- This lab will cover more recursion.
- 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 lab, during office hours, and on Piazza.

Coding

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

Question 3 was an optional question on Lab 6. If you have already completed it, you may move on to Question 4.

1. Write a function that takes in two strictly increasing lists and returns the intersection of elements between them in a list. Implementation runtime must be O(n+m), where n is the size of lst1 and m is the size of lst2 (10 minutes)

```
ex) lst1 = [1,2,3,4]
    Lst2 = [3,4,5]
    intersectionOfLst(lst1,lst2) -> [3,4]

    def intersectionOfLst(lst1,lst2):
      """
      : lst1 type: list[int]
      : lst2 type: list[int]
      : return type: list[int]
```

2. Write a **recursive** function that takes in a positive integer n and returns True if it is a power of 2 and False otherwise. Implementation run-time must be logarithmic. (10 minutes)

```
ex) n = 1, isPowerOfTwo(n) -> True

    def isPowerOfTwo(n):
      """
      : n type: non-negative int
```

```
        : return type: bool
        """
```

3. Write a **recursive** function that takes in a list of integers and **mutates** it so that all the even numbers are at the front and all the odd numbers are in the back. You do not have to maintain the relative order; just focus on separating them. You are also given low and high, the range of indices to consider. Implementation run-time must be linear.
(15 minutes)

ex) lst = [4, -5, 2, 3, -1, -6, 7, 9, 0]
split_parity(lst) → [4, 0, 2, -6, -1, 3, 7, 9, -5]

```
def split_parity(lst, low, high):
    """
    : lst type: list[int]
    : low, high type: int
    : return type: None
    """
```

4. Write a function that takes in a list with elements from 0 to n-1, where n is the length of the list and sorts them in place. The input may contain duplicate elements. Ensure that the implementation has a linear runtime. (15 minutes)

```
ex) lst = [2,0,2,1,1,2] , SortLst(lst) -> [0,0,1,1,2,2]

    def SortLst(lst):
      """
      : lst type: list[int]
      : return type: None
      ""
```

5. A nested list of integers is a list that stores integers in some hierarchy. The list can contain integers and other nested lists of integers. An example of a nested list of integers is [[1, 2], 3, [4, [5, 6, [7], 8]]]. **(30 minutes)**
Write a **recursive** function to find the total sum of a nested list of integers.

ex) If lst = [[1, 2], 3, [4, [5, 6, [7], 8]]], the function should return 36.

```
def nested_sum(lst):
    """
    : lst type: list
    : output type: int
```

```
    """
```

Note:

To check the type of an object, use the isinstance function. You may use for loops inside your function. No run-time requirement.

ex)
```
lst = [1, 2, 3, 4]
if isinstance(lst, list): #returns True
if isinstance(lst, int): #returns False
```

6. Write a **recursive** function that returns the depth level of the nested list. **(30 minutes)**

```
def nested_depth_level(lst):
    """
    : lst type: list
    : output type: int
    """
```

The depth of a list is determined by its number of nesting.

ex) [1, 2] has depth = 1
[[1], 2] has depth = 2
If lst = [[1, 2], 3, [4, [5, 6, [7], 8]], [[[[9]]]]], nested_depth_level(lst) would return 5 because [[[[9]]]] has 4 levels of nesting and the whole thing is inside a list itself = 4 + 1 levels.
If lst = [[1, 2], 3, [4, [5, 6, [7], 8]]], nested_depth_level(lst) would return 4 because [4, [5, 6, [7], 8]] has 3 levels of nesting, with [7] being the deepest, and is inside a list itself = 3 + 1 = 4 levels.

OPTIONAL

1. Write a **recursive** function that reverses the order of values of each list in the hierarchy. **(30 minutes)**

ex)

If lst = [[1, 2], 3, [4, [5, 6, [7], 8]], [[[[9]]]]], deep_reverse(lst) should modify it so that it now has [[[[[9]]]] , [[8, [7], 6, 5], 4], 3, [2, 1]],

```
def deep_reverse(lst, low, high):
    """
    : lst type: list
    : low, high type: int
    : output type: None
    """
```

2. Write a recursive generator function that takes in a nested list, and yields each integer of the list, from left to right. Note that you do not need to flatten the list. **(20 minutes)**

```
def yield_flattened(lst):
    """
    : lst type: list
    : yield type: int
    """

def print_flattened(lst):
    print("[" + ",".join(str(num) for num in yield_flattened(lst))+
"]")
```

ex)

If lst = [[1, 2], 3, [4, [5, 6, [7], 8]], [[[[9]]]]]
print_flattened(lst) should output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Note:

`yield` value will only yield values from **one** function call. To yield values from another **recursive** call, you should have for elem in recursive_function(...): `yield` elem, or use `yield from` recursive_function(...).

ex)

You can shorten for i in range(n): yield i → yield from range(n)

```
def sample(n):                    def sample(n):
    for i in range(n):      →         yield from range(n)
        yield i
```