Vitamins

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.

   a.

```
lst = [1, 2, 3]
lst2 = lst
lst.append(4)
lst2.append(5)

print(lst)

_____________________________________________
print(lst2)

_____________________________________________
```

   b.

```
s = "aBc"
s = s.upper()
t = s
t = t.lower()

print(s)

_____________________________________________
print(t)

_____________________________________________
```

   c.

```
s = "abc"
def func(s):
    s = s.upper()
    print("Inside func s =", s)

func(s)

_____________________________________________
```

```
print(s)
```

_____

d.

```
lst = [1, 2, 3]
def func(lst):
     lst.append(4)
     lst = [5, 6, 7, 8]
     print("Inside func lst =", lst)

func(lst)
```

_____

```
print(lst)
```

_____

2. 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.

a.

```
import copy
lst = [1, 2, [3, 4]]
lst_copy = copy.copy(lst)
lst[0] = 10
lst_copy[2][0] = 30


print(lst)
```

_____

```
print(lst_copy)
```

_____

b.

```
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)

_____________________________________________
```

c.

```
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)

_____________________________________________
```

3. 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.

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
```

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

1. Implement the following function (30 minutes):

```
def can_construct(word , letters):
    """
    word - type: str
    letters - type: str
    return value - type: bool
    """
```

This function is passed in a string containing a word, and another string containing letters in your hand. When called, it will return True if the word can be constructed with the letters provided; otherwise, it will return False.

Notes:

- Each letter provided can only be used one.
- You may assume that the word and letters will only contain lower-case letters.
- **You may not use a dictionary for this question.**
- Hint : Try to think about how you can use a list to implement a dictionary.

ex) `can_construct("apples", "aples")` will return `False`.

ex) `can_construct("apples", "aplespl")` will return `True`.

2. Implement the `UnsignedBinaryInteger` class to represent non-negative integers by their binary (base 2) representation

   a. Decimal number 13 as an `UnsignedBinaryInteger` object is initialized with the string '`1101`'.

- `__init__ (self, num_str)`: Initialize the `UnsignedBinaryInteger` class with a string representing the binary number.
- `__lt__(self, other)`: Returns True if self is less than other, or False otherwise
- `__gt__(self, other)`: returns True if self is greater than other, or False otherwise
- `__eq__(self, other)`: returns True if self is equal to other, or False otherwise
- `is_twos_power(self)`: returns True if self is a power of 2, or False otherwise
- `largest_twos_power(self)`: returns the largest power of 2 that is less than or equal to self
- `__repr__(self)`: Creates and returns the string representation of self. The string representation starts with 0b, followed by a sequence of 0s and 1s

—---------- *optional*

- `__add__(self, other)`: Returns an `UnsignedBinaryInteger` object that represent the sum of self and other (also of type `UnsignedBinaryInteger`) the result also shouldn't have excess leading 0's
- `__or__(self, other)`: Returns a `UnsignedBinaryInteger` object that represents the bitwise or result of self and other
  - *Example:*
  - `1010 or 1001 results in 1011`
    - `1 or 1 → 1`
    - `0 or 0 → 0`
    - `1 or 0 → 1`
    - `0 or 1 → 1`
- `__and__(self, other)`: Returns a `UnsignedBinaryInteger` object that represents the bitwise and result of self and other
  - *Example:*
  - `1010 and 1001 results in 1000`
    - `1 and 1 → 1`

- `0 and 0 → 0`
- `1 and 0 → 0`
- `0 and 1 → 0`

Notes and assumptions:

- Your implementation should account for the edge case where both numbers do not have the same number of digits.
- `bin_num_str passed` in the constructor does not have excess leading '0' in the front and will always begin with a '1' for positive numbers, and a single '0' for 0.
- In Python, the bitwise OR is represented by a single vertical bar, |, and the bitwise AND is represented by a single 'and' symbol, &.

Starter Template

```
class Python:
    def __init__(self, bin_num_str):
        """
        :type coefficients: list
        """
       self.data = bin_num_str
    def __lt__(self, other):
        """
        :type other: Polynomial
        :return type: Boolean
        """
    def __gt__(self, other):
        """
        :type other: Polynomial
        :return type: Boolean
        """
    def __eq__(self, other):
        """
        :type other: Polynomial
        :return type: Boolean
```

```
        """
    def is_twos_power(self):
        """
        :return type: Boolean
        """
    def largest_twos_power(self):
        """
        :return type: int
        """
    def __repr__(self):
        """
        :return type: string
        """

  def __add__(self, other):
        """
        :type other: Polynomial
        :return type: Polynomial
        """
  def __or__(self, other):
        """
        :type other: Polynomial
        :return type: Polynomial
        """
  def __and__(self, other):
        """
        :type other: Polynomial
        :return type: Polynomial
        """
```

3. Reverse List

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 $\Theta(n)$, where n is the length of the list (10 minutes).

```
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 $\Theta(n)$, 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]
```

4. Move Zeros

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.

```
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?

5. Maximum Sum Subarray:

You are given an integer array `nums` consisting of `n` elements, and an integer `k`.

Find a contiguous subarray whose length is equal to k that has the maximum sum value and *return this value*.

Your solution must run in $\Theta(n)$, where n is the length of the list. (30 minutes)

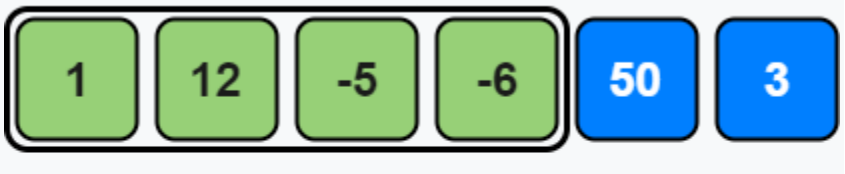
For example,

```
Input: nums = [1,12,-5,-6,50,3], k = 3
Output: 47
Explanation: Maximum sum is (-6 + 50 + 3) = 47. The window size is 3
```

Hint: Use two pointers with a fixed distance and increase both the start and end points to update your sums.

Contiguous Subarray = a smaller array nested within an array

In this array, `[1,12,-5,-6,50,3]`, the array `[1,12,-5,-6]` is a contiguous subarray because each number is adjacent to the next. `[1,12,50]` is not a contiguous subarray.



Optional Question

1. 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

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

$$f_{17}(n) = \sqrt{n^3}$$
$$f_{18}(n) = n!$$

2. Define a function `shift()`:

```
def shift(lst, k):
    """
    : lst type: list
    : k type: int
    : return type: None
    """
```

The function takes in a list and shifts it to the left by k steps.

ex) shift([1, 2, 3, 4, 5, 6], **2**) → [3, 4, 5, 6, **1, 2**]

Intuitively, you probably want to solve it using the list methods, pop() and insert() to shift the list or manually shift the list each time using an extra loop. Know that your solution will not be not linear if you use either of these methods. Since the run-time of the list methods have not been discussed at this point, do not use any of the methods for this question.

Instead, you will attempt to solve this with run-time in mind. That is, your solution must run in $\Theta(n)$, where n is the length of the list.

The direction will also change so that the **shift function will shift to the right instead.**

ex) shift([1, 2, 3, 4, 5, 6], **2**) → [**5, 6,** 1, 2, 3, 4]

Hint: You should use the function reverse_list(lst, low, high) function from part 3.b to solve this problem.