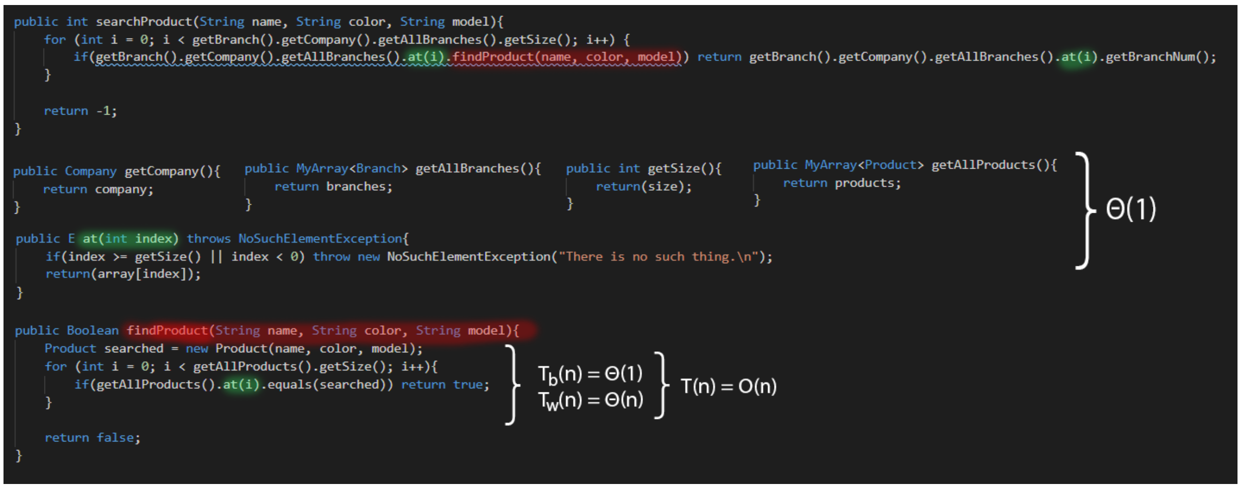
GTU Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework 2

Part 1:

I colorized methods in the other method to make it easier to follow as you can see in the screen shots.

I. Searching a product.



```java
public int searchProduct(String name, String color, String model){
    for (int i = 0; i < getBranch().getCompany().getAllBranches().getSize(); i++) {
        if(getBranch().getCompany().getAllBranches().at(i).findProduct(name, color, model)) return getBranch().getCompany().getAllBranches().at(i).getBranchNum();
    }

    return -1;
}

public Company getCompany(){          public MyArray<Branch> getAllBranches(){     public int getSize(){      public MyArray<Product> getAllProducts(){
    return company;                       return branches;                         return(size);              return products;
}                                     }                                        }                          }

public E at(int index) throws NoSuchElementException{
    if(index >= getSize() || index < 0) throw new NoSuchElementException("There is no such thing.\n");
    return(array[index]);
}

public Boolean findProduct(String name, String color, String model){
    Product searched = new Product(name, color, model);
    for (int i = 0; i < getAllProducts().getSize(); i++){
        if(getAllProducts().at(i).equals(searched)) return true;
    }

    return false;
}
```

