IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Advanced Course on Computational Complexity

Lecture 1: Three Ways to Express Problems

David Mix Barrington and Alexis Maciel
July 17, 2000

1. Overview

We'll begin the basic course by describing two *models of computation* and two *resource measures* for each. Then we can begin practicing complexity theory, by studying the resources needed to solve computational problems and particularly by studying *complexity classes* — sets of problems defined by being solvable under certain resource constraints.

We will define a *problem* as the decision problem for a *formal language* (a subset of the set Σ^* of all strings over some alphabet Σ). The decision problem is to take a string as input and determine whether it is in the language. Our two models will be *machines*, with the resources of *time* and *space*, and *boolean circuits*, with the resources of *size* and *depth*.

In the first week of these advanced lectures we will look instead at three other ways to define formal languages and thus to specify computational problems. Each of these will be motivated by some familiar area of pure mathematics, so we will not depend immediately on background from the basic course. Furthermore, each of these models can be extended from its original form, taking us eventually to some topics of current research.

The three models will be:

- Languages as inverse images of monoid homomorphisms,
- Languages built up from simple languages by simple operations, and
- Languages defined by properties expressed in first-order logic.

2. Defining Languages With Algebra

If Σ is a finite alphabet, the set Σ^* of finite strings over Σ forms a *monoid* under the operation of string concatenation. Like groups, monoids are closed under some associative operation and have an identity element — unlike groups, elements need not have inverses. A monoid *homomorphism* is a function from one monoid to another, satisfying the rule that $f(xy) = f(x)f(y)$ for any elements x and y of the first monoid. (We normally write the monoid operation multiplicatively, as we have here.)

We define a language $A \subset \Sigma^*$ to be *recognizable* if it is the *inverse image*, under a *monoid homomorphism*, of some subset of a *finite monoid*. That is, there must be a finite monoid M and a homomorphism ϕ from Σ^* (under concatenation) to M (under whatever its operation happens to be), and A must be the set $\phi^{-1}(X)$ for some subset X of M. Note that if A is so defined, we have a particular decision algorithm for A — given a string w, we compute $\phi(w)$ and determine whether it is in X. If we write w as $w_1 \ldots w_n$, note that because ϕ is a homomorphism we can calculate $\phi(w)$ as $\phi(w_1) \cdots \phi(w_n)$, so the key step of the decision operation is to "multiply" together a sequence of elements of M.

To illustrate this process, let Σ be $\{a, b, c\}$ and let A be the set of strings that have at least one a that occurs before at least one b. We define the monoid M to have four elements e, u, v, y, and z, where e is the identity, $uu = u$, $uv = uy = uz = y$, $vu = vz = z$, $vy = y$, $yu = yv = yy = yz = y$, $zu = z$, and $zv = zy = zz = y$. The reader may verify that this operation is associative, and that a string of u's and v's multiplies to y if and only if it has at least one u before at least one v. Given this, we choose ϕ to have $\phi(a) = u$, $\phi(b) = v$, and $\phi(c) = e$ (note that this defines a unique value of $\phi(w)$ for any string w in Σ^*). Now we can see that if we let $X = \{y\}$, then $\phi(w) \in X$ if and only if $w \in A$, so that $A = \phi^{-1}(X)$ and A is thus recognizable.

Some of you may recognize this notion as a disguised form of language recognition by *deterministic finite automata*, a restricted form of the machine model we will use in the basic course. Just as (by Cayley's Theorem) any finite group is isomorphic to a group of permutations of some finite set (under the operation of functional composition), any *finite monoid* is isomorphic to a set of functions (*transformations*) of some finite set (also under functional composition). Consider a recognizable language A as above, where M is a monoid of transformations of some finite set Q, and X is the set of transformations in M that take a particular element q_0 to an element of some subset F of Q. To determine whether a string w is in A, we can take q_0, apply the function $\phi(w_1)$ to it, apply $\phi(w_2)$ to the result, and so on until we finally apply $\phi(w_n)$ and see whether the final result is in F. This is *exactly* the behavior of a certain DFA on input w, where the DFA's transition function δ is given by $\delta(q, a) = \phi(a)(q)$

(the function $\phi(a)$ applied to q).

