_{p<2x} 1 \right) > x$$

Notice that the sum on the left hand side is precisely $\pi(2x)$; thus

$$\pi(2x) > \frac{x}{\log 2x} = \frac{2x}{2 \log 2x}$$

which establishes the theorem for even values. For odd values, notice that

$$\pi(2x) = \pi(2x-1) > \frac{2x}{2 \log 2x} > \frac{(2x-1)}{2 \log(2x-1)}$$

since $x/\log x$ is an increasing function for $x \geq 3$.

Proof Of Claim 35.1. Notice that

$$\nu_p(X) = \sum_{i>1} \left(\lfloor 2x/p^i \rfloor - 2\lfloor x/p^i \rfloor \right) < \log 2x / \log p$$

The first equality follows because the product $2x! = (2x)(2x-1)\ldots(1)$ includes a multiple of p^i at most $\lfloor 2x/p^i \rfloor$ times in the numerator of X; similarly the product $x! \cdot m!$ in the denominator of X *removes* it exactly $2\lfloor x/p^i \rfloor$ times. The second inequality follows because each term in the summation is at most 1 and after $p^i > 2x$, all of the terms will be zero. □

□

Miller-Rabin Primality Testing

An important task in generating the parameters of a many cryptographic scheme will be the identification of a suitably large prime number. Eratosthenes (276-174BC), the librarian of Alexandria, is credited with devising a very simple sieving method to enumerate all primes. However, this method is not practical for choosing a large (i.e., 1000 digit) prime.

Instead, recall that Fermat's Little Theorem establishes that $a^{p-1} = 1 \bmod p$ for any $a \in \mathbb{Z}_p$ whenever p is prime. It turns out that when p is not prime,

then a^{p-1} is usually *not* equal to 1. The first fact and second phenomena form the basic idea behind the Miller-Rabin primality test: to test p, pick a random $a \in \mathbb{Z}_p$, and check whether $a^{p-1} = 1 \bmod p$. (Notice that efficient modular exponentiation is critical for this test.) Unfortunately, the second phenomena is *on rare occasion* false. Despite there rarity (starting with $561, 1105, 1729, \ldots$,, there are only 255 such cases less than 10^8!), there are infact an infinite number of counter examples collectively known as the *Carmichael* numbers. Thus, for correctness, we must test for these rare cases. This second check amounts to verifying that none of the *intermediate* powers of a encountered during the modular exponentiation computation of a^{n-1} are non-trivial square-roots of 1. This suggest the following approach.
For positive n, write $n = u2^j$ where u is odd. Define the set

$$L_N = \{\alpha \in \mathbb{Z}_N \mid \alpha^{N-1} = 1 \text{ and if } \alpha^{u2^{j+1}} = 1 \text{ then } \alpha^{u2^j} = 1\}$$

The first condition is based on Fermat's Little theorem, and the second one eliminates errors due to Carmichael numbers.

Theorem 37.1. *If N is an odd prime, then $|L_N| = N - 1$. If $N > 2$ is composite, then $|L_N| < (N-1)/2$.*

We will not prove this theorem here. See **[?]** for a formal proof. The proof idea is as follows. If n is prime, then by Fermat's Little Theorem, the first condition will always hold, and since 1 only has two square roots modulo p (namely, $1, -1$), the second condition holds as well. If N is composite, then either there will be some α for which α^{N-1} is not equal to 1 or the process of computing α^{N-1} reveals a square root of 1 which is different from 1 or -1—recall that when N is composite, there are at least 4 square roots of 1. More formally, the proof works by first arguing that all of the $\alpha \notin L_N$ form a *proper* subgroup of $\mathbb{Z}_N^*$. Since the order of a subgroup must divide the order of the group, the size of a proper subgroup must therefore be less than $(N-1)/2$.

