

## Homework 01

**Extra Credit Deadline:** Sunday, January 12 2024 @ 11:59 pm  
**Regular Credit Deadline:** Tuesday, January 14 2024 @ 11:59 pm

### Directions:

1. From the File Menu, make a copy of this document in your own Google Drive account OR download an MS Word version.
2. Provide **professional-quality, type-set** solutions for each of the stated questions.
  - Do not delete the actual question statement. However, you may add additional space between the questions to accommodate your answers.
  - **Your solutions must be typeset**. The solutions you incorporate into this document MAY NOT be handwritten. This includes not writing them on an iPad and then copying them (or a screen shot) into this document. I suggest you use some type of equation editor in your writing tool of choice or use LaTeX to typeset your responses where appropriate.
  - Any source code or pseudocode should be typed in a *monospaced font* like Courier New or Fira Mono, both available in Google Docs. **DO NOT paste screenshots of your code.**
  - To reiterate, handwritten solutions in any form will receive NO CREDIT.
3. **Generate a PDF of your document with your solutions.** DO NOT upload screenshots, images, or pictures. Reminder, DO NOT delete the question statements. If you compose your solutions in LaTeX, please retype the questions.
4. Upload your solutions document to GradeScope by the due date. **It is your responsibility to associate each question with your solution in GradeScope and click the submit button afterwards.** Failure to do so will result in no credit for any questions not linked with your solution and will not be a valid reason to request a re-grade.

### Academic Collaboration Reminder:

Remember that you may not look at, copy, capture, screenshot, or otherwise take possession of any other students' solutions to any of these questions. Further, you may not provide your solutions in part or in whole to any other student. Doing any of the above constitutes a violation of academic honesty which could result in an F in this class and a referral to OSCCR.

What is permissible? You are free and encouraged to talk to your peers about the conceptual material from the lectures or the conceptual material that is part of this assignment. I'm confident you know where the line between collaborative learning and cheating sits. Please don't cross that line.

### 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 (**in Python**) 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 (in Python) 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]

**\*The code below was copied and pasted from Pycharm (not a screenshot)**

```
import numpy as np

def closest_unsorted(arr):
    '''
    Algorithm that identifies the two closest values (smallest abs value of difference)
    in an unsorted array of n integers

    Argument:
        arr (numpy array) - array of unsorted integers

    Return:
        closest_pair (list) - list of the two closest integers

    '''

    # save the magnitude of the difference between the first two numbers
    small_diff = abs(arr[1] - arr[0])

    # initialize variables for the two closest integers
    first_int = None
    second_int = None

    # iterate through each number in the array except the last position
    for i in range(len(arr) - 1):

        # get the difference between the ith number and every other number
        # in the array that falls after i
        for j in range(i+1, len(arr)):

            # save the differences for each pair of numbers
            diff = abs(arr[j] - arr[i])

            # check if the difference is smaller than the current smallest difference
            if diff < small_diff:

                # set small_diff equal to the new smallest difference
                small_diff = diff

                # add the values associated with that difference to closest pair list
                first_int = arr[i]
                second_int = arr[j]

    return first_int, second_int

def closest_sorted(arr):
    '''
    Algorithm that identifies the two closest values (smallest abs value of difference)
```

*in a sorted array of  $n$  integers*

**Argument:**

*arr (numpy array) - array of sorted integers*

**Return:**

*closest\_pair (list) - list of the two closest integers*

**'''**

*# save the magnitude of the difference between the first two numbers*

*small\_diff = abs(arr[1] - arr[0])*

*# initialize variables for the two closest integers*

*first\_int = None*

*second\_int = None*

*# iterate through each element of the array and subtract it from the following element*

*for i in range(len(arr) - 1):*

*# save the difference of each pair of adjacent numbers*

*diff = abs(arr[i+1] - arr[i])*

*# check if the difference is smaller than the current smallest difference*

*if diff < small\_diff:*

*# set small\_diff equal to the new smallest difference*

*small\_diff = diff*

*# add the values associated with that difference to closest pair list*

*first\_int = arr[i]*

*second\_int = arr[i+1]*

*return first\_int, second\_int*

The second algorithm, which finds the closest two integers in a sorted list, is more efficient and requires fewer comparisons than the first algorithm for unsorted lists. In the algorithm for the sorted list, each number in the list is subtracted from its following number, with the exception of the very last number; this is a total of  $n-1$  difference comparisons. In the unsorted list, however, we cannot assume that each number is closer to its adjacent neighbors than to any other number in the list. Therefore, each number must be subtracted from every subsequent number in the list – not just the following number. This requires far more comparisons than the algorithm for the sorted list.

