Notes on Complexity Theory: Fall 2005 Last updated: September, 2005

Lecture 1

Jonathan Katz

1 Review

We began class with a quick review of some basic concepts that students have (presumably) already seen in an earlier course. These included:

- Turing machines. Deciding/recognizing languages.
- Time/space complexity of a Turing machine. (We also pointed out the subtlety in defining sub-linear space bounds for Turing machines.)
- We mentioned the speed-up theorem, mainly to justify ignoring constant factors.
- The classes $\text{TIME}(f(n))$, $\text{SPACE}(f(n))$, and $\mathcal{P}$.

2 Preliminaries

We first give some definitions of "well-behaved" functions. Note that various (subtly different) definitions appear in the literature, but the following will be our working definitions throughout the course.

Definition 1 A function $f : \mathbb{N} \to \mathbb{N}$ is *space constructible* if $f(n)$ is non-decreasing and there exists a Turing machine which on input 1^n outputs the binary representation of $f(n)$ without ever exceeding $O(f(n))$ space. Note that if f is space constructible, then there exists a Turing machine which on input 1^n marks off exactly $f(n)$ tape cells on its work-tape (say, using a special symbol) without ever exceeding $O(f(n))$ space. $\diamondsuit$

Definition 2 A function $f : \mathbb{N} \to \mathbb{N}$ (with $f(n) \geq n$ for all n) is *time constructible* if it is non-decreasing and there exists a Turing machine running in time $O(f(n))$ which on input 1^n outputs the binary representation of $f(n)$. Note that if f is time constructible, then there exists a Turing machine which on input 1^n runs for $O(f(n))$ steps and then halts. $\diamondsuit$

Most "natural" functions one encounters (unless one is specifically interested in counterexamples) are space and time constructible.

The following is a simple but useful fact:

Fact 1 *Let M be a Turing machine using space $s(n)$ on inputs of length n. The number of configurations $\mathcal{C}_M(n)$ of M on any fixed input of length n is bounded by:*

$$\mathcal{C}_M(n) \leq |Q_M| \cdot n \cdot s(n) \cdot |\Sigma_M|^{s(n)}, \tag{1}$$

where Q_M are the states of M and Σ_M is the alphabet of M. In particular, when $s(n) = \Omega(\log n)$ we have $\mathcal{C}_M(n) = 2^{O(s(n))}$.

Proof The first term in Eq. (1) comes from the number of states, the second from the possible positions of the input head, the third from the possible positions of the work-tape head, and the last from the possible values stored on the work tape. (Note that since the input is fixed and the input tape is read-only, we do not need to consider all possible length-n strings that can be written on the input tape.) ■

Two important facts follow from the above:

Lemma 2 *If the computation of $M(w)$ exceeds $\mathcal{C}_M(|w|)$ steps, then $M(w)$ will run forever.*

Proof This follows from the simple observation that if a configuration is ever repeated, then M goes into an infinite loop. ■

Lemma 3 *Let $s(n) \geq \log n$ be space constructible. If there exists a machine M which* recognizes *a language L using space s, there exists a machine M' which* decides *L using space $O(s)$ (i.e., M' always halts).*

Proof We construct M' which simulates M but only for at most $\mathcal{C}_M(n)$ steps (where n is the input length), using a counter to keep track of how many steps have been taken so far. By the previous lemma, M' decides the same language that M recognizes. The space used by M' on any input w of length $|w| = n$ is exactly the space used by M on that input, plus the additional space used to store the counter. We already know that M uses space $O(s(n))$. Since $s(n) \geq \log n$, the counter requires space $\log \mathcal{C}_M(n) = O(s(n))$ (by Fact 1) and so the total space used by M' is $O(s(n))$. ■

As a consequence, without loss of generality we may consider space-bounded Turing machines which always halt (except when we briefly discuss sub-logarithmic space classes).

Corollary 4 *If $s(n) = \Omega(\log n)$ is space constructible, then* $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$.

Proof Let $L \in \text{SPACE}(s(n))$ and say M accepts L and uses space $O(s(n))$. By the previous lemma, we may assume M halts on all inputs. Since M would go into an infinite loop if it ever repeats a configuration, the maximum running time of M on inputs of length n is $\mathcal{C}_M(n) = 2^{O(s(n))}$. ■

For completeness, we mention the following "counterpart" to the above:

Lemma 5 *For any $s(n)$ we have* $\text{TIME}(s(n)) \subseteq \text{SPACE}(s(n))$.

