February 11, 2026

Problem Set 1

Exercise 1: Turing Machines. A Turing Machine (TM) is a mathematical model of computation describing an abstract machine that reads and writes symbols on a memory tape according to a fixed internal program. We imagine a TM to have a right-infinite memory tape which it stores information on during the program execution, accessible by a read-write head initialized at the leftmost tape cell. TMs are powerful enough to compute any mathematical function that we would typically deem "computable." (Indeed, computability is formally defined to be "computable by a TM.")

Formally, a Turing Machine is a 7-tuple $M = \{Q, \Gamma, \square, \Sigma, \delta, q_0, F\}$:

- The state set Q is a finite set of program states.
- The tape alphabet Γ is a finite set of symbols that can be written to the tape.
- The blank symbol, $\square$, is an element of Γ.
- The input Σ is a subset of $\Gamma \setminus \{\square\}$ containing the symbols which are allowed to appear in the input.
- The transition relation $\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{Left, Stay, Right\}$ details the internal program.
- The start state q_0 is an element of Q.
- The final state set F contains special states q_{accept}, q_{reject}.

Given an input $w \in \Sigma^*$, the TM M starts with $w = w_1 \dots w_n$ placed in the first $|w|$ tape cells. The first program instruction $\delta(q_0, w_1) = (q, a, D) \in Q \times \Gamma\{Left, Stay, Right\}$ says that M should *write* a under the tape head, *move* the tape head in direction D, and *transition* to program state q. Subsequent program instruction are obtained by feeding δ the current "snapshot" of M's configuration, i.e. the current state and the current letter under the tape head. The TM M is said to *accept* its input if it eventually transitions to q_{accept}, *reject* its input if it eventually transitions to q_{reject}, or *loop* if it never transitions to a state in F. The *language* decided by M, denote $\mathcal{L}(M)$, is the set of inputs which M accepts.

The first two questions are meant to help you understand the basics of "programming" a Turing Machine.

1. ($\star$) Design a TM whose language is the set of all palindromes. You can do this by describing what states the TM has, and what its transition relation looks like.
2. ($\star$) Design a TM whose language is the set of all properly nested, balanced parenthesis with $\Sigma = \{(,), [,]\}$.

The following question does not depend on the rest:

3. ($\star$) Show that neither of the above languages are computable by a DFA. (**Hint:** For each language L, construct an infinite set of L-distinguishable strings.)

The last two questions are about *resource-bounded* Turing Machines. Note that $DSPACE(s(n))$ is the set of all languages that are recognized by a TM using at most $s(n)$ space on its work tape, where n is the input length, and REG is the set of regular languages.

4. ($\star\star$) Show that $REG = DSPACE(O(1))$. (**Hint:** How many possible configurations does a $O(1)$-space TM have? Formally, what is a "configuration?")

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

Exercise 2: Extremal Graph Theory. Let H and G be graphs. We say that G is *H-free* if H is not a subgraph of G. Let $e(G)$ denote the number of edges in a graph G, and define the *extremal number* of H to be

$$\mathrm{ex}(n, H) = \max\{e(G) \mid G \text{ is an } n\text{-vertex } H\text{-free graph}\}.$$

That is, $\mathrm{ex}(n, H)$ is the maximum number of edges a graph can have without necessarily having a copy of H.

We will be interested in the asymptotic behavior of $\mathrm{ex}(n, H)$ as n tends to infinity, which means that we will want to prove upper bounds and lower bounds on $\mathrm{ex}(n, H)$ for various fixed H. It is easy to prove a lower bound on $\mathrm{ex}(n, H)$ by giving an example of any H-free graph G. By definition, it holds that $e(G) \leq \mathrm{ex}(n, H)$. On the other hand, it is hard to prove upper bounds on $\mathrm{ex}(n, H)$. In order to show $\mathrm{ex}(n, H) < m$, you must show that *every* n-vertex graph G with m edges has a copy of H.

1. ($\star$) Prove $\lfloor \frac{n^2}{4} \rfloor \leq \mathrm{ex}(n, K_3)$ by considering a complete bipartite graph $K_{a,b}$ where $a + b = n$. Show that this is the best possible bound you can get if we only pay attention to complete bipartite graphs.

