What are the limits of regular languages? What lies beyond finite automata and regular expressions in the computability landscape? In this problem set, you'll explore the answers to these questions along with their practical consequences.

Note that some of the questions will require you to submit a context-free grammar you create via our online editors. Please follow the instructions in the handout to access these tools and submit your work. There is no need to copy your grammar back into your written submission as we will look only at your online submission for grading these questions.

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!
1 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:

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

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.

**Proof:** Consider the set $S = \{a^n \mid n \in \mathbb{N}\}$. This set is infinite because it contains one string for each natural number. We will prove that any two distinct strings in $S$ are distinguishable relative to $L$. To do so, consider any distinct strings $a^m, a^n \in S$, and assume without loss of generality that $m > n$. Then $b^m a^m \in L$ because this string contains the same number of $a$’s and $b$’s, but $b^m a^n \notin L$ because it contains $m$ $b$’s and $n$ $a$’s and $m > n$. Therefore, we see that $a^m \not\equiv_L a^n$. This means that $S$ is an infinite set of strings that are pairwise distinguishable relative to $L$. Therefore, by the Myhill-Nerode theorem, $L$ is not regular. □

What’s wrong with this 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.*
2 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 $\{ \} \in L_1$, $\{\{\}\} \in L_1$, $\{\{\}\} \in L_1$, $\varepsilon \in L_1$, 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 most languages that support some sort of nested parentheses, such as most programming languages and HTML, aren’t regular and so can’t be parsed using regular expressions.

As with all problems involving nonregular languages, proceed with this one in stages. First, 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 an infinite set of strings that are all pairwise distinguishable, so try creating an infinite set of strings, one for each possible value that this information could take on.

Let’s say that the **nesting depth** of a string of balanced parentheses is the maximum number of unmatched open parentheses at any point inside the string. For example, the string $\{\{\}\} \} \} \}$ has nesting depth three, the string $\{\{\}\} \}$ has nesting depth two, and the string $\varepsilon$ 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 $\{\{\}\} \} \} \in L_2$ because although it’s a string of balanced curly braces, the nesting goes five levels deep. We won’t make you formally prove this, but the language $L_2$ is regular. (Great exercise: try designing a DFA for this language!)

ii. 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. After completing this problem, make sure you can explain to yourself why $L_1$ is nonregular while $L_2$ is regular without making reference to DFAs, NFAs, or the Myhill-Nerode theorem.
3 State Lower Bounds

The Myhill-Nerode theorem we proved in lecture is actually a special case of a more general theorem about regular languages that can be used to prove lower bounds on the number of states necessary to construct a DFA for a given language.

i. Let $L$ be a language over $\Sigma$. Suppose there’s a set $S$, which may be finite and which may be infinite, such that any two distinct strings $x, y \in S$ are distinguishable relative to $L$ (that is, $x \not\equiv_L y$ for any two strings $x, y \in S$ where $x \neq y$.) Prove that 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 $L$.)

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 and consider the following language:

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

This is the language of all legal tweets, assuming the empty string is a legal tweet. 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.*

Your results from parts (ii) and (iii) collectively establish 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 certainly can’t do worse than this”) 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!
4 The Extended Transition Function

As you saw on Problem Set Five, 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.

ii. 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.

iii. Explain why, formally speaking, we can define \(L(D) = \{w \in \Sigma^* \mid \delta^*(w) \in F\}\).

That’s a lot of symbols! Write out what each of them mean. Compare this definition against the one from lecture – see if you can account for why this definition has the same meaning.

iv. 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 \(z \in \Sigma^*\) of length \(n\), we have \(\delta^*(xz) = \delta^*(yz)\) for all \(n \in \mathbb{N}\).

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?

Make sure you have an intuition as to what you’re asked to prove here. After you peel back the layers of notation, you’re left with a nice statement about the behavior of DFAs that’s related to something from lecture. While you should use your intuition to guide you, your proof should specifically use the 5-tuple definition of a DFA and the formal definition of the extended transition function. For example, you should not include statements like “run the DFA on \(xz\)” or “the DFA ends in an accepting state,” since you now have more formal notation available to express those ideas.

(This problem is continued on the next page)
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.

v. Using your result from part (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 (iv) 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.*

*Take a look at the proof sketch we did in class of the first theorem and look at how we utilized the fact that \( x \not\equiv_L y \). Perhaps you can adopt a similar strategy here, but with this new formal definition of \( \delta^* \) ?

*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 six weeks!"
5 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 \( \neq_L \). A closely related binary relation is the 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:

\[
x \equiv_L y \text{ if } \forall w \in \Sigma^*. (xw \in L \leftrightarrow yw \in L).
\]

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 \( \Sigma^* \).

Remember that to conclude \( x \equiv_L y \), you have to prove both directions of the biconditional!

Whenever you see an equivalence relation, you should immediately start thinking about what its equivalence classes are. Doing so will usually tell you something interesting.

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

ii. How many equivalence classes does the \( \equiv_M \) relation have? Briefly describe what those equivalence classes are and give a system of representatives for \( \equiv_M \).

Need a refresher on systems of representatives? Check out Problem Set Three.

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 every string in \( [x]_{\equiv_L} \) is in \( L \) or that no strings in \( [x]_{\equiv_L} \) are.

