CSE-321 Introduction to Algorithm Design, Fall 2020
Homework 2

ANSWERS

Q1 – Insertion sort algorithm ; to sort an array if size n in ascending order.

Step 1 =Iterate from array[1] to array[n] over the array.

Step 2= Comprate the current element key to its predecessor.

Step 3 = If the key element is smaller than its predecessor, compore it to the elements before.Move the greater elements one position up to make space for swapped element.

Start array = 

Then sorting ;

For loop for i =1 to 6

Step 1 = We start array[1], since 5 is smaller than 6, move 6 and insert 5 before 6.

New array = 

For loop for i =2 to 6

Step 2 = Since 3 smaller than 5 and 6 then 3 will move to the beginning all other elements, 5 and 6 move 1 position ahead.

Then new array = 

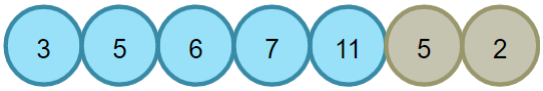
For loop for i =3 to 6

Step 3 = Since 11 greater than 3,5 and 6, then 11 remain at current position.

Then new array= 

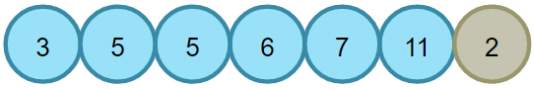
For loop for i =4 to 6

Step 4 = Since 7 greater than 6, 7 will move to position after 6, and 11 will one position ahead of its current position.

Then new array = 

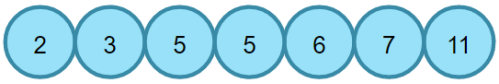
For loop i=5 to 6

Step 5 = Since 5 equall 5 (arr[5] = arr[1]), 5 will move to position after 5, and 6,7,11 will one position ahead of their current position.

Then new array = 

For loop i=6 to 6

Step 6 = Since 2 smaller than all other elements, 2 will move to beginning and all other elements will move one position ahead of their current position.

Then new array = 

Then sorting finished.

Q2 –
a)

```
function(int n){
   if (n==1)
     return;
   for (int i=1; i<=n; i++){
     for (int j=1; j<=n; j++){
        printf("*");
        break;
     }
   }
}
```

Even though the inner loop is bounded by n it is executing 1 time because there is break statement.So inner loop time comlexty O(1). But outer loop executes n times, so time complexity of this function is O(n) * O(1) = O(n).

Answer = O(N)

b)

```
void function(int n)
    int count = 0;

     for (int i=n/3; i<=n; i++)
       for (int j=1; j+n/3<=n; j = j++)
          for (int k=1; k<=n; k = k * 3)
               count++;
}
```

The inner (third) loop variable is multiplied by a constant 3 each time.So the inner loop executes **logn** times.Then, time complexity of inner loop is **O(logn).**

The middle (second) loop variable is increment by a constant variable n/3 and 1.So the middle loop executes **n/3** times.So, time complexity of middle loop is **O(n).**

The outer (first) loop variable started n/3 and loop variable is increment by a constant variable 1.So the outer loop executes **2n/3** times.So, time complexity of first loop is **O(n).**

So time complexity of function is **O(n).O(n).O(logn) = O(n^2logn).**

Answer = O(n^2logn)

Q3 – Since we are asked to reach the solution in O(nlogn) complexity, in this question I used the AVL tree, which is a self balanced binary search tree. Because the insertion,searching and removing processes of self balanced binary search trees are performed in O(logn) time complexity.
Source for AVL tree= https://en.wikipedia.org/wiki/AVL_tree

Using the AVL tree, I could find the pairs in nlogn, so I used the avl tree.

First, I implemented the AVL tree. Then I added all the elements in the array to AVL tree. Then I found the numbers that are divided into the target number in the array. I searched the AVL tree for the numbers I found. If there were any, I printed them and deleted those numbers to avoid finding them again.

