**Vitamins (70 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.

lst = [1, 2, 3]

lst2 = lst

lst.append(4)

lst2.append(5)

print(lst)

[1, 2, 3, 4, 5]

print(lst2)

[1, 2, 3, 4, 5]

b.

s = **"aBc"**

s = s.upper()

t = s

t = t.lower()

print(s)

ABC

print(t)

abc

c.

s = **"abc"**

**def** func(s):

s = s.upper()

print(**"Inside func s ="**, s)

func(s)

Inside func s = ABC

print(s) #abc

abc

d.

lst = [1, 2, 3]

**def** func(lst):

lst.append(4)

lst = [5, 6, 7, 8]

print(**"Inside func lst ="**, lst)

func(lst)

Inside func lst = [5, 6, 7, 8]

print(lst)

[1, 2, 3, 4]

1. For each of the following, print the result of the list object created using python’s list comprehension syntax (10 minutes):

[i//i for i in range(-3, 4) if i != 0]

[1, 1, 1, 1, 1, 1]

[**'Only Evens'**[i] for i in range(10) if i % 2 != 0]

['n', 'y', 'E', 'e', 's']

[((-i)\*\*3) for i in range(-2, 5)]

[8, 1, 0, -1, -8, -27, -64]

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. (30 minutes)

**import** copy  
lst = [1, 2, [3, 4]]  
lst\_copy = copy.copy(lst)

lst[0] = 10  
lst\_copy[2][0] = 30

print(lst)

[10, 2, [30, 4]]

print(lst\_copy)

[1, 2, [30, 4]]

**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)

[10, [2, 'ABC'], [3, [4]], 7]

print(lst\_deepcopy)

[1, [2, 'abc'], [3, [40]], 7]

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)

[1, [4, 3], ['aa', 'b'], 'c']

print(lst\_slice)

[1, [4, 3], ['aa', 'b']]

print(lst\_assign)

[1, [4, 3], ['aa', 'b'], 'c']

1. Given the generator function, write the output: (10 minutes)

**def** sum\_to(n): #also known as triangle numbers

for i in range(1, n+1):

total = i \* (i + 1)//2

yield total

for i in sum\_to(10):

print(i, end = ', ')

1, 3, 6, 10, 15, 21, 28, 36, 45, 55,

**OPTIONAL VITAMINS**

1. Use python’s list comprehension syntax to generate the following lists: (10 minutes)
2. [1, -2, 4, -8, 16, -32, 64, -128]

[(-2)\*\*i for i in range(8)]

1. [1, 11, 111, 1111, 11111, 111111, 1111111]

[int('1'\*i) for i in range(1, 8)]

1. Finish the python’s list comprehension syntax. The result is a list of characters of the input repeated twice. **Do not use any arithmetic operators or additional libraries.**

Your answer must use my\_str and length. (10 minutes)

my\_str = input("Enter a string:")

length = len(my\_str)

print([my\_str[i] for i in range(-length, length)])

my\_str = "Python"

my\_str = "Java"

→ ["J","a","v","a","J","a","v","a"]

→ ["P","y","t","h","o","n","P","y","t","h","o","n"]

**Coding (See CS1134 Lab2 Solutions.py)**

In this section, it is strongly recommended that you solve the problem on paper before writing code. For the **OPTIONAL** sections, we recommend you do these after lab for practice.

1. Define a generator that takes in a number n and returns the powers of 2 up to n:
   1. *def powers\_of\_two(n)*
   2. For example: powers\_of\_two(6) will return 1, 2, 4, 8, 16, 32
2. For this question, you will define a class to represent a polynomial. For this class, you will use a list as a data member to represent the coefficients. The index of each coefficient in the list will be its corresponding power of x. (45 minutes)

For example, the coefficient list of the polynomial

is [3, 7, 0, -9, 2].

index 0, coeff = 3 → . index 1, coeff = 7 → . index 2, coeff = 0 →

index 3, coeff = -9 → . index 4, coeff = 2 →

Notice that is included and that the coefficients in the list are in reversed order.

Your class should include the following:

1. A *constructor* that takes a list as a parameter, and initiates a polynomial with coefficients as given in the list. If no list is given at construction, your polynomial should be . **Name the list member variable self.data.**

Note: You may assume that the last element in the list (representing the coefficient of the highest power), is not 0.

* 1. *\_\_add\_\_* operator. The operator should take another polynomial object, and create a new polynomial object representing the sum of the two polynomials. Adding polynomials simply means adding their coefficients, but note that different polynomials might have different highest powers.

For example:

1. *\_\_call\_\_* operator, that takes a number and returns the value of the polynomial for that number when evaluated. For instance, calling p(1) where

p = Polynomial([3, 7, 0, -9, 2]) should return 3 because

.

If your Polynomial class works properly, you should see the following behavior:

#TEST CODE

#Constructor

poly1 = Polynomial([3, 7, 0, -9, 2]) #2x^4 - 9x^3 + 7x + 3

poly2 = Polynomial([2, 0, 0, 5, 0, 0, 3]) # 3x^6 + 5x^3 + 2

#add operator

poly3 = poly1 + poly2 # 3x^6 + 2x^4 - 4x^3 + 7x + 5 print(poly3.data) #[5, 7, 0, -4, 2, 0, 3]

#call operator

val1 = poly1(1)

print(val1) #3

val2 = poly2(1)

print(val2) #10

val3 = poly3(1)

print(val3) #13 (same result of 3 + 10; poly1(1) + poly2(1))

**OPTIONAL**

1. *\_\_repr*\_\_ operator, that returns a str representation of a polynomial in the format presented above. Instead of superscript, we will represent powers using the caret symbol ^ . You may format it as such:

2x^4 + -9x^3 + 0x^2 + 7x^1 + 3x^0

To achieve the formatting, you may want to use the join function.

