PART I

```
@Override
public boolean searchProducts(String name,String model,String color) throws Exception{
    Product tempProduct = new Product(name,model,color,0); Θ(1)
T2  for (int i = 0; i < getCompany().getBranches().size(); i++) {
T1      for (int j = 0; j < getCompany().getBranches().get(i).getProducts().size(); j++) {
            if (getCompany().getBranches().get(i).getProducts().get(j).equals(tempProduct)) { → Θ(1)
                System.out.println("************************"); Θ(1)
                System.out.println(); Θ(1)
                System.out.println(getCompany().getBranches().get(i).getProducts().get(j)); Θ(1)
                System.out.println("************************"); Θ(1)
                return true; Θ(1)
            }
        }
    }
    throw new Exception();
}
```

$T_{2b}(n) = \Theta(1)$

$T_{2w}(n) = \Theta(n)$

$\Theta(1)$

$T_{1b}(m) = \Theta(1)$

$T_{1w}(m) = \Theta(n)$

$$T_b(n,m) = \Theta(1) + T_{1_b}(m) * T_{2_b}(n) = \Theta(1)$$
$$T_w(n,m) = \Theta(1) + T_{1_w}(m) * T_{2_w}(n) = \Theta(n*m)$$
$$T(n,m) = O(n*m)$$

I assume that some functions such as getters have Θ(1) time complexity so i did not consider them.

```
@Override
@SuppressWarnings("unchecked")
public boolean add(Object e) {
    int i; Θ(1)
    if (size>=capacity) { → Θ(1)
        capacity=capacity+1; Θ(1)
    }                                   Θ(1)
    if (data==null) { → Θ(1)
        data = (E[])new Object[this.capacity]; Θ(1)
    }                                   Θ(1)
    E[] tempArr = (E[])new Object[capacity]; Θ(1)
    for (i = 0; i < size; i++) {
        tempArr[i]=data[i]; Θ(1)
    }                                   Θ(n)
    tempArr[size]=(E)e; Θ(1)
    size=size+1; Θ(1)
    data=tempArr; Θ(1)
    return true; Θ(1)
}
```

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(n) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1)$$
$$= \Theta(n) \ \#$$

This is the function which is my container's add function. To investigate addProduct function's time complexity i have to calculate this function's complexity first.

```
@Override
public void addProduct(String name,String model,String color,int stock)throws Exception{
    Product tempProduct = new Product(name, model, color, stock); Θ(1)
    if (!searchProducts(name, model, color, stock)) { → O(m*n)                O(m*n)
        if (!branch.getProducts().add(tempProduct)) → Θ(n)     Θ(n)
            throw new Exception(); Θ(1)
        }
    }
}
```

$T(m,n) = \Theta(1) + O(m*n)$

$= O(m*n)$ #

In my design, i did not use container's remove function in the removeProduct function. Yet, i want to show you the complexity of my container's remove function. My remove function contains some helper functions, i calculated time complexity of these functions too.

```
public int findIndex(E e){
    for (int i = 0; i < size; i++) {
        if (data[i].equals(e)) → Θ(1)        Θ(1)
            return i; Θ(1)
        }
    }
    return -1;
}
```

$T_b(n) = \Theta(1)$

$T_w(n) = \Theta(n)$

$T(n) = O(n)$ #

```
@Override
@SuppressWarnings("unchecked")
public boolean remove(E e) {
    int index= findIndex(e); O(n)
    if (index!=-1) {
        if (size==1) {
            size=0;              Θ(1)
            return true;
        }else{
            E[] tempArr = (E[])new Object[size-1]; Θ(1)
            for (int i = 0,j=0; i < size; i++) {
                if (i!=index) {
                    tempArr[j++] = data[i];     Θ(1)     Θ(m)     Θ(m)
                }
            }
            data=tempArr; Θ(1)
            size=tempArr.length; Θ(1)
            return true; Θ(1)
        }
    }else
        return false; Θ(1)
}
```

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

#

```java
@Override
public boolean removeProduct(int wantedID,int quantity)throws Exception {
    for (int j = 0; j < branch.getProducts().size(); j++) {
        if (branch.getProducts().get(j).getId()==wantedID) {
            if (branch.getProducts().get(j).getStock()>=quantity) {
                branch.getProducts().get(j).decreaseStock(quantity);
                return true;
            }
        }
    }
    throw new Exception();
}
```

