1 Part 1

1.I Searching a product

```
public void searchProducts(String productName, int productModel, int productColor, int mode){
    int name;
    for (name = 0; name < productNames.length && !productName.equals(productNames[name]); name++) ; }O(n)
    if(mode == 2) { }Θ(1)
Θ(n) {  for (int i = 0, stock; i < branches.length; i++) { -> Θ(n)
            stock = branches[i].getStockInfo(name, productModel, productColor);}Θ(1)
            System.out.println(branches[i].name + " has " + stock + " of chosen product in its stocks");}Θ(1)  }Θ(1)
        }
    }
    else{
Θ(1) {  System.out.println("Online stock of chosen product is " + StoreHouse.getStockInfo(name, productModel, productColor));
    }
}
```

```
public int getStockInfo(int productName, int productModel, int productColor){
    return allProducts[productName].stockInfo(productModel, productColor);}Θ(1)
}
```

```
public int stockInfo(int model, int color) {
    return models[model][color]; }Θ(1)
}
```

Lets assume that searchProducts(String productName, int productModel, int productColor, int mode) $= T(n)$
$T(n) = O(n) + \Theta(1) + max(\Theta(n), \Theta(1)) \implies O(n) + \Theta(1) + \Theta(n) = O(n)$

1.II Add product

```
public void addProduct(String name, int model, int color, int num){
    int index = getNameIndex(name);} O(n)
    int initialStock = Branch.getStockInfo(index, model, color);}Θ(1)
Θ(1){Branch.changeStock(index, model, color, num: initialStock + num);
}
```

```
public int getNameIndex(String productName){
    int index;
O(n){for(index = 0; index < productNames.length && !productName.equals(productNames[index]); index++);  } O(n)
    return index;}Θ(1)
}
```

```
public int getStockInfo(int productName, int productModel, int productColor){
    return allProducts[productName].stockInfo(productModel, productColor);}Θ(1)
}
```

```
public int stockInfo(int model, int color) {
    return models[model][color]; }Θ(1)
}
```

```
public void changeStock(int productName, int productModel, int productColor, int num){
    allProducts[productName].changeStock(productModel, productColor, num); }Θ(1)
}
```

```
public void changeStock(int model, int color, int stock) {
    models[model][color] = stock;}Θ(1)
}
```

Lets assume that addProduct(String name, int model, int color, int num) $= T(n)$
$T(n) = O(n) + \Theta(1) + \Theta(1) = O(n)$

1.III Querying the products that need to be supplied

```java
public boolean queryStocks(){
    int initialStock;
    int counter = 0; Θ(1)
    for(int i = 0; i < modelNums.length; i++){ → Θ(n)
        for(int j = 0; j < modelNums[i]; j++){ → Θ(m)
            for(int k = 0; k < colorNums[i]; k++){ → Θ(k)
                initialStock = StoreHouse.getStockInfo(i, j, k); Θ(1)
                if(initialStock < 50){ Θ(1)
                    System.out.println(productNames[i] + " Model" + (j + 1) + " Color" + (k + 1) +
                            " had a stock less than 50 and resupplied to " + (initialStock + 50)); Θ(1)
                    StoreHouse.changeStock(i, j, k, num: initialStock + 50); Θ(1)
                    counter++; Θ(1)
                }
            }
        }
    }
    if(counter > 0){ Θ(1)
        return true; Θ(1)
    }
    return false; Θ(1)
}
```

Θ(1) Θ(k) Θ(m.k) Θ(n.m.k)

```java
public int getStockInfo(int productName, int productModel, int productColor){
    return allProducts[productName].stockInfo(productModel, productColor); Θ(1)
}
```

```java
public void changeStock(int productName, int productModel, int productColor, int num){
    allProducts[productName].changeStock(productModel, productColor, num); Θ(1)
}
```

```java
public void changeStock(int model, int color, int stock) {
    models[model][color] = stock; Θ(1)
}
```

Lets assume that queryStocks() $= T(m,n,k)$
$T(m,n,k) = \Theta(1) + \Theta(m.n.k) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(m.n.k)$

2 Part 2

2.a

The running time of algorithm A is at least O(n^2) is meaningless to say because Big O notation provides an upper bound for the running time function of algorithm A, not a lower bound.

2.b

$max(f(n), g(n)) = \Theta(f(n) + g(n))$ to prove this equation we must obtain $O(f(n) + g(n))$ and $\Omega(f(n) + g(n))$

$$\left.\begin{aligned} max(f(n), g(n)) \geq f(n) \\ max(f(n), g(n)) \geq g(n) \end{aligned}\right\} \quad 2max(f(n), g(n)) \geq f(n) + g(n) \implies \underbrace{max(f(n), g(n))}_{\text{T(n)}} \geq \underbrace{\frac{1}{2}}_{\text{c}} \underbrace{[f(n) + g(n)]}_{\text{H(n)}}$$

$\implies max(f(n), g(n)) = \Omega(f(n) + g(n))$

$$\left.\begin{aligned} f(n) \leq f(n) + g(n) \\ g(n) \leq f(n) + g(n) \end{aligned}\right\} \quad max(f(n), g(n)) \leq f(n) + g(n) \implies \underbrace{max(f(n), g(n))}_{\text{T(n)}} \leq \underbrace{1}_{\text{c}} \underbrace{[f(n) + g(n)]}_{\text{H(n)}}$$