The number of equivalence classes of an equivalence relation is called its index; the index of an equivalence relation \( R \) is denoted \( I(R) \). 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:

**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 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 very nice intuition for what this theorem says. 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.
6 Designing CFGs

For each of the following languages, design a CFG for that language. **Please use our online tool to design, test, and submit the CFGs in this problem.** To use it, visit the CS103 website and click the “CFG Editor” link under the “Tools” header. You should only have one member from each team submit your grammars; tell us who this person is when you submit the rest of the problems through GradeScope.

i. Given $\Sigma = \{a, b, c\}$, write a CFG for the language $\{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$. For example, the strings $aa$, $baac$, and $ccaabb$ are all in the language, but $aba$ is not.

ii. Given $\Sigma = \{a, b\}$, write a CFG for the language $\{w \in \Sigma^* \mid w \text{ is not a palindrome}\}$, the language of strings that are not the same when read forwards and backwards. For example, $aab \in L$ and $baabab \in L$, but $aba \notin L$, $bb \notin L$, and $\varepsilon \notin L$.

Don’t try solving this one by starting with the CFG for palindromes and making modifications to it. In general, there’s no way to mechanically turn a CFG for a language $L$ into a CFG for the language $\overline{L}$, since the context-free languages aren’t closed under complementation. However, the idea of looking at the first and last characters of a given string might be a good idea.

iii. Let $\Sigma$ be an alphabet containing these symbols:

$$\emptyset \quad \mathbb{N} \quad \{\} \quad \cup$$

We can form strings from these symbols which represent sets. Here’s some examples:

$$\emptyset \quad \{\emptyset, \mathbb{N}\} \cup \emptyset \cup \mathbb{N} \quad \{\emptyset\} \cup \mathbb{N} \cup \{\emptyset\} \quad \{\emptyset, \emptyset, \emptyset\} \cup \mathbb{N}$$

Notice that some of these sets, like $\{\emptyset, \emptyset\}$ are syntactically valid but redundant, and others like $\{\}$ are syntactically valid but not the cleanest way of writing things. Here’s some examples of strings that don’t represent sets or aren’t syntactically valid:

$$\varepsilon \quad \emptyset\{\emptyset\} \quad \emptyset\{\mathbb{N}\} \quad \{\emptyset\} \quad \emptyset\{\emptyset\} \cup \emptyset \cup \mathbb{N} \cup \{\emptyset\} \quad \{\emptyset, \emptyset, \emptyset\} \cup \mathbb{N}$$

Write a CFG for the language $\{w \in \Sigma^* \mid w \text{ is a syntactically valid string representing a set}\}$. When using the CFG tool, please use the letters $n$, $u$, and $o$ in place of $\mathbb{N}$, $\cup$, and $\emptyset$, respectively.

**Fun fact:** The starter files for Problem Set One contain a parser that’s designed to take as input a string representing a set and to reconstruct what set that is. The logic we wrote to do that parsing was based on a CFG we wrote for sets and set theory. Take CS143 if you’re curious how to go from a grammar to a parser!

Test your CFG thoroughly! In Fall 2017, a quarter of the submissions we received weren’t able to derive the string $\{\emptyset, \emptyset, \emptyset\}$.

As a hint, as is often the case when writing CFGs, we recommend that you use different nonterminals to represent different components of the string. For example, the structure of a comma-separated list is very different from the structure of an expression combining multiple sets together.

Your grammar will have to be able to handle both “a set” and “a comma-separated list of sets”. You may want to try figuring out how to write a grammar for “a comma-separated list of sets” in isolation before trying to tackle all of the different possible structures here.
7 The Complexity of Addition

This problem explores the following question:

**How hard is it to add two numbers?**

Suppose that we want to check whether \( x + y = z \), where \( x, y, \) and \( z \) are all natural numbers. If we want to phrase this as a problem as a question of strings and languages, we will need to find some way to standardize our notation. In this problem, we will be using the **unary number system**, a number system in which the number \( n \) is represented by writing out \( n \) 1’s. For example, the number 5 would be written as 11111, the number 7 as 1111111, and the number 12 as 111111111111.

Given the alphabet \( \Sigma = \{ 1, +, = \} \), we can consider strings encoding \( x + y = z \) by writing out \( x, y, \) and \( z \) in unary. For example:

- \( 4 + 3 = 7 \) would be encoded as 1111+111=1111111
- \( 7 + 1 = 8 \) would be encoded as 1111111+1=11111111
- \( 0 + 1 = 1 \) would be encoded as +1=1

Consider the alphabet \( \Sigma = \{ 1, +, = \} \) and the following language, which we’ll call \( ADD \):

\[
\{ 1^m+1^n=1^{m+n} \mid m, n \in \mathbb{N} \}
\]

For example, the strings 111+1=111 and +1=1 are in the language, but 1+11=11 is not, nor is the string 1+1+1=111.

i. Prove or disprove: the language \( ADD \) defined above is regular.

ii. Write a context-free grammar for \( ADD \), showing that \( ADD \) is context-free. (Please submit your CFG online.)

You may find it easier to solve this problem if you first build a CFG for this language where you’re allowed to have as many numbers added together as you’d like. Once you have that working, think about how you’d modify it so that you have exactly two numbers added together on the left-hand side of the equation.
Optional Fun Problem: Generalized Fooling Sets (Extra Credit)

In this Problem Set, you saw how to use 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 an 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 generalized fooling set for $L$ is a set $F \subseteq \Sigma^* \times \Sigma^*$ is a set with the following properties:

- For any $(x, y) \in F$, we have $xy \in L$.
- For any distinct pairs $(x_1, y_1), (x_2, y_2) \in F$, 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 $F$ for $L$ that contains $n$ pairs of strings, then any NFA for $L$ must have at least $n$ states.

Don’t let the notation scare you off. This is a really cool problem to work through!