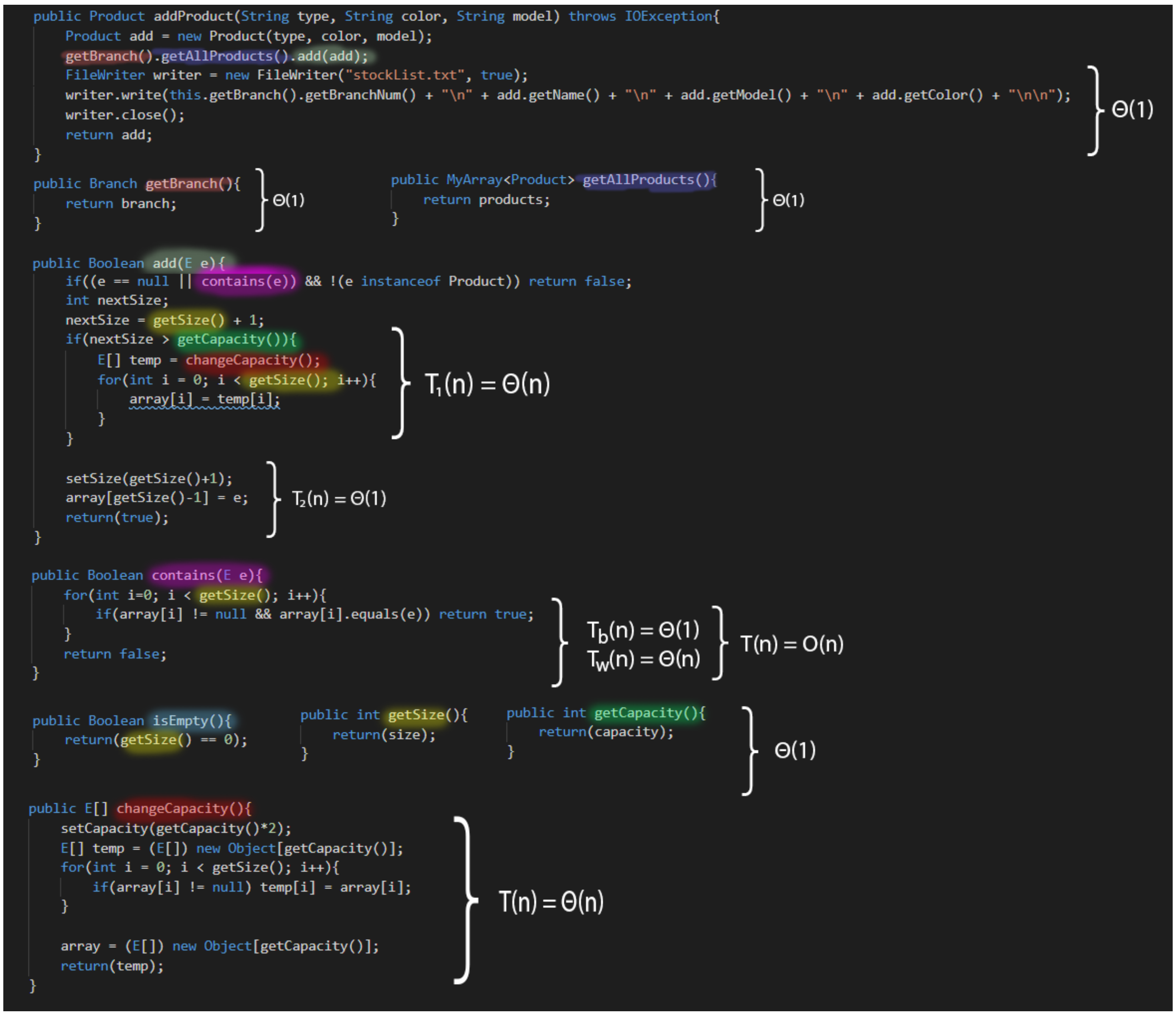
In searchProduct method, I used findProduct method of the branch. It basically takes all the branches and search for product in them. Let m be the number of the branches and n be the number of products in the branches.

In best case, product can be found in the first branch's first product and method can return the branch num of the branch and it takes constant time **θ(1).**

We must analyze the worst case now.
In findProduct method, we can't find the product in the i^{th} branch and loop n times. Time complexity is **θ(n) in this case.**
In searchProduct method, we can't find the product in any branches and loop m times. Time complexity is **θ(m) in this case.**

Since they are nested loops, **$T_{worst}(m,n) = \theta(m*n)$ and $T_{best}(m,n) = \theta(1)$.**
We can also say $T(m,n) = O(mn)$.

II. Add/remove product.



```java
public Product addProduct(String type, String color, String model) throws IOException{
    Product add = new Product(type, color, model);
    getBranch().getAllProducts().add(add);
    FileWriter writer = new FileWriter("stockList.txt", true);
    writer.write(this.getBranch().getBranchNum() + "\n" + add.getName() + "\n" + add.getModel() + "\n" + add.getColor() + "\n\n");
    writer.close();
    return add;
}

public Branch getBranch(){
    return branch;
}

public MyArray<Product> getAllProducts(){
    return products;
}

public Boolean add(E e){
    if((e == null || contains(e)) && !(e instanceof Product)) return false;
    int nextSize;
    nextSize = getSize() + 1;
    if(nextSize > getCapacity()){
        E[] temp = changeCapacity();
        for(int i = 0; i < getSize(); i++){
            array[i] = temp[i];
        }
    }

    setSize(getSize()+1);
    array[getSize()-1] = e;
    return(true);
}

public Boolean contains(E e){
    for(int i=0; i < getSize(); i++){
        if(array[i] != null && array[i].equals(e)) return true;
    }
    return false;
}

public Boolean isEmpty(){
    return(getSize() == 0);
}

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

public int getCapacity(){
    return(capacity);
}

public E[] changeCapacity(){
    setCapacity(getCapacity()*2);
    E[] temp = (E[]) new Object[getCapacity()];
    for(int i = 0; i < getSize(); i++){
        if(array[i] != null) temp[i] = array[i];
    }

    array = (E[]) new Object[getCapacity()];
    return(temp);
}
```

