CS 121: Introduction to Theoretical Computer Science Fall 2019

Section 1

Prof. Boaz Barak

1 Representation

1.1 Representing Numbers

1.1.1 Naturals and Rationals

To represent a non-negative integer $x \in \mathbb{N}$, we take its binary expansion:

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^{n-1-i}, \qquad x_i \in \{0,1\}.$$

Above, n is the smallest number such that $2^n > x$, and we will have that the string $(x_0, x_1, \ldots, x_{n-1})$ is the binary representation of x.

How can we use the work we have done above to represent negative integers and hence get representations for all of the integers? We can simply add a digit in front that tells us the *sign* of the number, so for a given $x \in \mathbb{Z}$, we can use a similar representation as before but add a sign big σ to the front.

Finally, we can think about how to represent rational numbers. Since rationals are a/b where $a, b \in \mathbb{Z}$, we just have to figure out a way to put the representation of two numbers together. To do this, ideally we could just append the character ; to the alphabet, and then write the representation of a, insert the ; in between, and write the representation of b. In practice, we do something that's kind of clever: for each 1 and 0 we would typically write, we write 11 and 00 instead. Then, for the ; symbol, we write 01. This kind of logic generalizes between representing rationals: what we are doing here is essentially writing an ordered tuple of integers, and we can apply the same logic for combining the representations of more than 2 integers.

1.1.2 Reals

In practice, we are able to very closely approximate the representation of reals with a rational number that is extremely close to that real number. However, in this section, we will see that we are not able to represent the reals exactly using binary strings $x \in \{0,1\}^*$.

Theorem 1. *There is no one-to-one function from* $\mathbb{R} \to \{0,1\}^*$.

Proof. First, define the set

$$\{0,1\}^\infty = \{f \mid f : \mathbb{N} \to \{0,1\}\}.$$

That is, $\{0,1\}^\infty$ is the set of functions from the natural numbers to $\{0,1\}$. The proof of this theorem can be seen in two parts.

1. There exists a one-to-one map $\{0,1\}^\infty \to \mathbb{R}$.
2. There does not exist a one-to-one map $\{0,1\}^\infty \to \{0,1\}^*$.

In these notes, we will take the first part for granted: the proof is not particularly relevant or interesting (see the lecture notes for this). We focus on the second step: proving that there does not exist a one-to-one map $\{0,1\}^\infty \to \{0,1\}^*$.

This is equivalent to proving that there does not exist an onto map from $\{0,1\}^* \to \{0,1\}^\infty$. To do this, we use the famous diagonalization argument. Suppose we had a map between strings and functions $StF : \{0,1\}^* \to \{0,1\}^\infty$. We will produce some function $f : \mathbb{N} \to \{0,1\} \in \{0,1\}^\infty$ such that $f \neq StF(x)$ for all $x \in \{0,1\}^*$, as this will show that StF is not onto.

To get such a function, we construct f iteratively. For each binary string x, let $n(x)$ be the number it corresponds to in the binary representation of natural numbers. Note that by looking at all $x \in \{0,1\}^*$, we will get all $n \in \mathbb{N}$. Then, we define

$$f(n(x)) = 1 - StF(x)(n(x)).$$

Let's break this down: we define f such that the value it takes at a number, n, is exactly *not* the value that the function obtained by mapping the binary representation of n into $\{0,1\}^\infty$ takes at n. Defining the map this way ensures that $f \neq StF(x)$ for all x, since we know that for each x, $f(n(x)) \neq StF(x)(n(x))$. However, this means that f is distinct from all of the functions in the image of StF, but $f \in \{0,1\}^\infty$, so StF cannot be onto, as desired. $\square$

1.2 Beyond Representing Numbers

1.2.1 Some Formalism

First, we will formalize some ideas that we have seen for representing various objects (perhaps not necessarily numbers).

Definition 2. Let $\mathcal{O}$ be some set. Then, a *representation scheme* for $\mathcal{O}$ consists of a pair of functions (E, D) where $E : \mathcal{O} \to \{0,1\}^*$ is a total **one-to-one** function and $D : \{0,1\}^* \to \mathcal{O}$ is a (possibly partial) **onto** function such that $D(E(o)) = o$ for all $x \in \mathcal{O}$. E is the *encoding function*, and D is the *decoding function*.

Proposition. *Given a one-to-one encoding function $E : \mathcal{O} \to \{0,1\}^*$, there exists a decoding function D as desired by the definition above to form a representation scheme for $\mathcal{O}$.*

Hence, to write down a representation scheme, we only really need to find a one-to-one encoding.

