Lecture 19

Time-bounded computations

In the final couple of lectures of the course, we will discuss the topic of *computational complexity theory*, which is concerned with the inherent difficulty (or *hardness*) of computational problems and the effect of resource constraints on models of computation. We will only have time to scratch the surface; complexity theory is a rich subject, and many researchers around the world are engaged in a study of this field. Unlike formal language theory and computability theory, many of the central questions of complexity theory remain unanswered to this day.

In this lecture we will focus on the most important resource (from the view of computational complexity theory), which is *time*. The motivation is, in some sense, obvious: in order to be useful, computations generally need to be performed within a reasonable amount of time. In an extreme situation, if we have some computational task that we would like to perform, and someone gives us a computational device that will perform this computational task, but only after running for one million years or more, it is practically useless. One can also consider other resources besides time, such as space (or memory usage), communication in a distributed scenario, or a variety of more abstract notions concerning resource usage.

We will start with a definition of the running time of a DTM.

Definition 19.1. Let M be a DTM with input alphabet Σ. For each string $w \in \Sigma^*$, let $T(w)$ denote the number of steps (possibly infinite) for which M runs on input w. The *running time* of M is the function $t : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined as

$$t(n) = \max\{T(w) \,:\, w \in \Sigma^*,\ |w| = n\} \tag{19.1}$$

for every $n \in \mathbb{N}$. In words, $t(n)$ is the maximum number of steps required for M to halt, over all input strings of length n.

We will restrict our attention to DTMs whose running time is finite for all input lengths.

19.1 DTIME and time-constructible functions

Deterministic time complexity classes

For every function $f : \mathbb{N} \to \mathbb{N}$, we define a class of languages called $\mathrm{DTIME}(f)$, which represents those languages decidable in time $O(f(n))$.

Definition 19.2. Let $f : \mathbb{N} \to \mathbb{N}$ be a function. A language A is contained in the class $\mathrm{DTIME}(f)$ if there exists a DTM M that decides A and whose running time t satisfies $t(n) = O(f(n))$.

We define $\mathrm{DTIME}(f)$ in this way, using $O(f(n))$ rather than $f(n)$, because we are generally not interested in constant factors or in what might happen in finitely many special cases. One fact that motivates this choice is that it is usually possible to "speed up" a DTM by defining a new DTM, having a larger tape alphabet than the original, that succeeds in simulating multiple computation steps of the original DTM with each step it performs.

When it is reasonable to do so, we generally reserve the variable name n to refer to the input length for whatever language or DTM we are considering. So, for example, we may refer to a DTM that runs in time $O(n^2)$ or refer to the class of languages $\mathrm{DTIME}(n^2)$ with the understanding that we are speaking of the function $f(n) = n^2$, without explicitly saying that n is the input length.

We also sometimes refer to classes such as

$$\mathrm{DTIME}\big(n\sqrt{n}\big) \quad \text{or} \quad \mathrm{DTIME}\big(n^2 \log(n)\big), \tag{19.2}$$

where the function f that we are implicitly referring to appears to take non-integer values for some choices of n. This is done in an attempt to keep the expressions of these classes simple and intuitive, and you can interpret these things as referring to functions of the form $f : \mathbb{N} \to \mathbb{N}$ obtained by rounding up to the next nonnegative integer. For instance, $\mathrm{DTIME}(n^2 \log(n))$ means $\mathrm{DTIME}(f)$ for

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^2 \log(n) \rceil & \text{otherwise.} \end{cases} \tag{19.3}$$

Example 19.3. The DTM for the language $\mathrm{SAME} = \{0^m 1^m : m \in \mathbb{N}\}$ from Lecture 12 runs in time $O(n^2)$ on inputs of length n, and therefore SAME is contained in the class $\mathrm{DTIME}(n^2)$.

It is, in fact, possible to do better: it is the case that $\mathrm{SAME} \in \mathrm{DTIME}(n \log(n))$. To do this, one may define a DTM that repeatedly "crosses out" every other symbol on the tape, and compares the parity of the number of 0s and 1s crossed out after

each pass over the input. Through this method, SAME can be decided in time $O(n \log(n))$ by making just a logarithmic number of passes over the portion of the tape initially containing the input.

After considering the previous example, it is natural to ask if one can do even better than $O(n \log(n))$ for the running time of a DTM deciding the language SAME. The answer is that this is not possible. This is a consequence of the following theorem (which we will not prove).

