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

Ahmet Tuğkan Ayhan
1901042692

Part 1

I. Searching a product (Time Complexity = O(n))

```
public void addNewOrder(FurnitureBranch branch, User customer) throws InputMismatchException {
    // scanner
    Scanner scanner = new Scanner(System.in); // Initializing scanner Time Complexity = Θ(1)
    // product
    String      productName;     // name of the product which user chose Time Complexity = Θ(1)
    Model       productModel;     // model of the product which user chose Time Complexity = Θ(1)
    Color       productColor;     // color of the product which user chose Time Complexity = Θ(1)
    Furniture[] productArr; // product will be removed from this array Time Complexity = Θ(1)
    int         productAmount = 0;  // shows how many available products are there for the specified product
                        Time Complexity = Θ(1)

    System.out.print("Product name  : "); Time Complexity = Θ(1)
    productName = scanner.nextLine(); Time Complexity = Θ(1)
    productArr = branch.getFurnitureArr(productName); Time Complexity = Θ(1) (Method explained below after this screenshot)
    System.out.print("Product model : "); Time Complexity = Θ(1)
    productModel = Model.getModel(scanner.next()); Time Complexity = Θ(1)     public Model getModel() { return model; }
    System.out.print("Product color : "); Time Complexity = Θ(1)
    productColor = Color.getColor(scanner.next()); Time Complexity = Θ(1)     public Color getColor() { return color; }

O(n)
    try {
        productAmount = branch.getCount(productArr, productColor, productModel)   Worst case (Tw) : Θ(n),
    } catch (NullPointerException n) {                                           Best case (Tb) : Θ(1)     O(n)
        System.err.println("Product array is Null."); Θ(1)                       n = productArr.length
    }

                                              If-else block:
                                              Worst case (Tw) = Θ(n) , n = productArr.length    O(n)
                                              Best case (Tb) = Θ(1)
    // getting customer's information
    if(productAmount == 0)
        System.out.println("Sorry! The product you are looking for is out of stock."); Time Complexity = Θ(1)
    else {
        // removing the product from stock
        branch.removeProduct(productArr, productColor, productModel); Θ(n)
        // adding as previous order                                                  Methods are explained below
        customer.setOrder(productName, productModel, productColor); Θ(1)
        // informing the customer
Θ(1)    System.out.println("One \"" + productName + " - " + productModel + " - " + productColor +
                "\" is removed from the stock.\nYou can give it to the customer. Customer's information:" +
                "\nName    : " + customer.getName() + Θ(1)              @Override
                                                                        public String getName() { return name; }
                "\nSurname : " + customer.getSurname() + Θ(1)           @Override
                                                                        public String getSurname() { return surname; }
                "\nEmail   : " + customer.getEmail() + Θ(1)             @Override
                                                                        public String getEmail() { return email; }
                "\nID      : " + customer.getCustomerID()); Θ(1)        /** Returns the ID of customer ...*/
                                                                        public int getCustomerID() { return customerID; }
    }
}
```

```
      public void setOrder(String productName, Model productModel, Color productColor) throws ArrayIndexOutOfBoundsException {
          // setting product's full name
          String product = productName + " - " + productModel + " - " + productColor; Θ(1)
Θ(1)      // set full name as previous order
          previousOrders[getOrderCount()] = product; Θ(1)
          // increasing order count by one
          setOrderCount(getOrderCount() + 1); Θ(1)
      }
```

```java
public Furniture[] getFurnitureArr(String name) {
    switch (name) {
        case "Office Chair":
            return officeChairs; Θ(1)
        case "Office Desk":
            return officeDesks; Θ(1)
        case "Meeting Table":
            return meetingTables; Θ(1)
        case "Bookcase":
            return bookcases; Θ(1)
        case "Office Cabinet":
            return officeCabinets; Θ(1)
        default:
            return null; Θ(1)
    }
}
```