We have defined the basic notions of *algebraic automata theory*, which allows some of the tools of algebra to be applied to languages and computation. Later we'll see some applications of this.

3. Defining Languages By Operations

A natural way to define a mathematical system is *bottom-up*, where we define certain basic objects and ways to create new objects by combining old ones. A good example of this is the specification of languages by *regular expressions*, which defines the class of *regular languages*.

Fix an alphabet Σ. The basic regular languages over Σ are the empty language $\emptyset$ and the language $\{a\}$ for every letter a in Σ. These languages are denoted by the regular expressions "$\emptyset$" and "a" respectively. If S and T are regular languages, so are the following:

- The language ST of all strings uv where $u \in S$ and $v \in T$,
- The language $S \cup T$, the union of S and T, and
- The language S^* consisting of all strings that can be made by concatenating zero or more strings from S.

If R_S and R_T are regular expressions denoting S and T, we denote these three new languages by the regular expressions "$R_S R_T$", "$R_S \cup R_T$", and "R_S^*" respectively. In practice, we identify a regular language with its expression, so that we might refer to the "language $a(a \cup b)^* b^*$", for example.

A related set of languages are the *star-free regular languages*, where we replace the "star" operation with the operation of *complementation*: $\overline{S}$ is the set of all strings in Σ^* that are *not* in S. (It is a theorem that the regular languages themselves are closed under complementation — this will follow from Kleene's Theorem in Advanced Lecture 2. We need this fact to be sure that all star-free regular languages are in fact regular.)

Recall our example language A, the set of strings that have an a before a b. Any string in A is of the form $uavbw$ where u, v, and w are arbitrary strings in Σ^*. The concatenation operation on languages lets us express A as $\Sigma^* a \Sigma^* b \Sigma^*$, proving that A is regular by giving a regular expression for it (once we let "Σ" be an abbreviation for the regular expression "$a \cup b \cup c$"). Because Σ^* can be written without a star operation as "$\overline{\emptyset}$", the language A is star-free regular as well.

In the next lecture we'll prove Kleene's Theorem, that a language is regular iff it is recognizable. This has at least two interesting consequences. Since two independent simple definitions have led to the same class of languages, we have some evidence that this class is mathematically significant. And because we will be able to convert each kind of language definition to the other, we have some hope that techniques designed for one definition can be used in new ways for the other.

4. Defining Languages With First-Order Logic

A formal language implicitly defines a *property* that the strings in the language have and other strings do not. Our third new method of defining languages is to look at properties that can be *expressed* in a certain logical formalism, that of *first-order logic*. First-order logic builds formulas from certain *atomic predicates* using *first-order quantifiers*, which bind individual variables.

In our model the variables will range over *input positions* in the string, and there will be three atomic predicates:

- $x = y$, meaning that x and y are the same position,
- $x < y$, meaning that position x is to the left of y, and
- $I_a(x)$, meaning that position x contains the letter a (we have one of these predicates for each letter $a \in \Sigma$).

Variables start out *free*, and are *bound* by $\exists$ or $\forall$ quantifiers in the familiar way. A *sentence* is a formula with no free variables, and it is either true or false when considered for a particular string. (In logic terms, the string is a *model* which either *satisfies* the sentence or not.)

For example, we can express the property of having an a to the left of a b by the sentence:

$$\exists x : \exists y : I_a(x) \wedge I_b(y) \wedge (x < y)$$

We have thus expressed our example language A by a first-order formula, and shown that it is a *first-order expressible* language. The question of when a sentence is true of a string is straightforward given an understanding of quantifiers from a typical discrete math course. But in order to prove things about first-order expressibility, it's worthwhile to be a little more careful about the formal semantics.

We want to define satisfaction as a relationship between a string and a formula, and do so by induction on formulas. The problem is that induction on formulas starts with formulas that have free variables, and only gets to sentences at the end. It is not meaningful to talk about strings satisfying general formulas, so we must extend our notion of strings. If Σ is an alphabet and $\langle x_1, \ldots, x_k \rangle$ is a sequence of variables, we define a *marked word* to be a string $w \in \Sigma^*$ together with an function assigning each variable to a position in w. We can write a marked word using subscripting to indicate the *mark* for each variable — for example, $ab_x ab_y cc_z$ denotes the marked word consisting of the string *ababcc* and the assignment of x to the second, y to the fourth, and z to the sixth position.

