# CS 5000: F23: Theory of Computability
# Assignment 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 9, 2023

## Learning Objectives

1. Equivalence of NFA with $\epsilon$-transitions and NFA without $\epsilon$-transitions.
2. Subset Construction (i.e., Equivalence of NFAs and DFAs)
3. DFA Implementation

## Introduction

Problems 1 and 2 show two typical stages of converting an NFA to a DFA that we discussed in Lectures 3 and 4. In stage 1, shown in Problem 1, an NFA with $\epsilon$-transitions is converted to an equivalent NFA without $\epsilon$-transitions. In stage 2, shown in Problem 2, an NFA without $\epsilon$-transitions is converted to an equivalent DFA with the subset construction algorithm. This conversion is used in many areas of computer science and engineering, e.g., robotics, to gain speed and determinism, because DFA are faster than NFA (but take a lot more space!). It is also used in compiler design to obtain deterministic parsers for programming languages. Problem 3 will give you some experience with implementing DFA. We'll build up on it in the next assignment when we implement the subset construction algorithm.

## Problem 1: (1 point)

Consider the following NFA $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b, c\}$, $F = \{q_1, q_2, q_3\}$, and $\delta$ is defined by the following transitions.

1. $\delta(q_0, \epsilon) = \{q_1, q_2, q_3\}$;
2. $\delta(q_1, a) = \{q_1\}$;
3. $\delta(q_2, b) = \{q_2\}$;
4. $\delta(q_3, c) = \{q_3\}$.

1. Define $L(M)$ in the set former notation;
2. Convert $M$ to an equivalent NFA $M'$ with no $\epsilon$-moves and give formal and pictorial definitions of $M'$ (i.e., one formal definition and one picture).

## Problem 2: (1 point)

Consider that following NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F)$, where $Q_N = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $F_N = \{q_1\}$, and the transition function $\delta_N$ is defined as follows:

1. $\delta_N(q_0, 0) = \{q_0, q_1\}$;
2. $\delta_N(q_0, 1) = \{q_1\}$;
3. $\delta_N(q_1, 0) = \{q_2\}$;
4. $\delta_N(q_1, 1) = \{q_2\}$;
5. $\delta_N(q_2, 0) = \{q_2\}$;
6. $\delta_N(q_2, 1) = \{q_2\}$.

Apply the subset construction algorithm to convert $M$ into an equivalent DFA $M_D$. Specify $M_D$ by drawing its graph (similar to the graph in Figure 1) and by formally defining it like $M_N$ above.

## Problem 3: (2 points)

In this problem, we'll design and implement in Python a DFA that determines if an integer represented as a binary string is divisible by 7.

To warm up, let's design and implement in Python a DFA to determine if an integer represented as a binary string is divisble by 3. You'll follow the same logical steps to design the DFA for modulo 5 division.

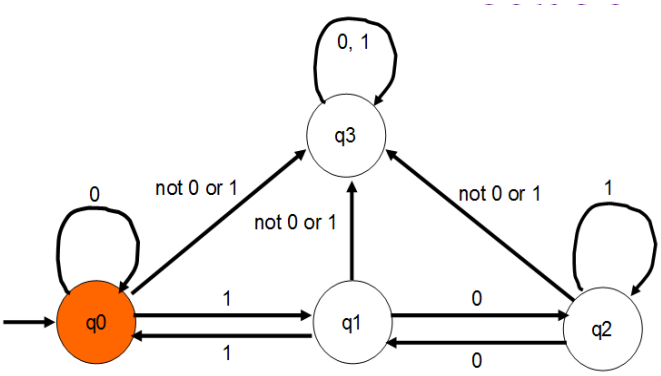


Figure 1: Modulo 3 DFA.

Let's reason as follows. Modulo 3 division places all integers into one (and only one!) of the three classes (aka equivalence classes): 1) the remainder 0 class – integers divisible by 3 (e.g., 12) that leave the remainder of 0; 2) the remainder 1 class – integers that leave the remainder of 1 (e.g., 7); 3) the remainder 2 class – integers that leave the remainder of 2 (e.g., 8).

Our DFA will have three states: $q_0$, $q_1$, and $q_2$. The state $q_0$ is the only accepting state and is meant to be reachable only on binary strings that fall into the remainder 0 class (i.e., evenly divisible by 3). The state $q_1$ is reachable on binary strings that represent integers that leave the remainder of 1 when divided by 3. The state $q_2$ is reachable on binary strings that represent integers that leave the remainder of 2 when divided by 3.

When constructing the transitions for the Module 3 DFA we can proceed as follows. The DFA starts at $q_0$. If it sees 0 in the input, it stays in $q_0$, because 0 is divisible by 3. If it sees 1 in the input in $q_0$, it goes to $q_1$, because 1 leaves the remainder of 1 when divided by 3.

If the DFA sees 0 in $q_1$, it goes to $q_2$, because the input 10 in binary is 2 in decimal (i.e., $(10)_2 = (2)_{10}$) and 2 leaves the remainder of 2 when divided by 3. If it sees 1 in $q_1$, it goes back to $q_0$, because 11 in

binary is 3 in decimal and 3 is divisible by 3 (i.e., leaves the reaminder of 0).

