- This lab will cover Queues.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

---

**Vitamins (maximum 45 minutes)**

---

1. What is the output of the following code?

```
import ArrayQueue

q = ArrayQueue.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. Draw the memory image of the following ArrayQueue object as elements are added and deleted. Keep track of the number_of_elems and indicates the values at each index.

```
import ArrayQueue

q = ArrayQueue.ArrayQueue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.enqueue(6)
q.enqueue(7)
q.dequeue()
q.dequeue()
q.dequeue()
q.enqueue(8)
q.enqueue(9)
q.enqueue(10)
q.enqueue(11)
q.enqueue(12)
```

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

```
from ArrayQueue import *
from ArrayStack import *

def mystery(string_input):
    stack1 = ArrayStack()
    queue1 = ArrayQueue()

    for char in string_input:
        stack1.push(char)
        queue1.enqueue(char)

    while not stack1.is_empty():
        if stack1.pop() != queue1.dequeue():
            return False

    return True
```

## Coding

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

1. Implement a double-ended queue (deque). A deque differs from a queue in that elements can be inserted at the front and back of the queue and elements can be removed from the front and the back of the queue. The standard operations for a deque still occur in **O(1) amortized runtime** like a queue. Your implementation should include the following methods:

   You should use a circular array to implement the deque. You can modify the `ArrayQueue` class that was implemented in lecture.

   a.)
   ```
   def __len__(self):
       '''Return the number of elements in the deque.'''
   ```

   b.)
   ```
   def is_empty(self):
       '''Return True if the deque is empty.'''
   ```

   c.)
   ```
   def first(self):
       '''Return (but don't remove) the first element in the deque.'''
   ```

   d.)
   ```
   def last(self):
       '''Return (but don't remove) the last element in the deque.'''
   ```

   e.)
   ```
   def add_first(self, elem):
       '''Add element to the front of the deque.'''
   ```

   f.)
   ```
   def add_last(self, elem):
       '''Add element to the back of the deque.'''
   ```

   g.)
   ```
   def delete_first(self):
       '''Remove and return the first element from the deque. Return an
   ```

```
        error if the deque is empty.'''
```

h.)
```
def delete_last(self):

    '''Remove and return the last element from the deque. Return an
    error if the deque is empty.'''
```

2. In this question, we will suggest a data structure to implement a *boost queue ADT*. A *boost queue* is like a queue, but it also supports a special "boost" operation, that moves the element in the back of the queue a few steps forward.

   The operations of the *boost queue ADT* are:

   - `__init__(self)`: creates an empty `BoostQueue` object.
   - `__len__(self)`: returns the number of elements in the queue
   - `is_empty(self)`: returns `True` if and only if the queue is empty
   - `enqueue(self, elem)`: adds `elem` to the back of the queue.
   - `dequeue(self)`: removes and returns the element at the front of the queue. If the queue is empty an exception is raised.
   - `first(self)`: returns the element in the front of the queue without removing it from the queue. If the queue is empty an exception is raised.
   - `boost(self, k)`: moves the element from the back of the queue k steps forward. If the queue is empty an exception is raised. If k is too big (greater or equal to the number of elements in the queue) the last element will become the first.

   For example, your implementation should provide the following behavior:

```
>>> boost_q = BoostQueue()
>>> boost_q.enqueue(1)
>>> boost_q.enqueue(2)
>>> boost_q.enqueue(3)
>>> boost_q.enqueue(4)
>>> boost_q.boost(2)
>>> boost_q.dequeue()
1
>>> boost_q.dequeue()
```

```
4
>>> boost_q.dequeue()
2
>>> boost_q.dequeue()
3
```

**Implementation requirements:**

- All queue operations should run in θ(1) worst case, besides the `boost(k)` operation, which should run in θ(k) worst case.
- You should use a circular array in your implementation. You can modify the `ArrayQueue` class to create the `BoostQueue` class.

3. Write a nonrecursive python function to find the total sum of a nested list of integers. Use a queue to accomplish this.

   For example, If the input list is `[[1, 2], [3, [[4], 5]], 6]`, the output should be 21.