Definition 3. A length n string x is a *prefix* of a length $n' \geq n$ string x' if $x_i = x'_i$ for all $1 \leq i \leq n$. A representation is *prefix free* if there do not exist two elements $o, o' \in \mathcal{O}$ such that $E(o)$ is a prefix of $E(o')$.

The following theorem (from the lecture notes) tells us a pretty useful fact: if we have a prefix free representation for an alphabet, we can automatically get a representation for the set of strings that can be formed using that alphabet for free.

Theorem 4. *Suppose that $E : \mathcal{O} \to \{0,1\}^*$ is prefix-free. Then, if we define $\overline{E} : \mathcal{O} \to \{0,1\}^*$ for every $o_0, \ldots, o_{k-1} \in \mathcal{O}$ to be*

$$\overline{E}(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1)\cdots E(o_{k-1})$$

$\overline{E}$ is one-to-one.

Sketch of Proof. The main intuition here is that if we want to decode $(o_0, \ldots, o_n)$ from its representation $x = E'(o_0, \ldots, o_n) = E(o_0)\ldots E(o_n)$, we can do so by first finding the first prefix x_0 of x such is a representation of some object, decode this object, remove x_0 from x to obtain a new string x', and then find the first prefix x_1 of x', etc. That E is prefix-free will ensure that x_0 will be $E(o_0)$, x_1 will be $E(o_1)$, etc. □

1.2.2 Graphs

Let's review the idea of using adjacency matrices and adjacency lists to represent graphs. Recall that a graph consists of a set of vertices V and edges E. Let's identify the vertices V with the set $\{1, \ldots, n\} = [n]$ where $n = |V|$. Then, the adjacency matrix representation is an $n \times n$ matrix A such that $A_{ij} = 1$ if and only if there exists an edge $(i, j) \in E$. If the graph is undirected, when there is an edge $\{i, j\}$, we say that $(i, j), (j, i) \in E$ (replace the edge with two directed edges going both ways). On the other hand, the adjacency matrix representation is a set of n lists, where the i-th list gives the neighbors of vertex i.

1.3 Practice Problems

1. Can we come up with a representation of the complex numbers? Why or why not? What if we had an exact representation of the reals?

2. Write down an encoding $E : \{0,1,2\}^* \to \{0,1\}^*$. Can you come up with a one-to-one function $S \to \{0,1,2\}^*$, where S is the set of n-tuples of natural numbers?

3. For each of the following sets S, determine if there exists an encoding $S \to \{0,1\}^*$:
   (a) S is the set of infinite integer sequences that are uniformly zero after some point
   (b) S is the set of infinite integer sequences that are uniformly zero before some point

2 Computation

2.1 Circuits and Programs

The first set of building blocks that we might think to use for building a model of computation would be the AND, OR, and NOT operations. This leads to **Boolean circuits** and **AON-CIRC programs**.

A Boolean circuit with n inputs, m outputs, and s gates, is a labeled directed acyclic graph (DAG) $G = (V, E)$ with $s + n$ vertices. There are exactly n of the vertices with no in-neighbors, which we call the inputs and label them with X[0],. . .,X[$n - 1$]). The other s vertices are gates, and each is labeled with $\wedge$, $\vee$ or $\neg$. Gates labeled with $\wedge$ (AND) or $\vee$ (OR) have two in-neighbors, and those labeled with $\neg$ (NOT) have one in-neighbor. Finally, exactly m of the gates are outputs, and are labeled with Y[0], . . .,Y[$m - 1$].

For Boolean circuit C and input $x \in \{0, 1\}^n$, we find $C(x)$ by assigning the input vertices X[0],. . .,X[$n-1$] the values $x_0, \ldots, x_{n-1}$, apply each gate on the values of its in-neighbors, and then output the values $y_0, \ldots, y_{m-1}$ that correspond to the output vertices Y[0],. . .,Y[$m - 1$].

We say that a Boolean circuit C computes a function $f : \{0, 1\}^n \to \{0, 1\}^m$ iff $C(x) = f(x)$ for every $x \in \{0, 1\}^n$.

An AON-CIRC program is a string of lines of the form "foo = AND(bar,blah)", "foo = OR(bar,blah)", and "foo = NOT(bar)", where "foo", "bar" and "blah" are variable names. Variables of the form X[i] are known as input variables, and variables of the form Y[j] are known as output variables. In every line, the variables on the right hand side of the assignment operators must either be input variables or variables that have already been assigned a value before.

