GEBZE TECHNICAL UNIVERSITY

CSE 222/505 DATA STRUCTURES AND ALGORITHMS

HW 2 REPORT

171044076
SEYMA NUR CANBAZ

PART 1

I)
```
public void searchProduct (int productID){
    int flag=0;  -> O(1)
    for(Branch br : branch){  -> O(m)
        if(br.isInThere(productID)){  -> O(n)
            for(int i=0; i<br.allProducts.length; i++){
                if(br.allProducts[i].getID == productID){
                    System.out.println("Product in" + br.getName()+ "branch");
                    flag++;
                }
            }
        }
    }
    if(flag==0)
        System.out.println("Product not found");   ] O(1)
}
```

O(m.n)   O(n)

```
public boolean isInThere (int productID){
    if(allProducts.length==0)   ] O(1)
        return false;
    else {
        for(int i=0; i<allProducts.length; i++){
            if(allProducts[i].getID() == productID)   ] O(n)
                return true;
        }
    }
    return false;  -> O(1)
}
```

O(n)

isInThere function: $T(n)_{worst} = O(n)$

$T(n)_{best} = \Omega(1)$

searchProduct function: $T(n) = O(m.n)$

n: size of the product array

m: size of the branch array

(∵ Each branch has product array seperetaly)

II) public void addProduct(Product newProduct, int branchID){

```
for (Branch br : branch){
     if(br.getID() == branchID){
          if(br.allProducts.length == 0){
O(1)           br.allProducts = new Product[1];
               br.allProducts[0] = new Product;
               System.out.println(newProduct.getID()+ "product is added");
     }
     else{
O(1) ← int flag=0;
          for(int i=0; i<br.allProducts.length; i++){
O(n)           if(br.allProducts[i].getID() == newProduct.getID())
                    flag++;
          }
O(n)      if(flag != 0)
O(1)           System.out.println(newProduct.getID() + "product is already added");
          else{
O(1) ←         Product[] temp = new Product[br.allProducts.length];
               for(int i=0; i<br.allProducts.length; i++)
O(n)                temp[i] = br.allProducts[i];
O(1) ←         br.allProducts = new Product[br.allProducts.length+1];
               for (int i=0; i<temp.length; i++)
O(n)                br.allProducts[i] = temp[i];
O(1) ←         br.allProducts[br.allProducts.length-1] = new Product;
O(1) ←         System.out.println(newProduct.getID() + "product is added");
          }
     }
}
```

O(m.n)

$T(n)_{worst}$	$= O(m.n)$
$T(n)_{best}$	$= \Omega(m)$

branch : branch array

allProduct : product array in each branch

m: length of branch array

n: length of product array

```
public void removeProduct (Product removedProduct, int branchID){
    for (Branch br : branch){
        if (br.getID() == branchID){
O(1)        if (br.allProducts.length == 0)
                return;

            else {
O(1)  <-        int flag = 0;
O(1)  <-        int index = -1;
                for (int i=0; i< br.allProducts.length; i++){
O(n)                if (allProducts[i] == removedProduct){
                        flag++;
                        index = i;
                    }
                }
                if (flag != 0){
O(n)                for (int i = index; i< br.allProducts.length-1; i++)
                        br.allProducts[i] = br.allProducts[i+1];
O(n)
O(1) <-             Product[] temp = new Product[br.allProducts.length-1];
O(n)                for (int i=0; i<temp.length; i++)
O(n)                    temp[i] = br.allProducts[i];
O(1) <-             br.allProducts = new Product[temp.length];
O(n)                for (int i=0; i< br.allProducts.length; i++)
                        br.allProducts[i] = temp[i];
O(1) <-             System.out.println(removedProduct.getID() + "product is removed");
                }
O(1)            else
                    System.out.println("There is no product called" + removedProduct.getID());
            }
        }
    }
}
```

O(n)

O(n)

O(m.n)

