[DOCUMENT TITLE]

CS234 Winter 2018

JIADONG MAI

2557203

Assignment 2

Q2

A:

Function quicksort(list):

If len(list) <= 1: return list # O(1)

Pivot <- average(list) # O(1)

Lesser <- all elements in list that are < pivot # O(n)

Greater <- all elements in list that are > pivot # O(n)

Equal <- all elements in list that are = pivot # O(n)

Return quicksort(Lesser) + Equal + quicksort(Greater) # 2 T(n/2) + O(1)

In this algorithm, pivot is the average of the list, it might would exist or not exist in the list.

If the pivot is not exist in the list, I create three empty list for Lesser, Greater and Equal = [] and use the character k to represent the number in the for loop, like for k in rang(num).

Using a example list to explain the question: list_a: [9,1,6,3,6] Average(list_a): 5

At the beginning: k = 6, it is larger than 5, therefore, Greater.append(6), Greater: [9].

And then the k = 1, it is less than 5, Lesser.append(1), Lesser = [1]

And then go ahead, k = 9, it is bigger than 5, Greater.append(6), Greater: [9,6]

And then the k = 3, it is less than 5, Lesser.append(3), Lesser:[1,3]

And then the k = 6, it is bigger than 5, Greater.append(6), Greater.append(6), Greater: [9,6,6]

Return quicksort([1,3]) + [] + quicksort([9,6,6])

The next processes are like the processes as above

If the Pivot is exist in the list, it is similar to the original quicksort(), store the elements in list of Equal: [], and continue for the quicksort for lesser or greater.

Therefore, this algorithm always terminates.

B:

Basis on the Helpful recurrence relations

T(n) = 3*O(n) + 3*O(1) + 2*T(n/2) -> O(n) + 2*T(n/2) -> O(n log n)

Therefore, the worst case is O(n log n), which is similar to the merge sort. In the worst case, the number of comparison of this algorithm makes is equal to or slightly smaller than (nlog(n) – 2^(logn) +1), which is between (nlogn – n +1) and (n log(n) + n + O(log n)). Therefore, the worst case is O(n log n)

An example worst case could be given, the elements in the list based on the rule form largest to smallest, like my_list = [13,12,11,10,9,8,7,6,5,4,3,2,1,7] average(my_list): 7

In the first run of quicksort(my_list), Greater: [13,12,11,10,9,8], Equal: [7,7], Lesser:[6,5,4,3,2,1],
it will return quicksort([13,12,11,10,9,8]) + [7,7] + quicksort([6,5,4,3,2,1])

And then the quicksort(Greater) average(Greater): 10.5

In the list of Greater of quicksort(Greater), it will return quicksort([13,12,11]) + quicksort([10,9,8])

And then quicksort([13,12,11]) -> will return quicksort([11]) + [12] + quicksort([13])

….

Finally, For the quicksort(Greater), quicksort([13,12,11,10,9,8]) will produce [8,9,10,11,12,13]

Similar process to the quicksort(Lesser)

….

Quicksort(my_list) will produce [1,2,3,4,5,6,7,7,8,9,10,11,12,13]

Q3

A:

I suppose len(head1) >= len(head2)   len(head1) = n,   len(head2) = m

head.element will return the value of the Node, example, a = Node(2, Node(3, None)), a.element == 2

head.next will return the next Node of the head, example, a.next == Node(3, None)

```
function find_common_node1:
        while head1 is not None:                                                        # O(n) times
                for head2 is not None:                                                  # O(m) times
                        if head1 == head2:  return head1.element or head2.element       # O(1)
                        head2 = head2.next                                              # O(1)
                head1 = head1.next                                                      # O(1)
```

T(n) = O(n) * (O(m) * O(1)) = O(nm)

Therefore, this algorithm that has O(nm) time complexity

B:

I suppose len(head1) >= len(head2)   len(head1) = n,   len(head2) = m

head.remove_front(num) will mutate the head , by getting rid of the num of item from head, example, a = Node(2, Node(3, Node(4, None))) after running a.remove_front(2), a will become a: Node(4, None)

head.element will return the value of the Node, example, a = Node(2, Node(3, None)), a.element == 2

```
function find_common_node2:
        n = len(head1)                                    # O(n)
        m = len(head2)                                    # O(m)
        new_head1 = head1.remove_front(n-m)               # O(1)
        I = 0:                                            # O(1)
        while I < m:                                      # O(m)
                if new_head1 == head2:                    # O(1)
                        return head2.element              # O(1)
                else:
                        new_head1.remove_front(1)         # O(1)
                        head2.remove_front(1)             # O(1)
                        I += 1                            # O(1)
        return None
```

T(n) = O(n) + O(m) = O(n + m)

Therefore, this algorithm that has O(n+m) time complexity