$\implies max(f(n), g(n)) = O(f(n) + g(n))$

$max(f(n), g(n)) = \Theta(f(n) + g(n))$ if and only if $[max(f(n), g(n)) = O(f(n) + g(n))]$ and $[max(f(n), g(n)) = \Omega(f(n) + g(n))]$, therefore $max(f(n), g(n)) = \Theta(f(n) + g(n))$

2.c

2.a.I $2^{n+1} = \Theta(2^n)$

$$\left.\begin{array}{l} 2.2^n \leq c.2^n \quad n \geq n_0 \\ c=3 \quad n_0 = 1 \end{array}\right\} \quad 2^{n+1} = O(2^n) \qquad\qquad \left.\begin{array}{l} 2.2^n \geq c.2^n \quad n \geq n_0 \\ c=1 \quad n_0 = 2 \end{array}\right\} \quad 2^{n+1} = \Omega(2^n)$$

Since we obtained both $2^{n+1} = O(2^n)$ and $2^{n+1} = \Omega(2^n)$, we can say that $2^{n+1} = \Theta(2^n)$

2.b.II $2^{2n} = \Theta(2^n)$

$2^{2n} \leq c.2^n \Longrightarrow \frac{2^{2n}}{2^n} \leq \frac{c.2^n}{2^n} \Longrightarrow 2^n \leq c \Longrightarrow n \leq \log_2 c$
We cannot determine a n_0 that satisfies the equation for all $n \geq n_0$ because n will always be less than or equal to $log_2 c$. So we cannot obtain Big O notation and therefore we cannot obtain Theta notation.

2.c.III Let $f(n) = O(n^2)$ and $g(n) = \Theta(n^2)$. Prove or disprove that: $f(n) * g(n) = \Theta(n^4)$

$$\left.\begin{array}{r} f(n) \leq c.n^2 \\ c_1.n^2 \leq g(n) \leq c_2.n^2 \end{array}\right\} \quad f(n) * g(n) \leq c_0.n^4$$

We cannot estimate anything about left bound thus $f(n) * g(n) = O(n^4)$. e.g. Lets assume that $g(n) = n^2$ and $f(n) = n$. $f(n) * g(n)$ would be $\Theta(n^3)$. As you can see f(n)'s degree can be any number in between 0(i.e. constant) and 2 and therefore we cannot use Theta notation so we use Big O notation.

3 Part 3

$\log n < (\log n)^3 < \sqrt{n} < n.log^2 n < n^{1.01} < 5^{\log_2 n} < 2^n = 2^{n+1} < n.2^n < 3^n$

$$\lim_{n\to\infty} \left(\frac{(\log n)^3}{\log n}\right) \Rightarrow \underbrace{\lim_{n\to\infty} \left((\log n)^2\right)}_{\infty} = \infty \Longrightarrow \log n = o((\log n)^3$$

$$\left.\begin{array}{l} \sqrt{n} = n^{0.5} \quad (n^{0.5})^2 \Longrightarrow n \\ ((\log n)^3)^2 \Longrightarrow (\log n)^6 \Longrightarrow \log^6 n \end{array}\right\} \quad \log^k n = o(n) \quad \text{for any constant k} \Longrightarrow (\log n)^3 = o(\sqrt{n})$$

$$\lim_{n\to\infty} \left(\frac{n.\log^2 n}{\sqrt{n}}\right) \Rightarrow \lim_{n\to\infty} \left(\sqrt{n}.\log^2 n\right) \Rightarrow \underbrace{\lim_{n\to\infty} \left(\sqrt{n}\right)}_{\infty} . \underbrace{\lim_{n\to\infty} \left(\log^2 n\right)}_{\infty} = \infty \Longrightarrow \sqrt{n} = o(n.\log^2 n)$$

$5^{\log_2 n} = n^{\log_2 5} > n^2$ because $\log_2 5$ is bigger than 2.
Order of growth of functions that are in n^k format where k is a positive constant increases when k is increased.
$\sqrt{n} = o(n^{1.01})$, $\quad n^{1.01} = o(5^{\log_2 n})$

$5^{\log_2 n} = o(2^n)$ because $n^3 = o(2^n)$ and $5^{\log_2 n}$ is between quadratic functions and cubic functions.
$2^{n+1} = \Theta(2^n)$ because $2^{n+1} = 2.2^n$ and constants doesn't affect the order of growth.

$$\lim_{n\to\infty} \left(\frac{n.2^n}{2^n}\right) \Rightarrow \underbrace{\lim_{n\to\infty} (n)}_{\infty} \Longrightarrow 2^n = o(n.2^n)$$

Order of growth of exponential functions increases when base number is increased.
$2^n = o(3^n)$

4 Part 4

4.1 Find the minimum-valued item

```
min ← arraylist.get(0) } θ(1)
for i:=0 to n do } θ(n)
    if min > arraylist.get(i) then } θ(1)        } θ(n)   } θ(n)
        min ← arraylist.get(i) } θ(1)
    end if
end for
```