Θ(1)

```java
public int getCount(Furniture[] f, Color c, Model m) {
    int count = 0; Θ(1)
    for (int i=0; i<f.length; i++) Θ(n)
        if(f[i] != null) Θ(1)                                     Θ(1)
            if (f[i].getColor() == c && f[i].getModel() == m)
                count++; Θ(1)

    return count; Θ(1)
}
```

Θ(n) Θ(1)

```java
/** Removes one product from the furniture array with given color and model ...*/
public void removeProduct(Furniture[] f, Color c, Model m) {
    // Temporary Furniture array
    Furniture[] temp = f; Time Complexity = Θ(1)
    Furniture removed = null; Time Complexity = Θ(1)
    // Copying furniture array
    for(int i=0; i<f.length; i++) {
        try {                                Θ(1)
            // skipping selected furniture
            if(f[i].getModel() == m && f[i].getColor() == c)
                f[i] = null; Θ(1)
            else
                f[i] = temp[i]; Θ(1)
        } catch (NullPointerException ignored) { }
    }
}
```

n = f.length Θ(n)

Θ(1)

Time Complexity = Θ(n)

addNewOrder :

- Worst case (Tw) : Θ(n) , n = f.length
- Best case (Tb) : Θ(1) , Most appropriate : O(n)

II. Add / remove product

A. Add Product (Time Complexity = O(n))

```
/** Adds a product to the furniture array with given color and model. ...*/
public void addProduct(Furniture[] f, Color c, Model m) {
    // allocating new furniture
    Furniture furniture = null; Θ(1)
    // deciding furniture type                  public String getName() { return name; } Θ(1)
    switch (f[0].getName()) {
        case "Office Chair":
            furniture = new OfficeChair(c, m); Θ(1)
            break;
        case "Office Desk":
            furniture = new OfficeDesk(c, m); Θ(1)
            break;
        case "Meeting Table":                                                            Θ(1)
            furniture = new MeetingTable(c, m); Θ(1)
            break;
        case "Bookcase":
            furniture = new Bookcase(m); Θ(1)
            break;
        case "Office Cabinet":
            furniture = new OfficeCabinet(m); Θ(1)
            break;
        default:
            System.err.println("Couldn't add the product."); Θ(1)
            break;
    }
    if(furniture == null) Θ(1)
        return; Θ(1)                Θ(1)
                                                  Worst Case : Θ(n)  → O(n)
    // assigning new furniture to the null element    Best Case : Θ(1)
    for(int i=0; i<f.length; i++) Θ(n)
        if(f[i] == null) Θ(1)             Θ(n)
            f[i] = furniture; Θ(1)
}
```

Time Complexity ;

Worst Case : Θ(n), n = f.length

Best Case : Θ(1)

Most appropriate : O(n)

B. Remove Product (Time Complexity = Θ(n))

```java
/** Removes one product from the furniture array with given color and model ...*/
public void removeProduct(Furniture[] f, Color c, Model m) {
    // Temporary Furniture array
    Furniture[] temp = f; Time Complexity = Θ(1)
    Furniture removed = null; Time Complexity = Θ(1)
    // Copying furniture array
    for(int i=0; i<f.length; i++) {
        try {                                   Θ(1)
            // skipping selected furniture
            if(f[i].getModel() == m && f[i].getColor() == c)
                f[i] = null; Θ(1)
            else
                f[i] = temp[i]; Θ(1)
        } catch (NullPointerException ignored) { }
    }
}
```

n = f.length
Θ(n)
Θ(1)
Time Complexity = Θ(n)

```java
public Color getColor() { return color; }
```

time complexity : O(1) == Θ(1)

```java
public Model getModel() { return model; }
```

time complexity : O(1) == Θ(1)

