CSE 321 HOMEWORK 2
SOLUTIONS

171044098
Akif Kartal

1) In insertion algorithm if we write pseudocode from our lecture.

```
procedure InsertionSort(L[1:n])
    for i=2 to n do
        current = L[i]
        position = i-1
        while((position ≥ 1) and (current < L[position])) do
            L[position+1] = L[position]
            position = position-1
        end while
        L[position+1] = current
    end for
end
```

⇒ So, from the algorithm we have two variables:

- Current
- Position

by using these let's apply insertion sort on array.

*** Insertion Sort Step by step**

6	5	3	11	7	5	2

pos current

(No element in the left side change current)

6	5	3	11	7	5	2

current

(Now, compare current element with all elements in the left side of current element (5<6 so swap them))

(No left element, then change current element)

5	6	3	11	7	5	2

current

(Compare current element with all elements in the left side)

3	5	6	11	7	5	2

current

(3 < 6 so, swap them)

(3 < 5 so, swap them)

(No left element change current element)

(Compare 11 with 6, 5 and 3 but none of them is bigger so change current element without any swap.

3	5	6	11	7	5	2

current

(Compare current element with all elements in the left side

(7 < 11 so, swap them)

(No bigger element so change current element)

3	5	6	7	11	5	2

current

Compare current element with all element in the left side

(5 < 11 so, swap them)

(5 < 7 so, swap them)

(5 < 6 so, swap them

(5 = 5 so, no swap is needed)

(No bigeer element so change current element)

3	5	5	6	7	11	2

Current

Compare current element with all element in the left side

(2 < 11 so, swap them)

(2 < 7 so, swap them)

(2 < 6 so, swap them)

(2 < 5 so, swap them)

(2 < 5 so, swap them)

(2 < 3 so, swap them)

(No bigger element left side)

2	3	5	5	6	7	11

Sorting is done.

2)

a)

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

Apply Table Method

Steps	freq	Total
1	1	1
1	1	1
2	n+1	2n+2
2	n	2n
1	n	n
1	n	n

Total = 6n+4

Analysis

When we observe the algorithm, we see that normally we have nested loop which can give us a quadratic time complexity. But in nested loop since we have "break" statement that stops inner loop, instead of quadratic time we have linear time complexity. Since we have a condition statement in algorithm let's make analysis with $f(n) = 6n+4$;

⇒ Best Case

$T_{best}(n) = \Theta(1)$ (Since we have a condition we have best case) (if n==1)

⇒ Worst Case

$T_{worst}(n) = \Theta(f(n)) = \Theta(n)$

⇒ Average Case

$T_{avg}(n) = O(f(n)) = O(n)$

b)

Line numbers

```
void function(int n)
    int count = 0;  ——————————————> (1)
    for (int i = n/3; i<=n; i++) ————> (2)
        for(int j = 1; j+n/3 <= n; j = j++) ——> (3)
            for(int k=1; k<=n; k=k*3) ——> (4)
                Count++; ——————> (5)
}
```

⇒ Let's apply table method by using Line numbers

Line number	steps	freq	Total
1	1	1	1
2	2	$\frac{2n+3}{3}+1$	$\frac{2n+6}{3}$
3	2	$\frac{2n+3}{3}\left(\frac{4n}{3}+1\right)$	$\frac{8n^2+18n+9}{9}$
4	2	$\frac{8n^2+12n}{9}\left(\log_3 n+1\right)$	$\frac{8n^2+12n}{9}\log_3 n+\frac{8n^2+12n}{9}$
5	1	$\frac{8n^2+12n}{9}\log_3 n\,(1)$	$\frac{8n^2+12n}{9}\log_3 n$

$$\text{Total} = \frac{16\log_3 n.\,n^2+24\log_3 n.\,n}{9}+\frac{16n^2+36n+36}{9}$$

if we choose biggest degree function from total count; $f(n)=n^2\log_3 n$, since we don't have any condition statement in the algorithm we have;

$$\boxed{T(n)=\Theta(f(n))=\Theta(n^2\log_3 n)}$$ (ignore constant terms)

3) if we observe the question we will see that, in order to solve this problem in O(nlogn) complexity we have to use a sorting algorithm which has O(nlogn) complexity. Because when we sort the array, our Job will be easier, since on sorted array we can find pairs for desired number with a simple algorithm that has linear complexity.