This suggests the following algorithm for testing primality:

Algorithm 3: Miller-Rabin Primality Test

```
Handle base case N = 2
for t times do
    Pick a random α ∈ Z_N
    if α ∉ L_N then Output "composite"
Output "prime"
```

Observe that testing whether $\alpha \in L_N$ can be done by using a repeated-squaring algorithm to compute modular exponentiation, and adding internal checks to make sure that no non-trivial roots of unity are discovered.

Theorem 38.1. *If N is composite, then the Miller-Rabin test outputs "composite" with probability $1 - 2^{-t}$. If N is prime, then the test outputs "prime."*

Selecting a Random Prime

Our algorithm for finding a random n-bit prime will be simple: we will repeatedly sample a n-bit number and then check whether it is prime.

Algorithm 4: SamplePrime(n)

```
repeat
    x ←r {0,1}^n
    if Miller-Rabin(x) = "prime" then Return x
until done
```

There are two mathematical facts which make this simple scheme work:

1. There are many primes
2. It is easy to determine whether a number is prime

By Theorem 35.1, the probability that a uniformly-sampled n-bit integer is prime is greater than $(N/\log N)/N = \frac{c}{n}$. Thus, the expected number of guesses in the guess-and-check method is polynomial in n. Since the running time of the Miller-Rabin algorithm is also polynomial in n, the expected running time to sample a random prime using the guess-and-check approach is polynomial in n.

2.6 Factoring-based Collection

The Factoring Assumption

Let us denote the (finite) set of primes that are smaller than 2^n as

$$\Pi_n = \{q \mid q < 2^n \text{ and } q \text{ is prime}\}$$

Consider the following assumption, which we shall assume for the remainder of this book:

Conjecture 38.1 (Factoring Assumption)**.** *For every non-uniform p.p.t. algorithm $\mathcal{A}$, there exists a negligible function ϵ such that*

$$\mathsf{Pr}\left[p \xleftarrow{r} \Pi_n; q \leftarrow \Pi_n; N = pq : \mathcal{A}(N) \in \{p, q\}\right] < \epsilon(n)$$

The factoring assumption is a very important, well-studied conjecture that is widely believed to be true. The best provable algorithm for factorization runs in time $2^{O((n \log n)^{1/2})}$, and the best heuristic algorithm runs in time $2^{O(n^{1/3} \log^{2/3} n)}$. Factoring is hard in a concrete way as well: at the time of this writing, researchers have been able to factor a 663 bit numbers using 80 machines and several months.

Under this assumption, we can prove the following result, which establishes our first realistic collection of one-way functions:

Theorem 39.1. *Let $\mathcal{F} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}_{i \in I}$ where*

$$\begin{aligned} I &= \mathbb{N} \\ \mathcal{D}_i &= \{(p, q) \mid p, q \text{ prime, and } |p| = |q| = \frac{i}{2}\} \\ f_i(p, q) &= p \cdot q \end{aligned}$$

Assuming the Factoring Assumption holds, then $\mathcal{F}$ is a collection of OWF.

Proof. We can clearly sample a random element of $\mathbb{N}$. It is easy to evaluate f_i since multiplication is efficiently computable, and the factoring assumption says that inverting f_i is hard. Thus, all that remains is to present a method to efficiently sample two random prime numbers. This follows from the section below. Thus all four conditions in the definition of a one-way collection are satisfied. □

2.7 Discrete Logarithm-based Collection

Another often used collection is based on the discrete logarithm problem in the group $\mathbb{Z}_p^*$.

Discrete logarithm modulo p

An instance of the discrete logarithm problem consists of two elements $g, y \in \mathbb{Z}_p^*$. The task is to find an x such that $g^x = y \bmod p$. In some special cases (e.g., $g = 1$), it is easy to either solve the problem or declare that no solution exists. However, when g is a *generator* or Z_p^*, then the problem is believed to be hard.

Definition 39.1 (Generator of a Group)**.** A element g of a multiplicative group G is a generator if the set $\{g, g^2, g^3, \ldots\} = G$. We denote the set of all generators of a group G by Gen_G.