2. ($\star\star$) Prove $\mathrm{ex}(n, K_3) \leq \lfloor \frac{n^2}{4} \rfloor$. (**Hint:** Induction.)

Another function of great interest in extremal graph theory is the *Ramsey number* $R(s, t)$, defined to be the smallest n for which any $(green, red)$-colored K_n contains a green K_s or red K_t. For the sake of intuition, you can think about the graph as representing social relations between people at a party. A green edge (u, v) means u knows v and v knows u, while a red edge means u and v don't know each other. The number $R(s, t)$ thus represents the minimum number of party guests required to guarantee that there is either a group of s friends or t strangers.

3. ($\star$) $R(3, 3) =?$.

4. ($\star\star$) $R(3, 4) =?$

Exercise 3. Residue systems for fast polynomial operations. You may have seen Fast multiplication of polynomials explained via Fast Fourier Transforms, but to me, that seems like CS black magic. Instead, we'll explore a more general computer algebra technique that demonstrates exactly why FFT-based algorithms are so fast. To begin, we note that there are 4 crucial polynomial operations we want to find a good time complexity for:

- Multiplication of two polynomials: Compute fg.
- Division with remainder: Compute r and q such that $g = qf + r$, with $\deg r < \deg f$.
- Evaluation at multiple points: Compute $f(r_i)$ for a set of given r_i.
- Interpolation: Compute f given a set of r_i and y_i, such that $f(r_i) = y_i$ (to be exact, there are multiple such f, we want to compute the unique f modulo $g = \prod(x - r_i)$).

If arithmetic on the underlying field L costs $O(1)$, the naive runtime of all the above is $O(mn)$ where $m = \deg f$ and $n = \deg g$ are the degrees of the polynomials involved (or the size of the given set for the last two). Fast effectively means $O((n + m) \log(n + m))$.

1. ($\star$) Assume that f is "sparse" (i.e. has $O(\log n)$ non-zero coefficients). Make a fast algorithm for multiplication and division with remainder by f.

In the next problems, the set $r_1, \ldots, r_m$ is fixed and known beforehand (i.e. any preprocessing that only involves them takes amortized cost and can be considered constant, but computations that also depends on the inputs can incur m as a factor):

2. ($\star\star$) Given some f, how can we compute the evaluations $f(r_i)$ fast, assuming that division with remainder is fast? Key ideas to get you started:
   - Evaluating at r_i is the same as modding out by $x - r_i$ (i.e. setting $x - r_i = 0$).
   - Division with remainder allows you to limit the degree of your remainder.
   - If you computed $r \equiv f \bmod (ab)$, then you can compute $f \bmod a$ as $r \bmod a$.
3. ($\star\star$) Given some set $y_1, \ldots, y_m$ (one for each r_i), how can we compute the interpolation f fast, assuming that multiplication is fast? Hint: use Chinese Remainder Theorem to reverse what you did in part 1!
4. ($\star\star$) Using fast evaluation and interpolation at $r_1, \ldots, r_m$, how can you compute fast multiplication of polynomials? Are there any caveats that depends on the set r_i?

If our goal is fast multiplication, it would seem that we're being circular above since we needed fast multiplication for fast interpolation. This is where part (1.) comes in:

5. ($\star\star$) In the interpolation algorithm above, can you ensure that every multiplication is by something that depends only on r_i and not the inputs? (i.e. one factor might depend on y_i, but the other thing can be precomputed in terms of just r_i). Do the same for fast evaluation.

6. ($\star\star\star$) Find some choice of r_i where we can ensure that those precomputed things are all sparse. If you already know FFT-based algorithms, this is the part that shows why roots of unity are important.

7. ($\star\star\star\star$) What if the base field L does not have the r_i you need? How costly would it be to "add" the needed values to your field, and how can you guarantee that things remain fast? For example, if you're working with just integers and rationals, it would seem very costly and loss-y to extend $\mathbb{Q}$ all the way to $\mathbb{C}$ just to get the required numbers.

8. ($\star\star\star\star$) What about general fast division with remainder? (Note: this is not exactly an easy algorithm to come up with, it only depends on fast multiplication and an extra nifty technique called Hensel lifting/Newton's method). With this final piece, we've derived fast algorithms for all four of the operations listed at the start!