Algorithms and Complexity Assignment 2

Student: Lixian Sun
Student ID: 938295

Q1. (a)

Function: A pre-sorting based algorithms to find the duplicates of array in O(nlogn).
Input: Array A[0..n-1].
Output: List that contains the duplicates of A[0..1].
Complexity: I choose merge sort to sort the array because the merge sort is stable and. its time complexity is O(nlogn). And the time complexity of the "for loop" in the algorithm is O(n). So the time complexity of my algorithm is O(nlogn) + O(n) = O(nlogn).

```
function FINDDUPLICATES(A[0..n − 1])
    MERGESORT(A[0..n − 1]) //use merge sort method to sort the array A
    Initialize empty linked list duplicate //initialize a list to store duplicates
    for i ← 0 to n − 2 do
        if A[i] = A[i + 1] and duplicate.val != A[i]
            duplicate.insert(A[i]) //if haven't been recorded, insert the duplicate
    return duplicate
```

(b)

Function: A hashing based algorithms to find the duplicates of array in O(n). I don't have to specify the hash function and I assume that the elements. could be inserted into the hash table successfully and distributed.
Input: Array A[0..n-1].
Output: List contains the duplicates of A[0..n-1].
Complexity: In order to visit each element of array A[0..n-1], I use a "for loop" in my. algorithm. So the time complexity is O(n).
Lookup: If the element exists in hash table, return true. If the element doesn't exist in hash table, insert the element into hash table and return false.

```
function FINDDUPLICATES(A[0..n − 1])
    Intialize hash table individualHash //to store non duplicate elements
    Intialize hash table duplicateHash //to avoid inserting duplicate for many times
    Initialize empty linked list duplicate //initialize a list to store duplicates
    for i ← 0 to n − 1 do
        //lookup: If the element exists in hash table return true. If the element
        //dosen't exit, insert the element into hash table and return false.
        if individualHash.lookup(A[i]) //check if there are duplicates
            if !duplicateHash.lookup(A[i]) //check if duplicate have been recorded
                duplicate.insert(A[i])
    return duplicate
```

Q2. (a)

Function: A algorithm to compute the mean value of each variable in given m samples. I use the linked list to store the indices of items that are included in a solution. So each element in X[0..m-1] is a linked list.

Input: Sample X[0..m-1]

Output: mean value of m samples meanValue.

Complexity: I use a "for loop" to visit each element in X[0..m-1], which contains the. 1st to mst elements. And for each elements in X[0..m-1], I visit every element in the linked list, so the time complexity for my algorithm is $O(\sum_{i=1}^{m}|L_i|)$.

$function\ COMPUTEMEAN(X[1..m])$

$values\ in\ meanValue[1..n]\ are\ all\ 0\ \ //initialize\ array\ to\ store\ mean\ value$

$for\ i \leftarrow 1\ to\ m\ do$

$Initialize\ empty\ linked\ list\ node$

$node \leftarrow X[i]$

$while\ node\ != null\ \ //check\ if\ all\ of\ the\ elements\ in\ node\ has\ been\ visited$

$meanValue[node.val] \leftarrow Value[node.val] + \frac{1}{m}$

$node.next\ \ //visit\ next\ value\ of\ the\ linked\ list$

$return\ meanValue$

(b)

i. The greedy algorithm is always choosing the next piece that offers the most obvious and immediate benefit, which means having the maximum value versus weight ratio in the knapsack problem.

Taking the mean value algorithm into consideration, we can assume that benefit for each item I_i means each items' mean value $meanValue[i-1]$ multiply its' value V_i. So new strategy of the algorithm is always choosing the next item which has the maximum benefit and put it into the knapsack until there is no space left in the knapsack. If there are several items having same benefit, choose the item which has the greatest value V.

For example, if knapsack size is 7, item1is weight2 and value3, item2 is weight3 and value4, item3 is weight4 and value6, item4 is weight5 and value 8. The m possible solution is {1,0,0}, {0,1,0}, {0,0,1}, {1,1,0}. In this situation, we will pick up item3 at first because its benefit is the biggest, then we pick up item1, and we can not find the best solution.

Solution	Item1	Item2	Item3	Item4	Value
1	1	1	0	0	7
2	1	0	1	0	9
3	1	0	0	1	11
4	0	1	1	0	10
Mean Value	0.75	0.5	0.5	0.25	
Benefit	2.25	2.0	3.0	2.0	

ii. I did several tests for the algorithm, only a few of them can find the best solution and most of them found locally optimal solutions. and I found that items who have small value are more likely to be seen in the possible solution, and this will have some impacts on the benefit of each item.