In my addProduct method “getBranch().getAllProducts().add(add);” line does the adding operation. Other lines are for keeping data in a file and their time complexity is **θ(1), constant time.**

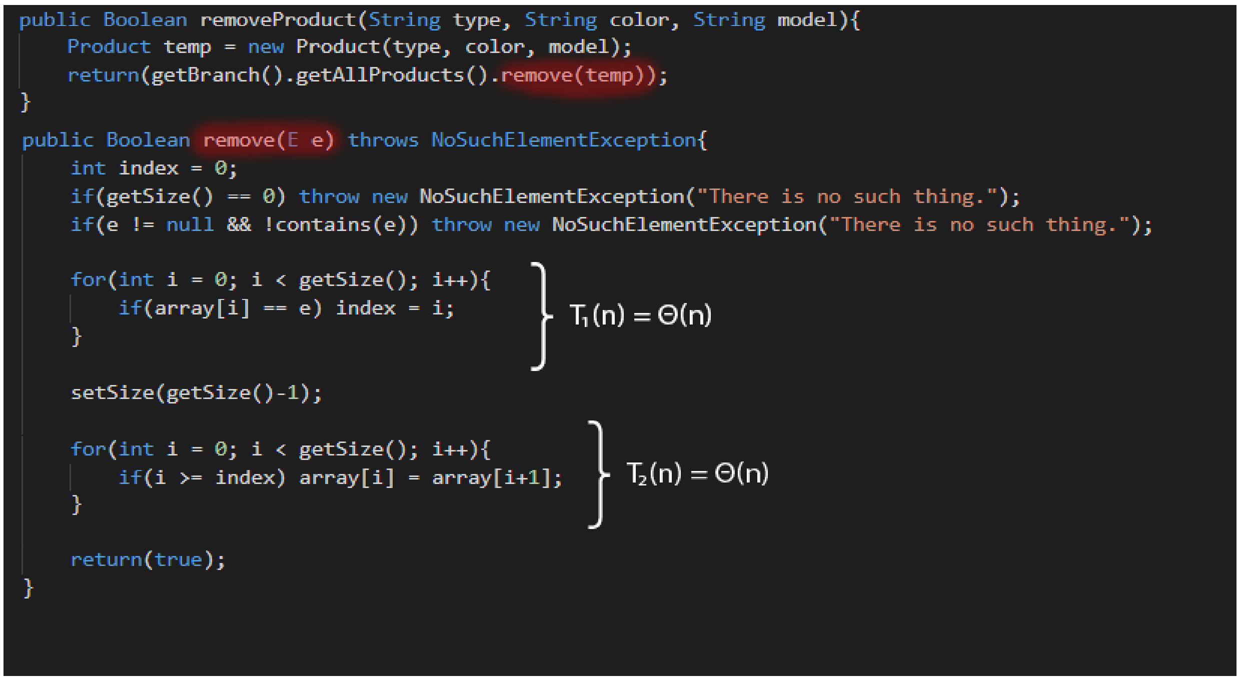
getBranch() and getAllProducts() also takes constant time, **θ(1).** So, I have to analyze add() method.

(n is the size of our array.)
In add(), E is Product in this case. It can’t return in first line because it is instance of Product. But it will call contains method.

contains’s complexity is constant time **θ(1) if Product is found in index 0 and return true in the best case. In worst case it does not contain, or Product is in the nth index and loop whole array n times, so worst time complexity is θ(n). So we can say $T_{contains}(n)$ = O(n).**

Then next 2 lines take constant time. After that we have if condition. If we enter the condition, changeCapacity’s time complexity is **θ(n)** because of the loop. Then we have another loop in the if condition and its time complexity is also **θ(n)** because of the loop. We add running times in consequtive statements so, general **T_1(n) = θ(n+n) = = θ(n). (if condition)**

So, for the add method, in worst case array is out of capacity and it must change its capacity with its double. In this case, it enters the if condition. **T_w(n) = θ(n).**
In best case it just adds the product in the end and capacity is not full. It won’t enter the if condition. And adding in the best case **T_b(n) = θ(1). In general, we can say T(n) = O(n).**
Thus, addProduct method’s T(n) = O(n). Again T_b(n) = θ(1) and T_w(n) = θ(n). Also we can say adding is amortized constant time.

