Problem Set 6

Now that you're more familiar with the regular languages, it's time to start reasoning about their limits. This problem set explores the boundaries of regular languages and how to navigate them. Once you're done with this problem set, you will have a much better understanding of what sorts of problems can be solved with computing machines as simple as DFAs.

Start this problem set early. It contains five problems (plus one survey question and one extra credit problem), 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, structural induction, or well-founded 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, November 11th at 2:15 PM

Problem One: Finding Flaws in Regular Expressions (20 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, given 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. $\Sigma = \{\mathbf{0}, \mathbf{1}\}$, and L = { w | w consists of all **0**s or all **1**s }. The regular expression is **(0 | 1)***.

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

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

Problem Two: The Complexity of Exponentiation (25 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 $\Sigma = \{ \mathbf{1} \}$. That is, L contains all strings whose lengths are a power of two. For example, the smallest strings in L 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 Three: The Complexity of String Searching (25 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 $\Sigma = \{\mathbf{0}, \mathbf{1}, \mathbf{?}\}$. We can then encode instances of the string search problem as **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* | *p*, *t* ∈ {**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 Four: The Complexity of Addition (25 Points)

As we saw in lecture, if L_1 and L_2 are regular, then $\overline{L_1}$, $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, L_1L_2, L_1^*, and $h(L_1)$ are regular languages. These properties are sometimes called *closure properties*, because if we begin with the set of all regular languages and then add into that set all languages we could form using the above operations, we would not end up with anything we didn't already have. In other words, we could not "escape" from the set of regular languages this way, so the set of regular languages is said to be **closed** under these operations.

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.

In lecture, we saw that regular languages are closed under *homomorphism*. If you'll recall, string homomorphisms were defined as follows. Suppose that we have two alphabets Σ_1 and Σ_2 and consider any function h such that $h : \Sigma_1 \to \Sigma_2^*$. This function takes any character in Σ_1 and associates it with some string composed of characters in Σ_2. For example, if $\Sigma_1 = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots, \mathbf{z}\}$ and $\Sigma_2 = \{\mathbf{A}, \mathbf{B}, \ldots, \mathbf{Z}\}$, we could consider a simple function defined as follows:

- $h(\mathbf{a}) = \mathbf{A}$
- $h(\mathbf{b}) = \mathbf{B}$
- …
- $h(\mathbf{z}) = \mathbf{Z}$

In this case, the function maps lowercase characters in Σ_1 to their corresponding uppercase letters in Σ_2.

Once we have a function $h : \Sigma_1 \to \Sigma_2^*$ which maps single characters in Σ_1 to strings in Σ_2^*, we can extend this function to form a new function $h : \Sigma_1^* \to \Sigma_2^*$ that maps *strings* in Σ_1^* to strings in Σ_2^* by applying h to each of the characters in the input string. It is unfortunate that the choice of notation means that h stands for both a function on strings and a function on characters, but this is the common convention. For simplicity, we'll introduce new notation to disambiguate. If $h : \Sigma_1 \to \Sigma_2^*$ is a function mapping characters to strings, then we'll let

$$h^* : \Sigma_1^* \to \Sigma_2^*$$

be the **string homomorphism** based on h. h^* is defined such that if w is a string whose characters are $w_1, w_2, \ldots, w_n$, then

$$h^*(w) = h(w_1)\, h(w_2) \ldots h(w_n)$$

That is, we apply h to each of the characters of w individually, then concatenate the result together.

Alternatively, you may find it useful to think about h^* like this: if $x, y \in \Sigma_1^*$, then

$$h^*(xy) = h^*(x)h^*(y)$$

(You do not need to prove this, but it might be a good exercise to do so.)

Let's see what this function h^* looks like in practice. For example, suppose that h is the function defined above that maps lower-case letters to upper-case letters. Then we have the following:

- h^*(**hello**) = **HELLO**
- h^*(**quiet**) = **QUIET**

However, homomorphisms do not need to map single characters to single characters; it's perfectly fine to define a homomorphism that associates characters in the Σ_1 with arbitrary strings in Σ_2. For example, consider Σ_1 = { **e**, **m**, **r**, **t** } and Σ_2 = { **m**, **n**, **o**, **s** } as above and this new homomorphism h:

- h(**e**) = **on**
- h(**m**) = **m**
- h(**t**) = **so**
- h(**r**) = ε

Then we have that

$$h^*(\textbf{meter}) = \textbf{monsoon}.$$

Notice that h maps **e** and **t** to two-letter strings and associates **r** with the empty string, meaning that any **r**'s that appear in a string are completely removed by h^*.

