Notes on Complexity Theory Last updated: December, 2011

Lecture 2

Jonathan Katz

1 Review

The *running time* of a Turing machine M on input x is the number of "steps" M takes before it halts. Machine M is said to run in time $T(\cdot)$ if for every input x the running time of $M(x)$ is at most $T(|x|)$. (In particular, this means it halts on all inputs.) The *space* used by M on input x is the number of cells written to by M on all its *work* tapes[1] (a cell that is written to multiple times is only counted once); M is said to use space $T(\cdot)$ if for every input x the space used during the computation of $M(x)$ is at most $T(|x|)$. We remark that these time and space measures are *worst-case* notions; i.e., even if M runs in time $T(n)$ for only a fraction of the inputs of length n (and uses less time for all other inputs of length n), the running time of M is still T. (Average-case notions of complexity have also been considered, but are somewhat more difficult to reason about. We may cover this later in the semester; or see [1, Chap. 18].)

Recall that a Turing machine M *computes* a function $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all x. We will focus most of our attention on boolean functions, a context in which it is more convenient to phrase computation in terms of *languages*. A language is simply a subset of $\{0,1\}^*$. There is a natural correspondence between languages and boolean functions: for any boolean function f we may define the corresponding language $L = \{x \mid f(x) = 1\}$. Conversely, for any language L we can define the boolean function f by $f(x) = 1$ iff $x \in L$. A Turing machine M *decides a language* L if

$$\begin{aligned} x \in L &\quad\Rightarrow\quad M(x) = 1 \\ x \notin L &\quad\Rightarrow\quad M(x) = 0 \end{aligned}$$

(we sometimes also say that M *accepts* L, though we will try to be careful); this is the same as computing the boolean function f that corresponds to L. Note in particular that we require M to halt on all inputs.

What is complexity theory about? The fundamental question of complexity theory is to understand the inherent complexity of various languages/problems/functions; i.e., what is the most efficient algorithm (Turing machine) deciding some language? A convenient terminology for discussing this is given by introducing the notion of a *class*, which is simply a set of languages. Two basic classes are:

- $\textsc{time}(f(n))$ is the set of languages decidable in time $O(f(n))$. (Formally, $L \in \textsc{time}(f(n))$ if there is a Turing machine M and a constant c such that (1) M decides L, and (2) M runs in time $c \cdot f$; i.e., for all x (of length at least 1) $M(x)$ halts in at most $c \cdot f(|x|)$ steps.)
- $\textsc{space}(f(n))$ is the set of languages that can be decided using space $O(f(n))$.

[1]Note that we do not count the space used on the input or output tapes; this allows us to meaningfully speak of sub-linear space machines (with linear- or superlinear-length output).

Note that we ignore constant factors in the above definitions. This is convenient, and lets us ignore low-level details about the model of computation.[2]

Given some language L, then, we may be interested in determining the "smallest" f for which $L \in \textsc{time}(f(n))$. Or, perhaps we want to show that $\textsc{space}(f(n))$ is strictly larger than $\textsc{space}(f'(n))$ for some functions f, f'; that is, that there is some language in the former that is not in the latter. Alternately, we may show that one class contains another. As an example, we start with the following easy result:

Lemma 1 *For any $f(n)$ we have $\textsc{time}(f(n)) \subseteq \textsc{space}(f(n))$.*

Proof This follows from the observation that a machine cannot write on more than a constant number of cells per move. ■

2 $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-Completeness

2.1 The Class $\mathcal{P}$

We now introduce one of the most important classes, which we equate (roughly) with *problems that can be solved efficiently.* This is the class $\mathcal{P}$, which stands for *polynomial time*:

$$\mathcal{P} \stackrel{\text{def}}{=} \bigcup_{c \geq 1} \textsc{time}(n^c).$$

That is, a language L is in $\mathcal{P}$ if there exists a Turing machine M_L and a polynomial p such that $M_L(x)$ runs in time $p(|x|)$, and M_L decides L.

Does $\mathcal{P}$ really capture efficient computation? There are debates both ways:

- For many problems nowadays that operate on extremely large inputs (think of Google's search algorithms), only linear-time are really desirable. (In fact, one might even want *sub*linear-time algorithms, which are only possible by relaxing the notion of correctness.) This is related to the (less extreme) complaint that an n^{100} algorithm is not really "efficient" in any sense.

  The usual response here is that n^{100}-time algorithms rarely occur. Moreover, when algorithms with high running times (e.g., n^8) *do* get designed, they tend to be quickly improved to be more efficient.

