- This lab will cover stacks and queues.
- 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, think before you code. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- Your TAs are available to answer questions in the lab, and during office hours.

Part A
Stacks
(2:00 PM - 4:00 PM)

Vitamins (25 minutes)

---

1. What is the output of the following code? (5 minutes)

```
s = ArrayStack()
i = 2

s.push(1)
s.push(2)
s.push(4)
s.push(8)

i += s.top()
s.push(i)
s.pop()
s.pop()
print(i)
print(s.top())
```

2. Trace the following function with different list inputs. Describe what the function does, and give a meaningful name to the function: (5 minutes)

```
def mystery(lst):
    s = ArrayStack()
    for i in range(len(lst)):
        s.push(lst.pop())
    for i in range(len(s)):
        lst.append(s.pop())
```

3. Trace the following function, which takes in a stack of integers. Describe what the function does, and give a meaningful name to the function: (5 minutes)

```
def mystery(s):
   if len(s) == 1:
     return s.top()
   else:
     val = s.pop()
     result = mystery(s)

     if val < result:
          result = val
     s.push(val)
     return result
```

4. Fill out the prefix, infix, postfix table below: (10 minutes)

Prefix	**Infix**	**Postfix**	**Value**
- * 3 4 10	3 * 4 - 10		2
	(5 * 5) + (10 / 2)	5 5 * 10 2 / +	30
		10 2 - 4 / 8 +	
+ * 6 3 * 8 4			
	(8 * 2) + 4 - (3 + 6)		

Coding (60 minutes)

In this section, it is strongly recommended that you solve the problem on paper before writing code. Note that you should not access the underlying list in the ArrayStack implementation. Treat it as a **black box** and only use the **len, is_empty, push, top, and pop** methods.

Download the **Lab4_ArrayStack.py** file on Brightspace
Note: import the class like so → `from Lab4_ArrayStack import *`

1. Write a **recursive function** that takes in a Stack of integers and returns the sum of all values in the stack. Do not use any helper functions or change the function signature. Note that the stack should be restored to its original state if you pop from the stack. (10 minutes)

   ex) s contains [1, -14, 5, 6, -7, 9, 10, -5, -8] from top → bottom.
   stack_sum(s) returns -3

   ```
   def stack_sum(s):
       """
       : s type: ArrayStack
       : return type: int
       """
   ```

   Hint: See how the stack is restored in the code snippet from vitamins question 3.

2. Create an **iterative function** that evaluates a valid prefix string expression. You may only use **one ArrayStack** as an additional data structure to the given setup. Do not use any helper functions or change the function signature.

   In addition, each character is separated by one white space and numbers may have more than one digit. Therefore, we will use the split function to create a new list of each substring of the string separated by a white space. You may assume all numbers will be positive. (30 minutes)

   ex) exp_str is " - + * 16 5 * 8 4 20"
   exp_lst = exp.split(" ") → ["-", "+", "*", "16", "5", "*", "8", "4", "20"]
   eval_prefix(exp_str) returns = 92

```
def eval_prefix(exp_str):
      """
      : exp type: str
      : return type: int
      """
      exp_lst = exp_str.split( )
```

Hint:
To check if a string contains digits, use `.isdigit( )`.
To check if a string is an operator, you may want to do `if char in "-+/*"` similar to how you checked for vowels.

As you parse the expression lst, think about when you would push/pop the operator or number to/from the stack. Try to trace this execution on paper first before writing your code.

Test your code with the various prefix expressions from the Vitamins q4.

3. Write an **iterative functio**n that flattens a nested list while retaining the left to right ordering of its values using one **ArrayStack** and its defined methods. That is, you should not directly access the underlying array in the implementation. Do not use any helper functions or change the function signature. (20 minutes)

   In addition, do not create any other data structure other than the ArrayStack.

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

```
def flatten_list(lst):
      """
      : lst type: list
      : return type: None
      """
      s = ArrayStack()
```

Hint: You may want to traverse the list from the end for 2 reasons: pop and append has an amortized cost of O(1) and a stack reverses the collection order because of FIFO.

Part B
Queues
(4:10 PM - 5:50 PM)

Vitamins (30 minutes)

---

1. What is the output of the following code? (5 minutes)

```
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. (5 minutes)

```
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: (5 minutes)

```
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 this "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.

```
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. (15 minutes)

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 (30 minutes)

In this section, it is strongly recommended that you solve the problem on paper before writing code. Note that you should not access the underlying list in the ArrayStack implementation. Treat it as a **black box** and only use the **len, is_empty, enqueue, first, and dequeue** methods.

Download the **Lab4_ArrayQueue.py** file on Brightspace
Note: import the class like so → `from Lab4_ArrayQueue import *`

1. Using the `ArrayQueue` class as the underlying data structure, 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. (30 minutes)

```
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'''
```

Optional

2. Write an **iterative function** that flattens a nested list by its nesting depth level using one **ArrayQueue** and its defined methods. 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 = []
       . . .
       return new_lst
   ```

   Hint: *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?*