T_1 $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$

$T(n) = O(n)$ #

$T_{1b} = \Theta(1)$

$T_{1w} = \Theta(n)$

```java
@Override
public void queryProductState() throws Exception{
    if (getCompany().getBranches().size()==0) {
        throw new Exception();
    }
    for (int i = 0; i < getCompany().getBranches().size(); i++) {
        for (int j = 0; j < getCompany().getBranches().get(i).getNeedToBeSupplied().size();j++) {
            System.out.println(getCompany().getBranches().get(i).getNeedToBeSupplied().get(j));
        }
    }
}
```

$T(n,m) = T_1(m) * T_2(n) = O(n*m)$ #

$\Theta(1)$

T_2 T_1

$T_2(n) = \Theta(n)$

$T_1(m) = \Theta(m)$

PART II

a) Let f(n)=O(n^2), this means that f(n) could be any function smaller than n^2. Saying that The running time of algorithm A is at least O(n^2) is meaningless since O(n^2) represents upper bounds of f(n). The expression should have been "Running time for every algorithm is at least constant".

b) Let's assume that T1(N)=Θ(n^2) and T2(N)=Θ(n);
max(T1(N),T2(N))=Θ(n^2) and also T1(N)+T2(N)=Θ(n^2+n) = Θ(n^2) because of low-order terms are insignificant. So we can say that max(f (n), g(n)) = Θ(f(n) + g(n)).

c) I.

We should know that

$$\lim_{N\to\infty}\frac{f(N)}{g(N)} = 0 \quad \Rightarrow f(N) = o(g(N))$$
$$= c \neq 0 \quad \Rightarrow f(N) = \theta(g(N))$$
$$= \infty \quad \Rightarrow g(N) = o(f(N))$$
$$= \text{oscilate} \Rightarrow \text{there is no relation}$$

$$2^{n+1} = \Theta(2^n)$$ #

So, Let 2^(n+1) =f(N) and 2^n = g(N). Now, we can implement L'hospital rule.

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \frac{\cancel{\ln 2} \cdot \cancel{2}^{\cancel{n}+1}}{\cancel{\ln 2} \cdot \cancel{2^n}} = 2$$

II.

We should know that

$$\lim_{N\to\infty}\frac{f(N)}{g(N)} = 0 \quad \Rightarrow f(N) = o(g(N))$$
$$= c \neq 0 \quad \Rightarrow f(N) = \theta(g(N))$$
$$= \infty \quad \Rightarrow g(N) = o(f(N))$$
$$= \text{oscilate} \Rightarrow \text{there is no relation}$$

$$2^{2n} \neq \Theta(2^n)$$ #

So, Let f(N)=2^(2n) and
g(N)=2^n Now, we can implement L'hospital rule.

$$\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \frac{\cancel{\ln 2} \cdot 2^{\cancel{2}n}}{\cancel{\ln 2} \cdot \cancel{2^n}} = 2^n = \infty$$

III.

f(n)=O(n^2) represents the any function smaller than n^2,in other words it represents upper bounds. But g(n)=Θ(n^2) is certain and tight. So, multiplication of these functions cannot be certain function. Equation is wrong. It must be O(n^4).

PART III

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

It is enough to compare these functions with each other.

$$n^{1.01} > n^{0.5}$$

$$\log^3 n > \log n$$ by using the power rule

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \left(\frac{3}{2}\right)^n = \infty, \quad 3^n > 2^n = 2^{n+1}$$

$$\lim_{n\to\infty} \frac{2^n}{5^{\log_2 n}} = \infty, \quad 2^n > 5^{\log_2 n}$$