Proof This follows from the trivial observation that a machine cannot write on more than one work-tape cell per move. ■

The above can be improved to $\text{TIME}(s(n)) \subseteq \text{SPACE}(s(n)/\log s(n))$; we may see a proof of this in a later class.

3 Hierarchy Theorems

We focus first on the case of space bounds, since the result is easier to prove. We remark that we do not prove the tightest possible statement; the reader interested in further discussion is referred to [3, Chap. 12].

Theorem 6 *Let G be space constructible, and $g(n) = \Omega(\log n)$. If $g(n) = o(G(n))$, then there exists a language L such that $L \in \text{SPACE}(G(n))$ but $L \notin \text{SPACE}(g(n))$. Thus, $\text{SPACE}(g(n))$ is a proper subset of $\text{SPACE}(G(n))$.*

Proof We define L by giving a Turing machine M_L (using space $O(G(n))$) that decides it. The tricky part will be to show that no Turing machine running in space $O(g(n))$ decides it. M_L does the following on input w of length $|w| = n$:

1. If w is not of the form $\langle M \rangle 10^*$ for some Turing machine M (w.r.t. some fixed encoding), reject.

2. Compute $G(n)$, and mark off this much space *twice* on the work tape. The first $G(n)$ cells will be called the *workspace*, and the second $G(n)$ cells will be called the *counterspace*.

3. Run $M(w)$ for at most $2^{G(n)}$ steps using the workspace to perform the simulation of M, and using the counterspace to keep track (in binary) of the number of steps taken in the simulation of M. If the allotted time is exceeded (i.e., the counter reaches the value $2^{G(n)}$), or if the computation of $M(w)$ tries to go beyond the allotted workspace, reject. Otherwise, if $M(w)$ accepts, reject, and if $M(w)$ rejects, accept.

By construction, M_L uses total space $O(G(n))$ (since G is space constructible).

We now need to show that no machine using space $O(g(n))$ can decide L. Assume the contrary. Then there exists a machine M deciding L and running in space $\tilde{g}(n) = O(g(n))$. Without loss of generality, we may assume that M uses only a single work-tape and a binary tape alphabet (this is actually not very hard to see, but the reader is referred to [3, Chap. 12] for details). Let c represent the space "overhead" needed (by M_L) to simulate $M(w)$; i.e., this is the space used to simulate the computation of $M(w)$ minus the space used by M itself on input w. (This overhead can be used to keep track of what state the simulated copy of M is in, for example.) A key point is that c depends on M but *not* on w. (You should convince yourself that this is true; note that we allow M_L to use a non-binary tape alphabet and multiple tapes, if needed. So the first step of M_L might be to copy $\langle M \rangle$ to a work-tape, and then the only additional space needed is to store the current state of the simulated copy of M.)

Now, choose n_0 large enough so that for $n \geq n_0$ we have $\tilde{g}(n) + c < G(n)$ and $\mathcal{C}_M(n) < 2^{G(n)}$; such an n_0 exists because $\tilde{g}(n) = o(G(n))$ and $\tilde{g}(n) = \Omega(\log n)$. When we run M_L on input $\tilde{w} \stackrel{\text{def}}{=} \langle M \rangle 10^{n_0}$, the machine M_L allots enough space and time to run $M(\tilde{w})$ to completion. But now M_L outputs the opposite of whatever M outputs, and so M_L and M cannot decide the same language. ■

The analogous result for the case of time complexity classes is proved in a similar manner, but the details are more difficult and there is a slight complication:

Theorem 7 *Let G be time constructible, and $g(n) \geq n$ for all n. If $g(n) \log g(n) = o(G(n))$, then there exists a language L such that $L \in \text{TIME}(G(n))$ but $L \notin \text{TIME}(g(n))$.*

Proof The high-level structure of the proof is the same as in the proof of the previous theorem. We define L by giving a Turing machine M_L (using time $O(G(n))$) that decides it. M_L does the following on input w of length $|w| = n$:

1. If w is not of the form $\langle M \rangle 10^*$ for some Turing machine M, reject.

2. We use five tapes. The first four tapes are used to simulate an execution of M on input w (one tape is used to keep a copy of M, one tape records the state of the simulated copy of M, and the other two tapes will be used as the workspace for the simulated copy of M). The last tape is used to run, in parallel, a machine that runs for exactly $G(n)$ steps. If this second machine halts and the simulation of M is not done, reject.

