- This lab will cover recursion.
- It is assumed that you have reviewed chapters 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 lab, during office hours, and on Edstem.

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:])
```

Coding

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

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. Write a **recursive** function that returns the product of all the even numbers from 1 to n **(assume n is passed in as an even number)**. (5 minutes)

   ex) if n = 8 → 2 * 4 * 6 * 8 = 384, so the function returns 384.

```
def product_evens(n):
    """
    : n 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. 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
    """
```

4. **Valid Palindrome – *Leetcode 125***
   A palindrome is a phrase where all characters are the same backward and forward. (must be recursive)

   Given a string `str` and its range of indices to consider, return `True` if it is a palindrome, or `False` otherwise. Assume all characters are lowercase. Must run in O(n) time.

   ex) is_palindrome("racecar", 0, 6) returns True # racecar
   is_palindrome("racecar", 1, 5) returns True # aceca
   is_palindrome("race car", 0, 6) returns False # empty space counts as a character
   is_palindrome("racecar", 1, 3) returns False # ace

```
def is_palindrome(str, low, high):
    """
    : str type: str
    : low, high type: int
    : return type: bool
    """
```

5. **Binary Search – *Leetcode 704***
   Given an array of integers `nums` which is sorted in ascending order, `low` and `high` indices, and an integer `target`, write a function to search for `target` in `nums`. If `target` exists between `low` and `high`, then return its index. Otherwise, return `None`.

   You must write an algorithm with O(log n) runtime complexity. (must be recursive)

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

6. **Vowel and Consonant Count**
   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)
    """
```

7. **OPTIONAL**

Given a list of integers, write a function to print the sum triangle of the array. Each row in the sum triangle is defined as the sum of each number and the one exactly after it.

Note that if there is no number after it, this number is removed. That is, there will always be one fewer number after each row.

Example, if `lst = [1, 2, 3, 4, 5]`, the sum triangle will be printed as followed:

```
[48]            ← # [20+28]
[20, 28]        ← # [8+12, 12+16]
[8, 12, 16]     ← # [3+5, 5+7, 7+9]
[3, 5, 7, 9]    ← # [1+2, 2+3, 3+4, 4+5]
[1, 2, 3, 4, 5] ← starting list
```

```
def list_sum_triangle(lst):
    """
    : lst type: list
    : output type: None
    """
```

Analyze the run-time of your implementation.

8. In class, you learned about *binary search*, which has a run-time of $O(log(n))$for searching through a **sorted list**. With binary search, the lower bound begins at index 0 while the upper bound is the last index, len(list) - 1.

You will write a modification of the binary search called *exponential search* (also called doubling or galloping search). With exponential search, we start at index i = 1 (after checking index 0) and we double i until it becomes the upper bound.

Let's try to find `15` in the given sample list by checking index `i = 0` first:

```
i = 0, (lst[0] = 1) != (val = 15):
```

[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

From there, you will make a comparison to see if the value is at index, i = 1. If not, you will double i until the index is out of range or until the value at the index is larger than the value you're searching for. **It is important for the two conditions to be in that order!**

`i = 1, (lst[i] = 3) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 2, (lst[i] = 6) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 4, (lst[i] = 10) < (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

`i = 8, (lst[i] = 22) > (val = 15):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

If the value is less than i or is out of bound, then you know it has to be between `i//2` and `i` (or `i//2` and `len(list)-1`). After confirming your bounds, you perform a binary search (in that range only).

`(lst[4] = 10) < val < (lst[8] = 22):`
[1, 3, 6, 7, 10, 12, 15, 20, 22, 24, 29, 33, 39, 55, 61, 64, 99, 101, 134, 150]

With the sample list above, we confirm that val must be between i = 4 and i = 8.

Recap:

1. Start with `i = 0` and check if `val == lst[0]` (if list is not empty).
2. If `val != lst[0]`, start the exponential search (doubling process) at `i = 1`.
3. If `val != lst[1]`, check if `val > lst[i]`. If so, double `i`.

4. Keep doubling while `i*2 < len(lst) and lst[i] < val`.
5. Set left bound to `i//2` and right bound to `i`
6. If `lst[right] < val`, then upper bound `right = len(lst)-1`
7. Perform binary search between the left and right bounds.

What is the run-time for *exponential search* and what advantage might this algorithm have over binary search? (35 minutes)

```
def exponential_search(lst, val):
"""
: lst type: list[int]
: val type: int
: return type: int(if found), None(if not found)

"""

     if len(lst) > 0:           #check if list is not empty
          if lst[0] == val:     #check index 0 first
               return 0
          else:
               i = 1 #start at 1 for the exponential search
               ...
     return None
```

Here is a simple test code to verify that your searching algorithm works.

```
#TEST CODE

lst = [-1111, -818, -646, -50, -25, -3, 0, 1, 2, 11, 33, 45, 46, 51,
58, 72, 74, 75, 99, 110, 120, 121, 345, 400, 500, 999, 1000, 1114,
1134, 10010, 500000, 999999]


#Testing exponential

print("\nTESTING VALUES IN LIST: exponential \n")

for val in lst:
     if exponential_search(lst, val) is None:
          print(val, "FAILED - DID NOT FIND")

#just to make sure you're not stuck in an infinite loop
print("TEST COMPLETED")
```