Math/CMSC 38800, Complexity Theory Notes

Yakov Shalunov

Contents

Headings

Theorems

Definitions

Lecture 1

Week 1, Tuesday (2025-03-25)

History

1.1 Introduction

Historical lecture, many people date complexity theory back to one particular paper, but we'll cover some older things first. Will cover Time Hierarchy Theorem at the end.

Discrete computations can be represented as functions $f : \{0,1\}^* \to \{0,1\}^*$ taking finite binary strings to finite binary strings. You can encode arbitrary structures using binary strings.

Remark (Alphabets). We are choosing the alphabet $\Sigma = \{0,1\}$, rather than working in an arbitrary alphabet. In computability theory, it is typical to use $\Sigma = \{1\}$ and, e.g., represent $n \in \mathbb{N}$ in the unary encoding 1^n.

In computability theory, this makes no difference, but in *complexity* theory, this makes an enormous difference, since for $n \in \mathbb{N}$, the unary representation has length x and the binary representation has length approximately $\log_2 x$.

The exponential difference is enormous, covering the span of most complexity theory.

Any other fixed finite alphabet Σ, e.g., $\{0,1,...,9\}$, this will not change much because then the length of x is about $\log_{10} x$, which is a constant factor difference from $\log_2 x$, which is immaterial to complexity theory.

Remark (Constant factors). We use "big-O" notation, where $f = O(g)$ if $\exists C \in \mathbb{R}_+, N \in \mathbb{N}$ such that $\forall n \geq N,\ f(n) \leq C \cdot g(n)$. We generally do not care about these constant factor differences and will generally make these constants implicit via big-O rather than explicit.

Similarly, $f = \Omega(g)$ if $\exists C \in \mathbb{R}_+, N \in \mathbb{N}$ such that $\forall n \geq N,\ f(n) \geq C \cdot g(n)$.

Finally, $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Remark (Machine independence). Due to ignoring constant factors, most results are machine/language and technology independent.

We are interested in lower-bounds and upper-bounds on how hard problems are. Upper-bounds are "easily" proven by providing examples solving the problem, whereas lower-bounds require cleverer techniques.

Mostly for historical reasons, we use the model of "Turing machines," though it is functionally equivalent for our purposes to, e.g., Python.

Definition 1.1 (Turing machine): A Turing machine consists of type finite alphabets, Σ (the "symbol alphabet") and Q the "internal alphabet" or "set of states." We presume that there is are a distinguished "blank" symbol $s_0 \in \Sigma$, a "start state" $q_1 \in Q$, and a "halt state" $q_h \in Q$.

While Turing machines can be defined many ways, we will assume that there is a single, infinite tape on which the machine can write symbols. All but finitely many symbols of the tape will always be the "blank" symbol s_0.

The machine has a "head" located at some position on the tape and is in some internal state $q \in Q$. The Turing machine then has a program or "transition function," $T : \Sigma \times Q \to Q \times (\Sigma \cup \{L, R\})$.

The transition function governs how the machine evolves from one time step to the next. The machine reads the symbol $s \in \Sigma$ under the head, and then combined with the current internal state $q \in Q$, it outputs a new state and either a new symbol $s' \in \Sigma$ or one of L and R, representing left and right movements of the type.

Our standard alphabet is $\Sigma = \{0, 1, s_0\}$. To run the machine on an input $x \in \{0,1\}^*$, the tape is initialized such that it contains x surrounded by infinite cells containing s_0 and the head is placed on the first character of x.

The machine then evolves according to T until it reaches q_h, at which point the output is the contents of the tape with all s_0 removed. Note that a Turing machine may never reach q_h.

Every Turing machine M computes a partial function $M : \{0,1\}^* \to \{0,1\}^*$, where $M(x)$ is undefined if M on input x never reaches q_h.

Definition 1.2 (Halting problem): The problem $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ is defined to be 1 on a description of a Turing machine M and input x to that machine if M halts on x, and 0 otherwise.

Theorem 1.1 (Halting problem): *There is no Turing machine computing* HALT.

Proof. Suppose H computes HALT. Let

$$H'(M) = \begin{cases} \text{output } 0 & \text{if } H(M,M) = 0 \\ \text{loop forever} & \text{if } H(M,M) = 1 \end{cases}$$

Observe then that if $H'(H')$ halts, $H(H', H') = 1$, so $H'(H')$ loops forever. Conversely, if $H'(H')$ loops forever, $H(H', H') = 0$, so $H'(H') = 0$ halts. Thus, we have a contradiction and so can conclude H cannot exist. □

However, for our purposes, we are interested in Turing machines running in bounded time. We can consider our machines to keep a running clock and once the clock elapses, they halt regardless of other state of the computation.

We are in particular interested in decision problems:

Definition 1.3 (Decision problems and languages): A computational problem $f : \{0,1\}^* \to \{0,1\}^*$ is a decision problem if the image of f is $\{0,1\}$.

The corresponding language $L \subseteq \{0,1\}^*$ is $f^{-1}(1)$.

We will often identify decision problems and the corresponding languages.

which are equivalent with only a "moderate" slowdown to arbitrary computational problems.

Definition 1.4 (Time and space complexity of a Turing machine): Given a Turing machine M, we can define the time complexity $t_M(x)$ to be the number of steps the machine runs for before halting, and we define the space complexity $s_M(x)$ to be the number of *distinct* cells that the machine reads during its execution.

If $M(x)$ is undefined, so is $t_M(x)$.

These metrics, and the trade offs between them, are of primary interest to complexity theory.

We have defined time and space complexity of a specific Turing machine. But we would also like to define them for a *computational problem* $f : \{0,1\}^* \to \{0,1\}^*$.

