

1 Relations

- **Def:** A *relation* on a set S is a set of ordered pairs of elements of S , i.e. $R \subseteq S \times S$. $(a, b) \in R$ often denoted aRb .
- **Properties of Relations:**
 - R is *reflexive* if $(a, a) \in R$ for all $a \in S$.
 - R is *symmetric* if $(a, b) \in R \Rightarrow (b, a) \in R$ for all $a, b \in S$.
 - R is *transitive* if $(a, b) \in R, (b, c) \in R \Rightarrow (a, c) \in R$ for all $a, b, c \in S$.
 - R is an *equivalence relation* if R is reflexive, symmetric, and transitive.
- **Examples:**
 - $<$ on \mathbb{Z}
 - \leq on \mathbb{Z}
 - $=$ on \mathbb{Z}
 - “is an ancestor of” on people.
- If R is an equivalence relation on S and $a \in S$, then the *equivalence class of a* is the set $[a]_R = \{b \in S : (a, b) \in R\}$.
- **Proposition:** If R is an equivalence relation, then its equivalence classes form a partition of S . That is, every element $a \in S$ is in exactly one equivalence class (namely $[a]$).
Proof: In book.
- **The Congruence Relation:** For $a, b \in \mathbb{Z}$, we define $a \equiv b \pmod{n}$ if $a \bmod n = b \bmod n$. Equivalently, $n|(b - a)$.
 - The congruence relation modulo n is an equivalence relation on integers, and we’ll denote the equivalence class of integer a by

$$[a]_n = \{b \in \mathbb{Z} : b \equiv a \pmod{n}\} = \{\dots, a - 2n, a - n, a, a + n, a + 2n, \dots\} = [a \bmod n]_n.$$
 This is known as the *the congruence class of a modulo n* .
 - **Q:** How many distinct congruence classes are there modulo n ?

¹These notes are copied mostly verbatim from the lecture notes from the Fall 2010 offering, authored by Prof. Salil Vadhan. I will attempt to update them, but apologies if some references to old dates and contents remain.

- **Example:** The congruence classes modulo 3 are:
- We can do arithmetic on the equivalence classes. That is, we can *define* $[a]_n + [b]_n$ to be $[a + b]_n$, and $[a]_n \cdot [b]_n$ to be $[ab]_n$. These are well-defined (i.e. if $[a]_n = [a']_n$ and $[b]_n = [b']_n$, then $[a + b]_n = [a' + b']_n$) by the Homomorphic Properties of Mod.

2 Algorithms

Algorithms — step-by-step procedures for solving problems — have been a part of algebra since its earliest days. In his *Elements* (c. 300 BC), Euclid described the Euclidean algorithm, which remains to this current day (and we’ll see it below). The word *algorithm* is derived from the name of the Persian mathematician al-Khwarizmi (c. 780), who is considered one of the founders of algebra and gave the first general methods for solving linear and quadratic equations.

As hinted in the previous lecture, we too will be interested in algorithmic issues. We will not be satisfied to know that certain algebraic objects or solutions to equations exist, but we will want to know whether there are general and efficient methods for constructing them.

A precise mathematical model for what constitutes an algorithm wasn’t given until the 1930’s in the work of Church and Turing, who used it to show that certain well-defined mathematical problems have no algorithm whatsoever. These mathematical models were a major inspiration for the design of the first general-purpose computers in the 1940’s, and have remained relevant to understanding the power of computers even as technology has changed.

It will be too much of a detour for us to discuss these formal models of algorithms. Instead, we will informally introduce some of the basic terminology for discussing and comparing algorithms and give some examples. If you’re interested in reading more about algorithms, a standard textbook is Cormen-Leiserson-Rivest-Stein (see syllabus).

