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

Lecture 1: Problems, Models, and Resources

David Mix Barrington and Alexis Maciel
July 17, 2000

1. Overview

Complexity theory is the *mathematical* study of computation and the resources it requires. Saunders Mac Lane, in *Mathematics: Form and Function*, gives the following description of the mathematical method:

> Mathematics, at the beginning, is sometimes described as the science of Number and Space—better, of Number, Time, Space, and Motion. The need for such a science arises with the most primitive human activities. These activities presently involve counting, timing, measuring, and moving, using numbers, intervals, distances, and shapes. Facts about these operations and ideas are gradually assembled, calculations are made, until finally there develops an extensive body of knowledge, based on a few central ideas and providing formal rules for calculation. Eventually this body of knowledge is organized by a formal system of concepts, axioms, definitions, and proofs.

The human activity that we want to abstract and study is *computation*, in particular *discrete computation*. Humans have computed things by hand for millenia, and more recently have begun to do so using electronic computers. We have found our ability to compute to be limited by constraints on certain *resources* as well as by our limited knowledge — we can write computer programs for computations that require more time or memory than any conceivable computer could provide.

Although we certainly hope that complexity theory will have practical consequences, we will stick closely to the mathematical method that Mac Lane describes. We will choose formal models for problems and computations that are *simple* (so we

can do mathematics on them easily) but *practical* (so that they are related to computation in the real world). We will know that we are on the right track if these models turn out to be *robust*, that is, if the same mathematical objects occur in our theories despite changes in the definitions, and if apparently different definitions give rise to the same objects.

A good example to keep in mind is the definition of *what is computable*. In the 1930's, several different mathematicians proposed definitions of "computable functions": Turing machines, Church's lambda-calculus, Kleene's recursive functions, and Post's rewriting systems. It was found that in each system there was a class of functions computable by terminating computations (Turing-decidable, recursive) and a larger class of functions definable by computations that do not terminate (Turing-recognizable, recursively enumerable). Furthermore, the corresponding classes in each system are *provably* the same class, telling us that these two classes are important mathematical objects and not just artifacts of the definitions.

2. Problems

Our basic notion of a problem will be the *decision problem* for a formal language. Our input will be considered to be a sequence of *bits*, or sometimes as a sequence of letters in some finite *alphabet*. (This corresponds with actual practice in computers, as all data is represented as bits, sometimes broken into groups of bits such as bytes, words, or characters in ASCII or Unicode.) We represent the set of all finite bit strings by the notation $\{0,1\}^*$, and the set of all finite strings over an alphabet A by the notation A^*. A *formal language* over the alphabet A is any subset of A^*, that is, any set of strings whose letters are in A. For example, the set of all words in a given dictionary forms a (finite) formal language over $A = \{a, b, \ldots, z\}$. The set of all words over this alphabet with an even number of j's, for example, forms an infinite formal language.

The decision problem for a formal language is to take an input string and determine whether it is in the language. Naturally, there are different kinds of problems we want to solve with discrete computations. A more general definition of a problem might be to compute a function from strings to strings. But given our models and resource measures, we will find that most of the difficulty of computations can be captured by particular decision problems. For example, if f is a function from A^* to A^*, we can look at the *graph* of f, which is the set of ordered pairs $\langle a, b\rangle$ such that $f(a) = b$. The decision problem for the graph is roughly equivalent to the problem of computing the function. Similarly, other kinds of discrete problems can be represented as decision problems for formal languages.

It is convenient, then, to think of a set of problems as a set of formal languages. In computability theory, for example, we can now speak of the set of recursive languages, or the set of recursively enumerable languages. The basic object of our study will be a *complexity class* — the set of languages whose decision problems are solvable under a given resource constraint.

3. Asymptotics

Typically, the resources needed to solve an instance of a decision problem increase as the size of the input increases. We thus define the *complexity* of a language according to some resource measure as a *function*, where $f(n)$ is the greatest amount of resource used to solve the decision problem for any input of size n. (In these lectures we will deal only with this *worst-case* complexity measure — in other settings we might be interested in the resources used on an average input of size n, or in still other measures.) We thus need a mathematical formalism to deal with such functions.

In general our greatest concern is the way in which the complexity function increases as n goes to infinity. Because the units with which the resources are measured are often arbitrary, we don't worry about multiplying the function by constant factors. And if the complexity function should jump around rather than steadily increase, we are most interested in those values of n where it is large (since we would like to have a guarantee that a particular resource bound always works).

