

**GIT Department of Computer Engineering**  
**CSE 222/505 - Spring 2021**  
**Homework 2 # Report**

**Yeşim Yalçın**  
**200104004094**

## Part 1:

### 1-) Searching a Product

In my HW1, in CompanySystem.java, I have a public void searchProduct(Product searchedProduct) method. This method searches for a specific product in every stock and prints out in which stocks it is found and the amount in the stocks. In this method there are some other methods:

- public void isInStock(Product searchedProduct)
- public int getNo()
- public Product getProduct(int index)
- public int getAmount()

I will firstly examine these method's complexity to be able to examine searchedProduct method's complexity.

#### - public int getNo()

```
public int getNo() {return branchNumber;} —————→  $\Theta(1)$ 
```

This is a simple method with a time complexity  $T(n)=\Theta(1)$

#### - public int getAmount()

```
public int getAmount(){return amount;} —————→  $\Theta(1)$ 
```

This is a simple method with a time complexity  $T(n)=\Theta(1)$

#### - public Product getProduct(int index)

```
public Product getProduct(int index) throws InvalidIndex
{
    if(stock==null || index<0 || index>stock.length-1) —————→  $\Theta(1)$ 
        throw new InvalidIndex("There is not a product that is stored in this index."); —————→  $\Theta(1)$ 

    return stock[index]; —————→  $\Theta(1)$ 
}
```

This is a simple method with a time complexity  $T(n)=\Theta(1)$

#### - public void isInStock(Product searchedProduct)

```
public void isInStock(Product searchedProduct)
{
    for(int i=0; i<stock.length; ++i) —————→  $\Theta(1)$ 
        if(stock[i].equals(searchedProduct) && stock[i].getAmount()>0) —————→  $\Theta(1)$ 
            return i; —————→  $\Theta(1)$ 
    }
    return -1; —————→  $\Theta(1)$ 
}
```

$\left. \begin{array}{l} \text{for loop} \\ \text{if statement} \\ \text{return i;} \end{array} \right\} T(n)=O(n)$

Firstly starting to analyze from the inside. The if statement's time complexity in the for loop is always  $T(n)=\Theta(1)$ . However the for loop's time complexity depends on how many times the loop is run. We need to determine the best and the worst cases.

$T_b(n)=\Theta(1)$  is when the loop is done for once because the item is found in the first index of the stock array.

$T_w(n)=\Theta(n)$  is when the item cannot be found or the item is found in the last index of the stock array.

$T(n)=O(n)$

### - searchProduct(Product searchedProduct)

```
public void searchProduct(Product searchedProduct)
{
    try
    {
        for(int i=0; i<StoreList.length; ++i) →  $\Theta(1)$ 
        {
            if(StoreList[i].isInStock(searchedProduct)!=-1 && i!=0) →  $O(n)$ 
            {
                System.out.println(searchedProduct+" can be found in Branch
                "+StoreList[i].getNo()+".");
                System.out.println("There are
                "+(StoreList[i].getProduct(StoreList[i].isInStock(searchedProduct))).getAmount()
                ); →  $O(n)$ 
            }
            else if(StoreList[i].isInStock(searchedProduct)!=-1) →  $O(n)$ 
            {
                System.out.println(searchedProduct+" can be found in the Online Store.");
                System.out.println("There are
                "+(StoreList[i].getProduct(StoreList[i].isInStock(searchedProduct))).getAmount()
                ); →  $O(n)$ 
            }
        }
    }
    catch(InvalidIndex exc) →  $\Theta(1)$ 
    {
        System.err.print(exc);
        System.exit(-1);
    }
}
```

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

Analyzing the if statement:

The time complexity of the if statement's inside is

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

$T_w(n)=\Theta(n)$

$T(n)=O(n)$

Depending on the isInStock method. If the product is found in the first index, it is the best case.

If statement itself also has worst and best case scenarios according to the isInStock method. Both of the methods will run with the same searchProduct that's why their complexities will be the same.

