HOMEWORK #2 INTRODUCTION TO ALGORITHM

Mustafa Tolgöz
171044097

Answers of Questions:

1)

Array = {6, 5, 3, 11, 7, 5, 2} n = 7

Insertion Sort:

First step:

i = 2 i ≤ n

current element = Array[i] = Array[2] = 5

position = i - 1 = 1

because of position ≥ 1 and

current element < Array[position]

5 6

Array[2] = Array[1]

{6, 6, 3, 11, 7, 5, 2}

position = position - 1 ⇒ position = 0

then position is NOT ≥ 1

Array[position + 1] = current

Array[1] = 5

Array = {5, 6, 3, 11, 7, 5, 2}

i++ ⇒ i = 3

Second Step = Array = {5, 6, 3, 11, 7, 5, 2} i ≤ n

↑ i = 3

current element = 3

{5, 6, 3, 11, 7, 5, 2}
↑position position=2

because of 3 < 6 ∧ position ≥ 1 (Array[position+1]=Array[position]

Array={5, 6, 6, 11, 7, 5, 2}
↑position position=1

because of 3 < 5 ∧ position ≥ 1 (Array[position+1] =Array[position]

Array={5, 5, 6, 11, 7, 5, 2}
↑position=0

because of position=0 ⇒ Array[position+1]=current element

Array={3, 5, 6, 11, 7, 5, 2}

i++ ⇒ i=4

Third Step= i≤n

Array={3, 5, 6, 11, 7, 5, 2}
↑i position=3

current element=11

{3, 5, 6, 11, 7, 5, 2}
↑

because of 11 is not < 6 then Array[position+1]=current
Array[4]=11

i++ ⇒ i=5

Fourth Step=

i≤n n=7

Array = {3,5,6,11,7,5,2}
↑ i=5

current element= 7

position= 4

{3,5,6,11,7,5,2}
↑ position=4

because of 7<11 ∧ position≥1

Array[position+1] = Array[position]

Array = {3,5,6,11,11,5,2}
↑ position=3

because of 7 is NOT <6

then

Array[position+1] = current element

Array = {3,5,6,7,11,5,2}

i++ ⇒ i=6

Fifth Step=

i≤n i=6

Array = {3,5,6,7,11,5,2}
↑ i

position= 5

current element=5

{3,5,6,7,11,5,2}
↑ position=5

becouse of

$5 < 11$ and position ≥ 1

Array[position+1] = Array[position]

{3, 5, 6, 7, 11, 11, 2}

↑ position = 4

becouse of $5 < 7$ and position ≥ 1

Array[position+1] = Array[position]

{3, 5, 6, 7, 7, 11, 2}

↑ position = 3

becouse of $5 < 6$ and position ≥ 1.

Array[position+1] = Array[position]

{3, 5, 6, 6, 7, 11, 2}

↑ position

becouse of 5 is NOT < 5

then

Array[position+1] = current element

Array = {3, 5, 5, 6, 7, 11, 2}

i++ => i = 7

Sixth Step =

$i \leq n$ i = 7

position = 6

current element = 2

{3, 5, 5, 6, 7, 11, 2}

↑ position

because of 2<11 and position≥1 position=6

Array[position+1] = Array[position]

Array = {3, 5, 5, 6, 7, 11, 11}
↑ position=5

because of 2<11 and position≥1

Array[position+1] = Array[position]

Array = {3, 5, 5, 6, 7, 7, 11}
↑ position=4

because of 2<4 and position≥1

Array[position+1] = Array[position]

Array = {3, 5, 5, 6, 6, 7, 11}
↑ position=3

because of 2<5 and position≥1

Array[position+1] = Array[position]

Array = {3, 5, 5, 5, 6, 7, 11}
↑ position=2

because of 2<5 and position≥1

Array[position+1] = Array[position]

Array = {3, 5, 5, 5, 6, 7, 11}

Array = {3, 5, 5, 5, 6, 7, 11}
↑ position 1

because of 2 < 3 and position ≥ 1

Array[position + 1] = Array[position]

Array = {3, 3, 5, 5, 6, 7, 11}

position = 0

because of position is NOT ≥ 1

Array[position + 1] = current element

Array = {2, 3, 5, 5, 6, 7, 11}

i++ => i = 8

because of i is NOT ≤ 7 7 = 7

For loop stops

So the Array = {2, 3, 5, 5, 6, 7, 11}

2)

