# Problem Set 7

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 6th at 12:00PM noon Pacific**

## 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. Provide your answers by downloading the starter files for Problem Set Seven from Canvas and editing the file `res/RegularExpressions.regexes`. (Unlike the DFA/NFA editor from last time, you'll need to manually edit these files by opening them in Qt Creator.) Feel free to test your answers locally, and submit your work on GradeScope.

i. Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$. Write a regular expression for the language $L = \{ w \in \Sigma^* \mid$ the letters in $w$ are sorted alphabetically $\}$. For example, $\texttt{abcde} \in L$, $\texttt{bee} \in L$, $\texttt{a} \in L$ and $\varepsilon \in L$, but $\texttt{decade} \notin L$.

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

*There's no simple way to start with a regex for a language L and to turn it into a regex for $\overline{L}$.*

iii. On Unix-style operating systems like macOS or Linux, files are organized into directories. You can reference a file by giving a *path* to the file, a series of directory names separated by slashes. For example, the path `/home/username/` might represent a user's home directory, and a path like `/home/username/Documents/PS7.tex` might represent that person's solution to this problem set. Paths that start with a slash character are called *absolute paths* and say exactly where the file is on disk. Paths that don't start with a slash are called *relative paths* and say where, relative to the current folder, a file can be found. For example, if I'm logged into my computer and am in my home folder, I could look up the file `Documents/PS7.tex` to find my solution to this problem set.

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, and it might optionally end with a slash, but isn't required to. However, you can't have two consecutive slashes.*

Let $\Sigma = \{\texttt{a}, \texttt{/}\}$. Write a regular expression for $L = \{ w \in \Sigma^* \mid w$ represents the name of a file path on a Unix-style system $\}$. For example, $\texttt{/aaa/a/aa} \in L$, $\texttt{/} \in L$, $\texttt{a} \in L$, $\texttt{/a/a/a/} \in L$, and $\texttt{aaa/} \in L$, but $\texttt{//a//} \notin L$, $\texttt{a//a} \notin L$, and $\varepsilon \notin L$.

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.

iv. Suppose you are taking a walk with your dog on a leash of length two. Let $\Sigma = \{\texttt{y}, \texttt{d}\}$ 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 $\texttt{yyddddyy} \in L$ 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, $\texttt{ddydyy} \in L$. However, $\texttt{yyyyddd} \notin L$, since halfway through your walk you're three steps ahead of your dog; $\texttt{ddyd} \notin L$, because your dog ends up two steps ahead of you; and $\texttt{ddyddyyy} \notin L$, 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.*

v. Let $\Sigma = \{\texttt{M}, \texttt{D}, \texttt{C}, \texttt{L}, \texttt{X}, \texttt{V}, \texttt{I}\}$ and let $L = \{ w \in \Sigma^* \mid w$ is number less than 2,000 represented in Roman numerals $\}$. For example, $\texttt{CMXCIX} \in L$, since it represents the number 999, as are the strings `L` (50), `VIII` (8), `DCLXVI` (666), `CXXXVII` (137), `CDXII` (412), and `MDCXVIII` (1,618). However, we have that $\texttt{VIIII} \notin L$ (you'll never have four `I`'s in a row; use `IX` or `IV` instead), that $\texttt{MM} \notin L$ (it's a Roman numeral, but it's for 2,000, which is too large), that $\texttt{VX} \notin L$ (this isn't a valid Roman numeral), and that $\texttt{IM} \notin L$ (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 $\varepsilon \in L$ 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

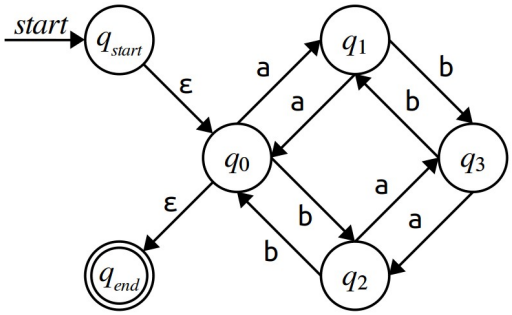
A language $L$ is called ***finite*** if $L$ contains finitely many strings (that is, $|L|$ is a natural number). Given a finite language $L$, explain how to write a regular expression for $L$. Briefly justify your answer; no formal proof is necessary. This shows that all finite languages are regular.