3. Otherwise, if the simulated copy of M finishes then: if $M(w)$ accepts, reject and if $M(w)$ rejects, accept.

By construction, M_L runs in time $O(G(n)) + O(n) = O(G(n))$.

We now want to show that no machine M running in time $O(g(n))$ can decide L. A subtlety is that such a machine M might use more than two work-tapes, but M_L has a *fixed* number of work-tapes (which happens to be five in the above description). However, it is known that if there exists a machine deciding a language L in time $O(g(n))$ (using any number of tapes), then there exists a *two-tape* machine deciding L in time $\tilde{g}(n) = O(g(n) \log g(n))$; see [3, Chap. 12]. (We may also assume without loss of generality that this machine uses a binary tape alphabet.) Let M' be such a machine.

Again, there will be some (time) overhead in the simulation of M' by M_L. This overhead is now a multiplicative one: to simulate a step of M', the machine M_L might have to scan through all of M' (and perform multiple scans over the current state of the simulated copy of M'). As before, however, this overhead depends on M' but is independent of w. In summary, there exist constants c_1, c_2 such that an execution of M' that runs in time $t(n)$ can be simulated in time $c_1 t(n) + c_2$.

Now, choose n_0 large enough so that for $n \geq n_0$ we have $c_1 \tilde{g}(n) \log \tilde{g}(n) + c_2 < G(n)$; such an n_0 exists by the assumption of the theorem. As before, when we run M_L on input $\tilde{q} \stackrel{\text{def}}{=} \langle M' \rangle 10^{n_0}$, machine M_L allots enough time to run $M'(\tilde{w})$ to completion. But now M_L outputs the opposite of whatever M' outputs, and so M_L and M' cannot decide the same language. ■

It is unknown whether the above is optimal. Similarly, it is unknown whether a better simulation of k-tape Turing machines by two-tape Turing machines is possible.

4 Introduction to $\mathcal{NP}$

Recall the definition of the class $\mathcal{P}$: a language L is in $\mathcal{P}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x)$ runs in time $p(|x|)$, and (2) $x \in L$ iff $M_L(x) = 1$ (where we view an output of "1" as acceptance and an output of "0" as rejection).

The typical way of defining $\mathcal{NP}$ is by introducing the notion of *non-deterministic* Turing machines. An alternate, arguably more intuitive, definition is given by the following: a language L is in $\mathcal{NP}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time $p(|x|)$, and (2) $x \in L$ iff there exists a w such that $M_L(x, w) = 1$. We remark that in condition (1) it is essential that the running time of M_L be measured in terms of the length of x only (an alternate approach is to require the length of w to be at most $p(|x|)$ in condition (2)). For an $\mathcal{NP}$-language L, we will sometimes speak of the relation R_L defined by: $R_L(x, w) = 1$ iff $M_L(x, w) = 1$. This gives the following, alternate way of thinking about $\mathcal{NP}$: namely, a language L is in $\mathcal{NP}$ if there exists a language $R_L \in \mathcal{P}$ and a polynomial p such that[1]

$$x \in L \Leftrightarrow \exists w : (x, w) \in R_L \text{ and } |w| \leq p(|x|).$$

It is a good exercise to show that the "traditional" definition of $\mathcal{NP}$ (i.e., in terms of non-determinism) is equivalent to the definition(s) above.

For the rest of this lecture, we let R denote a relation which is *polynomially bounded* (namely, there exists a polynomial p such that if $(x, w) \in R$ then $|w| \leq p(|x|)$) as well as *polynomially-verifiable* (i.e., $R \in \mathcal{P}$). Note that any such R defines the $\mathcal{NP}$-language L_R given by:

$$L_R \stackrel{\text{def}}{=} \{x \mid \exists w : (x, w) \in R\}$$

(this is the "dual" of the conversion from an $\mathcal{NP}$-language L to a relation R_L described earlier, although we remark that there are multiple relations R which define the same language L_R).

5 $\mathcal{NP}$ Completeness

We first discuss the notion of *reductions*. We first define a Cook(-Turing) reduction:

Definition 3 A *Cook(-Turing) reduction* from a language L to a language L' is a polynomial-time oracle machine M such that, if M' is any machine that decides L', then $M^{M'}$ decides L. We express the above by writing $L \leq_T^P L'$. ♢

Another important, yet immediate, result is that (1) if there is a Cook reduction from L to L' and (2) $L' \in \mathcal{P}$, then $L \in \mathcal{P}$ as well. Note, however, that this is not believed to be the case for languages in $\mathcal{NP}$. For example, every co$\mathcal{NP}$ language is Cook-reducible to an $\mathcal{NP}$ language, but it is not believed that co$\mathcal{NP} \subseteq \mathcal{NP}$.