a)
```
function(int n){
    if(n==1)
        return;          ⟶ O(1)

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

O(1) (inner loop) — O(n) (outer loop)

The best case = Ω(1) because if n is equal to 1 then it returns and it takes constant time.

The complexity = O(n)

If we look inner loop it takes O(1) time because of break, the inner loop is only print "*" once.
And the outer loop takes n times because it begins 1 to n. And we multiply outer loop complexity by inner loop complexity time the the function $O(1) + O(n) * O(1) = O(n)$ time complexity.

b)

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

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

$O(n^2 \log n)$ = $O(n \log n)$ × $O(\log_3 n)$

When we look first loop it begins from $1/3$ to n so it takes $n - 1/3$ so $O(n - 1/3)$

the second loop is if we re-write the second loop like that for(j=1; j<=n-1/3; j++) So It begins 1 to $n-1/3$ So It takes $O(n-1/3)$

the third loop, it goes $1, 3, 3^2, 3^3, 3^4 \dots 3^k$

$3^k = n$ $k = \log_3 n$ So it takes $O(\log_3 n)$

Because of that they are all inner

The time complexity of this function is =>

$$O(n - 1/3) \cdot O(n - 1/3) \cdot O(\log_3 n) = O((n-1/3)^2 \cdot \log_3 n)$$

$$\Rightarrow \boxed{O(n^2 \log_3 n)}$$

It also $\Omega(n^2 \log_3 n)$ So $\boxed{\Theta(n^2 \log_3 n)}$

3)

```
Procedure  MergeSort (array)
    if  array length > 1
    var mid = array length /2
     var L as array = array[0], array[1], array[2],
                      ... array [array length/2]
     var R as array = array[array length/2 +1],
                      array [array length/2 +2],
                      ... array[array length]

        L= mergeSort(L)
        R= mergeSort(R)
        var i=k=J=0
        while i< L length and J< R length
            if L[i] < R[J]
                array[k] = L[i]
                i= i+1
            else
                array[k] = R[J]
                J = J +1
            k=k+1
        end if
    end while

    while i< L length
            array[k] = L[i]
            i= i +1
            k= k+1
    end while
```

```
            while J < R length
                array[k] = R[J]
                J = J+1
                k = k+1
            end while

    end if

procedure find_pair (array, desired)
    var left=0
    var right= array length -1

        while left < right
            if (array[left] + array[right] == desired)
                print ( array[left], array[right]
                left = left +1

            else if (array[left] * array[right] < desired)
                left = left +1

            else
                right = right +1

            end if
        end while

procedure main
    var arr = [1,2,3,6,5,4]
    desired_number = 6
    MergeSort (arr)
    find_pair(arr, desired_number)
```

Complexity of the program:

for the merge sort part:

We use master teorem and because of that merge sort devides two that length is 1/2 and merge them O(1) time. The recurrence is $T(n)=2T(n/2)+n$

$a=2 \quad b=2 \quad f(n)=n$

$a>0$ and $b>1$

if $n^{\log_b a} == f(n) \Rightarrow n^{\log_2 2} == n \Rightarrow n=n$

So $T(n) = O(n^{\log_b a} \log(n)) = O(n \log n)$

for the find_pair part

It takes $O(n/2)$ times because of while loop while loop takes $n/2$ times. So for this part the complexity is $O(n)$.

→ for main part

Total $= O(n \log n) + O(n) + O(1)$

So the complexity of the program = $\boxed{O(n \log n)}$

the best case is $\Omega(1)$ when array length 0 or 1.

4)

First we create an array that is inorder from the first binary tree. and to do this we traverse from at the left bottom node to right bottom node by adding to the array. It takes $O(1)$ time. Then we create another array that is inorder from the second binary tree and to do this we do the same thing as we do for first array. Also this takes $O(1)$ time, 1 is number of element. Then we merge these two array into another array. It takes $O(1) + O(1)$ time that is $O(1)$ time also. Then we construct a binary search tree from the merged array. To construct this binary search tree, we firstly get the middle element of array then assign to root. Then we do the same thing for left side of middle and right side middle recorsively with getting the middle of left half and making it left child of the root and getting the middle of right half and make it right child of the root. It takes $O(1)$ time.

Therefore merging two binay search tree takes

$O(n)$ time.

5)

pseudocode:

```
precedure  algorithm( array1 , array2)
        var bigger=1
        if (length of array1 < length of array2):
            bigger=2
        end if

        HashTable table

        if bigger ==1
            for i from 0 to length array1
                table.insert(i, array1[i])
            end for
            for i from 0 to length array2
                if table.search(array2[i])
                    // Noting to do
                else
                    return false.
                end if
            end for
        end if

        if bigger==2
            for i from 0 to length array2
                table.insert(i, array2[i])
            end for
            for i from 0 to length array1
                if table.search(array1[i])
                    // Noting to do
                else
                    return false
```

```
        end if
    end for
end if
    return true
end procedure
```

The worst case Complexity of the algorithm =

$O(1)$ for variables and if sides

$O(n)$ for for loops

$O(n)$ for HashTable search and insert worst cases

$O(1)$ for HashTable search and insert normal cases

var, if ↑ for loop ↑ hash insert ↑ for loop ↑ hash search ↑

So $O(1) + O(n) * O(1) + O(n) * O(n)$

Therefore The worst case complexity is $\boxed{O(n^2)}$

because of insert and search. But normal case the algorithm takes (mean time $O(n)$. Search and insert functions of Hashtable actualy takes $O(1)$ but in the worst case if an element's key is equal to another's element key then element will be placed empty place after there so it takes $O(n)$ time for insert and will be searched other elements after there for search fuctions. So insert and search methods takes $O(n)$ time in the worst case.