- From the other side, one might object that $\mathcal{P}$ does not capture all efficiently solvable problems. In particular, a *randomized* polynomial-time algorithm (that is correct with high probability) seems to also offer an efficient way of solving a problem. Most people today would agree with this objection, and would classify problems solvable by randomized polynomial-time algorithms as "efficiently solvable". Nevertheless, it may turn out that such problems all lie in $\mathcal{P}$ anyway; this is currently an unresolved conjecture. (We will discuss the power of randomization, and the possibility of derandomization, later in the semester.)

  As mentioned previously, *quantum* polynomial-time algorithms may also be considered "efficient". It is fair to say that until general-purpose quantum computers are implemented, this is still debatable.

[2]This decision is also motivated by "speedup theorems" which state that if a language can be decided in time (resp., space) $f(n)$ then it can be decided in time (resp., space) $f(n)/c$ for any constant c. (This assumes that $f(n)$ is a "reasonable" function, but the details need not concern us here.)

Another important feature of $\mathcal{P}$ is that it is closed under composition. That is, if an algorithm A (that otherwise runs in polynomial time) makes polynomially many calls to an algorithm B, and if B runs in polynomial time, then A runs in polynomial time. See [1] for further discussion.

2.2 The Classes $\mathcal{NP}$ and co$\mathcal{NP}$

Another important class of problems are those whose solutions can be *verified* efficiently. This is the class $\mathcal{NP}$. (Note: $\mathcal{NP}$ does *not* stand for "non-polynomial time". Rather, it stands for "non-deterministic polynomial-time" for reasons that will become clear later.) Formally, $L \in \mathcal{NP}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time[3] $p(|x|)$, and (2) $x \in L$ iff there exists a w such that $M_L(x, w) = 1$; such a w is called a *witness* (or, sometimes, a *proof*) that $x \in L$. Compare this to the definition of $\mathcal{P}$: a language $L \in \mathcal{P}$ 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 $M_L(x) = 1$.

Stated informally, a language L is in $\mathcal{P}$ if membership in L can be decided efficiently. A language L is in $\mathcal{NP}$ if membership in L can be efficiently verified (given a correct proof). A classic example is given by the following language:

$$\text{IndSet} = \left\{ (G, k) \ : \ \begin{array}{c} G \text{ is a graph that has} \\ \text{an } \textit{independent set} \text{ of size } k \end{array} \right\}.$$

We do not know an efficient algorithm for determining the size of the largest independent set in an arbitrary graph; hence we do not have any efficient algorithm deciding IndSet. However, if we know (e.g., through brute force, or because we constructed G with this property) that an independent set of size k exists in some graph G, it is easy to prove that $(G, k) \in$ IndSet by simply listing the nodes in the independent set: verification just involves checking that every pair of nodes in the given set is *not* connected by an edge in G, which is easy to do in polynomial time. Note further than if G does *not* have an independent set of size k then there is no proof that could convince us otherwise (assuming we are using the stated verification algorithm).

It is also useful to keep in mind an analogy with mathematical statements and proofs (though the correspondence is not rigorously accurate). In this view, $\mathcal{P}$ would correspond to the set of mathematical statements (e.g., "1+1=2") whose truth can be easily determined. $\mathcal{NP}$, on the other hand, would correspond to the set of (true) mathematical statements that have "short" proofs (whether or not such proofs are easy to find).

We have the following simple result, which is the best known as far as relating $\mathcal{NP}$ to the time complexity classes we have introduced thus far:

Theorem 2 $\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$.

Proof The containment $\mathcal{P} \subseteq \mathcal{NP}$ is trivial. As for the second containment, say $L \in \mathcal{NP}$. Then 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 $M_L(x, w) = 1$. Since $M_L(x, w)$ runs in time $p(|x|)$, it can read at most the first $p(|x|)$ bits of w and so we may assume that w in condition (2) has length at most $p(|x|)$. The following is then a deterministic algorithm for deciding L:

[3] It is essential that the running time of M_L be measured in terms of the length of x alone. An alternate approach is to require the length of w to be at most $p(|x|)$ in condition (2).

On input x, run $M_L(x, w)$ for all strings $w \in \{0,1\}^{\leq p(|x|)}$. If any of these results in $M_L(x, w) = 1$ then output 1; else output 0.