*Watch for edge cases!*

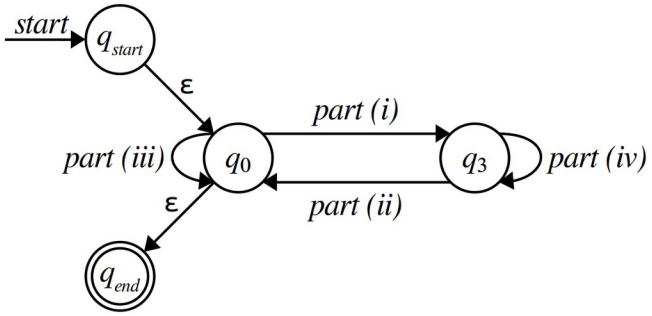
## Problem Three: State Elimination

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. In this problem, you'll use the state elimination algorithm to produce a regular expression for a language.

Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ has an even number of `a`'s and an even number of `b`'s$\}$. Below to the left is a finite automaton for $L$ that we've prepared for the state elimination algorithm by adding in a new start state $q_{start}$ and a new accept state $q_{end}$. If you run two steps of the state elimination algorithm on the above automaton, first eliminating state $q_1$, then eliminating state $q_2$, you will get an automaton whose shape matches the diagram on the right.




*Initial Automaton*




*After Eliminating $q_0$ and $q_1$*

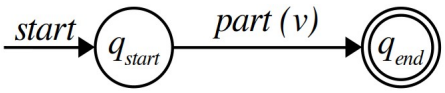
Edit the file `res/StateElimination.regexes` to answer the following questions.

i., ii., iii., iv. What regular expressions go at the indicated positions in the diagram?

*Remember that to eliminate a state $q$, you should identify all pairs of states $q_{in}$ and $q_{out}$ where there's a transition from $q_{in}$ to $q$ and from $q$ to $q_{out}$, then add shortcut edges from $q_{in}$ to $q_{out}$ to bypass state $q$. Remember that $q_{in}$ and $q_{out}$ 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, neither transition should use the Kleene star.*

Now, eliminate $q_3$ and then $q_0$. Your automaton should look like this:




The regular expression on this edge has the same language as the original automaton!

v. What regular expression goes at this spot in the above diagram?

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

Optional but recommended activity: look at the regular expression you ended up with in part (v) 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?

## Problem Four: Embracing the Braces

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

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

For example, we have **{}** ∈ $L_1$, **{{}}** ∈ $L_1$, **{{}{}}{}** ∈ $L_1$, ε ∈ $L_1$, and **{{}}{{{}}}** ∈ $L_1$, but **}{** ∉ $L_1$, **{{}** ∉ $L_1$, and **{}}}}** ∉ $L_1$. 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 ε has nesting depth zero.

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

ii. Write a regular expression for $L_2$ in the file `res/EmbracingTheBraces.regexes`. This shows 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.

*Could you write a regex for strings of braces with nesting depth at most one? At most two? See a pattern?*

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 Five: 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 natual 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 $L$.)

*A later 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.*

On Twitter, all tweets need to be 280 characters or fewer. 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:

$$TWEETS = \{ w \in \Sigma^* \mid |w| \leq 280 \}.$$

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. Prove that any DFA for *TWEETS* must have at least 282 states.

