CAS CS 535: Complexity Theory

Lecturer: Mark Bun — Fall 2023

Lecture Notes 1:

Course Welcome, Turing Machines, Decidability

Reading.

- Arora-Barak § 0, 1.1-1.5, 1.7 (optional), A.1-A.2

1 Logistics

Instructor: Mark Bun
Teaching Fellow: Mandar Juvekar
Course Website: `https://cs-people.bu.edu/mbun/courses/535_F23`

- Prerequisites: CS 332 (undergrad Theory of Computation) and "mathematical maturity"
- Piazza: All announcements here. Ask me for the entry code.
- Gradescope: All assignments to be submitted here. Entry code ZWZGPW.
- Assignments: Weekly homework due Tuesday (11:59 PM). In-class midterm and final. Term paper. Participation: weekly reading responses and discussion attendance.
- Collaboration Policy: You can discuss (most) homework problems with up to 3 other students, but you must acknowledge your collaborators and write solutions by yourself. Please read, sign, and submit to Gradescope the course collaboration policy.

2 What's this course about?

At the risk of a bit of oversimplification, the goal of computational complexity theory is to address the following question:

$$\text{What } \underset{(3)}{\underline{\text{resources}}} \text{ are needed to } \underset{(2)}{\underline{\text{solve}}} \ \underset{(1)}{\underline{\text{computational problems}}}\text{?}$$

To answer this question scientifically, we of course need to define what we mean by each of these terms.

(1) For simplicity, we will usually focus on **decision problems**, i.e., computational problems with yes/no answers, over inputs presented as bit strings. More precisely, solving a decision problem corresponds to:

- Evaluating a Boolean function $f : \{0,1\}^* \to \{0,1\}$, or equivalently

- "Deciding" a language $L \subseteq \{0,1\}^*$ defined by the correspondence $x \in L \iff f(x) = 1$.

Other types of problems we'll encounter include search, optimization, approximation, counting, and sampling.

(2) We will be able to talk precisely about the computational processes used to solve problems via abstract *models of computation*. Typical (and some not-so-typical) models include:

- Turing machines (TMs)
- RAM machines
- Boolean circuits
- Arithmetic circuits
- Quantum TMs/circuits
- Cellular automata, DNA computing, slime molds, ...

(3) The key feature that distinguishes computational complexity from the related field of computability theory is its focus on quantifying computational costs or resource requirements. These resources include:

- Running time
- Space (memory)
- Nondeterminism
- Circuit size
- Circuit depth
- Randomness
- Parallel time
- Communication/interaction
- Oracle queries

Some of the more specific objectives in complexity theory include:

- Designing efficient algorithms, i.e., finding upper bounds on the resources needed to solve problems.
- Establishing lower bounds, i.e., proving that some problems are inherently difficult to solve.
- Identifying relationships between different computational problems and between the resources needed to solve them.

Some examples of questions we'd <u>like</u> to answer include:

$\mathbf{P} \overset{?}{=} \mathbf{L}$ Can every time-efficient computation be done with small memory?

$\mathbf{P} \overset{?}{=} \mathbf{NC}$ Can every time-efficient computation be parallelized?

$\mathbf{NP} \overset{?}{=} \mathbf{PCP}$ Can one check a long proof by reading only a small random number of bits?

$\mathbf{NP} \overset{?}{=} \mathbf{IP}$ Are interactive conversations more powerful than static proofs?