$T(n)_{worst} = O(m.n)$

$T(n)_{best} = \Omega(m)$

m: length of the branch array

n: length of the product array in the branch

III)

```
public int getStockInfo (String productType){
    int counter = 0;   → O(1)
    for(int i=0; i<allProducts.length; i++){
        if(allProducts[i].getType() == productType)      ] O(n)
            counter++;
    }
    return counter;   → O(1)
}
```

Θ(n)

```
public void queryTheProducts(){
    int type1, type2, type3, type4, type5;   → O(1)
    type1 = br.getStockInfo("officeCabinet");   → Θ(n)
    type2 = br.getStockInfo("bookcase");   → Θ(n)
    type3 = br.getStockInfo("officeChair");   → Θ(n)
    type4 = br.getStockInfo("meetingTable");   → Θ(n)
    type5 = br.getStockInfo("officeDesk");   → Θ(n)

    System.out.println("office Cabinet: " + type1 +
                       "bookcase: " + type2 +
                       "office Chair: " + type3 +
                       "meeting table: " + type4 +
                       "office desk: " + type5 );      ] O(1)
}
```

Θ(n)

$$\boxed{\begin{array}{l} T(n)_{worst} = O(n) \\ T(n)_{best} = \Omega(n) \end{array} \Rightarrow T(n) = \Theta(n)}$$

n: length of product array (allProducts)

PART 2

a) Because we use the big-O notation to express the worst case while doing the algorithm analysis. In other words, we calculate the maximum running time of the algorithm in this way. We express the running speed of the algorithm with sigma notation. This hypothesis is incorrect because the minimum running time of the algorithm is expressed with big-O notation instead of sigma notation in the hypothesis.

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

Assume $f(n) \leq g(n)$, $\Rightarrow \max(f(n), g(n)) = g(n)$

Consider, $g(n) \leq \max(f(n), g(n)) \leq g(n)$

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

$\Rightarrow \frac{1}{2}g(n) + \frac{1}{2}g(n) \leq \max(f(n), g(n)) \leq f(n)+g(n)$

From what we assumed, we can write

$\Rightarrow \frac{1}{2}f(n) + \frac{1}{2}g(n) \leq \max(f(n), g(n)) \leq f(n)+g(n)$

$\Rightarrow \frac{1}{2}(f(n)+g(n)) \leq \max(f(n), g(n)) \leq f(n)+g(n)$

By the definition of theta,

$\max(f(n), g(n)) = \Theta(f(n)+g(n))$ $\boxed{\text{TRUE}}$

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

$$\lim_{n\to+\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to+\infty} \frac{2.2^n}{2^n} = \lim_{n\to+\infty} 2 = 2$$

It is known that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, $c \in \mathbb{R}$ then $f(n) = \Theta(g(n))$.

So the hypothesis is TRUE.