If the DFA sees 0 in $q_2$, it goes to $q_1$, because the input $(100)_2 = (4)_{10}$ and 4 leaves the remainder of 1 when divided by 3. If it sees 1 in $q_2$, it stays in $q_2$, because the input $(101)_2 = (5)_{10}$ and 5 leaves the remainder of 2 when divided by 3.

What happens if the DFA sees a symbol in the input that's neither 0 nor 1? Well, it sinks, i.e., transitions to the sink state $q_3$ and stays there. Sink states are never final.

Let's implement this DFA in Python. The simplest is to implement a DFA is to implement each state as a function, which is what we'll do. We'll define the class `Mod3DFA` in `Mod3DFA.py` and write a method `__q0()`, the double under sign (aka *dunder*) means that the method is private.

```
class Mod3DFA:

    def __q0(self, s, pos):
        if len(s) == pos:
            return True
        elif s[pos] == '0':
            return self.__q0(s, pos+1)
        elif s[pos] == '1':
            return self.__q1(s, pos+1)
        else:
            return self.__q3(s, pos+1)
```

Let's figure out what `__q0()` does. It takes a bit string `s` and the current position in that string `pos` that points to the bit the DFA is currently reading. If the current position equals the length of the bit string, the DFA accepts (i.e., returns `True`).

If the bit at the current position is 0, the DFA reads it and stays in the state $q_0$. Note that the position is incremented, because the DFA's reading head must transition one symbol to the right.

```
        elif s[pos] == '0':
            return self.__q0(s, pos+1)
```

If the bit at the current position is 1, the DFA reads it and transitions to $q_1$.

```
        elif s[pos] == '1':
            return self.__q1(s, pos+1)
```

Otherwise, the DFA sinks.

```
       else:
          return self.__q3(s, pos+1)
```

If you take a look at the DFA's diagram in Figure 1, you'll see that the method `__q0()` mirrors the transitions from the state $q_0$. Take a look at the implementations of `__q1()`, `__q2()`, and `_q3()` in `Mod3DFA.py`.

The only thing left for us to do is to define `Mod3DFA.processString()` method that will get `Mod3DFA` going on the input string from $q_0$. Here it is.

```
    def processString(self, s):
        if s[0] == '0':
            return self.__q0(s, 1)
        elif s[0] == '1':
            return self.__q1(s, 1)
        else:
            return self.__q3(s, pos+1)
```

We can write a unit test to verify that the DFA behaves correctly. Here's one that tests it on the integers from 0 to 1,000,000 (the Py source is in `cs5000_f23_hw02_uts.py`).

```
    def test_assgn_02_ut_01(self):
        print('\n***** Mod3DFA: Hw02: UT01 ************')
        lower, uppper = 0, 1000001
        dfa = Mod3DFA()
        for i in range(lower, upper):
            bstr = '{0:b}'.format(i)
            if i % 3 == 0:
                assert dfa.processString(bstr) == True
            else:
                assert dfa.processString(bstr) == False
        print('Mod3DFA: HW02: UT01 passed...')
```

The line `bstr = '{0:b}'.format(i)` converts an integer `i` to a binary string. You can do the same thing with `bin(i)[2:]`. A couple more tricks of converting decimal into binary and back.

```
>>> bin(12)
'0b1100'
>>> int('0b1100', 2)
12
>>> int(bin(12), 2)
12
>>> int(bin(12), 2) == 12
True
>>> int('0b1101', 2) == 13
True
```

Now design and implement, in a similar fashion, a DFA to determine if an integer represented as a binary string is divisible by 7 and implement it in `Mod7DFA.py`. I've written some starter code for you in `Mod7DFA.py`. Test your implementation with this unit test in `cs5000_f23_hw02_uts.py`.

```
    def test_assgn_02_ut_02(self):
        print('\n***** Mod7DFA: HW03: UT02 ************')
        lower, upper = 0, 1000001
        dfa = Mod7DFA()
        for i in range(lower, upper):
            bstr = '{0:b}'.format(i)
            if i % 7 == 0:
                assert dfa.processString(bstr) == True
            else:
                assert dfa.processString(bstr) == False
        print('Mod7DFA: HW02: UT02: passed...')
```

## What to Submit

Type your solutions to Problems 1 and 2 in a math-friendly editor and save them as `hw02.pdf`. Save your solution to problem 3 in `Mod7DFA.py`. Zip your `hw02.pdf` and your `Mod7DFA.py` into `hw02.zip` and submit it in Canvas. The appendix below gives my output of running both unit tests in `cs5000_f23_hw02_uts.py`.

Happy Thinking and Hacking!

## My Output of UTs 1 and 2

```
>>> python cs5000_f23_hw02_uts.py
```

```
***** Testing Mod3DFA: Hw02: UT01 ************
Mod3DFA: HW02: UT01 passed...
.
***** Mod7DFA: HW03: UT02 ************
Mod7DFA: HW02: UT02: passed...
.
----------------------------------------------
Ran 2 tests in 137.676s
OK
```