Notes on Complexity Theory Last updated: September, 2015

Lecture 1

Notes taken from Jonathan Katz, lightly edited by Dov Gordon

1 Turing Machines

I assume that most students have encountered Turing machines before. (Students who have not may want to look at Sipser's book [3].) A Turing machine is defined by an integer $k \geq 1$, a finite set of states Q, an alphabet Γ, and a transition function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ where:

- k is the number of (infinite, one-dimensional) tapes used by the machine. In the general case we have $k \geq 3$ and the first tape is a read-only *input tape*, the last is a write-once *output tape*, and the remaining $k-2$ tapes are *work tapes*. For Turing machines with boolean output (which is what we will mostly be concerned with in this course), an output tape is unnecessary since the output can be encoded into the final state of the Turing machine when it halts.
- Q is assumed to contain a designated start state q_s and a designated halt state q_h. (In the case where there is no output tape, there are two halting states $q_{h,0}$ and $q_{h,1}$.)
- We assume that Γ contains $\{0, 1\}$, a "blank symbol", and a "start symbol".
- There are several possible conventions for what happens when a head on some tape tries to move left when it is already in the left-most position, and we are agnostic on this point. (Anyway, by our convention, below, that the left-most cell of each tape is "marked" there is really no reason for this to ever occur...).

The computation of a Turing machine M on input $x \in \{0, 1\}^*$ proceeds as follows: All tapes of the Turing machine contain the start symbol followed by blank symbols, with the exception of the input tape which contains the start symbol followed by x (and then the remainder of the input tape is filled with blank symbols). The machine starts in state $q = q_s$ with its k heads at the left-most position of each tape. Then, until q is a halt state, repeat the following:

1. Let the current contents of the cells being scanned by the k heads be $\gamma_1, \ldots, \gamma_k \in \Gamma$.
2. Compute $\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma'_2, \ldots, \gamma'_k, D_1, \ldots, D_k)$ where $q' \in Q$ and $\gamma'_2, \ldots, \gamma'_k \in \Gamma$ and $D_i \in \{L, S, R\}$.
3. Overwrite the contents of the currently scanned cell on tape i to γ'_i for $2 \leq i \leq k$; move head i to the left, to the same position, or to the right depending on whether $D_i = L, S$, or R, respectively; and then set the current state to $q = q'$.

The output of M on input x, denoted $M(x)$, is the binary string contained on the output tape (between the initial start symbol and the trailing blank symbols) when the machine halts. (When there is no output tape, then the output is '1' if M halts in state $q_{h,1}$ and the output is '0' is M halts in state $q_{h,0}$.) It is also possible that M never halts when run on some input x. We return to this point later.

The *running time* of a Turing machine M on input x is simply the number of "steps" M takes before it halts; that is, the number of iterations (equivalently, the number of times δ is computed) in the above loop. Machine M is said to run in time $T(\cdot)$ if for every input x the running time of $M(x)$ is at most $T(|x|)$. 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 said to be T. (Average-case notions of complexity have also been considered, but are somewhat more difficult to reason about.)

A Turing machine M *computes a function* $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all x. Assuming f is a total function, and so is defined on all inputs, this in particular means that M halts on all inputs. 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 as the set $L = \{x \mid f(x) = 1\}$. Conversely, for any language L we can define the boolean function f so that $f(x) = 1$ iff $x \in L$. A Turing machine M *decides a language* L if

$$x \in L \Leftrightarrow M(x) = 1$$

(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.

1.1 Comments on the Model

Turing machines are *not* meant as a model of modern computer systems. Rather, they were introduced (before computers were even built!) as a mathematical model of *what computation is*. Explicitly, the axiom is that "any function that can be computed in the physical world, can be computed by a Turing machine"; this is the so-called *Church-Turing thesis*. (The thesis cannot be proved unless one can formally define what it means to "compute a function in the physical world" without reference to Turing machines. In fact, several alternate notions of computation have been defined and shown to be equivalent to computation by a Turing machine; there are no serious candidates for alternate notions of computation that are *not* equivalent to computation by a Turing machine. See [1] for further discussion.) In fact, an even stronger axiom known as the *strong Church-Turing thesis* is sometimes assumed to hold: this says that "any function that can be computed in the physical world, can be computed with at most a polynomial reduction in efficiency by a Turing machine". This thesis is challenged by notions of *randomized* computation that we will discuss later. In the past 15 years or so, however, this axiom has been called into question by results on *quantum computing* that show polynomial-time algorithms in a quantum model of computation for problems not known to have polynomial-time algorithms in the classical setting.

