- The homework is due on May 9, 23:59pm. Please submit your solutions to Gradescope.
- Starting from Homework 1, all homework sets allow *group submissions* up to 2 people. Please write down the names of the members *very clearly* on the first page of your solutions.
- Answer the questions in a way that is clear, correct, convincing, and concise. The level of details to aim for is that your peers in this class should be convinced by your solutions.
- You can use any statements proved during the working sessions/lectures without proofs in your solutions.
- You might notice the difficulty of the homework problems are much higher than the worksheets. *This is by design.* These problems are meant to stretch your ability and solidify your understanding of the core concepts.
- You are expected to spend a reasonable amount of time (measured in hours) working on these problems. Remember you are allowed to utilize any resources. Make sure to cite all the people/webpages/source of infomation that helped.
- Some problems are marked with a *star*; these are more challenging (and fun) extra credit problems. They are optional and do not count toward raw grades.

---

1. ***Self-referential machines.***

   (a) Design an algorithm $B$ that, assuming that its own source code $\langle B \rangle$ is conveniently given as input when starts, outputs the *source code* of another algorithm $A$ that prints the source code of $B$ on execution. (The algorithm should be described using pseudocode, not an explicit Turing machine.)

   (b) Design an algorithm that outputs its own source code without any input. (Remember that your algorithm does not have access to its own source code. The executable of the algorithm alone has to generate its source code back.)[1]

   *[Hint: As a warm-up, can you construct a Turing machine $M$ that takes no input, and output a sequence of 1s on the tape, where the number of 1s is equal to two times the number of transitions (edges) in $M$?]*

2. ***Redundant code detector.*** Imagine there is a way to automatically check if some of your codes are *redundent*, as in that there are no inputs that will ever lead to certain part of the code being run. How convenient is that!

   Alas, life is not that simple. We can intuitively view a single-line instruction / code as a state in the Turing machine. Call a state $q$ in a Turing machine ***useless*** if no input string would ever lead to the TM entering $q$.

   Prove that the following language is undecidable.

$$\textsc{Useless?} := \Big\{ \langle M \rangle \mid \text{Turing machine } M \text{ has a useless state} \Big\}$$

[1] Algorithm with such property is called a ***quine***. Wait, you never wrote a quine before? Put the homework on pause and write a quine in your favorite programming language. It might sound like a silly trick, but in fact, this is how compilers can be written in the same language they are trying to compile. Or how computer virus was done.

*3. ***Punched-tape machine.*** *Punched-tape machine* is a Turing machine equipped with an infinite two-way single-row tape, where each cell can be either *punched* or *unpunched*. Furthermore, once a cell is punched, it can never be unpunched. Alternatively, a punched-tape machine is just a Turing machine with unary alphabet consisting of a single symbol ■ (together with the empty cell symbol □), where any symbol ■ written on the tape can never be erased.

Prove that the punched-tape machine is equivalent in power to the standard Turing machine. In other words, the languages accepted by the punched-tape machines are exactly all the computable languages.