Question 1:

Linear search and Binary search aim to find the location of a specific value within a list of values (sorted list for binary search). The next level of complexity is to find two values from a list that together satisfy some requirement.

Propose an algorithm to search for a pair of values in an unsorted array of *n* integers that are closest to one another. Closeness is defined as the absolute value of the difference between the two integers. Your algorithm should not first sort the list. [10 points]

Next, propose a separate algorithm for a sorted list of integers to achieve the same goal. [10 points]

Briefly discuss which algorithm is more efficient in terms of the number of comparisons performed. A formal analysis is not necessary. You can simply state your choice and then justify it. [5 points]

```
def find_closest_values_unsorted(values: list[int]):
    n = len(values)
    if n < 2:
        return
    closest_values = (values[0], values[1])
    lowest_diff = abs(values[1] - values[0])
    for i in range(n):
        for j in range(i+1, n):
            diff = abs(values[i] - values[j])
            if diff < lowest_diff:
                lowest_diff = diff
                closest_values = (values[i], values[j])
    return closest_values

def find_closest_values_sorted(values: list[int]):
    n = len(values)
    if n < 2:
        return
    closest_values = (values[0], values[1])
    lowest_diff = abs(values[1] - values[0])
    for i in range(1, n):
        diff = abs(values[i] - values[i-1])
        if diff < lowest_diff:
            lowest_diff = diff
            closest_values = (values[i], values[i-1])
    return closest_values
```

The second algorithm (for a sorted array) is more efficient in terms of the number of comparisons performed. The unsorted algorithm has to check all combinations of integers in the array, leading to $\sum_{i=1}^{n-1}(i)$ comparisons with an $O(n^2)$ time complexity. Since the sorted algorithm is given a sorted list of integers, we know that the pair of values closest to one another (by the absolute value of the difference) will be adjacent to each other. With this, the sorted algorithm only has to check pair values adjacent to each other, leading to $(n-1)$ comparisons and a time complexity of $O(n)$. Since $(n-1)$ comparisons is always less than or equal to $\sum_{i=1}^{n-1}(i)$ comparisons, the sorted algorithm is more efficient in terms of the number of comparisons performed.

Question 2:

Implement in Python an algorithm for a level order traversal of a binary tree. The algorithm should print each level of the binary tree to the screen starting with the lowest/deepest level on the first line. The last line of output should be the root of the tree. Assume your algorithm is passed the root node of an existing binary tree whose structure is based on the following BinTreeNode class. You may use other data structures in the Python foundation library in your implementation, but you may not use an existing implementation of a Binary Tree or an existing level order traversal algorithm from any source.

Construct a non-complete binary tree[1] of at least 5 levels. Call your level order traversal algorithm and show that the output is correct. [20 points]

```
class BinTreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left  = left
        self.right = right


from collections import deque

def print_binary_tree_levels(root):
    if not root:
        return

    queue = deque([root])

    levels = []

    while queue:
        level = []
        cur_level_len = len(queue)

        for _ in range(cur_level_len):
            cur_node = queue.popleft()
            level.append(str(cur_node.value))

            if cur_node.left:
                queue.append(cur_node.left)
            if cur_node.right:
                queue.append(cur_node.right)

        levels.append(level)

    for level in reversed(levels):
        print(" ".join(level))
```

[1] A complete binary tree is a binary tree where every level except possibly the last level is full, meaning no additional nodes could fit on that level.

Non-complete binary tree of 5 levels:

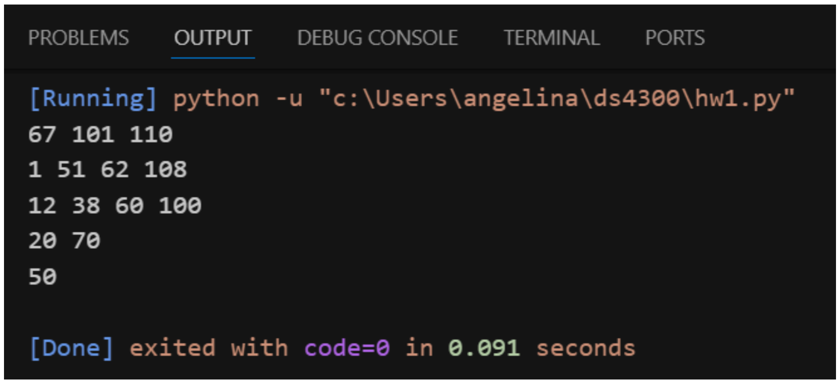
```
            50
          /    \
        20       70
       /  \     /  \
      12   38  60   100
     /        / \     \
    1       51  62    108
                  \   /  \
                  67 101  110
```

```
root = BinTreeNode(50)
root.left = BinTreeNode(20)
root.right = BinTreeNode(70)
root.left.left = BinTreeNode(12)
root.left.right = BinTreeNode(38)
root.right.left = BinTreeNode(60)
root.right.right = BinTreeNode(100)
root.left.left.left = BinTreeNode(1)
root.right.left.left = BinTreeNode(51)
root.right.left.right = BinTreeNode(62)
root.right.right.right = BinTreeNode(108)
root.right.left.right.right = BinTreeNode(67)
root.right.right.right.left = BinTreeNode(101)
root.right.right.right.right = BinTreeNode(110)
```

When `print_binary_tree_levels(root)` is called, the expected output should be:

```
67 101 110
1 51 62 108
12 38 60 100
20 70
50
```

Actual output when running `print_binary_tree_levels(root)`:



```
[Running] python -u "c:\Users\angelina\ds4300\hw1.py"
67 101 110
1 51 62 108
12 38 60 100
20 70
50

[Done] exited with code=0 in 0.091 seconds
```

Expected and actual output match!