Yunus Emre Öztürk
1901042619

Part 1:

Array Container Part:

n = Array size

```
public void add(T element){
    if(size >= cap){
        setCap(2*cap);
    }
    arr[size++] = element;
}
```

O(1) Amortized constant O(1) } O(1)

```
public int getSize(){
    return size;
}
```

} O(1)

```
public T get(int index) throws IndexOutOfBoundsException{
    if(index < getSize() && index >= 0)
        return arr[index];
    else
        throw(new IndexOutOfBoundsException());
}
```

$T_b = O(1)$
$T_w = O(1)$

```
public boolean contains(Object o){
    @SuppressWarnings("unchecked")
    T element = (T) o;
    for(int i = 0; i < getSize(); ++i){
        if(arr[i].equals(element))
            return true;
    }
    return false;
}
```

O(n)

$T_b = O(1) \Rightarrow O(n)$
$T_w = O(n)$

Item Class:

```
public String getName(){
    return name;
}
```

O(1) O(1)

```
public void setQuantity(int quantity) {
    if(quantity >= 0)
        this.quantity = quantity;
    else{
        throw(new IllegalArgumentException());
    }
}
```

O(1) O(1) O(1)

$T_b = O(1)$

$T_w = O(1)$

```
public int getQuantity() {
    return quantity;
}
```

O(1)

1) **Search Product:** => O(n)

```
public ArrayContainer<Item> searchProduct(String name){
    ArrayContainer<Item> founded = new ArrayContainer<>();
    for(int i = 0; i < items.getSize(); ++i){
        if(items.get(i).getName().equals(name)){
            founded.add(items.get(i));
        }
    }
    return founded;
}
```

O(n) O(1) O(n) O(1) O(n)

2) **Add/Remove Product:** Add: O(n) Remove: O(n)

```
public boolean addToDepot(Item item){
    if(item == null){
        throw(new IllegalArgumentException());
    }
    if(item.getQuantity() < 0)
        return false;

    if(!items.contains(item))
        items.add(item);
    else
        for(int i = 0; i < items.getSize(); ++i)
            if(items.get(i).equals(item))
                items.get(i).setQuantity(items.get(i).getQuantity() + item.getQuantity());
    return true;
}
```

O(1) O(1) O(1) O(n) O(n) O(n) O(1)

$T_b = O(1)$, $T_w = O(n)$ => O(n)

```
public boolean removeFromDepot(Item item){
    if(item == null){
        throw(new IllegalArgumentException());
    }
    for(int i = 0; i < items.getSize(); ++i)
        if(items.get(i).equals(item)){
            if(items.get(i).getQuantity() >= item.getQuantity()){
                items.get(i).setQuantity(items.get(i).getQuantity() - item.getQuantity());
                return true;
            }
            else{
                return false;
            }
        }
    return false;
}
```

O(1) O(1) O(1) O(1) O(n) O(n) O(1)

$T_b = O(1)$, $T_w = O(1)$

3) **Querying the product:** => O(1)

```
public void addRestockInfo(Branch branch, Item item){
    if(item == null || branch == null){
        throw(new IllegalArgumentException());
    }
    box.add(branch.getName() + item.toString());
}
```

O(1) O(1) O(1) => O(1)

(model, color, name) so toString is constant

Part 2:

a) It's meaningless to use "at least" for "O" notations because "O" notations shows us the upper bound of time complexities. We should know "Ω" notation to explain "at least" for algorithms. We can use "at most" for "O" notations.

