Homework 4

COMS W3261, Summer A 2022

This homework is due **Tuesday, 6/21/2022, at 11:59pm EST**. Submit to GradeScope (course code: 2KGDW8).

Grading policy reminder: LaTeX is preferred, but neatly typed or handwritten solutions are acceptable. I recommend using the .tex file for the homework as a template to write up your answers. Your TAs may dock points for indecipherable writing.

Proofs should be complete; that is, include enough information that a reader can clearly tell that the argument is rigorous.

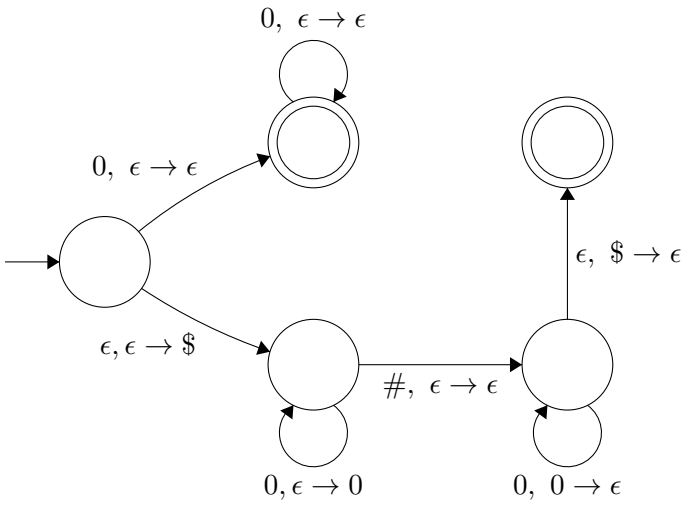
If a question is ambiguous, please state your assumptions. This way, we can give you credit for correct work. (Even better, post on Ed so that we can resolve the ambiguity.)

LaTeX resources.

- The website Overleaf (essentially Google Docs for LaTeX) may make compiling and organizing your .tex files easier. Here's a quick tutorial.
- Detexify is a nice tool that lets you draw a symbol and returns the LaTeX codes for similar symbols.
- The tool Table Generator makes building tables in LaTeX much easier.
- The tool Finite State Machine Designer may be useful for drawing automata. See also this example (PDF) (.tex) of how to make fancy edges (courtesy of Eumin Hong).
- The website mathcha.io allows you to draw diagrams and convert them to LaTeX code.
- To use the previous drawing tools (and for most drawing in LaTeX), you'll need to use the package Tikz (add the command "\usepackage{tikz}" to the preamble of your .tex file to import the package).
- This tutorial is a helpful guide to positioning figures.

1 Problem 1 (7 points)

The following questions concern the pushdown automaton P pictured below.



1. (3 points.) Which of the following six strings are accepted by this PDA? 00#00, 11#11, 0000, #000, 0, 000#00.

   Only 00#00, 0000, and 0. On 00#00, the bottom branch of computation pushes $ followed by two 0's, reads in the #, reads in and pops two more 0's, pops the $ from the bottom of the stack and accepts. 0000 and 0 are accepted by the top branch.

   11#11 is rejected, as every path from the start to the accept state reads in an 0 or an # as the first input character. One branch of computation on #000 pushes $, reads #, and pops $, but then dies in the accept state because there are more input characters that cannot be read in. 000#00 cannot pop the $ to reach the accept state because there is still an 0 on the stack.

2. (4 points.) Consider the following grammar G:

$$S \to A \mid 0B0$$
$$A \to 0A \mid \epsilon$$
$$B \to 0B0 \mid \#$$

   The language of G is almost the same as the language recognized by P, but not quite.

   (a) Name a string in $L(P)$ but not in $L(G)$, and explain why it can't be derived from S in G.

   The string # is accepted by P but cannot be derived using G, as any derivation that ends with the substitution rule $B \to \#$ must have used the rule $S \to 0B0$, creating two 0's.

