Student:

Problem 1 (1 point) - Implement a **recursive** function that takes a string, *s*, and returns its last uppercase letter. If *s* does not contain any uppercase letter, it should return None.
Note: You can use the method *isupper*(), which is a Python built-in method used for string handling. The *isupper*() methods returns True if all characters in the string are uppercase. Otherwise, it returns False.

Solution:

```
def lastUpper(s):
    if s is None or len(s)==0:
        return None

    last=s[len(s)-1]
    if last.isupper():
        return last

    return lastUpper(s[0:len(s)-1])
```

Problem 2 (1.5 point) - Implement a **recursive** function taking two parameters: a string, s, and a character, c. The method returns the number of occurrences of *c* in *s*. The solution must be based on the **divide-and-conquer** strategy (other approaches will not be evaluated).

Solution:

```
def count(s,c):
    if s is None or len(s)==0:
        return 0

    m=len(s)//2
    result=0
    if s[m]==c:
        result +=1

    count1=count(s[0:m],c)
    count2=count(s[m+1:],c)

    return result + count1+count2
```

Problem 3 (2.5 points) - Given the classes:

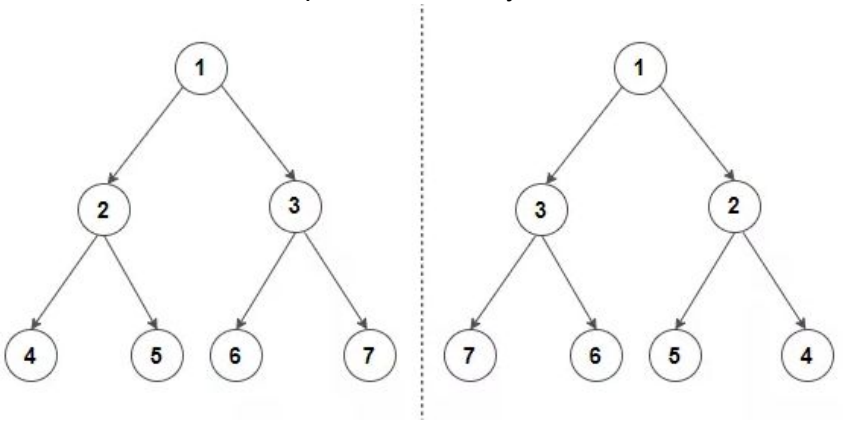
```
class Node:

  def __init__(self,elem=None):
    self.elem=elem
    self.leftChild=None
    self.rightChild=None
    self.parent=None

class BinaryTree:

  def __init__(self):
    self.root=None
     ….
```

In the BinaryTree class, add a **recursive** function, named **mirror,** to convert the binary tree to its mirror. For example, these binary trees are mirror of each other:



Solution:

```
  def mirror(self):
    self._mirror(self.root)

  def _mirror(self,node):
    if node is None:
      return

    self._mirror(node.leftChild)
    self._mirror(node.rightChild)

    temp=node.leftChild
    node.leftChild=node.rightChild
    node.rightChild=temp
```

Problem 4 (1 point): The binary search algorithm is a search algorithm that finds the position of a target value within a sorted list. What is the time complexity of the binary search algorithm?. Please, explain your answer.

Solution:

Binary search runs in logarithmic time in the worst case, making O(log n) comparisons, where n is the number of elements in the list.

After every comparison with the middle term, we only have to search into one of the half of the input list (if the target value is not the middle element)

So, for example, for finding one element in a list of 16 elements, we will have to divide the list 4 times in the worst case.

For n elements, we will have to divide the list k times:

n*(1/2^k) = n/2^k = 1 => n=2^k => k=log(n)

Problem 5 (2 points) –Given a singly linked list, implement a function, *deleteLast*, taking a number, c, and removing the last occurence of c in the list. For example, if given linked list is 5->3->2->5->3->1 and c=3, then linked list should be modified to 5->3->2->5->1.

What is the time complexity of this method?. Explain your answer.

You must use the SList (Singly Linked List) class studied during the course. You must implement those methods of the class that you use in your solution. It is not allowed to use the Python List class.

Solution:

```
def deleteLast(self,x):
    if self.isEmpty():
      print('list is empty')
      return

    node=self.head
    lastIndex=-1
    i=0
    while node:
      if node.element==x:
        lastIndex=i
      i=i+1
      node=node.next

    if lastIndex!=-1:
      self.removeAt(lastIndex)
```

The time complexity is O(n). In the worst case, the last occurence of c is in the last element of the list.

Problem 6 (2 points). In a graph**,** the bread-first-search algorithm starts at a vertex *v* and visits, first the neighbours of v, then the neighbours of the neighbours of v, then the neighbours of the neighbours of the neighbours of v, and so on. Given the class:

```
class Graph:
    def __init__(self):
        self.vertices = {}

    def addVertex(self,u):
        if u not in self.vertices:
            self.vertices[u]=[]

    def addEdge(self,u,v):
        if u not in self.vertices:
            self.addVertex(u)
        if v not in self.vertices[u]:
            self.vertices[u].append(v)
```

Implement a function, **breadth**, which takes a vertex, *v*, and prints the breadth traversal starting at this vertex.
Note: In this problem, it is allowed to use Python data structures such as Python lists, sets or queues.

Solution:

```
    def breadth(self, s):

        visited = [False] * (len(self.vertices))
        queue = []

        queue.append(s)
        visited[s] = True

        while queue:
            s = queue.pop(0)
            print (s, end = " ")

            for i in self.vertices[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```