<https://www.geeksforgeeks.org/join-function-python/>

1. *\_\_mul\_\_* operator. The operator should take another polynomial object, and create a new polynomial object representing the multiplication of the two polynomials. To multiply polynomials, multiply all pairs of coefficients from both lists, and group the ones of the same order.

For example:

You may want to start with a simpler example first to test your code:

1. A *derive* method that mutates the polynomial object to its derivative. You will have to implement the power rule. The modification must be in-place, that means you are not creating a new list with new values.

More on Derivatives: <https://www.khanacademy.org/math/ap-calculus-ab/ab-derivative-rules/ab-diff-negative-fraction-powers/a/power-rule-review>

For example, for the polynomial, the derive method will modify it to be:

**def** derive(self):

"""

:

"""

3. In this question, we will implement the UnsignedBinaryInteger class to represent non-negative integers by their binary (*base 2*) representation. Each object will have a data-member of type string, containing the binary representation of the number.

ex) the UnsignedBinaryInteger object representing the decimal-number 13, will have '1101', as its string data-member. **Assume that the 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.**

Your implementation should account for the edge case where both numbers do not have the same number of digits. (45 minutes)

**class** UnsignedBinaryInteger:

**def** \_\_init\_\_(self, bin\_num\_str):  
 self.data = bin\_num\_str

**def** \_\_add\_\_(self, other):  
 ''' Creates and 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'''

**def** decimal(self):

''' returns the decimal value of the binary integer'''

**def** \_\_lt\_\_(self, other):  
 ''' returns True if self is less than other, or False

otherwise'''

**def** \_\_gt\_\_(self, other):

''' returns True if self is greater than other, or False

otherwise'''

**def** \_\_eq\_\_(self, other):

''' returns True if self is equal to other, or False

otherwise'''

**def** is\_twos\_power(self):  
 ''' returns True if self is a power of 2, or False

otherwise'''

**def** largest\_twos\_power(self):  
 ''' returns the largest power of 2 that is less than or

equal to self'''

**def** \_\_repr\_\_(self):

''' Creates and returns the string representation  
 of self. The string representation starts with 0b,

followed by a sequence of 0s and 1s'''

**Note:** the 0b is only included in repr when printing the object. Do not add 0b to the string for self.data.

Example test code provided on the next page.

#TEST CODE

b1 = UnsignedBinaryInteger('10011')

b2 = UnsignedBinaryInteger('100')

print("b1 is: ", b1) #0b10011; b1.data is 10011

print("b2 is: ", b2) #0b100; b2.data is 100

b3 = b1 + b2

print("b3 is: ", b3) #0b10111

print("\nChecking decimal values:\n")

print(b1.decimal()) #19

print(b2.decimal()) #4

print(b3.decimal()) #23

print("\nChecking comparisons:\n")

print(b1 < b2) #False

print(b2 < b1) #True

print(b1 > b2) #True

print(b2 > b1) #False

print(b1 + b2 == b3) #True

print("\nChecking is\_twos\_power:\n")

print(b1.is\_twos\_power()) #False

print(b2.is\_twos\_power()) #True

print(b3.is\_twos\_power()) #False

print("\nChecking largest\_twos\_power:\n")

print(b1.largest\_twos\_power()) #16

print(b2.largest\_twos\_power()) #4

print(b3.largest\_twos\_power()) #16

**OPTIONAL**

Define two additional operators for bitwise operations in the UnsignedBinaryInteger class. In python, the bitwise or is represented by a single vertical bar, |, and the bitwise and is represented by a single and symbol, &.

Your implementation should account for the edge case where both numbers do not have the same number of digits.

**bitwise OR**

1010 or 1001 results in 1011

1 or 1 → 1

0 or 0 → 0

1 or 0 → 1

0 or 1 → 1

**def** \_\_or\_\_(self, other):

''' Creates and returns a BinaryPositiveInteger object

that represents the bitwise or result of self and other''

**bitwise AND**

1010 and 1001 results in 1000

1 and 1 → 1

0 and 0 → 0

1 and 0 → 0

0 and 1 → 0

**def** \_\_and\_\_(self, other):

''' Creates and returns a BinaryPositiveInteger object

that represents the bitwise and result of self and other''

#TEST CODE

b1 = UnsignedBinaryInteger('10011')

b2 = UnsignedBinaryInteger('100')

print("\nTesting b1: ", b1, "b2: ", b2)

b3 = b1 | b2

b4 = b1 & b2

print(b1, "|", b2, "=", b3) #0b100

print(b1, "&", b2, "=", b4)

b1 = UnsignedBinaryInteger('1010')

b2 = UnsignedBinaryInteger('1001')

print("\nTesting b1: ", b1, "b2: ", b2)

b3 = b1 | b2

b4 = b1 & b2

print(b1, "|", b2, "=", b3)

print(b1, "&", b2, "=", b4)

Output:

Testing b1: 0b10011 b2: 0b100

0b10011 | 0b100 = 0b10111

0b10011 & 0b100 = 0b0

Testing b1: 0b1010 b2: 0b1001

0b1010 | 0b1001 = 0b1011

0b1010 & 0b1001 = 0b1000