**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 (in Python) 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]**

```
# imports
import collections
from collections import deque
```

```

import random
def search_unsorted(ls):
    """
    This function searches for the pair of values that are closest to each in an unsorted
    array.
    In this case, closeness is defined as the absolute value of the difference between the
    two integers.
    Args:
    - list of values
    Returns:
    - int value 1
    - int value 2
    """
    # initiate values
    closest = None
    val_pairs = (None, None)
    # iterate through nested for loop of the same list
    for i in range(0, len(ls)):
        val1 = ls[i]
        for j in range(0, len(ls)):
            # skip if the index is the same (the result will always 0)
            if i == j:
                continue;
            else:
                val2 = ls[j]
                # check if closest has not been instantiated yet
                if closest == None:
                    closest = abs(val1 - val2)
                    val_pairs = (val1, val2)
                # if closest has been set, check if the difference between the test values
                # is smaller than the current closest
                if abs(val1 - val2) < closest:
                    val_pairs = (val1, val2)

    return val_pairs

```

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

```

def search_sorted(ls):
    """
    This function finds the two closest values in a sorted list. In this case, closest is
    defined as the absolute value of the
    difference between the two integers
    Args:
    - list of values
    Returns:
    - int value 1
    - int value 2
    """

```

```

"""
# initiate values
closest = None
val_pairs = (None, None)
if len(ls) <= 1:
    return "not enough values to find the closest values"
# iterate through the list, since the list is sorted, the closest value will always be
the next list value which means you only need to
# run through the list once
for i in range(0, len(ls)):
    if closest == None:
        closest = abs(ls[i + 1] - ls[i])
    if i + 1 < len(ls):
        if abs(ls[i + 1] - ls[i]) < closest:
            val_pairs = (ls[i], ls[i+1])

return val_pairs

```

**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]**

Algorithm 1 at worst will be  $O(n^2)$ . This is because the function is composed of two nested for-loops and at worst, the function will iterate through the upper list  $n$  times and the inner list  $n$  times. In contrast, Algorithm 2 is more efficient. Since the list is already sorted, for any given number at index  $i$ , the closest possible difference between  $i$  and the rest of the numbers in the list will always be either between  $i-1$  and  $i$  or  $i$  and  $i + 1$ . Therefore, to find the closest difference of all the numbers, you only need to iterate through each element in the list 1 time , making the run time  $O(n)$ .

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]

```

def unsort_closest_pair(arr):
    diff = float('inf')
    closest_pair = [0,0]
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            abs_diff = abs(arr[i] - arr[j])
            If abs_diff < diff:
                diff = abs_diff
                closest_pair = [arr[i], arr[j]]
    return closest_pair

```

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

```

def sort_closest_pair(arr):

```

```

diff = float('inf')
closest_pair = [0,0]
for i in range(len(arr) - 1):
    abs_diff = abs(arr[i] - arr[i + 1])
    if abs_diff < diff:
        diff = abs_diff
    closest_pair = [arr[i], arr[i+1]]
return closest_pair

```

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]

**The `sort_closest_pair` algorithm is more efficient because it performs comparisons only between adjacent values, whereas the `unsort_closest_pair` algorithm compares every value with every subsequent value.**

