- This lab will cover <u>Queues</u>.
- It is assumed that you have reviewed chapters 6 of the textbook. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
- When approaching the problems, <u>think before you code</u>. Doing so is good practice and can help you lay out possible solutions.
- <u>Think of any possible test cases</u> that can potentially cause your solution to fail!
- If you finish early, you may leave early after showing the TA your work. Or you may stay and help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time. <u>Ideally you should not spend more time than suggested for each problem.</u>
- Your TAs are available to answer questions in lab, during office hours, and on Piazza.

Vitamins

1. What is the output of the following code?

```
q = ArrayQueue()
i = 2

q.enqueue(1)
q.enqueue(2)
q.enqueue(4)
q.enqueue(8)

i += q.first()
q.enqueue(i)
q.dequeue()
q.dequeue()
print(i)
print(q.first())
```

2. Describe what the following function does and give it an appropriate name. Trace the function with a queue of integers.

```
def mystery(q):
    if (q.is_empty()):
        return

    else:
        val = q.dequeue()
        mystery(q)
        if val % 2 != 0:
```

```
            q.enqueue(val)
```

3. Trace the following function with different string inputs. Describe what the function does, and give a meaningful name to the function:

```python
def mystery(input_str):
    s = ArrayStack()
    q = ArrayQueue()

    for char in input_str:
        s.push(char)
        q.enqueue(char)

    while not s.is_empty():
        if s.pop() != q.dequeue():
            return False

    return True
```

4. Consider the "circular" array implementation of a queue, similar to `ArrayQueue` that we studied in class, where the only difference is that the initial capacity is set to 4

```python
class ArrayQueue:

    INITIAL_CAPACITY = 4

    def __init__(self):

        self.data_arr= make_array(ArrayQueue.INITIAL_CAPACITY)

        self.num_of_elems = 0

        self.front_ind = None

    def __len__(self): ...

    def is_empty(self): ...

    def enqueue(self, elem): …

    def dequeue(self): ...

    def first(self): ...

    def resize(self, new_cap): ...
```

Show the values of the data members: `front_ind`, `num_of_elems`, and the contents of each `data_arr[i]` after each of the following operations. If you need to increase the capacity of `data_arr`, add extra slots as described in class.

operation	front_ind	num_of_elems	data_arr
q=ArrayQueue()	None	0	[None, None, None, None]
q.enqueue('A')			
q.enqueue('B)			
q.dequeue()			
q.enqueue('C')			
q.dequeue()			
q.enqueue('D')			
q.enqueue('E')			
q.enqueue('F')			
q.enqueue('G')			
q.enqueue('H')			

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Implement the `MeanQueue` class. The `MeanQueue` enqueues only integers and floats and rejects any other data type (bool, str, etc). It can also provide the sum and average of all numbers stored in **O(1) run-time**. **You may define additional member variables of O(1) extra space for this ADT.**

   The MeanQueue will use an ArrayQueue as its underlying data member. To test the data type, you may use the "type(var)" function in python.

   ```python
   class MeanQueue:

       def __init__(self):
           self.data = ArrayQueue()
           ...

       def __len__(self):
       '''Return the number of elements in the queue'''

       def is_empty(self):
       ''' Return True if queue is empty'''


       def enqueue(self, e):
       ''' Add element e to the front of the queue. If e is not
       an int or float, raise a TypeError '''


       def dequeue(self):
       ''' Remove and return the first element from the queue. If
       the queue is empty, raise an exception'''


       def first(self):
       ''' Return a reference to the first element of the queue
       without removing it. If the queue is empty, raise an
       exception '''


       def sum(self):
   ```

```
''' Returns the sum of all values in the queue'''



def mean(self):
''' Return the mean (average) value in the queue'''
```

Download the **ArrayQueue.py** file under Resources/Lectures on Brightspace.
```
from ArrayQueue import *
```

2. Implement the `ArrayDeque` class, which is an **array based implementation** of a Double-Ended Queue (also called a deque for short).

   A deque differs from a queue in that elements can be inserted to and removed from both the front and the back. (Think of this as a queue and stack combined).

