DS4300 HW 1

January 12, 2025

1 DS4300 HW 1

Lesrene Browne, Jan 14 2025

1.0.1 Problem 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]

[7]:
```python
# Problem 1, Part 1 Algorithm

def closest_pair_v1(lst):

    if len(lst) < 2:
        return None # this lst doesn't have enough elements to have pairs

    smallest_closeness = float('inf')
    pair = None

    for curr_pos in range(len(lst)):
        for pos in range(curr_pos+1, len(lst)):
            closeness = abs(lst[curr_pos] - lst[pos])
            if closeness < smallest_closeness:
                smallest_closeness = closeness
                pair = (lst[curr_pos], lst[pos])
    return pair
```

[9]:
```python
# Problem 1, Part 1 Test

practice_array = [4, 1, 20, 8, -2, 7, 3]

# should return (4,3)
practice_result1 = closest_pair_v1(practice_array)
practice_result1
```

[9]: (4, 3)

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

[10]:
```
# Problem 1, Part 2 Algorithm

# lst will be sorted before being plugged in
def closest_pair_v2(lst):

    if len(lst) < 2:
        return None # this lst doesn't have enough elements to have pairs

    smallest_closeness = float('inf')
    pair = None

    for pos in range(len(lst)-1): # doesn't need to make it to the last element␣
 ↪because the pairs will be adjacent since the elements are sorted so the last␣
 ↪element will already have been considered when the 2nd-to-last element pair␣
 ↪closeness is evaluated
        closeness = abs(lst[pos] - lst[pos+1])
        if closeness < smallest_closeness:
            smallest_closeness = closeness
            pair = (lst[pos], lst[pos+1])
    return pair
```

[11]:
```
# Problem 1, Part 2 Test

sorted_practice_array = sorted(practice_array)

# should return (3,4)
practice_result2 = closest_pair_v2(sorted_practice_array)
practice_result2
```

[11]: (3, 4)

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]

 # Problem 1, Part 3

The second algorithm is more efficient because it only has to traverse the list once (since the list is already sorted). Since it only traverses the list once it makes less comparisons. In the worst case the time complexity for the first algorithm would be O(n^2), because we have to visit every element of the list at least twice due to the two loops and the time complexity for the second algorithm is O(n), as we scan through the list once which is clearly better.

1.0.2 Problem 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 of at least 5 levels. Call your level order traversal algorithm and show that the output is correct. [20 points]

[14]:
```
class BinTreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

[22]:
```
# Problem 2 Algorithm

from collections import deque

def reverse_level_order_traversal(root):

    result = []
    de = deque([root])

    while de:
        level_size = len(de)  # number of nodes that are supposed to be in the current level
        current_level = []

        for unprocessed_node in range(level_size):
            node = de.popleft()  # taking nodes from the right would show the output from right to left or include child nodes which isn't correct
            current_level.append(node.value)

            # adds the children of the current node to the end of the queue
            # so that every node in the current level's children will be in the queue to be processed from left to right
            if node.left:
                de.append(node.left)
            if node.right:
                de.append(node.right)

        # once the amount of unprocessed nodes from the current level is complete, the current level's list of vals is added as a list to the result
```

```
        result.append(current_level)

    for lst in reversed(result):
        print(lst)
```

[23]:
```
# Problem 2 Test

# generating a non-complete BST that has 5 levels
root = BinTreeNode(10)
root.left = BinTreeNode(5)
root.right = BinTreeNode(15)

root.left.left = BinTreeNode(3)
root.left.right = BinTreeNode(7)

root.left.left.left = BinTreeNode(2)

root.right.right = BinTreeNode(20)
root.right.right.left = BinTreeNode(17)
root.right.right.right = BinTreeNode(25)

root.right.right.left.right = BinTreeNode(18)

practice_result3 = reverse_level_order_traversal(root)
practice_result3
```

```
[18]
[2, 17, 25]
[3, 7, 20]
[5, 15]
[10]
```