## 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.

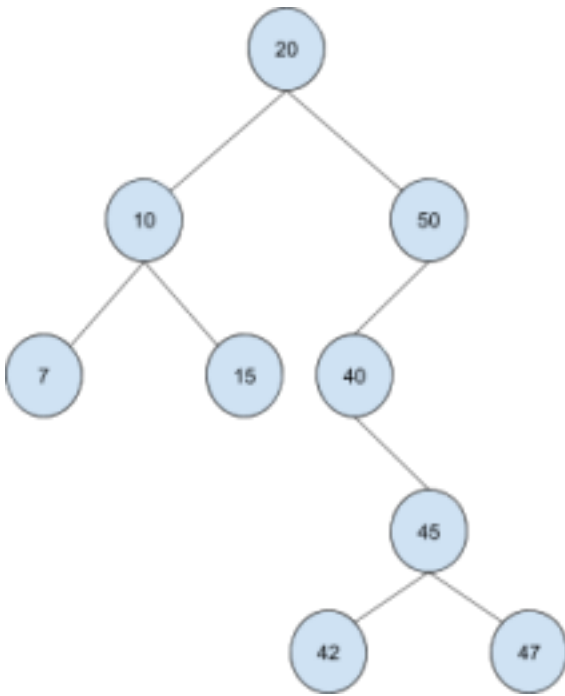
Construct a non-complete binary tree<sup>1</sup> of at least 5 levels. Call your level order traversal algorithm and show that the output is correct. [20 points]

```

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

```

Diagram of binary tree created to test the algorithm:



**\*The code below was copied and pasted from Pycharm (not a screenshot)**

```
from collections import deque
```

```
class BinTreeNode:
```

```
    '''
```

```
    Class for nodes within a binary search tree
```

```
    Attributes:
```

```
        value (int) = numerical value stored within the node
```

```
        left (BinTreeNode object) - left child node
```

```
        right (BinTreeNode object) - right child node
```

```
    '''
```

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

```
        self.value = value
```

```
        self.left = left
```

```
        self.right = right
```

```
def level_traversal(root_node):
```

```
    '''
```

```
    Algorithm for level order traversal of a binary search tree, which prints each level of
    the tree from deepest to the root
```

```
    Argument:
```

```
        root_node (BinTreeNode object) - highest level node in the binary search tree
```

```
    Returns:
```

```
        None
```

```
    '''
```

<sup>1</sup>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.

```

# initialize empty list to contain nested lists of values on each level of the tree
level_groupings = []

# initialize empty deque and begin by adding the root node to the queue
level_queue = deque()
level_queue.append(root_node)

# while loop that breaks when the level queue is empty
while level_queue:

    # initialize temporary list to store the values for each tree level
    temp_lst = []

    # loop through each node in the queue for a given level
    for i in range(len(level_queue)):

        # pop the current first node in the queue
        popped_node = level_queue.popleft()

        # add its value to the temporary list
        temp_lst.append(popped_node.value)

        # check if the popped node has left and right nodes, and if so,
        # append them to the end of the level queue
        if popped_node.left:
            level_queue.append(popped_node.left)

        if popped_node.right:
            level_queue.append(popped_node.right)

    # after adding each value in the level to the temp_lst, append it to the
    # level grouping list
    level_groupings.append(temp_lst)

# loop through each nested list in level groupings and remove/print them in reverse order
for i in range(len(level_groupings)):
    print(level_groupings.pop())

```

```
def main():
```

```

# define root node and all child nodes for the binary search tree
root = BinTreeNode(20)
root.left = BinTreeNode(10)
root.right = BinTreeNode(50)
root.left.left = BinTreeNode(7)
root.left.right = BinTreeNode(15)
root.right.left = BinTreeNode(40)
root.right.left.right = BinTreeNode(45)
root.right.left.right.left = BinTreeNode(42)
root.right.left.right.right = BinTreeNode(47)

# call the level order traversal function
level_traversal(root)

```



```
if name == 'main':  
    main()
```

### Output:

```
/Users/sarah/anaconda3/bin/python /Users/sarah/Documents/ds4300/Hw1/bintreetest.py  
[42, 47]  
[45]  
[7, 15, 40]  
[10, 50]  
[20]
```