=> I will use merge sort to sort array, since merge sort has very efficent algorithm such that in O(nlogn) complexity

Pseudocodes for both algorithms

```
getMultiplicantPairs(array[], arr_length, desired_element){
O(nlogn) <- mergeSort(array)   // first sort the given array
        down = 0
        up = arr_length - 1
C1 <-   PairArray[] // pairs that was found.
        k=0

        while(down < up){
            multResult= array[down] * array [up]
            if(multResult == desired_element){
                newPair
                newPair.x = array[down]
                newPair.y = array[up]
                PairArray[k] = newPair
O(n) <-         k++ down++ up-- // update values
            }
            else if (multResult > sum){
                up--
            }
            else{
                down++
            }
        }
} C2 <- return PairArray
```

⇒ Merge Sort Algorithm

Steps of merge sort

1) Split the array into two halves
2) Sort the left half
3) Sort the right half
4) Merge the two

Pseudocode

```
mergeSort(A){
      n = lenght(A)
      if(n<2) // base case
            return
C1    mid = n/2
      left[mid]
      right[n-mid]
      for i=0 to mid-1
c2.n      left[i] = A[i]
      for i=mid to n-1
          right[i-mid] = A[i]
T(n/2) <- mergeSort(left)
T(n/2) <- mergeSort(right)
c3.n+c4 <- merge(left,right,A)
}// end of merge sort
```

c_1, c_2, c_3 and c_4 are constant.

```
merge(L,R,A){ //merge operation
   nL = lenght(L)
   nR = lenght(R)
   i=0, j=0, k=0
   while(i<nL && j<nR){
      if(L[i] <= R[j]){
          A[k] = L[i]
          k = k+1
          i = i+1
      }
      else{
          A[k] = R[j]
          k = k+1
          j = j+1
      }
   }
   while(i<nL){
       A[k] = L[i]
       i = i+1
       k = k+1
   }
   while(j<nR){
       A[k] = R[j]
       j = j+1
       k = k+1
   }
}//end of merge function
```

Analysis's of Algorithm

if we observe algorithm we will see that, I have used merge sort in algorithm so, lets first analyze merge sort algorithm.

⇒ Properties of merge sort

1) Divide and Conquer algorithm
2) Recursive
3) Stable
4) Not In-place
5) $O(n\log n)$ time complexity
6) $O(n)$ space complexity

⇒ Prove of time complexity of m.s

From the pseudocode we have:

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(n/2) + c.n, & \text{if } n > 1 \end{cases}$$

if we divide $T(n/2)$ by two contiunally

$$T(n) = 2\{2T(n/4) + c.\tfrac{n}{2}\} + c.n$$
$$= 4T(n/4) + 2c.n$$
$$= 4\{2T(n/8) + c.n/4\} + 2cn$$
$$8T(n/8) + 3cn$$

⋮ it will contiune like this

$$= 2^k T(n/2^k) + k.c.n$$
$$= \frac{n}{2^k} = 1 \Rightarrow 2^k = n$$
$$k = \log_2 n$$
$$= 2^{\log_2 n} T(1) + \log_2 n.c.n$$
$$= n.c + c.n.\log n$$
$$\boxed{= O(n\log n)} \text{ (ignore constant terms)}$$

Now, we have found and prove merge sort complexity is in $O(n\log n)$ time.

Let's find our algorithm complexity. From pseudocode of my algorithm we have:

$$\Rightarrow O(n\log n) + c_1 + O(n) + c_2$$

⇒ if we ignore constant terms and if we choose biggest degree BigOh(O) function we will get $O(n\log n)$ complexity from my algorithm

$$\Rightarrow \boxed{T(n) = O(n(\log n))}$$

Note = I used master teorem to prove merge sort algorithm.

Note = Running time of steps can be found on psedocodes.

4) When we observe this question to merge two binary tree first option is for each node in Second bst take one them and first delete it and after inset it into appropriate position in first bst.

But if we analyze this operation deletion from bst will take $O(n)$ time and add will take $O(\log n)$ time for each element so eventualy we will get $O(n \log n)$ time to merge two tree.

But if we think more about BST we can find better solution.

Let see a better and simple solution for this problem.

Note that we are always trying to find better algorithms.

