University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: William Cao
Date: January 31, 2024

LECTURE
7

Undecidable Problems and the Time Hierarchy

In the previous lecture, we introduced necessary prerequisites for diagonalization arguments: the binary encoding of Turing machines and the existence of a universal (nondeterministic) Turing machine. In this lecture, we finally apply these ideas to diagonalization arguments to show the existence of an undecidable language, and notably that HALT is undecidable. We then discuss the intuition that if we have two different time bounds, one of which is larger than the other, that there should some function that is computable in the more generous bound but not in the more tight. We first present two adjacent results to make us question if this intuition is actually correct. We then once again apply diagonalization to prove the time hierarchy theorem, the formalization of that intuition.

7.1 Undecidability

We first prove a major theorem: the existence of an undecidable language. To do this, we apply a diagonalization proof: by considering a table of all possible Turing machines and inputs, we can deliberately construct a language that does not match any Turing machine, and so is undecidable.

THEOREM 7.1. *There is a language $L \in \{0,1\}^*$ that is not decided by any Turing machine.*

Proof. Consider the following table of Turing machine outputs:

	0	1	00	01	…	x
M_0	0	1	1	-		1
M_1	-	0	-	1		0
$\vdots$						
M_α	1	-	1	0		0

In this table, the rows represent Turing machines ordered by their index α, and the columns represent inputs to those Turing machines. The entry in a cell (M_α, x) represents $M_a(x)$: the output of Turing machine M_α on input x. These outputs are labeled with either the binary values 0/1, or $-$ indicating any other result (the machine not terminating, or an output other than 0/1).

The idea behind the proof is as follows: all possible ”solutions” are included in this table, as all Turing machines have some encoding corresponding to a binary string. Therefore, if we can construct something outside of the table, that language cannot be decided by any Turing machine.

To do this, we consider all TM-input pairs (M_α, α). We describe a new language L, such that for every string α we take the opposite of what is output by $M_\alpha(\alpha)$, formally $L = \{x : M_x(x) = 0\}$. By construction, this language cannot be decided by any TM in the table: for any TM M_α, it will give the incorrect output on input α. These pairs are along the diagonal of the table, and this is what gives the technique of diagonalization its name.

As we have constructed a language that is not decided by any TM, we are done.

□

We now present the definition of HALT, a key undecidable problem:

Definition 7.2. HALT $= \{(M, x) : M$ halts on $x\}$

Based on our language earlier, we can show the well-known undecidability of the halting problem. To do this, we observe that knowing if a machine halts means we can indeed decide the above language.

Corollary 7.3. HALT *is undecidable.*

Proof. Suppose we have a TM H deciding HALT.

Let $L = \{x : M_x(x) = 0\}$, the language we have shown to be undecidable.

We then construct the following decider for L using H:

TM 1 Decider for L

```
Input: $x$
1: if $M_x$ halts on input $x$ then
       run $M_x(x)$ and return $[M_x(x) = 0]$
2: else
       output 0
```

This will exactly compute the language L: when $M_x(x) = 0$, we will return 1 (as the equality holds); otherwise, we will return 0. As L is uncomputable, this is a contradiction, and so no such H can exist.

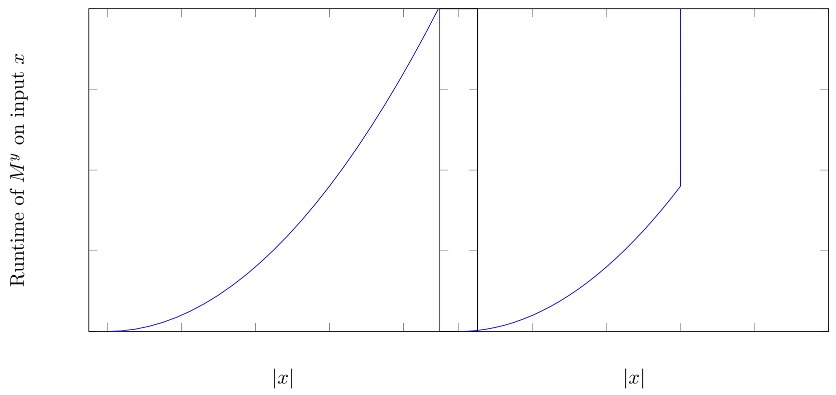
□

As another example, we present yet another language, this time a class of Turing machines:

Definition 7.4. QUADTIME $= \{M : M$ runs in time $O(n^2)\}$

This language is also undecidable. We show this using a reduction from HALT. To do this, we make use of the universal Turing machine to simulate the input Turing machine for exactly n^2 time steps.