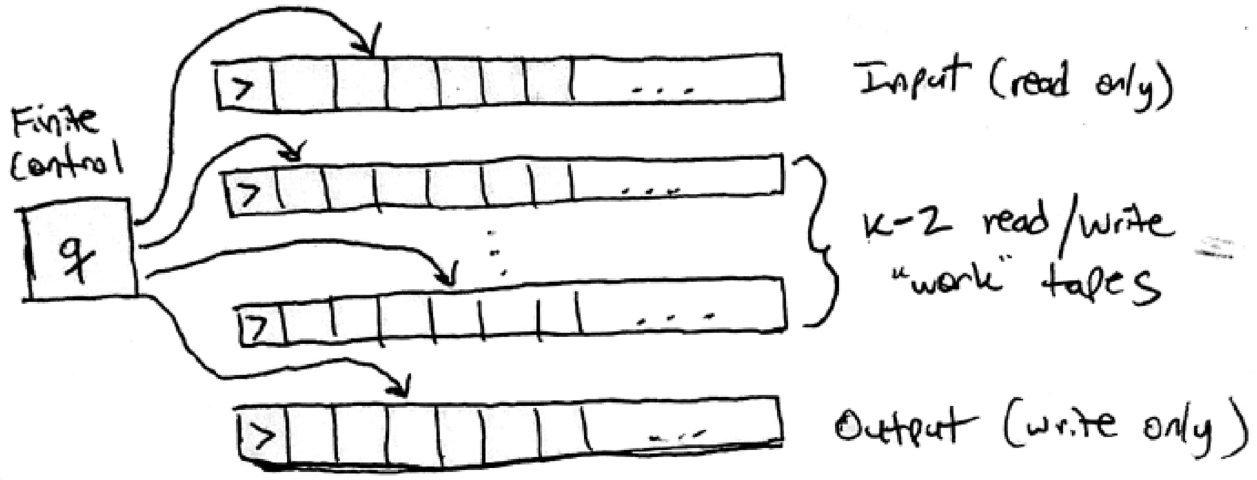
$\mathbf{EXP} \overset{?}{\subseteq} \mathbf{P}_{/\mathrm{poly}}$ Can specialized hardware always be used to compute hard functions?

$\mathbf{BPP} \overset{?}{=} \mathbf{P}$ Can every efficient randomized algorithm be derandomized?

So far we (meaning, sixty+ years of complexity theorists) have only been able to answer one of these questions...though in my opinion it's the wildest of the bunch.

3 Multi-tape Turing machines

The primary model of computation we'll work with in this class is the k-tape Turing machine.



Initially: The input is written on the input tape, and all other tapes are blank. The control is in the start state, denoted q_{start}. All heads are in the left-most position.

In one step of computation: Based on the current state and the symbols under the read heads,

1. Change symbols under the write heads
2. Move heads (one step left, one step right, or stay)
3. Update state

Computation halts when the control is in a special state q_{halt}.

More formally, a k-tape TM M is described by a tuple (Γ, Q, δ) where

- Γ is a finite tape alphabet such that $0, 1, \Box, \triangleright \in \Gamma$. (Here, $\Box$ is the blank symbol and $\triangleright$ demarcates the left end of each tape.)
- Q is a finite state set such that $q_{\text{start}}, q_{\text{halt}} \in Q$.
- $\delta : Q \times \Gamma^{k-1} \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ is a transition function describing how one step of computation proceeds.

For a Turing machine M and an input $x \in \{0,1\}^*$, we define

$$M(x) = \begin{cases} \text{content of the output tape if } M \text{ halts on input } x \\ \perp \text{ otherwise.} \end{cases}$$

We say that M computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for every $x \in \{0,1\}^*$.

In the special case where the codomain of f is just $\{0,1\}$, i.e., f corresponds to a language $L \subseteq \{0,1\}^*$, we say that M decides the language L if $M(x) = 1 \iff x \in L$.

4 The Church-Turing Thesis

The Church-Turing Thesis is an assertion that connects the formal mathematical study of Turing machines to the real-world explanatory goals of computability and complexity theory. There are (at least) two versions of the thesis.

Turing machines capture:

1. Our intuitive notion of algorithms
2. Every physically realizable model of computation

The Extended Church-Turing Thesis is the version relevant to computation complexity theory. It's the same as the above, but with the stronger assertion that simulation via TMs can be done with at most polynomial overhead.

Note that there are credible challenges to the validity of the Extended Church-Turing Thesis, e.g., from randomized, parallel, and quantum computation. The statement is much less controversial if we restrict its scope to deterministic, sequential computation.

Positive evidence for the (Extended) Church-Turing Thesis comes from simulation arguments, which show how to simulate a powerful model of computation using a seemingly less powerful one. The basic resources we'll consider are:

Time Turing machine M runs in time $T(n)$ if for all $x \in \{0,1\}^*$, M halts on input x in at most $T(|x|)$ steps.

Space Turing machine M runs in space $S(n)$ if for all $x \in \{0,1\}^*$, M halts on input x while accessing at most $S(|x|)$ spaces of its work tapes.

