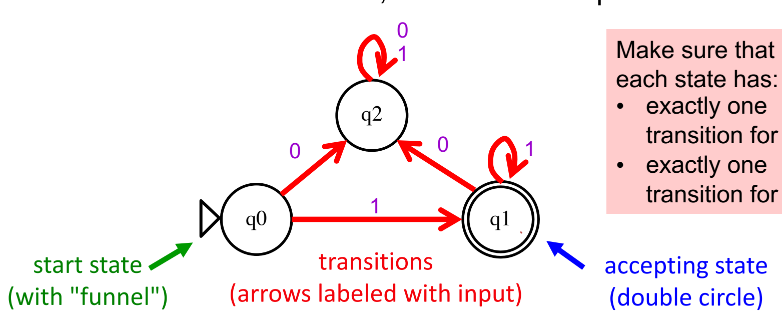
Finite State Machines, part II; Final Project Revisited

Computer Science 111
Boston University

Vahid Azadeh Ranjbar, Ph.D.

Recall: Finite State Machine (FSM)

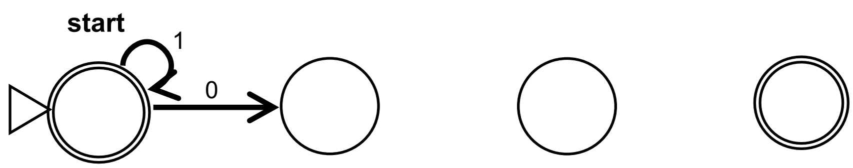
- An abstract model of computation
- Consists of:
  - one or more states
    - *exactly one* of them is the *start / initial state*
    - *zero or more* of them can be an *accepting state*
  - a set of possible input characters (we're using {0, 1})
  - *transitions* between states, based on the inputs



Make sure that each state has:
- exactly one transition for 0
- exactly one transition for 1

Add the Missing Transitions!

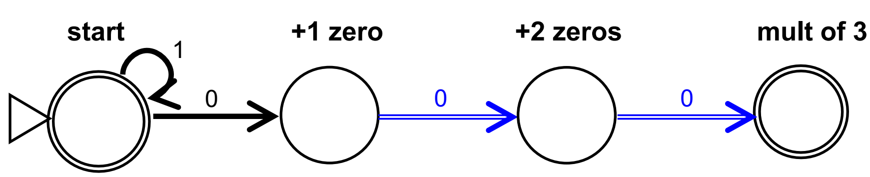
Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



- multiple of 3 = 0, 3, 6, 9, …
- number of 1s doesn't matter
- **accepted** strings include: 110101110, 11, 0000010
- **rejected** strings include: 101, 0000, 111011101111
- ***you may not need all four states!***

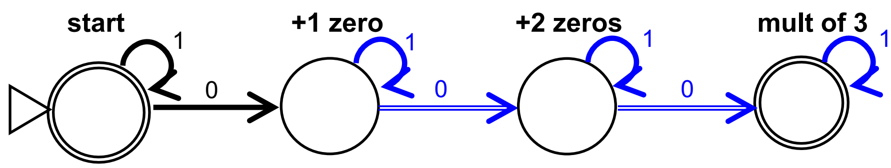
Add the Missing Transitions!

Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



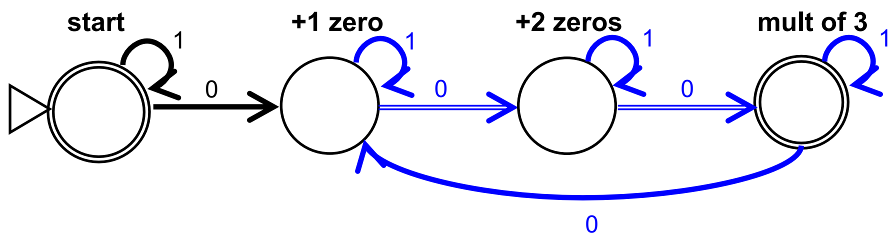
Add the Missing Transitions!

Construct a FSM accepting strings in which the **number of 0s** is a **multiple of 3**.



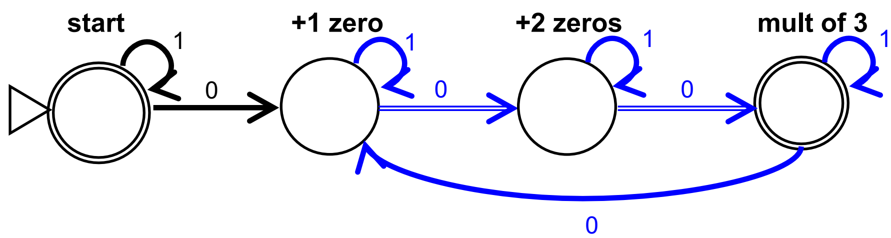
Number of 0s Is a Multiple of 3

Construct a FSM accepting strings in which the **number of 0s** is a **multiple of 3**.