2.1 Defining Algorithms

Example: Grade-School Addition. We illustrate the idea of an algorithm with the simple example of grade-school addition, which takes two n -digit numbers $x = x_{n-1} \cdots x_0$ and $y = y_{n-1} \cdots y_0$ (say in base 10) and computes their sum $z = z_n z_{n-1} \cdots z_0$. As we all know, the algorithm computes the digits z_i of the result from the least significant ($i = 0$) to the most significant ($i = n$), keeping track of carry digits c_i as it goes.

GradeSchoolAddition($x_{n-1} \cdots x_0, y_{n-1} \cdots y_0$):

- Let $c_0 = 0$.
- For $i = 0$ to $n - 1$, let $c_{i+1}z_i$ be the base 10 representation of $x_i + y_i + c_i$.
- Output $c_n z_{n-1} z_{n-2} \cdots z_0$.

In general, an *algorithm* is an unambiguous, “step-by-step” procedure for transforming “inputs” to “outputs”. To understand this definition, we need to specify a few things:

- “inputs and outputs”: these are taken to be *strings* (finite sequences of symbols) over some fixed, finite *alphabet* of symbols, e.g. $\{0, 1, 2, \dots, 9\}$, $\{0, 1\}$, $\{a, b, c, \dots, z\}$.
- “step”: operation on individual symbols, e.g. on individual digits.
- Can be made mathematically precise via any one of many equivalent models (Turing Machine, RAM, ...).

Def: Algorithm A *computes* a function f if for every input string x , the output string, denoted $A(x)$, equals $f(x)$.

- The same set of rules should work for *all* possible inputs (infinitely many).
- **Example:** Grade-school addition computes the function $f(x\#y) = z$, where z is the base-10 representation of the sum of the numbers for which x and y are the base-10 representations, and $\#$ is an additional alphabet symbol just used to separate the two summands.
- Representation of input and output not too important. Any reasonable representation can be easily converted into any other, e.g. base 10 vs. binary (=base 2) representation.
- Not all well-defined functions have algorithms! For example, there is no algorithm for finding integer solutions to polynomial equations in many variables, e.g. $10wx^2y - 3xz + 2wz^9 - \dots = 0$ (aka Diophantine Equations). On PS2, you’ll see an algorithm for *linear* Diophantine Equations. Computability Theory (covered in cs121) studies which problems have algorithms and which do not.

2.2 Measuring Complexity

Some algorithms are better than others (even if they compute the same function). In particular, we prefer algorithms that take fewer steps.

Def: Algorithm A *runs in time* $T : \mathbb{N} \rightarrow \mathbb{R}^+$ if for every $n \in \mathbb{N}$ and *every* input x of length n , running $A(x)$ takes at most $T(n)$ steps.

- n is *length* of input (e.g. in binary or decimal) of the input numbers, not the numbers themselves. The length n of a positive integer x is $n = \lfloor \log_2 x \rfloor + 1$.
- This definition captures *worst-case analysis*, meaning that we require that the running time is at most $T(n)$ on *all* inputs of length n (even the “worst” ones).

The goals of most work on algorithms (cs124) and computational complexity theory (cs121, cs221) are to:

- Find the “fastest” possible algorithm for a problem.
- Distinguish easy problems (ones that have “fast” algorithms) from hard problems (ones that do not have *any* “fast” algorithms).

Typically, linear-time algorithms (e.g. $T(n) = 10n$ or grade-school addition) and quadratic time algorithms (e.g. $T(n) = 3n^2 + 60n$, grade-school multiplication) are considered “fast”, but exponential time (e.g. $T(n) = (1.2)^n$) are considered “slow”.

In order for our measure of running time to not depend too much on the specific model of computation (hardware) or on the representation of the data (eg decimal vs. binary numbers), we need to allow some slackness. In the next lecture, we’ll see convenient notation for ignoring constant factors.

3 Asymptotic Notation

In order for our measure of running time to not depend too much on the specific model of computation (what constitutes a single “step,” or the hardware of our computer) or on the representation of the data (eg decimal vs. binary numbers), we need to allow some slackness. The following is very convenient notation for ignoring constant factors and focusing on dominant terms (useful in many areas of mathematics, not just the analysis of algorithms).

