Part 1:

(1) searching a product

```
public List<Product> search (String productName){
    ArrayList<Product> temp = new ArrayList<Product>();
    for(int i=0 ; i < Admin.branches.size() ; i++){
        for (int j=0; j< Admin.branches.get(i).getProducts().size();j++){
            if(Admin.branches.get(i).getProducts.get(j).getProductName().
                equals(productName)){
                    temp.add (Admin.branches.get(i).getproducts().get(j));
                }
        }}
    return temp;
```

The add and get method of Arraylist has a time complexity of O(1). So we can ignore their use in the function as they wont effect. All the getters also has a constant time complexity. So as they wont effect the whole time complexity we ignore them too.

Then we have a for loop which will run upto Admin.branches size(). Let's denote it by $(n+1)$ including the false check.

Then we have a innerloop. That will run for $(m+1)$ times in worst case.

The if check will not hault the loop execution as it will just add a product in temp variable.

After eliminating the constant and other unnecessary parts we get,

$(m+1)(n+1)$ or, $mn + m + n + 1$ in best and worst case scenario. So we can say, time complexity is,

O $(m * n)$ where $m \neq n$.

The complexity is in O (Big-oh) notation because the average and best case scenario will be same as O(m*n) for this algorithm.

(11) Add/Remove Product

```
public void increaseAmount(int amount){
    this.amountAvailable += amount;
}
```

- this function has time complexity O(1).

```
public void decreaseAmount(int amount){
    this.amountAvailable -= amount;
}
```

- this function also has constant time complexity. O(1).

```
Public boolean addProduct(int ProductId, int amountToAdd){
  for(int i=0; i<this.assignedBranch.getProducts().size(); i++){
  if(this.assignedBranch.getProducts.get(i).getProductId()==ProductId){
    this.assignedBranch.getProducts.get(i).increaseAmount(amountToAdd);
    return true;
  }
  }
  return false;
}
```

The inner function all has O(1) complexity. So we can ignore them. for this function the for loop has a if statement inside it.

In best case if the product is found on first iteration then we simply increas it and return. All the operations have constant Time Complexity. So for best case scenario time complexity is, $\Omega(1)$.

In worst case the loop will run for (n+1) times. So worst case complexity is O(n).

The average case complexity : $\frac{\Sigma \text{ all possibilities}}{\text{total no. of possibilites}}$

here, for first input we have 1 possibility,
for second input we have 2 possibility,
for n input we have n possibities,
if we don't find then we have n possibities to compare,

And the total possibilities is $n+1$.

So, $\frac{\Sigma\ 1+2+3+n+n}{n+1}$

$= \frac{\frac{n(n+1)}{2}+n}{n+1}$

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

$= \frac{n^2+n+2n}{2n+1}$

$= \frac{n^2}{n}$ [After removing the parts that won't effect the time complexity]

$= n$

So, average complexity is $\Theta(n)$.

```
public boolean removeProduct(int ProdId, int amount){
for(int i=0; i< this.assignedBranch.getProducts().size(); i++){
if(this.assignedBranch.getProducts.get(i).getProductId()==ProductId){
  this.assignedBranch.getProducts.get(i).decreaseAmount(amount);
  return true;
}
}
return false;
}
```

This function has same complexity as Add Product method. So,

Best case : $\Omega(1)$

Worst Case = $O(n)$

Average Case : $\Theta(n)$.

(iii) query product

```
public Product queryProduct (int ProductId) {
    for(int i=0; i< Branch.getproduct().size(); i++){
        if(Branch.getProduct().get(i).getProductId() == ProductId){
        return Branch.getProduct.get(i);
    }
  }
  return null;
}
```

All the getters has constant time complexity so they can be ignored.

Now. we have a for loop with a if statement So, it has the same time complexity and explanation like add/remove product method.

Best Case : $\Omega(1)$ we find the product in first iteration.

Worst Case : $O(n)$ $(n+1)$ number of comparison.

Average case : $\Theta(n)$

Let's assume $f(n) = \Omega(n^2)$ and the problem says the running time of $f(n)$ is at least $\Omega(n^2)$. Now as the running time is upper bound so all the smaller values which are smaller than $\Omega(n^2)$ are also the running time of the function. So that included $O(1)$ or constant. Thats why it doesn't make any sense as all the running time in at least constant as it is the base.