III. Querying the products that need to be supplied

```java
/** Selectable menu for branches and prints all products of selected branch ...*/
public void inquireStock(Company company, FurnitureBranch[] branches) {
    // User Input & Scanner initializing
    int branch, furniture;Θ(1)
    Furniture[] f = null;Θ(1)
    String inquire; Θ(1)
    Scanner scanner = new Scanner(System.in); Θ(1)

    // Branch Menu
    for(int i = 0; i<company.getBranchNumber(); i++) {Θ(n)
        System.out.println( (i+1) + ") Branch " + (i+1));Θ(1)
    }
    System.out.print("Enter: ");Θ(1)

    branch = scanner.nextInt() - 1;Θ(1)
                @Override
                public int getBranchNumber() { return branchNumber; }Θ(1)
    if(branch < 0 && branch >= company.getBranchNumber()) {
        System.err.println("Invalid input");Θ(1)
        return;Θ(1)
    }

    // Furniture Menu
    System.out.println("1) Office Chairs");Θ(1)
    System.out.println("2) Office Desks");Θ(1)
    System.out.println("3) Meeting Tables");Θ(1)
    System.out.println("4) Bookcases");Θ(1)
    System.out.println("5) Office Cabinets");Θ(1)
    System.out.print("Enter: ");Θ(1)

    furniture = scanner.nextInt();Θ(1)
```

Θ(1)
Θ(n)
Θ(n)
Θ(1)
Θ(1)

Best Case : if condition holds and method returns,
Time Complexity : Θ(n)

Method continues...

```
switch (furniture) { Θ(1)          O(1)   O(m)   Θ(1)   Θ(m)
    case 1:
        branches[branch].printFurnitureArr(branches[branch].officeChairs); Θ(m . c . n)
        f = branches[branch].officeChairs; Θ(1)
        break; Θ(1)
    case 2:
        branches[branch].printFurnitureArr(branches[branch].officeDesks); Θ(m . c . n)
        f = branches[branch].officeDesks; Θ(1)
        break; Θ(1)          O(m)   Θ(m)
    case 3:
        branches[branch].printFurnitureArr(branches[branch].meetingTables); Θ(m . c . n)
        f = branches[branch].meetingTables; Θ(1)
        break; Θ(1)
    case 4:
        branches[branch].printFurnitureArr(branches[branch].bookcases); Θ(m . c . n)
        f = branches[branch].bookcases; Θ(1)
        break; Θ(1)
    case 5:
        branches[branch].printFurnitureArr(branches[branch].officeCabinets); Θ(m . c . n)
        f = branches[branch].officeCabinets; Θ(1)
        break; Θ(1)
    default:
        System.err.println("Invalid input."); Θ(1)
        break; Θ(1)
}
// Inquire admin
System.out.println("Inquire manager about products which out of stock? (Yes: y, No: n)"); Θ(1)
System.out.print("Enter: "); Θ(1)
// input
inquire = scanner.next(); Θ(1)
if(inquire.equals("y") || inquire.equals("Y")) Θ(1)
    informManager(branches, f, branch); O(m.c.n)
}
```

Worst Case
Θ(m . c . n)
Best Case
Θ(1)
Hence
O(m.c.n)

Worst Case
Θ(m.c.n)
Best Case
Θ(1)
O(m.c.n)

O(m.c.n)

O(m.c.n)
+
O(m.c.n)
=
$\boxed{O(m.c.n)}$

Method Ends...

Time Complexity : O(m.c.n)

Worst Case : Θ(m.c.n)

Best Case : Θ(1)