The algorithm clearly decides L. Its running time on input x is $O\left(p(|x|) \cdot 2^{p(|x|)}\right)$, and therefore $L \in \text{TIME}\left(2^{n^c}\right)$ for some constant c. ■

The "classical" definition of $\mathcal{NP}$ is in terms of non-deterministic Turing machines. Briefly, the model here is the same as that of the Turing machines we defined earlier, except that now there are *two* transition functions δ_0, δ_1, and at each step we imagine that the machine makes an arbitrary ("non-deterministic") choice between using δ_0 or δ_1. (Thus, after n steps the machine can be in up to 2^n possible configurations.) Machine M is said to output 1 on input x if there exists *at least one* sequence of choices that would lead to output 1 on that input. (We continue to write $M(x) = 1$ in this case, though we stress again that $M(x) = 1$ when M is a non-deterministic machine just means that $M(x)$ outputs 1 for *some* set of non-deterministic choices.) M decides L if $x \in L \Leftrightarrow M(x) = 1$. A non-deterministic machine M runs in time $T(n)$ if for every input x and every sequence of choices it makes, it halts in time at most $T(|x|)$. The class $\text{NTIME}(f(n))$ is then defined in the natural way: $L \in \text{NTIME}(f(n))$ if there is a non-deterministic Turing machine M_L such that $M_L(x)$ runs in time $O(f(|x|))$, and M_L decides L. Non-deterministic space complexity is defined similarly: non-deterministic machine M uses space $T(n)$ if for every input x and every sequence of choices it makes, it halts after writing on at most $T(|x|)$ cells of its work tapes. The class $\text{NSPACE}(f(n))$ is then the set of languages L for which there exists a non-deterministic Turing machine M_L such that $M_L(x)$ uses space $O(f(|x|))$, and M_L decides L.

The above leads to an equivalent definition of $\mathcal{NP}$ paralleling the definition of $\mathcal{P}$:

Claim 3 *$\mathcal{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$.*

This is a good exercise; a proof can be found in [1].

The major open question of complexity theory is whether $\mathcal{P} \overset{?}{=} \mathcal{NP}$; in fact, this is one of the outstanding questions in mathematics today. The general belief is that $\mathcal{P} \neq \mathcal{NP}$, since it seems quite "obvious" that non-determinism is stronger than determinism (i.e., verifying should be easier than solving, in general), and there would be many surprising consequences if $\mathcal{P}$ were equal to $\mathcal{NP}$. (See [1] for a discussion.) But we have had no real progress toward proving this belief.

Conjecture 4 *$\mathcal{P} \neq \mathcal{NP}$.*

A (possibly feasible) open question is to prove that non-determinism is even *somewhat* stronger than determinism. It is known that $\text{NTIME}(n)$ is strictly stronger than $\text{TIME}(n)$ (see [2, 3, 4] and references therein), but we do not know, e.g., whether $\text{TIME}(n^3) \subseteq \text{NTIME}(n^2)$.

2.2.1 The Class co$\mathcal{NP}$

For any class $\mathcal{C}$, we define the class co$\mathcal{C}$ as co$\mathcal{C} \overset{\text{def}}{=} \left\{L \mid \bar{L} \in \mathcal{C}\right\}$, where $\bar{L} \overset{\text{def}}{=} \{0,1\}^* \setminus L$ is the complement of L. Applied to the class $\mathcal{NP}$, we get the class co$\mathcal{NP}$ of languages where *non*-membership can be efficiently verified. In other words, $L \in$ co$\mathcal{NP}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time[4] $p(|x|)$, and (2) $x \in L$ iff for all w we have $M_L(x, w) = 1$. Note why this (only) implies efficiently verifiable proofs of *non*-membership:

[4]See footnote 3.

a single w where $M_L(x,w)=0$ is enough to convince someone that $x \notin L$, but a single w where $M_L(x,w)=1$ means nothing.

A $\mathsf{co}\mathcal{NP}$ language is easily obtained by taking the complement of any language in $\mathcal{NP}$. So, for example, the complement of IndSet is the language

$$\text{NoIndSet} = \left\{ (G,k) \; : \; \begin{array}{c} G \text{ does } \textit{not} \text{ have} \\ \text{an independent set of size } k \end{array} \right\}.$$

Let us double-check that this is in $\mathsf{co}\mathcal{NP}$: we can prove that $(G,k) \notin \text{NoIndSet}$ by giving a set of k vertices that *do* form an independent set in G (this assumes the obvious verification algorithm); note that (assuming we use the obvious verification algorithm) we can never be "fooled" into believing that (G,k) is not in NoIndSet when it actually is.

As another example of languages in $\mathcal{NP}$ and $\mathsf{co}\mathcal{NP}$, consider the *satisfiability problem* which asks whether a boolean formula in conjunctive normal form is satisfiable (see [1] for a formal definition if you have not encountered these terms before). That is,

$$\mathsf{SAT} = \{\phi \mid \phi \text{ has a satisfying assignment}\}.$$

Then $\overline{\mathsf{SAT}}$ consists of boolean formulae with *no* satisfying assignment. We have $\mathsf{SAT} \in \mathcal{NP}$ and $\overline{\mathsf{SAT}} \in \mathsf{co}\mathcal{NP}$. As another example, consider the language TAUT of tautologies:

$$\mathsf{TAUT} = \{\phi : \phi \text{ is satisfied by every assignment}\}.$$

TAUT is also in $\mathsf{co}\mathcal{NP}$.