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

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

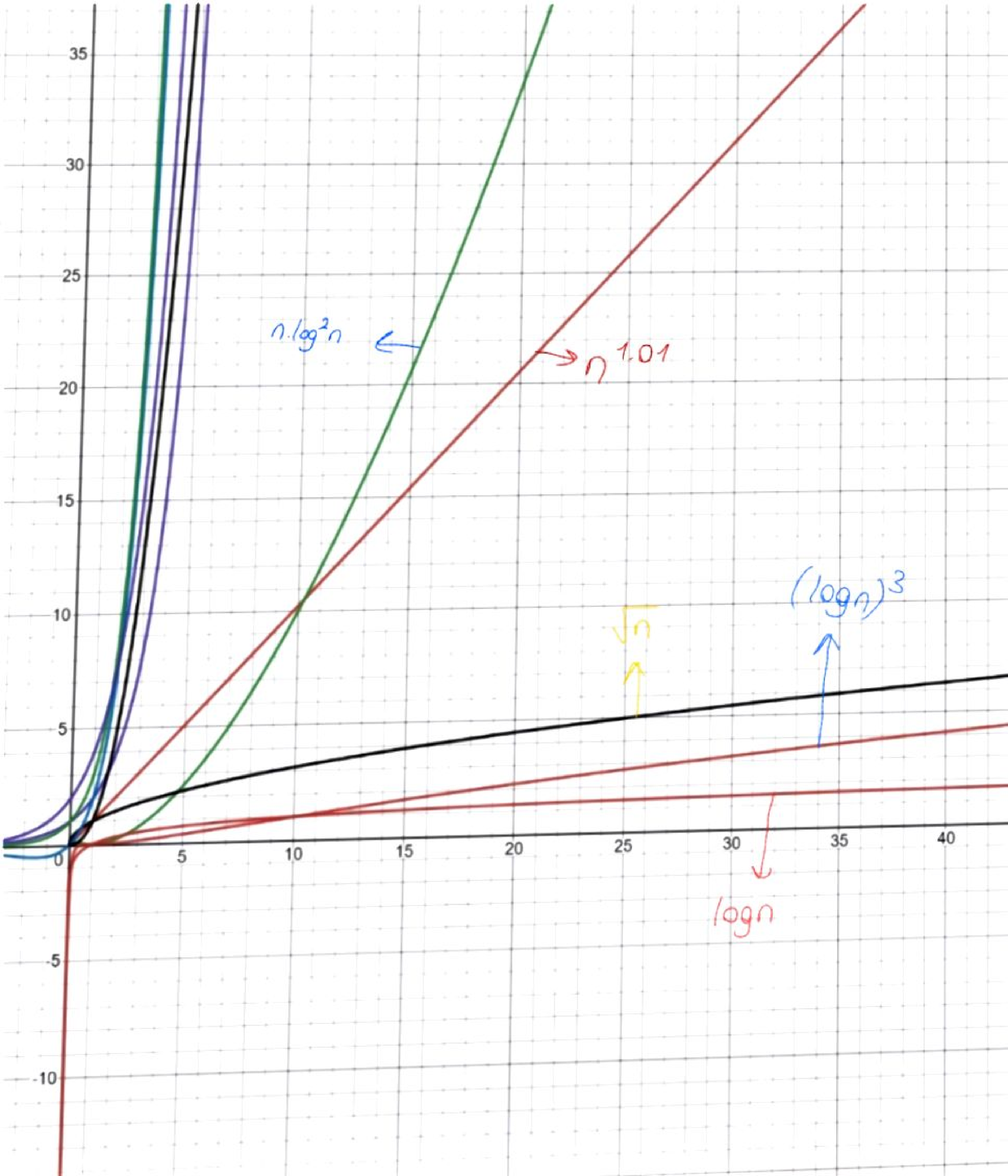
It is known that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c \in \mathbb{R}$ then $f(n) = \Theta(g(n))$. So this hypothesis is false because of this definition