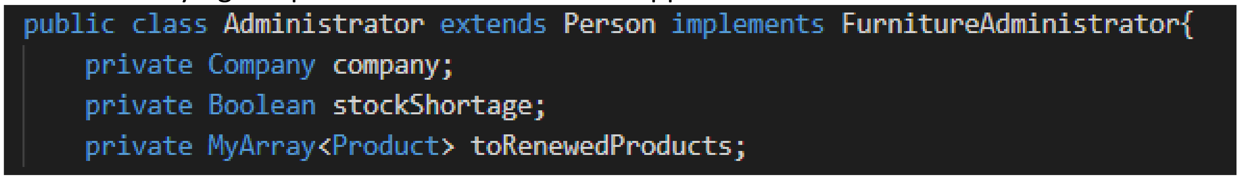


```
public Boolean removeProduct(String type, String color, String model){
    Product temp = new Product(type, color, model);
    return(getBranch().getAllProducts().remove(temp));
}
public Boolean remove(E e) throws NoSuchElementException{
    int index = 0;
    if(getSize() == 0) throw new NoSuchElementException("There is no such thing.");
    if(e != null && !contains(e)) throw new NoSuchElementException("There is no such thing.");

    for(int i = 0; i < getSize(); i++){
        if(array[i] == e) index = i;
    }

    setSize(getSize()-1);

    for(int i = 0; i < getSize(); i++){
        if(i >= index) array[i] = array[i+1];
    }

    return(true);
}
```

In my removeProduct method, I again call a method of my own container.
In remove method, getting the index of the to-be-removed Product takes **θ(n) time.** Shifting the array is also takes **θ(n) time. So, remove method's time complexity is θ(n).**

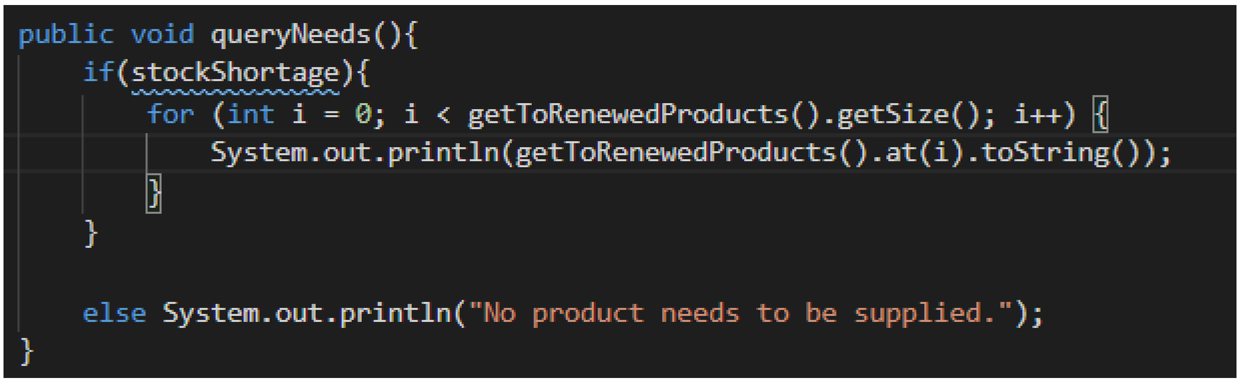
Thus, removeProduct method's T(n) = θ(n) also.

III. Querying the products that need to be supplied.



```
public class Administrator extends Person implements FurnitureAdministrator{
    private Company company;
    private Boolean stockShortage;
    private MyArray<Product> toRenewedProducts;
```

Employees make Boolean stockShortage true when any product needs to be supplied and add the product into toRenewedProducts. With this information,



```
public void queryNeeds(){
    if(stockShortage){
        for (int i = 0; i < getToRenewedProducts().getSize(); i++) {
            System.out.println(getToRenewedProducts().at(i).toString());
        }
    }

    else System.out.println("No product needs to be supplied.");
}
```

queryNeeds function just prints if stockShortage is false, which is our best case in terms of time complexity. **T_b(n) = θ(1). (n is the size of the toRenewedProducts array)**

In our worst case we enter the loop and run n times. **T_w(n) = θ(n).**
In general case, T(n) = O(n).

Part 2:

a) Explain why it is meaningless to say: “The running time of algorithm A is at least $O(n^2)$”.

