What can you do with regular expressions? What are the limits of regular languages? And how does the material on discrete structures from the first half of this quarter come into play in this latter half on automata and computation? In this problem set, you'll explore the answers to these questions along with their practical consequences.

As always, please feel free to drop by office hours, ask on Piazza, or send us emails if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, November 15th at 2:30PM
Problem One: Designing Regular Expressions

Below are a list of alphabets and languages over those alphabets. For each language, write a regular expression for that language. Please use our online tool to design, test, and submit your regular expressions. Typed or handwritten solutions will not be accepted. To use it, visit the CS103 website and click the “Regex Editor” link under the “Resources” header. If you submit in a pair, please tell us in your GradeScope submission who submitted your answers to this question. Also, as a reminder, please test your submissions thoroughly, since we’ll be grading them with an autograder.

As with NFAs, there’s no simple way to start with a regex for a language \( L \) and to turn it into a regex for \( \overline{L} \). The general pattern here is that a file path consists of a series of directory or file names separated by slashes. That path might optionally start with a slash, but isn’t required to. However, you can’t have two consecutive slashes. Let \( \Sigma = \{ a, / \} \). Write a regular expression for \( L = \{ w \in \Sigma^* \mid w \) represents a walk with your dog on a Unix-style system \} \). For example, /aaa/a/aa ∈ L, / ∈ L, a ∈ L, /a/a/a/ ∈ L, and aaa/ ∈ L, but //a/a/a/ ∈ L, a/a/a/ ∈ L, and ε ∈ L.

Fun fact: this problem comes from former CS103 instructor Amy Liu, who fixed a bug in industrial code that arose when someone wrote the wrong regex for this language. Oops.

Write a regular expression for the complement of the language from part (i) of this problem.

Suppose you are taking a walk with your dog on a leash of length two. Let \( \Sigma = \{ y, d \} \) and let \( L = \{ w \in \Sigma^* \mid w \) represents a walk with your dog on a leash where you and your dog both end up at the same location \}. For example, we have yydddyy ∈ L because you and your dog are never more than two steps apart and both of you end up four steps ahead of where you started; similarly, ddydyy ∈ L. However, yyyyddd ∈ L, since halfway through your walk you’re three steps ahead of your dog; ddyd ∈ L, because your dog ends up two steps ahead of you; and ddyddy ∈ L, because at one point your dog is three steps ahead of you. Write a regular expression for \( L \).

Note that, unlike Problem Set Six, you and your dog must end at the same position.

Let \( \Sigma = \{ M, D, C, L, X, V, I \} \) and let \( L = \{ w \in \Sigma^* \mid w \) is number less than 2,000 represented in Roman numerals \}. For example, CMXCIX ∈ L, since it represents the number 999, as are the strings L (50), VIII (8), DCLXVI (666), CXXXVII (137), CDXII (412), and MDCLXVI (1,618). However, we have that VIII ≠ L (you’ll never have four I’s in a row; use IX or IV instead), that MM ≠ L (it’s a Roman numeral, but it’s for 2,000, which is too large), that MX ≠ L (this isn’t a valid Roman numeral), and that IM ≠ L (the notation of using a smaller digit to subtract from a larger one only lets you use I to prefix V and X, or X to prefix L and C, or C to prefix D and M). The Romans didn’t have a way of expressing 0, so to make your life easier we’ll say that ε ∈ L and that the empty string represents 0. (Oh, those silly Romans.) Write a regular expression for \( L \).

(As a note, we’re using the “standard form” of Roman numerals. You can see a sample of numbers written out this way via this link.)

* In some cases you technically can have multiple consecutive slashes, but we’ll ignore that for now.
Problem Two: Finite Languages

A language $L$ is called finite if $L$ contains finitely many strings (that is, $|L|$ is a natural number).

Given a finite language $L$, explain how to write a regular expression for $L$. Briefly justify your answer; no formal proof is necessary. This shows that all finite languages are regular.

*Watch for edge cases!*

Problem Three: State Elimination

The state elimination algorithm gives a way to transform a DFA or NFA into a regular expression. It's a beautiful algorithm once you get the hang of it.