(b) Name a string in $L(G)$ but not in $L(P)$, and explain why it is rejected by P.

The string ϵ can be derived using G, according to the derivation $S \to A \to \epsilon$. It is rejected by P (the top branch accepts strings matching 0^+, while the bottom branch requires a # to reach the accept state).

Rationale: The goal of this problem is to practice evaluating pushdown automata (and context-free grammars).

References: Sipser p.112 for an introduction to PDAs, p.113 for the formal definition and pp.114-116 for examples. See also Lightning Review 6 on Pushdown Automata.

2 Problem 2 (8 points)

Use the context-free pumping lemma to prove that the following language is not context-free. [Hint: the CFPL has the form "In any context-free language, all sufficiently long strings can be divided into substrings in a way that satisfies three properties". To show that a language is not context-free, we need to contradict this statement; i.e., show that "In this particular language, there exists at least one long string such that *no* division of this string into substrings satisfies all three properties."]

1. (8 points).

$$L_1 = \{0^a\#1^a\#2^{2a} \mid a \geq 0\}.$$

Assume for contradiction that L_1 is context-free. Then L_1 satisfies the context-free pumping lemma: there exists a pumping length p such that for every string $s \in L_1$ with $|s| \geq p$, we can divide s into five pieces $s = uvxyz$ such that (1) $|vy| > 0$, (2) $|vxy| \leq p$, and (3) $uv^ixy^iz \in L_1$ for all $i \geq 0$.

Consider the string $s = 0^p\#1^p\#2^{2p}$ which is in L_1 and longer than p. We proceed to show that every possible division of s into five substrings fails the conditions. We consider three cases:

(a) Either v or y contains a $\#$. In this case, the string $uvvxyyz$ contains at least three $\#$ characters and is not in the language.

(b) If neither v nor y contains a $\#$ character, v and y can contain characters from at most one of the three substrings separated by $\#$. Thus at least one of the substrings is disjoint from (has no characters in common with) v and y. As $|vy| > 0$ by assumption, $uvxyz \neq uvvxyyz$. There are three subcases:

i. v and y contain no 0's. In this case, if $uvvxyyz$ increases the length of the 1 substring, the 0 and 1 substrings will not be the same length and the result will not be in the language. If $uvvxyyz$ does not increase the length of the 1 substring, it increases the length of the 2 substring without increasing the length of the other two substrings and the result is not in the language.

ii. v and y contain no 1's. This case is similar to the previous.

iii. v and y contain no 2's. In this case, $uvvxyyz$ increases the number of 0's and/or the number of 1's without increasing the number of 2's, so the result is not in the language.

Thus there is no way to divide s into 5 substrings in a way that meets the conditions of the pumping lemma. L_1 fails the CFPL and is thus non-context-free.

Rationale: The goal of this question is to practice using the context-free pumping lemma.

References: The CFPL is stated in Sipser p.125, with examples on pp. 128-129. See also the example video on the CFPL, which contains an example.

3 Problem 3 (8 points)

An implementation-level description is less detailed than the *formal description* of a TM as a 7-tuple. You do not need to specify specific states or the transition function for this question. An implementation-level description describes in prose how the TM moves its head around and manages memory.

Example implementation-level description: A TM that recognizes the language

$$\{0^a\#0^b\#0^c \mid 2a + b = c \text{ and } a, b, c \geq 1\}.$$

$M =$ "On input string w:

- Scan the input from left to right to determine whether it matches the regular expression $0^+\#0^+\#0^+$. We can do this in a single pass without writing to the tape because $0^+\#0^+\#0^+$ is a regular expression and can be recognized by a DFA (i.e., a TM without the power to manipulate the tape.)
- Return the head to the left end of the tape.
- Shuttle back and forth between the 0^a and 0^c substring. Each time, we cross off one 0 in 0^a and exactly two 0's from 0^c. Reject if we run out of 0's in 0^c.
- Shuttle back and forth between 0^b and 0^c, crossing off one of each until all 0's in 0^b are gone. Reject if we run out of 0's in 0^c.
- Accept if 0^c is entirely crossed off; reject if there are uncrossed 0's remaining."