Conjecture 40.1 (Discrete Log Assumption)**.** *For every non-uniform p.p.t. algorithm $\mathcal{A}$, there exists a negligible function ϵ such that*

$$\mathsf{Pr}\left[p \xleftarrow{r} \Pi_n; g \xleftarrow{r} Gen_p; x \xleftarrow{r} \mathbb{Z}_p; y \leftarrow g^x \bmod p : \mathcal{A}(y) = x\right] < \epsilon(n)$$

Theorem 40.1. *Let* $\boldsymbol{DL} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}_{i \in I}$ *where*

$$\begin{aligned} I &= \{(p, g) \mid p \in \Pi_k, g \in Gen_p\} \\ \mathcal{D}_i &= \{x \mid x \in \mathbb{Z}_p\} \\ \mathcal{R}_i &= \mathbb{Z}_p^* \\ f_{p,g}(x) &= g^x \bmod p \end{aligned}$$

Assuming the Discrete Log Assumption holds, the family of functions $\boldsymbol{DL}$ *is a collection of one-way functions.*

Proof. It is easy to sample the domain D_i and to evaluate the function $f_{p,g}(x)$. The discrete log assumption implies that $f_{p,g}$ is hard to invert. Thus, all that remains is to prove that I can be sampled efficiently. Unfortunately, given only a prime p, it is not known how to efficiently choose a generator $g \in Gen_p$. However, it is possible to sample both a prime *and* a generator g at the same time. One approach proposed by Bach and later adapted by Kalai is to sample a k-bit integer x in factored form (i.e., sample the integer and its factorization at the same time) such that $p = x + 1$ is prime. Given such a pair $p, (q_1, \ldots, q_k)$, one can use a central result from group theory to test whether an element is a generator.

Another approach is to pick primes of the form $p = 2q + 1$. These are known as Sophie Germain primes or "safe primes." The standard method for sampling such a prime is simple: first pick a prime q as usual, and then check whether $2q + 1$ is also prime. Unfortunately, even though this procedure always terminates in practice, its basic theoretical properties are unknown. That is to say, it is unknown even (a) whether there are an infinite number of Sophie Germain primes, (b) and even so, whether such a procedure continues to quickly succeed as the size of q increases. Despite these unknowns, once such a prime is chosen, testing whether an element $g \in \mathbb{Z}_p^*$ is a generator consists of checking $g \neq \pm 1 \bmod p$ and $g^q \neq 1 \bmod p$. □

As we will see later, the collection **DL** is also a special collection of one-way functions in which each function is a permutation.

2.8 RSA collection

Another popular assumption is the RSA-assumption.

Conjecture 41.1 (RSA Assumption)**.** *Given (N, e, y) such that $N = pq$ where $p, q \in \Pi_n$, $\gcd(e, \Phi(N)) = 1$ and $y \in \mathbb{Z}_N^*$, the probability that any non-uniform p.p.t. algorithm A is able to produce x such that $x^e = y \bmod N$ is a negligible function $\epsilon(n)$.*

$$\Pr[p, q \xleftarrow{r} \Pi_n;\ n \leftarrow pq;\ e \xleftarrow{r} \mathbb{Z}_{\Phi(N)}^*;\ y \leftarrow \mathbb{Z}_N^* : A(N, e, y) = x \text{ s.t. } x^e = y \bmod N] < \epsilon(n)$$

Theorem 41.1 (RSA Collection)**.** *Let **RSA** $= \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}_{i \in I}$ where*

$$\begin{aligned} I &= \{(N, e) \mid N = pq \text{ s.t. } p, q \in \Pi_n \text{ and } e \in \mathbb{Z}_{\Phi(N)}^*\} \\ \mathcal{D}_i &= \{x \mid x \in \mathbb{Z}_N^*\} \\ \mathcal{R}_i &= \mathbb{Z}_N^* \\ f_{N,e}(x) &= x^e \bmod N \end{aligned}$$

*Assuming the RSA Assumption holds, the family of functions **RSA** is a collection of one-way functions.*