The class $\mathsf{co}\mathcal{NP}$ can also be defined in terms of non-deterministic Turing machines. This is left as an exercise.

Note that $\mathcal{P} \subseteq \mathcal{NP} \cap \mathsf{co}\mathcal{NP}$. (Why?) Could it be that $\mathcal{NP} = \mathsf{co}\mathcal{NP}$? Once again, we don't know the answer but it would be surprising if this were the case. In particular, there does not seem to be any way to give an efficiently verifiable proof that, e.g., a boolean formula does *not* have any satisfying assignment (which is what would be implied by $\overline{\mathsf{SAT}} \in \mathcal{NP}$).

Conjecture 5 $\mathcal{NP} \neq \mathsf{co}\mathcal{NP}$.

2.3 $\mathcal{NP}$-Completeness

2.3.1 Defining $\mathcal{NP}$-Completeness

What does it mean for one language L' to be harder[5] to decide than another language L? There are many possible answers to this question, but one way to start is by capturing the intuition that if L' is harder than L, then an algorithm for deciding L' should be useful for deciding L. We can formalize this idea using the concept of a *reduction*. Various types of reductions can be defined; we start with one of the most central:

Definition 1 *A language L is* Karp reducible *(or* many-to-one reducible*) to a language L' if there exists a polynomial-time computable function f such that $x \in L$ iff $f(x) \in L'$. We express this by writing $L \leq_p L'$.*

[5] Technically speaking, I mean "at least as hard as".

The existence of a Karp reduction from L to L' gives us exactly what we were looking for. Say there is a polynomial-time Turing machine (i.e., algorithm) M' deciding L'. Then we get a polynomial-time algorithm M deciding L by setting $M(x) \stackrel{\text{def}}{=} M'(f(x))$. (Verify that M does, indeed, run in polynomial time.) This explains the choice of notation $L \leq_p L'$. We state some basic properties, all of which are straightforward to prove.

Claim 6 *We have:*

1. *(Transitivity) If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.*
2. *If $A \leq_p B$ and $B \in \mathcal{P}$ then $A \in \mathcal{P}$.*
3. *If $A \leq_p B$ and $B \in \mathcal{NP}$ then $A \in \mathcal{NP}$.*

A problem is $\mathcal{NP}$-hard if it is "at least as hard to solve" as any problem in $\mathcal{NP}$. It is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and also in $\mathcal{NP}$. Formally:

Definition 2 *Language L' is $\mathcal{NP}$-hard if for every $L \in \mathcal{NP}$ it holds that $L \leq_p L'$. Language L' is $\mathcal{NP}$-complete if $L' \in \mathcal{NP}$ and L' is $\mathcal{NP}$-hard.*

Note that if L is $\mathcal{NP}$-hard and $L \leq_p L'$, then L' is $\mathcal{NP}$-hard as well.

co$\mathcal{NP}$-completeness is defined analogously: a language L' is co$\mathcal{NP}$-hard if for every $L \in \text{co}\mathcal{NP}$ it holds that $L \leq_p L'$; language L' is co$\mathcal{NP}$-complete if L' is co$\mathcal{NP}$-hard and $L' \in \text{co}\mathcal{NP}$.

2.3.2 Existence of $\mathcal{NP}$-Complete Problems

A priori, it is not clear that there should be any $\mathcal{NP}$-complete problems. One of the surprising results from the early 1970s is that $\mathcal{NP}$-complete problems exist. Soon after, it was shown that many important problems are, in fact, $\mathcal{NP}$-complete. Somewhat amazingly, we now know thousands of $\mathcal{NP}$-complete problems arising from various disciplines.

Here is a trivial $\mathcal{NP}$-complete language:

$$L = \left\{(M, x, 1^t) : \exists w \in \{0,1\}^t \text{ s.t. } M(x,w) \text{ halts within } t \text{ steps with output } 1.\right\}.$$

Next time we will show more natural $\mathcal{NP}$-complete languages

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[2] R. Kannan. Towards separating nondeterminisn from determinisn. *Math. Systems Theory* 17(1): 29–45, 1984.

[3] W. Paul, N. Pippenger, E. Szemeredi, and W. Trotter. On determinism versus non-determinisn and related problems. FOCS 1983.

[4] R. Santhanam. On separators, segregators, and time versus space. *IEEE Conf. Computational Complexity* 2001.