There are several variant definitions of Turing machines that are often considered; none of these contradict the strong Church-Turing thesis. (That is, any function that can be computed on any of these variant Turing machines, including the variant defined earlier, can be computed on any other

[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).

variant with at most a polynomial increase in time/space.) Without being exhaustive, we list some examples (see [1, 2] for more):

- One may fix Γ to *only* include $\{0, 1\}$ and a blank symbol.
- One may restrict the tape heads to only moving left or right, not staying in place.
- One may fix $k = 3$, so that there is only one work tape. In fact, one may even consider $k = 1$ so that there is only a single tape that serves as input tape, work tape, and output tape.
- One can allow the tapes to be infinite in both directions, or two-dimensional.
- One can allow random access to the work tapes (so that the contents of the ith cell of some tape can be read in one step). This gives a model of computation that fairly closely matches real-world computer systems, at least at an algorithmic level.

The upshot of all of this is that it does not matter much which model one uses, as long as one is ok with losing polynomial factors. On the other hand, if one is concerned about "low level" time/space complexities then it is important to fix the exact model of computation under discussion. For example, the problem of deciding whether an input string is a palindrome can be solved in time $O(n)$ on a two-tape Turing machine, but requires time $\Omega(n^2)$ on a one-tape Turing machine.

1.2 An example

We take an example directly out of Sipser's book [3]. The turing machine M_L accepts the language $L = \{0^{2^n} | n \geq 0\}$. M_L uses the alphabet $\Gamma = \{\sqcup, 0, x\}$. Recall that the transition function maps from $Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$. Make sure you understand how to map this formalism onto the notation in the Figure below.

Intuitively, what this machine does is: a) it checks if there are no "0"s on the input tape (q_1) and