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

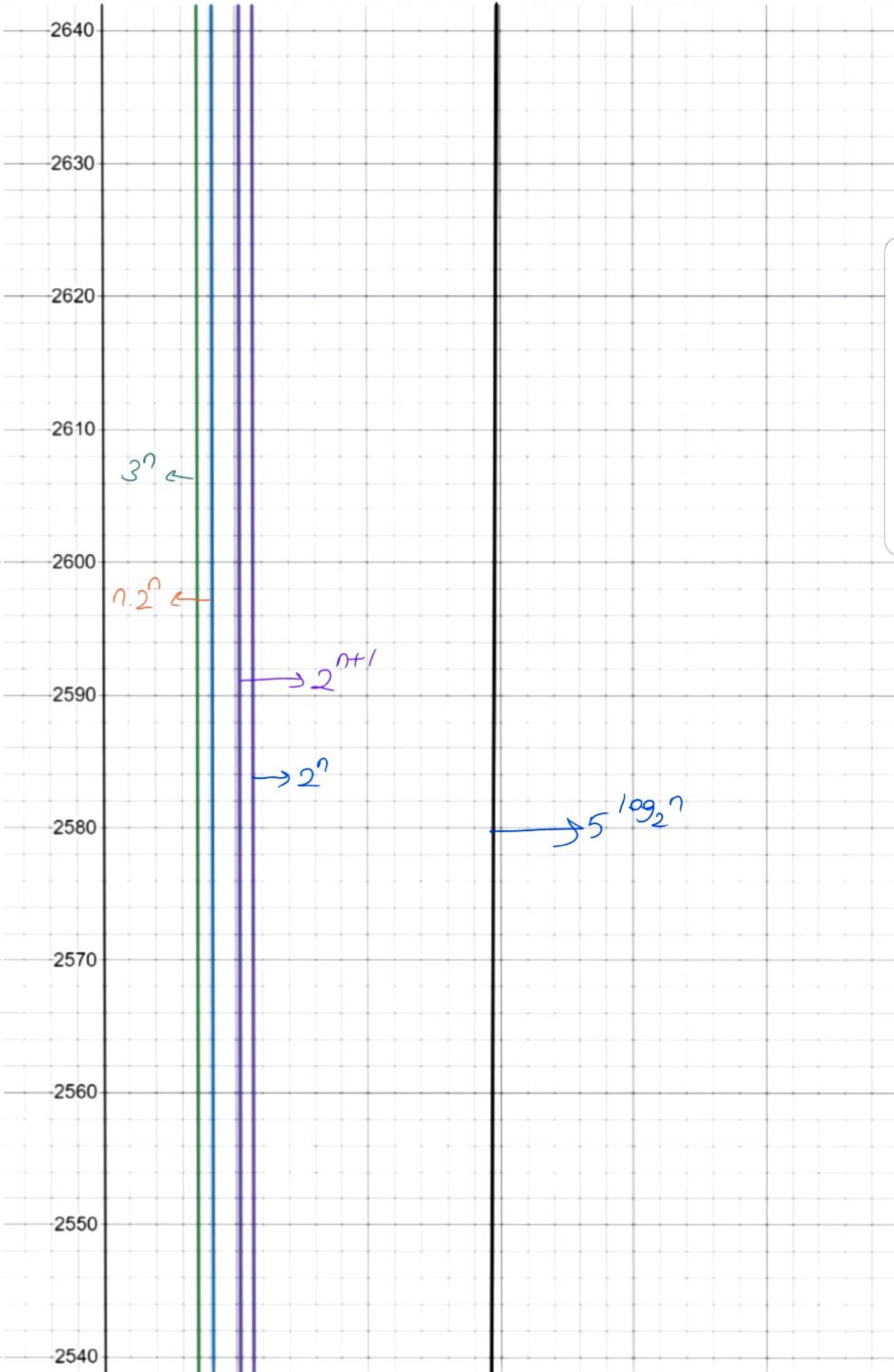
This hypothesis is wrong. The time complexity of the g function in worst case and best case are the same. But even though f function's worst case time complexity is $O(n^2)$, best case can be a smaller value than $O(n^2)$. For this reason, we cannot express the running time of the product of two functions with theta notation, because we need to know the best case working speed of the function f to be able to say this.

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

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



n.log²n
n^1.01
√n
(logn)³
logn


2640
2630
2620
2610
2600
2590
2580
2570
2560
2550
2540
3^n
$n \cdot 2^n$
2^{n+1}
2^n
$5^{\log_2 n}$

PART 4

a)

```
findMin (arr[], n)
{
    int min = arr[0]            → O(1)

    for (i=0; i<n; i++)         ]
    {                           ]
        if (min > arr[i])       ] O(n)
            min = arr[i]        ]
    }
    return min                  → O(1)
}
```