*Use your result from part (i) of this problem. It might be easier to tackle this problem if you consider replacing 280 and 282 with some smaller numbers (say, 2 and 4) to build up an intuition.*

iii. Define a 282-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 280 and 282 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.*

*Look back at PS6 for some examples of how to define a DFA as a 5-tuple.*

Your results here show that the smallest possible DFA for *TWEETS* has exactly 282 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 Six: 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 $\delta^* : Q \times \Sigma^* \to Q$ called the ***extended transition function of D***. Intuitively, the function $\delta^*$ takes as input a state $q$ and a string $w$, then outputs what state you'd end up in if you started in state $q$ and then read string $w$. The function $\delta^*$ is defined, recursively, as follows:

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

Here's one way to think about this. We want $\delta^*(q, w)$ to mean "the state you end up in if you begin in state $q$ and then read $w$." The base case, $\delta^*(q, \varepsilon) = q$, says "if you start in state $q$ and then read $\varepsilon$, you end in state $q$." The recursive case, $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$, says "if you start in state $q$ and want to see where the string $wa$ ends up, first see where you end up with when reading $w$ (that's $\delta^*(q, w)$), then follow the transition at that state labeled $a$ (which is done by computing $\delta(\delta^*(q, w), a)$)."

The rest of this question explores properties of $\delta^*$ and how to switch between higher-level concepts with automata and formal notation.

i. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $q \in Q$ be a state in $D$. Prove that if $x, y \in \Sigma^*$, then $\delta^*(\delta^*(q, x), y) = \delta^*(q, xy)$.

*Intuitively, this says "if you start in state q and read xy, the state you end up in ($\delta^*(q, xy)$) is the state you end up in if you first read just x ($\delta^*(q, x)$), then read y after that ($\delta^*(\delta^*(q, x), y)$)." Your goal is to prove that the formal definition actually matches this intuition by making reference to that specific definition.*

*The $\delta^*$ function is defined recursively, so prove this inductively. As a hint, use induction on the length of the string y. Feel free to use the fact that any string u of length $k+1$ can be written as $u = va$ for a string v of length k and some $a \in \Sigma$.*

*Watch your types! You have two functions to work with, the transition function $\delta : Q \times \Sigma \to Q$ and the extended transition function $\delta^* : Q \times \Sigma^* \to Q$. The first argument to each function is a state. The second argument to $\delta$ must be a single character, and the second argument to $\delta^*$ is a string.*

ii. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Fill in the blanks below with the symbolic notation equivalent to the high-level intuitions we've been using. No justification is necessary. We've filled in one of these for you.

| | |
|---|---|
| "The character $a$ is in the alphabet of the DFA." | $a \in \Sigma$ |
| "The state that string $w$ ends in when run through $D$." | _____ |
| "$D$'s start state is an accepting state." | _____ |
| "$D$ accepts $w$." | _____ |
| "Strings $x$ and $y$ end in the same state when run through $D$." | _____ |
| "Strings $xw$ and $yw$ end in the same state when run through $D$." | _____ |

iii. Formally speaking, we define $\mathscr{L}(D) = \{ w \in \Sigma^* \mid \delta^*(q_0, w) \in F \}$. Explain how this mathematical definition accords with the plain English one we've been using thus far.

## Problem Seven: Formalizing Myhill-Nerode

In lecture, we wrote a proof of the Myhill-Nerode theorem. The proof from lecture is perfectly fine, but it would be nice to tie up one loose end. The third paragraph references an earlier result:

> ***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 never actually proved that this is true, and instead just sketched out a visual argument explaining it. Now that we have the extended transition function δ*, we can write a rigorous, formal proof of the theorem. The first step is to use our 5-tuple definition of DFAs to pin down, more specifically, what the theorem says. Specifically, in the language of our 5-tuple notation, we have the following formalized version of the theorem:

> ***Theorem (Formalized):*** Let $L$ be a language over $\Sigma$ and let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA where $\mathscr{L}(D) = L$. Then for any strings $x, y \in \Sigma^*$ where $x \not\equiv_L y$, we have $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

We're aware that this is a lot of symbolic notation, so take a few minutes to read over it and convince yourself that it indeed says the same thing as the (informal) first version of the theorem.

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 this one, but use 5-tuple definition of a DFA and the formal definition of the extended transition function in your proof. The table from part (ii) of the previous problem is there to help you figure out how to make your argument rigorous.*

*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!*

## Optional Fun Problem: Generalized Fooling Sets

In Problem Five, 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 ***generalized fooling set*** for $L$ is a set $\mathscr{F} \subseteq \Sigma^* \times \Sigma^*$ is a set with the following properties:

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