**CS 121: Introduction to Theoretical Computer Science** Fall 2019

Section 4

*Prof. Boaz Barak*

# 1 Uncomputability

Before we dive into the main topic of this section, we review the concept of computability.

## 1.1 Recall: computability

So far, the functions $F$ that we considered in this class had $\{0,1\}^n$ as its domain, where $n \in \mathbb{N}$. We've seen that *finite* functions

$$F : \{0,1\}^n \to \{0,1\}$$

are *computable* in the sense that we can always find a NAND-TM program $P_F$ such that $P_F(s) = F(s)$ for all $s \in \{0,1\}^n$.

The question we ask is, *would this still be the case when the domain of the function is* $\{0,1\}^*$*?* Recall that $\{0,1\}^*$ is simply the set that contains binary strings of all lengths. In other words, given any function $G$ with

$$G : \{0,1\}^* \to \{0,1\}$$

can we find a NAND-TM program $P_G$ with $P_G(s') = G(s')$ for all $s' \in \{0,1\}^*$?

As it turns out, we *can't* for some functions.

## 1.2 Theorem: existence of an uncomputable function

**Theorem 1.** *There exists a function that is* not *computable by any NAND-TM program.*

*Intuition.* It is important to get the intuition here. Part of the difficulty of expanding the domain of functions from $\{0,1\}^n$ to $\{0,1\}^*$ is that there are only finitely many strings of $n$ bits but infinitely many of arbitary length. This difference explains why we cannot apply the same approach to computing every function that we used with functions on $\{0,1\}^n$, which relied on enumerating the finite set of outputs.

However, there is another distinction which suggests why there are some functions we cannot compute: NAND-TM consists of objects with finite descriptions, while functions $\{0,1\}^* \to \{0,1\}$ have infinitely long description lengths. We can imagine a NAND-TM program as a compressed

representation of the functions with "nice patterns", but the number of possible functions is so large that it overwhelms the ability of a finitely-described NAND-TM program to capture all of them.

### 1.3 Proof:

*Proof.* Consider the set of all NAND-TM programs $P : \{0,1\}^* \to \{0,1\}$ (takes any binary input string and outputs one bit). Since *all* NAND-TM programs have an encoding, we can lexicographically order them (they are countably infinite). Suppose that $(P_0, P_1, P_2, \dots)$ is the lexicographic ordering of all NAND-TM programs.

| | 0 | 1 | 10 | 11 | 100 | 101 | 110 | ... |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | 0 | 1 | 1 | 0 | 1 | 1 | 1 | ... |
| $P_1$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| $P_2$ | 1 | 1 | 0 | 1 | 1 | 1 | 1 | ... |
| $P_3$ | 1 | 1 | 1 | doesn't halt | 1 | 1 | 1 | ... |
| $P_4$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| $P_5$ | 0 | 0 | 1 | 1 | 1 | 1 | doesn't halt | ... |
| $P_6$ | 1 | 1 | 0 | 1 | 1 | 1 | 1 | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | |

(Note: this table has been filled randomly just for the sake of illustrating the procedure.)

The first column of the above table, as we discussed, is just an ordering of *all* NAND-TM programs, and the first row is the standard lexicogrphical ordering of all strings. Remember again that the first column contains *ALL* NAND-TM programs. So *if we can construct a function that disagrees with all the programs (returns a different output for some string from all of the programs in the first column), that proves the claim.*

Consider this function $F_{impossible}$ defined by flipping the bits in the green diagonal above. Note that we just consider "doesn't halt" to be the same as 0.

| | 0 | 1 | 10 | 11 | 100 | 101 | 110 | ... |
|---|---|---|---|---|---|---|---|---|
| $F_{impossible}$ | 1 | 0 | 1 | 1 | 0 | 0 | 0 | ... |

Now the claim is that $F_{impossible}$ is different from all of the programs in the first column. $F_{impossible}$ is different from the function simulated by $P_0$, since they return different outputs for the string 0. It is also different from $P_1$ since their outputs differ on 1. Similarly, $P_2$ on 10, and $P_3$ on 11. It is not too difficult to see that $P_n$ is going to disagree with $F_{impossible}$ on the binary representation of $n$.

Therefore, $F_{impossible}$ is different from all of the programs in the first column of the table, i.e. no NAND-TM program can simulate $F_{impossible}$.

### 1.4 Exercise

Consider the set $P(\mathbb{N})$ of all subsets of $\mathbb{N}$. Show that there is no one-to-one and onto function between $\mathbb{N}$ and $P(\mathbb{N})$.

### 1.5 Exercise

Is the set of NAND-TM programs countable? What about the set of functions $\{0,1\}^* \to \{0,1\}$? How can we get an alternate proof of Theorem 1 using these facts?