1. (8 points). Consider the function f defined on integers as follows: If n is even, $f(n) = n/2$. If n is odd, $f(n) = 3n+1$. It is conjectured that repeatedly applying f to any integer eventually results in the number 1. For example, $f(6) = 3$, $f(3) = 10$, $f(10) = 10/2 = 5$, $f(5) = 16$, $f(16) = 8$, $f(8) = 4$, $f(4) = 2$, $f(2) = 1$.

   Write an implementation-level description of a TM that recognizes the language

   $$F = \{0^n \mid \text{ repeatedly applying } f \text{ to } n \text{ eventually yields } 1.\}.$$

   We will write an implementation-level description of a TM that repeatedly applies f to n, representing the current number as the number of zeroes on the tape.

   $M_1 =$ "On input string w:

   (a) Scan the input from left to right to see if it matches 0^+. If not, reject. If the input a single 0, accept.

   (b) Scan the input from right to left, tracking whether the length of the input is odd or even.

      i. If the input length is even, shuttle back and forth between the beginning and the end of the string. Each time, we cross off a 0 at the beginning and erase a 0 at the end. Once all zeroes are crossed out, uncross all zeroes. (This procedure divides the length of our string by 2.)

ii. If the input length is odd, shuttle to the right and mark the last 0 of our string with a dot. Then, shuttle back and forth between the beginning and the end of the string. Each time, we cross off an 0 at the beginning and write two 0's at the end. After crossing off the marked 0, we write three 0's at the end, then remove all crossings and marks. (This procedure multiplies the length of the string by 3 and adds 1.)

(c) Scan the input from left to right, and accept if the input is a single 0. Otherwise, repeat step (b)."

Alternatively, we could define a machine M_2 that accepts all strings matching 0^+ (i.e., accepts for all n). In order to show that this machine is correct on all n, you should prove the Collatz Conjecture.

Rationale: The goal of this question is to practice thinking about TM memory management; that is, writing to and reading from the tape as necessary.

References: Sipser pp. 174-175 contains two additional implementation-level descriptions of TMs. See also our review video on Turing Machines, which has an implementation-level description on a TM that checks multiplication. For fun, see the wiki page on the Collatz Conjecture (nothing on this page is required to solve this question).

4 Problem 4 (12 points)

Provide *high-level descriptions* of Turing Machines that recognize the following languages.

A high-level description is an algorithm for a Turing Machine described in prose, ignoring implementation details such as the way information is encoded or where the head needs to move. However, your TM's behavior should still be *precisely specified*: it should be clear what the Turing Machine does in every case.

Example high-level description: We'll build a Turing Machine M that recognizes the language

$$\{\langle G\rangle \mid \langle G\rangle \text{ encodes a } \textit{connected graph.}\}$$

. Our Turing Machine will implement a breadth-first search on the encoded input as follows:

- First, check to see if the input encodes a graph and reject if not.
- Mark the first vertex v in the graph.
- Review the encoded graph and mark every vertex adjacent to a marked vertex. Repeat this step until no new nodes are marked.
- Accept if all vertices are marked and reject otherwise.

1. (2 points) $A_1 = \{c \mid a^2 + b^2 = c^2 \text{ for some pair of integers } a, b \geq 1\}$.

   Our TM to recognize A_1 will operate as follows:

   - First, we scan c and reject bad input (i.e., anything that's not a number). (This step is implicit; including it in a high-level description is optional.)
   - Compute $a^2 + b^2$ for all integers a and b between 0 and c. Accept if $a^2 + b^2 = c^2$ for any pair and reject otherwise.

