Miray YILDIZ
161044023

CSE-321 Introduction to Algorithm Design
Homework 2

1- In insertion sort, we are trying to place the i^{th} element in the proper position in the already sorted part.

Let's examine Array={6,5,3,11,7,5,2} and apply insertion sort.

First version of the array:

0	1	2	3	4	5	6
6	5	3	11	7	5	2

↳ current

1) We start 1^{th} element in array. It's 5. It's current element.
Compare the current element to its predecessor.
5<6, so move greater element one position up.

0	1	2	3	4	5	6
5	6	3	11	7	5	2

2) It has no more predecessor, so 2^{th} element is current element.
Compare the current element to its predecessor
3<6, so move greater element one position up.

0	1	2	3	4	5	6
5	3	6	11	7	5	2

It has one more predecessor, it is 5.
compare 5 and 3.
3<5, so move greater element one position up.

0	1	2	3	4	5	6
3	5	6	11	7	5	2

3) It has no more predecessor, so 3^{th} element is current element.
Compare the current element to its predecessor.
11>6 so no swap.

0	1	2	3	4	5	6
3	5	6	11	7	5	2

→ No change

4) 4^{th} element is current element. Compare the current element to its predecessor.

$7 < 11$, so move greater element one position up.

0	1	2	3	4	5	6
3	5	6	7	11	5	2

It has more predecessor but $7 > 6$ and others.

5) 5^{th} element is current element. Compare the current element to its predecessor.

$5 < 11$, so move greater element one position up.

0	1	2	3	4	5	6
3	5	6	7	5	11	2

It has more bigger than itself predecessor, 7 and 6.

$5 < 7$, move.

0	1	2	3	4	5	6
3	5	6	5	7	11	2

$5 < 6$, move.

0	1	2	3	4	5	6
3	5	5	6	7	11	2

It has no more bigger than itself predecessor.

6) 6^{th} (last) element is current element. Compare the current element to its predecessor.

$2 < 11$, so move greater element one position up.

0	1	2	3	4	5	6
3	5	5	6	7	2	11

It has more bigger than itself predecessor, 7, 6, 5, 5 and 3.

2<7, move.

0	1	2	3	4	5	6
3	5	5	6	2	7	11

2<6, move

0	1	2	3	4	5	6
3	5	5	2	6	7	11

2<5, move

0	1	2	3	4	5	6
3	5	2	5	6	7	11

2<5, move

0	1	2	3	4	5	6
3	2	5	5	6	7	11

2<3, move → loops ended an it is solution.

0	1	2	3	4	5	6
2	3	5	5	6	7	11

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;
        }
    }
}
```

When we execute the function, it see if condition. If parameter n is equal to 1, the function ends. Based on this, we can say that the function's best case is $\Theta(1)$.

If n is not equal to 1, the function continues. When it comes to nested loops, each loop start 1. In the second for loop, program prints * and it break. So, every time, second loop runs 1 time. First loop run n times. Based on this information, worst case is $\Theta(n)$.

Time complexity is $O(n)$ ✓

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

When we execute the function, first loop will run $n-\frac{n}{3}=\frac{2n}{3}$ times. Second loop will run $\frac{2n}{3}$ times too. In the last loop, k is increases to k*3. It is logarithmic. So, this function's complexity is $O(n^2 \log n)$

3) For this part, I sorted array with merge sort and I found pairs whose multiplication yields the desired numbers with binary search.

merge function: This function merge two subarrays of Array[]

```
merge (Array, Left, middle, right)
    n1 ← middle-Left+1
    n2 ← right-middle
    L ← [0]*(n1)
    R ← [0]* (n2)

    for i ← 0 to n1
        L[i] ← Array[i+left]
    end for
    for J ← 0 to n2
        R[J] ← Array[middle+1+J]
    end for
    i ← 0
    J ← 0
    k ← left
```

continued on other page

```
while i<n1 and J<n2
        if L[i] <= R[J]
            Array[k] ← L[i]
            i ← i+1
        end if
        else
            Array[k] ← R[J]
            J ← J+1

        k ← k+1
        end else
end while
while i<n1
        Array [k] ← L[i]
        i ← i+1
        k ← k+1
end while
while J<n2
        Array [k] ← R[J]
        J ← J+1
        k ← k+1
end while
end
```

```
mergeSort (Array, left, right)
        if left<right
            middle = (left + (right - 1)) / 2
            mergeSort ( Array, left, middle)
            merge Sort (Array, middle+1, right)
            merge(Array, left, middle, right)
        end if
end
```

MergeSort is recursive and its complexity is O(nlogn)

continued on → other page

```
findMultiYields (Array, desired, n)
    mergeSort (Array, 0, n-1)
    right ←Len(Array)
    left=0

    while left<right
        value ← Array[left]* Array[right]
        if value=desired
            print ("Found!")
            print ("(", Array[left] , ",", Array[right], ")")
            left ←left+1
        endif
        elif value<desired
            left ← left+1
        end elif
        elif value>desired
            right ← right-1
        end elif
    end while
end

#Testing
Array = [1,2,3,5,6,4]
n = len (Array)
findmultiyields (arr, 6, n)
```

END

In the pseudocode, findMultiyields function actually does binary search. Binary search's time complexities is O(log n). Because as we see in the code, it halves the input set in each iteration. MergeSort function is recursion function. Its complexity is O(nlogn). So, the complexity of the program is O(nlogn).

4) Convert first binary search tree into an array in an order. First binary search tree has n elements. So, this takes $O(n)$ time. Same way, convert second binary search tree into an array in an order. Second binary search tree has n elements, too. So, this takes $O(n)$ time. We create two arrays. Now, we merge this. merge first and second array. It takes $O(n+n)$ times. Build new binary search tree from the merged array. It takes $O(n+n)$ times.

So, time complexity is $O(n+n) \rightarrow O(2n) \rightarrow \underline{\underline{O(n)}}$

5th question $\rightarrow$ on other page

5) In this question, we can use hashset to make the result linear time.

```
isSubset (big_arr, big_length, small_arr, small_length)
        Create hashset.
        for i ← 0 to big_length
              hashset.add(big_arr[i])
        end for
        for i ← 0 to small_length
              if small_arr[i] in hashset
                    continue
              end if
              else
                    return False
              end else
        end for
        return True
end
```

↳ If this function returns true, the elements of the small array in the big array.

I think worst case is O(n) because of same elements or same arrays.