- You know the drill now: Find students around you to form a ***small group***; use ***all resources*** to help to solve the problems; ***discuss*** your idea with other group member and ***write down*** your own solutions; raise your hand and pull the ***course staffs*** to help; ***submit*** your writeup through Gradescope in *24 hours*.

Our topic for this working session is *Turing machine equivalences*.

In class we learned that the precise definition of Turing machines *does not matter*. This is a somewhat surprising fact because we would expect, by providing different types of accessory devices (say various modes of memory), the power of the machines would be different; the more powerful the device, the more problems one can solve.

The reason why we don't do a lot of Turing machine construction exercises in this class is because *we already did every time we program*. There is really no point in pushing people to program in highly-restrictive programming languages like BrainFuck besides the pure enjoyment for challenge. However, here are some intuition I didn't have time to provide in the lecture so I'll put it here. In a regular sequential programming language:

- States are *lines* of the pseudocode;
- Current state correspond to the *current line* we are executing (often seen in debugger);
- Accepting states correspond to `return`;
- State transitions are the *branching operations*, like `jump`, `while`, and `if-then`.
- Tape reading are used in operations like `if-then`, or `mov` and `pop` in assembly
- Tape writing comes in huge variety. Modern instruction sets support at least basic arithmetics and bit operations.
- There is also the subroutine calls, and consequently, recursion.

The point is, while the definition of Turing machine looks alien, we are very experienced with its equivalent models; in fact, we can view the study of computer architechtures as efficient implementations of Turing machines for practical purposes. We own von Neumann a lot.

Example. Argue that each of the following Turing machine model is equivalent in power to the standard Turing machine with a single two-way infinite tape.

- Turing machine with *multiple tapes*.

Solution: Let M' be a k-tape Turing machine; our goal is to construct a standard Turing machine M with a single two-way tape that simulates M'.

Data representation. We will store the data on tape i of machine M', at the cells on the tape of M where the index is i modulo k. In other words, if we denote $T_i'[n]$ to be the n-th symbol on tape i of M', then we will store $T_i'[n]$ index $k * n + i$ on the tape of M.

We also store the position of the current pointer on each tape as a *marked symbol*; for every symbol a, there is a marked version ă that signifies that the pointer is currently at the symbol.

Transition simulation. At each transition, machine M' takes input from (1) a state q, (2) the k symbols read by the k pointers, and output the following: (1') the next state q', (2') k symbols to be written by the k pointers, (3') k left/right movements of the k pointers.

In machine M, to transition from state q to q', we will run a subroutine to first search for the unique marked symbol on each tape, read its value and stored in a global constant-size register (or just in the states), and look for the next tape until all current symbols are read. Then perform the transition using the values of the read symbols, by again search for the unique marked symbol on each tape, replace it with the proper symbol to be written (without the mark), then find the proper adjacent symbol and replace it with the marked version based on the required cursor movement. This way we maintain the behavior of the multiple-tape Turing machine using a single-tape machine.

Argue that each of the following Turing machine models are equivalent in power to the standard Turing machine with a single two-way infinite tape. (Or multiple tapes. Or one-way tapes. Or many piece of papers. It doesn't matter as long as you can show they are equivalent.)

1. Turing machine with *no empty cell symbol (*$\square$) and only binary alphabet (`0` and `1`).
2. Turing machine with *multiple heads*. *[Hint: You might want to create some new symbols that represent "markings" on the tape.]*
3. Turing machine with *two stacks*.

To think about later: (No submissions needed)

4. Turing machine with *random access memory*. *[Hint: There are many ways to achieve this; given a memory address, how to extract the corresponding value?]*
5. Turing machine with *two registers*, each stores a binary string (of unbounded size), with left/right shifts and increment operations. *[Hint: Simulate a stack using a register.]*
6. Turing machine with *three counters*, each stores an integer (without bounds) and allows increment and decrement operations. *[Hint: This is known as counter machine; many Turing-equivalent models aim to simulate the counters instead of tapes.]*

Conceptual question: What is the power of Turing machines with *a single stack*? Can it be used to recognize non-regular languages? Are they equivalent to the standard Turing machines?