Problem Set 5

This fifth problem set explores the regular languages, their properties, and their limits. This will be your first foray into computability theory, and I hope you find it fun and exciting!

Start this problem set early. It contains six problems (plus one survey question and one extra-credit question), several of which require a fair amount of thought. I would suggest reading through this problem set at least once as soon as you get it to get a sense of what it covers.

As much as you possibly can, please try to work on this problem set individually. That said, if you do work with others, please be sure to cite who you are working with and on what problems. For more details, see the section on the honor code in the course information handout.

In any question that asks for a proof, you **must** provide a rigorous mathematical proof. You cannot draw a picture or argue by intuition. You should, at the very least, state what type of proof you are using, and (if proceeding by contradiction, contrapositive, or induction) state exactly what it is that you are trying to show. If we specify that a proof must be done a certain way, you must use that particular proof technique; otherwise you may prove the result however you wish.

If you are asked to prove something by induction, you may use weak induction, strong induction, the well-ordering principle, or structural induction. In any case, you should state your base case before you prove it, and should state what the inductive hypothesis is before you prove the inductive step.

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

This problem set has 125 possible points. It is weighted at 7% of your total grade. The earlier questions serve as a warm-up for the later problems, so do be aware that the difficulty of the problems does increase over the course of this problem set.

Good luck, and have fun!

Due Friday, May 11h at 2:15 PM

Problem One: Constructing DFAs (24 Points)

For each of the following languages over the indicated alphabets, construct a DFA that accepts precisely those strings that are in the indicated language. Your DFA does not have to have the fewest number of states possible. Unless otherwise noted, you should specify your DFA as either a state-transition diagram (the graphical representation we've seen in class) or as a table.

We have an online tool you can use to design, test, and submit the DFAs in this problem. This is a prototype that we think will make it easier to work through these problems. If you would like to use it, visit https://www.stanford.edu/~rwong2/cgi-bin/nfa/nfa.php. There are tutorials available on the website with information about how to use the development environment. If you submit through this system, please make a note of it in your problem set submission so that we know to look online for your answers.

i. For the alphabet Σ = {**0**, **1**, **2**}, construct a DFA for the language *L* = { *w* | *w* contains exactly two **2**s. }

ii. For the alphabet Σ = {**0**, **1**}, construct a DFA for the language *L* = { *w* | *w* contains the same number of instances of the substring **01** and the substring **10** }

iii. For the alphabet Σ = {**a**, **b**, **c**, …, **z**}, construct a DFA for the language *L* = { *w* | *w* contains the word "**cocoa**" as a substring }. Remember that as a shorthand, you can specify multiple letters in a transition by using set operations on Σ (for example, Σ – {**a**, **b**}) *

iv. Suppose that you are taking a walk with your dog along a straight-line path. Your dog is on a leash that has length two, meaning that the distance between you and your dog can be at most two units. You and your dog start at the same position. Consider the alphabet Σ = {**Y**, **D**}. A string in Σ* can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string "**YYDD**" means that you take two steps forward, then your dog takes two steps forward. Let L = { *w* | *w* describes a series of steps that ensures that you and your dog are never more than two units apart }. Construct a DFA for this language.

Problem Two: Constructing NFAs (24 Points)

For each of the following languages over the indicated alphabets, construct an NFA that accepts precisely those strings that are in the indicated language. Unless otherwise noted, you should specify your NFA as either a state-transition diagram (the graphical representation we've seen in class) or as a table. Your NFA may use ε-transitions if you wish. **As in Problem One, you can design, test, and submit your automata through our online system if you wish.**

i. For the alphabet Σ = {**0**, **1**, **2**}, construct an NFA for the language { *w* | *w* ends in **0**, **11**, or **222**. }

ii. For the alphabet Σ = {**a**, **b**, **c**, **d**, **e**}, construct an NFA for the language { *w* | the last character of *w* appears nowhere else in the string, and |*w*| ≥ 1 }

iii. For the alphabet Σ = {**0**, **1**}, construct an NFA for the language { *w* | *w* contains two 1's with exactly five characters in-between them. } For example, **1000001** is in the language, as is **00100110100** and **01111010000**, but **11111** is not, nor are **11101** or **000101**.

---

* DFAs are the basis of a fast algorithm called the *Knuth-Morris-Pratt algorithm* for finding whether a string contains a given substring. The algorithm works by automatically constructing an automaton like the one you'll be building in this problem, then running the automaton on the string to be searched.

