CSE-321 HOMEWORK 2

1) 6 5 3 11 7 5 2

→ Highlighted green elements to the left are always sorted. We begin with the element in position 0 in the sorted part, and we will be moving the element in position 1 (in blue) to the left until it is sorted.

→ Move the blue element to the left until it reaches the correct position.

Swap ⇒ 5 6 3 11 7 5 2

→ Processing element in position 2.

Swap ⇒ 5 3 6 11 7 5 2

3 5 6 11 7 5 2

→ Processing element in position 3.

3 5 6 11 7 5 2

→ Processing element in position 4.

Swap ⇒ 3 5 6 7 11 5 2

→ Processing element in position 5.

→ Move the blue element (position 5) to the left until it reaches the correct position

Swap ⇒ 3 5 6 7 5 11 2

3 5 6 5 7 11 2

3 5 5 6 7 11 2

→ Processing element in position 6.

Swap ⇒ 3 5 5 6 7 2 11

3 5 5 6 2 7 11

3 5 5 2 6 7 11

3 5 2 5 6 7 11

3 2 5 5 6 7 11

2 3 5 5 6 7 11

Done Sorting!

2) a)

```
for (int i=1; i<n; i++){          → (n-1) times
    for (int j=1; j<=n; j++){      → 1 time
        print ("*");
        break;
    }
}
```

If n is 1, it enters an if statement and the program ends. (Best case $O(1)$). In the first loop, $(n-1)$ operations are done. Since there is a break in the second loop, it performs once and finishes the loop no matter how many n times.

Therefore; time comlexity $= O(n)$

b)

```
c. for (int i=n/3; i<=n; i++)              → (n/3) times
b.   for (int j=1; j+n/3<=n; j=j++)        → (n/3) times
a.     for (int k=1; k<=n; k=k*3)          → (logn) times
         count++;
```

a.

$k=1 \rightarrow 3^0$
$k=3 \rightarrow 3^1$
$k=9 \rightarrow 3^2$
$k=27 \rightarrow 3^3$
$k=n \rightarrow 3^i$

$n = 3^i \rightarrow \log_3 n = i$

$O = (\log n)$

b. $\frac{n}{3}+1+\frac{n}{3}+2 \dots + n \Rightarrow O\left(n-\frac{n}{3}\right) = O\left(\frac{2n}{3}\right) = O\left(\frac{n}{3}\right)$

c. $\frac{n}{3}+1+\frac{n}{3}+2 \dots + n \rightarrow O\left(n-\frac{n}{3}\right) = O\left(\frac{2n}{3}\right) = O\left(\frac{n}{3}\right)$

Therefore; time complexity $= O(n^2 \log n)$

3.

```
procedure MergeSort (L[0:n])
        if n<1
            return 0
        end if
        left = L[0:n/2]
        right = L[n/2+1:n]
        MergeSort (left)
        MergeSort (right)
        Merge (left, right, L)
    end
```

CamScanner ile tarandı

```
procedure Merge (Left[0:n/2], Right[n/2+1:n], L[0:n])
    i = j = k = 0
    while i < len(Left) and j < len(Right) do
        if Left[i] < Right[j]
            L[k] = Left[i]
            i++
        else
            L[k] = Right[j]
            j++
        end if
        k++
    end while
    while i < len(Left) do
        L[k] = Left[i]
        i++
        k++
    end while
    while i < len(Right) do
        L[k] = Right[j]
        j++
        k++
    end while
end

procedure findPairs (Arr[0:n], number)
    MergeSort (Arr)
    left = 0
    right = n-1
    while left < right do
        sum = Arr[left] * Arr[right]
        if sum < number
            left++
        elif sum > number
            right --
        else
            print ("Number: " + Arr[left], Arr[Right])
            left++
            right --
        end if
    end while
end
```

→ MergeSort Analysis

Master Theorem: Let $x(n)$ be an eventually non-decreasing function that satisfies the recurrence relation:

$$x(n) = a\,x(n/b) + f(n) \;;\; n = b^k \quad (k = 1, 2 \ldots)$$

$x(1) = c$, where $a \geq 1$, $b \geq 2$, $c > 0$

If $f(n) \in \Theta(n^d)$ where $d > 0$ then

$$x(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So $T(n) = 2T(n/2) + n$

→ merge part

→ dividing part

$a = 2$
$b = 2$
$d = 1$

$a = b^d \Rightarrow 2 = 2^1 \qquad \Theta(n \log n)$

→ Find Pairs Analysis

As seen in our while loop, it starts scanning from the begining of the left array and right from the end. At worst, only one of these values (left or right) changes and one stays in place. This means the loop continues only $(n-1)$ times

So, $\Theta(n\log n + (n-1)) \Rightarrow \Theta(n \log n)$

4) We can think of this problem in 2 ways. The first is array logic. In this we need to insert element by element.

In the second case, we think with node logic. So, when root is inserted, the whole tree is connected. Because we can traverse the tree from the root as right child and left child. Nodes are interconnected

x → element to insert. Since it will be inserted into an existing tree, it is always inserted at the end of the tree.

$x = L[1]$ or $x = L[n]$

Compare x with $L[2^{k-1}], L[2^{k-2}], \ldots\, L[1]$

$n = 2^{k-1} \rightarrow k = \log_2(n+1)$

so, Search → $\Theta(\log n)$

→ node insert $\Theta(1) \rightarrow \in \Theta(\log n)$

→ array element one by one (n elements) (n times search) $\rightarrow \in \Theta(n \log n)$

CamScanner ile tarandı

5) We can only do this problem with the hash method in linear time. In python, what it meet is the dictionary.

Get dictionary $\rightarrow O(1)$

```
procedure   check ( big-arr , small-arr ):
        for i in big-arr :
                dict[i] = 1
        end for
        for i in small-arr :
                if dict.get(i) == None :
                        return 0
                end if
        end for
        return 1
end
```

Size of big-arr is n and size of small-arr is m.

First loop, adding big-arr to dictionary is $O(n)$ complexity.

Second loop, function of get() is $O(1)$ complexity. Loop returns m times

Best Case : First element of small-arr is not in the dict.

$$\text{So, } O(n+1) = O(n)$$

Worst Case: Each element of small-arr is element of big-arr

$$\text{So }, O(m+n)$$