Showing inner methods that is used in the inquireStock method :

```
/** Prints available furniture products with their properties to the console ...*/
public void printFurnitureArr(Furniture[] f) {
    int amount = 0, index = 0; Θ(1)          Θ(m . c . n)
    // Furniture
    for(Model m : f[0].models) { Θ(m)
        for(Color c : f[0].colors) { Θ(c)
            try {
                amount = getCount(f, c, m); Θ(n)
            } catch (NullPointerException n) {       Θ(n)
                amount = 0; Θ(1)
            }

            index++; Θ(1)

            if(amount == 0)
Θ(1)        Θ(1) System.out.println(index + ")\t" + f[0].getName() + "\t(Model: " + m + ")\t(Color: " + c + ") \t(Amount: " + amount + ") ! Out of stock");
            else
            Θ(1) System.out.println(index + ")\t" + f[0].getName() + "\t(Model: " + m + ")\t(Color: " + c + ") \t(Amount: " + amount + ")");
        }
    }
    System.out.print("\n"); Θ(1)
}
```

already found it's time complexity

Θ(m . c . n)

```
public void informManager(FurnitureBranch[] branches, Furniture[] furniture, int branch) {      O(m.c.n)
    String userInput; Θ(1)
    Scanner scanner = new Scanner(System.in); Θ(1)
Θ(1)
    System.out.println("You informed the manager. Your manager told you that you can refill the branch stock."); Θ(1)
    System.out.println("Do you want the refill? (Yes: y , No: n)"); Θ(1)
    System.out.print("Enter: "); Θ(1)
    // user input
    userInput = scanner.next(); Θ(1)

    if(userInput.equals("y") || userInput.equals("Y"))                     Worst Case
        branches[branch].addAllProducts(furniture); Θ(m . c . n)            Θ(m.c.n)
    else                                                                    Best Case       O(m.c.n)
        System.err.println("Invalid input."); Θ(1)                          Θ(1)
}
```

```
public void addAllProducts(Furniture[] f) {
    int amount = 0, count = 0; Θ(1)                       Θ(m . c . n)
    for(Model m : f[0].models) { Θ(m)
        for(Color c : f[0].colors) { Θ(c)
            try {
                amount = getCount(f, c, m); Θ(n)
            } catch (NullPointerException n) {            Θ(n)
                amount = 0; Θ(1)                                    inside inner for loop:
            }                                                       worst case
Θ(m . c . n)                                                        Θ(n) + O(n) == Θ(n)
            if(amount == 0) {                                       best case
                count++; Θ(1)                                       Θ(n)
                addProduct(f, c, m); O(n)                 O(n)
            }
                Already found it's time complexity
        }
    }

    if(count != 0)
Θ(1)    Θ(1) System.out.println("Adding was successful. " + count + " product added in total.");
    else
        Θ(1) System.out.println("Stock is already full. No product is added.");
}
```

Part 2

a) Because when we say $O(n^2)$, we claim that this algorithm can't consume more time than "n^2". By the definition of Big O notation, the upper bound for the time consumed by any algorithm is presented as O(...). If we say "the running time of algorithm A is at least $O(n^2)$", we mean something like x <= 3 and the minimum x value can be 3. It can be true but there is no point to say anything like that since there is no value is avaliable for x except 3. The proper expression could be "the running time of algorithm A is at **most** $O(n^2)$."

b) Let's say $f(n) = n^2$ and $g(n)= n^3$, then max(f(n) , g(n)) would be equal to max(n^2, n^3) which equals to "n^3". On the other hand, $\Theta(f(n) + g(n))$ would be equal to $\Theta(n^2 + n^3)$ and since equation increases with the power 3 we can ignore "n^2" so final equation for theta is equal to $\Theta(n^3)$. Hence, $\max(n^2 , n^3) = \Theta(n^2 + n^3)$.

c)

I. $2^{n+1} = 2^n * 2$, and let's say:

$f(n) = 2^n$

$f(n) \Rightarrow \Theta(f(n))$

By the general properties of asymptotic notations:

$\Theta(f(n) * 2) = \Theta(f(n))$

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

Because since our algorithm increases with the rate of f(n), we can't say this algorithm's increasing rate is changing according to a constant value. To correct definition should be "This algorithm's increasing rate is changing according to value of n."

II. $2^{2n} = 2^n * 2^n$, and let's say:

$f(n) = 2^n$

$f(n) \Rightarrow \Theta(f(n))$

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

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

$\Theta((2^n)^2) = \Theta((2^{2n}))$, hence the equation is wrong. Because the increase rate of the algorithm is changing according to value of n. Since we multiply two " 2^n " values, we can't ignore one of them.

