Lecture 5: Complexity classes, Hierarchy Theorems

Anup Rao

April 9, 2024

Let us talk *complexity classes*. We are interested in classifying functions according to their complexity, so it makes sense to lump functions into sets of similar complexity:

Definition 1. *Define* $\mathsf{DTIME}(t(n))$ *to be the set of functions*

$$\mathsf{DTIME}(t(n)) = \{f : \{0,1\}^* \to \{0,1\} | f \text{ is computable in time } O(t(n))\}.$$

Similarly,

Definition 2. *Define* $\mathsf{DSPACE}(s(n))$ *to be the set*

$$\mathsf{DSPACE}(s(n)) = \{f : \{0,1\}^* \to \{0,1\} | f \text{ is computable in space } O(s(n))\}.$$

Once we have these definitions, we can try to define what it means for a function $f : \{0,1\}^* \to \{0,1\}$ to be *efficiently computable*. A reasonable definition of efficient computation should allow enough time to read all of the input, which takes $\Omega(n)$ time. So we should definitely include $\mathsf{DTIME}(n)$ in our set of efficiently computable functions. Further, if one algorithm calls another as a subroutine, and both are efficient, we would like to say that the combined algorithm is also efficient. The minimal class satisfying these assumptions is the class

Definition 3. $\mathbf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$.

Of course there is a whole spectrum of classes above P. For example:

Definition 4. $\mathbf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{n^c})$.

And,

Definition 5. $\mathbf{E} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{cn})$.

For space bounded computation, we need to have enough space to manipulate pointers into the inputs, which takes $\log n$ bits, before we get interesting classes. The first such class is:

Definition 6. $\mathbf{L} = \mathsf{DSPACE}(\log n)$.

Definition 7. $\mathbf{PSPACE} = \bigcup_{c \geq 1} \mathsf{DSPACE}(n^c)$.

Obviously if $t(n) = O(t'(n))$, then $\mathsf{DTIME}(t(n)) \subseteq \mathsf{DTIME}(t'(n))$. But is the containment strict? Does giving a Turing Machine more time actually allow it to compute things that it cannot compute without the extra time?

IN THE LAST LECTURE, we showed that there are natural functions that cannot be computed by Turing machines. To do this, we used the technique of diagonalization. In this lecture, we shall combine diagonalization with the universal simulation ability of Turing machines to show that Turing machines with more time/space are strictly more powerful than Turing machines with less time/space.

We are going to use diagonalization to show that Turing Machines that have more time can compute things that are not computable by Turing Machines with less time. Such a result is called a *hierarchy theorem*, it shoes that there is a hierarchy of power that comes with increasing computational resources. The basic idea is that a Turing Machine with more resources can simulate every machine that requires fewer resources and do the opposite of what it does on *some* input. To formally prove the hierarchy theorems, we need some more concepts:

Definition 8 (Time Constructible Functions)**.** *We say that the map $t : \mathbb{N} \to \mathbb{N}$ is* time constructible *if $t(n) \geq n$ and on input x there is a Turing Machine that computes $t(|x|)$ in time $O(t(|x|))$.*

Almost every running time or space bound you can think of like $n^5, 2^n, 2^{2^n}$ is time constructible and space constructible. (But not all functions are time constructible, since not all functions can be computed by turing machines). We shall also need a result about simulating turing machines by Turing Machines, that we discussed in the third lecture:

Theorem 9. *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 $T \geq 1$ 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 are now ready to prove our first hierarchy theorem:

Theorem 10 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n) \log r(n) = o(t(n))$, then $\mathsf{DTIME}(r(n)) \subsetneq \mathsf{DTIME}(t(n))$.*

Proof Recall that M_α denotes the Turing Machine whose code is α. The key idea is to use a function very similar to the one we defined in the last lecture for our diagonalization proofs:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) \text{ halts and outputs 0 after } t(|\alpha|) \text{ steps of the simulator,} \\ 0 & \text{else.} \end{cases}$$

We claim:

Claim 11. *f can be computed in time $O(t(n))$.*

To compute f, we first compute $t(|\alpha|)$ and then apply Theorem 9 to simulate $M_\alpha(\alpha)$ for $t(|\alpha|)$ steps of the simulator. So, f can be computed in time $O(t(n))$.

On the other hand, we shall show:

Claim 12. *f cannot be computed in time $O(r(n))$.*

If β is the code of a machine that computes f in time $c \cdot r(n)$. Let C_β be such that the execution of r steps of the machine M_β can be simulated in $C_\beta r \log r$ steps by the universal machine. Then there must be some binary string β' that represents the same machine as β, but is long enough so that

$$t(|\beta'|) > C_\beta \cdot c \cdot r(|\beta'|) \log r(|\beta'|)$$

This is because by assumption $r(n) \log r(n) = o(t(n))$, and so for large enough n,

$$t(n) > 2C_\beta \cdot c \cdot r(n) \log(r(n)) > C_\beta \cdot c \cdot r(n) \log(c \cdot r(n)).$$

Moreover, we can always add redundant lines to the code in β, until the code becomes long enough for $t(|\beta'|) > 2C_\beta \cdot c \cdot r(|\beta'|) \log r(|\beta'|)$.

If $M_\beta(\beta') = 0$, then $M_{\beta'}(\beta') = 0$ and so $f(\beta') = 1$ by the guarantee of Theorem 9. If $M_\beta(\beta') = 1$, $M_{\beta'}(\beta') = 1$, and so $f(\beta') = 0$, which proves that M_β does not compute f. ■