Theorem 19.4. *Let A be a language. If there exists a DTM M that decides A in time $o(n \log(n))$, meaning that the running time t of M satisfies*

$$\lim_{n \to \infty} \frac{t(n)}{n \log(n)} = 0, \tag{19.4}$$

then A is regular.

It is, of course, critical that we understand the previous theorem to be referring to ordinary, one-tape DTMs. With a two-tape DTM, for instance, it is easy to decide some nonregular languages, including SAME, in time $O(n)$.

Time-constructible functions

The complexity class $\mathrm{DTIME}(f)$ has been defined for an arbitrary function of the form $f : \mathbb{N} \to \mathbb{N}$, but there is a sense in which most functions of this form are uninteresting from the viewpoint of computational complexity—because they have absolutely nothing to do with the running time of any DTM.

There are, in fact, some choices of functions $f : \mathbb{N} \to \mathbb{N}$ that are so strange that they lead to highly counter-intuitive results. For example, there exists a function f such that

$$\mathrm{DTIME}(f) = \mathrm{DTIME}(g), \qquad \text{for } g(n) = 2^{f(n)}; \tag{19.5}$$

even though g is exponentially larger than f, they both result in exactly the same deterministic time complexity class. This does not necessarily imply something important about time complexity, it is more a statement about the strangeness of the function f.

For this reason we define a collection of functions, called *time-constructible functions*, that represent well-behaved upper bounds on the possible running times of DTMs. Here is a precise definition.

Definition 19.5. Let $f : \mathbb{N} \to \mathbb{N}$ be a function satisfying $f(n) = \Omega(n \log(n))$. The function f is said to be *time constructible* if there exists a DTM M that operates as follows:

1. On each input 0^n the DTM M outputs $f(n)$ (written in binary notation), for every $n \in \mathbb{N}$.
2. M runs in time $O(f(n))$.

It might not be clear why we would define a class of functions in this particular way, but the essence is that these are functions that can serve as upper bounds for DTM computations. That is, a DTM can compute $f(n)$ on any input of length n, and doing this does not take more than $O(f(n))$ steps—and then it has the number $f(n)$ stored in binary notation so that it can then use this number to limit some subsequent part of its computation (perhaps the number of steps for which it runs during a second phase of its computation).

As it turns out, just about any reasonable function f with $f(n) = \Omega(n \log(n))$ that you are likely to care about as a bound on running time is time constructible. Examples include the following:

1. For any choice of an integer $k \geq 2$, the function $f(n) = n^k$ is time constructible.
2. For any choice of an integer $k \geq 2$, the function $f(n) = k^n$ is time constructible.
3. For any choice of an integer $k \geq 1$, the functions

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^k \log(n) \rceil & \text{otherwise} \end{cases} \tag{19.6}$$

   and

$$f(n) = \left\lceil n^k \sqrt{n} \right\rceil \tag{19.7}$$

   are time constructible.
4. If f and g are time-constructible functions, then the functions

$$h_1(n) = f(n) + g(n), \quad h_2(n) = f(n)g(n), \quad \text{and} \quad h_3(n) = f(g(n)) \tag{19.8}$$

   are also time constructible.

19.2 The time-hierarchy theorem

What we will do next is to discuss a fairly intuitive theorem concerning time complexity. A highly informal statement of the theorem is this: more languages can

The DTM K operates as follows on input $w \in \{0,1\}^*$:

1. If the input w does not take the form $w = \langle M \rangle 01^k$ for a DTM M with input alphabet $\{0,1\}$ and $k \in \mathbb{N}$, then reject.
2. Compute $t = f(|w|)$.
3. Simulate M on input w for t steps. If M has rejected w within t steps, then accept, otherwise reject.

Figure 19.1: This DTM decides a language that cannot be decided in time $o(f(n))$.

be decided with more time. While this is indeed an intuitive idea, it is not obvious how a formal version of this statement is to be proved. We will begin with a somewhat high-level discussion of how the theorem is proved, and then state the strongest-known form of the theorem (without going through the low-level details needed to obtain the stronger form).

Suppose that a time-constructible function f has been selected, and define a DTM K as described in Figure 19.1. It is not immediately apparent what the running time is for K, because this depends on precisely how the simulation of M in step 3 is done; different ways of performing the simulation could of course lead to different running times. For the time being, let us take g to be the running time of K, and we will worry later about how specifically g relates to f.