Def (O-notation): For $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, we write $f = O(g)$ if there is a constant $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Informally $f = O(g)$ means “ f grows no more quickly than g ”
- Despite the strange (but convenient) notation, this is simply a binary relation on functions. You should think of $f = O(g)$ as simply “ $f \preceq g$ ” for a binary relation \preceq .

Examples:

- $n^3 = O(13n^3 + 2n^2 - 4n + 2)$?
- $13n^3 + 2n^2 - 4n + 2 = O(n^3)$?
- $n^{10} = O(13n^3 + 2n^2 - 4n + 2)$?
- $13n^3 + 2n^2 - 4n + 2 = O(n^{10})$?
- $13n^3 + 2n^2 - 4n + 2 = O(2^n)$?
- $2^n = O(13n^3 + 2n^2 - 4n + 2)$?
- Not all functions comparable: $\exists f, g$ s.t. $f \neq O(g)$ and $g \neq O(f)$.

When $O(g)$ is used in the middle of an expression, it is shorthand for some function f such that $f = O(g)$. Thus, $13n^3 + 2n^2 - 4n + 2 = 13n^3 + O(n^2)$.

Proposition:

1. If $\lim_{n \rightarrow \infty} f(n)/g(n) = c < \infty$ then $f = O(g)$.
2. If $\lim_{n \rightarrow \infty} f(n)/g(n) = \infty$ then $f \neq O(g)$.

Q: why aren't these "if and only if"?

When we express the running time of an algorithm using O notation, it allows us to ignore low-level details such as whether summing 3 digits takes 1 step or 5 steps:

- Grade-school addition algorithm takes time $O(n)$, as does subtraction.
- Kindergarten multiplication (via repeated addition) can take time more than 2^n .
- Grade-school multiplication and division algorithms take time $O(n^2)$. But there are asymptotically faster algorithms (see AM206 PS1)! Best known is roughly $O(n \log n)$. (This can be an AM206 essay topic.) It is a long-standing open question whether multiplication can be done in time $O(n)$.

Some other useful asymptotic notation:

- $f = \Omega(g)$ if $g = O(f)$. (" f grows at least as quickly as g ")
- $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$ (" f and g grow at the same rate", equivalence relation on functions)
- $\log_3 n = \Theta(\log_7 n)$.
- $6n^2 + 7 = \Theta(23n^2 - 6n + 8)$.

Polynomial Time. Before we said that linear and quadratic time are both considered "fast". More generally, we often look for algorithms that run in *polynomial time*, namely time $O(n^c)$ for some constant c . This is a coarse theoretical approximation to "feasibility." While not all polynomial-time algorithms are practical, this notion has the advantage that it is very robust to changes in computational model, representation of input, and composition of algorithms.

4 The Euclidean Algorithm

Goal: compute $\text{gcd}(x, y)$ for any two given integers x, y , not both zero.

A first attempt is to simply work from the definition of gcd.

TrialDivisionGCD(x, y):

1. Assume WLOG $x \geq y > 0$.
2. for $z = \min\{x, y\}$ down to 0,
 - if $z|x$ and $z|y$, then halt and output z .

Note that now we are describing algorithms at a higher level than before. Since we have already seen that there are efficient (polynomial-time) algorithms for basic arithmetic, we can use these operations freely in our algorithms. Thus, we usually do not need to refer directly to the decimal representations of the inputs, and can simply treat them as integers. (However, when we analyze the running time of our algorithm, we need to take into account not only the number of times we perform basic arithmetic operations, but also the sizes of the numbers on which these operations are performed — as this affects the time taken by the operations.)

Q: Does TrialDivisionGCD run in polynomial time?

To obtain a more efficient algorithm, we instead show how to reduce computing $\gcd(x, y)$ to computing $\gcd(x', y')$ for “significantly smaller” numbers x', y' . This is done via the following two lemmas:

Lemma 1: If $y > 0$, then $\gcd(x, y) = \gcd(y, x \bmod y)$.