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | w$ has an even number of a's and an even number of b's\}. Below is a finite automaton for $L$ that we’ve prepared for the state elimination algorithm by adding in a new start state $q_{\text{start}}$ and a new accept state $q_{\text{acc}}$:

![Finite Automaton Diagram]

We’d like you to use the state elimination algorithm to produce a regular expression for $L$.

i. Run two steps of the state elimination algorithm on the above automaton. Specifically, first remove state $q_1$, then remove state $q_2$. Show your result at this point. 

*Remember that to eliminate a state $q$, you should identify all pairs of states $q_{\text{in}}$ and $q_{\text{out}}$ where there’s a transition from $q_{\text{in}}$ to $q$ and from $q$ to $q_{\text{out}}$, then add shortcut edges from $q_{\text{in}}$ to $q_{\text{out}}$ to bypass state $q$. Remember that $q_{\text{in}}$ and $q_{\text{out}}$ may be the same state. To help you check your work: there are four such pairs for state $q_1$. If you’ve done everything properly, at the end of this stage, no transitions in your automaton should have Kleene stars on them.*

ii. Finish the state elimination algorithm by first eliminating $q_3$, then $q_0$. Show your result.

*This is where you’ll start seeing Kleene stars.*

iii. Submit your resulting regular expression through our regular expression tool along the lines of what you did in Problem One. If you’re working in a pair, tell us on GradeScope who submitted it.

*Not sure whether you have the right answer? Test your result thoroughly!*

Something we strongly recommend: look at the regular expression you ended up with in part (ii) of this problem. How does it work? That is, how does that regular expression match all and only strings with an even number of a’s and an even number of b’s? The particular mechanism it uses is quite clever, and it’s amazing that something that sophisticated dropped out of a mechanical transformation on a DFA! You don’t need to submit an answer to this question, but you should aim to have a rock-solid understanding.
Problem Four: At Least Three Point Five

The Myhill-Nerode theorem is a powerful tool for finding nonregular languages, but it can take some adjusting to get used to.

Let $\Sigma = \{a, b\}$ and consider the following language:

$$L = \{ w \in \Sigma^* \mid \text{there are at least as many a’s as b’s in } w \}.$$

This language $L$ isn’t regular; make sure you have an intuition for why this is.

Below is an attempted proof that $L$ isn’t regular. Although the claim it’s proving is indeed true, this proof contains an error that renders it incorrect.

⚠️ Theorem: $L$ is not a regular language.

(Incorrect!) Proof: Consider the set $S = \{ a^n \mid n \in \mathbb{N} \}$. We will prove that $S$ is infinite and that $S$ is a distinguishing set for $L$.

To see that $S$ is infinite, note that it contains one string for each natural number.

To see that $S$ is a distinguishing set for $L$, consider any distinct strings $a^n, a^m \in S$, and assume without loss of generality that $m > n$. Then $b^m a^n \in L$ because this string contains the same number of a’s and b’s, but $b^n a^m \notin L$ because it contains $m$ b’s and $n$ a’s and $m > n$. Therefore, we see that $a^n \not\equiv_L a^m$.

Because $S$ is a distinguishing set for $L$ that contains infinitely many strings, by the Myhill-Nerode theorem we see that $L$ is not regular. ■

⚠️ Alas, this proof contains an error.

i. What’s wrong with the above proof? Be as specific as possible.

The best way to identify a flaw in a proof is to point to a specific claim that’s being made that’s not true or not properly substantiated and to explain why.

Here’s another attempt at a proof that $L$ isn’t regular:

⚠️ Theorem: $L$ is not a regular language.

(Incorrect!) Proof: Consider the set $S = \{ a^n \mid n \in \mathbb{N} \}$. We will prove that $S$ is infinite and that $S$ is a distinguishing set for $L$.

To see that $S$ is infinite, note that it contains one string for each natural number.

To see that $S$ is a distinguishing set for $L$, consider any distinct strings $a^n, a^{n+1} \in S$. Then $a^{n+1} b^{n+1} \in L$ because this string contains the same number of a’s and b’s, but $a^{n} b^{n+1} \notin L$ because it contains $n+1$ b’s but only $n$ a’s. Therefore, we see that $a^{n+1} \not\equiv_L a^n$.