III. the algorithm f(n) has a best case and worst case because it is represented with big O notation.

worst case of f(n) : $\Theta(n^2)$

best case of f(n) : $\Theta(1)$

for worst case of f(n) and $g(n) = \Theta(n^2)$;

$f(n) * g(n) = \Theta(n^2 * n^2) = \Theta(n^4)$

for best case of f(n) and g(n) = $\Theta(n^2)$;
f(n) * g(n) = $\Theta(1 * n^2) = \Theta(n^2)$
now let's say f(n) * g(n) = h(n), then h(n) has two cases because of f(n);
best case of h(n) = $\Theta(n^2)$
worst case of h(n) = $\Theta(n^4)$, hence
h(n) = $O(n^4)$ and h(n) = $\Omega(n^2)$, so we can say that statement was wrong.

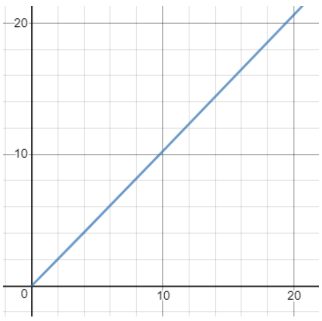
Part 3

First we can eliminate one of the functions because it's growth rate is equal to another function and we prove it that is true in part 2. 2^{n+1} and 2^n both have same growth rate so if we can find growth rate of one of them we can say that it is true for both of them.

For the question I will try to find their graph and see their growth rate.
And finally I will give (relatively big) a number to see if my order is true or not.
Except some functions which are easy to see their growth rate, I will give another number (a much bigger number) to choose which one's growth rate is lower or greater. Because some functions look slower at first but as we increase our value, they can make big difference.

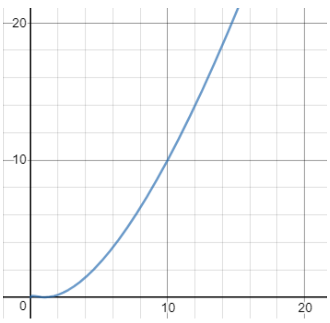
Graphs

$n^{1.01}$ =>

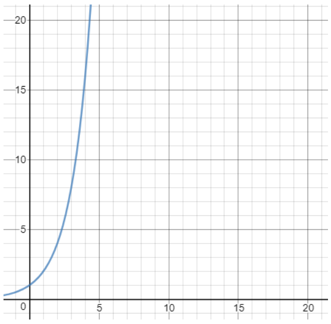


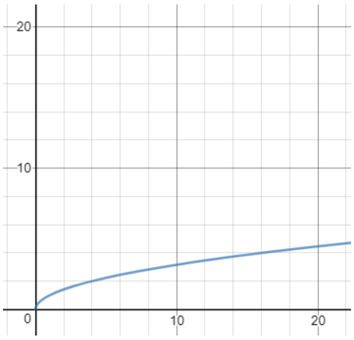
and n = 20, f(n) = 20.608 ; also n = 1000 , f(n) = 1071

