Final Exam

COMS W3261, Summer A 2022

This final exam was released on GradeScope on Wednesday, 6/29 at 12:01am EST. The submission window closes on **Friday, 7/1, at 9:00pm EST with NO EXTENSIONS**. You have **12 hours** from when you download the file to work on the test before uploading. Submit to GradeScope (course code: 2KGDW8).

Final exam rules. For this final, you may refer to your notes, your past homeworks, the textbook, and resources linked from the course webpage, including review videos, lecture notes and the course skeleton. **Other internet resources are not allowed,** with the exception of LaTeX tutorials (like those below) unrelated to the course content.

You may post on Ed for clarifications, but **please do not include specifics about your answers or answer strategies.** The course staff will answer only simple clarification questions about the problem statements.

Grading policy reminders: LaTeX is preferred, but neatly typed or handwritten solutions are acceptable. I recommend using the .tex file for the final as a template (I'll make this available). 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.)

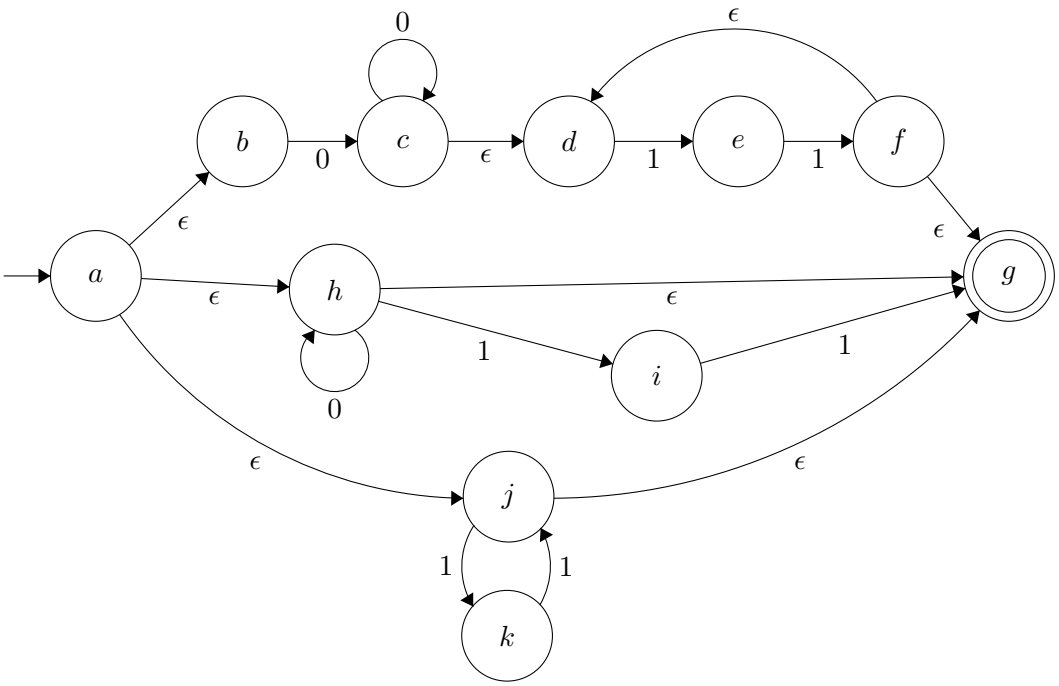
LaTeX resources.

- 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 (11 points)

Consider the NFA state diagram pictured below.



1. (3 points) Which of the following strings does the NFA accept? No justification required. ϵ, 0, 1, 110, 0011, 000111.

   The strings that this NFA accepts include ϵ, 0, and 0011.

2. (3 points) Any accepting branch of computation must take one of three "routes": the 'top route' (any path that uses b, c, d, e, and/or f as intermediate states), the 'middle route' (any path that uses h and/or i as intermediate states), or the 'bottom route' (any path that uses j and/or k as intermediate states). For each of the three routes, write a regular expression that matches strings for which an accepting branch of computation uses this route. No justification required.

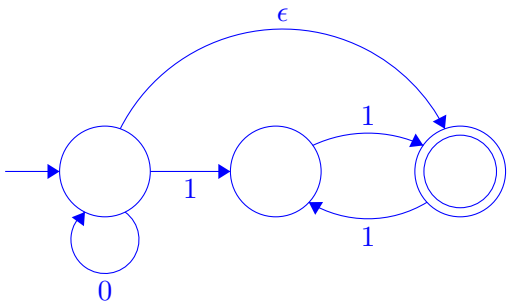
   Example: 011 is a string for which an accepting branch of computation takes the top route. 0111 is not (a branch travels from a to g using the top route, but has input left unread so it dies and does not accept). 11 is also not (there is an accepting branch for this string that takes the bottom route, but not the top route.)

