0.1 Modular Arithmetic

We write the set of integers and the set of natural numbers as:

\[ \mathbb{Z} = \ldots, -2, -1, 0, 1, 2, \ldots ; \]
\[ \mathbb{N} = \{0, 1, 2, \ldots \}. \]

**Theorem 0.1** (Division Theorem)

For all \( a, n \in \mathbb{Z} \) with \( n \neq 0 \), there exist unique \( q, r \in \mathbb{Z} \) satisfying \( a = qn + r \) and \( 0 \leq r < |n| \).

Since \( q \) and \( r \) are unique, we use \( \lfloor a/n \rfloor \) to denote \( q \) and \( a \% n \) to denote \( r \). Hence:

\[ a = \lfloor a/n \rfloor n + (a \% n). \]

The \( \% \) symbol is often called the modulo operator. Beware that some programming languages also have a \( \% \) operator in which \( a \% n \) always has the same sign as \( a \). We will instead use the convention that \( a \% n \) is always nonnegative.

**Definition 0.2**

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 \( kn = x \).

We say that \( a \) and \( b \) are congruent modulo \( n \), and write \( a \equiv_n b \), if \( n \mid (a - b) \). Equivalently, \( a \equiv_n b \) if and only if \( a \% n = b \% n \).

We write \( \mathbb{Z}_n \defeq \{0, \ldots, n-1\} \) to denote the set of integers modulo \( n \).

In other textbooks you may have seen “\( a \equiv_n b \)” written as “\( a \equiv b \pmod{n} \)”.

There is a subtle — and often confusing — distinction between the expressions “\( a \% n = b \)” and “\( a \equiv_n b \).” In the first expression, “\( a \% n \)” refers to an integer that is always between 0 and \( n - 1 \), and the equals sign denotes equality over the integers. In the second expression, the “\( \equiv_n \)” symbol denotes congruence modulo \( n \), which is a weaker condition than equality over the integers. Note that \( a = b \) implies \( a \equiv_n b \), but not vice-versa.

**Example**

99 \( \equiv_{10} 19 \) because 10 divides 99 – 19 according to the definition. But 99 \( \neq 19 \% 10 \) because the right-hand side evaluates to the integer 19 \% 10 = 9, which is not the same integer as the left-hand side 99.

When adding, subtracting, and multiplying modulo \( n \), it doesn’t affect the final result to reduce intermediate steps modulo \( n \). That is, we have the following facts about modular arithmetic:

\[ (a + b) \% n = \left( (a \% n) + (b \% n) \right) \% n; \]
\[ (a - b) \% n = \left( (a \% n) - (b \% n) \right) \% n; \]
\[ ab \% n = \left( (a \% n)(b \% n) \right) \% n. \]

Division is not always possible in \( \mathbb{Z}_n \); we will discuss this fact later in the class.
0.2 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 \( n \) zeroes and \( n \) ones, respectively.

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. So, for example, \( 0011 \oplus 0101 = 0110 \). The following facts about the xor operation are frequently useful:

- \( x \oplus x = 0^{|x|} \)  
  xor’ing a string with itself results in zeroes.
- \( x \oplus 0^{|x|} = x \)  
  xor’ing with zeroes has no effect.
- \( x \oplus 1^{|x|} = \overline{x} \)  
  xor’ing with ones flips every bit.
- \( x \oplus y = y \oplus x \)  
  xor is symmetric.
- \( (x \oplus y) \oplus z = x \oplus (y \oplus z) \)  
  xor is associative.

As a corollary:

\[
a = b \oplus c \iff b = a \oplus c \iff c = a \oplus b.
\]

We use notation \( x\|y \) to denote the concatenation of two strings \( x \) and \( y \).

0.3 Functions

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

- **injective** (1-to-1) if it never maps two different inputs to the same output. Formally:  
  \( x \neq x' \Rightarrow f(x) \neq f(x') \).
- **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 \( f(x) = y \).
- **bijective** (1-to-1 correspondence) if \( f \) is both injective and surjective. If this is the case, we say that \( f \) is a bijection. Note that bijectivity implies that \( |X| = |Y| \).

0.4 Probability

Definition 0.3 A *(discrete) probability distribution* \( \mathcal{D} \) over a set \( X \) of outcomes is a function \( \mathcal{D} : X \to [0,1] \) that satisfies the condition:

\[
\sum_{x \in X} \mathcal{D}(x) = 1.
\]

We say that \( \mathcal{D} \) assigns probability \( \mathcal{D}(x) \) to outcome \( x \). The set \( X \) is referred to as the support of \( \mathcal{D} \).

A special distribution is the *uniform* distribution over a finite set \( X \), which assigns probability \( 1/|X| \) to every element of \( X \).
Let $\mathcal{D}$ be a probability distribution over $X$. We write $\Pr_{\mathcal{D}}[A]$ to denote the probability of an event $A$, where probabilities are according to distribution $\mathcal{D}$. Typically the distribution $\mathcal{D}$ is understood from context, and we omit it from the notation. Formally, an event is a subset of the support $X$, but it is typical to write $\Pr[\text{cond}]$ where “cond” is the condition that defines an event $A = \{x \in X \mid x \text{ satisfies condition cond}\}$. Interpreting $A$ strictly as a set, we have $\Pr_{\mathcal{D}}[A] \stackrel{\text{def}}{=} \sum_{x \in A} \mathcal{D}(x)$.

The conditional probability of $A$ given $B$ is defined as $\Pr[A \mid B] \stackrel{\text{def}}{=} \Pr[A \land B] / \Pr[B]$. When $\Pr[B] = 0$, we let $\Pr[A \mid B] = 0$ by convention, to avoid dividing by zero.

Below are some convenient facts about probabilities:

$$\Pr[A] = \Pr[A \mid B] \Pr[B] + \Pr[A \mid \neg B] \Pr[\neg B];$$
$$\Pr[A \lor B] \leq \Pr[A] + \Pr[B].$$

(union bound)

**Precise Terminology**

It is common and tempting to use the word “random” when one really means “uniformly at random.” We’ll try to develop the habit of being more precise about this distinction.

It is also tempting to describe an outcome as either random or uniform. For example, one might want to say that “the string $x$ is random.” But an outcome is not random; the process that generated the outcome is random. After all, there are many ways to come up with the same string $x$, and not all of them are random. So randomness is a property of the process and not an inherent property of the result of the process.

It’s more precise and a better mental habit to say that an outcome is “sampled or chosen randomly,” and it’s even better to be precise about what the random process was. For example, “the string $x$ is chosen uniformly.”

**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:

$\leftarrow$ When $\mathcal{D}$ is a probability distribution, we write “$x \leftarrow \mathcal{D}$” to mean “sample $x$ according to the distribution $\mathcal{D}$.”

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 $X$.

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

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

\[
\text{return } x \neq y \quad \iff \quad \begin{align*}
\text{if } x = y: \\
\quad \text{return } \text{true} \\
\text{else:} \\
\quad \text{return } \text{false}
\end{align*}
\]

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

## 0.5 Asymptotics (Big-\( O \))

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

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

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

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

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

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

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

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