CS 535: Complexity Theory, Fall 2026

Homework 1

Due: 11:59PM, Friday, September 11, 2026.

Reminder. Homework must be typeset with LaTeX preferred. Problems 1, 2(a), and 3 (all parts) are required. Problem 2(b) is optional.

Problem 0 (Housekeeping)**.** Please fill out the tutorial scheduling form (`https://forms.gle/9C4viRSy9MAHreCi7`) if you have not done so already.

Problem 1 ((*Required*) Recursion Review)**.** Let $c > 0$ be an arbitrary constant. Define a function f recursively via $f(1) = 1$ and $f(n) = f\left(\frac{n}{2}\right) + c \log n$. Determine the asymptotic rate of growth of f. That is, find a simple, closed form (i.e., not recursively defined) function g such that $f(n) = \Theta(g(n))$ and justify your answer.

Hint: The constants hidden in Θ can depend on c, so it does not need to appear in your expression for g. Don't completely forget about this function since it will be helpful to us when we study space complexity.

Problem 2 (Arithmetizing Turing Machines)**.** In this class, we usually think of Turing machines as computing functions $f : \{0,1\}^* \to \{0,1\}^*$, but TMs and their variants are also naturally suited to computing functions $f : \mathbb{N} \to \mathbb{N}$ over the natural numbers. In this problem you'll explore a few such TM variants that historically helped bridge the machine view of computability with the recursive function and λ-calculus view.

(a) (*Required*) A **binary arithmetic machine** maintains a tuple of counters $(C_1, \ldots, C_m)$ and computes a function $f : \mathbb{N} \to \mathbb{N}$ as follows. At initialization, the first counter C_1 is set to the input $z \in \mathbb{N}$ and all other counters are set to 0. In one step of computation, the machine "reads" the contents of each counter by testing whether it is equal to zero. Based on this information, and its finite control state, it may then (possibly independently for each counter) update each counter using one of the following arithmetic operations:

$$
\begin{aligned}
&C_i \leftarrow C_i && \text{[no change]},\\
&C_i \leftarrow 0,\\
&C_i \leftarrow 1,\\
&C_i \leftarrow 2C_i + C_j && \text{for some other counter } C_j,\\
&C_i \leftarrow \lfloor C_i/2 \rfloor,\\
&C_i \leftarrow C_j \mod 2 && \text{for some other counter } C_j.
\end{aligned}
$$

All right-hand sides are evaluated at the beginning of the step, and all updates take effect simultaneously. When the machine halts, its output is the natural number loaded in the last counter C_m.

(i) Show (using enough detail to convince your human reader) that a binary arithmetic machine using at most $3k$ counters can simulate an arbitrary k-tape Turing machine. That is, every k-tape TM can be converted into a $3k$-counter binary arithmetic machine that solves the same problem. Here you may use any reasonable convention for converting between natural numbers and binary strings.

(ii) What is the runtime overhead of your simulation? That is, given an ordinary TM running in time $T(n)$ on inputs of length n, for some function $T(n) \geq n$, what is the runtime of your binary arithmetic machine as a function of $T(n)$?

(b) (*Optional*) A **counter machine** has the same initialization and output conventions as a binary arithmetic machine. In each step of computation it is also reads its counters by testing whether they are equal to zero, but then is only allowed update each counter by either incrementing by one, decrementing by one, or leaving it the same. (Attempting to decrement a zero counter leaves it as zero.)

(i) Show that a binary arithmetic machine using m counters can be simulated by a counter machine using $O(m)$ counters.

(ii) Let n denote the length of the standard binary representation of an input z. Suppose a binary arithmetic machine runs in at most $T(n)$ steps on every n-bit input, for some function $T(n) \geq n$. What is the runtime of your simulating counter machine as a function of $T(n)$?

(iii) Is it possible to significantly improve your runtime overhead? Prove your answer.

Problem 3 ((*Required*) Encodings and Undecidability)**.** An encoding is an unambiguous way to represent an object, e.g., a natural number, a graph, or a Turing machine, as a string in Σ^* for some alphabet Σ. Encodings allow us to present complex objects as inputs to Turing machines and other computational devices. Following Section 1.4 of Arora-Barak, one way to formalize a *nice encoding* of a set of objects D is as a function $f : \Sigma^* \to D$ such that, for every $y \in D$, there are infinitely many $x \in \Sigma^*$ such that $f(x) = y$. (Check that this indeed formalizes the discussion there!)

We will typically fix a reasonable, unspecified way of encoding objects as binary strings (strings over the binary alphabet $\Sigma = \{0, 1\}$) and refer to an object and its shortest encoding interchangeably. But it's sometimes worth keeping in mind how the choice of encoding affects which statements we can prove.

(a) Show that the set of Turing machines $\mathcal{M}$ can be nicely encoded using the unary alphabet $\Sigma = \{1\}$ via a function $f_1 : \{1\}^* \to \mathcal{M}$. You may use without proof the fact that Turing machines can be nicely encoded in binary as described in Section 1.4.

(b) A unary language L is a language over the alphabet $\{1\}$, i.e., a set $L \subseteq \{1\}^*$. Show that there exists an undecidable unary language.

Hint: I know of at least three good ways to solve this problem. Keep this fact in mind for when we study circuit complexity later.