- This lab will review asymptotic analysis, searching, and recursion.
- It is assumed that you have reviewed **chapters 3 and 4 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!
- **You must stay for the duration of the lab**. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time. Ideally, you should not spend more time than suggested for each problem.
- Your TAs are available to answer questions in the lab, and during office hours.

Part A
Asymptotic Analysis and Searching
(2:00 PM - 3:50 PM)

Vitamins (30 minutes)

For **big-O proof**, if there exists constants c, and n_0 such that f(n) ≤ c*g(n) for every n ≥ n_0, then f(n) = O(g(n)).

For **big-Θproof**, if there exists constants c1, c2, and n_0 such that c1*g(n) ≤ f(n) ≤ c2*g(n) for every n ≥ n_0, then f(n) = Θ(g(n)).

For **big-Ω proof**, if there exists constants c, and n_0 such that f(n) ≥ c*g(n) for every n ≥ n_0, then f(n) = O(g(n)).

1. Use the **formal proof of big-O and big-Θ** in order to show the following (10 minutes):

   a) $n^2 + 5n - 2$ is $O(n^2)$

   b) $\frac{n^2 - 1}{n + 1}$ is $O(n)$

   c) $\sqrt{5n^2 - 3n + 2}$ is $\Theta(n)$

2. State **True** or **False** and explain why for the following (5 minutes):

a) $8n^2(\sqrt{n})$ is $O(n^3)$

b) $8n^2(\sqrt{n})$ is $\Theta(n^3)$

3. For each of the following $f(n)$, write out the summation results, and provide a tight bound $\Theta(f(n))$, using the $\Theta\ notation$ (5 minutes).

Given n numbers:

$1 + 1 + 1 + 1 + 1 \ldots + 1 =$ ____________ = Θ(___________)

$n + n + n + n + n \ldots + n =$ ____________ = Θ(___________)

$1 + 2 + 3 + 4 + 5 \ldots + n =$ ____________ = Θ(___________)

Given $log(n)$ numbers, where *n* is a power of *2*:

$1 + 2 + 4 + 8 + 16 \ldots + n =$ ____________ = Θ(___________)

$n + n/2 + n/4 + n/8 \ldots + 1 =$ ____________ = Θ(___________)

$1 + 2 + 3 + 4 \ldots + log(n) =$ ____________ = Θ(___________)

4. For each of the following code snippets, find $f(n)$for which the algorithm's time complexity is $\Theta(f(n))$in its **worst case** run and explain why. (10 minutes)

```
a) def func(lst):
       for i in range(len(lst)):
           if (len(lst) % 2 == 0):
               return
```

```
b) def func(lst):
       for i in range(len(lst)):
           if (lst[i] % 2 == 0):
               print("i =", i)
           else:
               return
```

Optional (4c - 4e)

```
c) def func(lst):
       for i in range(len(lst)):
           for j in range(len(lst)):
               if (i+j) in lst:
                    print("i+j = ", i+j)
```

```
d) def func(n):
       for i in range(int(n**(0.5))):
           for j in range(n):
               if (i*j) > n*n:
                     print("i*j = ", i*j)
```

```
e) def func(n):
       for i in range(n//2):
           for j in range(n):
               print("i+j = ", i+j)
```

Coding (45 minutes)

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

1. Given a list of values (`int, float, str, …`), write a function that reverses its order in-place. You are not allowed to create a new list. Your solution must run in Θ(n), where n is the length of the list (15 minutes).

   You are not allowed to use any list methods like pop or append that modify the list

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

   Example:

   lst = [1, 2, 3, 4, 5, 6]
   reverse_list(lst)
   print(lst) → [6, 5, 4, 3, 2, 1]

2. Given a **sorted** list of positive integers with zeros mixed in, write a function to move all zeros to the end of the list while maintaining the order of the non-zero numbers. For example, given the list [0, 1, 0, 3, 13, 0], the function will modify the list to become [1, 3, 13, 0, 0, 0]. Your solution must be in-place and run in Θ(n), where n is the length of the list. (15 minutes)

   You are not allowed to use any list methods like pop or append that modify the list

   ```
   def move_zeros(nums):
         """
         : nums type: list[int]
         : return type: None
         """
   ```

   Hint: You should traverse the list with 2 pointers, both starting from the beginning. One pointer will traverse through the entire list but when should the other pointer move?