   (a) Regular expression for top route: $0^+(11)^+$.

   (b) Regular expression for middle route: $0^*(\epsilon \cup 11)$ or $0^* \cup 0^*11$.

   (c) Regular expression for bottom route: $(11)^*$.

3. (5 points) This NFA is overly complicated. Draw the state diagram of an equivalent NFA (one that accepts exactly the same set of strings) that uses at most 3 states (partial credit for 4 or 5-state NFAs). [Hint: A 3-state NFA must recognize a relatively simple language. If you're stuck, test many short strings and see if the set of accepting short strings is consistent with a simple language. If you have a working NFA with greater than 3 states, can you remove ϵ-transitions or condense states near the start or the end while leaving the function of the NFA unchanged?]



State the language of strings your new NFA recognizes and explain in a few sentences why the new NFA recognizes the same set of strings as the old NFA.

The pictured NFA recognizes the language $0^*(11)^*$. The language recognized by the old NFA is the union of the three regular expressions for each route, which turns out to be the same. $0^*(11)^*$ matches all strings generated by $0^+(11)^+$, 0^*, 0^*11, and $(11)^*$. Likewise, every string generated by $0^*(11)^*$ matches one of these four regular expressions. To see this, observe that the only strings in $0^+(11)^+ \setminus 0^*(11)^*$ are those that match 0^* or $(11)^*$.

Note: It's easy to get this NFA slightly wrong. Your NFA should accept ϵ, 0, and 11, and it shouldn't accept any string that contains 0's after 1's.

Rationale: The purpose of this question is to test your skill evaluating NFAs on strings, figuring out which set of strings an NFA accepts, and designing efficient NFAs that recognize a given language.

References: Sipser pp. 47-53 (NFAs), the Lightning Review on NFAs linked on the website, the Bonus Review on ϵ-transitions, and the Lightning Review on regular expressions. See also HW1 problem 3 and HW2 problem 1.10.

2 Problem 2 (9 points)

For this question, consider the following context free grammar G:

$$S \to 111A3C \mid A!$$
$$A \to 111A3 \mid B$$
$$B \to 2B11 \mid \#$$
$$C \to !$$

G has a total of 7 production rules (two for S, two for A, two for B and two for C.) It has three variables ($V = \{S, A, B, C\}$) and five terminals ($\Sigma = \{1, 2, 3, !, \#\}$).

1. (3 points) Which of the following six terminal strings does this CFG generate? No justification required. $!, \#!, 1112\#113!, 1113\#112!, 111111\#33!, 111222\#1111113!$.

   The strings in the language are $\#!$, $1112\#113!$, $111111\#33!$, and $111222\#1111113!$.

2. (3 points) Describe the language recognized by this CFG using either set notation or 1-2 sentences. No justification required, but if you are unsure, include your reasoning for partial credit. [Hint: testing strings is never a bad idea.]

   (Example of a language written in set notation: $L = \{0^n 1^n \mid n \geq 0\}$. Example of sentences describing the same language: "L is the language of all strings consisting of a substring of 0's followed by a substring of 1's, where both substrings have the same length. It includes the empty string $\epsilon = 0^0 1^0$.")

   $$L(G) = \{1^{3k} 2^m \# 1^{2m} 3^k ! \mid k, m \geq 0\}.$$

3. (3 points) This CFG is overly complicated. Write an equivalent CFG **with at most 5 production rules** that generates the same language and a short explanation of why your CFG generates the language you specified in question 2.2.

   $$S \to A!$$
   $$A \to 111A3 \mid B$$
   $$B \to 2B11 \mid \#$$

   In the original CFG, the rules $S \to 111A3C$ and $C \to !$ are redundant. A first simplification is to remove C and replace the first rule with $S \to 111A3!$. Then, we might recognize that any string derived using $S \to 111A3!$ can be derived using $S \to A!$ and $A \to 111A3$ instead.

Rationale: The purpose of this question is to test your skill deriving strings using CFGs, recognizing what language a CFG generates and building CFGs.