If statement overall:

$T_b(n)=\Theta(1)+\Theta(1)=\Theta(1)$

$T_w(n)=\Theta(n)+\Theta(n)=\Theta(n)$

$T(n)=O(n)+O(n)=O(n)$

Analyzing the else if statement:

It has the same methods with the if statement therefore,  $T_b(n)=\Theta(1)$ ,  $T_w(n)=\Theta(n)$ ,  $T(n)=O(n)$ .

Analyzing the loop:

The loop's time complexity depends on how many times it runs. In this case, the loop will run  $n$  times in any scenario. That is why it has a time complexity of  $T(n) = \Theta(n)$ .

Rest of the methods have  $\Theta(1)$  time complexity that is why they do not change the outcome.

Analyzing the method as a whole:

The method has worst and best cases depending on the `isInStock` method's time complexity.

**$T_b(n) = \Theta(n)$**

**$T_w(n) = \Theta(n^2)$**

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

## 2-) Adding/Removing a Product

In my HW1, in `Branch.java` and `OnlineStore.java`, I have a public void `changeStock(Product addedProduct, int amount)` method. This method can change the amount of the products in stock also saves the change to the stock files. It can do both adding and removing. The method is the same in `Branch.java` and `Online.java`. The only difference is, the file name of the stocks that is why I will be analyzing the method in `Branch.java` only.

In this method there are some other methods:

- public void `changeAmount(int changeValue)`

I will firstly examine this method's complexity to be able to examine `changeStock` method's complexity.

### - public void `changeAmount(int changeValue)`

```
public void changeAmount(int changeValue) throws InvalidAmount
```

```
{
    setAmount(getAmount()+changeValue); —————>  $\Theta(1)$ 
}
```

```
private void setAmount(int newAmount) throws InvalidAmount
```

```
{
    if(newAmount<0) —————>  $\Theta(1)$ 
        throw new InvalidAmount("There cannot be negative amount of products."); —————>  $\Theta(1)$ 

    amount=newAmount; —————>  $\Theta(1)$ 
}
```

`setAmount` method is a simple method with a time complexity  $T(n) = \Theta(1)$ . `changeAmount` is also a simple method with a time complexity  $T(n) = \Theta(1)$ .

# - public void changeStock(Product addedProduct, int amount)

```
public void changeStock(Product addedProduct, int amount)
```

```
{
```

```
    int i;
```

```
try
```

```
{
```

```
    //Firstly changes the stock array
```

```
    for(i=0; i<stock.length; ++i) →  $\Theta(1)$ 
```

```
    {
```

```
        if(stock[i].equals(addedProduct)) →  $\Theta(1)$ 
```

```
        {
```

```
            stock[i].changeAmount(amount); →  $\Theta(1)$ 
```

```
            break; →  $\Theta(1)$ 
```

```
        }
```

```
    }
```

$T(n)=O(n)$

```
    //Then edits the branch's store file
```

```
    File inputFile = new File("Branch"+getNo()+"Stock.txt"); →  $\Theta(1)$ 
```

```
    File outputFile = new File("temp.txt"); →  $\Theta(1)$ 
```

```
    Scanner sc = new Scanner(inputFile); →  $\Theta(1)$ 
```

```
    FileWriter fw = new FileWriter(outputFile); →  $\Theta(1)$ 
```

```
    fw.write(sc.nextLine()+"\n"); →  $\Theta(1)$ 
```

```
    for(int j=0; j<i; ++j) →  $\Theta(1)$ 
```

```
        fw.write(sc.nextLine()+"\n"); →  $\Theta(1)$ 
```

$T(n)=O(n)$

```
    for(int j=0; j<3; ++j) →  $\Theta(1)$ 
```

```
    {
```

```
        if(j!=2) →  $\Theta(1)$ 
```

```
        fw.write(sc.next()+" "); →  $\Theta(1)$ 
```

```
        else
```

```
        fw.write(sc.next()+"\t"); →  $\Theta(1)$ 
```