Next, let us think about the language $\mathrm{L}(K)$ decided by K. This is a language over the binary alphabet, and it is obvious that $\mathrm{L}(K) \in \mathrm{DTIME}(g)$, because K itself is a DTM that decides $\mathrm{L}(K)$ in time $g(n)$. What we will show is that $\mathrm{L}(K)$ cannot possibly be decided by a DTM that runs in time $o(f(n))$.

To this end, assume toward contradiction that there does exist a DTM M that decides $\mathrm{L}(K)$ in time $o(f(n))$. Because the running time of M is $o(f(n))$, we know that there must exist a natural number n_0 such that, for all $n \geq n_0$, the DTM M halts on all inputs of length n in strictly fewer than $f(n)$ steps. Choose k to be large enough so that the string $w = \langle M \rangle 01^k$ satisfies $|w| \geq n_0$, and (as always) let $n = |w|$. Because M halts on input w after fewer than $f(n)$ steps, we find that

$$w \in \mathrm{L}(K) \;\Leftrightarrow\; w \notin \mathrm{L}(M). \tag{19.9}$$

The reason is that K simulates M on input w, it completes the simulation because M runs for fewer than $f(n)$ step, and it answers *opposite* to the way M answers (i.e., if M accepts, then K rejects; and if M rejects, then K accepts). This contradicts the

assumption that M decides $\mathrm{L}(K)$. We conclude that no DTM whose running time is $o(f(n))$ can decide $\mathrm{L}(K)$.

It is natural to wonder what the purpose is for taking the input to K to have the form $\langle M \rangle 01^k$, as opposed to just $\langle M \rangle$ (for instance). The reason is pretty simple: it is just a way of letting the length of the input string grow, so that the asymptotic behavior of the function f and the running time of M take over (even though we are really interested in fixed choices of M). If we were to change the language, so that the input takes the form $w = \langle M \rangle$ rather than $\langle M \rangle 01^k$, we would have no way to guarantee that K is capable of finishing the simulation of M on input $\langle M \rangle$ within $f(|\langle M \rangle|)$ steps—for it could be that the running time of M on input $\langle M \rangle$ exceeds $f(|\langle M \rangle|)$ steps, even though the running time of M is small compared with f for significantly longer input strings.

What we have proved is that, for any choice of a time-constructible function $f : \mathbb{N} \to \mathbb{N}$, the proper subset relation

$$\mathrm{DTIME}(h) \subsetneq \mathrm{DTIME}(g) \tag{19.10}$$

holds whenever $h(n) = o(f(n))$, where g is the running time of K (which depends somehow on f).

Remark 19.6. There is an aspect of the argument just presented that is worth noting. We obtained the language $\mathrm{L}(K)$, which is contained in $\mathrm{DTIME}(g)$ but not $\mathrm{DTIME}(h)$ assuming $h(n) = o(f(n))$, not by actually describing the language explicitly, but by simply describing the DTM K that decides it. Indeed, in this case it is hard to imagine a description of the language $\mathrm{L}(K)$ that would be significantly more concise than the description of K itself. This technique can be useful in other situations. Sometimes, when you wish to prove the existence of a language having a certain property, rather than explicitly defining the language, it is possible to define a DTM M that operates in a particular way, and then take the language you are looking for to be $\mathrm{L}(M)$.

If you work very hard to make K run as efficiently as possible, the following theorem can be obtained.

Theorem 19.7 (Time-hierarchy theorem)**.** *If $f, g : \mathbb{N} \to \mathbb{N}$ are time-constructible functions for which $f(n) = o(g(n)/\log(g(n)))$, then*

$$\mathrm{DTIME}(f) \subsetneq \mathrm{DTIME}(g). \tag{19.11}$$

The main reason that we will not go through the details required to prove this theorem is that optimizing K to simulate a given DTM as efficiently as possible gets very technical. For the sake of this course, it is enough that you understand

the basic idea of the proof. In particular, notice that it is another example of a proof that uses the diagonalization technique; while it is a bit more technical, it has a very similar flavor to the proof that DIAG is not semidecidable, and to the proof that $\mathcal{P}(\mathbb{N})$ is uncountable.

From the time-hierarchy theorem, one can conclude the following down-to-earth corollary.

Corollary 19.8. *For all $k \geq 1$, it is the case that* $\mathrm{DTIME}(n^k) \subsetneq \mathrm{DTIME}(n^{k+1})$.