We can generalize the notion of a homomorphism from applying to individual strings to applying to languages as a whole. In particular, if $h^* : \Sigma_1^* \to \Sigma_2^*$ is a homomorphism, then if L is a language, the language h^*(L) is the language defined as

$$h^*(L) = \{ w \mid \exists x \in L.\ h^*(x) = w \}$$

That is, h^*(L) is the language formed by applying the homomorphism h^* to each of the elements of L. For example, let $h : \{\textbf{0}, \textbf{1}\} \to \{\textbf{a}, \textbf{b}\}^*$ be defined as

- h(**0**) = **aa**
- h(**1**) = **bbb**

Then if L = { **0**, **1**, **00**, **11**, **000**, … }, then h^*(L) = { **aa**, **bbb**, **aaaa**, **bbbbbb**, **aaaaaa**, … }.

At this point homomorphisms might seem like little more than a curiosity, 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 $h^* : \Sigma_1^* \to \Sigma_2^*$ 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 $\Sigma = \{\textbf{0}, \textbf{1}, \textbf{?}\}$, the language

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

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?" Interestingly, using the pumping lemma it can also be shown (you do not need to prove this, though it would be good practice!) that if we let $\Sigma = \{\mathbf{0}, \mathbf{?}\}$, then the language

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

is also not regular (again, 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 $\Sigma = \{\mathbf{1}, +, =\}$, 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**
2 + 2 = 4 would be encoded as **11+11=1111**
0 + 1 = 1 would be encoded as **+1=1**

Consider the language $ADD = \{\mathbf{1}^m\mathbf{+}\mathbf{1}^n\mathbf{=}\mathbf{1}^{m+n} \mid m, n \in \mathbb{N} \}$. 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 $h : \{\mathbf{1}, +, =\} \rightarrow \{\mathbf{0}, \mathbf{?}\}^*$ defined as

- $h(\mathtt{1}) = \mathtt{0}$
- $h(\mathtt{+}) = \varepsilon$
- $h(\mathtt{=}) = \mathtt{?}$

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

h^*(**11+111=11111**) = **00000?00000**
h^*(**1+=1**) = **0?0**
h^*(**1111+11=111111**) = **000000?000000**
h^*(**1+111=1111**) = **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) ⊆ SAME. (Hint: If x ∈ h*(ADD), then x = h*(w) for some w ∈ ADD. What do you know about strings in ADD?)*

ii. Prove that *SAME ⊆ h*(ADD). (Hint: Show that for any x ∈ SAME, there is some w ∈ 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. Prove or disprove: If L is not regular and h^* is a homomorphism, then h^*(L) is not regular.

Problem Five: The Complexity of Pet Ownership (25 Points)

Homomorphism is only one of many closure properties of regular languages. You'll explore some other uses of closure properties here.

i. Suppose that R is a regular language and L is an arbitrary language. Prove that if R ∩ L is not regular, then L is not regular.

We will now address the question

How hard is it to walk your dog without a leash?

On Problem Set 5, you saw how to construct a DFA that would accept strings representing taking your dog for a walk. You had an alphabet whose symbols corresponded to you or your dog taking a step forward. You had a leash of length two, meaning that you and your dog could not be very far apart from one another as you took a walk. As you saw, the language of valid walks with your dog was regular, since the leash never let the two of you get too far apart.

Now that you've grown very close to your dog, you decide to take your dog off a leash and go for a walk together. The distance between you and your dog can now grow arbitrarily large, but you and your dog trust one another to arrive at the same destination.

Let Σ = { **Y**, **D** }, where **Y** represents you moving one unit forward and **D** represents your dog moving one step forward. Then a string of **Y**s and **D**s represents you and your dog going for a walk. For example, in the string **YYDY**, you end up two steps ahead of your dog, while in the string **DDDDYY**, your dog takes off and ends up two steps ahead of you (perhaps there was some other dog it wanted to meet).