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

```
    }
```

```
    fw.write(String.valueOf(stock[i].getAmount())); →  $\Theta(1)$ 
```

```
    sc.next(); →  $\Theta(1)$ 
```

```
    if(sc.hasNextLine()) →  $\Theta(1)$ 
```

```
    {
```

```
        fw.write(sc.nextLine()); →  $\Theta(1)$ 
```

```
        for(int j=0; j<stock.length-i-1; ++j) →  $\Theta(1)$ 
```

```
            fw.write("\n"+sc.nextLine()); →  $\Theta(1)$ 
```

$T(n)=O(n)$

```
    }
```

```
    sc.close(); →  $\Theta(1)$ 
```

```
    fw.close(); →  $\Theta(1)$ 
```

```
    sc=null; →  $\Theta(1)$ 
```

```
    System.gc(); →  $\Theta(1)$ 
```

```
    inputFile.delete(); →  $\Theta(1)$ 
```

```
    outputFile.renameTo(inputFile); →  $\Theta(1)$ 
```

```
}
```

```
catch(InvalidAmount exc) →  $\Theta(1)$ 
```

```
{
```

```
    System.err.print(exc);
```

```

        System.exit(-1);
    }
    catch (FileNotFoundException e2) →  $\Theta(1)$ 
    {
        e2.printStackTrace();
    }
    catch (IOException e1) →  $\Theta(1)$ 
    {
        e1.printStackTrace();
    }
}

```

Analyzing the inside of the first for loop:

There is a simple if statement in the first loop. This statement has methods that have  $\Theta(1)$  time complexity only. Therefore, it is  $T(n) = \Theta(1)$ .

Analyzing the first for loop:

The time complexity of the loop depends on how many times it runs. In this scenario, it has worst and best cases.  $T_w(n) = \Theta(n)$  happens if the product is in the last index of the stock array.  $T_b(n) = \Theta(1)$  happens if the product is in the first index of the stock array.  $T(n) = O(n)$

Analyzing the second for loop:

Inside of the loop has a time complexity of  $T(n) = \Theta(1)$ . The time complexity of the loop depends on how many times it runs. In this scenario, it has worst and best cases.  $T_w(n) = \Theta(n)$  happens when  $i$  is equal to  $n$ .  $i$  is equal to  $n$  if the first for loop statement was run with its worst case scenario.  $T_b(n) = \Theta(1)$  happens when  $i$  is equal to 0.  $i$  is equal to 0 if the first for loop statement was run with its best case scenario.  $T(n) = O(n)$ .

Analyzing the third for loop:

Inside of the for loop has a time complexity of  $T(n) = \Theta(1)$ . The loop runs exactly 3 times that is why it also has a time complexity of  $T(n) = \Theta(1)$ .

Analyzing the fourth for loop inside the if statement:

Inside of the for loop has a time complexity of  $T(n) = \Theta(1)$ . The for loop runs exactly  $n-i-1$  times however the value of  $i$  depends on the first for loop that is why it has worst and best cases.

$T_w(n) = \Theta(n-1) = \Theta(n)$  happens if  $i$  is 0.  $i$  is 0 if the first loop was run with its best case scenario.  $T_b(n) = \Theta(1)$  happens when  $i$  is equal to  $n$ .  $i$  is equal to  $n$  if the first loop was run with its worst case scenario.  $T(n) = O(n)$ .

Also `sc.hasNextLine()` is false if and only if  $i$  is equal to  $n$  which would make the  $T_b(n) = \Theta(1)$  however it overlaps with the  $T_b(n)$  of the for loop that's why nothing changes.

Rest of the methods have  $\Theta(1)$  time complexity that is why they do not change the outcome.

Analyzing the method as a whole:

The method has worst and best cases depending on the first, second and the fourth loop's time complexity. These loop's depend on the value of  $i$ .

If  $i$  is 0:

$T(n) = \Theta(1) + \Theta(1) + \Theta(n) = \Theta(n)$

If  $i$  is  $n$ :

$T(n) = \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$

This happens even the worst cases or the best cases happens, the complexity will be  $\Theta(n)$ .