Problem Three: Finding Flaws in Regular Expressions (16 Points)

Below are a list of alphabets and languages over those alphabets. Each language is accompanied by a regular expression that claims to match that language, but which does not correctly do so (either it matches a string it should reject, or it rejects a string it should accept). In each case, give an example of a string that is either incorrectly accepted or incorrectly rejected by the regular expression, then write a regular expression that does correctly match the given language.

i. $\Sigma = \{$**0**, **1**$\}$, and $L = \{ w \mid w$ consists of all **0**s or all **1**s $\}$. The regular expression is **(0|1)***.

ii. $\Sigma = \{$**0**, **1**$\}$, and $L = \{ w \mid w$ contains an even number of **0**s. $\}$ The regular expression is **(1*01*0)***

iii. $\Sigma = \{$**0**, **1**$\}$, and $L = \{ w \mid w$ does not contain **01** as a substring. $\}$ The regular expression is **(0|1|00|10|11)***

Problem Four: The Complexity of Exponentiation (16 Points)

How hard is it to check if a number is a perfect power of two?

A number is a power of two if it can be written as 2^n for some natural number n. Consider the language $POWER2 = \{ 1^{2^n} \mid n \in \mathbb{N} \}$ over the simple alphabet $\Sigma = \{$ **1** $\}$. That is, *POWER2* contains all strings whose lengths are a power of two. For example, the smallest strings in *POWER2* are **1**, **11**, **1111**, and **11111111**.

Prove that *POWER2* is not regular. *(Hint: You may want to use the fact that $n < 2^n$ for all $n \in \mathbb{N}$)*

Problem Five: The Complexity of String Searching (20 Points)

How hard is it to search a string for a substring?

A common task in computer programming is to search a string to see if some other string appears as a substring. This task arises in computational biology (searching an organism's genome for some particular DNA sequence), information storage (finding all copies of some phrase in the full text of a book), and in spam filtering (searching for some key words in an email).

More formally, we can define the substring search problem as follows. The **string search problem** is given a string to search for (called the **pattern**) and a string in which the search should be conducted (called the **text**), to determine whether the pattern appears in the text. To encode this as a language problem, let $\Sigma = \{$**0**, **1**, **?**$\}$. We can then encode instances of the string search problem as the string **pattern?text**. For example:

"Does **0110** appear in **1110110** ?" would be encoded as **0110?1110110**

"Does **11** appear in **0001** ?" would be encoded as **11?0001**

"Does ε appear in **1100** ?" would be encoded as **?1100**

Let the language $SEARCH = \{ p$**?**$t \mid p, t \in \{$**0**, **1**$\}^*$ and p is a substring of $t \}$. Prove that *SEARCH* is not regular, which means that no DFA, NFA, or regular expression is powerful enough to describe *SEARCH*.

Problem Six: The Complexity of Addition (20 Points)

As we saw in lecture, if L_1 and L_2 are regular, then $\overline{L_1}$, $L_1 \cup L_2$, $L_1 \cap L_2$, L_1L_2, L_1*, L_1^R, and $h*(L_1)$ are also regular languages. These properties are sometimes called closure properties. In lecture, you saw how to use the pumping lemma to prove that a particular language is not regular. However, many languages can be shown to be nonregular without using the pumping lemma by using the closure properties of regular languages. In this problem, you'll explore how to do this.

Homomorphisms might have seemed like little more than a curiosity in lecture, but they are invaluable tools in computability theory for showing that certain languages are or are not regular. Recall that if $L \subseteq \Sigma_1*$ is a regular language and $h* : \Sigma_1* \rightarrow \Sigma_2*$ is a homomorphism, then $h*(L)$ is a regular language as well. By the contrapositive, this means that if $h*(L)$ is **not** a regular language, then L is **not** a regular language either. In other words, we can prove that a language L is not regular by finding some homomorphism $h*$ such that $h*(L)$ is not regular.

In Friday's lecture, we saw that for the alphabet $\Sigma = \{$**0**, **1**, **?**$\}$, the language

$$EQUAL = \{ x\textbf{?}x \mid x \in \{\texttt{0}, \texttt{1}\}* \}$$

is not regular. This language corresponds to instances of the problem "are these two strings of **0**s and **1**s equal to one another?" Using the pumping lemma, it can also be shown that if we let $\Sigma = \{$**0**, **?**$\}$, then the language

$$SAME = \{ \texttt{0}^n\textbf{?}\texttt{0}^n \mid n \in \mathbb{N} \}$$

is also not regular (you don't need to prove this, but you may find it to be good practice). Here, the language $SAME$ corresponds to solving the problem "given two strings of **0**s separated by a **?**, does the string on the left-hand side contain the same number of **0**s as the string on the right-hand side?"