Because $S$ is a distinguishing set for $L$ that contains infinitely many strings, by the Myhill-Nerode theorem we see that $L$ is not regular. ■

Sadly, this proof is also wrong:

ii. What’s wrong with the above proof? Be as specific as possible.

Fun fact: the error in this proof is one of the most common mistakes we see people make in Myhill-Nerode proofs.
Problem Five: Embracing the Braces

Let $\Sigma$ be an alphabet containing two characters, the open curly brace character $\{$ and the close curly brace character $\}$. Consider the following language over $\Sigma$:

$$L_1 = \{ w \in \Sigma^* \mid w \text{ is a string of balanced curly braces } \}$$

For example, we have $\{\} \in L_1$, $\{\{\}\} \in L_1$, $\{\{\}\} \in L_1$, and $\{\{\}\}\{\{\}\}\{\}\} \in L_1$, but $\} \notin L_1$, $\{\} \notin L_1$, and $\{\}\} \notin L_1$. This question explores properties of this language.

i. Prove that $L_1$ is not a regular language. One consequence of this result – which you don’t need to prove – is that real-world languages that support some sort of nested structures, such as most programming languages and HTML, aren’t regular and so can’t be parsed using regular expressions.

As a first step, ask yourself: if you were reading an input string from left to right, what information would you have to keep track of? The Myhill-Nerode theorem asks you to find a distinguishing set of infinite size. Based on that, find two distinguishable strings by finding two strings that have different “information” associated with them, where, here, “information” corresponds to what you found in the first step.

Once you’ve done that, find a third string distinguishable from the previous two strings. It should correspond to some different piece of “information.” Once you’ve done this, keep adding in more strings until you’ve spotted a pattern that lets you define an infinite distinguishing set.

Let’s say that the nesting depth of a string of balanced braces is the maximum number of unmatched open braces at any point inside the string. For example, the string $\{\}\{\}\}\{\}\$ has nesting depth three, the string $\{\}\{\}\}\{\}\}$ has nesting depth two, and the string $\}$ has nesting depth zero.

Consider the language $L_2 = \{ w \in \Sigma^* \mid w \text{ is a string of balanced curly braces with nesting depth at most } 4 \} \}$. For example, $\{\} \in L_2$, $\{\{\}\} \in L_2$, and $\{\{\}\}\{\}\} \in L_2$, but $\{\{\}\}\{\}\} \notin L_2$ because although it’s a string of balanced curly braces, the nesting goes five levels deep.

ii. Design a DFA for $L_2$, showing that $L_2$ is regular. A consequence of this result is that while you can’t parse all programs or HTML with regular expressions, you can parse programs with low nesting depth or HTML documents without deeply-nested tags using regexes. Please submit this DFA using the DFA editor on the course website and tell us on GradeScope who submitted it.

iii. Look back at your proof from part (i) of this problem. Imagine that you were to take that exact proof and blindly replace every instance of “$L_1$” with “$L_2$.” This would give you a (incorrect) proof that $L_2$ is nonregular (which we know has to be wrong because $L_2$ is indeed regular.) Where would the error be in that proof? Be as specific as possible.

Again, you should be able to point at a specific spot in the proof that contains a logic error and explain exactly why the statement in question is not true or not supported by the preceding statements. If you can’t do this, it likely means you have an error in your proof from part (i)!

Intuitively, regular languages correspond to problems that can be solved using only finite memory. Make sure you understand why, given that intuition, $L_1$ “ought to” be nonregular while $L_2$ “ought to be” regular. This sort of intuition will be extremely helpful going forward.
Problem Six: State Lower Bounds

The Myhill-Nerode theorem we proved in lecture is actually a special case of a more general theorem about regular languages. This problem explores how to generalize that result.

i. Let \( L \) be a language over \( \Sigma \) and let \( S \) be a distinguishing set for \( L \). Prove that if \( S \) is finite (that is, \( |S| \) is a natural number), then any DFA for \( L \) must have at least \( |S| \) states. (You sometimes hear this referred to as lower-bounding the size of any DFA for \( L \).)

