Lecture 1: Computational Models and Branching Programs

Anup Rao

January 3, 2023

Our goal in this course is to mathematically capture the concept of computation. In the broadest sense, what is computation? What is the best way to model computation? How can we reason about whether something can be efficiently computed or not? What are the resources that are worth measuring with regards to computational efficiency? Complexity theory is about addressing these questions.

During this class we shall explore many mathematical concepts that will help us deal with these topics. Along the way, we will naturally be led to studying some very interesting questions. Here is a sample of some of the questions we will discuss:

1. Can every function be computed?

2. Can every true statement be proved?

3. Can every slow algorithm be replaced with a faster algorithm that computes the same function?

4. Can every slow sequential algorithm be replaced with a faster parallel algorithm?

5. Is it possible to check the correctness of a computation that took a long time using much less time?

Throughout, we will use clever definitions and ideas from mathematics to reason about these questions.

Before we begin, we need to define what we mean by *computation*.

Computation

A first attempt at defining a computation might be to say that it is a process that *manipulates information* in some way, or has an *input-output* behavior. For example, when reading this sentence, your brain takes the information encoded graphically and translates that information into letter, words and ideas, and so performs a computation.

At this point you may feel that we have made the model too general to be useful, so it might be useful to explain what is *not* computation. A useful mathematical abstraction that captures some of the

things we have discussed is the abstraction of *functions*. Given two sets D, R, a function

$$f : D \to R$$

assigns a value $f(x) \in R$ to every element of $x \in D$. So, if we think of D as the set of all possible images, and R as the set of all sentences, f can be defined to be the function that maps the picture of a sentence to the actual sentence. This abstraction misses something that is inherent about physical computational processes: computations are local. At any point, the state of two parts of the brain that are far away from each other cannot affect each other.

Informally, a computational process is a process that manipulates information in some *local* or restricted way. Interesting computational processes are ones that manage to have a complicated global effect through incremental local steps. There are many such computational processes, and we will not be able to talk about all of them in detail in this course. The aim of this course is to show you, in broad strokes, how you can start to reason about such computational processes and their complexity.

Computational Complexity

HOW CAN WE DISTINGUISH functions that are easy to compute from functions that are hard? What makes some things easy and other things hard?

In order to tackle this kind of question, we first need mathematical models that captures exactly what the process we are interested in can do cheaply, and what takes more effort. We would like our models to be general enough that they capture most real computational processes, and simple enough that we can ask and understand easy questions about them.

A crucial issue is how the information being manipulated is encoded. For example, if numbers are encoded using their prime factorization (both in the input and output), then it is slightly easier for us to multiply two 100 digit numbers than if they are encoded using their digits. Under this representation, addition of numbers, and comparing two numbers becomes harder.

So, given a function $f : D \to R$, we would like to be able to quantify how difficult it is to compute this function. We shall make two immediate simplifications.

- We shall identify the the input domain with the set of binary strings of arbitrary length $\{0,1\}^*$, or the set of strings of some fixed length $\{0,1\}^n$. Since every countable set can be mapped to

For example, we are very good at reading text, but multiplying 100 digit numbers takes us considerably more time, even though the amount of information is contained in a picture is much more than the information contained in a 100 digit number.

Addition is hard when numbers are represented in terms of their factorization because we would have to factor the sum to bring it back in this representation. We do not know of any efficient algorithms for factoring numbers.
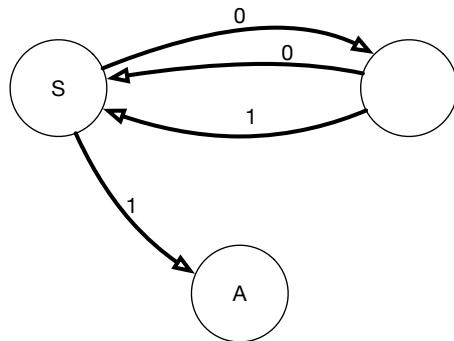
the first set, and every finite set can be mapped to the second, this does not lose too much generality.

