The material in this section is meant as a review. Despite that, many students report that they find this review useful for the rest of the book.

0.1 Logs & Exponents

You probably learned (and then forgot) these identities in middle school or high school:

\[(x^a)(x^b) = x^{a+b}\]
\[(x^a)^b = x^{ab}\]
\[\log_x(ab) = \log_x a + \log_x b\]
\[a \log_x b = \log_x (b^a)\]

Well, it’s time to get reacquainted with them again.

In particular, never ever write \((x^a)(x^b) = x^{ab}\). If you write this, your cryptography instructor will realize that life is too short, immediately resign from teaching, and join a traveling circus. But not before changing your grade in the course to a zero.

0.2 Modular Arithmetic

We write the set of integers as:

\[\mathbb{Z} \overset{\text{def}}{=} \{\ldots, -2, -1, 0, 1, 2, \ldots\}\],

and the set of natural numbers (nonnegative integers) as:

\[\mathbb{N} \overset{\text{def}}{=} \{0, 1, 2, \ldots\}\].

Note that 0 is considered a natural number.

Definition 0.1  For \(x, n \in \mathbb{Z}\), we say that \(n\) divides \(x\) (or \(x\) is a multiple of \(n\)), and write \(n \mid x\), if there exists an integer \(k\) such that \(x = kn\).

Remember that the definitions apply to both positive and negative numbers (and zero). We generally only care about this definition in the case where \(n\) is positive, but it is common for \(x\) to be both positive and negative.

Example  7 divides 84 because we can write 84 = 12 \cdot 7.
7 divides 0 because we can write 0 = 0 \cdot 7.
7 divides –77 because we can write –77 = (–11) \cdot 7.
–7 divides 42 because we can write 42 = (–6) \cdot (–7).
1 divides every integer (so does –1). The only integer that 0 divides is itself.
If \( d \mid x \) and \( d \mid y \), then \( d \) is a \textbf{common divisor} of \( x \) and \( y \). The largest possible such \( d \) is called the \textbf{greatest common divisor (GCD)}, denoted \( \gcd(x,y) \). If \( \gcd(x,y) = 1 \), then we say that \( x \) and \( y \) are \textbf{relatively prime}. The oldest “algorithm” ever documented is the procedure that Euclid described for computing GCDs (ca. 300 BCE):

\[
\text{gcd}(x, y): \quad \text{// Euclid’s algorithm}
\]

\[
\begin{align*}
\text{if } y & = 0 \text{ then return } x \\
\text{else return } \text{gcd}(y, x \mod y)
\end{align*}
\]

\begin{definition} \textbf{(% operator)} \end{definition}

Let \( n \) be a positive integer, and let \( a \) be any integer. The expression \( a \mod n \) (usually read as “\( a \) \mod \( n \)”\) represents the remainder after dividing \( a \) by \( n \). More formally, \( a \mod n \) is the \textbf{unique} \( r \in \{0, \ldots, n-1\} \) such that \( n \) divides \( a - r \).\(^1\)

Pay special attention to the fact that \( a \mod n \) is always a \textbf{nonnegative} number, even if \( a \) is negative. A good way to remember how this works is that “\( a \) is \( (a \mod n) \) more than a multiple of \( n \).”

\begin{example} \end{example}

\begin{align*}
21 \mod 7 & = 0 \text{ because } 21 = 3 \cdot 7 + 0. \\
20 \mod 7 & = 6 \text{ because } 20 = 2 \cdot 7 + 6. \\
-20 \mod 7 & = 1 \text{ because } -20 = (-3) \cdot 7 + 1. \text{ (\text{-20 is one more than a multiple of 7.})} \\
-1 \mod 7 & = 6 \text{ because } -1 = (-1) \cdot 7 + 6.
\end{align*}

Unfortunately, some programming languages define \( \% \) for negative numbers as \((\text{-}a) \mod n\), so they would define \(-20 \mod 7\) to be \((\text{-}20 \mod 7) = \text{-}6\). This is madness, and we can’t continue to let programming language designers get away with it! For now, if you are using some programming environment to play around with the concepts in the class, be sure to check whether it defines \( \% \) in the correct way.

\begin{definition} \textbf{(}Z_n\text{)} \end{definition}

For positive \( n \), we write \( \mathbb{Z}_n \equiv \{0, \ldots, n-1\} \) to denote the set of \textbf{integers modulo} \( n \). These are the possible remainders one obtains by dividing by \( n \).\(^2\)

\begin{definition} \textbf{(}\equiv_n\text{)} \end{definition}

For positive \( n \), we say that integers \( a \) and \( b \) are \textbf{congruent modulo} \( n \), and write \( a \equiv_n b \), if \( n \mid (a - b) \). An alternative definition is that \( a \equiv_n b \) if and only if \( a \mod n = b \mod n \).