If we consider the language of strings representing walks where you and your dog end up at the same location, we get the language *DOGWALK* = { w | w has the same number of **Y**s and **D**s }. This language is not regular; in fact, it's pretty much identical to the language *BALANCE* = { w | w has the same number of **0**s and **1**s } that we proved is not regular in lecture. But suppose that you didn't know that this language was not regular and didn't want to use the pumping lemma to prove this (since you'd like to spend more time with your dog). In this problem, you'll see how to do this.

ii. Prove that *DOGWALK* ∩ **Y*D*** = { $\mathbf{Y}^n\mathbf{D}^n$ | n ∈ ℕ }.

iii. Using the fact that $\{ \mathbf{Y}^n\mathbf{D}^n \mid n \in \mathbb{N} \}$ is not regular, along with your results from (i) and (ii), prove that *DOGWALK* is not regular. Do **not** use the pumping lemma in your proof.

Now, suppose that you want to train your dog to "stay" and "come." You unleash your dog and go for a walk where you cover some distance while your dog sits and waits, then signal your dog to catch up to your current position. If you do this for a while, you will find that your walks might look something like this:

YYYDDDYDYYYYDDDDYYYYYYDDDDDDYYDDYD

That is, you always take some *n* steps forward, then your dog takes *n* steps forward, then you take some *m* steps forward, then your dog takes some *m* steps forward, etc. In other words, your walk is a series of smaller walks, each of which is formed by you walking some number of steps and then waiting for your dog to catch up to you.

iv. Consider the language *TRAINING* = $\{ \mathbf{Y}^n\mathbf{D}^n \mid n \in \mathbb{N} \}^*$. Prove that *TRAINING* is not a regular language. Do **not** use the pumping lemma in your proof; instead, use closure properties of regular languages.

v. Prove or disprove: If L is not a regular language, then L* is not a regular language.

Problem Six: 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 problem 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 four ways to submit this assignment:

1. Hand in a physical copy of your answers at the start of class.

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 Venture Fund Laboratories." There will be a clearly-labeled filing cabinet into which you can submit your homework.

3. Submit a physical copy of your answers in the drop-off box in the Gates basement. We'll send out an email announcement about the location of this box on the night of Friday, November 11.

4. Send an email with an electronic copy of your answers to **cs103@cs.stanford.edu**

Extra Credit Problem: The Myhill-Nerode Theorem

The pumping lemma for regular languages gives a *sufficient* condition for a language to be regular, not a *necessary* condition. In other words, it is possible for a language to be nonregular, but for the conditions of the pumping lemma to apply to it. We saw an example of this in lecture: the language of strings containing equal numbers of 0s and 1s is not regular, but it passes the weak pumping lemma.

There is a substantially more powerful result from formal language theory called the **Myhill-Nerode Theorem** that provides an exact characterization of the regular languages. The theorem works by defining an equivalence relation $=_L$ (called the *indistinguishability relation*) over strings. This relation is defined as follows:

$$x =_L y \quad \equiv \quad \forall w \in \Sigma^*. (xw \in L \leftrightarrow yw \in L)$$

Informally, this says that x and y are considered equal according to $=_L$ if and only if for any string w, then either both xw and yw are contained in L or neither xw nor yw are. In other words, there is no suffix w that you could append to x and y to make xw accepted but not yw not or vice-versa.

Recall that an *equivalence class* is the set $[x]$ defined as $\{ y \mid y \in \Sigma^* \text{ and } x =_L y \}$. Every string w belongs to some equivalence class (namely, the class $[w]$), but no string belongs to more than one equivalence class. Consequently, these equivalence classes partition Σ^* into some (potentially infinite) number of disjoint sets. The Myhill-Nerode theorem states that there are finitely many equivalence classes (that is, all strings in Σ^* can be split into finitely many sets of strings, each of which are equal to one another under the $=_L$ relation) precisely when the language L is regular. More formally:

Theorem: L is a regular language iff there are finitely many equivalence classes under the $=_L$ relation.

i. Using the Myhill-Nerode theorem, come up with an alternate proof that any finite language is regular.

ii. Using the Myhill-Nerode theorem, come up with an alternate proof that the language $L = \{ 0^n 1^n \mid n \in \mathbb{N} \}$ is not regular.

iii. Prove the Myhill-Nerode theorem.