2. (5 points) $A_2 = \{\langle D\rangle \mid \langle D\rangle \text{ encodes a DFA that accepts at least one string matching } (01)^*\}$.

   Our TM to recognize A_2 will operate as follows:

   - First, we reject any input that does not encode a DFA. (This step is implicit; including it in our high-level description is optional.)
   - Our encoded DFA includes all the information we need to simulate D on an input string: we simply track which input characters we've read and which state we're in, following the transition function.
   - Simulate D on all strings in $(01)^*$ in increasing order of length: $\{\epsilon, 01, 0101, 010101, ...\}$. Halt and accept if any string accepts. Note we should specify some order of checking strings in $(01)^*$, as this is an infinite set and we can't just "check them all." We can assume a simulation of D halts on any input string because D is a DFA.

   As written, this TM halts only when it finds a string matching $(01)^*$ that the input DFA accepts. This is sufficient to recognize A_2, but not to decide it, as our TM will run forever on input DFAs that fail our criteria. A puzzle: could we build a TM that *decides* this language?

3. (5 points) A_3, the language containing every encoded graph $\langle G \rangle$ that contains a *Hamiltonian path*, i.e., a path containing each vertex exactly once.

   Our TM to recognize A_3 isn't fast, but it solves the problem. It operates as follows:

   - First, we reject any input that does not encode a graph. (This step is implicit; including it in our high-level description is optional.)
   - Let n be the number of vertices in our graph. For each of the $n!$ sequences of n vertices, we attempt to traverse the path indicated by the vertex sequence. Accept if any path is in the graph.
   - Reject if every potential path contains at least one edge not contained in the graph.

   Side note: the problem of deciding whether or not an encoded graph is in A_3 is NP-complete. This means that unless $P = NP$ (a statement which no one has proved or disproved, but many theoreticians consider to be unlikely), no Turing Machine can solve this problem in time polynomial in n (i.e., in fewer than n^k steps, for some k.) However, it *is* possible to solve the problem using far fewer than $n!$ paths (wiki page).

Rationale: The goal of this question is to practice thinking about Turing Machines as general-purpose computers that can execute algorithms for solving complex decision problems.

References: Sipser pp. 186-187 contains the high-level description given as an example, as well as a breakdown from the high level to the implementation level. TM descriptions later in the book (for instance, those on pp. 195-196) are also given at the high-level.

5 Problem 5 (2 Extra Credit Points): Turing Completeness

I know, Turing Machines are wonderful. But there are other wonderful things. For this week's extra credit problem, we will learn about things that are just as wonderful as TMs— things that are **Turing Complete** (Wikipedia).

5.1 Lambda Calculus (1 pt)

Since it's called the Church-Turing Thesis instead of just "Turing Thesis", maybe do Alonzo Church a favor and read this short tutorial on Lambda Calculus. It is known that Lambda Calculus and Turing Machines are equivalent, i.e. Lambda Calculus is Turing Complete. In your own words, describe how you find the two equivalent models different in "flavor." One short paragraph suffices. (Hint: think about their names, Lambda *Calculus* vs. Turing *Machines*.)

(Anything along the line is okay.) Lambda Calculus looks more like a programming language/functions/arithmetics, while TMs have more architecture and machinery; Lambda Calculus emphasizes evaluation (the *what*, the software), TMs emphasize the action of heads on the tapes (the *how*, the hardware).

Side note: Pretty much everything in Computational Complexity is defined based on TMs, while Lambda Calculus is the foundation of Programming Language Theory.

5.2 Accidentally Turing Complete (1 pt)

As intelligent and ambitious CS students, it is probably your worst nightmare to be stuck at a PowerPoint-making job. As it turns out, however, PowerPoints are *wonderful* too. Read this paper which convinces you the above statement, then Google around and tell us one more example of unlikely Turing Completeness.

This website listed many: Accidentally Turing Complete.