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

### 3-)Querying the products that need to be supplied

In my HW1, in CompanySystem.java, I have a public void querySupplyList() method. This method prints out the products that need to be supplied if there is any.

In this method there are some other methods:

- public void printSupplyList()

I will firstly examine this method's complexity to be able to examine querySupplyList method's complexity.

#### - public void printSupplyList()

```
public void printSupplyList()
{
    if(SupplyList!=null) →  $\Theta(1)$ 
    {
        for(int i=0; i<SupplyList.length; ++i) →  $\Theta(1)$ 
        O(n) { System.out.println(SupplyList[i]+" to Store "+SupplyList[i].getStoreInfo());
    }
}
```

This method's time complexity depends on how many times the for loop runs.  $T_w(n) = \Theta(n)$  if the SupplyList is not empty or null.  $T_b(n) = \Theta(1)$  if the SupplyList is empty or null.

#### - public void querySupplyList()

```
public void querySupplyList()
{
    if(SupplyList==null || SupplyList.length==0) →  $\Theta(1)$ 
    {
        System.out.println("There are no products that need to be supplied."); →  $\Theta(1)$ 
    }
    else
    {
        System.out.println("These products should be supplied:"); →  $\Theta(1)$ 
        printSupplyList(); →  $O(n)$ 
    }
}
```

This method's time complexity depends on if it enters the if or else statements. If statement has a time complexity of  $T(n) = \Theta(1)$ . Else statement's time complexity depends on printSupplyList's time complexity. printSupplyList has worst and best cases however in this else statement, always the worst case of it will happen that's why  $T(n) = \Theta(n)$

Outcome:

**$T_b(n) = \Theta(1)$**

**$T_w(n) = \Theta(n)$**

**$T(n) = O(n)$**

### Part 2:

a-) It is meaningless to say "The running time of algorithm A is at least  $O(n^2)$ " because O notation provides an upper bound not a lower bound. Saying the running time of algorithm A is at most  $O(n^2)$  would be correct.

If we wanted to say the running time of algorithm A is at least ..., we should have used  $\Omega(n^2)$ .

b-) If  $f(n)$  and  $g(n)$  are non-decreasing and non-negative functions.  $\text{Max}(f(n), g(n)) = \Theta(f(n) + g(n))$  is true. Because when we are calculating the time complexity of combined multiple methods, the method with less grow rate will not have a significant effect comparing to the method with bigger grow rate. For example if  $f(n) = \Theta(n)$  and  $g(n) = \Theta(1)$ ,  $f(n) + g(n) = \Theta(n) + \Theta(1)$ . Here  $\Theta(1)$ 's grow rate is so slow compared to  $\Theta(n)$ 's grow rate, it will not have any effect when the two methods are combined, therefore  $T(n) = \Theta(n)$ .

c-)

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

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

$2(2^n) = \Theta(2^n)$  Because we do not care about the added constants, multiplied constants while determining the time complexity. They do not have significant effect.

Another way to prove:

For this to be true,  $2^{n+1} = O(2^n)$  and  $2^{n+1} = \Omega(2^n)$  should also be true.

$2^{n+1} \leq c(2^n)$  If there is a  $c$  that satisfies this for a big  $n$  value, then  $2^{n+1} = O(2^n)$  is right.

$2^{n+1} \geq c(2^n)$  If there is a  $c$  that satisfies this for a big  $n$  value, then  $2^{n+1} = \Omega(2^n)$  is right.

If  $c=2$ , the equations become  $2^{n+1} \leq 2^{n+1}$  and  $2^{n+1} \geq 2^{n+1}$  which proves our statements.