We use big-O notation for asymptotic upper bounds, $O(n^2)$ is upper bound of the function. So, it is meaningless to say **at least.** Running time of the algorithm A can at most $O(n^2)$. Because f(n) should be equal or smaller than n^2**.**

b) Let f(n) and g(n) be non-decreasing and non-negative functions. Prove or disprove that: max(f (n), g(n)) = Θ(f(n) + g(n)).

To prove max(f (n), g(n)) = Θ(f(n) + g(n)) we must also prove max(f (n), g(n)) = O(f(n) + g(n)) and max(f (n), g(n)) = Ω (f(n) + g(n)) by the definition of the theta that I state in part c.

Let’s start with max(f (n), g(n)) = O(f(n) + g(n)). Let max(f (n), g(n)) be t(n) and (f(n) + g(n)) be h(n).
So, we must prove t(n) = O(h(n)).
We can write inequality like, t(n) ≤ f(n) or t(n) ≤ g(n), because maximum of them can be smaller or equal to f(n) and g(n).
If we sum these we get, 2*t(n) ≤ g(n) + f(n) = h(n).
We can also write it as, $\boldsymbol{t(n) \le \frac{1}{2} h(n)}$. If we compare what we got and the definition of big-oh, we can see it is basically same ½ is some constant.

Now, we must prove t(n) = Ω(h(n)).
We can write t(n) like t(n) ≥ f(n) and t(n) ≥ g(n) too because maximum of them can be greater or equal to f(n) and g(n). If we apply same steps we get, $\boldsymbol{t(n) \ge \frac{1}{2} h(n)}$. And it is like the omega definition.

Since we proved max(f (n), g(n)) = O(f(n) + g(n)) and max(f (n), g(n)) = Ω (f(n) + g(n)) then max(f (n), g(n)) = Θ(f(n) + g(n)) is also proved.

c) Are the following true? Prove your answer.

- **T(N) = θ(h(N)) if and only if T(N) = O(h(N)) and T(N) = Ω(h(N)).**
- **T(N) = O(f(N)) if there are positive constants c and n_0 such that T(N) ≤ c f(N) when N ≥n_0.**
- **T(N) = Ω(f(N)) if there are positive constants c and n_0 such that T(N) ≥ c f(N) when N≥ n_0.**

I will use these definitions to prove.

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

T(n) = Θ(2^n) iff T(n) = O(2^n) and T(n) = Ω(2^n).
Firstly, we can check big-oh notation. We should find a positive constant c and n_0 that meet the requirements. We can choose c = 2 and n_0 = 0 and we can see, $2^{n+1} \le 2.2^n = 2^{n+1}$ for all n ≥ 0. Thus, we can say, $2^{n+1} = O(2^n)$.

Secondly, we should check omega notation. We can choose c = 1 and n_0 = 0 and we can see, $2^{n+1} \ge 1.2^n$ for all n ≥ 0. Thus, we can say $2^{n+1} = \Omega(2^n)$.
Since, $2^{n+1} = O(2^n)$ and $2^{n+1} = \Omega(2^n)$, then $2^{n+1} = \Theta(2^n)$.

We can also prove with the limit rules. If the solution of the limit is c ∈ R ≠ 0 then f(n)=Θ(g(n)).

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to\infty} \frac{2^{n+1}.\ln(2)}{2^n.\ln(2)} \rightarrow \text{L'Hospital Rule} \rightarrow \frac{2^{n+1}.\ln(2)}{2^n.\ln(2)} = 2 \rightarrow \lim_{n\to\infty}(2) = 2 \neq 0.$$

$2^{n+1} = \Theta(2^n)$ proved again. **This statement is true.**

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

We should find c_1 and c_2 such that $c_1.2^n \leq 2^{2n} \leq c_2.2^n$ just like the I^st problem. If we look at the part that we try to prove it is $O(2^n)$. $2^{2n} \leq c_2.2^n = 4^n \leq c_2.2^n$ we can see that there is no such c_2 and n_0 that meet the requirements for all $n \geq n_0$. **So, $2^{2n} \neq O(2^n)$.**
Since, the definition of the theta notation stated if $2^{2n} \neq O(2^n)$ then $2^{2n} \neq \Theta(2^n)$. Thus, this statement is false.