Corollary 7.5. QUADTIME *is undecidable.*



(a) $M(y)$ doesn't halt (b) $M(y)$ does halt

FIGURE 7.1: Runtime of M^y in cases where $M(y)$ does and does not halt

Proof. Given an input (M, y) to HALT, we define a new TM M^y as follows:

TM 2 TM M^y for halting problem reduction

```
Input: $x$
1: for $|x|^2$ time steps do
       Simulate $M$ on input $y$
2: end for
3: if $M$ has halted during the simulation then
       loop forever
4: else
       halt
```

Consider the cases where $M(y)$ doesn't halt, and where $M(y)$ does halt. The behavior of M^y's runtime is given in Figure 7.1. In the former, the first branch of the if statement will never be taken, and so the resulting TM M^y will always halt in $|x|^2$ time steps. Therefore, $M^y \in$ QUADTIME. In the latter, the TM M^y will halt eventually when $|x|$ is long enough for the simulation to finish, and in those cases will loop forever. This means that $M^y \notin$ QUADTIME. Note that both of these chains of reasoning can be done in the reverse direction, and so $(M, y) \in \text{HALT} \Leftrightarrow M^y \notin$ QUADTIME. As we have thus reduced HALT to QUADTIME, QUADTIME must be undecidable.

∎

7.2 Does more time mean more computational power?

Later in this lecture, we will discuss the time hierarchy theorem, which in broad terms states that for two reasonably different time bounds, there exists a machine computable within one of the bounds but not the other. While this may seem like an obvious result, we will now first state two surprising results about time bounds that will make us question this belief.

First, consider the following two sets of functions, as visualized in figure 7.2. The outer set consists of those functions computable in time an^2, while the inner consists of functions

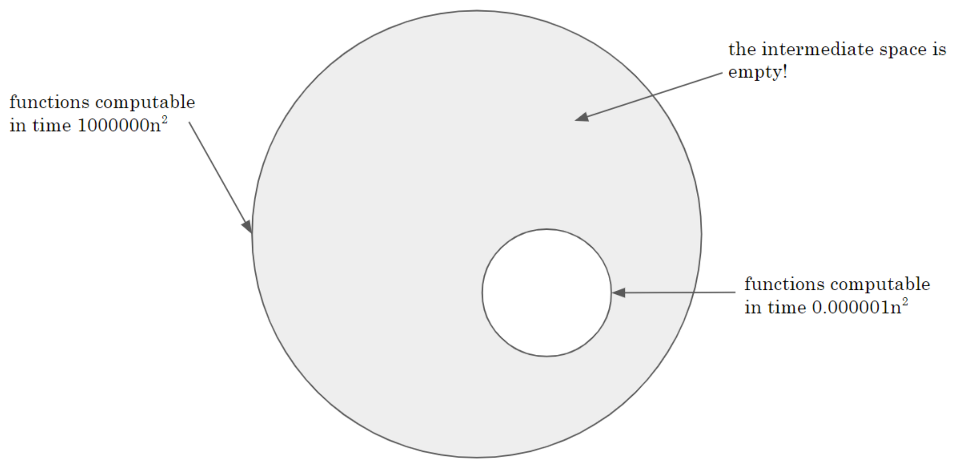

functions computable in time 1000000n^2
the intermediate space is empty!
functions computable in time 0.000001n^2

FIGURE 7.2: Visualization of the linear speedup theorem

computable in time bn^2, where $a > b$. Clearly, the latter is a subset of the former.

The shocking result is that even when $a \gg b$ (say, for example $a = 1000000$, while $b = (\frac{1}{10^{82}})^2$ - the square of the inverse of the upper bound on the number of atoms in the universe), the space in between is always empty. This is the linear speedup theorem, with formal statement given below. In essence, we can always get a multiplicative improvement in the time needed to compute a function, with small additional overhead linear in the length of the input.

THEOREM 7.6. *Linear Speedup Theorem*

If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$, it is computable in time $\epsilon \cdot T(n) + 2n$ for any $\epsilon > 0$.

Proof. Left as a challenge problem. □

Next, consider the following two sets of functions, illustrated in Figure 7.3. We first fix some function $T(n)$ such that T is monotonically increasing, $T(n) \geq n$ (grows at least linearly), and T is Turing computable - all reasonable constraints on a time bound. We then consider an inner set of functions (those computable in time $T(n)$) and an outer set (those computable in time $2^T(n)$. Even though there is an exponential gap between these two bounds, it turns out that there still exists a function $T(n)$ such that these two sets are identical. This theorem, known as the gap theorem, is formally stated below:

THEOREM 7.7. *Trakhtenbrot-Borodin gap theorem*

There is a Turing-computable function T which is monotonically increasing and for which $T(n) \geq n$ such that every function $f : \{0,1\}^* \to \{0,1\}^*$ computable in time $2^T(n)$ is also computable in time $T(n) + O_f(1)$.