If P is an AON-CIRC program and $x \in \{0, 1\}^n$ is an input, $P(x)$ is the string $y \in \{0, 1\}^m$ corresponding to the values of the output variables Y[0],. . .,Y[$m - 1$] in the execution of P line by line where we initialize the input variables X[0],. . .,X[$n - 1$] to the values $x_0, \ldots, x_{n-1}$.

An AON-CIRC program P computes a function $f : \{0, 1\}^n \to \{0, 1\}^m$ iff $P(x) = f(x)$ for every $x \in \{0, 1\}^n$.

Another building block we can use is the NAND function. We can define **NAND circuits** and **NAND-CIRC programs**, as well as the notion of computability for those models, analogously to Boolean circuits and AON-CIRC programs, respectively.

These aren't the only building blocks we can use. We could also use the operations IF, ZERO, and ONE where ZERO : $\{0, 1\} \to \{0\}$ and ONE : $\{0, 1\} \to \{1\}$ are the constant zero and one functions, and IF : $\{0, 1\}^3 \to 0, 1$ is the function that on input (a, b, c) outputs b if $a = 1$ and c otherwise.

2.2 Equivalence of Models

Theorem 5. *Let $f : \{0, 1\}^n \to \{0, 1\}^m$ and $s \geq m$ be some number. Then f is computable by a Boolean circuit of s gates if and only if f is computable by an AON-CIRC program of s lines.*

Sketch of Proof. The idea is that AON-CIRC programs and Boolean circuits are just different ways of describing the exact same computational process. Consider for example, an AND gate in a Boolean circuit corresponding to computing the AND of two previously-computed values. In a AON-CIRC program this will correspond to the line that stores in a variable the AND of two previously-computed variables. The formal proof pretty much just involves describing an explicit method for constructing an AON-CIRC program corresponding to any given Boolean circuit, and vice versa. ☐

Theorem 6. *For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.*

Sketch of Proof. The key here is that we can compute AND, OR, and NOT each with at most 3 NAND gates:

- $\text{NOT}(a) = \text{NAND}(a, a)$
- $\text{AND}(a, b) = \text{NAND}(\text{NAND}(a, b), \text{NAND}(a, b))$
- $\text{OR}(a, b) = \text{NAND}(\text{NAND}(a, a), \text{NAND}(b, b))$

Then, for any given Boolean circuit, we can replace its gates with the corresponding NAND gates to result in a NAND circuit that is at most 3 times larger than the original Boolean circuit. ☐

Theorem 7. *For every $f : \{0,1\}^n \to \{0,1\}^m$ and $s \geq m$, f is computable by a NAND-CIRC program of s lines if and only if f is computable by a NAND circuit of s gates.*

Sketch of Proof. This is pretty much the same as the proof of showing AON-CIRC programs and Boolean circuits are equivalent, except we now just have one gate, NAND, instead of having AND, OR, and NOT. ☐

Theorem 8. *For every sufficiently large s, n, m and $f : \{0,1\}^n \to \{0,1\}^m$, the following conditions are all equivalent to one another:*

1. *f can be computed by a Boolean circuit (with $\wedge, \vee, \neg$ gates) of at most $O(s)$ gates.*
2. *f can be computed by an AON-CIRC straight-line program of at most $O(s)$ lines.*
3. *f can be computed by a NAND circuit of at most $O(s)$ gates.*
4. *f can be computed by a NAND-CIRC straight-line program of at most $O(s)$ lines.*

Sketch of Proof. This follows pretty directly from theorems 5, 6, and 7. Theorem 5 yields that statements 1 and 2 are equivalent. Theorem 7 tells us that statements 3 and 4 are equivalent. Finally, theorem 6 gives us that statement 1 implies statement 3. We can prove the converse of theorem 6 with a similar technique (we can replace instances of $NAND$ and a NOT and a $NAND$). Hence, statements 1 and 3 are equivalent. This finishes up chaining up the four statements together as equivalent, so we're done. ☐

2.3 Practice Problems

1. Define CMP that on input (a, b, c, d), CMP function outputs 1 if the natural number represented by (a, b) is greater than the natural number represented by (d, e). Describe an AON-CIRC program computing the CMP function.

2. The NOR operation to, on input (a, b), output 1 if $a = b = 0$ and 0 otherwise. Let NOR-CIRC be the programming language where we have just the NOR operation. Compare the power of AON-CIRC programs and NOR-CIRC programs (i.e. either show that there is some function that only one type of program can compute or if a function is computable by one type of program iff it is computable by the other).