The next problem on this problem set talks about writing proofs like these using the formal 5-tuple definition of a DFA. We are not expecting you to do this here; feel free to structure your proof for this part of the problem along the lines of the proofs on DFAs that you saw in lecture.

According to old-school Twitter rules, all tweets need to be 140 characters or less. Let \( \Sigma \) be the alphabet of characters that can legally appear in a tweet (which includes most scripts from most parts of the world, plus things like emojis, mathematical symbols, etc.). Then, consider the following language:

\[
TWEETS = \{ w \in \Sigma^* \mid |w| \leq 140 \}.
\]

This is the language of all legal tweets. (We’ll count the empty string as a legal tweet for the purposes of this problem. Many tweets would be improved by replacing them with the empty string.) The good news is that this language is regular. The bad news is that any DFA for it has to be pretty large.

ii. Using your result from part (i), prove that any DFA for \( TWEETS \) must have at least 142 states.

It might be easier to tackle this problem if you consider replacing 140 and 142 with some smaller numbers (say, 2 and 4) to build up an intuition. And work backwards – what will you need to do to invoke part (i)?

iii. Define a 142-state DFA for \( TWEETS \) using the formal 5-tuple definition of a DFA. Briefly explain how your DFA works. No formal proof is necessary.

Again, this might be a lot easier to do if you first reduce 140 and 142 to 2 and 4, respectively, and see what you come up with. Start by drawing out what the DFA would look like, then think about how you’d formalize your idea as a 5-tuple.

As with the 5-tuple question from Problem Set Six, please define your transition function \( \delta \) by writing out an expression of the form \( \delta(q, a) = _____ \), filling in the blank appropriately. Feel free to use piecewise functions, like you did on Problem One of PS4.

Your results from parts (ii) and (iii) show that the smallest possible DFA for \( TWEETS \) has exactly 142 states. This approach to finding the smallest object of some type – using some theorem to prove a lower bound (“we need at least this many states”) combined with a specific object of the given type (“we need at most this many states”) is a common strategy in algorithm design and computational complexity theory. If you take classes like CS161, CS254, etc., you’ll likely see similar sorts of approaches!
Problem Seven: The Extended Transition Function

As you saw on Problem Set Six, formally speaking, a DFA is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\). You used the 5-tuple definition to pin down edge cases of DFAs. But we can also use this formal definition to rigorously define concepts about automata that, at this point, we’ve only discussed at a high-level.

Let \(D = (Q, \Sigma, \delta, q_0, F)\) be a DFA. We’re going to define a function \(\delta^* : \Sigma^* \rightarrow Q\) called the extended transition function of \(D\). Intuitively, the function \(\delta^*\) takes as input a string and outputs what state that string would end up in if run through the DFA \(D\). The function \(\delta^*\) is defined, recursively, as follows:

- **Base case:** \(\delta^*(\varepsilon) = q_0\).
- **Recursive case:** If \(w \in \Sigma^*\) and \(a \in \Sigma\), then \(\delta^*(wa) = \delta(\delta^*(w), a)\).

That’s quite a mouthful, but there’s a nice explanation for what’s going on here.

i. Explain \(\delta^*\)’s base case in plain English and why that makes sense given what \(\delta^*\) represents. Then, explain \(\delta^*\)’s recursive case in plain English and why that makes sense given what \(\delta^*\) represents.

The notation here is dense, so proceed slowly. What’s the input to the function \(\delta^*\) in this case? What are \(w\) and \(a\)? You may want to draw out an actual DFA and try expanding out \(\delta^*\) for a couple of strings.

ii. Formally speaking, we define \(\mathcal{L}(D) = \{ w \in \Sigma^* \mid \delta^*(w) \in F \}\). Explain how this mathematical definition accords with the plain English one we’ve been using thus far.

That’s a lot of symbols! Write out what each of them mean.

iii. Let \(D = (Q, \Sigma, \delta, q_0, F)\) be a DFA, and let \(x, y \in \Sigma^*\) be two strings where \(\delta^*(x) = \delta^*(y)\). Prove that, for any string \(z \in \Sigma^*\), we have \(\delta^*(xz) = \delta^*(yz)\).