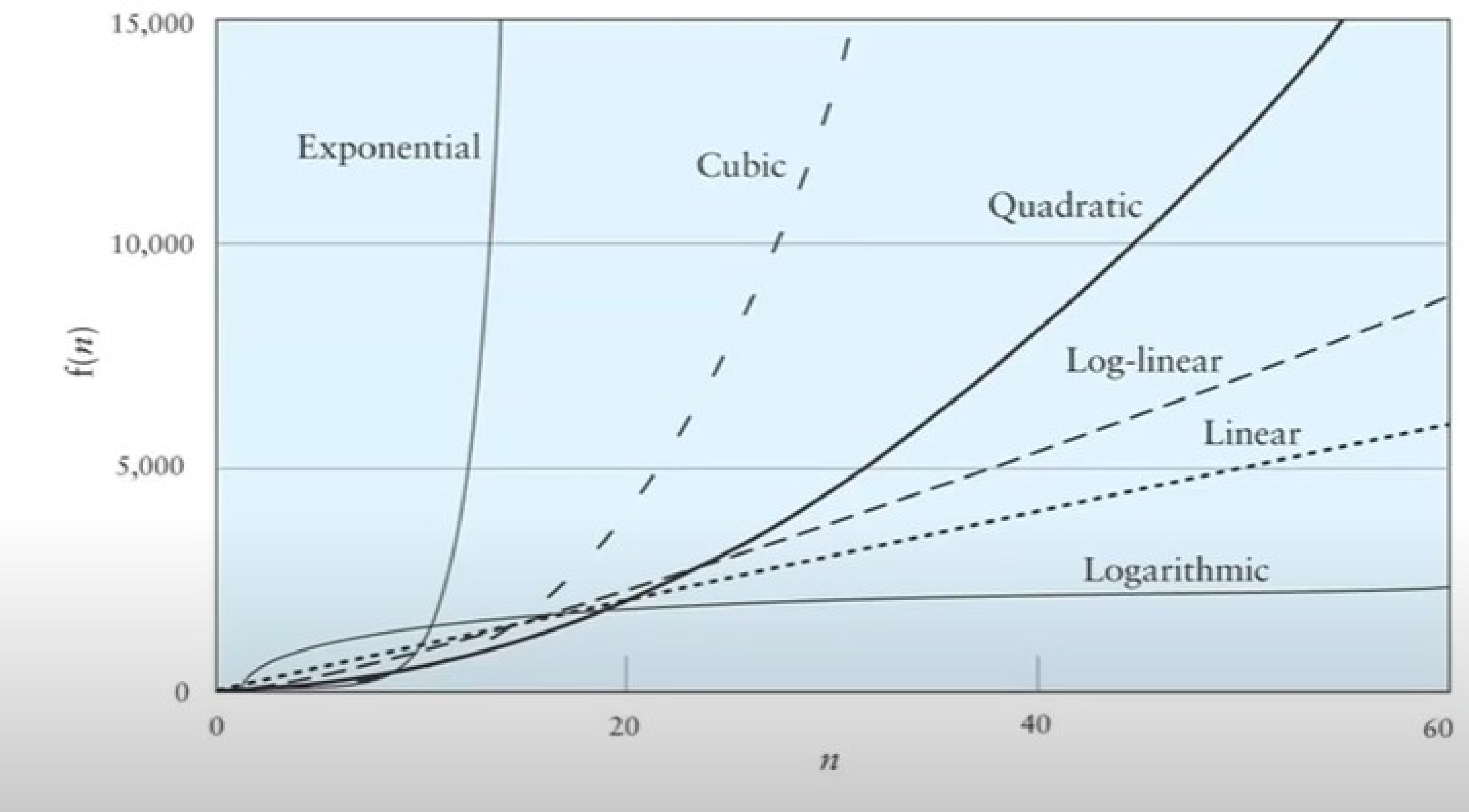
$$\lim_{n\to\infty} \frac{n.\cancel{2^n}}{\cancel{2^n}} = \infty, \quad n.2^n > 2^n$$

$$\lim_{n\to\infty} \frac{n^{1,01}}{n.\log^2 n} = \frac{\cancel{n}.n^{0,01}}{\cancel{n}.\log^2 n} = \infty$$

Polynomial function grows much more faster than logaritmic function.

$$\lim_{n\to\infty} \frac{n.\log^2 n}{\sqrt{n}} = \sqrt{n}.\log^2 n = \infty, \quad n\log^2 n > \sqrt{n}$$

According to the this graph;