Proof: We’ll show that the set of common divisors of x and y is the same as the set of divisors of y and $x \bmod y$. Write $x = qy + r$, where $r = x \bmod y$. So if d divides y and r , then it also divides y and $qy + r = x$. And if d divides y and x , then it also divides y and $x - qy = r$.

Lemma 2: If $0 < y \leq x$, then $x \bmod y < x/2$.

Proof: Case analysis.

If $y \leq x/2$, then $x \bmod y < y \leq x/2$.

If $x \geq y > x/2$, then $x \bmod y = x - y < x/2$.

This gives rise to the following algorithm.

Euclid(x, y):

1. Assume WLOG that $x \geq y \geq 0$.
2. If $y = 0$, output x .
3. Otherwise, output $\gcd(y, x \bmod y)$.

An equivalent formulation (replacing the recursion with a loop):

Euclid(x, y):

1. Assume WLOG $x \geq y > 0$.
2. Set $i = 1$, $x_1 = x$, $x_2 = y$.
3. Repeat until $x_{i+1} = 0$:
 - (a) Compute $x_{i+2} = x_i \bmod x_{i+1}$ (using a division algorithm).
 - (b) Increment i .
4. Output x_i .

Example: Euclid(60, 33)

Let's analyze the Euclidean Algorithm. Lemma 1 implies that the algorithm correctly computes the gcd function. For the running time, we have:

Proposition: If x and y are n -bit numbers, then $\text{Euclid}(x, y)$ performs at most $2n$ divisions, each being on numbers of bit-length at most n . In particular, the running time of Euclid is $O(n^3)$.

Proof: Let $D(z)$ denote the maximum number of divisions used in $\text{Euclid}(x, y)$ on inputs x, y whose product is at most z . Then $D(z) \leq D(z/2) + 1$ and $D(1) = 1$. Thus $D(z) \leq (\log_2 z) + 1$. If x and y are n -bit numbers, then $xy < 2^n \cdot 2^n = 2^{2n}$, so we use at most $D(xy) < 2n + 1$ divisions. (On PS2, you will improve this bound on the number of divisions.)

The Extended Euclidean Algorithm: In addition, to computing $\text{gcd}(x, y)$, this algorithm computes the coefficients s, t such that $\text{gcd}(x, y) = sx + ty$.

- Can be recovered from quotients and remainders in divisions in Euclidean algorithm.
- In addition to computing each x_i , write it as an integer linear combination of x and y . If $x_i = s_i x + t_i y$, $x_{i+1} = s_{i+1} x + t_{i+1} y$, and $x_i = q_i x_{i+1} + x_{i+2}$, then

$$x_{i+2} = x_i - q_i x_{i+1} = (s_i - q_i s_{i+1})x + (t_i - q_i t_{i+1})y.$$

Example: ExtendedEuclid(60, 33)

5 Algorithms for Primality and Factorization

- Algorithms for Primality: Given an integer x , is it prime?
 - Trial division: can take up to $2^{n/2}$ trials on n -bit integers. (Q: why not 2^n ?)
 - Randomized polynomial-time algorithms: $O(n^3)$ (very practical, taught in many algorithms courses)
 - Deterministic polynomial-time algorithm (2002!): $O(n^{6.01})$ (not yet practical)
- Algorithms for Factoring: Given an integer x , find its prime factorization.
 - Best fully proven algorithm: time $2^{O(\sqrt{n \log n})}$.

- Best heuristic algorithm: time $2^{O(n^{1/3}(\log n)^{2/3})}$.
- Largest factored RSA challenge ($x = pq$ for random $n/2$ -bit primes p, q): $n = 640$ (< 200 digits) in 2005. In contrast, testing even much larger p, q for primality and multiplying takes only seconds on a laptop.
- Gaps of this type are the basis for cryptography, e.g. construct encryption algorithms that are “easy” to use but “hard” to break.
- Hardness of factoring is a *conjecture* that might be wrong!