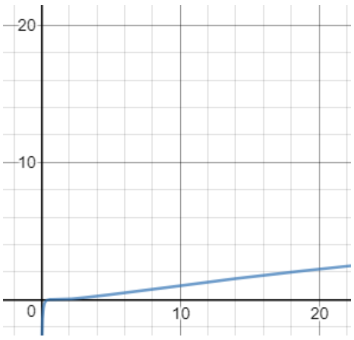
$n\log^2 n$ =>

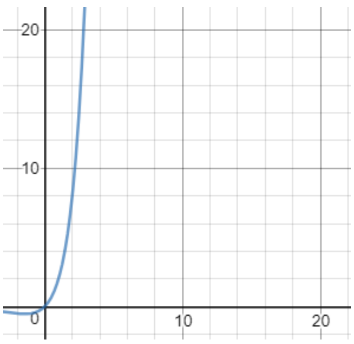


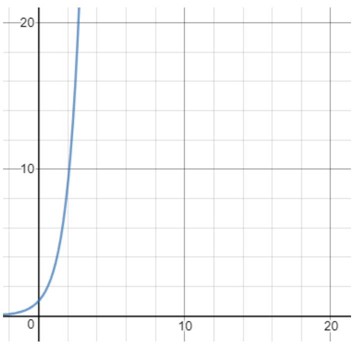
and n = 20, f(n) = 33.9 ; also n = 1000, f(n) = 9000

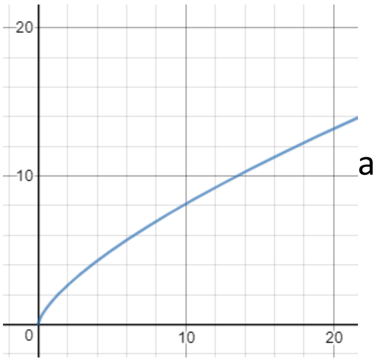
2^n and 2^{n+1}=>  and n = 20, f(n) = 1,048,576

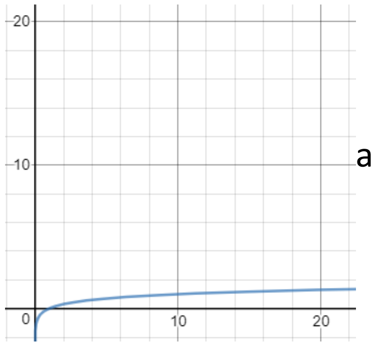
$\sqrt{n}$ =>  and n = 20, f(n) = 4.472

$(\log n)^3$ =>  and n = 20, f(n) = 2.2

$n2^n$ =>  and n = 20, f(n) = 20 * 1,048,576

3^n =>  and n = 20, f(n) = 3,486,784,401

$5^{\log_2 n}$ =>  and n = 20, f(n) = 13.18

logn =>  and n = 20, f(n) = 1.30

We can see that $n\log^2 n$ gives smaller numbers than $n^{1.01}$ until n = 10. But if we increase our index further, $n\log^2 n$ growth rate shows it's difference and when we give n = 1000, $n\log^2 n$ gives a value that almost 9 times bigger than $n^{1.01}$ value. Because of that I am considering n_0 as 10. Efficiency with larger numbers is more important.

<u>Order</u>

$n_0 = 10$;

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

<u>Part 4</u> (I assumed that arrayList with n element was taken as parameter)

<u>Find the minimum-valued item</u>

1)**INIT** n with arraylist's size **Θ(1)**

2)**INIT** lowestNumber with the first element of arrayList **Θ(1)**

3)**INIT** index with 1 **Θ(1)**

4)**WHILE** index is not equal to n **Θ(n*(1+1+1)) = Θ(n)**

5) **IF** arraylist element at index is lower then lowestNumber **THEN** **Θ(1)**

6) **ASSIGN** arraylist element into lowestNumber **Θ(1)**
7) **INCREMENT** index by 1 **Θ(1)**
8)**ENDWHILE**
9)**End**

Time Complexity = Θ(n)
1-3) Constant time = Θ(1)
4-8) Linear Time = Θ(n*(1+1+1)) = Θ(n)
    5) Constant time = Θ(1)
    6) Constant time = Θ(1)
    7) Constant time = Θ(1)
9) End

