- This lab will review all material covered so far in lecture in preparation for the midterm.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 45 minutes)

---

1. What would be printed if the following line was entered at the console?

```
[x * 3 for x in range(5, 20, 3) if x * 3 % 7 != 0]
```

2. For each of the following code snippets:
   A. If *n=4*, trace the execution of each code snippet. Write down all outputs in order and what the functions returns.

   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) running time of the function

a.

```
def f(n):
   if (n == 1):
     return 1
   else:
     print("foo")
     return f(n-1) + f(n-1)
```

b.

```
def f(n):
  if (n == 1):
     print(1)
   else:
      f(n-1)
       for i in range(n):
            print(i, end = ' ')
          print('')
```

c.

```
def f(n):
  if (n == 1):
     print(1)
   else:
      f(n//2)
      for i in range(n):
         print(i, end = ' ')
        print('')
```

3. Suppose a program takes 0.05 seconds to run on an input size of 2048. Estimate how long it would run for an input size of 2^{13} if:
   a. The program had an $O(n)$ running time.
   b. The program had an $O(n^2)$ running time.
   c. The program had an $O(n^4)$ running time.

4. For each section below, write what would be the output. To explain your answer, **draw the memory image** for the execution of these lines of code.

```
import copy
lst = ['a', ['b', 1], ['c', [2, 3], 'd'], 'e']
lst_copy = copy.copy(lst)
lst_copy[0] = 'A'
lst_copy[1][1] += 1
lst_copy[2][1][0] *= 2
print(lst)
print(lst_copy)

lst = ['a', ['b', 1], ['c', [2, 3], 'd'], 'e']
lst_deepcopy = copy.deepcopy(lst)
lst_deepcopy[0] = 'A'
lst_deepcopy[1][1] += 1
lst_deepcopy[2][1][0] *= 2
print(lst)
print(lst_deepcopy)
```

Coding

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

1. Implement a function `def powers_of_two(num)`. This function is given a positive integer num, and evaluates to a generator, that when iterated over, will list num values of the sequence of powers of 2.

   For example, if we execute the following code:

   ```
   for curr_value in powers_of_two(6):
         print(curr_value)
   ```

   The expected output is: `2 4 8 16 32 64`

2. Write a python function to find the total sum of a nested list of integers.

   For example, If the input list is `[[1, 2], [3, [[4], 5]], 6]`, the output should be 21.

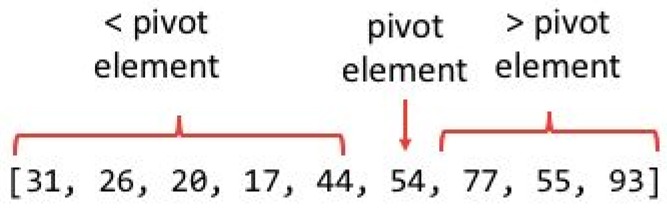
   Hint: Use recursion.

3. Write a function that takes a list and reorganizes it using lst[0] as a "pivot element." Items in the list that are less than the pivot element should come before items that are greater than the pivot element.

   For example, the following code :

   ```
   lst = [54, 26, 93, 17, 77, 31, 44, 55, 20]
   sort_first(lst)
   print(lst)
   ```

   should print the following list:



```
def sort_first(lst):
        """
        : input_int type: list
        : return value type: None
        """
```

Note: the order of the elements greater than and less than the pivot-value doesn't matter. You should perform the reorganization in-place. When you are done, think about how a series of recursive calls on smaller instances of the list could sort the list entirely.

4. Implement a MyString class. Use c-arrays as the internal data structure as we did for the MyList class. You may want to use the MyList class as a reference. In your implementation of the class, you must preserve the **immutable** property of strings - that is, no MyString class method should mutate the calling MyString object. Note that in creating new MyString objects, you are allowed to manipulate the c-array, but once the MyString object is created, no method should change or add to the character values in the c-array.