Number of 0s Is a Multiple of 3

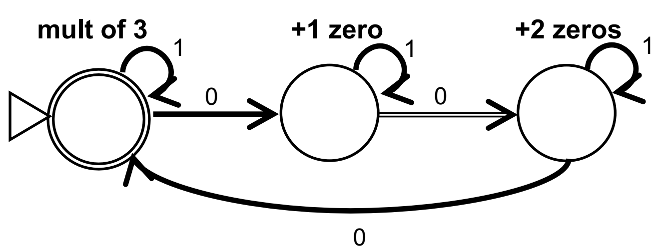
Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



How could this be simplified?

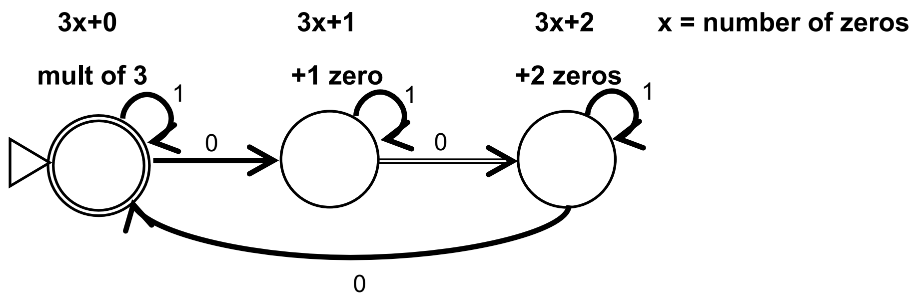
Number of 0s Is a Multiple of 3

Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



Number of 0s Is a Multiple of 3

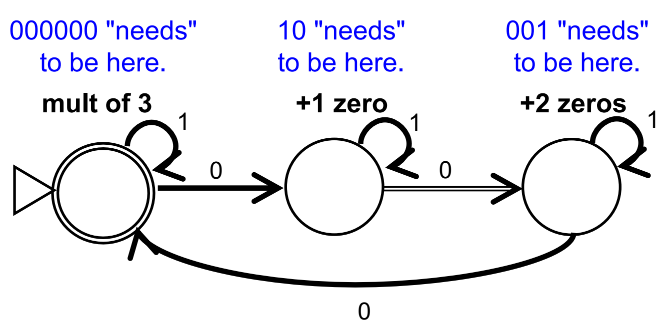
Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



Could we get by with even fewer states? ***No!***

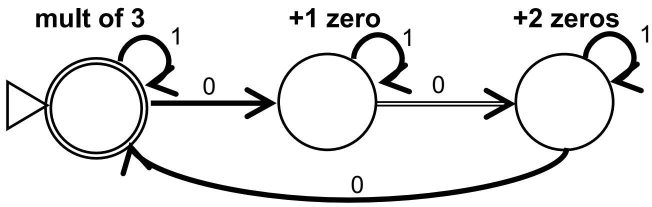
Number of 0s Is a Multiple of 3

Construct a FSM accepting strings in which
the **number of 0s** is a **multiple of 3**.



Could we get by with even fewer states? ***No!***

State == Set of Equivalent Input Strings



- Two input strings are ***not*** equivalent if adding the same characters to each of them produces a different outcome.
  - one of the resulting strings is accepted
  - the other is rejected
- Example: are '10' and '001' equivalent in the mult-of-3-0s problem?

  '10' + '00' → '1000' (accepted)

  '001' + '00' → '00100' (rejected)

  → '10' and '001' are *not* equivalent in this problem; they *must* be in *different* states!