The algorithm described in part i can't always produce an optimal solution, sometimes it only produces a locally optimal solutions that approximate the globally optimal solution. Besides, simply increase the number of possible solutions may not let the algorithm find the optimal solution, so I think such algorithm is not effective.

Q3. (a)

Function: A algorithm to count the number of nodes, which is greater than lo and lower. than hi, in AVL tree. The lo and hi will be given when the function is calling. E means empty node in the algorithm.

Input: Root T of the AVL tree, the lowest value lo, the highest value hi.

Output: The number of the node n.

Complexity: My algorithm visits every node in the AVL tree for one time, so the time. complexity of the algorithm is O(n), and n is the number of nodes in the AVL tree.

$function\ RANGECOUNT(T, lo, hi)$

$if\ T\ is\ empty$

$number \leftarrow 0$

$if\ T.key < lo$

$number \leftarrow RANGECOUNT(T.right, lo, hi)$

$if\ T.key > hi$

$number \leftarrow RANGECOUNT(T.left, lo, hi)$

$if\ T.key \geq lo\ and\ T.key \leq hi$

$number \leftarrow 1 + RANGECOUNT(T.left, lo, hi) + RANGECOUNT(T.right, lo, hi)$

$return\ number$

(b)

I assumed that the AVL tree has two additional attributes called T.numLeft and T.numRight for each node. T.numLeft means the number of nodes in the left sub-tree of node T. And T.numRight means the number of nodes in the right sub-tree of node T.

Algorithm: The task to find the number of nodes in the AVL tree whose value is >= lo and <= hi can be divided into two steps. If the root node is >= lo and <= hi, just operate the step1 and step2. But if the start node is <lo or >hi, then firstly find the child node of the root node which is >= lo and <= hi and operate step1 and step2.

Step1: Start from the root node T and use function LeftCount. For the left-hand side of the root node, keep finding T.left of the node T and return 1+ T.numRight + LeftCount(T.left) when node T is >= lo and <= hi. If node T <lo, then return LeftCount(T.right) until the node get back to >= lo or reach bottom of the tree. If node successfully get back to >= lo, then keep operating Step1 until reach bottom.

Step2: Start from the root node T and use function RightCount. For the right-hand side

of the root node, keep finding T.right of the node T and return 1+ T.numLeft + RightCount(T.right) when node T is >= lo and <= hi. If node T>hi, then return RightCount(T.left) until the node get back to <= hi or reach bottom of the tree. If node successfully get back to <= hi, then keep operating Step2 until reach bottom.
Finally, the algorithm will return the number of nodes in the AVL tree whose value is >= lo and <= hi.
Complexity: The algorithm only have to visit the nodes on the left edge and right edge, and the number of nodes between the left edge and right edge can be known using the T.numLeft and T.numRight attributes. So time complexity of the algorithms is O(logn) + O(logn) = O(logn).

Q4. (a)
Function: A algorithm to calculate the minimum number of operations to transform one. string into another given string.
Input: String A[0..n-1], string B[0..m-1]
Output: Minimum number.
Complexity: The time complexity is O(mn).

$function\ TRANSFORM(A[0..n-1], B[0..m-1])$
    $Initialize\ matrix\ Num[0..n][0..m]$
    $for\ j \leftarrow 0\ to\ n\ do$
        $Num[0][j] \leftarrow j$
    $for\ i \leftarrow 0\ to\ m\ do$
        $Num[i][0] \leftarrow i$
    $for\ i \leftarrow 1\ to\ m\ do$
        $for\ j \leftarrow 1\ to\ n\ do$
            $if\ B[i-1] = A[j-1]$
                $Num[i][j] \leftarrow Num[i-1][j-1]$
            $else$
                $Num[i][j] \leftarrow 1 + \min(Num[i-1][j-1], Num[i-1][j], Num[i][j-1])$
    $return\ Num[m][n]$

(b)
There are m*n possible sub solutions to solve the problem, the algorithm computes the number of operations that need to be done for each sub solution using the dynamic programing. So the time complexity is O(mn). Use the function to operate the given arrays as an example, The blue square is the matrix, and function return 2.

		0	1	2	3	4
	0	1	2	3	4	5
0	1	0	1	2	3	4
1	2	1	0	1	2	3
3	3	2	1	1	1	2
5	4	3	2	2	2	2

Just from the function, the time complexity of the first “for loop” is O(m). the time complexity of the second “for loop” is O(n). The third and forth “for loop” are nested. The time complexity of the third and forth “for loops” is O(mn). So the time complexity of the function is O(mn).