19.3 Polynomial and exponential time

We will finish off the lecture by introducing a few important notions based on deterministic time complexity. First, let us define two complexity classes, known as P and EXP, as follows:

$$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k) \quad \text{and} \quad \mathrm{EXP} = \bigcup_{k \geq 1} \mathrm{DTIME}\big(2^{n^k}\big). \tag{19.12}$$

In words, a language A is contained in the complexity class P if there exists a DTM M that decides A and has *polynomial running time*, meaning a running time that is $O(n^k)$ for some fixed choice of $k \geq 1$; and a language A is contained in the complexity class EXP if there exists a DTM M that decides A and has *exponential running time*, meaning a running time that is $O\big(2^{n^k}\big)$ for some fixed choice of $k \geq 1$.

As a very rough but nevertheless useful simplification, we often view the class P as representing languages that can be *efficiently* decided by a DTM, while EXP contains languages that are decidable by a *brute force* approach. These are undoubtedly over-simplifications in some respects, but for languages that correspond to "natural" computational problems that arise in practical settings, this is a reasonable picture to keep in mind.

By the time-hierarchy theorem, we can conclude that $\mathrm{P} \subsetneq \mathrm{EXP}$. In particular, if we take $f(n) = 2^n$ and $g(n) = 2^{2n}$, then the time hierarchy theorem establishes the middle (proper) inclusion in this expression:

$$\mathrm{P} \subseteq \mathrm{DTIME}(2^n) \subsetneq \mathrm{DTIME}(2^{2n}) \subseteq \mathrm{EXP}. \tag{19.13}$$

There are many examples of languages contained in the class P. If we restrict our attention to languages we have discussed thus far in the course, we may say the following.

- The languages $\mathrm{A}_{\mathrm{DFA}}$, $\mathrm{E}_{\mathrm{DFA}}$, $\mathrm{EQ}_{\mathrm{DFA}}$, and $\mathrm{E}_{\mathrm{CFG}}$ from Lecture 15 are all certainly contained in P; if you analyzed the running times of the DTMs we described for those languages, you would find that they run in polynomial time.

- The languages A_{NFA}, A_{REX}, E_{NFA}, and E_{REX} are also in P, but the DTMs we described for these languages in Lecture 15 do not actually show this. Those DTMs have exponential running time, because the conversion of an NFA or regular expression to a DFA could result in an exponentially large DFA. It is, however, not too hard to decide these languages in polynomial time through different methods.

  In particular, we can decide A_{NFA} in polynomial time through a more direct simulation in which we keep track of the set of all states that a given NFA could be in when reading a given input string, and we can decide A_{REX} by performing a polynomial-time conversion of a given regular expression into an equivalent NFA, effectively reducing the problem in polynomial time to A_{NFA}. The language E_{NFA} can be decided in polynomial time by treating it as a graph reachability problem, and E_{REX} can be reduced to E_{NFA} in polynomial time.

- The language A_{CFG} is also contained in P, but once again, the DTM for this language that we discussed in Lecture 15 does not establish this. A more sophisticated approach based on the algorithmic technique of *dynamic programming* does, however, allow one to decide A_{CFG} in polynomial time. This fact allows one to conclude that every context-free language is contained in P.

There does not currently exist a proof that the languages EQ_{NFA} and EQ_{REX} fall outside of the class P, but this is conjectured to be the case. This is because these languages are complete for the class PSPACE of languages that are decidable within a polynomial amount of space. If EQ_{NFA} and EQ_{REX} are in P, then it would then follow that P = PSPACE, which seems highly unlikely. It is the case, however, that EQ_{NFA} and EQ_{REX} are contained in EXP, for in exponential time one can afford to perform a conversion of NFAs or regular expressions to DFAs and then test the equivalence of the two (possibly exponential size) DFAs in the same way that we considered earlier.

Finally, let us observe that one may consider not only languages that are decided by DTMs having bounded running times, but also functions that can be computed by time-bounded DTMs. For example, the class of *polynomial-time computable functions*, which are functions that can be computed by a DTM with running time $O(n^k)$ for some fixed positive integer k, are critically important in theoretical computer science, and algorithms courses typically discuss many practically important examples of such functions.[1]

[1] Algorithms courses usually consider computational models that represent machines having random access memory, as opposed to Turing machines. However, because a Turing machine can simulate such a model with no more than a polynomial slowdown, the class of polynomial-time computable functions is the same for the two types of models.