References: Sipser p. 102, Lightning Review 5 (CFG definition and deriving strings), Sipser p.105 (figuring out the language of a CFG), and Sipser p.106-107 (tips for building CFGs). See also HW3 problem 1.

3 Problem 3 (8 points)

Use the context-free pumping lemma to show that the language

$$A = \{0^{2n}1^n \text{ or } 0^n1^n0^n \text{ or } 1^n0^{2n} \mid n \geq 0\}$$

is not context free. [Hint: Assume for contradiction that A is context-free, write down what the pumping lemma tells you under your assumption, choose a contradiction string *carefully*, and find a contradiction.]

The language A can be thought of as the union of three languages: $0^{2n}1^n$, $0^n1^n0^n$, and 1^n0^{2n}, each for any $n \geq 0$. Each includes two substrings with n 0's and one substring with n 1's, but only the sub-language containing strings of the form $0^n1^n0^n$ is non-context-free! This will make it important to choose a contradiction string that matches this pattern.

We assume for contradiction that A is a context-free language, in which case the CFPL applies. The CFPL tells us that there exists some number p such that for any string $s \in A$ of length $|s| \geq p$, s can be divided into five substrings $s = uvxyz$ satisfying

1. uv^ixy^iz for any integer $i \geq 0$,
2. $|vxy| \leq p$, and
3. $|vy| \geq 0$.

We choose the contradiction string $s = 0^p1^p0^p$. $|s| \geq p$ and $s \in L$, so if A is context-free the conditions above should apply to s.

First, we observe that Condition 2 implies that $|vxy|$ can overlap at most two of our three substrings 0^n, 1^n, and 0^n. (Overlapping with all three substrings would require at least $p + 2$ characters.) Furthermore, Condition 3 implies that v or y must contain at least one character. Consider the following two subcases:

1. v and y contain at most one type of character each. In this case, uv^2xy^2z contains three substrings of uneven length, as we increase the length of at least one and at most two of the substrings.
2. v or y contains at least one 0 and at least one 1. In this case, uv^2xy^2z repeats a string in 0^+1^+ or 1^+0^+ and mixes up digits, creating five distinct substrings of 0's and 1's.

Thus neither subcase satisfies Condition 1. We conclude that no division of s satisfies the conditions of the CFPL, and thus A cannot be context-free.

Note 1: Both $0^{2p}1^p$ and 1^p0^{2p} can be pumped, so they're not good contradiction strings. To see this, consider

$$s = 0^{2p}1^p = uvxyz$$

where $u = 0^{2p-2}$, $v = 00$, $x = \epsilon$, $y = 1$, and $z = 1^{p-1}$. This meets all three conditions and generates new strings in the language when pumped up or down.

Rationale: The goal of this question is to test your ability to use the CFPL.

References: References: The CFPL is stated in Sipser p.125, with examples on pp. 128-129. See also the example video on the CFPL. See also HW4 problem 2.

4 Problem 4 (8 points)

Write high-level descriptions of Turing Machines for the following languages. You don't need to worry about efficiency or time complexity. For a refresher on what counts as a high-level description, consult HW4 Problem 4 (and solutions). In short, if your description fully specifies a program that deals with finite data objects (or behaves carefully when dealing with potentially infinite objects), this is fine.

1. (4 points) Define a Turing Machine that decides the language

$$B = \{\langle D_1, D_2\rangle \mid D_1, D_2 \text{ are DFAs such that } L(D_1) = \overline{L(D_2)}\}.$$

You may assume that accepting pairs of DFAs D_1 and D_2 are defined over the same alphabet Σ and that the complement is determined relative to the universe Σ^*. (That is, $\overline{L(D_1)} = \Sigma^* \setminus L(D_1)$.)

There are a few strategies that work here. They include:

   (a) We could build a TM that uses our procedures for building DFAs that accept the union and intersection of the languages accepted by other DFAs to build DFAs that accept $L(D_1) \cap L(D_2)$ and $L(D_1) \cup L(D_2)$. We could then check if $L(D_1) \cap L(D_2) = \emptyset$ and $L(D_1) \cup L(D_2) = \Sigma^*$, either using deciders we have built previously or by examining state diagrams. This is equivalent to showing that $L(D_1) = \overline{L(D_2)}$.

   (b) We could flip the accept states on D_2 to create a DFA that accepts $\overline{L(D_2)}$, then use our decider for EQ_{DFA} to see if $L(D_1) = \overline{L(D_2)}$.