Simulation arguments can be used to show the (extraordinary) robustness of the multi-tape TM model. For instance, the following modifications to the TM model incur at most polynomial increases in time and space:

1. Assuming the tape alphabet is $\Gamma = \{0, 1, \Box\}$
2. Taking $k = 3$ (only one work tape), or even $k = 1$ read/write tape (taking some care with how to measure space usage)
3. Pointer access to tape cells (RAM model)
4. Bidirectionally infinite or multi-dimensional tapes

5 The Universal TM

Turing's original paper made the game-changing observation that general-purpose computation is possible. Underlying this is the basic idea that *programs* (computing devices) can themselves be represented as *data* that is presented as input to other programs.

Specifically, the description of any TM can be encoded as a binary string. The notation we'll use for this correspondence is:

$$\alpha \in \{0,1\}^* \leftrightarrow \text{TM } M_\alpha$$

$$\lfloor M \rfloor \in \{0,1\}^* \leftrightarrow \text{TM } M.$$

The idea of a general-purpose computer is then captured by the Universal TM:

Theorem 1. *There exists a Turing machine U such that for all $\alpha, x \in \{0,1\}^*$, we have $U(\alpha, x) = M_\alpha(x)$. Moreover, for all α there exists a constant C_α such that*

1. *If M_α halts on x in T steps, then $U(\alpha, x)$ halts in at most $C_\alpha \cdot T \log T$ steps.*
2. *If M_α halts on x in space S, then $U(\alpha, x)$ uses space at most $C_\alpha \cdot S$.*

That is, there exists a Turing machine U that can simulate the behavior of any other TM M_α. Moreover, the simulation is efficient in that it incurs at most logarithmic time and constant space overhead relative to running the machine M_α itself.

Proof idea. We'll sketch a proof of the weaker statement that simulation is possible with quadratic time overhead (i.e., replace $T \log T$ in item 1 above with T^2).

On input (α, x), the UTM U first transforms the machine M_α into an equivalent machine M'_α with a simpler structure, namely:

1. M'_α has exactly one work tape, and
2. M'_α uses tape alphabet $\Gamma = \{0, 1, \Box, \triangleright\}$.

This new machine has the property that if M_α runs in time $T(n)$, then M'_α runs in time $O(T(n)^2)$.

Now U simply simulates the execution of M'_α step-by-step on input x using three work tapes. The contents of the tapes are as follows:
Work tape 1: Description of the transition function of M'_α
Work tape 2: Current state of M'_α
Work tape 3: Content of work tape of M'_α Output tape: Same as M'_α.

To simulate one computation step, U does the following:

- Read current symbols from the input tape and work tape 3.
- Traverse work tapes 1 and 2 to determine how to update.
- Traverse work tapes 2 and 3 to perform the update.

For the runtime/space analysis: Observe that simulating each step takes time/space depending on the contents of work tapes 1 and 2, which are constants depending on M'_α but not on the input x. $\Box$

6 Undecidability

As a check on the power of Turing machines, and via the Church-Turing Thesis on computation in general, let us briefly review the idea of uncomputability/undecidability. Some functions cannot be computed by TMs, even using arbitrary computational resources.

Example 2. Define the language $\mathsf{SNA} \subseteq \{0,1\}^*$ by

$$x \in \mathsf{SNA} \iff M_x(x) \text{ does } \underline{\text{not}} \text{ halt with } 1 \text{ on its output tape.}$$

The language SNA is undecidable.

Proof. Assume for the sake of contradiction that TM M decides SNA. Then

$$\begin{aligned} \lfloor M \rfloor \in \mathsf{SNA} &\iff M(\lfloor M \rfloor) = 1 && \text{(definition of deciding)} \\ &\iff \lfloor M \rfloor \notin \mathsf{SNA} && \text{(definition of } \mathsf{SNA}\text{),} \end{aligned}$$

which is a contradiction. □

The language SNA is admittedly contrived. But luckily, it turns out to be useful for showing that other more natural functions are uncomputable. For example, consider:

$$\mathsf{HALT} = \{\langle \alpha, x \rangle \mid M_\alpha \text{ halts on input } x\}.$$

We can prove that HALT is also undecidable by a *reduction* from the undecidable language SNA. That is, suppose there were a TM M_{HALT} deciding HALT. Then we can use M_{HALT} to construct a new TM M_{SNA} deciding SNA, a contradiction.

Here's a description of the new machine M_{SNA}.

On input x:

1. Run $M_{\mathsf{HALT}}(x, x)$. If 0, halt and output 1. Else:
2. Use the Universal TM U to compute $b = M_x(x)$.
3. If $b = 1$, output 0.
   If $b = 0$, output 1.

Next time: Time complexity classes, nondeterminism.