Homework 2

Due date: March 25 2021, 9:30 AM

Part 1:

Analyze the time complexity (in most appropriate asymptotic notation) of the following procedures by your solutions for the Homework 1:

I. Searching a product.

Amount of branches = n
Amount of products at a branch = m

```
public void search(String productName) throws IllegalStateException{
    if(loggedIn) {
        for (int i = 0; i < Administrator.branches.size(); i++) {
            for (int j = 0; j < Administrator.branches.at(i).getProducts().size(); j++) {
                if (Administrator.branches.at(i).getProducts().at(j).getName().equals(productName)){
                    System.out.print("Branch code " + Administrator.branches.at(i).getBranchID()+ " has "
                            + Administrator.branches.at(i).getProducts().at(j).getAmount() + " of this product.\n");
                }
            }
        }
    else throw new IllegalStateException("This user is not logged in!");
}
```

loggedIn = True $\Rightarrow T_W(n,m) = O(n.m^2)$
loggedIn = false $\Rightarrow T_B(n,m) = \Theta(1)$
$T(n) = O(n.m^2)$

Note: m is different for every branch!

II. Add/remove product.

Amount of branches = n
Amount of products at a branch = m

```
public static void createProduct(int branchID,int productId,String name,String color)
                                                    throws IllegalArgumentException,NoSuchElementException{
    boolean branchFlag=false;

    int i;
    for(i=0;i<branches.size();i++){

        if(branchID == branches.at(i).getBranchID()){

            branchFlag=true;

            for(int j=0;j<branches.at(i).getProducts().size();j++){

                if(branches.at(i).getProducts().at(j).getId()==productId)

                    throw new IllegalArgumentException("This product already exists.");
            }

            branches.at(i).getProducts().insert(new Product(name,productId,color));
        }

    }

    if(!branchFlag)

        throw new NoSuchElementException("There is no branch with this ID.");
}
```

$T_W(m) = \Theta(m)$, $T_B(m) = \Theta(1)$

If a branch with this ID exists:

$T_W(n,m) = O(n \cdot m^2)$

If it doesn't:

$T_B(n,m) = \Theta(n)$

$T(n,m) = O(n \cdot m^2) \wedge \Omega(n)$

```java
public static void removeProduct(Product pr){
    for(int i=0;i< branches.size();i++){
        branches.at(i).getProducts().erase(pr); //returns true if removal is successful. Doesn't throw.
    }
}
```

O(n) O(1) O(m) O(m.n)

O(m.n)

III. Querying the products that need to be supplied.

Amount of branches = n
Amount of products at a branch = m

```java
public static CompanyContainer<Product> query(Branch b) {
    return b.getNeededProducts();
}
```

O(1) O(1)

After query:

```java
public static void addProduct(int branchID,int productId,int amount) throws NoSuchElementException,IllegalArgumentException{
    if(amount<=0)
        throw new IllegalArgumentException("You can't add zero or less products!");
    else if(getBranch(branchID) == null || getBranch(branchID).getProduct(productId)==null)
        throw new NoSuchElementException("Product or branch doesn't exist!");
    else
        getBranch(branchID).getProduct(productId).addProduct(amount);
}
```

O(1) O(n) O(m) O(1) O(n) O(m.n) O(m.n) O(m.n) O(n) O(m) O(1)