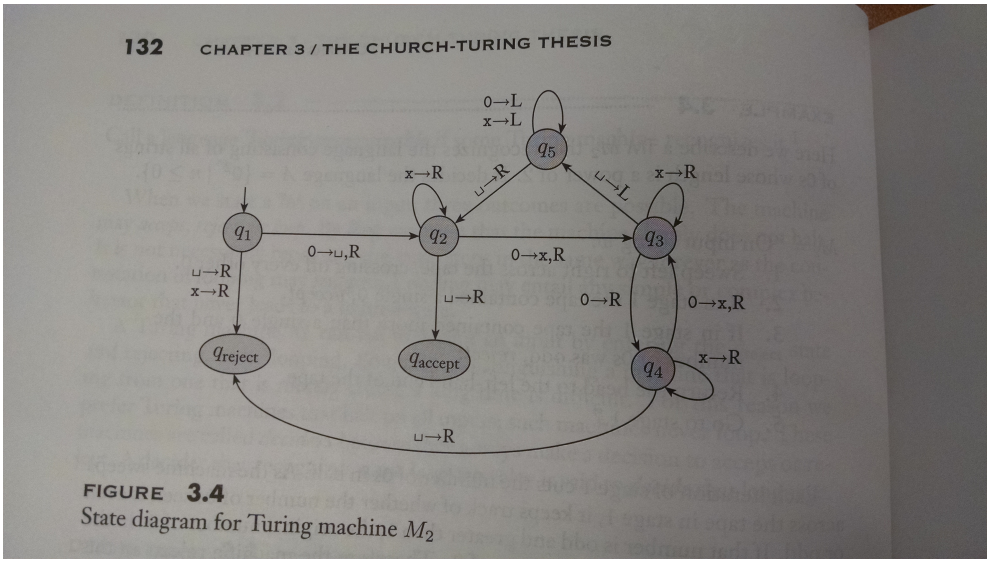

Figure 1: M_L accepting language $L = \{0^{2^n} | n \geq 0\}$

rejects if this is the case. Otherwise, it scrolls right until it finds the second "0" on the tape (q_2) and crosses this off by replacing it with an x. If it cannot find a second "0", then there is exactly one "0" remaining on the tape, and it accepts (also q_2). After that, it crosses off every other "0" (q_3 and q_4). If it reaches the end (i.e. by finding a $\sqcup$) when there are an odd number of "0"s, it rejects (q_4), otherwise, it rewinds to the first "0" and begins again (q_5). Note: there is one confusing thing in this state diagram! When q_1 replaces the first "0" with a $\sqcup$, this is to help the machine find the beginning of the tape during the rewinding. But you should think of this first "0" as still remaining, and *not* as thought it has been crossed off.

1.3 Universal Turing Machines and Uncomputable Functions

An important observation (one that is, perhaps, obvious nowadays but was revolutionary in its time) is that *Turing machines can be represented by binary strings.* In other words, we can view a "program" (i.e., a Turing machine) equally well as "data", and run one Turing machine on (a description of) another. As a powerful example, a *universal* Turing machine is one that can be used to simulate any other Turing machine. We define this next.

Fix some representation of Turing machines by binary strings, and assume for simplicity that every binary string represents some Turing machine (this is easy to achieve by mapping badly formed strings to some fixed Turing machine). Consider the function $f(M, x) = M(x)$. Is f computable? Perhaps surprisingly, f is computable. We stress that here we require there to be a *fixed* Turing machine U, with a fixed number of tapes and a fixed alphabet (not to mention a fixed set of states) that can simulate the behavior of an *arbitrary* Turing machine M that may use any number of tapes and any size alphabet. A Turing machine computing f is called a *universal* Turing machine.

Note that the function $f(M, x) = M(x)$ is a partial function, since in this context the given Turing machine M may not halt on the given input x and we leave f undefined in that case. To simplify things, we will consider the total function

$$f'(M, x, 1^t) = \begin{cases} 1 & \text{if } M(x) \text{ halts within } t \text{ steps with output } 1 \\ 0 & \text{otherwise} \end{cases},$$

whose computability is closely linked to that of f.

Theorem 1 *There exists a Turing machine U such that (1) $U(M, x, 1^t) = M(x)$ if M halts on x within t steps, and 0 otherwise. Furthermore, (2) for every M there exists a constant c such that the following holds: for all x, if $M(x)$ halts within t steps, then $U(M, x, 1^t)$ halts within $c \cdot t \log t$ steps.*

Proof We only give an informal overview here, based on what appears in Arora and Barak [1]. See their text for more details. We also will only sketch how to achieve a bound on the running time of $c \cdot t^2$ steps, though the stronger statement can be proven. U will have an input tape and output tape, as well as 5 work tapes, and the alphabet Γ that only includes $\{0, 1\}$, a blank symbol, and a start symbol. Let $\gamma = |\Gamma_M|$ be the size of M's alphabet, $q = |Q|$ be the size of M's state space, and δ be its transition function, which we assume is written as a full function table. Let k be the number of tapes used by M. Note that if U had $k+2$ tapes, alphabet Γ_M, and were allowed more than q states, the problem would be simple: U could store a description of M on one of the extra tapes, and simply scan the description at each step to see how M transitions. (The other

extra tape would be used for any computation needed during the scan of M.) Instead, U will have five tapes containing the following (in Binary):

- Tape 1: A description of M's transition function.
- Tape 2: Current state of M.
- Tape 3: A counter that starts at t and is decremented until 0.
- Tape 4: Contents of M's working tapes.
- Tape 5: An additional work tape.

U then proceeds through the following steps:

1. Read through M's work tapes (on Tape 4) and pull out the k symbols currently pointed to by M. Write those on the work tape 5.
2. Scan through M's transition function on Tape 1, and determine how to update the state (Tape 2), based on the k values stored on Tape 5.
3. Update M's work tapes as specified by the transition function (on Tape 4).
4. Update the input and output tapes as M specified by the transition function.
5. Decrease the counter on tape 3 by 1, check if the value is 0, and halt if it is.

