AM 106/206: Applied Algebra **Prof. Salil Vadhan**

Lecture Notes 3

September 13, 2010

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

    $$[a]_n = \{b \in \mathbb{Z} : b \equiv a \pmod{n}\} = \{\ldots, a-2n, a-n, a, a+n, a+2n, \ldots\} = [a \bmod n]_n.$$

    This is known as the *the congruence class of* a *modulo* n.
  - **Q:** How many distinct congruence classes are there modulo n?
  - **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 has 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 ($i = 1$), 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. $\{0, 1, 2, \ldots, 9\}$, $\{0, 1\}$, $\{a, b, c, \ldots, z\}$.

- "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. $10wx^2y - 3xz + 2wz^9 - \cdots = 0$ (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* $T : \mathbb{N} \to \mathbb{R}^+$ 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.