Any correct decision algorithm gives an *upper bound* on the complexity, because it shows that the problem can be solved with certain resource constraints. More difficult in general are *lower bounds* on complexity, which are proofs that *no algorithm* can solve the decision problem without using certain resources.

Our primary means for talking about functions will be *asymptotic notation*, a mechanism for comparing functions and/or deeming them to be equivalent. We define five relations on functions, corresponding to the five order relations $<$, $\leq$, $=$, $\geq$, and $>$:

- $f = o(g)$ means that for all ϵ there exists an n_ϵ such that $f(n) \leq \epsilon g(n)$ whenever $n > n_\epsilon$. (For most purposes it is equivalent to say that the limit as n goes to infinity of $f(n)/g(n)$ is zero.)
- $f = O(g)$ means that for some c and some n_0, $f(n) \leq cg(n)$ whenever $n > n_0$.
- $f = \Theta(g)$ means that $f = O(g)$ and $g = O(f)$.

- $f = \Omega(g)$ means that for some c and for infinitely many n, $f(n) \geq cg(n)$.
- $f = \omega(g)$ means that for every c, $f(n) \geq cg(n)$ for infinitely many n.

Any good algorithms book, such as *Introduction to Algorithms* by Cormen, Leiserson, and Rivest, will contain many examples and exercises concerning this notation. We give a few exercises below as well.

In complexity theory we often use an even coarser measure on functions. It often happens that a language can be *recoded*, or expressed in a different way that changes the input size by a polynomial factor. For example, a graph with n vertices and $O(n)$ edges requires $O(n^2)$ bits to represent as an adjacency matrix but $O(n \log n)$ bits to represent by an adjacency list. Our favorite resource bounds, therefore, are chosen to be robust with respect to polynomial changes in n:

- A *constant* function is one that is $O(1)$, that is, there is some constant c such that $f(n) \leq c$ for all (sufficiently large) n.
- A *logarithmic* bound is one that is $O(\log n)$. Note that either a change in the base of the logarithm, or a polynomial change in n such as replacing it by n^c for some c, multiplies $\log n$ by a constant and thus keeps the meaning of "$O(\log n)$" the same.
- A *polynomial* bound is one that is $n^{O(1)}$, that is, that there is some c such that $f(n) \leq n^c$ for all sufficiently large n. Note that a polynomial change in n would simply change the value of c.
- An *exponential* bound is $2^{n^{O(1)}}$, which we can think of as a polynomial bound on $\log(f)$.

The distinction between polynomial and exponential running time, for example, has consequences that should be familiar. If $f(n)$ is a polynomial function of n, for example, and we double n, then $f(n)$ gets multiplied by some constant. If $f(n)$ is an exponential function of n and we double n, however, $f(n)$ gets raised to some power. Exponential functions become inconceivably large for relatively small values of n — for example, 2^{1000} is much larger than the number of electrons that would fit into the observed universe. But polynomial functions, particularly the ones with small exponents that usually occur in the analysis of algorithms, in general take conceivable numbers to conceivable numbers.

4. Models of Computation

In these lectures we will consider only two of the many possible models of computation, *machines* and *circuits*. A machine will be an abstraction of the typical computer that reads some input, operates by performing simple computations on its internal memory, and reports an output. A circuit will be an abstraction of the electronic circuits inside computer chips, and an important general model of *parallel* computation.

Our model of a *machine* is a particular variant of the original *Turing machine* from the 1930's. The *input* is considered to be a string, which can be *read* by the machine one letter at a time, and which is *read-only*. For definiteness, we will say that the machine has an *input read head* which begins at the first letter of the input string, sees only the letter under it, and can be moved left or right one position on each computation step. The *memory* of the machine is also a sequence of letters that can be read *and written* by the machine. For definiteness, we will say that the machine has a constant number of *read/write heads* in this memory, which see the letter under them and can both write a new letter and move one position right or left on each computation step. The basic action of a computation step, then, is to see what is under each of the heads, decide what to do, write a new character under each read/write head, and move each of the heads.