You’ll be in a better position to succeed in this class if you can understand the (subtle) distinction between \( a \equiv_n b \) and \( a = b \mod n \):

\[a \equiv_n b: \text{ In this expression, } a \text{ and } b \text{ can be integers of any size, and any sign. The left}
\]

\[\text{and right side have a certain relationship modulo } n.\]

\(^1\)The fact that only one value of \( r \) has this property is a standard fact proven in most introductory courses on discrete math.

\(^2\)Mathematicians may recoil at this definition in two ways: (1) the fact that we call it \( \mathbb{Z}_n \) and not \( \mathbb{Z}/(n\mathbb{Z}) \); and (2) the fact that we say that this set contains integers rather than congruence classes of integers. If you appreciate the distinction about congruence classes, then you will easily be able to mentally translate from the style in this book; and if you don’t appreciate the distinction, there should not be any case where it makes a difference.
$a = b \% n$: This expression says that two integers are equal. The “=” rather than “≡” is your clue that the expression refers to equality over the integers. “$b \% n$” on the right-hand side is an operation performed on two integers that returns an integer result. The result of $b \% n$ is an integer in the range \{0, \ldots, n − 1\}.

Example “99 ≡_{10} 19” is true. Applying the definition, you can see that 10 divides 99 − 19. On the other hand, “99 = 19 \% 10” is false. The right-hand side evaluates to the integer 9, but 99 and 9 are different integers.

In short, expressions like $a \equiv_n b$ make sense for any $a, b$ (including negative!), but expressions like $a = b \% n$ make sense only if $a \in \mathbb{Z}_n$.

Most other textbooks will use notation “$a \equiv b \pmod{n}$” instead of “$a \equiv_n b$.” I dislike this notation because “$\pmod{n}$” is easily mistaken for an operation or action that only affects the right-hand side, when in reality it is like an adverb that modifies the entire expression $a \equiv b$. Even though $\equiv_n$ is a bit weird, I think the weirdness is worth it.

Tips & Tricks

You may often be faced with some complicated expression and asked to find the value of that expression mod $n$. This usually means: find the unique value in $\mathbb{Z}_n$ that is congruent to the result. The straightforward way to do this is to first compute the result over the integers, and then reduce the answer mod $n$ (i.e., with the $\% n$ operator).

While this approach gives the correct answer (and is a good anchor for your understanding), it’s usually advisable to simplify intermediate values mod $n$. Doing so will result in the same answer, but will usually be easier or faster to compute:

Example We can evaluate the expression $6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 \% 11$ without ever calculating that product over the integers, by using the following reasoning:

\[
6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 = (42) \cdot 8 \cdot 9 \cdot 10 \\
\equiv_{11} 9 \cdot 8 \cdot 9 \cdot 10 \\
= (72) \cdot 9 \cdot 10 \\
\equiv_{11} 6 \cdot 9 \cdot 10 \\
= (54) \cdot 10 \\
\equiv_{11} 10 \cdot 10 \\
= 100 \\
\equiv_{11} 1
\]

In the steps that only work mod 11, we write “$\equiv_{11}$.” We can write “= ” when the step holds over the integers (although it is not wrong to use $\equiv_{11}$ for those steps, since if two expressions represent the same integer, then they are also congruent mod 11).

“Simplify intermediate values mod $n$” doesn’t always have to mean “reduce all intermediate values mod $n$ with the $\% n$ operation.” Sometimes an expression can by simplified by substituting a value with something not in the range \{0, \ldots, n − 1\}, as in the following example:
Example  I can compute $7^{500} \% 8$ in my head, by noticing that $7 \equiv_8 -1$ and simplifying thusly:

$$7^{500} \equiv_8 (-1)^{500} = 1.$$ 

Similarly, I can compute $89^2 \% 99$ in my head, although I have not memorized the integer $89^2$. All I need to do is notice that $89 \equiv_99 -10$ and compute this way:

$$89^2 \equiv_99 (-10)^2 = 100 \equiv_99 1$$

Since addition, subtraction, and multiplication are defined over the integers (i.e., adding/subtracting/multiplying integers always results in an integer), these kinds of tricks can be helpful.