Pseudocode :

```
findpair(array,target)
    create new AVL_TREE
    for i in array
        insert i in AVL_Tree
    for i in array
        if target % i ==0
            if target/i is exist in AVL_TREE
                print (i, target/i)
                delete i from AVL_TREE
                delete target/i from AVL_TREE
```

Analysis : First for loop executes n times, insert (avl tree) method works in logn time complexity.So first loop time complexity O(n).O(logn)=**O(nlogn)**

Second for loop executes n times , is exist (search) method works O(logn) time and delete method works also O(logn) time then time complexity inside the second for loop O(logn) then second for time complexity O(n).O(logn)=**O(nlogn).**

Function time complexity is O(nlogn)+ O(nlogn)= **O(nlogn)**

Q4- Merge two binary search tree withn n nodes.

Pseudocode :

```
merge_trees(tree1,tree2)
    convert tree1 to ordered array with size s1 named arr1
    convert tree2 to ordered array with size s2 named arr2
    create array with name merged
    counter1=0
    counter2=0
    while counter1 < s1 and counter2 < s2 do:
        if arr1[counter1] < arr2[counter2] :
            merged.add(arr1[counter1])
            counter1++
        else:
            merged.add(arr2[counter2])
            counter2++
    while counter1<s1 do:
        merged.add(arr1[counter1])
        counter1++
    while counter2<s2 do:
        merged.add(arr2[counter2])
        counter2++

    return sortedArrayToBST(merged,0,len(merged)-1)
```

```
sortedArrayToBST(array,start,end)
    if start > end :
        return None

    midElement=(start+end)/2
    Node = arr[mid]

    Node.left=sortedArrayToBST(arr,start,midElement-1)
    Node.right=sortedArrayToBST(arr,midElement+1,end)

    return Node
```

Analysis : Converting tree1 and tree2 to ordered array takes O(n) times both of them by using inorder traversal.Since tree1 and tree2 has n nodes, ordered arrays size s1 and s2 are n.Then merging orderes lists takes O(n)+O(n) = O(2n) (in short O(n)) times because there is 2 cunsor named counter1 and counter2.After merging into the one array named merged, then make a bst tree from sorted array named merged. Time complexity of the part up to here

Sorted array to bst function; Finding mid element and linking left and right subtrees take constant time.Creating left subtree O(n/2) and creating right subtree O(n/2) because each element visited only once, then sortedArrayToBST time complexity 2O(n/2)+c = **O(n)**

So the worst case of this function O(n)+O(n) = **O(n).**

Q5 – Finding small array elements in big array.

I took advantage of the hash table to solve this problem.Because search, insert and delete operations takes usually O(1) time. First, I placed all the elements in the large array into the hash table. Then I checked whether the elements in the small array are in the hash table. The element is common if it has a hash table.

Pseudocode :

```
findduplucate(arr1[],arr2[])
    if arr1.len >arr2.len
        create hashTable myHash
        for m in arr1
            add m to myHash
        for k in arr2
            if k is exist in myHash
                print k
    else
        findduplucate(arr2,arr1)
```

Analysis : If the size of the first incoming array is smaller than the size of the other array, it goes to the else and changes the location of the arrays and resends them to the same function.This process take O(1) constant time.First loop executest arr1.len times, adding in hash table takes constant time then first loop time complexity O(arr1.len) = **O(n)** . Second for loop executes arr2.len times, searching in hash table takes constant time then second loop time complexity O(arr2.len) = **O(m)**.Then time complexity of this function is linear time which is O(n)+O(m) = **O(n+m).**

But if too many elements were hashed into the same key in hash table or once a hash table has passed its load balance, it has to to rehash.In these cases hash table search and insert operations takes O(n) times.In the worst case for loop and hash table search/add operations both takes O(n) time.

So in the worst case findduplicate function takes O(n).O(n) = **$O(n^2)$.**

Source = https://en.wikipedia.org/wiki/Hash_table