   Like the ArrayQueue and ArrayStack, the standard operations for an ArrayDeque should occur in **O(1) amortized runtime**. **You may want to use and modify the ArrayQueue implementation done in lectures.**

   Your implementation should include the following methods:
   - `__init__`(self): *Initializes an empty Deque using an array as self.data*

   - `__len__`(self): *Return the number of elements in the Deque*

   - `is_empty`(self): *Return True if the deque is empty*

   - `first`(self): *Return (but don't remove) the first element in the ArrayDeque. Raise an Exception if it is empty*

   - `last`(self): *Return (but don't remove) the last element in the ArrayDeque. Raises an Exception if it is empty*

   - `enqueue_first`(self, elem): *Add elem to the front of the ArrayDeque*

   - `enqueue_last`(self, elem): *Add elem to the back of the ArrayDeque*

   - `dequeue_first`(self): *Remove and return the first element from the Deque. Raise an Exception if the ArrayDeque is empty*

   - `dequeue_last`(self): *Remove and return the last element from the Deque. Raise an Exception if the ArrayDeque is empty*

3. Write an **iterative function** that flattens a <u>nested list by its nesting depth level</u> using one **ArrayQueue** and its <u>defined methods</u>. You may create a new list to store the results. The original list should remain unchanged. You may not use any other data structures.

The nesting depth of an integer num in a nested list, is the number of "[" that are actively open when reading the string representation of the list from left to right until reaching the point where num appears.

ex)
```
lst = [ [[[0]]], [1, 2], 3, [4, [5, 6, [7]], 8], 9]
new_lst = flatten_list_by_depth(lst)
print(new_lst) → [3, 9, 1, 2, 4, 8, 5, 6, 0, 7]
```

- The nesting depth of 3 and 9 is 1 → [ ].
- The nesting depth of 1, 2, 4 and 8 is 2 → [ ].
- The nesting depth of 5 and 6 is 3 → [ ].
- The nesting depth of 0 and 7 is 4 → [ ].

```
def flatten_list_by_depth(lst):
    """
    : lst type: list
    : return type: list
    """
    q = ArrayQueue()
    new_lst = []
    ''' Remove and return the top element from the stack. If the
stack is empty, raise an exception'''

    return new_lst
```

<u>Hint:</u> *A queue follows a first in first out order. Start by placing all the values of the list into a queue. What will you do if the front of the queue is an integer? What will you do if the front of the queue is a list?*

4. Implement a *Stack* using just a *Queue* as the main underlying data collection. **You may only access the ArrayQueue's methods which include:** `len`, `is_empty`, `enqueue`, `dequeue`, and `first`.

Implement two different class implementations with different push & pop/top methods:

a. Implement one class that optimizes push. That is, the cost of push will be constant amortized, however there is no limit on the cost of pop.

b. Implement another class that optimizes pop. That is, the cost of pop will be constant amortized, however there is no limit on the cost of push.

c. Analyze the worst case run-time of your two sets of implementations for push and pop/top methods.

Your implementation should be like so:

```python
class QueueStack:

    def __init__(self):
        self.data = ArrayQueue()

    def __len__(self):
        return len(self.data)

    def is_empty(self):
        return len(self) == 0

    def push(self, e):
    ''' Add element e to the top of the stack '''

    def pop(self):
    ''' Remove and return the top element from the stack. If the stack
    is empty, raise an exception'''

    def top(self):
    ''' Return a reference to the top element of the stack without
    removing it. If the stack is empty, raise an exception '''
```

5. **OPTIONAL**

In homework 1, you implemented a generator function that produces the Fibonacci sequence. The sequence is like so: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … , with every subsequent number being the sum of the previous 2 numbers.

For this question, you will implement a generator function for the **n - bonacci sequence**. Every n - bonacci sequence starts off with n 1's, and every subsequent number in the sequence is the sum of the previous n numbers. The function will take two parameters, n for the bonacci value, and k, indicating the first k values in that sequence.

Note that for n = 2, you will get the Fibonacci sequence.

```python
def n_bonacci(n, k):
    """

    : n, k type: int
    : yield type: int
    """
```

ex) 4 - bonacci would look like this:

```python
for i in n_bonacci(4, 9):   #first 9 values
    print(i, end = " ")     #1, 1, 1, 1, 4, 7, 13, 25, 49

for i in n_bonacci(4, 2):   #first 2 values
    print(i, end = " ")     #1, 1
```

ex) 2 - bonacci (fibonacci) would look like this:

```python
for i in n_bonacci(2, 10): #first 10 values
    print(i, end = " ")     #1, 1, 2, 3, 5, 8, 13, 21, 34, 55

for i in n_bonacci(2, 1):   #first value
    print(i, end = " ")     #1
```

You may only use an ArrayQueue with additional O(1) extra space for your solution. To further save space, you may want to only hold up to n values at any time.

Give the worst-case run-time and extra space complexity of the following in terms of n and k:

```python
for i in n_bonacci(n, k):
    print(i, end = " ")
```