- We shall often restrict the output domain to be $R = \{0, 1\}$. This does lose some generality, but it will turn out that most of our ideas will easily translate to the situation when the output domain is bigger. Further, for most examples of functions to bigger domains that are hard to compute, we shall be able to easily find corresponding boolean functions that are hard to compute.

Next we give several examples of computational models, and discuss some strengths and weaknesses of each of them.

Finite Automata

A FINITE AUTOMATON CAN BE USED to compute functions $f : \{0, 1\}^* \to \{0, 1\}$. An example is shown in Figure 1. One starts at the start state (labeled S), and reads the input bit by bit, transitioning on the states. The output is 1 if and only if we ever hit the accept state (labeled A).

The weaknesses of this model:

- The set of functions $f : \{0, 1\}^* \to \{0, 1\}$ that can be computed by any finite automaton is not very general. Even more damning— finite automata cannot compute functions that we can efficiently compute in practice.

Only functions that represent *regular* languages can be computed by a finite automaton.
Try to prove that no finite automaton can compute the function whose output is 1 if and only if the input is a palindrome.

- It does not give a way to measure the complexity of computing functions. We can count the number of states in the automaton, but since this is always a constant independent of n, this measure

of complexity doesn't scale with the input size, and so does not map to what we are trying to capture — the level of difficulty involved in carryout the computation.

Finite automata are a *uniform* model of computation. This means that there is a single description of the process that can be used to carry out the computation no matter how long the length of the input is.

Branching Programs

BRANCHING PROGRAMS ARE VERY SIMILAR to finiate automata. This is a model that computes functions $f : \{0,1\}^n \to \{0,1\}$.

A branching program is a directed acyclic graph where every vertex is labeled by either a variable or an output value. Every vertex that is labeled by a variable has exactly two edges coming out of it, one labeled 1 and the other labeled 0. The vertices are partitioned into disjoint sets $L_0, \ldots, L_k$, and all edges go from some set L_i to L_{i+1}. There is a special designated start node in the graph. To carry out the computation, we start at the start state, and follow the indicated path by reading the variable that labels the state that we are on in each step. When we hit a node labeled by an output value, the output of the computation is that output value.

There are two measures of the complexity of the program. The *width* is the number of nodes in the largest layer. The length is the number of layers.

This model has the advantage that it can actually compute all functions:

Fact 1. *Every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a branching program with width 2^n, and length $n + 1$.*

Proof   Consider the branching program that is a rooted tree, where every node at level i reads the variable x_i. The program simply remembers the entire input. Each of the 2^n inputs x gets mapped to a distinct leaf, so that leaf can be labeled by the value $f(x)$ to compute f. ∎

Branching programs will be used later in the course as a way to capture computations that use a small amount of space (or memory). The main drawback of branching programs is that there are algorithms which run very quickly on computers in practice that we don't know how to model as small branching programs. So, the branching program complexity doesn't seem to capture everything we want to capture about efficient computation.

In class, we discussed how every finite automaton for a function $f : \{0,1\}^* \to \{0,1\}$ can be used to get a branching program of size $O(n)$ that computes f on all the inputs of length n.
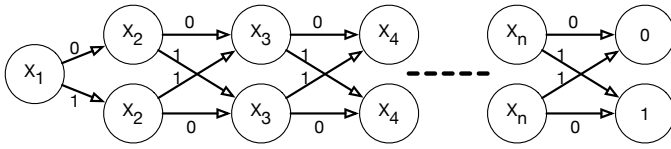
Figure 2: A branching program that computes whether or not the number of 1's in the input is even.

Branching programs define a *non-uniform* model of computation. The program only tells you how carry out computation on an input of length n. To talk about the asymptotic complexity of computing a function $f : \{0,1\}^* \to \{0,1\}$ that is defined on strings of all lengths, we need to talk about families of branching programs, and discuss the complexity of the programs as n gets larger and larger.