Process finished with exit code 0

```
class BinTreeNode:  
    def __init__(self, value=0, left=None, right=None):  
        self.value = value  
        self.left = left  
        self.right = right  
  
def level_order_traversal(root):  
    """  
    This function traverses through a binary tree using level order traversal.  
    Args:  
    - root node of a BinTree object  
    Returns:  
    - list of values of each node in the order it was traversed (in reverse order)  
    """  
    # initialize queue with the root node & the results list for the values  
    depth = 1  
    queue = deque([(root, depth)])  
    result = deque([])  
    # iterate through the queue until it is empty  
    while queue:  
        # remove the leftmost node for processing (node in the front)  
  
        node_tuple = queue.popleft()  
        node = node_tuple[0]  
        depth = node_tuple[1]  
        #print(node.value, depth)  
        result.appendleft((node.value, depth))  
        if node.left:  
            queue.append((node.left, depth + 1))  
        if node.right:  
            queue.append((node.right, depth + 1))  
  
    # print the tree in the correct order  
    temp = result  
    curr_depth = None  
    final = []
```

```

while temp:
    node, depth = temp.popleft()
    if len(final) == 0:
        final.append(node)
        curr_depth = depth
        continue;
    elif depth == curr_depth:
        final.append(node)
    else:
        temp.appendleft((node, depth))
        print(f"Current Depth is {curr_depth}: {final}")
        final = []

print(f"Current Depth is {curr_depth}: {final}")

```

```

# Level 1
root = BinTreeNode(50)

# Level 2
root.left = BinTreeNode(30)
root.right = BinTreeNode(70)

# Level 3
root.left.left = BinTreeNode(20)
root.left.right = BinTreeNode(40)
root.right.left = BinTreeNode(60)
root.right.right = BinTreeNode(100)

# Level 4
root.left.left.left = BinTreeNode(10)
root.left.left.right = BinTreeNode(25)
root.left.right.left = BinTreeNode(35)
root.left.right.right = BinTreeNode(46)
root.right.left.left = BinTreeNode(59)
root.right.left.right = BinTreeNode(63)
root.right.right.left = BinTreeNode(90)
root.right.right.right = BinTreeNode(120)

# Level 5
root.left.left.left.left = BinTreeNode(3)
root.left.left.left.right = BinTreeNode(21)
root.left.left.right.left = BinTreeNode(29)
root.left.left.right.right = BinTreeNode(32)
root.left.right.left.right = BinTreeNode(36)
root.left.right.right.left = BinTreeNode(45)
root.left.right.right.right = BinTreeNode(48)

```

```
level_order_traversal(root)
```

```

Current Depth is 5: [48, 45, 36, 32, 29, 21, 3]
Current Depth is 4: [120, 90, 63, 59, 46, 35, 25, 10]
Current Depth is 3: [100, 60, 40, 20]
Current Depth is 2: [70, 30]
Current Depth is 1: [50]

```

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<sup>1</sup> 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 rlo_transversal(root):
    queue = deque([root])
    stack = []
    while queue:
        level_len = len(queue)
        current_level = []

        for i in range(level_len):
            node = queue.popleft()
            current_level.append(node.value)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        stack.append(current_level)

    while stack:
        print(" ".join(map(str, stack.pop())))
```

```
rlo_transversal(root)
```

Binary Tree Test Code:

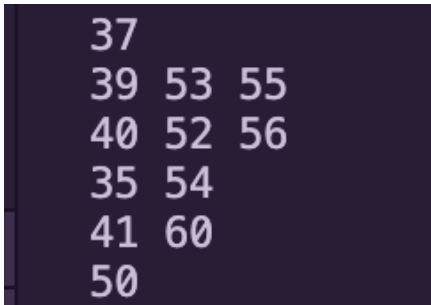
```
# root
root = BinTreeNode(50)
# level 2
root.left = BinTreeNode(41)
root.right = BinTreeNode(60)
# level 3
root.left.left = BinTreeNode(35)
root.right.left = BinTreeNode(54)
# level 4
```

---

<sup>1</sup> 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.

```
root.left.left.right = BinTreeNode(40)
root.right.left.left = BinTreeNode(52)
root.right.left.right = BinTreeNode(56)
# level 5
root.left.left.right.left = BinTreeNode(39)
root.right.left.left.right = BinTreeNode(53)
root.right.left.right.left = BinTreeNode(55)
# level 6
root.left.left.right.left.left = BinTreeNode(37)
```

Output:



```
37
39 53 55
40 52 56
35 54
41 60
50
```