On the other hand, dividing integers doesn’t always result in an integer. Does it make sense to use division when working mod $n$, where the result always has to lie in $\mathbb{Z}_n$? The answer to this question awaits later in the course.

### 0.3 Strings

We write $\{0, 1\}^n$ to denote the set of $n$-bit binary strings, and $\{0, 1\}^*$ to denote the set of all (finite-length) binary strings. When $x$ is a string of bits, we write $|x|$ to denote the length (in bits) of that string, and we write $\overline{x}$ to denote the result of flipping every bit in $x$. When it’s clear from context that we’re talking about strings instead of numbers, we write $0^n$ and $1^n$ to denote strings of $n$ zeros and $n$ ones, respectively (rather than the result of raising the integers 0 or 1 to the $n$ power).

**Definition 0.5**  
(*⊕, \text{xor}*): When $x$ and $y$ are strings of the same length, we write $x \oplus y$ to denote the bitwise exclusive-or (xor) of the two strings.

For example, $\textcolor{red}{0011} \oplus \textcolor{green}{0101} = \textcolor{blue}{0110}$. The following facts about the xor operation are frequently useful:

\[
\begin{align*}
    x \oplus x &= 000\cdots & \text{xor'ing a string with itself results in zeroes.} \\
    x \oplus 000\cdots &= x & \text{xor'ing with zeroes has no effect.} \\
    x \oplus 111\cdots &= \overline{x} & \text{xor'ing with ones flips every bit.} \\
    x \oplus y &= y \oplus x & \text{xor is symmetric.} \\
    (x \oplus y) \oplus z &= x \oplus (y \oplus z) & \text{xor is associative.}
\end{align*}
\]

See if you can use these properties to derive the very useful fact below:

$$a = b \oplus c \iff b = a \oplus c \iff c = a \oplus b.$$ 

There are a few ways to think about xor that may help you in this class:

- **Bit-flipping**: Note that xor’ing a bit with 0 has no effect, while xor’ing with 1 flips that bit. You can think of $x \oplus y$ as: “starting with $x$, flip the bits at all the positions where $y$ has a 1.” So if $y$ is all 1’s, then $x \oplus y$ gives the bitwise-complement of $x$. If $y = \textcolor{red}{1010}\cdots$ then $x \oplus y$ means “(the result of) flipping every other bit in $x$.”
Many concepts in this course can be understood in terms of bit-flipping. For example, we might ask “what happens when I flip the first bit of \( x \) and send it into the algorithm?” This kind of question could also be phrased as “what happens when I send \( x \oplus 1000\cdots \) into the algorithm?” Usually there is nothing special about flipping just the first bit of a string, so it will often be quite reasonable to generalize the question as “what happens when I send \( x \oplus y \) into the algorithm, for an arbitrary string \( y \)?”

**Addition mod-2**: \( \oplus \) is just addition mod 2 in every bit. This way of thinking about \( \oplus \) helps to explain why “algebraic” things like \((x \oplus y) \oplus z = x \oplus (y \oplus z)\) are true. They are true for addition, so they are true for \( \oplus \).

This also might help you remember why \( x \oplus x \) is all zeroes. If instead of \( \oplus \) we used addition, we would surely write \( x + x = 2x \). When working mod 2, we have \( 2 \equiv_2 0 \), so \( 2x \) is congruent to \( 0x \).

### Definition 0.6

We write \( x \| y \) to denote the **concatenation** of strings \( x \) and \( y \).

### 0.4 Functions

Let \( X \) and \( Y \) be finite sets. A function \( f : X \to Y \) is:

- **injective** (1-to-1) if it maps distinct inputs to distinct outputs. Formally: \( x \neq x' \Rightarrow f(x) \neq f(x') \). If there is an injective function from \( X \) to \( Y \), then we must have \( |Y| \leq |X| \).

- **surjective** (onto) if every element in \( Y \) is a possible output of \( f \). Formally: for all \( y \in Y \) there exists an \( x \in X \) with \( f(x) = y \). If there is a surjective function from \( X \) to \( Y \), then we must have \( |Y| \geq |X| \).

- **bijective** (1-to-1 correspondence) if \( f \) is both injective and surjective. If there is a bijective function from \( X \) to \( Y \), then we must have \( |X| = |Y| \).