Proof. The set I is easy to sample: generate two primes p, q, multiply then to generate N, and use the fact that $\Phi(N) = (p-1)(q-1)$ to sample a random element from $\mathbb{Z}_{\Phi(N)}^*$. Likewise, the set D_i is also easy to sample and the function $f_{N,e}$ requires only one modular exponentiation to evaluate. It only remains to show that $f_{N,e}$ is difficult to invert. Notice, however, that this does not *directly* follow from the our hardness assumption (as it did in previous examples). The RSA assumption states that it is difficult to compute the eth root of a *random* group element y. On the other hand, our collection first picks the root and then computes $y \leftarrow x^e \bmod N$. One could imagine that picking an element that is *known* to have an eth root makes it easier to find such a root. We will show that this is not the case—and we will do so by showing that the function$f_{N,e}(x) = x^e \bmod N$ is in fact a permutation of the elements of $\mathbb{Z}_N^*$. This would then imply that the distributions $\{x, e \xleftarrow{r} \mathbb{Z}_N^* :\ (e, x^e \bmod N)\}$ and $\{y, e \xleftarrow{r} \mathbb{Z}_N^* :\ (e, y)\}$ are identical, and so an algorithm that inverts $f_{N,e}$ would also succeed at breaking the RSA-assumption. □

Theorem 41.2. *The function* $f_{N,e}(x) = x^e \bmod N$ *is a permutation of* $\mathbb{Z}_N^*$ *when* $e \in \mathbb{Z}_{\Phi(N)}^*$.

Proof. Since e is an element of the group $Z_{\Phi(N)}^*$, let d be its inverse (recall that every element in a group has an inverse), i.e. $ed = 1 \bmod \Phi(N)$. Consider the inverse map $g_{N,e}(x) = x^d \bmod N$. Now for any $x \in \mathbb{Z}_N^*$,

$$\begin{aligned} g_{N,e}(f_{N,e}(x)) &= g_{N,e}(x^e \bmod N) = (x^e \bmod N)^d \\ &= x^{ed} \bmod N \\ &= x^{c\Phi(N)+1} \bmod N \end{aligned}$$

for some constant c. Since Euler's theorem establishes that $x^{\Phi(N)} = 1 \bmod N$, then the above can be simplfied as

$$x^{c\Phi(N)} \cdot x \bmod N = x \bmod N$$

Hence, RSA is a permutation. □

This phenomena suggests that we formalize a new, stronger class of one-way functions.

2.9 One-way Permutations

Definition 42.1. (One-way permutation). A collection $\mathcal{F} = \{f_i : \mathcal{D}_i \to R_i\}_{i \in I}$ is a collection of *one-way permutations* if $\mathcal{F}$ is a collection of *one-way functions* and for all $i \in I$, we have that f_i is a permutation.

A natural question is whether this extra property comes at a price—that is, how does the RSA-assumption that we must make compare to a natural assumption such as factoring. Here, we can immediately show that RSA is at least as strong an assumption as Factoring.

Theorem 42.1. *The RSA-assumption implies the Factoring assumption.*

Proof. We prove by contrapositive: if factoring is possible in polynomial time, then we can break the RSA assumption in polynomial time. Formally, assume there an algorithm A and polynomial function $p(n)$ so that A can factor $N = pq$ with probability $1/p(n)$, where p and q are random n-bits primes. Then there exists an algorithm A', which can invert $f_{N,e}$ with probability $1/p(n)$, where $N = pq$, $p,\ q \leftarrow \{0,1\}^n$ primes, and $e \leftarrow \mathbb{Z}_{\Phi(N)}^*$.