3. In class, you learned how to find a number in a **sorted list of numbers** in $\Theta(log(n))$ run-time with *binary search*.

   Now, given a **reverse sorted** list of **unique lowercase letters** (z -> a), and a letter to look for, write a function that returns the index of the letter or -1 if the letter does not exist in the list. Your solution must not modify the list and run in $\Theta(log(n))$.
   (15 minutes)

```python
def binary_search_lowercase(lst, char):
    """
    : lst type: list[str]
    : char type: str
    : return type: int
    """
```

   Example:

   lst = ["z", "t", "m", "k", "g", "f", "d", "a"]
   print(binary_search_lowercase(lst, "f")) → 5
   print(binary_search_lowercase(lst, "t")) → 1
   print(binary_search_lowercase(lst, "q")) → -1

Part B
Recursion
(4:00 PM - 5:50 PM)

Coding (65 mins)

---

1. Write a **recursive** function that returns the **sum** of all numbers from 0 to n. (5 minutes)

```
def sum_to(n):
     """
     : n type: int
     : return type: int
     """
```

2. Give a **recursive** implementation for the binary search algorithm. The function is given a **sorted** list, lst, a value, val, to search for, and two indices, low and high, representing the lower and upper bounds to consider. (15 minutes)

If the value is on the list (between low and high), return the index at which the value is located. If val is not found, the function should return -1.

```
def binary_search(lst, low, high, val):
     """
     : lst type: list[int]
     : val type: int
     : low type, high type: int
     : return type: int
     """
```

3. Write a **recursive** function to find the maximum element in a **non-empty**, **non-sorted** list of numbers. ex) if the input list is [13, 9, 16, 3, 4, 2], the function will return 16.

a. Determine the runtime of the following implementation. (7 minutes)

```
def find_max(lst):
    if len(lst) == 1:
        return lst[0]       #base case
    prev = find_max(lst[1:])
    if prev > lst[0]:
        return prev
    return lst[0]
```

b. Update the function parameters to include low and high. Low and high are int values that are used to determine the range of indices to consider. <u>Implementation run-time must be linear</u>. (13 minutes)

```
def find_max(lst, low, high):
    """
    : lst type: list[int]
    : low, high type: int
    : return type: int
    """
```

4. Given a string of letters representing a word(s), write a **recursive** function that returns a **tuple** of 2 integers: the number of vowels, and the number of consonants in the word.

Remember that <u>tuples are not mutable</u> so you'll have to create a new one to return each time when you're updating the counts. Since we are always creating a tuple of 2 integers each time, the cost is constant. <u>Implementation run-time must be linear</u>. (25 minutes)

ex) word = “NYUTandonEngineering”
vc_count(word, 0, len(word)-1) → (8, 12) # 8 vowels, 12 consonants

```
def vc_count(word, low, high):
    """
    : word type: str
    : low, high type: int
    : return type: tuple (int, int)
    """
```

Vitamins (15 minutes)

1. For each of the following code snippets:
    a. Given the following inputs, trace the execution of each code snippet. Write down all outputs in order and what the functions return.
    b. Analyze the running time of each. For each snippet:
        i. Draw the recursion tree that represents the execution process of the function, and the cost of each call
        ii. Conclude the total (asymptotic) run-time of the function.

a.
```
def func1(n):          #  Draw out func1(16)
     if (n <= 1):
          return 0
     else:
          return 10 + func1(n-2)
```

b.
```
def func2(n):          #  Draw out func2(16)
     if (n <= 1):
          return 1
     else:
          return 1 + func2(n//2)
```

c.
```
def func3(lst) #  Draw out func3([1, 2, 3, 4, 5, 6, 7, 8])
     if (len(lst) == 1):
          return lst[0]
     else:
          return lst[0] + func3(lst[1:])
```