Growth order will be;

$$3^n > n.2^n > 2^{n+1} = 2^n > 5^{\log_2 n} > n^{1,01} > n.\log^2 n > \sqrt{n} > \log^3 n > \log n$$

PART IV

Find the minimum-valued item.

```
min=arrayList[0]    Θ(1)
for i = 1 to arrayList.length()
    if arrayList[i] < min
        min = arrayList[i]    Θ(1)    Θ(1)    Θ(n)
    end-if
end-for
```

$$T(n) = \Theta(1) + \Theta(n) = \Theta(n) \quad \#$$

Find the median item. Consider each element one by one and check whether it is the median.

To find the median item we need some helper functions such as sort function.

```
SORT(arrayList[]):
for i = 0 to arrayList.length()
      for j = 0 to arrayList.length()
            if arrayList[j] > arrayList[j+1]
                  temp=arrayList[j]  Θ(1)
                  arrayList[j] = arrayList[j+1]  Θ(1)        Θ(1)      Θ(n)      Θ(n)
                  arrayList[j+1] = temp  Θ(1)
            end-if
      end-for
end-for
```

$T_{sort} = \Theta(n^2)$

```
FIND MEDIAN(arrayList[]):
sort(arrayList)  Θ(n²)
if arrayList.length() mod2 ==0
      return (arrayList[arrayList.length()/2] + arrayList[(arrayList.length()/2-1)]) / 2   Θ(1)
end-if
else
      return arrayList[arrayList.length()/2]   Θ(1)
end-else
```

$T_{median}(n) = \Theta(n^2)$

Find two elements whose sum is equal to a given value

```
Assume that N is a given value
for i = 0 to arrayList.length()
      for j = 0 to arrayList.length()
            if i != j and arrayList[i] + arrayList[j] == N
                  print "arrayList(i) + arrayList(j) = N"   Θ(1)      Θ(n)      Θ(n)
            end-if
      end-for
end-for
```

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

Assume there are two ordered array list of n elements. Merge these two lists to get a single list in increasing order.

```
MERGE(arr1[],arr2[]):
temp = new ArrayList()          Θ(1)
for i=0 to n
    temp.add(arr1[i])  Θ(n)      Θ(n²)
end-for
for i=0 to n
    temp.add(arr2[j])  Θ(n)      Θ(n²)
end-for
sort(temp)  Θ(n²)
```

$$T(n) = \Theta(1) + \Theta(n^2) + \Theta(n^2) + \Theta(n^2) = \Theta(n^2)$$ #

I have already calculated complexity of sort and add functions. I have directly used them.

PART V

a)

```
int p_1 (int array[]):
{
    return array[0] * array[2])   Θ(1)
}   Θ(1)
```

Time

$T(n) = \Theta(1)$

Space

$S(n) = \Theta(1)$

The amount of memory space needed the algorithm is tight, so space complexity is Θ(1) 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
}
```

Time | Space

$T(n) = \Theta(n)$ | $S(n) = \Theta(1)$

Int sum = 0: $\Theta(1)$ $\Theta(1)$

sum += array[i] * array[i]): $\Theta(1)$ $\Theta(1)$

for loop: $\Theta(\frac{n}{5})$ $\Theta(1)$

return sum: $\Theta(1)$ $\Theta(1)$

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])
}
```

Time | Space

$T(n) = \Theta(n.\log n)$ | $S(n) = \Theta(1)$

for (int i = 0; i < n; i++): $\Theta(n)$

for (int j = 0; j < i; j=j*2): $\Theta(\log n)$

printf("%d", array[i] * array[j]): $\Theta(1)$ $\Theta(1)$

$$T(n) = \Theta(n) * \Theta(\log n) = \Theta(n.\log n)$$

```
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))
}
```

Time

$T(n) = O(n\log n)$

Space

$S(n) = \Theta(1)$

$\Theta(n)$

$\Theta(n\log n)$

$\underbrace{p\_1(array)}_{\Theta(1)} * \underbrace{p\_2(array, n)}_{\Theta(n)}$

$\Theta(1) + \Theta(n)$

$T_b(n) = \Theta(n) + \Theta(n) + \Theta(1)$

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