The algorithm feeds the factoring algorithm A with exactly the same distribution of inputs as with the factoring assumption. Hence in the first step

Algorithm 5: Adversary $A'(N, e, y)$

Run $(p, q) \leftarrow A(N)$ to recover prime factors of N
if $N \neq pq$ **then** abort
Compute $\Phi(N) \leftarrow (p-1)(q-1)$
Compute the inverse d of e in $\mathbb{Z}^*_{\Phi(N)}$ using Euclid
Output $y^d \bmod N$

A will return the correct prime factors with probability $1/p(n)$. Provided that the factors are correct, then we can compute the inverse of y in the same way as we construct the inverse map of $f_{N,e}$. And this always succeeds with probability 1. Thus overall, A' succeeds in breaking the RSA-assumption with probability $1/p(n)$. Moreover, the running time of A' is essentially the running time of A plus $O(\log^3(n))$. Thus, if A succeeds in factoring in polynomial time, then A' succeeds in breaking the RSA-assumption in roughly the same time. □

Unfortunately, it is not known whether the converse it true—that is, whether the factoring assumption also implies the RSA-assumption.

2.10 Trapdoor Permutations

The proof that RSA is a permutation actually suggests another special property of that collection: if the factorization of N is unknown, then inverting $f_{N,e}$ is considered infeasiable; however if the factorization of N *is known*, then it is no longer hard to invert. In this sense, the factorization of N is a *trapdoor* which enables $f_{N,e}$ to be inverted.

This spawns the idea of trapdoor permutations, first conceived by Diffie and Hellman.

Definition 43.1. (Trapdoor Permutations). A *collection of trapdoor permutations* if a family $\mathcal{F} = \{f_i : \mathcal{D}_i \to \mathcal{R}_i\}_{i \in \mathcal{I}}$ satisfying the following properties:

1. $\forall i \in \mathcal{I}$, f_i is a permutation,
2. *It is easy to sample function:* $\exists$ p.p.t. Gen s.t. $(i, t) \leftarrow Gen(1^n),\ i \in \mathcal{I}$ (t is trapdoor info),
3. *It is easy to sample the domain:* $\exists$ p.p.t. machine that given input $i \in \mathcal{I}$, samples uniformly in $\mathcal{D}_i$.

4. f_i *is easy to evaluate:* $\exists$ p.p.t. machine that given input $i \in \mathcal{I}, x \in \mathcal{D}_i$, computes $f_i(x)$.
5. f_i *is hard to invert:* $\forall$ p.p.t. $\mathcal{A}$, $\exists$ negligible ϵ s.t.

$$\Pr[(i,t) \leftarrow Gen(1^n); x \in \mathcal{D}_i; y = f(x); z = A(1^n, i, y) : f(z) = y] \leq \epsilon(k)$$

6. f_i *is easy to invert with trapdoor information:* $\exists$ p.p.t. machine that given input (i,t) from Gen and $y \in \mathcal{R}_i$, computes $f^{-1}(y)$.

Now by slightly modifying the definition of the family **RSA**, we can easily show that it is a collection of trapdoor permutations.

Theorem 44.1. *Let **RSA** be defined as per Theorem 41.1 with the exception that*

$$[(N,e),(d)] \leftarrow Gen(1^n)$$
$$f^{-1}_{N,d}(y) = y^d \bmod N$$

where $N = pq$, $e \in \mathbb{Z}^*_{\Phi(N)}$ and $ed = 1 \bmod \Phi(N)$. Assuming the RSA-assumption, the collection RSA is a collection of trapdoor permutations.

The proof is an exercise.

2.11 Levin's One Way Function

As we have mentioned in previous sections, it is not known whether one-way functions exist. Although we have presented specific assumptions which have lead to specific constructions, a much weaker assumption is to assume only that *some* one-way function exists (without knowing exactly which one). The following theorem gives a single constructible function that is one-way if there this assumption is true:

Theorem 44.2. *If there exists a one-way function, then the following polynomial-time computable function f_{Levin} is also a one-way function.*

Proof. We will construct function f_{Levin} and show that it is weakly one-way. We can then apply the hardness amplification construction from §2.3 to get a strong one-way function.