amount ≤ 0 ⇒ $T_B(n,m) = O(1)$

else ⇒ $T_W(n,m) = O(n.m)$

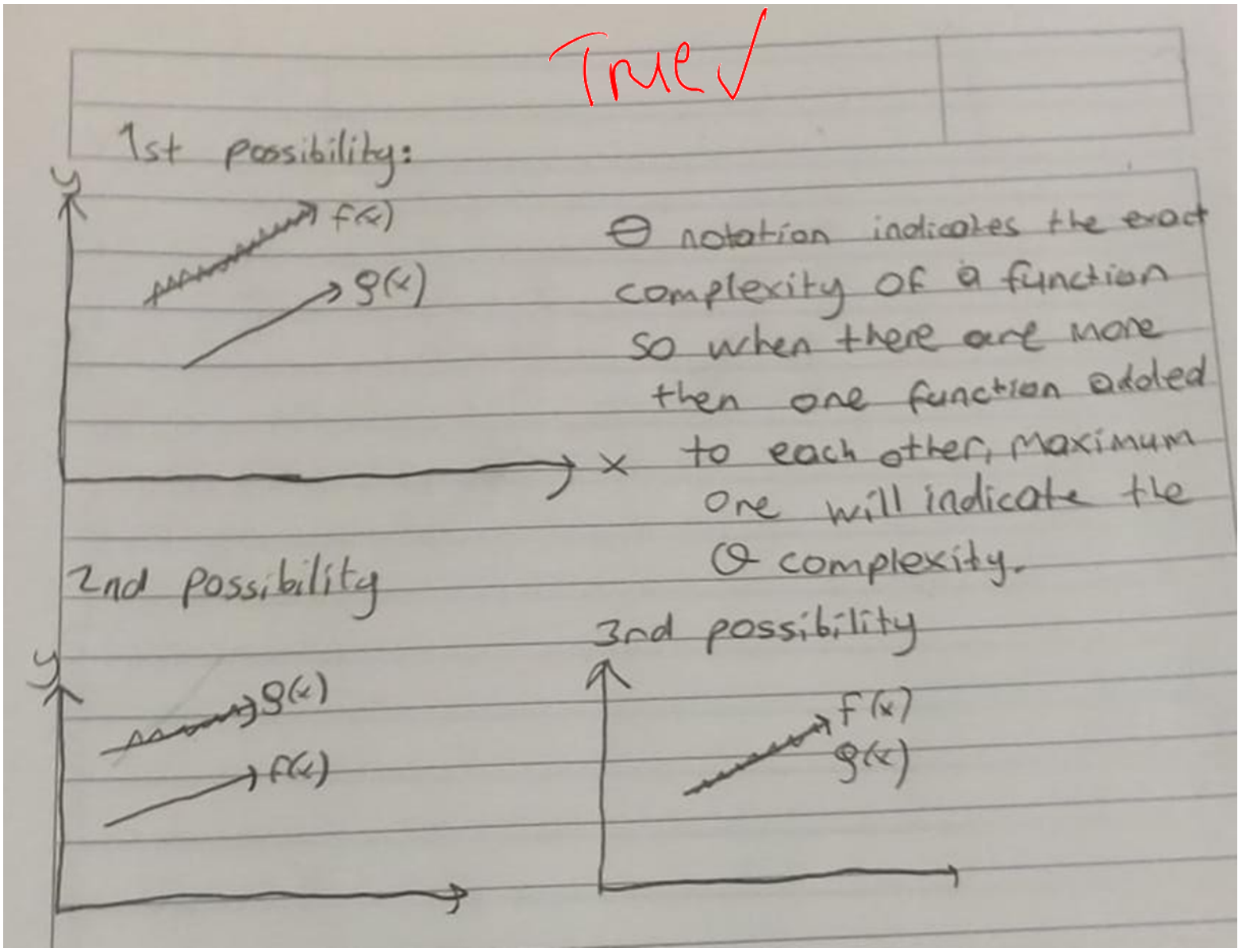
$T(n,m) = O(n.m)$

Part 2:

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

It is meaningless to say "The running time of algorithm A is at least O(n^2)" because Big Oh notation indicates the upper bound of the function. It might be meaningful to say "The running time of algorithm A is at max O(n^2)" but vice versa is not correct.

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)).



c) Are the following true? Prove your answer.