4.2 Find the median item

```
class counter
    index
    number

count ← 0 } θ(1)
counter[ ] ordered ← new counter[n] } θ(n)
for i:=0 to n do } θ(n)
    ordered[i] ← new counter() } θ(1)   } θ(n)
end for
for i:=0 to n do } θ(n)
    for j:=0 to n do } θ(n)
θ(n) {  if arraylist[i] > arraylist[j] then } θ(1)
            count ← count + 1 } θ(1)
        end if                                  } θ(n)   } θ(n^2)
    end for
    ordered[i].index ← i } θ(1)
    ordered[i].number ← count } θ(1)
    count ← 0 } θ(1)                                                } Θ(n^2)
end for
for i:=0 to n do } θ(n)
    for j:=i+1 to n do } θ(n)
        if ordered[i].number > ordered[j].number then } θ(1)
            temp ← ordered[i] } θ(1)                                } θ(n^2)
θ(n) {      ordered[i] ← ordered[j] } θ(1)
            ordered[j] ← temp } θ(1)
        end if
    end for
end for
if n%2 == 0 then } θ(1)
θ(1) { median ← (double) (arraylist[ordered[n/2].index] + arraylist[ordered[n/2 - 1].index])/2
else                                                                } θ(1)
θ(1) { median ← arraylist[ordered[(int)Math.floor(n/2)].index]
end if
```

4.3 Find two elements whose sum is equal to a given value

```
check ← 0 } θ(1)
for i:=0 to n do } θ(n)
    for j:=0 to n do } O(n)
        if arraylist.get(i) + arraylist.get(j) == given value and i ! = j then } θ(1)
            element1 ← array.get(i) } θ(1)
            element2 ← array.get(j) } θ(1)                      } O(n^2)   } O(n^2)
O(n) {      check ← 1 } θ(1)
            break } θ(1)
        end if
    end for
end for
if check = 0 then } θ(1)
    print error } θ(1)
end if
```

4.4 Merge two ordered array lists of n elements to get a single list in increasing order

```
orderedList ← new ArrayList<>() }θ(1)
j ← 0 }θ(1)
k ← 0 }θ(1)
for i:=0 to 2n do }θ(n)
  if i == 0 then }θ(1)
    if arraylist1.get(0) < arraylist2.get(0) then }θ(1)
      orderedList.add(arraylist1.get(0)) }θ(1)
      if j + 1 != n then }θ(1)
        j ← j + 1 }θ(1)
      end if
    else
      orderedList.add(arraylist2.get(0)); }θ(1)
      if k + 1 != n then }θ(1)
        k ← k + 1 }θ(1)
      end if
    end if
  else
    lastElement ← orderedList.get(orderedList.size() - 1) }θ(1)
    if lastElement < arraylist1.get(j) && lastElement ¡ arraylist2.get(k) then }θ(1)
      if arraylist1.get(j) < arraylist2.get(k) then }θ(1)
        orderedList.add(arraylist1.get(j)) }θ(1)
        if j + 1 != n then }θ(1)
          j ← j + 1 }θ(1)
        end if
      else
        orderedList.add(arraylist2.get(k)) }θ(1)
        if k + 1 != n then }θ(1)
          k ← k + 1 }θ(1)
        end if
      end if
    else if lastElement ¡ arraylist1.get(j) then }θ(1)
      orderedList.add(arraylist1.get(j)) }θ(1)
      if j + 1 != n then }θ(1)
        j ← j + 1 }θ(1)
      end if
    else if lastElement ¡ arraylist2.get(k) then }θ(1)
      orderedList.add(arraylist2.get(k)) }θ(1)
      if k + 1 != n then }θ(1)
        k ← k + 1 }θ(1)
      end if
    end if
  end if
end for
```

θ(n) θ(n)

5 Part 5

5.a

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

Space complexity is O(1) because there is no additional memory allocation that depends on some variable n

5.b

```
int p_2 (int array[], int n):
{
        Int sum = 0 } θ(1)
θ(n) {  for (int i = 0; i < n; i=i+5) } θ(n)
                sum += array[i] * array[i]) } θ(1)      } θ(n)
        return sum } θ(1)
}
```

Space complexity is O(1) because there is no additional memory allocation that depends on some variable n

5.c

```
void p_3 (int array[], int n):
{
Θ(n.log n) {  for (int i = 0; i < n; i++) } θ(n)
  Θ(log n) {      for (int j = 1; j < i; j=j*2) } Θ(log n)
                      printf("%d", array[i] * array[j]) } θ(1)
}
```

I started j from 1 as mentioned in Q&A forum
Space complexity is O(1) because there is no additional memory allocation that depends on some variable n

5.d

```
void p_4 (int array[], int n):
{
        If (p_2(array, n)) > 1000) } θ(n)
                p_3(array, n) } Θ(n.log n)                    } O(n.log n)
        else
   θ(n) { printf("%d", p_1(array) * p_2(array, n))
                       θ(1)          θ(n)
}
```

Space complexity is O(1) because there is no additional memory allocation that depends on some variable n