$T(n)_{worst} = O(n)$

$T(n)_{best} = \Omega(n)$

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

b)

```
void sort (arr[], n)
    temp = 0                                  → O(1)

    for (i=0; i<n; i++){                                          ]
        for (j=0; j<n-1; j++){                ]                   ]
            if (arr[j] > arr[j+1]){           ]                   ]
                temp = arr[j];                ] O(n-1)=O(n)       ] O(n^2)
                arr[j] = arr[j+1];            ]                   ]
                arr[j+1] = temp;              ]                   ]
            }                                                     ]
        }
}
```

$T(n)_{worst} = O(n^2)$

$T(n)_{best} = \Omega(n^2)$

$\Downarrow$

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

```
float findMedian (int arr[], int n){
    median = 0.0                   → O(1)
    temp[n]                        → O(1)

    for (i=0; i<n; i++){           ]
        temp[i] = arr[i]           ] O(n)
    }
    sort (temp, n);                → O(n^2)

    if (n%2 == 0)
        median = (temp[(n-1)/2] + temp[n/2]) /2.0     ] O(1)
    else
        median = temp[n/2]         ] O(1)
    return median                  → O(1)
}
```

c) void findSumEqual (arr[], n, sum){

```
for(i=0; i<n-1; i++){
    for(j=i+1; j<n; j++){
        if(arr[i]+arr[j] == sum){
            printf("First element: %d \n Second element: %d \n", arr[i], arr[j])
            return;
        }
    }
}
}
```

$\frac{n.(n-1)}{2}$

$T(n)_{worst} = O\left(\frac{n.(n-1)}{2}\right) = O\left(\frac{n^2-n}{2}\right) = O(n^2)$

$T(n)_{best} = \Omega(n^2)$

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

d) void merge (arr1[], arr2[], n, merged[]){ // Assume arr1 and arr2 are same size n.

```
for(i=0; i<n; i++)
    merged[i] = arr1[i]
for(j=0; j<n; j++){
    merged[i] = arr2[j]
    i++
}
```

$O(n)$

$O(n)$

```
for(k=0; k<2n; k++){
    for(l=0; l<2n-1; l++){
        if(merged[j] > merged[j+1]){
            temp = merged[j]
            merged[j] = merged[j+1]
            merged[j+1] = temp
        }
    }
}
}
```

$\rightarrow O(2n) = O(n)$

$O(2n-1) = O(n)$

$O(n^2)$

$T(n)_{worst} = O(n^2)$

$T(n)_{best} = \Omega(n^2)$

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

PART 5

a)
```
int p-1 (int array[]):
{
    return array[0] * array[2] ) → O(1)
}
```
$T(n)_{worst} = O(1)$

$T(n)_{best} = \Omega(1)$

$T(n)_{av} = \Theta(1)$

$S(n) = O(1)$

b)
```
int p-2 (int array[], int n):
{
    int sum = 0; → O(1)
    for (int i=0; i<n; i=i+5)                   ] O(n/5) = O(n)
        sum += array[i] * array[i])             ]
    return sum → O(1)
}
```
$T(n)_{worst} = O(n)$

$T(n)_{best} = \Omega(n)$

$T(n)_{av} = \Theta(n)$

$S(n) = O(1)$

c)
```
void p-3 (int array[], int n):
{
    for (int i=0; i<n; i++) → O(n)                              ] O(nlogn)
        for (int j=1; j<i; j=j*2)                   ] O(logn)   ]
            printf("%d", array[i] * array[j])       ]           ]
}
```
$T(n)_{worst} = O(n\log n)$

$T(n)_{best} = \Omega(n\log n)$

$T(n)_{av} = \Theta(n\log n)$

$S(n) = O(1)$

d)
```
void p-4 (int array[], int n):
{
    if (p-2 (array, n) > 1000) → O(n)           ] O(nlogn)
        p-3 (array, n) → O(nlogn)               ]
    else
        printf ("%d", p-1 (array) * p-2 (array, n)) → O(n)
}
```
$T(n)_{worst} = O(n\log n)$

$T(n)_{best} = O(n)$

$S(n) = O(1)$