- This lab will cover asymptotic analysis, problem solving, 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, during office hours, and on Piazza.

Vitamins (55 minutes)

---

1. For each section below, write the correct output shown after the Python code is run. Explain your answer by **drawing the memory image** for the execution of these lines of code. That is, you should draw the variables as they are organized in the call stack, and the data they each point to. (20 minutes)

   a.

```
import copy
lst = [1, [2, "abc"], [3, [4]], 7]
lst_deepcopy = copy.deepcopy(lst)
lst[0] = 10
lst[1][1] = "ABC"
lst_deepcopy[2][1][0] = 40

print(lst)

_____________________________________
print(lst_deepcopy)

_____________________________________
```

   b.

```
lst = [1, [2, 3], ["a", "b"] ]
lst_slice = lst[:]
lst_assign = lst
lst.append("c")
for i in range(1, 3):
      lst_slice[i][0] *= 2
```

```
print(lst)

_____________________________________

print(lst_slice)

_____________________________________

print(lst_assign)

_____________________________________
```

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)). **(20 mins problems below)**

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

a) $n^2 + 5n - 2$ is $\Theta(n^2)$

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

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

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

3. 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 $\Theta(n^3)$

c) $16\ log(n^2)\ +\ 2$ is $O(log(n))$

4. 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:

$1 + 1 + 1 + 1 + 1 \ldots + 1 =$ _____ = $\Theta($ _____ $)$

$n + n + n + n + n \ldots + n =$ _____ = $\Theta($ _____ $)$

$1 + 2 + 3 + 4 + 5 \ldots + n =$ _____ = $\Theta($ _____ $)$

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

$1 + 2 + 4 + 8 + 16 \ldots + n =$ _____ = $\Theta($ _____ $)$

$n + n/2 + n/4 + n/8 \ldots + 1 =$ _____ = $\Theta($ _____ $)$

$1 + 2 + 3 + 4 \ldots + log(n) =$ _____ = $\Theta($ _____ $)$

5. 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. (5 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 (5c-5e)

```
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)
```

Optional (#6)

6. Sort the following 18 functions in an increasing asymptotic order and write <, <=, between each two subsequent functions to indicate if the first is asymptotically less than, asymptotically greater than or asymptotically equivalent to the second function respectively.

For example, if you were to sort: $f_1(n) = n$, $f_2(n) = log(n)$, $f_3(n) = 3n$, $f_4(n) = n^2$, your answer could be $log(n) < (n <= 3n) < n^2$

Hint: Try grouping the functions like so: linear, quadratic, cubic, exponential … etc

$$f_1(n) = n$$
$$f_2(n) = 500n$$
$$f_3(n) = \sqrt{n}$$
$$f_4(n) = log(\sqrt{n})$$
$$f_5(n) = \sqrt{log(n)}$$
$$f_6(n) = 1$$
$$f_7(n) = 3^n$$
$$f_8(n) = n \cdot log(n)$$
$$f_9(n) = \frac{n}{log(n)}$$
$$f_{10}(n) = 700$$
$$f_{11}(n) = log(n)$$
$$f_{12}(n) = \sqrt{9n}$$
$$f_{13}(n) = 2^n$$
$$f_{14}(n) = n^2$$
$$f_{15}(n) = n^3$$
$$f_{16}(n) = \frac{n}{3}$$
$$f_{17}(n) = \sqrt{n^3}$$
$$f_{18}(n) = n!$$

7. **(10 mins)** 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):          #for tracing: n = 16
    if (n <= 1):
        return 0
    else:
```

```
            return 10 + func1(n-2)
```

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

Coding

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

1.

a. 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 (10 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
    """
```

b. Modify the function to include low and high parameters that represent the positive indices to consider. Your function should reverse the list from index low to index high, inclusively. By default, low and high will be None so these parameters are optional. If they're both None (no parameters passed), set low to 0 and high to len(lst) - 1 just like in the previous function above.

You are not allowed to create a new list. Your solution must run in Θ(n), where n is the length of the list (5 minutes).

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

```
    : return type: None
    """
```

Example:

lst = [1, 2, 3, 4, 5, 6], low = 0, high = 5
`reverse_list(lst) #default, no parameters passed`
`print(lst)` → `[6, 5, 4, 3, 2, 1]`

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

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 $\Theta(n)$, where n is the length of the list. (25 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. Complete the following (35 minutes):

   a. The function below takes in a **sorted** list with n numbers, all taken from the range 0 to n, with one of the numbers removed. Also, none of the numbers in the list is repeated. The function searches through the list and returns the missing number.

      For instance, lst = [0, 1, 2, 3, 4, 5, 6, 8] is a list of 8 numbers, from the range 0 to 8, with the number 7 missing. Therefore, the function below will return 7.

      Analyze the worst case run-time of the function:

```
def find_missing(lst):

      for num in range(len(lst) + 1):
            if num not in lst:
                  return num
```

b. Rewrite the function so that it finds the missing number with a better run-time: **Hint:** the list is sorted. Also, make sure to consider the edge cases.

```
def find_missing(lst):
      """
      : nums type: list[int] (sorted)
      : return type: int
      """
```

c. Suppose the given list is **not sorted** but still contains all the numbers from 0 to n with one missing.

For instance, lst = [8, 6, 0, 4, 3, 5, 1, 2] is a list of numbers from the range 0 to 8, with the number 7 missing. Therefore, the function below will return 7.

How would you solve this problem? Do not use the idea in step a, or sort the list and reuse your solution in step b.

```
def find_missing(lst):
      """
      : nums type: list[int] (unsorted)
      : return type: int
      """
```

4. 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
      """
```

5. 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. (20 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 None.

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

OPTIONAL

6. 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. Implementation run-time must be linear. (10 minutes)

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

7. 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 tuples are not mutable 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. Implementation run-time must be linear. (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)
    """
```