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

$c_1 2^n \leq 2^{n+1} \leq c_2 2^n$ $c_1 = 2, c_2 = 2, n_0 = 1$ ✓

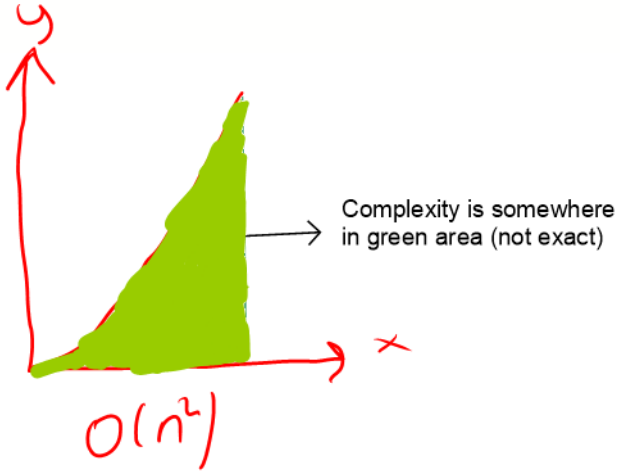
Assume true

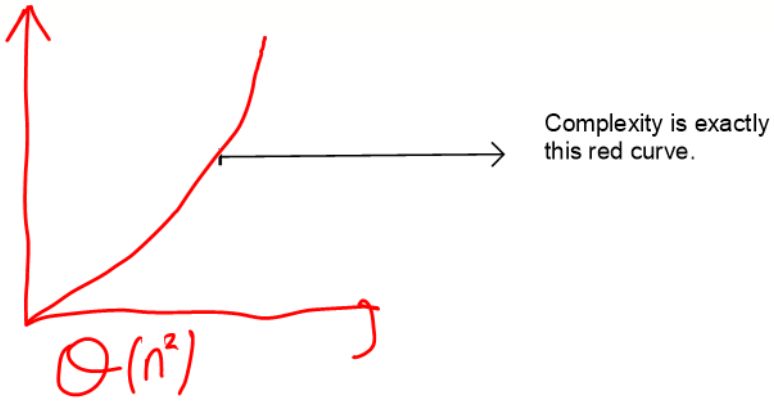
II. $2^{2n} = \Theta(2^n)$

$2^{2n} \leq c \cdot 2^n \Rightarrow 2^n \leq c$ ✗

Assume true ↳ constant

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





When this two are mutliplied, we lose the certainty. Complexity is somewhere between n^4 curve and 0.

So it is O(n^4), not Theta(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}, $5^{\log_2 n}$, $\log n$

log < root < linear < quadratic < exponential

log

log, quadratic

root

linear

linear x (log, quadratic)

exp. (log)

exp.

exp.

linear x exp.

exp.

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

Smallest Growth Rate

Biggest Growth Rate

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.

```
set min to zero
return null if array is empty
else
    iterate through the list until the end
        if current element is less then min
            set min to current element
return min
```

$T_W(n) = O(n)$

$T_B(n) = O(1)$

$T(n) = O(n)$

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

$T_W(n) = O(n), T_B(n) = O(1)$

$T(n) = O(n)$

```
iterate through the list (currentElement)
    set both higher and lower to zero
    iterate through current element to last element (i)
        increment higher by one if i is larger then currentElement.
        increment lower by one if i is smaller then currentElement.
    iterate through  current element to first element (j)
        increment higher by one if i is larger then currentElement.
        increment lower by one if i is smaller then currentElement.
    return currentElement if higher is equal to lower.
    return (sum of current element end the next one) / 2 if lower+1 is equal to lower.
    return (sum of current element end the previous one) / 2 if lower-1 is equal to lower.
return null if iteration of the list is finished.
```

$O(n^2)$

Find two elements whose sum is equal to a given value

```
iterate through list (current element: i)
    iterate through list (current element: j)
        if sum of i and j is equal to given value → Θ(1)
            create a new array list to hold two elements → Θ(1)
            add i to new list → Θ(1)
            add j to new list → Θ(1)
            return new list → Θ(1)
    return null (couldn't found) → Θ(1)
```

$T_W(n) = \Theta(n \cdot n) = \Theta(n^2)$

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

$T(n) = O(n^2)$

inner for:

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

$T_W(n) = \Theta(n)$

$T(n) = O(n)$

outer for:

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

$T_W(n) = \Theta(n)$

$T(n) = O(n)$

Total: $O(n \cdot n) = O(n^2)$

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

```
set both i and j to zero → Θ(1)
create a new array list (merged list) → Θ(1)

while i is less then list1's size and j is less then list2's size.  //this iterates until the  end of one of the lists.
    if i. index of the list1 is less then j. index of list2 → Θ(1)
        add i. index of the list1 to merged list. → Θ(1)
        increment i by one → Θ(1)
    else
        add j. index of the list2 merged list. → Θ(1)
        increment j by one → Θ(1)

//one of this two loops will iterate the remaining part of the other list. So in total, all three loops will add up to theta(n^2)

while i is less then list1's size
    add i. index of the list1 to merged list. → Θ(1)
    increment i by one → Θ(1)

while j is less then list2's size
    add j. index of the list2 to merged list. → Θ(1)
    increment j by one → Θ(1)
```

→ $\Theta(n^2)$ because in total, this three loops iterates through both lists once.

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

a)

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

Time: O(1)
Space: O(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: O(n) Space: O(n)

```
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])
}
```

Space: O(n) O(1)

Time: O(n) → exponential growth

space: O(1)

n × logn × space

Time: O(nlogn)
Space: O(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))
}
```

space: O(n) O(1)

p_2: O(n), O(n) (Time, space)

p_3: O(nlogn), O(n) (Time, space)

p_1: O(1) (Time), O(n) (space)

p_2: O(n) (Time), O(n) (space)

Time:

$T_B(n)$: $O(n) + O(n\log n) = O(n\log n)$ (if)

$T_W(n)$: $O(1) \cdot O(n) = O(n)$ (else)

Space:

It is O(n) for both situations. (O(n) + O(n) + O(n), O(n) + O(n))