Section 4 (with solutions)

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. There are infinitely many binary strings in $\{0,1\}^*$, while there are finite number of strings in $\{0,1\}^n$. And that's precisely what makes it impossible to compute some functions since NAND-TM programs are finite objects.

1.3 Proof:

Proof. Consider the set of all NAND-TM programs $P : \{0,1\}^* \rightarrow \{0,1\}$ (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	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

(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 Solution

Solution.

Suppose for contradiction that it is possible to find a one-to-one and onto map between $P(\mathbb{N})$ and $\mathbb{N}$. Then it is possible to list all elements in $P(\mathbb{N})$ without missing out any (just list at the top the element that matches with $0 \in \mathbb{N}$ and the second row the one that matches 1, etc).

We encode all elements of $P(\mathbb{N})$ as an infinite sequence of 1's and 0's.

	0	1	2	3	4	5	6	...
$\{\}$	0	0	0	0	0	0	0	...
$\{0\}$	1	1	0	0	0	0	0	...
$\{1\}$	0	1	0	0	0	0	0	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\{1,2\}$	0	1	1	0	0	0	0	...
$\{1,3\}$	0	1	0	1	0	0	0	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

There's no reason why the first element should be $\{\}$, but just for concreteness we are putting it at the top. Just like in the proof of the uncomputable function theorem, we consider the subset S of $\mathbb{N}$ that corresponds to the sequence that you get by flipping all the green bits. Then S is not in $P(S)$ and therefore the assumption that the listing is complete was false. Hence, it had to be the case that there is no one-to-one and onto function between $\mathbb{N}$ and $P(\mathbb{N})$.

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*

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

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^{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.

2.4 Solution

Solution Suppose for contradiction that E is computable. Then there is a NAND-TM program P_E that computes E. Using P_E as a subroutine, we build P_{HALT} that computes $HALT$ (which we know we shouldn't be able to).

Given (P, x),

1. Construct a NAND-TM program P' that works as follows: Given x',
   (a) Runs P on x.
   (b) returns 1 if P halts.
2. Run P_E on P'.

Note that P_E returns 1 if and only if P halts on x. Hence, we've built a NAND-TM program that computes P_{HALT} and that means the assumption was false – E is *not* computable.