-  $2^{2n} = \Theta(2^n)$

Diving both of them by  $2^{2n}$

$(2^{2n}) / (2^{2n}) = 2^{(2n-2n)} = 2^0 = 1$

$(2^n) / (2^{2n}) = 2^{(n-2n)} = 2^{-n} = (1/2)^n$

$1 = \Theta((1/2)^n)$  is not correct  $1 = \Theta(1)$  would be correct. The grow rate of 1 is a lot more than the grow rate of  $(1/2)^n$ . Because of that we cannot say that the time complexity of  $2^{2n}$  is exactly  $2^n$ . However  $(1/2)^n = o(1)$  could be correct because the grow rate of a constant is a lot faster than  $(1/2)^n$ . Therefore  $2^n = o(2^{2n})$  would be correct.

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

This is not true because  $f(n) = O(n^2)$  means that  $f(n)$  at most has the time complexity of  $n^2$ .

However we don't know it's exact value. It can have  $n$  or constant time complexity too. If  $f(n)$  had constant time complexity,  $f(n) * g(n)$  would be  $\Theta(n^2)$  making the given equation false.

However we can fix the equation by changing the notation.  $f(n) * g(n) = O(n^4)$  would be true.

Because it means the combined method's time complexity can be at most  $n^4$ . But it still can be less depending on the  $f(n)$ 's actual time complexity.

### Part 3:

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

The image shows handwritten notes on a piece of paper. On the left, there is a list of functions with arrows pointing to their growth rates:  $\rightarrow n^{1.01}$ ,  $\rightarrow n^{0.5}$ ,  $\rightarrow 2^n$ ,  $\rightarrow 2^n$ ,  $\rightarrow 2^{n+1}$ ,  $\rightarrow n2^n$ , and  $\rightarrow 5^{\log n}$ . In the center, there is a list of functions:  $\rightarrow n \log^2 n$ ,  $\rightarrow \log^3 n$ , and  $\rightarrow \log n$ . To the right, there is a box containing a table of growth rates. The table has two columns: 'Normally' and 'c = o(log N)'. The rows are 'Constants', 'Logarithmics', and 'Exponentials'. The 'Normally' column lists  $\log N = o(\log^2 N)$ ,  $\log^2 N = (N \log N)$ , and  $N = o(N \log N)$ . The 'c = o(log N)' column lists  $N = o(N^2)$ . Below the table, it says 'Grow Rate Increases'.

	Normally	c = o(log N)
Constants	$\log N = o(\log^2 N)$	$\log^2 N = (N \log N)$
Logarithmics	$\log^2 N = (N \log N)$	$N = o(N \log N)$
Exponentials	$N = o(N^2)$	

Grow Rate Increases



$$* \log n < \log^3 n < n \log^2 n$$

Because:

$$\rightarrow \log n = o(\log^3 n)$$

$$\rightarrow \log^3 n = o(N)$$

$$\rightarrow N = o(N \log^2 N)$$

$$* 2^n = 2^{n+1}$$

Because:

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

\* Multiplied constants do not matter.

$$* 2^n < 3^n < n 2^n$$

Because:

Taking logs:

$$n \log 2 \quad n \log 3 \quad n^n \log 2$$

↓

↓

↓

n

n log 3

n^n

$$* n = o(n^2)$$

$$* n < n \log 3$$

$$* n \log^2 n < 2^n$$

Because:

Taking logs:

$$2n \log(\log n) < n \log(n)$$

$$* \log(\log n) = o(\log n)$$

$$* 5^{\log n} < 2^n < n 2^n$$

Because:

Taking logs:

$$n \log 2 \quad n^n \log 2 \quad \log n \log 5$$

↓

↓

↓

n

n^n

log n log 5

\* log 5 is a constant, log n makes the real change

$$\rightarrow n = o(N^2), \rightarrow \log n = o(n)$$

$$* n^{0.5} < 5^{\log n} < n^{1.01}$$

Because:

Taking logs:

$$0.5 \log n \quad \log 5 \log n \quad 1.01 \log n$$

$$* \log 5 \approx 0.7$$

$$0.5 \log n < 0.7 \log n < 1.01 \log n$$

$$* \log^3 n < n^{0.5} < n$$

Because:

Taking logs:

$$3 \log(\log n) < 0.5 \log n < \log n$$

$$* \log(\log n) = o(\log n)$$

$$* n 2^n > 3^n > 2^n = 2^{n+1} > n \log^2 n > n^{1.01} > 5^{\log n} > \sqrt{n} > \log^3 n > \log n$$

Growth Rates Comparison Asymptotically:

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

## Part 4:

### a-) Finding the Minimum Valued Item

Start

If the given arraylist is empty

Return -1

Initialize minimum = the arraylist's first element

Initialize i = 1

While i < the arraylist's size

Check if the element in the arraylist with i'th index < the minimum

If it is, assign minimum = the element in the arraylist with i'th index

Increment i by one

Return minimum

End

If the arraylist is empty, the operation will end right after the first if statement. Therefore,  $T_b(n) = \Theta(1)$ .

All the methods inside the loop has  $\Theta(1)$  time complexity. However the loop has best and worst cases. If the arraylist has only one element, the loop will not run even for once.  $T_b(n) = \Theta(1)$ .

However, in other cases the loop will run n times.  $T_w(n) = \Theta(n)$ .  $T(n) = O(n)$ .

Rest of the methods all have  $\Theta(1)$  time complexity, they do not change the outcome.

Analyzing as a whole:

**$T_b(n) = \Theta(1)$**

**$T_w(n) = \Theta(n)$**

**$T(n) = O(n)$**

### d-) Finding the median

Start

If the given arraylist is empty

Return -1

Initialize counter as 0

Initialize temp as empty

For i = 0 to arraylist's size

For j = 0 to arraylist's size

If i != j and the element with index i in the list > the element with index j in the list

Increment counter by 1

Else If i != j and temp is not empty and the element with index j = temp

Increment counter by 1

If arraylist's size is not divisible by 2 and counter = (list's size - 1) / 2

Return the element with the index i in the list

Else If arraylist's size is divisible by 2 and temp is empty and counter = (list's size - 2) / 2

temp = the element with index i in the arraylist

Else If arraylist's size is divisible by 2 and temp is not empty and counter = list's size/2  
Return (temp + element with index i in the list)/2

counter=0

End

If the given arraylist is empty the loop will not run therefore  $T_b(n) = \Theta(1)$

Analyzing the the inner loop:

The inner loop has if and else statements. Both of the statements have operations that have  $\Theta(1)$  time complexity.

The inner loop runs n time everytime. That's why  $T(n) = \Theta(n)$

Analyzing the outer loop:

The outer loop has operations which have  $T(n) = \Theta(1)$  time complexity. However, it will run more than once and the run time will depend on the if and else if statements. In the worst case the loop will run  $n \times n$  times if the median element is in the last index.  $T_w(n) = \Theta(n^2)$

Combined:

**$T_b(n) = \Theta(1)$**

**$T_w(n) = \Theta(n^2)$**

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

c-) Finding two elements whose sum is equal to a given value

Start

If the given arraylist is empty

Print "there is no elements in the list"

Return

For i=0 to arraylist's size-1

For j=i+1 to arraylist's size

If the element with the i'th index + the element with j'th index = given value

Print the elements with i'th and j'th indexes

Print "These elements' summation is equal to the given value."

Print "Their indexes are:"

Print i, j

Return

Print "There is no 2 elements that has the given value as their summation."

End

If the arraylist is empty, the operation will end right after the first if statement. Therefore,  $T_b(n) = \Theta(1)$ .

Analyzing the inner for loop:

The if statement has methods with  $\Theta(1)$  time complexity. However the loop's time complexity depends on how many times it runs. If there is only one element in the arraylist, the loop will not run then  $T_b(n) = \Theta(1)$ . Moreover, if the searched elements are 0th and 1st elements, then the loop is run only once so again  $T_b(n) = \Theta(1)$ . If there is n elements in the list and the searched element is the nth element, the loop will run  $n-i-1$  times.

$$T_w(n) = \Theta(n-i-1) = \Theta(n)$$

$$T(n) = O(n)$$

Analyzing the outer loop:

The outer loop's time complexity depends on how many times it runs.