2. (4 points) Define a Turing Machine that recognizes the language

$$C = \{\langle M\rangle \mid M \text{ is a TM that halts and accepts with '42' on the tape on at least one input}\}.$$

(Example encoded Turing Machines in this language: the TM that writes '42' and accepts on every input string, the TM that writes '42' and accepts when given the input string 01101 and loops forever on any other input. Examples not in this language: TMs that have '42' on the tape at some point during execution but halt with something else on the tape.)

If you prefer, you can assume that accepting TMs are defined over $\Sigma = \{0, 1\}$ for simplicity.

Let $s_1, s_2, s_3, \ldots, = \epsilon, 0, 1, 00, 01, \ldots$ be a list enumerating all strings in Σ^*.

Our TM M_C that recognizes C will operate as follows:

M_C = "On input $\langle M\rangle$:

   (a) For $i = 1, 2, 3, \ldots$:

      i. Simulate M on strings s_1 through s_i for i steps per simulation.

      ii. If any simulation halts and accepts with '42' on the tape, halt and accept."

This procedure ensures that, if M halts and accepts with '42' on the tape on *any* input, we will eventually simulate M on that input for enough steps to reach the end of the computation and accept. (Our TM will run forever if given an encoded TM not in the language.)

Note: Here, it's important to make sure we don't get caught in an infinite loop if we should accept. For example, perhaps M halts with 42 on the tape on input '0110011', but runs forever on some earlier input. This is why it is important to set up the TM so that our simulations run for a finite number of steps only.

Note 2: The language C is intended to include all Turing Machines that, on some input w, accept and leave '42' on the tape.

However, C might be interpreted as "all Turing Machines M such that $M(42)$ (or M on some string containing 42 as a substring) halts." We won't take off points for this interpretation if the machine is implemented correctly.

Rationale: The goal of this question is to test your ability to think about Turing Machines as general-purpose computers that can execute algorithms for solving complex decision problems and simulate other automata.

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. See also Homework 4 problem 4, and Homework 5 problem 1, and Example 7 on the course website.

5 Problem 5 (6 points)

In the previous problem, you wrote a TM that recognizes the language

$$C = \{\langle M \rangle \mid M \text{ is a TM that halts and accepts with '42' on the tape on at least one input}\}.$$

1. (5 points) Show that C is undecidable by reduction. (Don't use Rice's theorem for this question.) [Hint: recall that the structure of such a proof is as follows: If I *could* decide C, I could use the decider as a subroutine to decide another language I know to be undecidable. This is a contradiction.]

   We will show that C is undecidable by reducing from

   $$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts } w\}.$$

   Assume for contradiction that C is decidable, and that some Turing Machine T decides C. We'll use T as a subroutine to build the following decider S for A_{TM}:

   S = "On input $\langle M, w \rangle$:

   (a) Write down the description of a Turing Machine K that (1) always ignores its input, (2) simulates $M(w)$, and (3) if $M(w)$ halts and accepts, writes '42' on the tape and accepts.

   (b) Simulate $T(\langle K \rangle)$. Accept if T accepts and reject otherwise."

   S always halts because its two functions (writing down the description of K and simulating the decider T on a certain input) are both finite procedures. Note that S doesn't actually *simulate/run* K, which might result in an infinite loop.

   K halts and accepts with '42' on the tape (on all inputs) if and only if $M(w)$ accepts. Thus $T(\langle K \rangle)$ accepts if and only if $\langle M, w \rangle \in A_{TM}$, so S decides A_{TM}. This is a contradiction, as we know that A_{TM} is undecidable.

2. (1 point) is the complement of C, $\overline{C}$, recognizable? Justify your answer in a sentence or two.

   We know that a language L is decidable if and only if L and $\overline{L}$ are recognizable. Since we know (Problems 4.2 and 5.1 on this test) that C is recognizable but not decidable, $\overline{C}$ must not be recognizable.

Rationale: The goal of this question is to practice showing undecidability by reducing one problem to another problem, as well as thinking about undecidability and unrecognizability.

References: See Sipser pp. 215-220, in which several languages are shown to be undecidable by reducing from the halting problem, as well as Homework 5 problem 3 and Example 8 on the course website. For the second part, see Sipser pp. 209-210, which talks about the relationship between undecidability and unrecognizability.

Thank you for participating in this course! We hope you succeeded in what you set out to achieve. Please don't be a stranger: feel free to reach out to us now and in the future. Have a great rest of the summer.

- Tim & the course staff