To "decide what to do", the machine has a *program* and an *internal state*. The number of possible internal states (unlike the size of the memory) must be constant, independent of the size of the input. The program is a lookup table, giving a single *behavior* for each possible *observable state* of the machine. A behavior consists of move and write instructions for each head, and a new internal state. An observable state consists of the internal state plus the letters seen by each head. Perhaps the easiest way to see this is to give a typical instruction, such as "if we are in state 3, see an a on the input head, see a b on memory head 1, and see an a on memory head 2, we should go to state 7, move the input head left, write a b with memory head 1 and move it right, and write a c with memory head 2 and move it right." The number of instructions like this may be large, but note that it is constant with respect to the input size. Also note that the machine is *deterministic* — its behavior depends only on the things that it can observe and same observables always lead to the same behavior.

A machine gives its output by entering one of two special internal states, the *accepting halt state* or the *rejecting halt state*. A decision algorithm eventually enters the accepting state (if the input is in the language) or the rejecting state (if it isn't) and then stops processing.

We will also sometimes consider machines that can output a string rather than

just accept or reject. These have a *write-only output stream*, and each instruction in the program must be altered so that it either does or doesn't output a letter in addition to its other behavior.

Our other model, the boolean circuit, consists of *gates* and *wires*. A particular circuit has a fixed input size — let us say that it receives n boolean inputs (bits) $x_1, \ldots, x_n$ and will give us one boolean output: 0 if the string $x_1 \ldots x_n$ is not in the desired language and 1 if it is. Note that to decide an entire language, which may contain strings of arbitrary lengths, we need a *family* of boolean circuits, one for each input length. (Our resource measures will thus again be functions of the input size n.)

The gates of the circuit consist of the n *input gates* and some number of AND, OR, and NOT gates. Each gate evaluates to some boolean value once the values of the input bits are decided. A non-input gate has some number of *input wires*, each of which is an *output wire* of some other gate. The circuit must be *acyclic*, that is, it must be impossible to go from any gate to itself by following wires in the correct direction. The value of an AND gate is 1 if *all* of its input wires carry the value 1 (including the special case where there are no input wires). The value of an OR gate is 1 if *at least one* of its input wires carries the value 1 (so an OR gate with no inputs evaluates to 0). A NOT gate is required to have exactly one input, and outputs the negation of that input. The output of the circuit is the value of a designated *output gate*.

A normal form for circuits that is sometimes convenient is to "push all the NOTs to the bottom". Using the two DeMorgan laws, we can transform any boolean circuit to an equivalent one (computing the same function from the inputs to the output) that has NOT gates only immediately above the input gates.

We will sometimes consider boolean circuits with multiple output gates, so that they compute a function from $\{0,1\}^n$ to $\{0,1\}^m$ for some number m. A family of such circuits computes a function from $\{0,1\}^*$ to itself rather than deciding a language. We can also consider the input to be a string over some alphabet other than $\{0,1\}$, if we choose some way to code each input letter as a sequence of bits.

5. Resource Measures

We can now define two explicit resource measures for each of our models of computation. In Lecture 2 we will see some examples of decision algorithms, and analyze carefully how much of these resources each one uses. The measures for machines are *time* and *space*, for boolean circuits *size* and *depth*.

The *running time* of a machine algorithm is the number of individual computation steps it takes before halting. Because we are using asymptotic analysis to compare running times, we aren't too concerned with exactly what can happen on a single step, only what can happen in $O(1)$ steps. Running times of less than n tend to be less interesting, as in our current model a machine needs $O(n)$ steps to read all of its input. The most important time classes are the class P (languages decidable in polynomial time) and the class EXP (languages decidable in exponential time). More generally, DTIME(f) refers to the class of languages decidable in $O(f)$ time. (The D stands for "deterministic".)

Informally, P is often thought of as the set of *feasibly decidable* languages, because running times greater than polynomial are prohibitive except for small inputs, and *most* polynomial decision procedures known have small exponents and thus scale reasonably well with increasing input size.

The *space* used by a machine algorithm is the number of bits in its memory. In general DSPACE(f) is the class of languages that can be decided using $O(f)$ space. We will be particularly interested in three classes:

- DSPACE(1) is the class of languages decidable in constant space, independent of the input size. Since the contents of the memory of a DSPACE(1) machine can only have a constant number of different values, the contents of the memory can be recorded in the internal states of the machine so we can think of DSPACE(1) machines as having no memory at all other than the internal state.