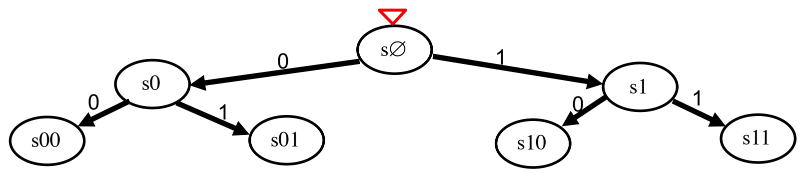
Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

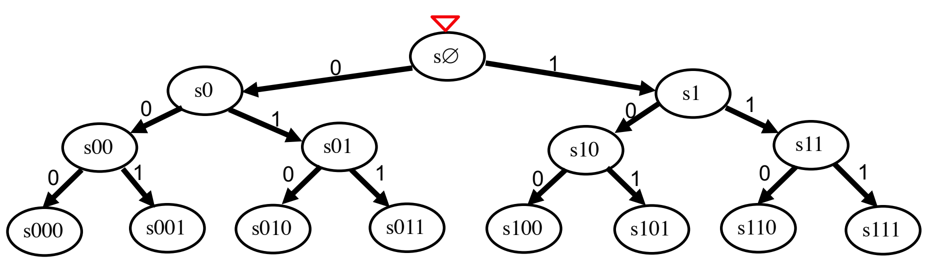
In theory, we could do something like this:



Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

In theory, we could do something like this:



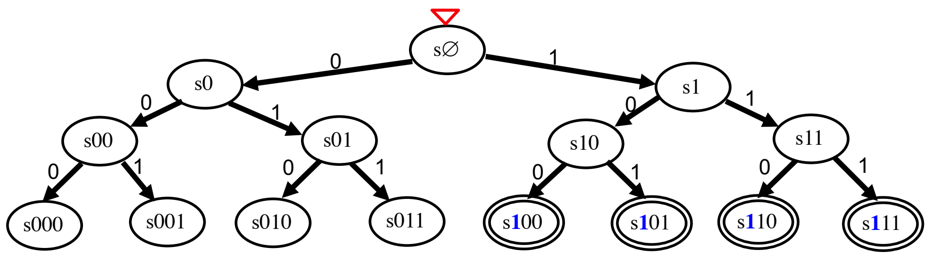
additional transitions are needed!

Which of these are accepting states?

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

In theory, we could do something like this:

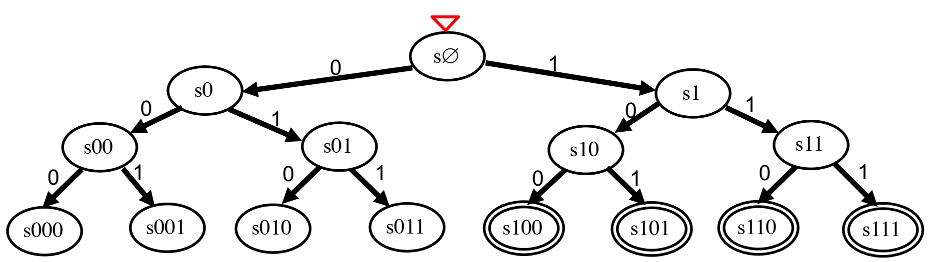


additional transitions are needed!

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

In theory, we could do something like this:



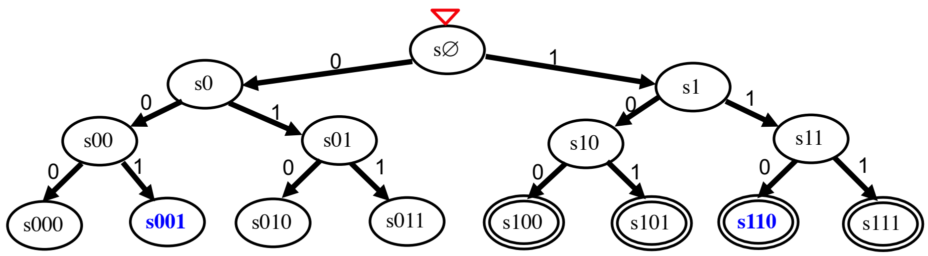
additional transitions are needed!

Which state should we enter if:
- we're in s111 and the next bit is a 0?
- we're in s100 and the next bit is a 1?

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which
the third-to-last bit is a 1.

In theory, we could do something like this:



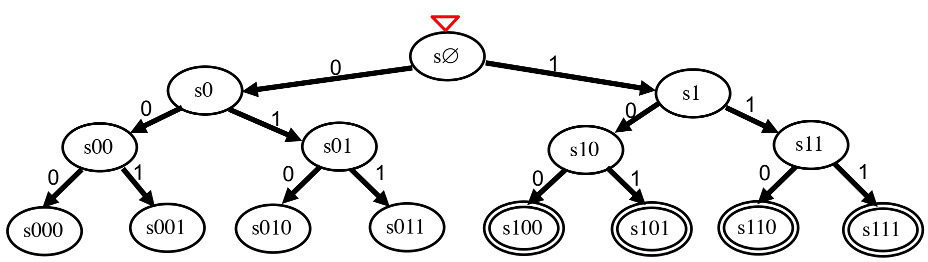
additional transitions are needed!