Find the median item

INIT n with arrayList's size **Θ(1)**
INIT lowestNumber **Θ(1)**
INIT assignIndex **Θ(1)**
INIT median **Θ(1)**
FOR index1 = 0 to n **Θ((n)(n+1)/2)** // sorting
    SET lowestNumber to arrayList element at index1 **Θ(1)**
    FOR index2 = index1 to n **Θ(n-index1)**
        IF arrayList element at index2 is lower than lowestNumber **THEN Θ(1)**
            ASSIGN arrayList element at index2 to lowestNumber **Θ(1)**
            ASSIGN index2 to assignIndex **Θ(1)**
        END IF
    ENDFOR
    ASSIGN arrayList element at index1 to arrayList with index of assignIndex **Θ(1)**
    ASSIGN lowestNumber to arrayList with index of index1 **Θ(1)**
ENDFOR
FOR index=0 to n **O(n/2) or simply O(n)**
    IF n is an odd number and index is equal to n/2 **THEN Θ(1)** (which is median)
        ASSIGN arrayList element with current index to median **Θ(1)**
    ELSE IF an is an even number and index is equal to (n-1)/2 **THEN Θ(1)** (left number)
        ASSIGN sum of two arrayList element with current index and current index + 1 divided by two to median **Θ(1)**
ENDFOR
RETURN median **Θ(1)**

Time Complexity = Θ((n)(n+1)/2) simplified as Θ(n^2)

Nested for loop always runs (n)*(n+1)/2 times since there is no return or break statement. And second for loop which responsible for finding median always runs n/2 times. Sum of these two is unnecessary since first loop increases quadreticly.

Find two elements whose sum is equal to a given value

INIT n with the size of arrayList **Θ(1)**
INIT value **Θ(1)**
INIT firstIndex to -1 **Θ(1)**
INIT secondIndex to -1 **Θ(1)**
INIT sum to 0 **Θ(1)**
FOR index1=0 to arrayList size **O((n-1)*(n) / 2)**
    FOR index2=index1+1 to arrayList size **O(n-index1)** **// -1 is ignored**
        ASSIGN sum of arrayList elements at index1 and index2 into **sum** **Θ(1)**
        IF sum is equal to value **THEN** **Θ(1)**
            ASSIGN index1 to firstIndex **Θ(1)**
            ASSIGN index2 to secondIndex **Θ(1)**
            EXIT inner for loop **Θ(1)**
        ENDIF
    ENDFOR
    IF firstIndex is not equals to -1 **THEN** **Θ(1)**
        EXIT outer for loop **Θ(1)**
    ENDIF
ENDFOR
IF firstIndex is not equal to -1 **THEN** **Θ(1)**
    PRINT arrayList element at firstIndex **Θ(1)**
    PRINT arrayList element at secondIndex **Θ(1)**
ELSE **Θ(1)**
    PRINT “There is no two elements that sum of both equals to given value” **Θ(1)**
End

Time Complexity = O((n-1)*n / 2) or more simply O(n^2)

1-5) **Θ(1)**

6-18) **Worst Case : Θ((n-1)*n / 2) // Happens when no valid sum is found**

Best Case : Θ(1) // Happens when sum is found in first try index1=0 and index2=1

Big O : O ((n-1)*n / 2) and simplified version O(n^2)

7-14) **Worst Case : Θ(n-index1) // Happens when no valid sum is found**

Best Case : Θ(1) // Happens when sum is found in first try index2=index1+1

Big O : O (n-index1) and can be simplified as O(n)

19-24) **Θ(1)**

Merge Two ArrayList by increasing order

INIT an empty newArrayList **Θ(1)**
INIT lowestNumber **Θ(1)**
INIT assignIndex **Θ(1)**
INIT k with size of arrayList1 **Θ(1)**
INIT m with size of arrayList2 **Θ(1)**
INIT n with sum of k and m **Θ(1)**
FOR index1=0 to k **Θ(k)**
    ADD arrayList1 element with index of current index1 value into newArrayList **Θ(1)**
FOR index2=0 to m **Θ(m)**
    ADD arrayList2 element with index of current index2 value into newArrayList **Θ(1)**
FOR newArrIndex=0 to n **Θ((n)(n+1)/2)** // sorting
    SET lowestNumber to newArrayList element with index of newArrIndex **Θ(1)**
    FOR newArrIndex2 = newArrIndex to n **Θ(n-newArrIndex)**
        IF newArrayList element with index of newArrIndex2 is lower than lowestNumber **THEN** **Θ(1)**
            ASSIGN newArrayList element with index of newArrIndex2 into lowestNumber **Θ(1)**
            ASSIGN newArrIndex2 into assignIndex **Θ(1)**
    ASSIGN newArrayList element at newArrIndex to newArrayList with index of assignIndex **Θ(1)**
    ASSIGN lowestNumber to newArrayList with index of newArrayIndex **Θ(1)**
    ENDFOR
ENDOR
End