  You may be familiar with a similar model of *finite-state machines* or *deterministic finite automata*. A DFA is a special kind of DSPACE(1) machine, one with no memory and restricted so it that its input head moves right at every step of the computation. In addition, a DFA does not have halting states. The computation simply ends when the head moves past the right end of the input. Some states are declared as accepting or final. If the computation ends in one of these states, the DFA accepts; otherwise, it rejects. It is easy to see that every language decided by a DFA can be decided by a DSPACE(1) machine. In the advanced course, we will prove that the reverse is also true.

- DSPACE($\log n$) or L is the class of languages decidable in logarithmic space. In $O(\log n)$ bits of memory, the machine can keep a *pointer* into its input, a number in the range from 0 to $n - 1$. This allows the machine to knowingly revisit the same place in the input, for example, something the DSPACE(1) machine cannot do. In fact, there is a sense in which any L machine can be thought of as keeping a constant number of pointers into the input, and nothing else, in its memory.

- PSPACE or DSPACE($n^{O(1)}$) is the class of languages decidable in polynomial space. Any language *not* in PSPACE is certainly not going to be practically decidable except for small inputs, because the memory requirements will go up to fast. But we will see that PSPACE also contains languages for which no P algorithm is known, and for which it is believed that none exists.

We now turn to our two resource measures for circuits. The *size* of a circuit is the total number of gates it contains. In our definition, every circuit contains at least n gates, the input gates. We will be concerned with the class PSIZE of languages decidable by circuit families where the size of each circuit is bounded by some polynomial in n. (Practically, a family of circuits is unlikely to be manufacturable unless it obeys a polynomial size bound.) In general, SIZE(f) is the class of languages decidable by circuit families where the size of the n-input circuit is $O(f(n))$.

The *depth* of a circuit is the length of the longest path from an input gate to the output gate. (Remember that the circuit must be acyclic, so that this is finite.) Depth is important because it corresponds to the time needed to evaluate the circuit in parallel (with something potentially happening at each gate). In a real electronic circuit, some physical event must happen at each gate to bring it to its correct value, taking some amount of time. The depth corresponds to the maximum number of these events that must occur *one after the other* to make the output gate have its correct value.

We will consider the classes AC^0 (constant depth, polynomial size) and AC^1 (logarithmic depth, polynomial size). (In this notation, the exponent refers to the power of $\log n$ being considered as a bound on the depth of the circuits.) Another set of classes includes a restriction on the circuits, that the *fan-in* of each AND and OR gate must be at most two. (The fan-in of a gate is the number of input wires coming into it.) Any circuit can be converted into an equivalent one obeying this restriction (see the exercises) but this conversion may increase the depth. We define NC^0 to be the class of languages decidable by circuit families with constant depth, polynomial size, and fan-in two. NC^1 is the class decidable by circuit families of logarithmic depth, polynomial size, and fan-in two.

Finally, we must mention a further resource used by circuit families. Our definitions above say that a language is in a given class if a certain circuit *exists*. But in the real world if a circuit is to be used to decide whether a string is in a language, that circuit must first be *constructed*. The simple classes we have defined so far are called *non-uniform* circuit classes. Later, when we want to establish relationships between circuit classes and machine classes, we will need to define *uniform* circuit classes, where there is a restriction on how difficult it can be to construct the circuits.

6. Exercises

1. What is A^* if A is the empty set? What if A has only one element? If A is finite, is A^* countable or uncountable?

2. Find the asymptotic relations among the following functions: $\sqrt{n}$, $3n+6$, $0.01n+1000000\log n + 17$, π^n, $2^n + n^3$, $\log_2 n$, and $\log_\pi n$.

3. Prove that if $f = O(g)$, then $f + g = \Theta(g)$.

4. Find a function f that satisfies $f = (\log n)^{\omega(1)}$ and $f = n^{o(1)}$.

5. Prove that $\log(n!) = \Theta(n \log n)$. (Hint: Show that $(n/2)^{n/2} \leq n! \leq n^n$.)

6. Find an explicit function f such that $f(n)! = n^{\Theta(1)}$.

7. Prove that pushing NOTs to the bottom does not change the depth of a circuit, or its total number of AND and OR gates. How could the size of the circuit be affected? (Hint: Use induction on the number of AND and OR gates.)

8. Find a function f such that *every* language is in SIZE(f).

9. Argue that $\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{AC}^1$. (Hint: Replace a gate with large fan-in by a binary tree of gates each with fan-in two.)

10. Argue that a language is in NC^0 iff for every n there is a set of $O(1)$ variables such that membership in the language depends only on those variables.