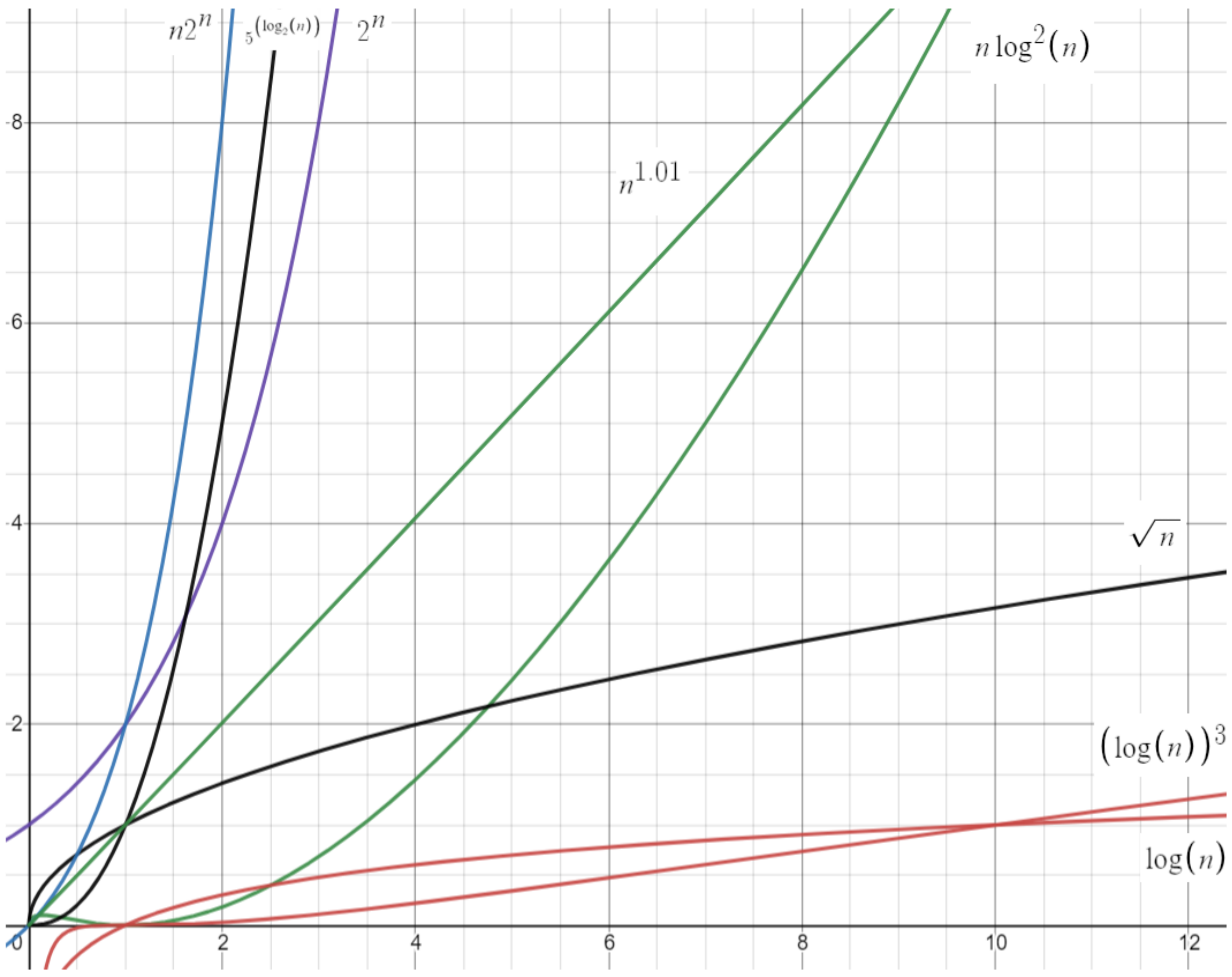
III. Let $f(n)=O(n^2)$ and $g(n)= \Theta(n^2)$. Prove or disprove that: $f(n) * g(n) = \Theta(n^4)$.

By the big-oh definition we can say, $f(n) \leq c_1.n^2$ for $n \geq n_0$.
Also, by the theta definition we can say, $c_1. n^2 \leq g(n) \leq c_3.n^2$ for $n \geq n_0$.
If we multiply these inequalities, $c_1. n^2 \leq f(n) * g(n) \leq c_1.c_3.n^4$
As we can see, right-side of the inequality proved that $f(n) * g(n) \subset O(n^4)$, but we can't say same thing about the left-side.
Thus, this statement is false. We can't directly say $f(n) * g(n) = \Theta(n^4)$. We can only say $f(n) * g(n) \subset O(n^4)$.

