

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*.

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).

Problem 3 (2.5 points) - Given the classes:

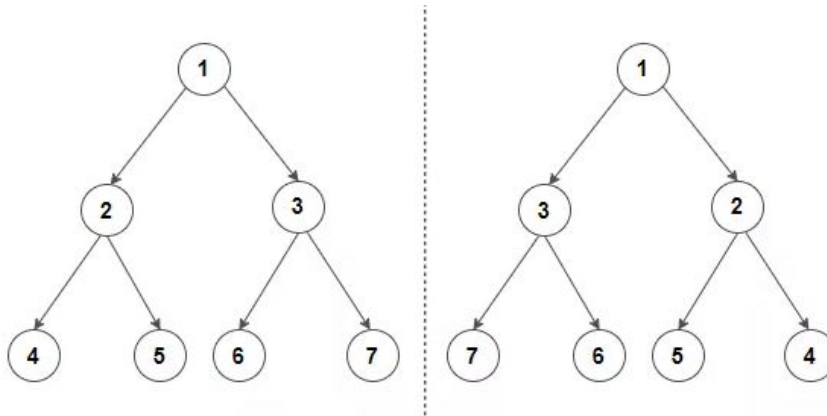
```
class Node:

    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:



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.

Problem 5 (2 points) –Given a singly linked list, implement a function, *deleteLast*, taking a number, *c*, and removing the last occurrence 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.

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.