A more restricted notion of a reduction is given next:

Definition 4 A *Karp reduction* (also called a *many-to-one reduction*) from a language L to a language L' is a polynomial-time computable function f such that $x \in L$ iff $f(x) \in L'$. We express this by writing $L \leq_m^P L'$. ♢

Note that any Karp reduction provides an immediate Cook reduction as well. However, here it is true that if there is a Karp reduction from L to L' and $L' \in \mathcal{NP}$, then $L \in \mathcal{NP}$.

Finally, we also define the following useful notion:

[1] We remark that we now need the length restriction on w because the machine M_{R_L} that decides R_L is allowed, by definition, to run in time polynomial in $|x| + |w|$ on input (x, w).

Definition 5 A *Levin reduction* from relation R_1 to relation R_2 is a triple of polynomial-time computable functions f, g, h such that:

– $(x, y) \in R_1 \Rightarrow (f(x), g(x, y)) \in R_2$

– $(f(x), z) \in R_2 \Rightarrow (x, h(x, z)) \in R_1$

The above, in particular, imply that $x \in L_{R_1} \Leftrightarrow f(x) \in L_{R_2}$. ♢

It may be verified that all the above reductions are transitive.

5.1 Defining $\mathcal{NP}$ Completeness

With the above in place, we define $\mathcal{NP}$-hardness and $\mathcal{NP}$-completeness:

Definition 6 A language L is $\mathcal{NP}$-hard if for every language $L' \in \mathcal{NP}$, there is a Karp reduction from L' to L. A language L is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and also $L \in \mathcal{NP}$. ♢

We remark that one could also define $\mathcal{NP}$-hardness via *Cook* reductions. However, this seems to lead to a different definition. In particular, oracle access to any co$\mathcal{NP}$-complete language is enough to decide $\mathcal{NP}$, meaning that any co$\mathcal{NP}$-complete language is $\mathcal{NP}$-hard w.r.t. Cook reductions. On the other hand, if a co$\mathcal{NP}$-complete language were $\mathcal{NP}$-hard w.r.t. Karp reductions, this would imply $\mathcal{NP} = \text{co}\mathcal{NP}$ (which is considered to be unlikely). (Further discussion of this issue can be found in [2, Chapter 7.1].)

We show the "obvious" $\mathcal{NP}$-complete language:

Claim 8 *Define language L via:*

$$L = \left\{ \langle M, x, 1^t \rangle \mid \begin{array}{c} M \text{ is a non-deterministic T.M.} \\ \text{which accepts } x \text{ within } t \text{ steps} \end{array} \right\}.$$

Then L is $\mathcal{NP}$-complete.

Proof It is not hard to see that $L \in \mathcal{NP}$. Given $\langle M, x, 1^t \rangle$ as input, non-deterministically choose a legal sequence of up to t moves of M on input x, and accept if M accepts. This algorithm runs in non-deterministic polynomial time and decides L.

To see that L is $\mathcal{NP}$-hard, let $L' \in \mathcal{NP}$ be arbitrary and assume that non-deterministic machine $M'_{L'}$ decides L' and runs in time n^c on inputs of size n. Define function f as follows: given x, output $\langle M'_{L'}, x, 1^{|x|^c} \rangle$. Note that (1) f can be computed in polynomial time and (2) $x \in L' \Leftrightarrow f(x) \in L$. We remark that this can be extended to give a Levin reduction (between R_L and $R_{L'}$, defined in the natural ways). ■

We remark that one can "adapt" the above language to give a language which is $\mathcal{NP}$-hard but not in $\mathcal{NP}$. Specifically, consider the language

$$L = \left\{ \langle M, x \rangle \mid \begin{array}{c} M \text{ is a non-deterministic T.M.} \\ \text{which accepts } x \end{array} \right\}.$$

Bibliographic Notes

The proofs of the space and time hierarchy theorems are adapted from [4, 3]. See also [1, Lecture 4] for more details regarding these proofs (and some discussion of how to efficiently simulate the computation of another Turing machine). Section 5 is based on [1, Lecture 2].

References

[1] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

[2] S. Homer and A.L. Selman. *Computability and Complexity Theory*. Springer, 2001.

[3] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[4] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.