We note that there is an additional constant factor, which depends on the function f we are trying to compute but not the length of the input. In addition, the condition

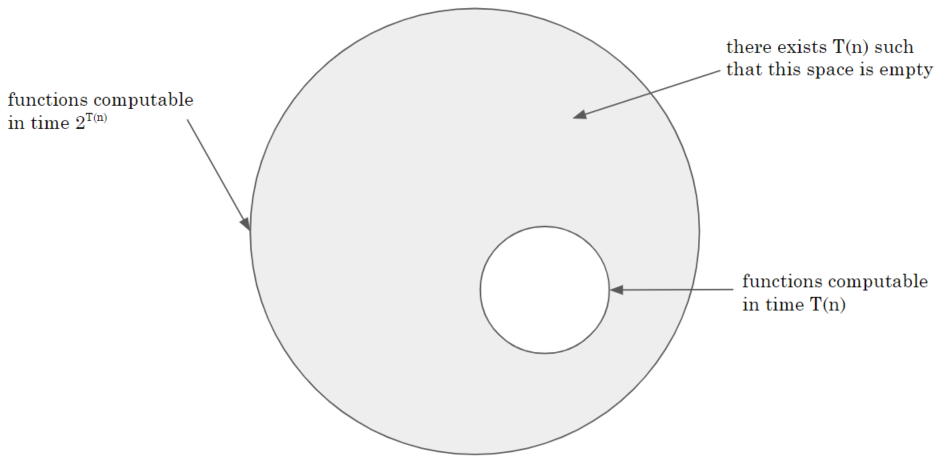

functions computable in time $2^{T(n)}$
there exists T(n) such that this space is empty
functions computable in time T(n)

FIGURE 7.3: Visualization of the gap theorem

$T(n) \geq n$ can be amended such that the right hand side holds a variety of functions; our following proof will not depend on this specific linear bound. To show this, we once again use a diagonalization argument, this time to construct a function T such that no TM M terminates in the range $(T, 2^T]$. The idea is that there are infinitely many disjoint intervals in which the lower bound and upper bound are exponentially far from each other; if we only look at finite subsets of Turing machines and inputs, there must exist at least one time interval in which none of these machine-input pairs terminate. By choosing these intervals, we can construct a suitable function.

Proof. Consider the following table:

	0	1	00	01	...	x
M_0	1	7	5	3		55
M_1	3	8	∞	10		101
$\vdots$						
M_α	310	834	∞	∞		∞

Like the previous table, our rows consist of different Turing machines, and our columns consist of inputs to those machines. However, rather than being a table of outputs, this is a table of running times: each cell indicates the amount of time the machine takes when running on that input. This can either be a finite number, indicating that the machine runs and halts after that time, or ∞ indicating that the machine never halts.

We let $\tau_k = \{\text{all runtimes of TMs } M_1, .., M_k \text{ on inputs of length } \leq k\}$. Note that in the table, these correspond to rectangles in the upper left corner. We also note that these τ_k are nested: each $\tau_k \subseteq \tau_l$ for $k \leq l$.

Consider the intervals $(2, 2^2], (2^2, 2^{2^2}], (2^{2^2}, 2^{2^{2^2}}], ...$ There are clearly infinitely many such intervals, all of which are disjoint. We define our function $T(k)$ by $T(k) = min\{t \geq 1 : \tau_k \cap (t, 2^t] = \emptyset\}$. As each τ_k is finite, and there exist infinitely many disjoint such intervals as stated, at least one such t must exist, and so this function is well-defined. This function is also monotonically increasing, as each τ_k is nested within larger ones as previously described. A suitable t for the interval must at least exclude those numbers contained in smaller τ_l, and so we can never decrease.

To show the final property, we have to demonstrate that our constructed T is Turing computable. To do this, we can simultaneously run $M_1, .., M_k$ on all inputs up to length $\leq k$. When a Turing machine halts at time m, we wait until the time 2^m. If no other machines halt in this interval, we have found a suitable interval in which no machine halts. Otherwise, we resume waiting at our new time m', until $2^{m'}$. As we have a finite number of simulations running, we must eventually find a suitable interval. Note that it does not matter that we may be simulating computation that doesn't halt, as those cases will never halt to restart our waiting, and we always have a finite upper bound to shoot for.

This function has our wanted properties. Consider a function f computable in time $2^{T(n)}$. Let M_k be a machine that computes f. We know that by construction, for inputs x of length $|x| \geq k$, the runtime of M_k is in the range $[0, 2^{T(|x|)}] \setminus (T(|x|), 2^{T(|x|)})$. It is included in the former set as we know f is computable in that time and excluded by the latter as T is constructed such that no such function will exist. We can ignore inputs of length $< k$ as they do not affect our analysis (for example, a machine could memorize the finitely many answers).

We have thus constructed a suitable $T(n)$, and so are done. $\square$

Both of these theorems show that there are cases where no computational gap exists when we would intuitively expect them to, and so should challenge our faith in them existing in general. In the next section, we will explain the necessary additional assumptions needed to prove such a gap exists.