If the arraylist has only one element or the searched element is found immediately, it runs only once.  $T_b(n) = \Theta(1)$ . However, if the searched elements are  $n-1$ 'th and  $n$ 'th elements then it runs  $n-1$  times and when we combine it with the inner loop, it runs:

$$(n-1) + (n-2) + (n-3) + \dots + 1 \text{ times} = (n^2 - n)/2$$

$$T_w(n) = \Theta((n^2 - n)/2) = \Theta(n^2)$$

$$\mathbf{T(n) = O(n^2)}$$

d-) Merging two lists to get a single list in increasing order.

Start

Clone the list1 to a temp arraylist

For  $i=0$  to list2's size

    For  $j=0$  to temp's size

        If the element with index  $i$  in list2 < the element with index  $j$  in temp

            Add the element with index  $i$  in list2 to the  $j$ 'th index of temp

            Break

        Else if  $j = \text{temp's size} - 1$

            Add the element with index  $i$  in list2 to temp

            Break

Return temp

End

If the list2 or the list1 or both is empty then the loops would not run or would run for only once with no operations done. Only the cloning will happen.  $T_b(n) = O(n)$

Analyzing the inner loop:

If and else if statement's time complexities are  $\Theta(1)$ . However, if statement has an add method which has  $O(m)$  time complexity. Else if statement, has an add method which has  $O(1)$  time complexity. The loop's run time depends on when we enter one of these statements.

The best case happens when the loop enters the if statement directly or if it enters the else if statement.  $T_b(n) = \Theta(1) \times O(n) = O(n)$ .

The worst case happens when the loop enters the if statement in  $(n-1)$ 'th run.

$$T_w(n) = \Theta(n-1) \times O(n) = O(n^2). \quad T(n) = O(n^2)$$

Analyzing the outer loop:

The outer loop's run time changes in each loop because everytime loop is repeated, the size of the temp will increase by one. The size of the temp will be  $n$  at the start. So the loop will run:

$$n + (n+1) + (n+2) + \dots + (n+n-1) \text{ times} = (3n^2 - n)/2$$

We need to consider the add methods in the loop too.

$T_b(n) = O(n) \times O(n) = O(n^2)$  (When the if statement is run directly in every loop or else if statement is run in every loop)

$$T_w(n) = \Theta((3n^2 - n)/2) \times O(n) = \Theta(n^3) \times O(n) = O(n^3)$$

$$T(n) = O(n^3)$$

Analyzing as a whole:

**Tb(n) = O(n)**

**Tw(n) = O(n<sup>3</sup>)**

**T(n) = O(n<sup>3</sup>)**

## Part 5:

a)

int p\_1 (int array[]):

```
{  
    return array[0] * array[2])  → Θ(1)  
}
```

Arrays can directly access the elements with the index. That's why array[0] and array[2] is Θ(1). Multiplying them then returning is also Θ(1).

**T(n) = Θ(1)**

**Space Complexity = O(1)** Because no extra space is used.

b)

int p\_2 (int array[], int n):

```
{  
    int sum = 0;  → Θ(1)  
    for (int i = 0; i < n; i=i+5)  → Θ(1)  
        sum += (array[i] * array[i])  → Θ(1) }  O(n)  
    return sum;  → Θ(1)  
}
```

Analyzing the inside of the for loop:

Arrays can directly access the elements with the index. That's why array[i] is Θ(1). Multiplying them then assigning is also Θ(1).

Analyzing the loop:

The loop's time complexity depends on how many times it runs. There are worst and best cases.

Tb(n) = Θ(1) happens when n is 0 or 1. If n is different than 0 or 1, the loop runs n/5 times.

Tw(n) = Θ(1/5 n) = Θ(n)

T(n) = O(n)

Rest of the methods have Θ(1) time complexity that is why they do not change the outcome.

Analyzing the method as a whole:

The time complexity of the method depends on how many times the loop runs. Therefore, we need worst and best cases.