Your proof will need to rely on the definition of \(\delta^*\). Since \(\delta^*\) is defined recursively, what style of proof do you think you might want to use here?

We are expecting you to write a formal proof here that uses the definition of the \(\delta^*\) function. As you’re reasoning through why this result is true, feel free to think at a high level about ideas like “we run the DFA on the string \(xz\)” and “the DFA ends in an accepting state.” However, your written answer should be expressed purely using the 5-tuple definition of a DFA, the formal definition of \(\delta^*\), the formal definition of \(\mathcal{L}(D)\), etc. The underlying rationale is that this problem is all about formalizing our intuitions about how DFAs operate by using those rigorous definitions, and so the written proof should be done purely using those definitions.

In lecture, we used this theorem about distinguishable strings to prove certain languages aren’t regular:

**Theorem:** Let \(x\) and \(y\) be strings where \(x \not\equiv_L y\). Then \(x\) and \(y\) cannot end up in the same state after being run through any DFA for the language \(L\).

We can recast this theorem in terms of the \(\delta^*\) function that we just defined above:

**Theorem (Formalized):** Let \(x\) and \(y\) be strings where \(x \not\equiv_L y\). Then for any DFA \(D\) for \(L\), if \(\delta^*\) is the extended transition function for \(D\), we have \(\delta^*(x) \neq \delta^*(y)\).

Now that you’re equipped with the formal definition of \(\delta^*\), you can rigorously prove the above statement.

iv. Prove the formalized theorem (the second one). Since the goal is to write a rigorous proof of the theorem, you should not cite the informal one from lecture as part of your proof.

*Use your intuition about DFAs to think through his one, but as with part (iii) of this problem, use the 5-tuple definition of a DFA and the formal definition of the extended transition function in your proof. For example, you should not use phrases like “run the DFA on \(x\)” or “the DFA ends up in an accepting state,” since you have more formal notation at your disposal.*

*Once you’ve finished, take a minute to marvel at the fact that you’re able to read (and prove!) statements like these. Not bad for seven weeks!*
Problem Eight: Regular Languages and Equivalence Relations

Throughout this problem set you’ve been working with the idea that we can take a language $L$ over some alphabet $\Sigma$, then work with its distinguishability relation $\not\equiv_L$. A closely related binary relation is the \textit{indistinguishability} relation for $L$, denoted $\equiv_L$. It’s also a binary relation over $\Sigma^*$, and its definition is the negation of the one for distinguishability:

$$ x \equiv_L y \iff \forall w \in \Sigma^*. (xw \in L \iff yw \in L). $$

Amazingly, this is always an equivalence relation, regardless of what $L$ is!

i. Prove that if $L$ is a language over $\Sigma$, then $\equiv_L$ is an equivalence relation over $\Sigma^*$.

You’ve seen how to prove that a binary relation is an equivalence relation, how to prove biconditionals, and how to work with universally-quantified statements. This proof is going to look very similar to the proofs that you did on Problem Set 3, with the only notable exception being that it involves proofs with strings. So, for example, don’t include first-order logic notation in your proof, make sure to properly scope and introduce your variables, etc.

Let’s make this more concrete. Let $\Sigma = \{ a, b \}$ and consider the language $M = \{ w \in \Sigma^* \mid \text{the number of b's in } w \text{ is congruent to 1 modulo 5 or to 3 modulo 5 } \}$. For example, $aba \in M$, $baaabaaab \in M$, and $bbbbbb \in M$, but $aa \notin M$ and $abba \notin M$.

ii. List all the equivalence classes of $\equiv_M$, and give a system of representatives for $\equiv_M$. Express your answer using formal mathematical notation. No justification is required.

Remember that an equivalence relation should partition all the elements of its underlying set, so you should find that every string $w \in \Sigma^*$ belongs to exactly one of the equivalence classes.