When we think about binary search trees, we see that they are actually sorted binary tree and because of this Property of binary search trees when we traverse all tree by using inorder traversal method we can easily get sorted array version of n nodes binary search tree.
Thus, with these sorted arrays by using extra space we can easily merge two binary search tree as we did in 3rd question of this assingment by using sorted array.

Let's explain the procedure

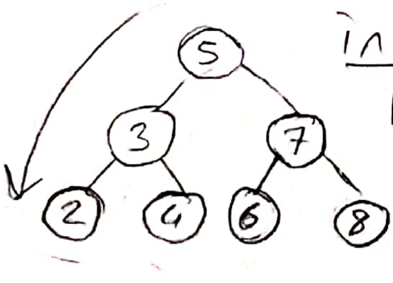
Steps of this procedure

1) Traverse both bst with inorder traversal method and get the sorted arrays.
2) Then, merge these sorted array by using merge function of merge sort algorithm.
3) Create a new BST from merged sorted array. Note that new merged BST will be balanced BST.

Time complexity analysis of these Steps.

Step 1 Let n = number of nodes in first bst.
Let m = number of nodes in second bst.

To explain it let's draw a simple BST



inorder traversal = Left - root - right

In this traversal method since we will traverse all n node of bst we will get $O(n)$ complexity for a n node binary search tree and if we say Second binary search tree has m node in total we will get $O(n+m)$ time coplexity. (n = number of nodes in fist tree. m = nmber of nodes in second tree)

Scanned with CamScanner

Step 2 When we got sorted arrays from step 1 we can start to merge them.

To merge two sorted array we will follow these steps;

1) Create a new array with size $n+m$
2) Traverse both sorted array while comparing current elements of these arrays and select smaller one and put it new created array.

⇒ In this algoritm in 2^{nd} step since we traverse both arrays we will get again $O(n+m)$ complexity for this step.

Step 3 Now, we can start to create a merged bst.

Note that our sorted array has $n+m$ element.

A simple algorithm for this operation can be following;

1) Determine middle element of array and make it root.
2) By using recursion;
   - Determine middle element of left side and make it left child of the root in step 1.
   - Determine middle element of right side and make it right child of the root in step 1.

Since our sorted array has $n+m$ element this operation will take in $O(n+m)$ completixty.

Analysis ($n+m$ is total number of nodes in two bst.)

Step1 = $O(n+m)$
Step 2 = $O(n+m)$ ⇒ Total we have;
Step 3 = $O(n+m)$

$T(n) = O(n+m)$ which is a linear time.

5) In algorithms world there is a tradeoff between Performance and space. Therefore, if you want to better performance from your algorithm you must sacrifice from your space.

Thus, this question wants to a linear time algorithm. Therefore we need to sacrifice from our space.

Let's find out some solutions for this problem

---

1) Classical nested loop (traverse both array)

- Complexity $= O(n*m)$ (Quadratic)

2) Since this assignment focus on sorting algorithms we can use a sorting algorithm to solve this problem if we use merge sort + binary search we will get $O(n\log n + m\log n)$ complexity which is not a linear time.

⇒ Thus, above two option we observe that, to solve this problem with linear time algorithm, we need to a data structure such that inserting, searching and deleting should be in $O(1)$ time complexity.

This definition lead us on HashTable data structure which support these conditions.

Let's write pseudocode by using HasTable data structure.

Let A[ ] = larger array | m = size of array A

B[ ] = smaller array | n = size of array B

```
find-if-exists( A[], m, B[], n){

1 ← HashTable table(m+n) // a hashtable with size m+n
      for i=0 to m-1 {
m ←       table.insert(A[i]) → O(1)
      }

      for i=0 to n-1 {
          if (table.search(B[i])) { → O(1)
              table.delete(B[i]) → O(1) // delete for repeated elements
          }
n ←       else{
              return -1 → O(1)
          }
      }
1 ← return 1
}
```

Total operation

$1+m+n+1 = m+n+2$

If we ignore constant time; → Linear Time

$\boxed{T(n) = O(n+m)}$ (Average Time)

Note= HashTable insert, delete and search method takes $O(1)$ time in both best and average case.

Worst-Case Analysis

In worst case, since HasTable operations (search, delete, insert) will take $O(n)$ time becouse of collisions with same elements. Thus worst case is;

$\boxed{W(n) = O(m^2+n^2)}$

which is quadratic time.

Note that this algorithm works repetitive elements in arrays because we are deleting if element exist.