Part 3:

List the following functions according to their order of growth by explaining your assertions.

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



I used graph plotting application to see general behavior of the functions. As we can see, exponential functions grow a lot faster than others. Polynomials and then logarithmic functions next in the order.

Now I will use limit to compare their grow rate between each type of function.

f(x) grows faster than g(x) if $\lim\limits_{n\to\infty} \frac{f(x)}{g(x)} = \infty$.
If solution is constant other than 0, f(x) and g(x)'s grow rate is same. If it is 0, then g(x) grow faster than f(x).

$\lim\limits_{n\to\infty} \frac{n2^n}{2^n} = n = \infty \rightarrow 2^n < n2^n$	$\lim\limits_{n\to\infty} \frac{2^n}{2^{n+1}} = \frac{1}{2} \rightarrow 2^{n+1} \leq 2^n$
$\lim\limits_{n\to\infty} \frac{\log(n)}{n\log^2 n} = \frac{1}{n\log n} = \frac{1}{\infty} = 0 \rightarrow \log(n) < n\log^2 n$	$\lim\limits_{n\to\infty} \frac{n\log^2 n}{(\log n)^3} = \frac{n}{\log(n)} = \frac{1}{\frac{1}{n\ln(10)}} = \ln(10).n = \infty$ $\rightarrow (\log(n))^3 < n\log^2 n$
$\lim\limits_{n\to\infty} \frac{3^n}{2^n} = (\frac{3}{2})^n = \infty \rightarrow 2^n < 3^n$	$\lim\limits_{n\to\infty} \frac{n^{1.01}}{\sqrt{n}} = n^{0.51} = \infty \rightarrow \sqrt{n} < n^{1.01}$
$\lim\limits_{n\to\infty} \frac{n\log^2 n}{\sqrt{n}} = \sqrt{n}\log^2 n = \infty.\infty = \infty$ $\rightarrow \sqrt{n} < n\log^2 n$	$\lim\limits_{n\to\infty} \frac{n\log^2 n}{n^{1.01}} = \frac{\log^2 n}{n^{0.01}} \rightarrow$ 2x L'Hospital $\rightarrow \frac{\frac{200}{n}}{\frac{0.01}{n^{0.99}}} = \frac{20k}{n^{0.01}} = 0$ $\rightarrow n\log^2 n < n^{1.01}$

$5^{\log_2 n}$ is an exponential but its power is logarithmic. So, among all the exponentials, it's grow rate is smallest.
So, with the help of the graph and my limit comparison results, I can list them as follow:

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

Part 4:

Give the pseudo-code for each of the following operations for an array list that has n elements and analyze the time complexity:

- Find the minimum-valued item.

INPUT: array list with n element
OUTPUT: minimum-valued item
 	
IF n is 0
   THROW exception
ENDIF
 	
SET min index 0 of array list
 	
FOR n times
      IF element is smaller than min
         min is the element
      ENDIF
ENDFOR
RETURN MIN
	ANALYZE:
	
	For the best case, array have 1 element and it is the minimum valued item and this operation takes constant time. (It can also throw exception and terminate)
	$T_b(n) = \Theta(1)$.
	
	For the worst case we must check each element and it takes linear time, because it loops n times. **$T_w(n) = \Theta(n)$.**
	
	We can generally say **$T(n) = O(n)$.**

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

```
INPUT: array list with n element
OUTPUT: median item

IF n is 0
  THROW exception
ENDIF

IF n is even
  SET median index n/2


ELSE
  SET median index (n+1)/2
ENDIF

FOR n times
  SET count 0
  For n times
    IF element indexes are not same and inner loop element is smaller than outer loop element
      INCREMENT count
    ENDIF
  ENDFOR
  IF median index is equal to count
    RETURN outer loop element
  ENDIF
  ENDFOR
```

ANALYZE:

For the best case:

median item is the first element of the array list. So, outer loop will run 1 time, but still inner loop will run n times.
$T_b(n) = T_1(n) * T_2(n) = \Theta (n*1) = \Theta(n)$.

For the worst case:

Median item is the last element of the array list, so it will run n time outer loop and n time inner loop.
$T_w(n) = T_1(n) * T_2(n) = \Theta (n*n) = \Theta(n^2)$.