You might have noticed that each equivalence class of $\equiv_M$ either consists of a bunch of strings not in $M$ or of a bunch of strings that are in $M$. That’s not a coincidence!

iii. Let $L$ be a language over some alphabet $\Sigma$ and let $x \in \Sigma^*$ be some string. Prove that either \textit{every} string in $[x]_{\equiv_L}$ is in $L$ or that \textit{no} strings in $[x]_{\equiv_L}$ are.

The \textit{index} of an equivalence relation $R$, denoted $I(R)$, is the number of equivalence classes of $R$. This quantity might be finite, or it might be an infinite cardinality like $\aleph_0$, or even one of the infinities bigger than that. Armed with the idea of an index, we can state a powerful theorem about finite automata:

\textbf{Theorem:} If $L$ is a language over $\Sigma$, then every DFA for $L$ must have at least $I(\equiv_L)$ states.

In other words, there’s a connection between the number of equivalence classes of a particular binary relation and the minimum sizes of DFAs for that language!

iv. Prove the above theorem. Feel free to use the \textbf{axiom of choice}, which says that every equivalence relation has at least one system of representatives.

Proving this theorem is mostly an exercise in connecting together ideas you’ve seen used in other places. Think about the relationship between indices and systems of representatives, between distinguishability and indistinguishability, and between what you’re doing here and what you’ve done earlier on this problem set.

There’s a lovely intuition for this theorem. You can think of the indistinguishability relation for a language $L$ as pinning down the idea “a DFA for $L$ can’t tell the difference between these two strings.” If you think back to our intuition behind DFA design – build a DFA where each state keeps track of some different piece of information – then you can think of $I(\equiv_L)$ as capturing the number of different pieces of information you’d need to remember. The theorem then says that if you want to build a DFA for a language $L$, you’ll need at least one state per piece of information.
We’ve got two wonderful Optional Fun Problems here for you to work through. Each of these problems are questions we considered putting onto the main problem set but decided not to include to keep the length down. So feel free to work through one or both of these, and submit answers to at most one of them. You know the drill – if you submit answers to more than one problem, we’ll choose which one to grade based on a whim.

**Optional Fun Problem One: Generalized Fooling Sets**

In Problem Seven, you used distinguishability to lower-bound the size of DFAs for a particular language. Unfortunately, distinguishability is not a powerful enough technique to lower-bound the sizes of NFAs. In fact, it's in general quite hard to bound NFA sizes; there's a $1,000,000 prize for anyone who finds a efficient algorithm (for some precise definition of “efficient”) that, given an arbitrary NFA, converts it to the smallest possible equivalent NFA!

Although it's generally difficult to lower-bound the sizes of NFAs, there are some techniques we can use to find lower bounds on the sizes of NFAs. Let $L$ be a language over $\Sigma$. A \textit{generalized fooling set} for $L$ is a set $\mathcal{F} \subseteq \Sigma^* \times \Sigma^*$ is a set with the following properties:

- For any $(x, y) \in \mathcal{F}$, we have $xy \in L$.
- For any distinct pairs $(x_1, y_1), (x_2, y_2) \in \mathcal{F}$, we have $x_1y_2 \notin L$ or $x_2y_1 \notin L$ (this is an inclusive OR.)

Prove that if $L$ is a language and there is a generalized fooling set $\mathcal{F}$ for $L$ that contains $n$ pairs of strings, then any NFA for $L$ must have at least $n$ states.

**Optional Fun Problem Two: Why Finite?**

A \textit{deterministic infinite automaton}, or DIA, is a generalization of a DFA in which the automaton has infinitely many different states. Formally speaking, a DIA is given by the same 5-tuple definition as a DFA, except that $Q$ must be an infinite set. Since DIAs have infinitely many states, they’re mostly an object of purely theoretical study. You couldn’t actually build one in the real world.

Prove that if $L$ is an arbitrary language over some alphabet $\Sigma$, then there is a DIA that accepts $L$ (that is, the DIA accepts every string in $L$ and rejects every string not in $L$.) To do so, show how to start with a language $L$, formally define a 5-tuple corresponding to a DIA for $L$, then formally prove that that DIA accepts all and only the strings in $L$ by using the formal definition of what the language of an automaton is.