7.3 The time hierarchy theorem

Suppose we have functions $t, T : \mathbb{N} \to \mathbb{N}$ such that $T \geq t$. We want to show what our intuition believes to be true: that there exists some function f such that f is computable in time $T(n)$ but not $t(n)$. However, there are obstacles in the form of the two results given in the previous section, both of which demonstrate some cases where this will not hold.

The linear speedup theorem suggests that any constant factor speedup is not sufficient; what is necessary is for one function to be asymptotically faster (that is, $\frac{T(n)}{t(n)} \to \infty$ in the limit). The gap theorem suggests that in addition to this, some other restriction is required.

The key for showing this gap turns out to be, in some way, "self-awareness": the algorithm must be able to clock itself, to check if it has exceeded some runtime. Otherwise, the machine operates in a space where time "does not exist." To this end, we provide the following definition.

Definition 7.8. $T : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if

- $T(n) \geq n$
- There exists some TM that maps 1..1 (a sequence of n 1s) to 1...1 (a sequence of $T(n)$ 1s) in time $O(T(n))$.

Example 7.9. Most "calculator functions" are time computable $(n, nlog(n), n^2, 2^n, ..)$.

Remark 7.10. It is actually harder to come up with a non time-constructible function; doing so is left as an instructive exercise.

This mapping Turing machine can be seen as an "alarm clock": it generates a "yardstick" of 1s for us, which we can use to keep track of how much time we have left. If we choose our functions t, T to have this property, we can then show the existence of a gap between what is computable in those times. Formally stated, this is the time hierarchy theorem given below.

Theorem 7.11. *Time hierarchy theorem*

Let $t, T : \mathbb{N} \to \mathbb{N}$ be time constructible functions such that $\frac{t(n)}{T(n)} \to 0$ in the limit. Then, $\mathrm{DTIME}(t(n)) \subsetneq \mathrm{DTIME}(T(n))$.

To show this, we will use our final diagonalization argument of the day. In this proof, we will once again look at the output table to try to construct a function that cannot be computed by any TM. However, we will now have the additional information of elapsed time, to construct a function that is computable in time $T(n)$ but not $t(n)$. The idea is that if an example computation computes in time $t(n)$, we always return the opposite, so that no TM can match it.

Proof. In this proof, we are once again diagonalizing the output table (example given below).

	0	1	00	01	...	x
M_0	0	1	1	-		1
M_1	-	0	-	1		0
$\vdots$						
M_α	1	-	1	0		0

We construct the following TM D.

TM 3 TM D for time hierarchy theorem proof

```
Input: $x$
1: for $T(|x|)$ time steps do
       Simulate $M_x$ on input $x$
2: end for
3: if $M_x$ has halted during the simulation then
       output the opposite answer as $M_x(x)$
4: else
       output 0
```

We use the notation $L(D)$ to denote the language decided by D. Clearly, $L(D) \in \mathrm{DTIME}(T(n))$. It remains to show that $L(D) \notin \mathrm{DTIME}(t(n))$.

Fix any TM M_α that runs in time $c \cdot t(n)$. From results from earlier lectures, we know that the time needed to simulate M_α on an input x is $\leq c_\alpha \cdot c \cdot t(|x|) log(c \cdot t(|x|)) < T(|x|)$ for all large enough $|x| \geq n_0$, by assumption. Therefore, the time needed to simulate M_α ($= M_{0^{n_0}\alpha}$, by our convention for TM encoding from a previous lecture) on the input $0^{n_0}\alpha$ is $< T(|0^{n_0}\alpha|)$. This means that by construction of our TM D, $D(0^{n_0}\alpha) \neq M_{0^{n_0}\alpha}(0^{n_0}\alpha) = M_\alpha(0^{n_0}\alpha)$. As thus $D \neq M_\alpha$ for all such M_α, we are done. $\square$

We also note this theorem's counterpart, the nondeterministic time hierarchy theorem.

Theorem 7.12. *Nondeterministic time hierarchy theorem*
Let $t, T : \mathbb{N} \to \mathbb{N}$ *be time constructible functions such that* $\frac{t(n+1)}{T(n)} \to 0$ *in the limit. Then,* $\mathrm{NTIME}(t(n)) \subsetneq \mathrm{NTIME}(T(n))$.

The proof of this theorem is too technical for this class, and so will not be discussed. However, we raise the question: why does the deterministic proof break? The answer is that diagonalization as we have been doing it is tricky in the nondeterministic context. Generally in our diagonalizers, we have been simulating a TM, then observing its result to try to do the opposite. However, in the nondeterministic context, a NTM which corresponds to an output of 1 could sometimes return 1, and sometimes 0. In this case, how do we know what to output?