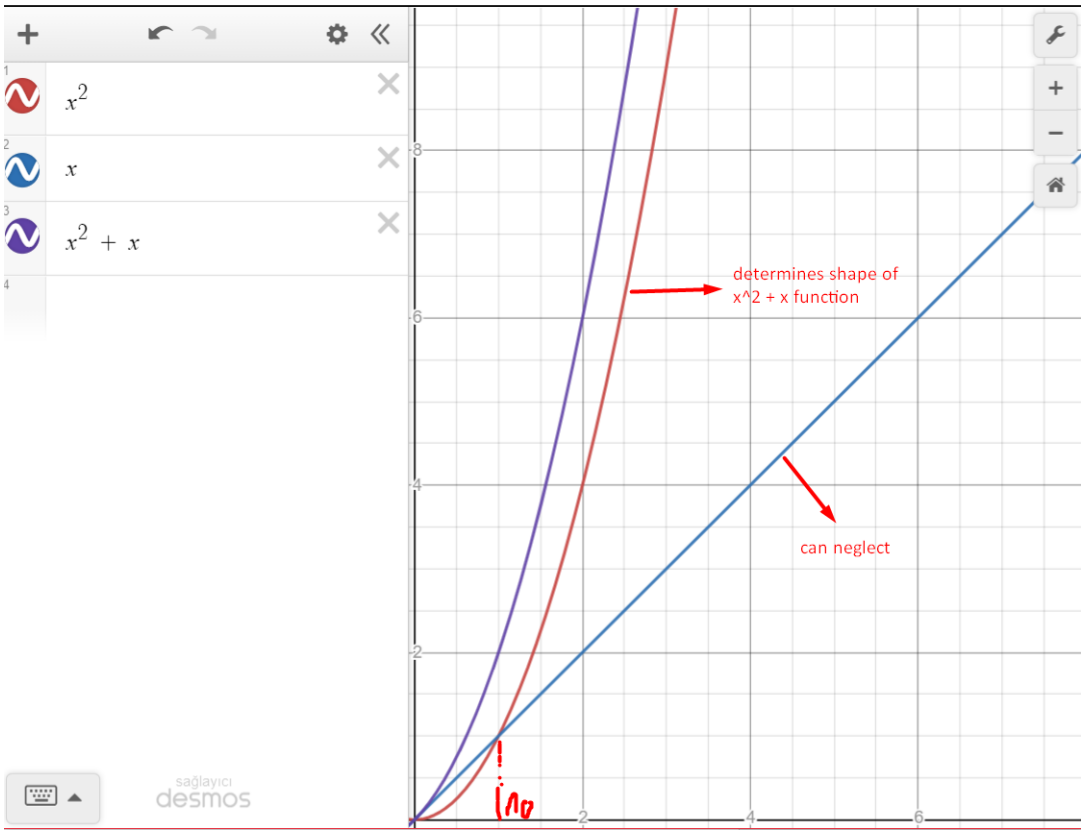
Big Oh notation definition: $T(n) \leq c \cdot f(n)$

it is the upper bound, it is meaningless to say at least

b) It's true because maximum function determines "Θ" notation's complexity, function with maximum degree determines complexity, lower degrees can be neglect because they don't have such an effect for graph.

For example: $f(n) = n^2$ $g(n) = n$

$\max(f(n), g(n)) = n^2 = \Theta(n^2+n) = \Theta(n^2)$



c)

I. $2^{n+1} = 2 \cdot 2^n$

can neglect this part for $n > n_0$

Upper bound and lower bound of this function is same

so: $2^{n+1} = O(2^n) = \Omega(2^n) \Rightarrow \Theta(2^n)$

II. $2^{2n} = (2 \cdot 2)^n = 2^n \cdot 2^n \Rightarrow$ Both of them increase by n value so we can't neglect any of them.

From definition:

If we accept $2^{2n} = \Theta(2^n)$ ✓

Then $c_1 \cdot 2^n \leq 2^{2n} \leq c_2 \cdot 2^n$ ✓

X $c_1 \leq 2^n \leq c_2 \Rightarrow c_1$ and c_2 are not constant in this case

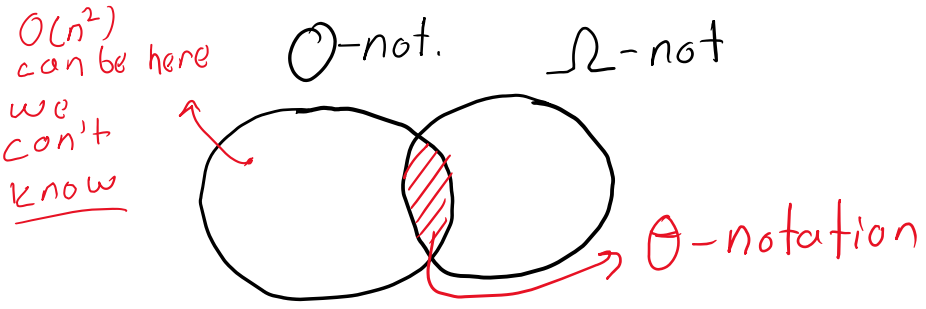
Should be constant

so $2^{2n} \neq \Theta(2^n)$.