Definition 1.5 ((Naive) complexity of a function): We define the complexity of a computable $f : \{0,1\}^* \to \{0,1\}^*$ to be the complexity of the "best" machine computing f, where "best" depends on specific property we're interested.

Unfortunately, "best" is hard to define.

1.2 Digression: Kolmogorov complexity (A success story)

Given an input x, rather than computing anything on x, we simply want to output x. I.e., we want a machine M such that $M(\Lambda) = x$, where Λ is the empty string.

Definition 1.6 (Kolmogorov complexity with respect to a machine): If we fix a Turing machine M, the Kolmogorov complexity with respect to M is $K_M(x) = \min\{|y| : M(y) = x\}$.

We can think of M as a decompression algorithm, and K_M measures the compressibility of x.

It turns out that in this case, we don't need to worry too much about

Theorem 1.2 (Kolmogorov): *There exists a universal machine U such that for any other M,*

$$K_U(x) \leq K_M(x) + c_M$$

Proof. The universal machine U takes an encoded pair (M, x) and simulates $M(x)$. Then if $K_M(y) = x$, $K_U(y') = x$ for $y' = (M, y)$, where $|y'| \leq y + c_M$. (We can choose an encoding of pairs which makes $|(M, y)| = c_M + y$.) □

We can fix a standard enumeration of all Turing machines $\{M_1, M_2, ..., M_e, ...\}$, and then let $U(e, x) = M_e(x)$.

We can then define Kolmogorov complexity in terms of K_U.

1.3 Machine-independent complexity

Introduced by Blum.

Note that the enumeration $\{M_1, ..., M_e, ...\}$ gives a fixed enumeration of all partial computable functions $\{\varphi_1, ..., \varphi_e, ...\}$. (Note that this list has a large number of duplicates, since there are many Turing machines computing a given function.)

Definition 1.7 (Abstract complexity measure): An abstract complexity measure is a sequence of functions $\{\Phi_1, ..., \Phi_e, ...\}$ satisfying Blum's axioms:
(1) $\text{dom}(\Phi_e) = \text{dom}(\varphi_e)$.
(2) The predicate (in 3 arguments) $\Phi_e(x) = y$ is decidable.

Example 1.1 (Time complexity). The sequence $\{t_1, ..., t_e, ...\}$ where $t_e = t_{M_e}$ represents an abstract complexity measure. The first axiom is trivial, since the domain of time complexity is the same as the domain of underlying machine.

The second axiom is satisfied since we can simply simulate machine M_e on input x for exactly y steps to see if $t_e(x) = y$.

Similarly, $\{s_1, ..., s_e, ...\}$ is an abstract complexity measure, with the first axiom again trivial and the second following from the fact that if we bound the machine to using s cells, it only has $|\Sigma|^s \cdot |Q| \cdot s$ distinct full configurations (the state of the tape, the internal state, and the location of the head), and if the configuration ever repeats, computation must proceed exactly the same way from it as it did the first time, so it must then loop forever. By pigeon hole principle it suffices to simulate the machine for $|\Sigma|^s \cdot |Q| \cdot s$ steps.

Theorem 1.3 (Blum's speedup theorem): *Assume that Φ is any abstract complexity measure and $s(n)$ is an arbitrary total computable function such that $\lim_{n\to\infty} s(n) = \infty$.*

Then there exists a total computable f such that the following holds:

For any e such that $\varphi_e = f$, there exists another machine e' such that $\varphi_{e'} = f$ and for almost all x, $\Phi_{e'}(x) \leq s(\Phi_e(x))$.

We will define our complexity in terms of a budget. A complexity class with respect to Φ and a budget is the set of all f that have at least one solution satisfying the budget.

1.4 Deterministic Time Hierarchy

We will talk for now about time complexity. The worst-case time complexity of a machine M is $t_M(n) = \max_{|x|\leq n} t_M(x)$.

Sometimes we will be interested in complexity in terms of some structural properties of the input rather than just the encoded length, e.g., for graph algorithms, we may be interested in complexity in terms of the number of vertices or edges, rather than total description size (which depends on the representation).

Definition 1.8 (Deterministic time): For an arbitrary monotonic function $t(n)$, we define the class deterministic time $t(n)$ to be

$$\mathsf{DTIME}[t(n)] = \{f \mid \exists M \text{ computing } f \text{ s.t. } t_M(n) \leq O(t(n))\}$$

It is worth noting that for low levels of the hierarchy, e.g., $\mathsf{DTIME}[n]$, we do actually care about the details of the model, such as number of heads and tapes.

We then have $\mathsf{DTIME}[n] \subseteq \mathsf{DTIME}\left[n \log^{O(1)} n\right]$ where the $O(1)$ in the exponent is taken to mean $\mathsf{DTIME}\left[n \log^{O(1)} n\right] = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}[n \log^k n]$. We have

$$\begin{aligned}
&\mathsf{DTIME}[n] && \text{(linear time)}\\
\subsetneq\ &\mathsf{DTIME}\left[n \log^{O(1)} n\right] && \text{(quasilinear time)}\\
\subsetneq\ &\mathsf{DTIME}[n^2] && \text{(quadratic time)}\\
\subsetneq\ &\mathsf{DTIME}\left[n^{O(1)}\right] = \mathsf{P} && \text{(polynomial time)}\\
\subsetneq\ &\mathsf{DTIME}\left[2^{O(n)}\right] = \mathsf{E} && \text{(simply exponential time)}\\
\subsetneq\ &\mathsf{DTIME}\left[2^{n^{O(1)}}\right] = \mathsf{EXP} && \text{(exponential time)}
\end{aligned}$$