### 0.5 Probability

**Definition 0.7** (Distribution)

A **(discrete)** probability distribution over a set \( X \) of outcomes is usually written as a function “\( \Pr \)” that associates each outcome \( x \in X \) with a probability \( \Pr[x] \). We often say that the distribution **assigns** probability \( \Pr[x] \) to outcome \( x \).

For each outcome \( x \in X \), the probability distribution must satisfy the condition \( 0 \leq \Pr[x] \leq 1 \). Additionally, the sum of all probabilities \( \sum_{x \in X} \Pr[x] \) must equal 1.

**Definition 0.8** (Uniform)

A special distribution is the **uniform distribution** over a finite set \( X \), in which every \( x \in X \) is assigned probability \( \Pr[x] = 1/|X| \).

We also extend the notation \( \Pr \) to **events**, which are collections of outcomes. If you want to be formal, an event \( A \) is any subset of the possible outcomes, and its probability is defined to be \( \Pr[A] = \sum_{x \in A} \Pr[x] \). We always simplify the notation slightly, so instead of writing \( \Pr[\{ x \mid x \text{ satisfies some condition} \}] \), we write \( \Pr[\text{condition}] \).
Example  A 6-sided die has faces numbered \(\{1, 2, \ldots, 6\}\). Tossing the die (at least for a mathematician) induces a uniform distribution over the choice of face. Then \(\Pr[3 \text{ is rolled}] = 1/6\), and \(\Pr[\text{an odd number is rolled}] = 1/2\) and \(\Pr[\text{a prime is rolled}] = 1/2\).

**Tips & Tricks**

Knowing one of the probabilities \(\Pr[A]\) and \(\Pr[\neg A]\) (which is "the probability that \(A\) doesn’t happen") tells you exactly what the other probability is, via the relationship

\[
\Pr[A] = 1 - \Pr[\neg A].
\]

This is perhaps the most basic fact about probability, but it is rather surprising how frequently it is useful. Often one of \(\Pr[A]\) and \(\Pr[\neg A]\) is much easier to calculate than the other. So if you get stuck trying to come up with an expression for \(\Pr[A]\), try working out an expression for \(\Pr[\neg A]\) instead.

Example  I roll a six-sided die, six times. What is the probability that there is some repeated value? Let’s think about all the ways of getting a repeated value. Well, two of the rolls could be 1, or three of rolls could be 1, or all of them could be 1, two of them could be 1 and the rest could be 2, etc. This is getting out of hand!

The probability that we want is

\[
1 - \Pr[\text{all 6 rolls are distinct}].
\]

This complementary event happens exactly when the sequence of dice rolls spell out a permutation of \(\{1, \ldots, 6\}\). There are \(6! = 720\) such permutations, out of \(6^6 = 46656\) total possible outcomes. Hence, the answer to the question is

\[
1 - \frac{6!}{6^6} = 1 - \frac{720}{46656} = \frac{45936}{46656} \approx 0.9846
\]

Another trick is one I like to call **setting breakpoints** on the universe. In other words, imagine stopping the universe at some point where some random choices have happened, and others have not yet happened. This is best illustrated by example:

Example  A classic question asks: when rolling two 6-sided dice what is the probability that the dice match? The most straight-forward way to answer is:

When rolling two 6-sided dice, there are \(6^2 = 36\) total outcomes, so each outcome gets probability \(1/36\). There are 6 outcomes that make the dice match: both dice 1, both dice 2, both dice 3, and so on. Therefore, the probability of rolling matching dice is \(6/36 = 1/6\).

A different way to arrive at the answer goes like this:

Imagine I roll the dice one after another, and I pause the universe (set a breakpoint) after rolling the first die but before rolling the second one. At this point, the universe has already decided the result of the first die; call that value \(d\). The dice will match only if the second roll comes up \(d\). Rolling \(d\) with the second die (indeed, rolling any particular value) happens with probability \(1/6\).

This technique of setting breakpoints is simple but powerful and frequently useful. Some other closely related tricks are: (1) postponing a random choice until the last possible moment, just before its result is used for the first time, and (2) switching the relative order of independent random choices.
Precise Terminology

It is tempting in this course to say things like "x is a random string." But a statement like this is sloppy on two accounts.

First, is 42 a random number? Is "heads" a random coin? These questions don’t make a lot of sense. Being "random" is not a property of an outcome (like a number or a side of a coin) but a property of the process that generates an outcome. Instead of saying "x is a random string," it’s much more precise to say “x was chosen randomly.”