(b) if $f(n) > g(n)$ then $f(n) \leq f(n) + g(n)$

and if $g(n) > f(n)$ then $g(n) \leq f(n) + g(n)$

So, $\max(f(n), g(n)) \in O(f(n) + g(n))$

Now say, $f(n) \geq g(n)$

if we add $f(n)$ on both side,

$$2f(n) \geq f(n) + g(n)$$

So we can deduct,

$$f(n) + g(n) \leq 2 \max(f(n), g(n))$$

So, $\max(f(n), g(n)) \in \Omega(f(n) + g(n))$

So, as it's true for upper and lower bound, we can say,

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

(c)

(I) Let's say $f(n) = 2^{n+1}$ and time complexity of this function $O(2^n)$.

From definition of Big-oh (O) notation we know that,

$O(g(n)) = \{ f(n)$: there exists some constants c and n_0 such that, $f(n) \leq c\, g(n)$ for $n \geq n_0 \}$

Now let's assume $n_0 = 0$ and $c = 2$ such that

$$2^{n+1} \leq c 2^n \quad \text{for } n \geq n_0$$

so, by definition $2^{n+1} = O(2^n)$.

(II) According to ~~definition~~ we need,

$$2^{2n} \leq c 2^n \quad \text{for all } n \geq n_0$$

or, $4^n \leq c 2^n$ for all $n \geq n_0$

As there is no such constant that satisfy the inequality we can say, $2^{2n} \neq O(2^n)$

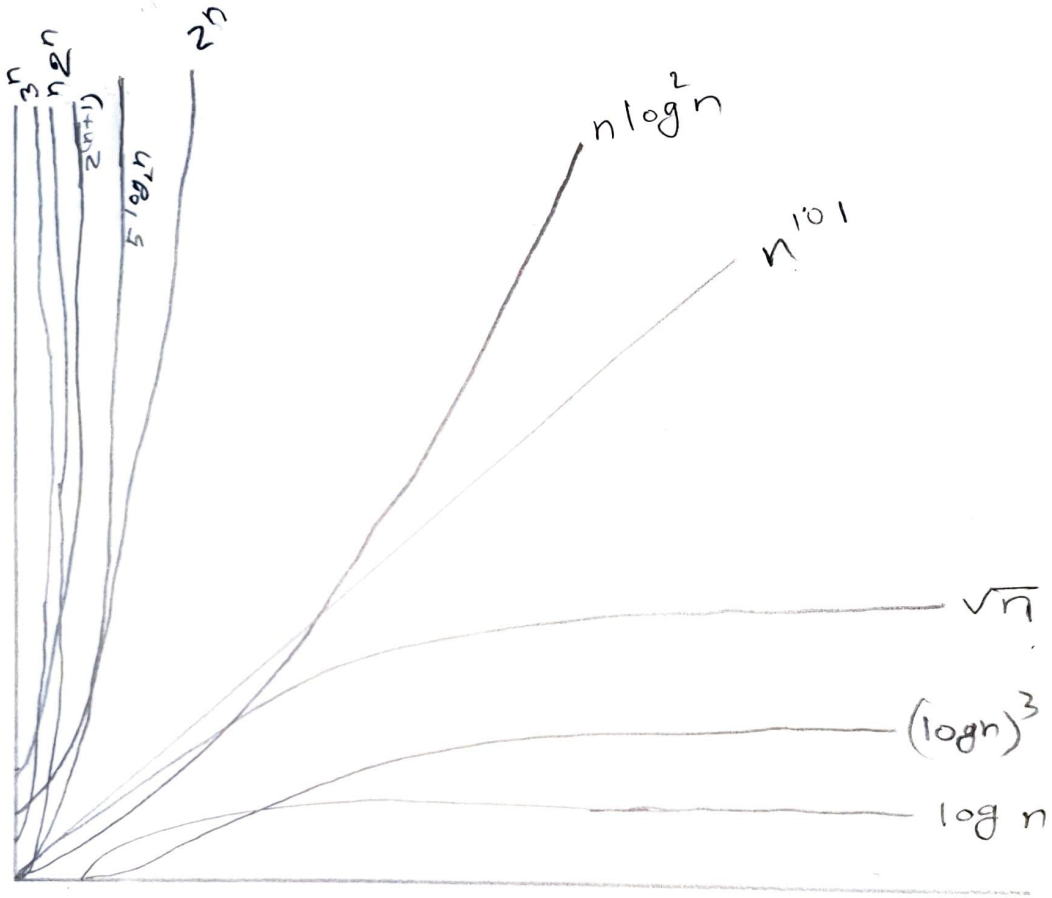
(III) we know that, if $T_1(N) = O(f(n))$ & $T_2(N) = O(g(n))$ then, $T_1(N) * T_2(N) = O(f(N) * g(N))$