Time Complexity : Θ((n)(n+1)/2 + k + m) or Θ(n^2)

Here k+m is equal to n since we declare n in the algorithm as k+m, because of that we can write time complexity as ((n)(n+1)/2 + k + m) **=>** Θ((n)(n+1)/2 + n) **=>** Θ((n^2+n)/2 + n) **=>** Θ(n^2) . The simplest representation may not be da most precise one but it is easy to follow.

In the algorithm I first added arrayList1 elements and arrayList2 elements to the new arrayList in index order. After that I used same algorithm in **finding median(only sorting algorithm)** to sort this arrayList.

Part 5

a)

	Time Complexity	Space Complexity
int p_1 (int array[]):	Θ(1)	O(1)

{

return array[0] * array[2]) O(1) Θ(1)

}

For a, there is no allocation so space complexity is O(1) and this code segment always runs in a constant time. Because of that theta notation of the code is Θ(1)

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

Time Complexity	Space Complexity
Θ(n)	O(1)

{

Int sum = 0 Θ(1) O(1)

O(1) Θ(n) for (int i = 0; i < n; i=i+5)

sum += array[i] * array[i]) Θ(1)

return sum Θ(1)

}

For b, there is no allocation either so space complexity is O(1). For loop always runs n/5 times and we can simplify Θ(n/5) as Θ(n) c)

Time Complexity: $\Theta(n*\log n)$ | Space Complexity: $O(1)$

```
void p_3 (int array[], int n):
{
    for (int i = 0; i < n; i++)
        for (int j = 1; j < i; j=j*2)
            printf("%d", array[i] * array[j])
}
```

Outer loop: $\Theta(n*\log n)$, $O(1)$; inner loop: $\Theta(\log n)$; printf: $\Theta(1)$

For c, printf call doesn't allocate memory, it uses already allocated memories' values. Inner for loop always runs logn times and because of that we need to write it's time complexity with theta notation which is Θ(logn). Then, outer loop always runs n times and since inner loop runs logn times, we multiply them. Finally outer loop time complexity becomes Θ(nlogn)

d)

Time Complexity:
Best Case : $\Theta(n)$
Worst Case : $\Theta(n+n*\log n)$
Big O : $O(n\log n)$, for $n_o = 10$
Omega : $\Omega(n)$

Space Complexity: $O(1)$

```
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))
}
```

If block: $\Theta(n+n*\log n)$, $O(1)$; If condition: $\Theta(n)$; p_3(array, n): $\Theta(n*\log n)$

Else block: $\Theta(n)$, $O(1)$; p_1(array): $\Theta(1)$; p_2(array, n): $\Theta(n)$; printf: $\Theta(n)$

For d, the code segment has best cases and worst cases. Worst case happens when first if condition holds, and best condition happens when else statement holds. First, if condition calls p_2 method which runs with time complexity Θ(n) and inside of if statement code calls p_3 method which runs with time complexity Θ(nlogn).

Then, else statement calls p_1 and p_2 methods which runs with Θ(1) and Θ(n) time complexity.
since logn is below 1 while n smaller than 10, big O notation doesn't hold since even with best case algorithm runs with Θ(n) time complexity. Because of that initial n value must be equal or greater than 10 in order to get proper big O notation.