Which state should we enter if:
- we're in s111 and the next bit is a 0? s110 (111 + 0 → 1**110**)
- we're in s100 and the next bit is a 1? s001 (100 + 1 → 1**001**)

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which
the third-to-last bit is a 1.

How could we simplify this?

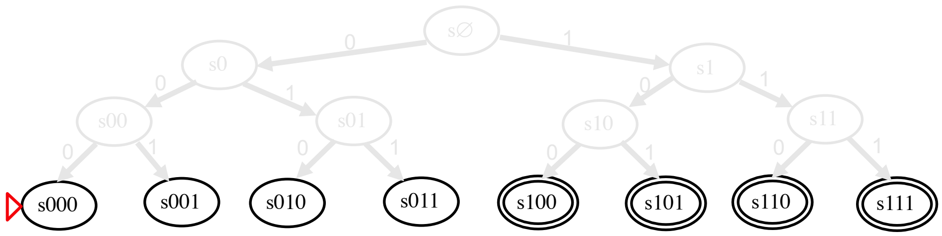


additional transitions are needed!

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

Because we only care about the last 3 bits, 8 states is enough!



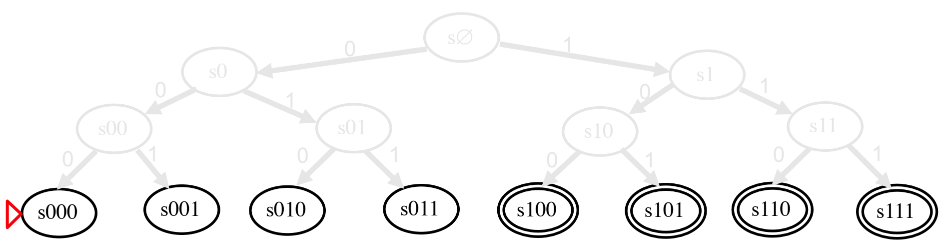
additional transitions are needed!

Examples of equivalent states:
- ø, 0, 00, 000: we're 3 transitions away from an accepting state
- 1, 01, 001: we're 2 transitions away from an accepting state

Third-to-Last Bit Is a 1

Construct a FSM accepting strings in which the third-to-last bit is a 1.

Because we only care about the last 3 bits, 8 states is enough!



additional transitions are needed!

Could we get by with even fewer? ***No!***

Final Project: Stemming

- word → *stem*/*root* of the word
- Examples:

```
stem('party')       ⇨  'parti'
stem('parties')     ⇨  'parti'

stem('love')        ⇨  'lov'
stem('loving')      ⇨  'lov'

stem('stems')       ⇨  'stem'
stem('stemming')    ⇨  'stem'
stem('stem')        ⇨  'stem'
```

There's No "Right Answer"!

- Example: Rather than doing this:

```
stem('party')       ⇨  'parti'
stem('parties')     ⇨  'parti'
```

we could do this instead

```
stem('party')       ⇨  'party'
stem('parties')     ⇨  'party'
```

Which Word(s) Does It "Get Wrong"?

```
def stem(word):
    if word[-3:] == 'ing':
        word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...

    return word
```

A. `playing`

B. `stemming`

C. `spammer`

D. `reader`

E. more than one (which ones?)

How could you fix the ones it gets wrong?

Which Word(s) Does It "Get Wrong"?

```
def stem(word):
    if word[-3:] == 'ing':
        word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...

    return word
```

A. `playing`

B. **`stemming`**

C. `spammer`

D. **`reader`**

E. more than one (which ones?)

How could you fix the ones it gets wrong?

Be Careful!

```
def stem(word):
    if word[-3:] == 'ing':
        if word[-4] == word[-5]:
            word = word[:-4]
        else:
            word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...

    return word
```

`stem('stemming')` → `'stem'`

`stem('killing')` → `'kil'`

`stem('sing')` → `IndexError`
(original version gave `'s'`)

Things to Consider When Stemming

- You could include the length of the word in some rules.
- You could use a dictionary of special cases.
- Be careful about the order in which rules are applied.
- Consider the use of recursion in some cases:

```
stem('readers')
    remove the 's' to get 'reader'
    stem_rest = stem('reader')
                    remove 'er' to get 'read'
                    return 'read'
    return stem_rest
    ⇨       'read'
```

- ***It doesn't need to be perfect!*** *(see assignment for minimum requirements)*