If a formula Φ has k free variables, and W is a marked word with k marks for k variables, it is now meaningful to ask whether W satisfies Φ. We can define the property of satisfaction by induction on the process by which formulas are defined:

- A marked word satisfies the atomic predicate $I_a(x)$ iff the x-mark is on an a.
- A marked word satisfies the atomic predicate $x = y$ iff the x-mark and y-mark are in the same place.
- A marked word satisfies the atomic predicate $x < y$ iff the x-mark is to the left of the y-mark.
- A marked word satisfies a boolean combination of formulas according to the usual rules for boolean operators $\wedge$, $\vee$, and $\neg$.
- A marked word W satisfies a formula $\exists x : \Psi$, where x is free in Ψ, iff it is possible to add an x-mark to W so that the resulting marked word satisfies Ψ.

In the exercises we look at two interesting two-player games that give another interpretation to the semantics of these first-order formulas. These games will be useful in proving *lower bounds* on the power of these formulas to define certain properties.

5. Extensions of These Models

Each of these three models is capable of defining only certain languages, which turn out to be rather easy to decide in the machine and circuit models. (For example, we will see in lecture 3 that all of them can be decided by machines using $O(1)$ space.) But it is possible to *extend* each of the models to talk about more difficult languages:

- The algebraic model is based on homomorphisms into finite monoids. We can instead talk about *infinite monoids* (with some kind of constraints to keep us from being able to define anything), finite *groupoids* (non-associative structures) instead of monoids, and finally mappings that are more general than homomorphisms.

- Regular expressions are a way of sequentially defining new languages from old by particular language operations. One can also use the same operations to *implicitly* define several languages in terms of each other. One way to do this is the model of *context-free grammars*.

- We can extend our first-order formulas by adding new atomic predicates (such as "$x + y = z$", where x, y, and z are input positions), adding new types of quantifiers (such as "$Q_M x : \Psi(x)$", which true iff $\Psi(x)$ is true for a *majority* of the possible values of x), or allowing the number of quantifiers to increase with the input size in some uniform way.

6. Exercises

1. Prove that any finite monoid is isomorphic to a monoid of transformations of some finite set.

2. The argument above shows that a language is recognizable *in a certain way* iff it is the language of some DFA. (The subset X of M must be of a certain form.) Prove that *any* recognizable language is the language of some DFA.

3. Consider the following two-player game involving a string w and a first-order sentence Φ using the alphabet of w. For each quantifier in Φ, one of the players places a mark on w corresponding to the bound variable. For example, if Φ were $\exists x : \forall y : \forall z : \Psi$, where Ψ had no quantifiers, the game would consist of White placing an x-mark and then Black placing a y-mark and a z-mark. At the end of the game, White wins iff the resulting marked word satisfies Ψ (or in general, the quantifier-free part of Φ). Prove that White has a winning strategy for this game using w and Φ iff w satisfies Φ. (Hint: Use induction on the number of quantifiers in Φ.)

4. Here is a more complicated two-player game called an *Ehrenfeucht-Fraïssé game*. Here we start with two strings u and v, and a number k. There are k rounds. In each round i one player, Samson, places a mark x_i somewhere on one of the two strings. The other player, Delilah, then places an x_i-mark somewhere on

the other string. At the end of k rounds u and v have become marked words with the same marks. Samson wins the game iff at this point there is any atomic predicate that is true for one of the two marked words and false for the other. Prove that Delilah wins the k-move game on u and v iff u and v satisfy *exactly the same* k-quantifier sentences. (We call this property of strings being *k-equivalent.*)

5. Show that the language $(ab)^*$ is star-free by finding a star-free regular expression for it. (Hint: Begin by expressing Σ^* as $\overline{\emptyset}$.)

6. Let X be the set of strings over Σ such that every a in the string has a b immediately before it and a c somewhere after it. Show directly that this language is recognizable, regular, star-free regular, and first-order expressible.