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).
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.

ii. The algorithm described in part i does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution. So I think such algorithm is 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)

$function\ RANGECOUNT(T, lo, hi, rootKey)$
    $if\ T = E$
        $number \leftarrow 0$
    $if\ T.key < lo$
        $number \leftarrow RANGECOUNT(T.right, lo, hi)$
    $if\ T.key > hi$
        $number \leftarrow RANGECOUNT(T.left, lo, hi)$
    $if\ T.key \geq lo\ and\ T.key \leq hi\ and\ T.key \leq rootKey$
        $number \leftarrow 1 + RANGECOUNT(T.left, lo, hi, rootKey) + T.left.numRight$
    $if\ T.key \geq lo\ and\ T.key \leq hi\ and\ T.key > rootKey$
        $number \leftarrow 1 + RANGECOUNT(T.right, lo, hi, rootKey) + T.right.numRight$
    $return\ number$

<u>Q4.</u> (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).

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

$for\ j \leftarrow 1\ to\ n-1\ do$

$if\ B[i] == A[j]$

$Num[i][j] \leftarrow Num[i-1][j-1]$

$else$

$Num[i][j] \leftarrow 1 + \min\left(Num[i-1][j-1], Num[i-1][j], Num[i][j-1]\right)$

$return\ Num[m-1][n-1]$

(b)

There are m multiply n possible sub solution 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).