   Your class should include the following:

   a. A *constructor* that takes an `initial_str` as input and initiates a string object. If no argument is given at construction, the string will be empty.

   b. *len* method, which returns the length of the string object

   c. *iter* method, which allows iteration over the string object

   d. *repr* method, that returns a representation of the string object

   e. *get_item* method, which returns the value at a given index of the string. The method should raise an error if the index is out of range.

   f. *add* operator, which takes another string object as argument, and creates a new string object representing the concatenation of the two strings.

   g. *radd* operator

   h. *upper* method which returns a copy of the string with all uppercase characters

   i. Extra: Think about how you might implement the *iadd* operator.

An object of MyString should support the following code

```
>>> st1 = MyString()
>>> print(st1)

>>> st1 = st1 + "hi"
"hi"
>>> st2 = "hello"
>>> st2 + st1
"hellohi"
>>> st1.upper()
"HI"
```

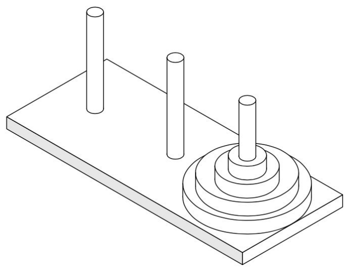
Extra:

1. Give a **recursive** implementation of a function that converts a value to binary. The function is given a positive integer value, `input_int`. The function will convert `input_int` into binary and return a string representing the binary version of the value.

   For example, calling the function with 6, will return "110"

```
      def decimal_to_binary(input_int):
            """
            : input_int type: int
            : return value type: str
            """
```

2. In the Towers of Hanoi puzzle, we are given a platform with three pegs: *a*, *b* and *c*, sticking out of it. On peg *a* is a stack of *n* disks, each larger than the next, so that the smallest is on top and the largest is on the bottom.



   The puzzle is to move all the disks from peg *a* to peg *c* (using peg *b* as an extra/temporary peg), moving one disk at a time, from the top of one peg to the top of another, so that **we never place a larger disk on top of a smaller one**.

   Here is a quick visual of Towers of Hanoi being solved: http://haubergs.com/hanoi

Observe the puzzle's solution for different number of disks. Try n = 2, 3, then 4.

Implement a **recursive** function: `def solve_hanoi(n, start, dest, spare)`

When called, this function solves the Towers of Hanoi puzzle for moving n disks from `start` to `dest` using `spare` as an extra peg. The function should print a sequence of lines describing the steps of the solution.

For example, the call: `solve_hanoi(3, ‘A’, ‘C’, ‘B’)` should print:

```
move disk from A to C
move disk from A to B
move disk from C to B
move disk from A to C
move disk from B to A
move disk from B to C
move disk from A to C
```

Hint: Consider first the subproblem of moving all but the *n*th disk from the `start` peg to the `spare` peg. Think about what inductive assumption you can make.

3. Advancing through an array (Source: Elements of Programming Interviews, page 67): In a particular board game, a player has to try to advance through a sequence of positions. Each position has a nonnegative integer associated with it, representing the maximum you can advance from that position in one move. You begin at the first position, and win by getting to the last position. For example, let `A = [3, 3, 1, 0, 2, 0, 1]` represent the board game, i.e, the *i*th entry in `A` is the maximum we can advance from *i*. The game can be won by the following sequence of advances through A:

   - take 1 step from A[0] to A[1]
   - take 3 steps from A[1] to A[4]
   - take 2 steps from A[4] to A[6]

   Note that A[0] = 3 >= 1, A[1] = 3 >= 3, and A[4] = 2 >= 2, so all moves are valid. If A instead was [3, 2, 0, 0, 2, 0, 1], it would not be possible to advance past position 3, so the game cannot be won.

   Write a program which takes an array of n integers, where `A[i]` denotes the maximum you can advance from index *i*, and returns whether it is possible to advance to the last index starting from the beginning of the array.

   Hint: Analyze each location, starting from the beginning.