1. Consider the Dyck language with two types of parenthesis. Prove that this language is decideable in time $O(n^2)$ on a one-tape Turing Machine. Prove a matching lower bound ($\Omega(n^2)$) for all one-tape TMs deciding this language.

On a multi-tape machine, we can use the second tape as a “stack” to match parenthesis. We go through the input, and each time we see an open parenthesis symbol, we copy it onto a second tape, and move the second tape head right. Each time we see a closed parenthesis symbol, we check that it is the same as the one under our current location on the second tape. If not, or if the second tape is empty, this is a mis-matched parenthesis, and we can reject. Otherwise, we erase the symbol on the second tape and move the tape head left, implicitly matching the parenthesis. At the end, we accept if the second tape is empty.

This takes linear time total. Since we showed in class that a single tape machine simulates a multi-tape machine in the square of its time, this means that we can solve the problem in $O(n^2)$ time on a single-tape machine.

For the lower bound, we showed that the language $\text{Pal} = \{x | x \in \{0, 1\}^n, x_1..x_n = x_n..x_1\}$ requires time $\Omega(n^2)$ on any one-tape machine. We reduce $\text{Pal}$ to $\text{Dyck}_2$ on a one-tape machine as follows.

First, we compute the length of $x$ using a counter, kept between two special symbols. This counter will grow to $\log n$, bits, and every time it grows by a bit, we’ll need to move the part of the input after it one cell to the right. Also, for every bit of the input, we need to increment the counter taking up to $\log n$ time, and then move the bit to the other side of the counter, also $\log n$ time.

All of these factors total $n \log n$. This means that computing these lengths take $O(n \log n)$ time on a one-tape machine. We then use a similar method to find the middle bit of $x$, keeping a second counter and stopping when it reaches half the first. (We need to move both counters with us). If $x$ has odd length, we erase the middle symbol. In either case, we then put a marker over the previous symbol. We then replace each 0 before the middle with a (, and each 1 with [, and each 0 and 1 after the middle with a ) or ]. This is a valid Dyck expression if and only if $x$ is a palindrome.

Thus, the time to solve $\text{Pal}$ is at most the time to solve $\text{Dyck}_2 + O(n \log n)$. Since this is $\Omega(n^2)$, any algorithm solving $\text{Dyck}_2$ requires time $\Omega(n^2) - O(n \log n) = \Omega(n^2)$.

2. It would be nice to have a programming language PL where: A) (Recognizability) we could computably tell whether a string was a valid PL program; B) (Termination guarantee) given a valid PL program and an input, we could simulate the program on the input computably, thus guaranteeing that all programs in our language halt; and C) (Generality) for every recursive language $L$, there is a program in PL that computes membership in $L$.

Show that no programming language exists having all three properties.
We'll assume property A and B hold, and conclude that property C fails. To formalize A and B, say that \( V(p) \) is a computable predicate that determines whether a string \( p \) is a valid program in the PL, and that \( I(p, x) \) is a computable interpreter that simulates valid program \( p \) on input \( x \), so that if \( V(p) \), then \( I(p, x) \) always terminates and decides whether \( x \) is in \( L_p \), the language described by program \( p \).

We'll assume that inputs to programs and programs both use the same symbols; if not, fix an ascii code to convert both to binary strings.

To contradict (C), we will construct a computable language \( D \) that is not \( L_p \) for any valid program \( p \). (\( D \) stands for “diagonal”, since we will view I as a matrix, and use the diagonal entries to create a function not equal to any row.) Let \( D = \{ p | p \) is a valid program and program \( p \) rejects input \( p \} \). To decide whether \( p \in D \), we could first use \( V(p) \) to determine whether it is a valid program. If it is not we can reject, and otherwise we can run \( I(p, p) \) to see whether \( p \) accepts \( p \) or rejects \( p \). Then we accept if and only if \( I(p, p) \) rejects. Since \( V \) only terminates, and we only run \( I \) on a valid program (so by (B), I will terminate), this procedure always terminates, and decides whether \( p \in D \).

If \( D \) were equal to \( L_{p_0} \) for some valid program \( p_0 \), then (since \( p_0 \) is valid), \( p_0 \in L_{p_0} \) if and only if \( I(p_0, p_0) \) accepts if and only if \( p_0 \notin D \). Since \( L_{p_0} \) was assumed to be \( D \), this is a contradiction. Therefore, \( D \) is a recursive language not expressable as any program in the language, so (C) fails.

3. Theorem 5.30 (p. 210) in the Sipser textbook gives such a language. Here is another way to solve the problem.

Let \( H \) be the halting language. We know that \( H \notin REC \), in other words, \( H \) is undecidable. Since \( H \) is recursively enumerable but not decidable, it follows that \( H \notin coRE \). Equivalently \( \overline{H} \notin RE \).

To construct our language that is neither in \( RE \) nor in \( coRE \), we take the disjoint union of \( H \) and \( \overline{H} \):

Let \( L = \{ (b, x) | b = 1 \) and \( x \in H \) or \( b = 0 \) and \( x \notin H \} \)

To show that \( L \) is neither \( RE \) nor \( coRE \), we reduce \( H \leq_m L \) and \( \overline{H} \leq_m L \) in the obvious way. (Map \( x \) to \( (1, x) \) or \( (0, x) \) respectively). Since \( H \notin coRE \) and \( \overline{H} \notin RE \), and mapping reductions preserve being in \( RE \), it follows that \( L \) is neither recursively enumerable not co-recursively enumerable, as desired.