III. $f(n) = O(n^2) \neq \Theta(n^2)$ because we cont know the lower bound (Ω-notation) of this function.

$g(n) = \Theta(n^2) \Rightarrow \Omega(n^2)$ and $O(n^2)$

Θ-notation is exact time complexity and holds Ω-notation and O-notation



So: $O(n^2) * \Theta(n^2) \neq \Theta(n^4)$

But we can multiply Big-Oh parts of f and g

$$O(n^2) * O(n^2) = O(n^4)$$

We can say product of them is $O(n^4)$

But we can't say it is $\Theta(n^4)$

Part 3.

(Exponential) > $(x^2, x^3, x^4 \ldots x^n)$ > (Linear) > (constant) > (Logarithm)

From this definition

I will make a guess from this definition

$n2^n$? 3^n > $2^{n+1} \geq 2^n > 5^{\log_2 n} > (n^{1.01} > \sqrt{n})$? $n \log^2 n > (\log n)^{\log n}$

$$\lim_{n\to\infty} \frac{3^n}{n \cdot 2^n} = \lim_{n\to\infty} \log\left(\frac{3^n}{n \cdot 2^n}\right) = \lim_{n\to\infty} (\log 3^n - \log n - \log 2^n)$$

$$= \lim_{n\to\infty} (\underbrace{n(\log 3 - \log 2)}_{\text{This part is bigger}} - \log n) = \infty$$

↳ So $3^n > n2^n$

$$\lim_{n\to\infty} \frac{n^{1.01}}{n\log^2 n} = \lim_{n\to\infty} \frac{(n^{0.01})'}{(\log^2 n)'} = \lim_{n\to\infty} \frac{n^{-0.99}\times 0.01}{2\log n \cdot \frac{1}{\ln(2)\, n}}$$

$$= \lim_{n\to\infty} \frac{n^{-0.99}}{\frac{\log n}{n}} = \lim_{n\to\infty} \frac{(n^{0.01})'}{(\log n)'} = \lim_{n\to\infty} \frac{n^{-0.99}}{\frac{1}{n}} = \lim_{n\to\infty} n^{0.01} = \infty$$

$$n^{1.01} > n\log^2 n$$

$$\lim_{n\to\infty} \frac{n^{0.5}}{n\log^2 n} = \lim_{n\to\infty} \frac{(n^{-0.5})'}{(\log^2 n)'} = \lim_{n\to\infty} \frac{-0.5\times n^{-1.5}}{2\log n \cdot \frac{1}{\ln(2)\, n}}$$

$$= \lim_{n\to\infty} \frac{(n^{-0.5})'}{(\log n)'} = \lim_{n\to\infty} \frac{-0.5\times n^{-1.5}}{\frac{1}{\ln(2)}\cdot\frac{1}{n}} = \lim_{n\to\infty} = \frac{1}{n^{0.5}} = 0$$

$$n^{1.01} > n\log^2 n > \sqrt{n}$$

Answer:

$$3^n > n2^n > 2^{n+1} > 2^n > 5^{\log_2 n} > n^{1.01} > n\log^2 n > \sqrt{n} > \log^3 n > \log n$$

Part 4:

I.

$T(n) = \Theta(n)$

```
findminval(arraylist):
    index = 0                                   Θ(1)
    min = arraylist.get(0)                      Θ(1)
    for i from 0 to arraylist.size():           → loop => Θ(n)
        if arraylist.get(i) < min:              Θ(1)
            index = i                           Θ(1)
            min = arraylist.get(i)              Θ(1)
    return arraylist.get(index)                 Θ(1)
```

Θ(1) } Θ(n * 1) = Θ(n) } Θ(n)

II.