Second, usually when we use the word "random," we don’t mean any old probability distribution. We usually mean to refer to the uniform distribution. Instead of saying “x was chosen randomly," it’s much more precise to say “x was chosen uniformly” (assuming that really is what you mean).

Every cryptographer I know (even your dear author) says things like “x is a random string” all the time to mean “x was chosen uniformly [from some set of strings].” Usually the meaning is clear from context, at least to the other cryptographers in the room. But all of us could benefit by having better habits about this sloppy language. Students especially will benefit by internalizing the fact that randomness is a property of the process, not of the individual outcome.³

0.6 Notation in Pseudocode

We’ll often describe algorithms/processes using pseudocode. In doing so, we will use several different operators whose meanings might be easily confused:

← When \( D \) is a probability distribution, we write “\( x \leftarrow D \)” to mean “sample \( x \) according to the distribution \( D \).”

If \( \mathcal{A} \) is an algorithm that takes input and also makes some internal random choices, then it is natural to think of its output \( \mathcal{A}(y) \) as a distribution — possibly a different distribution for each input \( y \). Then we write “\( x \leftarrow \mathcal{A}(y) \)” to mean the natural thing: “run \( \mathcal{A} \) on input \( y \) and assign the output to \( x \).”

We overload the “\( \leftarrow \)” notation slightly, writing “\( x \leftarrow X \)” when \( X \) is a finite set to mean that \( x \) is sampled from the uniform distribution over \( X \).

:= We write \( x := y \) for assignments to variables: “take the value of expression \( y \) and assign it to variable \( x \).”

?= We write comparisons as \( ? \) (analogous to “==” in your favorite programming language). So \( x ? y \) doesn’t modify \( x \) (or \( y \)), but rather it is an expression which returns true if \( x \) and \( y \) are equal.

³There is a branch of theoretical CS called Kolmogorov complexity that can actually give coherent meaning to statements like “x is a random string.” Kolmogorov complexity isn’t relevant to this book, but I have to at least acknowledge its existence when I’m making these sweeping statements.
You will often see this notation in the conditional part of an if-statement, but also in return statements as well. The following two snippets are equivalent:

\[
\text{return } x ? y \quad \iff \quad \begin{cases} \text{if } x \geq y: \\
\quad \text{return true} \\
\quad \text{else:} \\
\quad \text{return false}
\end{cases}
\]

In a similar way, we write \( x \in S \) as an expression that evaluates to true if \( x \) is in the set \( S \).

### 0.7 Asymptotics (Big-O)

Let \( f : \mathbb{N} \to \mathbb{N} \) be a function. We characterize the asymptotic growth of \( f \) in the following ways:

\[
f(n) \text{ is } O(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty
\]

\[
\iff \exists c > 0 : \text{for all but finitely many } n : f(n) < c \cdot g(n)
\]

\[
f(n) \text{ is } \Omega(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} > 0
\]

\[
\iff \exists c > 0 : \text{for all but finitely many } n : f(n) > c \cdot g(n)
\]

\[
f(n) \text{ is } \Theta(g(n)) \iff f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n))
\]

\[
\iff 0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty
\]

\[
\iff \exists c_1, c_2 > 0 : \text{for all but finitely many } n : \\
\quad c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)
\]

### Exercises

0.1. Consider rolling several \( d \)-sided dice, where the sides are labeled \( \{0, \ldots, d - 1\} \).

(a) When rolling two of these dice, what is the probability of rolling snake-eyes (a pair of 1s)?

(b) When rolling two of these dice, what is the probability that they don’t match?

(c) When rolling three of these dice, what is the probability that they all match?

(d) When rolling three of these dice, what is the probability that at least two of them match? This includes the case where all three match.

(e) When rolling three of these dice, what is the probability of seeing at least one 0?
0.2. When rolling two 6-sided dice, there is some probability of rolling snake-eyes (two 1s). You determined this probability in the previous problem. In some game, I roll both dice each time it is my turn. What is the smallest value $t$ such that:

\[ \Pr[\text{I have rolled snake-eyes in at least one of my first } t \text{ turns}] \geq 0.5? \]

In other words, how many turns until my probability of getting snake-eyes exceeds 50%?

0.3. Rewrite each of these expressions as something of the form $2^x$.

(a) $(2^n)^n = ??$
(b) $2^n + 2^n = ??$
(c) $(2^n)(2^n) = ??$
(d) $(2^n)/2 = ??$
(e) $\sqrt{2^n} = ??$
(f) $(2^n)^2 = ??$