October 16, 2025

Problem Set 2

Exercise 1: Turing Machines. A Turing Machine is an object defining computability, that is anything a computer can achieve a TM can achieve and more (try to see why this may be given the definition of a TM). A Turing Machine is defined in a similar way to a DFA with its finite states, head positions. However, it is different in the sense that the head can move both ways, and it reads and writes one its tape (this question deals with single-tape TMs). The input is placed on this infinitely long tape and all of the above movements are governed by the transition function.
Formally, a Turing Machine is a 7-tuple $M = \{Q, \Gamma, b, \Sigma, \delta, q_0, F\}$: Q is the finite state set, Γ is the tape alphabet (the set of letters that the TM can write on its tape), $b \in \Gamma$ is the blank string, (the non-input portion of the infinite tape is filled with b), $\Sigma \subseteq \Gamma \backslash \{b\}$ is the input alphabet, q_0 is the initial state, $F = \{q_{accept}, q_{reject}\}$ is the final state set. $\delta : Q \backslash F \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
On input x, the input is written on the leftmost end on the tape (and the rest of the tape is filled with blanks). The head position is the leftmost cell on the tape.
The TM is said to halt if it ever transitions into a state in F. It accepts x if on input x it halts by transitioning into $q_{acccept}$. The language decided by a TM M is the set of strings accepted by the TM, denoted as $L(M)$.

1. ($\star$) Design a TM whose language is the set of all palindromes. You can do this by describing how the TM's head moves and what it does.

2. ($\star$) Design a TM whose language is the set of all balanced parenthesis with $\Sigma = \{(,), [,]\}$.

The following question does not depend on the rest:

3. ($\star\star$) Show that the set of palindromes is not the language of any DFA.

The following questions are about representing DFA's using TMs: Note that $DSPACE(s(n))$ is the set of all languages that can be recognized by a TM that uses at most $s(n)$ space on its work tape. REG is the set of languages decided by a DFA.

4. ($\star\star$) Show that $REG = DSPACE(O(1))$.

5. ($\star\star\star$) Show that $REG = DSPACE(o(\log\log(n))$

Exercise 2: Kolmogorov Complexity. A *universal Turing machine* is a Turing machine U which can simulate any other Turing machine M given some encoding of it, along with an input to simulate it on. In more familiar terms, consider your favorite programming language L. Then a universal program U for L is a program which can simulate any other program, i.e. an *interpreter*.

The *Kolmogorov complexity* of a string $x \in \{0,1\}^*$, denoted $K_U(x)$, is the length of the shortest input which makes U output x on its tape. For all U, the function $K_U(\cdot)$ is not computable, however we can still give interesting and useful bounds on the Kolmogorov complexity of strings.

The first 4 questions are simple and meant to give you a working understanding of how $K_U(\cdot)$ behaves. The next 2 questions are more difficult, but showcase how Kolmogorov complexity can be a very versatile tool.

1. ($\star$) Show that for any two universal Turing machines U, U', there exists a constant c such that for all $x \in \{0,1\}^*$, we have $K_U(x) \leq K_{U'}(x) + c$.

   (**Note:** Hereafter we may simply write $K(\cdot)$, which implicitly relies on some unspecified universal Turing machine.)

2. ($\star$) Show that for all n, there exists $x \in \{0,1\}^n$ such that $K(x) \geq n$.

3. ($\star$) Show that for all $x, y \in \{0,1\}^*$, we have $K(xy) \leq K(x) + K(y) + O(1)$.

4. ($\star\star$) Show that it is not necessarily the case that for all $x, y \in \{0,1\}^*$, we have $K(xy) \geq K(x)$.

5. ($\star\star$) Show that there are infinitely many prime numbers using Kolmogorov complexity.

6. ($\star\star\star$) Let $p_1, p_2, p_3, \ldots$ be a list of all the primes in ascending order. Show that for all $n \geq 1$, we have $p_n = O(n \log^2 n)$ using Kolmogorov complexity.

   (**Note:** The Prime Number Theorem says $p_n = O(n \log n)$.)

Exercise 3: Matrix Multiplication (but faster). Consider matrices $A, B \in \mathbb{R}^{2^n \times 2^n}$. Multiplying matrices is a natural operation to consider, and is likely familiar if you have previously taken a course in linear algebra (e.g. Math 235). Matrix multiplication is used in all sorts of applications—for instance, many algorithms in AI rely on multiplying matrices quickly.

We will take a look at a fun way to multiply matrices more efficiently than the straightforward approach. You may assume that numbers can be multiplied in constant time.

1. ($\star$) Consider the naive algorithm for multiplying matrices A and B by hand. What is the time complexity of this approach?

2. ($\star\star$) Suppose A and B were both 2×2 matrices. How many scalar multiplications would you need to compute AB? (Hint: you can get it to 7)

3. ($\star\star\star$) Use (2) to construct an algorithm for multiplying matrices A and B with running time asymptotically better than the naive approach. (What can you do if you split A and B into 4 parts?)