$T(n) = \Theta(n^2)$

```
findmedian(arraylist):
    for i from 0 to arraylist.size():           Θ(n) because of return
        equal_count = 0                         Θ(1)
        less_count = 0                          Θ(1)
        for j from 0 to arraylist.size():       Θ(n)
            if arraylist.get(j) == arraylist.get(i) AND i!=j:   Θ(1)
                equal_count++                   Θ(1)
            else if arraylist.get(j) < arraylist.get(i):        Θ(1)
                less_count++                    Θ(1)

        if less_count <= arraylist.size() / 2 AND               Θ(1)
            arraylist.size() / 2 <= less_count + equal_count:   Θ(1)
                return arraylist.get(i)         Θ(1) => breaks loop
    return null                                 Θ(1)
```

Θ(n) } Θ(n²) } Θ(n²)

$T_b = \Theta(1)$

$T_w = \Theta(n^2)$ > $O(n^2)$

$T(n) = O(n^2)$

Best ← $T_b = \Theta(1)$

Worst ← $T_w = \Theta(n^2)$

III.

```
findItemsWithSameSum(arraylist, sum):
        ArrayList returnVal  Θ(1)
        for i from 0 to arraylist.size():  O(n) → because of return
            for j from i+1 to arraylist.size():  O(n) → because of return
                if arraylist.get(i) + arraylist.get(j) == sum:  O(1)
                    returnVal.add(arraylist.get(i))  O(1)
                    returnVal.add(arraylist.get(j))  O(1)
                    return returnVal  Θ(1)
        return null;  Θ(1)
```

$O(n-i-1) = O(n)$

Amortized constant

$O(n) \times O(n) = O(n^2)$

$O(n^2)$

IV.

size=n size=m $T(n,m) = \Theta(n+m)$

```
mergeLists(arraylist_1, arraylist_2):
    ArrayList returnVal Θ(1)
    index_1 = 0 Θ(1)
    index_2 = 0 Θ(1)
    while index_1 < arraylist_1.size() OR(||) index_2 < arraylist_2.size(): Θ(n+m)
        if index_1 >= arraylist_1.size(): Θ(1)
            returnVal.add(arraylist_2.get(index_2)) Θ(1)
            index_2++ Θ(1)
        else if index_2 >= arraylist_2.size(): Θ(1)
            returnVal.add(arraylist_1.get(index_1)) Θ(1)
            index_1++ Θ(1)
        else if arraylist_1.get(index_1) <= arraylist_2.get(index_2): Θ(1)
            returnVal.add(arraylist_1.get(index_1)) Θ(1)
            index_1++ Θ(1)
        else:
            returnVal.add(arraylist_2.get(index_2)) Θ(1)
            index_2++ Θ(1)

    return returnVal Θ(1)
```

Amortized Constant

$\Theta(n+m)$ $\Theta(n+m)$

Part 5:

a)

```
int p_1 (int array[]):
{
        return array[0] * array[2]
}
```

O(1) O(1)

O(1)

Time Comp. O(1)

Space Complexity O(1) => without counting input size

b)

```
int p_2 (int array[], int n):
{
        Int sum = 0
        for (int i = 0; i < n; i=i+5)
                sum += array[i] * array[i]
        return sum
}
```

Int sum = 0 O(1)

for (int i = 0; i < n; i=i+5) $O(\frac{n}{5}) = O(n)$

sum += array[i] * array[i] => O(1)

O(1) O(1)

return sum

O(1)

Time comp. O(n)

Space Complexity O(1) => without counting input size

c)

```
void p_3 (int array[], int n):
{
        for (int i = 0; i < n; i++)
                for (int j = 0; j < i; j=j*2)
                        printf("%d", array[i] * array[j])
}
```

for (int i = 0; i < n; i++) O(n)

for (int j = 0; j < i; j=j*2) O(logn)

printf("%d", array[i] * array[j])

O(1) O(1)

O(1)

Time Complexity O(nlogn)

Space Complexity O(1) => Without counting input size

will happen for both best and worst cases

d)

```
void p_4 (int array[], int n):
{
    If (p_2(array, n)) > 1000)
        p_3(array, n)
    else
        printf("%d", p_1(array) * p_2(array, n))
}
```

O(n)

Worst case

O(nlogn)

Best case

O(1) O(n)

O(1+n) = O(n)

Time Complexity

$T_b = O(n+n) = O(n)$

$T_w = O(n+n\log n) = O(n\log n)$

$\Rightarrow O(n\log n)$

$\Omega(n)$

Space Complexity

P_1 => O(1)

P_2 => O(1)

P_3 => O(1)

so P_4 => O(1)

without counting input size