One important subtlety needs to be addressed. When combining M's work tapes onto a single tape, there is no way to know much space each tape requires! This is handled by *interleaving* the contents of all those tapes. It is not hard to see that $U(M, x) = M(x)$ for any x for which $M(x)$ halts within t steps. As for the claim about the running time, let's consider a single simulation step: tapes 1, 2 and 5 will have only a constant number of bits on them. (This constant depends on q, γ and k, and thus is specific to M, but not at all on $|x|$.) Tape 3 requires $\log t$, and decreasing it in each step requires the same. Finally, tape 4 can never have more than $\gamma \cdot k \cdot t$ bits, so traversing it requires $O(t)$ steps on each simulation step. Each step uses $O(t + \log t)$, so the full computation requires $O(t^2)$ steps. ■

Another natural possibility is to consider the (total) function

$$f_{halt}(M, x) = \begin{cases} 1 & \text{if } M(x) \text{ halts with output } 1 \\ 0 & \text{otherwise} \end{cases};$$

What about f_{halt}? Is it computable? By again viewing Turing machines as data, we can show that this function is *not* computable!

Theorem 2 *The function f_{halt} is not computable.*

Proof Say there is some Turing machine M_{halt} computing f_{halt}. Then we can define the following machine M^*:

> On input (a description of) a Turing machine M, output $M_{halt}(M, M)$. If the result is 1, output 0; otherwise output 1.

What happens when we run M^* on *itself*? Consider the possibilities for the result $M^*(M^*)$:

- Say $M^*(M^*) = 1$. This implies that $M_{halt}(M^*, M^*) = 0$. But that means that $M^*(M^*)$ does not halt with output 1, a contradiction.
- Say $M^*(M^*) = 0$. This implies that $M_{halt}(M^*, M^*) = 1$. But that means that $M^*(M^*)$ halts with output 1, a contradiction.
- It is not possible for $M^*(M^*)$ to never halt, since $M_{halt}(M^*, M^*)$ is a total function (and so is supposed to halt on all inputs).

We have reached a contradiction in all cases, implying that M_{halt} as described cannot exist. ■

Remark: The fact that f_{halt} is not computable does *not* mean that the halting problem cannot be solved "in practice". In fact, checking termination of programs is done all the time in industry. Of course, they are not using algorithms that are solving the halting problem – this would be impossible! Rather, they use programs that may give *false negative*, i.e., that may claim that some other program does not halt when it actually does. The reason this tends to work in practice is that the programs that people want to reason about in practice tend to have a form that makes them amenable to analysis.

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:

- $\text{TIME}(f(n))$ is the set of languages decidable in time $O(f(n))$. (Formally, $L \in \text{TIME}(f(n))$ 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.)
- $\text{SPACE}(f(n))$ is the set of languages that can be decided using space $O(f(n))$.

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 $L \in \text{TIME}(f(n))$. Or, perhaps we want to show that $\text{SPACE}(f(n))$ is strictly larger than $\text{SPACE}(f'(n))$ 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 3 *For any $f(n)$ we have $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$.*

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

[2]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.)

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*:

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

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.

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 Class $\mathcal{NP}$

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,

Definition 1 *$L \in \mathcal{NP}$ 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 $M_L(x, w) = 1$.*

[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).

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 4 $\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$.

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:

> 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 5 $\mathcal{NP} = \bigcup_{c \geq 1} \text{NTIME}(n^c)$.

The major open question of complexity theory is whether $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$; 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 6 $\mathcal{P} \neq \mathcal{NP}$.

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 [4, 5, 6] and references therein), but we do not know, e.g., whether $\text{TIME}(n^3) \subseteq \text{NTIME}(n^2)$.

2.3 $\mathcal{NP}$-Completeness

2.3.1 Defining $\mathcal{NP}$-Completeness

What does it mean for one language L' to be harder[4] 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 2 *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 $L \leq_p L'$.*

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 will return to defining $\mathcal{NP}$-completeness in a later lecture.

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

References

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

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

[3] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.

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

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

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