**Tb(n) = Θ(1)**

**Tw(n) = Θ(n)**

**T(n) = O(n)**

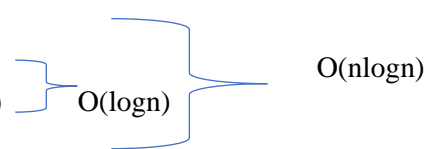
**Space Complexity = O(1)** Because the only extra space used is an integer which is 4 bytes.

c)

Note: In the original homework second for loop starts with  $j=0$  however, if  $j=0$  then the loop runs infinitely that is why I changed it as  $j=1$ .

p\_3 (int array[], int n):

```
{
    for (int i = 0; i < n; i++) →  $\Theta(1)$ 
        for (int j = 1; j < i; j=j*2) →  $\Theta(1)$ 
            printf("%d", array[i] * array[j]) →  $\Theta(1)$ 
    }
```



Analyzing the inner for loop's inside:

Arrays can directly access the elements with the index. That's why  $\text{array}[i]$  and  $\text{array}[j]$  is  $\Theta(1)$ .

Multiplying them then printing is also  $\Theta(1)$ .

Analyzing the inner for loop:

The loop's time complexity depends on how many times it runs. There are worst and best cases.

$T_b(n) = \Theta(1)$  happens when  $n$  is 0 or 1. In the other cases the for loop runs  $\log n$  times.

$T_w(n) = O(\log n)$

$T(n) = O(\log n)$

Analyzing the outer loop:

The loop's time complexity depends on how many times it runs. There are worst and best cases.

$T_b(n) = \Theta(1)$  happens when  $n$  is 0 or 1. In the worst case the loop runs  $n \times$  inner loop's run time.

It is  $n \times \log n$ .  $T_w(n) = O(n \log n)$

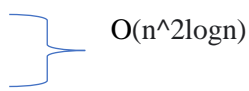
**$T(n) = O(n \log n)$**

**Space Complexity =  $O(1)$**  Because no extra space is used.

d)

void p\_4 (int array[], int n):

```
{
    If (p_2(array, n)) > 1000) →  $O(n)$ 
        p_3(array, n) →  $O(n \log n)$ 
    else
        printf("%d", p_1(array) * p_2(array, n)) →  $O(n)$ 
}
```



Analyzing the if statement:

p\_2 method inside the if statement is:

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

$T_w(n) = \Theta(n)$

$T(n) = O(n)$

p\_3 method inside the if statement is:

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

$T_w(n) = O(n \log n)$

$T(n) = O(n \log n)$

When combined:

$T_b(n) = \Theta(1)$  (If  $n=0$  or  $n=1$ )

$T_w(n) = O(n \log n) \times \Theta(n) = O(n^2 \log n)$  (Rest of the times)

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

Analyzing the else statement:

To be able to run the else statement the value of  $n$  is either 0 or  $p\_2$  returns a value which is smaller than 1000.

For the if statement:

$T_b(n) = \Theta(1)$  (If  $n$  is 0)

$T_w(n) = \Theta(n)$

$T(n) = O(n)$

For the methods in printf:

$p\_1$  method has  $\Theta(1)$  time complexity and  $p\_2$  has

$T_b(n) = \Theta(1)$  (If  $n$  is 0)

$T_w(n) = \Theta(n)$

$T(n) = O(n)$

When combined:

$T_b(n) = \Theta(1) + \Theta(1) = \Theta(1)$  (If  $n$  is 0)

$T_w(n) = \Theta(n) + \Theta(n) = \Theta(n)$

$T(n) = O(n)$

Analyzing as a whole:

**$T_b(n) = \Theta(1)$  (If  $n$  is 0)**

**$T_w(n) = \Theta(n) + O(n^2 \log n) = O(n^2 \log n)$**

**$T(n) = O(n^2 \log n)$**

**Space Complexity =  $O(1)$**  Because the only extra space used is an integer which is 4 bytes.