Theory and Practice of Algorithms Homework 2 Fall 2022

Due:Sept 29

1. Draw an NFA whose alphabet is $\{a, b\}$ and which accepts the language of strings such that any two consecutive as are followed by exactly 3 consecutive bs. Note that this does not require that 2 consecutive as ever appear in the input, just that anytime you see aa, you must then see exactly 3 consecutive bs. The following are examples of strings in the language of this NFA:

```
(epsilon)
a
babaabbba
aabbbaabbbabababbbbb
```

2. Write a regular expression whose alphabet is $\{a, b, c\}$ whose language is the set of strings where any as in the string must come before any cs in the string (put another way, there may not be an a after a c). Note that bs may be anywhere in the string. The following are examples of strings in this language:

```
(epsilon)
aababaccc
ac
a
c
bbbb
```

3. For each of the following languages, determine if the language is regular or not. If the language *is* regular, demonstrate its regularity by either drawing an NFA which accepts the language,

or writing a regular expression which accepts the language. If the language *is not* regular, **prove**[1] that it is not regular.

(a) The language of strings over the alphabet $\{a\}$ of the form a^i where $\exists k.i = k^2$ (*i.e.*, sequences of as where the number of as is a perfect square). Examples of strings in this language include:

```
(epsilon)
aaaa
aaaaaaaaaaaaaaaa
```

(b) The language of strings over the alphabet $\{a, b\}$ where ab occurs as many times as ba. Examples of strings in this language include:

```
(epsilon)
b
aba
abba
bbbaaaabaaabb
```

(c) The language of strings over the alphabet $\{a, b\}$ of the form $a^i b^j$ where $(i \bmod 2) + 1 = j \bmod 3$. Examples of strings in this language include:

```
b
abbbbb
aaaaabb
```

4. **Implementation** This last problem is going to be implementation in Python. You will submit `dfa.py` separately on Gradescope for this problem. For this problem, you will be implementing a DFA in Python. We have provided to you code to

[1] Recall from class that for a pumping lemma proof, you only have to the "core" of the proof.

read a DFA description (`readDFA`), which takes a filename, and returns a `Dfa` object. This DFA has two fields, `transitions` and `accepting`. The states of this DFA are integers from 0 to `len(transitions)`. The alphabet of this DFA is all ASCII characters. The start state is state 0. `accepting` is a list (think "array" if you are in 551) which indicates for each state whether it is accepting or not. If `accepting[i]` is `True` then state `i` is an accepting state. If `accepting[i]` is `False` then state `i` is not an accepting state.

The transition function is described in `transitions` as a 2D list (again, think 2D array). Namely

$$\delta(q, c) = \text{transitions}[q][c]$$

That is, you index the `transitions` list first by state, then by input symbol, and get the state number to transition to.

Your goal is to write the `doDFA` function, which takes in a DFA object (as returned by `readDFA`, and an input string `s`). This function should return `True` if the dfa accepts the input string, and `False` if not.

Note that this function should not be long nor complex (my implementation is 5 lines of code).

Examining the provided DFAs If you want to take a look at the sample DFAs that we provided, and what `readDFA` produces, you can start by looking in `odd1s.dfa`. This file is a DFA which recognizes the language of strings which have only 0s and 1s, and where the number of 1s in the string is odd. The first line names the accepting states. In this case, just state 1. If you do the following in Python (after evaluating the definitions in `dfa.py`):

```
d = readDFA("odd1s.dfa")
d.accepting
```

You will see

```
[False, True, False]
```

This result shows that states 0 and 2 are not accepting, and state 1 is accepting. The next lines in the `odd1s.dfa` file describe the transitions. Each one starts by specifying what state it is describing (0: means "from state 0..."). Then it has an input symbol and the state number (symbol=state). So the line for state 0 (0:1=1,0=0) says that if you see a '1' go to state 1, and if you see a '0' go to state 0. The next line (1:1=0,0=1) says that if you are in state 1 and you see a '1', go to state 0, and if you see a '0' goto state 1.

If you then evaluate

```
d.transitions
```

You will see that the result has 3 states (0, 1, and 2), and that most of the entries are 2. Our readDFA adds a state for any symbols not specified in the input file (like 'a' and '!' etc) and sends them to a non-accepting state which always loops to itself. If you evaluate

```
d.transitions[1][ord('0')]
```

You will get 1. Here `ord` converts the character '0' to its numerical value. This says that if you are in state 1, and you see a '0', you will go to state 1.