Theory of Computation
Problem Set 4
Universidad Politecnica de San Luis Potosi

Please start solving these problems immediately, don't procrastinate, and work in study groups.

Please do not simply copy answers that you do not fully understand;

Advice: Please try to solve the easier problems first (where the meta-problem here is to figure out which are the easier ones). Don't spend too long on any single problem without also attempting (in parallel) to solve other problems as well. This way, solutions to the easier problems (at least easier for you) will reveal themselves much sooner (think about this as a "hedging strategy" or "dovetailing strategy").

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/class/cs103/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 $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}\}$, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ contains exactly two **2**s. $\}$

ii. For the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ contains the same number of instances of the substring **01** and the substring **10** $\}$

iii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}\}$, construct a DFA for the language $L = \{ w \in \Sigma^* \mid w$ 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, $\Sigma - \{\texttt{a}, \texttt{b}\}$) *

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 $\Sigma = \{\texttt{Y}, \texttt{D}\}$. 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 $L = \{ w \in \Sigma^* \mid w$ 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 $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}\}$, construct an NFA for the language $\{ w \in \Sigma^* \mid w$ ends in **0**, **11**, or **222**. $\}$

ii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$, construct an NFA for the language $\{ w \in \Sigma^* \mid$ the last character of w appears nowhere else in the string, and $|w| \geq 1$ $\}$

iii. For the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, construct an NFA for the language $\{ w \in \Sigma^* \mid w$ contains two **1**'s with exactly five characters in-between them. $\}$ For example, **1**000001 is in the language, as is 00**1**00110**1**00 and 0**1**11110**1**00000001, 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 $L = \{ w \in \Sigma^* \mid w$ consists of all `0`s or all `1`s $\}$. The regular expression is `(0|1)*`.

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

iii. $\Sigma = \{$`0`, `1`$\}$, and $L = \{ w \in \Sigma^* \mid w$ 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 $h^* : \Sigma_1^* \to \Sigma_2^*$ 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 $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{?}\}$, the language

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

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 $\Sigma = \{\texttt{0}, \texttt{?}\}$, then the language

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

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 $\Sigma = \{\texttt{1}, \texttt{+}, \texttt{=}\}$, 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 $h : \{\texttt{1}, \texttt{+}, \texttt{=}\} \to \{\texttt{0}, \texttt{?}\}^*$ defined as

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

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

h^*(**11+11=1111**) = **0000?0000**

h^*(**1+=1**) = **0?0**

h^*(**111+11=11111**) = **00000?00000**

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 $h^*(ADD) \subseteq SAME$. (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? Does that seem unreasonably difficult or time-consuming for a five-unit class?

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

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

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