## 2 Reduction

In the preceding section, we showed that there is some function from $\{0,1\}^*$ to $\{0,1\}$ that can't be simulated by any NAND-TM program, i.e. an uncomputable function. However, the uncomputable function that we constructed seemed rather contrived. After all, $F_{impossible}$ is constructed just so that it's different from all the NAND-TM programs in the list. In this section, we look at the technique called *reduction* which can be used to show the uncomputability of some less contrived functions.

The big picture for reduction goes like this:

- You have a problem $A$ that you *know* you *can't* solve.
- And there's this other problem $B$ that you're wondering if you can solve.
- You imagine (assume) that $B$ is solvable (*this is for the sake of contradiction*).
- As it turns out, if $B$ is solvable, then we can *use it* to solve $A$.
- But since $A$ is just simply not solvable, something that we assumed must've been wrong.
- So we deduce that $B$ can't be solvable (since that was the only assumption we made along the way).

Using reduction, we now prove the following.

### 2.1 Theorem: uncomputability of HALT

**Theorem 2.** *Let* $HALT : \{0,1\}^* \to \{0,1\}$ *be the function such that*

$$HALT(P,x) = \begin{cases} 0 & P \text{ halts on input } x \\ 1 & \text{otherwise} \end{cases}$$

*Then* $HALT$ *is* not *computable.*

The roadmap from above would look like the below in this particular case:

- We have a function $F_{impossible}$ that we know is not computable.
- And we are wondering if $HALT$ is computable.
- Assume for contradiction that $HALT$ is computable.
- If $HALT$ is computable, then $F_{impossible}$ should also be computable.
- But $F_{impossible}$ is not computable.
- Hence, $HALT$ couldn't have been computable.

### 2.2 Proof:

*Proof.* The idea is pretty clear from the roadmap above (hopefully?), so we just prove the crux of the argument.

If $HALT$ is computable, then $F_{impossible}$ should also be computable.

Assume that $HALT$ is computable. Then there is some NAND-TM program $P_{haltsolver}$ that computes $HALT$. In other words,

$$P_{haltsolver}(P, x) = \begin{cases} 0 & P \text{ halts on input } x \\ 1 & \text{otherwise} \end{cases}$$

Using $P_{haltsolver}$ as a subroutine, we build $P_{impossiblesolver}$ as follows.

Given input $s \in \{0,1\}^*$,

1. Compute $n$, which is just the value of $s$ in decimal.
2. Using $n$, it constructs $P_n$, which can be done in finite time (run down the lexicographically ordered list of all the strings until a valid description of $n^{\text{th}}$ NAND-TM program comes up).
3. Run $P_{haltsolver}$ on $(P_n, s)$. If it tells us that $P_n$ halts on $s$, then we simply flip the output of $P_n$ on $s$ after it halts.

   If $P_{haltsolver}$ tells us that $P_n$ doesn't halt on $s$, return 1 (because we considered not halting to be the same as 0 earlier).

Now notice that the above builds exactly what we proved to be impossible in the previous theorem (i.e. $F_{impossible}$). Hence, something must've been wrong in the assumptions that we've made along the way, and we only made one assumption: $HALT$ is computable. We conclude that $HALT$ is *not* computable.

### 2.3 Exercise

Let $E$ be defined as follows.

$$E(P) = \begin{cases} 0 & P \text{ accepts any string from } \{0,1\}^* \\ 1 & \text{otherwise} \end{cases}$$

Show that $E$ is *not* computable.

## 3 Incompleteness

### 3.1 Introduction

The notion of incompleteness explores the related idea that there are some true statements that we cannot prove. This discussion will require that we have some rough notion of proofs and a system for verifying them.

**Definition 3** (Proofs and Proof Verifiers)**.** For our purposes we will encode statements $x$ and proofs $w$ as binary strings $x, w \in \{0,1\}^*$.

We will then provide these to a proof verifier $V$. We require $V$ to satisfy a few conditions:

1. Soundness: $\exists w, V(x,w) = 1 \implies w$ is true. That is, there is no proof of a false statement.
2. Effectiveness: $V$ is computable.

Ideally, $V$ would also be *complete*: for every true statement $x'$ there exists some $w'$ that proves it ($V(x', w') = 1$). However, we will see that this is not the case. There exists a true statement $x$ which does not have a proof under $V$.

### 3.2 Incompleteness from Halting Problem

We can show that a proof verifier cannot be complete using our knowledge of the halting problem. We will approach this by reducing the halting problem to proof verification.

**Theorem 4.** *A proof verifier $V$ as defined in Definition 3 cannot be complete.*

*Proof.* Given a program $P$ and input $x$ we will attempt to prove one of the statements "$P$ halts on input $x$" and "$P$ does not halt on input $x$" by brute force. That is, enumerate all possible proofs $w \in \{0,1\}^*$ in order of increasing length.

For each of these, run $V$("$P$ halts on $x$", $w$) and $V$("$P$ does not halt on $x$", $w$). If $V$ accepts one of these statements and proofs, we have answered the halting problem on $(P, x)$. □

**Exercise 1.** The proof of Theorem 4 uses each of the properties of soundness, effectiveness and completeness. Where?