Now, $f(n) = O(n^2)$ and $g(n) = \Theta(n^2)$

Now we know $f(n)$ has a upper bound of $O(n^2)$ but that doesn't imply about

lower bound of $f(n)$. For $\Theta(f(n))$ we need to know the strict bound of the function. So we can deduct $f(n) = O(n^4)$ and $g(n) = \Theta(n^2)$, in not same as $f(n) * g(n) = \Theta(n^4)$.

(3) we plot the graphs of the functions,



From the graph if we order them according to growth,

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

worst ⟶ best

(4)

* minimum_value : array.get(0);

while (array has element)

if element < minimum_value
then minimum_value = element

end while

- Now for the above pseudo-code we have n+1 number of comparison in every case.

So, best case is $O(n)$ and worst case is $\Omega(n)$

That's why $\Theta(n)$ will be the best representation of time complexity for this pseudocode.

* median item

minimum = array.get(0);
while (array has element)
if element < minimum
then minimum = element

end while

if (array.size() % 2 != 0)
then median = array.get (array.size() / 2);

else
median : (array.get (array.size() / 2) + array.get(
(array.size() - 1) / 2)) / 2;

The sorting part of this pseudocode has a complexity of $\Theta(n)$. The other parts are of constant time complexity. So the time

Complexity for finding median in $\Theta(n)$.

* two elements equal to a given value.

```
for (i = 0 to array.size())
    for (j = i+1 to arraysize())
        if (array.get(i) + array.get(j) == given value)
            print (i and j)
    end for
end for
```

It has two nested loop which will run for all the elements.

The time complexity for this algorithm is $O(n^2)$.

* merge array

```
i = 0 , j = 0 , r = 0
while (i < n and j < n)
    if (array-1.get(i) < array-2.get(j))
        array-3.add(array-one.get(i++))
    else
        array-3.add(array-two.get(j++))
end while
while (i < n)
    array-3.add(array-1.get(i++))
end while
while (j < n)
    array-3.add(array-1.get(j++))
end while
```

As the while loops will run for n times for all possible scenerios,

Time complexity is $O(n)$.

(5)

(a)

As the code has only one return statement which runs on constant time, the time complexity is $O(1)$.

As it's not using any extra memories. Space Complexity is $O(1)$.

(b)

For this function the statement inside the for loop going to run for $n/5$ times, if we ignore the constants we get a time complexity $O(n)$.

As the program is reusing memory and no extra memory is used. Space complexity for this program is $O(1)$.

(c) If $j = 0$ then we will have infinite loop so we took $j = 1$.

Now, for the inner loop print statement, it will run for,

$$\log n + 2\log n + 3\log n + \ldots + n\log n.$$

$$\Rightarrow \log n \left(\frac{n(n+1)}{2}\right)$$

If we remove the constants we get a

(c)

time complexity of $n^2 \log n$.

As no extra memory is used the space complexity is $O(1)$.

(d) we have two other function P_2 and P_3, called inside P_4 function. Let's say time complexity for P_2 is $O(P_2(n))$ and for P_3 $O(P_3(n))$. If they are of constant time complexity then whole program will have a constant time complexity $O(1)$. This is in case 1.

In Case 2, P_2 has constant time complexity and P_3 don't. Then we will have worst time complexity $O(P_3(n))$. And best is $O(1)$.

In case 3, P_2 is not constant but P_3 is. Then we will have worst time complexity $O(P_2(n))$. And Best is also $O(P_2(n))$.

If both of them aren't constant then we will have a worst case complexity $O(P_2(n) * P_3(n))$ and best case scenario will be $O(P_2(n))$.

The space complexity will be $O(1)$.