We can generally say, **$T(n) = O(n^2)$ and $T(n) = \Omega(n)$.**

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

```
INPUT: array list with n element, given value
OUTPUT: two elements

IF n is smaller than 2
  THROW exception
ENDIF

FOR n times
    For n times
        IF element indexes are not same and sum of the 2 elements are equal the given value
            PRINT x and y
            RETURN true
        ENDIF
    ENDFOR
ENDFOR
RETURN false
```

ANALYZE:

It can throw an exception and don't enter the loops and it takes constant time. **$\Theta(1)$.**

Otherwise we have two nested loops that runs n times both. But we can find our output in first index in outer loop and second index in inner loop (can't be first because elements shouldn't be same) and it takes constant time either because it returns afterwards.
$T_b(n) = \Theta(1)$.
For the worst case, there is no such elements and we must loop n times for both loops.
Since they are nested loops,
$T_w(n) = \Theta(n^2)$.

We can generally say, $T(n) = O(n^2)$.

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

```
INPUT: two ordered array list with n element
OUTPUT: merged single list

CREATE a new array list with 2n size
SET i 0
SET j 0

WHILE i and j is smaller than n
   IF ith element of array list1 is smaller than jth
      element of array list2
         ADD ith element of array list 1 to new
                array list
         INCREMENT i
   ELSE
         ADD jth element of array list 2 to new
                array list
         INCREMENT j
   ENDIF
ENDWHILE

WHILE i is smaller than n
   ADD ith element of array list 1 to new array list
   INCREMENT i
ENDWHILE

WHILE j is smaller than n
   ADD jth element of array list 2 to new array list
   INCREMENT j
ENDWHILE
RETURN new array list
```

ANALYZE:

For the best case:

if n is 1 first while loop only run 1 time. Adding to a list is amortized constant time. So, time complexity of first while loop is constant time, **$T_1(n) = \Theta(1)$.**

One of the while loops will run 1 time to add the last element of our merged single list and it takes constant time too.
$T_2(n) = \Theta(1)$.

$T_b(n) = T_1(n) + T_2(n) = \max(\Theta(1), \Theta(1)) = \Theta(1)$.

For the worst case:

First while loop will run n-1 times, functions n-1 and n grow at the same rate, so we can just say **$T_1(n) = \Theta(n)$.**

One of the second loops will run only 1 time to add the last element, so it is constant time. We initialize new array list with the total capacity of other 2, so it won't be out of the capacity.
$T_2(n) = \Theta(1)$.

$T_w(n) = T_1(n) + T_2(n) = \max(\Theta(n), \Theta(1)) = \Theta(n)$.

We can generally say $T(n) = O(n)$.

Part 5:

Analyze the time complexity and space complexity of the following code segments:

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

Time complexity: Θ(1), it takes constant time array size doesn't matter.
Space complexity: Θ(1), because it's not making any memory allocation depends on n, other than input array's size.

b)
```
    int p_2 (int array[], int n):
    {

            Int sum = 0
```

Time complexity: Θ(n), the loop runs n/5 time but constants doesn't effect the grow rate and other statements are simple.

`for (int i = 0; i < n; i=i+5)`
`sum += array[i] *`
`array[i])`
`return sum`
 	
`}`
	Space complexity: $\Theta(1)$, because it's not making any memory allocation depends on n, other than input array's 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])`
`}`
	Time complexity: Program doesn't terminate, infinite loop. Therefore, we can't analyze its time complexity. But if we change like int j = 1; then, inner loop is $\Theta(\log n)$ because it's growing exponentially. Outer loop is $\Theta(n)$, it runs n times. Since they are nested **$T(n) = \Theta(n.\log n)$.**
	Space complexity: $\Theta(1)$, because it's not making any memory allocation depends on n, other than input array's size.

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))`
`}`
	Time complexity:
	If statement → p_2's time complexity is $\Theta(n)$.
	$T_0(n) = \Theta(n)$.
	If the statement is true than p_3 will take $\Theta(n\log n)$.
	$T_1(n) = \Theta(n\log n)$.
	If the statement is not true p_1 and p_2 called. p_1's time complexity is $\Theta(1)$ and p_2's $\Theta(n)$.
	$T_2(n) = \Theta(1*n) = \Theta(n)$.
	
	Both in best case and worst case inside of if statement will run.
	But in best case it will enter else statement.
	So, **$T_b(n) = \max(T_0(n),T_2(n)) = \Theta(n)$.**
	$T_w(n) = \max(T_0(n),T_1(n)) = \Theta(n\log n)$.
	
	In general, we can say $O(n\log n)$ and $\Omega(n)$.
	
	Space complexity: : $\Theta(1)$, because neither of p_1, p_2 and p_3 and p_4 methods are not making any memory allocation depends on n, other than input array's size.