In the remainder of this problem, you will see how to use homomorphisms and the fact that $SAME$ is not regular to prove that another language is not regular either. The question we will address is

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 **111+1111=1111111**

7 + 1 = 8 would be encoded as **1111111+1=11111111**

0 + 1 = 1 would be encoded as **+1=1**

Consider the language $ADD = \{\texttt{1}^m\texttt{+}\texttt{1}^n\texttt{=}\texttt{1}^{m+n} \mid m, n \in \mathbb{N} \}$. That is, ADD consists of strings encoding two unary numbers and their sum. We will see how to prove that ADD is not regular. Consider the function $h : \{$**1**, **+**, **=**$\} \rightarrow \{$**0**, **?**$\}*$ defined as

- h(**1**) = **0**
- h(**+**) = ε
- h(**=**) = **?**

Now, let $h^* : \{1, +, =\}^* \rightarrow \{0, ?\}^*$ be the homomorphism derived from h. For example:

$h^*(\texttt{11+11=1111}) = \texttt{0000?0000}$

$h^*(\texttt{1+=1}) = \texttt{0?0}$

$h^*(\texttt{111+11=11111}) = \texttt{00000?00000}$

$h^*(\texttt{1+111=1111}) = \texttt{0000?0000}$

Notice that applying h^* to strings in *ADD* seems to produce strings in SAME. In fact, we might wonder whether $h^*(ADD) = SAME$. That is, does this homomorphism transform the set of strings in *ADD* into the set of strings in *SAME*?

i. Prove that $h^*(ADD) \subseteq SAME$. (Hint: If $x \in h^*(ADD)$, then $x = h^*(w)$ for some $w \in ADD$. What do you know about strings in *ADD*?)

ii. Prove that $SAME \subseteq h^*(ADD)$. *(Hint: Show that for any* $x \in SAME$*, there is some* $w \in ADD$ *such that* $h^*(w) = x$*)*

iii. Based on your answers to (i) and (ii), prove that *ADD* is not a regular language. Do **not** use the pumping lemma.

iv. Although it is true that if L is regular, then $h^*(L)$ is regular for any homomorphism h^*, the inverse of this statement is not true. That is, if L is not regular, it is still possible for $h^*(L)$ to be regular. Give an example of a nonregular language L and homomorphism h^* such that $h^*(L)$ is regular.

But wait a minute! Didn't we prove in lecture that addition is indeed a regular language? We did indeed build a DFA in lecture that could verify addition was done correctly, but in doing so we chose an unusual representation for our strings. Specifically, we build an alphabet out of columns of integers, then encoded the addition as binary addition. As you've just shown, though, if we were to change our encoding scheme and instead do unary addition, then the new language would not be regular.

This highlights a key different between *problems* and *languages*. When encoding a problem as a language, it is often the case that the difficulty of solving that problem hinges on how it is represented. Only *languages* can be regular or nonregular. We will return to this topic later when we cover complexity theory.

Problem Seven: Course Feedback (5 Points)

We want this course to be as good as it can be, and we'd really appreciate your feedback on how we're doing. For a free five points, please answer the following questions. We'll give you full credit no matter what you write (as long as you write something!), but we'd appreciate it if you're honest about how we're doing.

i. How hard did you find this problem set? How long did it take you to finish?

ii. Does that seem unreasonably difficult or time-consuming for a five-unit class?

iii. Did you attend Monday's midterm review session? If so, did you find it useful?

iv. How is the pace of this course so far? Too slow? Too fast? Just right?

v. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?

Submission instructions

There are three ways to submit this assignment:

1. Hand in a physical copy of your answers at the start of class. This is probably the easiest way to submit if you are on campus.
2. Submit a physical copy of your answers in the filing cabinet in the open space near the handout hangout in the Gates building. If you haven't been there before, it's right inside the entrance labeled "Stanford Engineering Venture Fund Laboratories." There will be a clearly-labeled filing cabinet where you can submit your solutions.
3. Send an email with an electronic copy of your answers to the submission mailing list (cs103-spr1112-submissions@lists.stanford.edu) with the string "[PS5]" somewhere in the subject line.

If you are an SCPD student, we would strongly prefer that you submit solutions via email. Please contact us if this will be a problem.

Extra Credit Problem: The Pumping Lemma Revisited (5 Points)

In lecture, we sketched a proof of the pumping lemma by using the fact that every regular language is accepted by a DFA. By looking at the structure of a DFA, we could conclude that the pumping lemma must hold. Write an alternative proof of the pumping lemma that is based on the regular expression representation of a regular language. Do not reference DFAs or NFAs in your proof.