The idea behind the construction is that f_{Levin} should combine the computations in all efficient functions in such a way that inverting f_{Levin} allows us to invert all other functions. The naïve approach is to interpret the input y to f_{Levin} as a machine-input pair $\langle M, x\rangle$, and then let the output of f_{Levin} be $M(x)$. The problem with this approach is that f will not be computable

Algorithm 6: Function $f_{\text{Levin}}(y)$: A Universal OWF

1 Interpret y as $\langle \ell, M, x \rangle$ where $\ell \in \{0,1\}^{\log\log|y|}$ and $|M| = \ell$
2 Run M on input x for $|y|^3$ steps
3 **if** M *terminates* **then** Output $M(x)$ **else** Output $\perp$

in polynomial time, since M may or may not even terminate on input x. In words, this function interprets the first $\log\log|y|$ bits of the input y as a length parameter between $[0, \log|y|]$. The next ℓ bits are interpreted as a machine M, and the remaining bits are considered input x. We claim that this function is weakly one-way. Clearly f_{Levin} is computable in $O(n^3)$ time, and thus it satisfies the "easy" criterion for being one-way. To show that it satisfies the "hard" criterion, we must assume that there exists some function g that is strongly one-way.

Without loss of generality, we can assume that g is computable in $O(n^2)$ time. This is because if g runs in $O(n^c)$ for some $c > 2$, then we can let $g'(\langle a, b \rangle) = \langle a, g(b) \rangle$, where $|a| \approx n^c$ and $|b| = n$. Then if we let $m = |\langle a, b \rangle| = O(n^c)$, the function g' is computable in time

$$\underbrace{|a|}_{\text{copying a}} + \underbrace{|b|^c}_{\text{computing } g} + \underbrace{O(m^2)}_{\text{parsing}} < 2m + O(m^2) = O(m^2)$$

Moreover, g' is still one-way, since we can easily reduce inverting g to inverting g'.

Now, if f_{Levin} is not weakly one-way, then there exists a machine $\mathcal{A}$ such that for every polynomial q and for infinitely many input lengths n,

$$\Pr\left[y \leftarrow \{0,1\}^n; \mathcal{A}(f(y)) \in f^{-1}(f(y))\right] > 1 - 1/q(n)$$

In particular, this holds for $q(n) = n^3$. Denote the event that $\mathcal{A}$ inverts as Invert.

Let M_g be the smallest machine which computes function g. Since M_g is a uniform algorithm it has some constant description size $|M_g|$. Denote $\log|M_g| = \ell_g$. Thus, on a random input $y = \langle \ell, M_g, x \rangle$ of size $|y| = n$, the probability that $\ell = \ell_g$ is $2^{-\log\log n} = 1/\log n$ and probability that machine $M = M_g$ is $2^{-\log n} = 1/n$. In other words

$$\Pr\left[y \xleftarrow{r} \{0,1\}^n : y = \langle \ell_g, M_g, x \rangle\right] > \frac{1}{n \log n}$$

Denote this event as event PickG.

We now lower bound the probability that $\mathcal{A}$ inverts a hard instance (i.e., one in which f has chosen M_g) as follows:

$$\begin{aligned}
\Pr[\textsf{Invert and PickG}] &= \Pr[\textsf{Invert}] - \Pr[\textsf{Invert and !PickG}] \\
&> \Pr[\textsf{Invert}] - \Pr[\textsf{!PickG}] \\
&> (1 - 1/n^3) - (1 - 1/n \log n) \\
&> 1/n \log n - 1/n^3
\end{aligned}$$

Applying Bayes rules to the previous inequality implies that $\Pr[\textsf{Invert} \mid \textsf{PickG}] > 1 - (n \log n)/n^3$ which contradicts the assumption that g is strongly one-way; therefore f_{Levin} must be weakly one-way. □

This theorem gives us a function that we can safely assume is one-way (because that assumption is equivalent to the assumption that one-way functions exist). However, it is extremely impractical to compute. First, it is difficult to compute because it involves repeatedly interpreting random turing machines. Second, it will require very large key lengths before the hardness kicks in. A very open problem is to find a "nicer" universal one way function (e.g. it would be very nice if f_{mult} is universal).