CSE531: Computational Complexity Theory January 4, 2012

Lecture 1 Turing Machines and Circuits

Lecturer: Anup Rao

1 What is Computation?

Our goal is to model the complexity involved in carrying out computation, though we shall not formally define exactly what computation is. Informally, a computational process is any process that manipulates information in some way. 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. One can think of the weather as a computational process: given the current state of clouds, ocean currents and many other factors, the laws of the universe produce an outcome that uses the information about the current state to generate a new state. At this point you may argue 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 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. A computational process is a process that manipulates information in this local way.

2 Computational Complexity

We would like to be able to distinguish computational processes that are doing something complicated from processes that are doing something simple. 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. 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 a computational process is. 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.

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 $\{0,1\}^*$. Since every countable set can be mapped to this set, this does not lose too much generality.

- We shall 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.

2.1 A uniform model: Turing Machines

A Turing Machine is essentially a program written in a particular programming language. The program has access to three arrays and three pointers:

- x which is accessed using the pointer i. x is an array that can be read but not written into.
- y which is accessed using the pointer j. y can be read and written.
- z which is accessed using the pointer k. z can be both read and written.

The machine is described by its code. Each line of code reads the bits x_i, y_j, z_k, and based on those values, writes new bits into y_j, z_k, and then possibly after incrementing or decrementing i, j, k, jumps to a different line of code or stops computing. Initially, the input is written in x and the goal is for the output to be written in z at the end. i, j, k are all set to 1 to begin with.

For example here is a program that copies the input to the output using a single line:

1. If x_i is empty, then HALT. Else set $z_k = x_i$ and increment each of i, k.

Here is another that outputs the input bits which are in odd locations:

1. If x_i is empty, then HALT. Else set $z_k = x_i$, increment each of i, k and jump to step 2.
2. If x_i is empty, then HALT. Else increment each of i, k and jump to step 1.

The exact details of this model are not important. The main reason we introduce it is to have a fixed model of computation in mind.

The reason this model is so powerful is because of the *Church-Turing Thesis*. The thesis says that *every* computational process can be modeled using a Turing machine as formalized above. Again, this is not a mathematical claim, but a wishy washy philosophical claim about the nature of the universe. As far as we know so far, it is a sound one. In particular if one changed the above model slightly (say by providing 10 arrays to the machine instead of just 3, or by allowing it to run in parallel), then one can simulate any program in the new model using a program in the model we have chosen. A lot of work has gone into proving that that the above model is *universal* in this sense, but we ignore the details.

Once we have fixed the model, we can start talking about the *complexity* of computing a particular function $f : \{0, 1\}^* \to \{0, 1\}$. Fix a turing machine M that computes a function f. There are two main things that we can measure:

- Time. We can measure many steps the turing machine takes in order to halt.
- Space. We can measure the maximum value of j during the run of the turing machine.

The following fact is immediate:

Fact 1. *The space used by a machine is at most the time it takes for the machine to run.*

The following theorem should not come as a surprise to most of you. It says that there is a machine that can compile and run the code of any other machine efficiently:

Theorem 2. *There is a turing machine M such that given the code of any Turing machine α and an input x as input to M, if α takes T steps to compute an output for x, then M computes the same output in $O(CT \log T)$ steps, where here C is a number that depends only on α and not on x.*

We shall say that a machine runs in time $t(n)$ if for every input x, the machine halts after $t(|x|)$ steps (here $|x|$ is the length of the string x). Similarly, we can measure the space complexity of the machine. The crucial point is that small changes to the model of Turing machines does not affect the time/space complexity of computing a particular function in a big way. Thus it makes sense to talk about the running time for computing a function f, and this measure is not really model dependent.

2.2 A non-uniform model: Circuits

A different kind of model is the model of circuits. A circuit computing a function $h : \{0,1\}^n \to \{0,1\}$ is a directed acyclic graph with the following properties. Every vertex (also called gate) has fan-in either 0 or 2. If the fan in is 0, then the vertex is labeled with an input variable x_i. If the fan-in is 2, then the vertex is labeled with a function that maps 2 bits to 1 bit. There is a designated output gate. The circuit is evaluated by evaluating each gate in turn until the output gate is evaluated.

There are two major quantities we can measure to capture the complexity of a circuit:

- *Size.* We can measure the number of gates in the circuit.
- *Depth.* We can measure the length of the longest input to output path. The depth complexity is a measure of how much parallel time it takes to compute the function.

As with Turing Machines, observe that small changes to the model do not really affect these complexity classes. For example, one can change the fan-in of the gates to 10, and this will affect the size of the smallest circuit computing a function only by a constant factor.

Given a function $f : \{0,1\}^* \to \{0,1\}$, we say that the function has a size $s(n)$ circuit family if for every n, there is a circuit of size $s(n)$ that computes the function correctly on inputs of length n. Similarly, we can talk about the depth complexity of computing a function.

Theorem 3. *Every function $h : \{0,1\}^n \to \{0,1\}$ can be computed in size $O(2^n/n)$ and depth $O(n)$.*

We shall prove the bound of $O(2^n)$. The idea is to use recursion. Let h_0 denote the function on $n-1$ bits given by $h_0(x) = h(x,0)$, and $h_1 = h(x,1)$. Then by induction we can compute h_0, h_1 recursively, and combine them using the value of the last bit to obtain h.

Thus, if S_n is the size of the resulting circuit when the underlying function takes an n bit input, we have proved that

$$S_n \leq 2S_{n-1} + c,$$

where c is a constant that is independent of n and the function f. Expanding this recurrence, and using the fact that $S_1 \leq c$, we get that

$$S_n \leq \sum_{i=1}^{n} 2^i c = c.(2^{n+1} - 1) \leq 2c \cdot 2^n,$$

where here we used the formula for computing the sum of a geometric series.