GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 2 Report

Muhammed Bedir ULUCAY

1901042697

Part 1

1. Search Product:

```
/**<p>Query furniture is in branch or not using linear search</p>*/
public boolean queryProduct(Furniture obj){

    if(obj == null)
        return false;
    for(int i=0; i<productNumber; ++i)
        if(products[i].equals(obj))
            return true;
    return false;
}
```

figure 1.1

We got an Furniture[] products let say its lenght is **n**. We use linear search for find wanted search

If its not include wanted furniture that means loop work for n time. **T(n) -> Θ(n)**
If the wanted furniture is first product that means loop work for just 1 time. **T(n) -> Θ(1)**
If the wanted funiture is last element that means loop work for just n time. **T(n) -> Θ(n)**

Best time complexity Θ (1) worst time complexity Θ(n) **this time complexity for T(n) = O(n)**

```
@Override
public boolean equals(Object o) {

    if(!(o instanceof OfficeChair ))
        return false;

    OfficeChair  obj = (OfficeChair ) o;
    return (obj.getModel() == this.getModel() && obj.getColor() == this.getColor());
}
```

figure 1.2

This function just include constant time complexity Θ (1) + Θ (1) + ... = Θ (1)
For this equals function T(n) = Θ (1)

Admin & Customer:

These users can see all products in all branch.

```
/**<p>It search for the wanted furniture in all branch using linear search</p>*/
public Branch queryProduct(Furniture obj) {
    for(int i = 0; i<currentBranchNumber; ++i) {
        if(branchs[i].queryProduct(obj))
            return branchs[i];
    }
    return null;
}
```

figure 1.3

Admin and customer can search a product in all store if accept taht number of branch is **m**:

We see the before **queryProdutc() time complexity = O(n)**
Best case for this functions is ; m = 1, n = 1 ---> T (m , n) = Θ (1)
Worst case for if don’t have furniture and end of product -----> T (m , n) = Θ(m*n)

This function time complexity for T(m, n) = O(m * n)

Employee:

This user can see just own working branch

```
/**<p>Query a product in working branch function</p>*/
public boolean queryProduct(Furniture obj){
    return  workBranch.queryProduct(obj);
}
```

figure 1.4

We see that before queryProduct() in figure1.1 **time complexity = O(n)**

2. Add/ Remove Product:

Add Product:

```
/**<p>Doubling Product array when it full</p>*/
public void enlargeProduct() {

    Furniture[] tmpF = new Furniture[ProductsCapacity*2];

    for(int i=0; i<productNumber; ++i)
        tmpF[i] = products[i];

    products = tmpF;
    ProductsCapacity *= 2;
}
```

figure 2.1

enlargeProduct function **time coplexity is T(n) - > Θ (n)**

```
/**<p>adding Product next free space in products array</p>*/
public boolean addProduct(Furniture newProduct) {

    if(productNumber >= ProductsCapacity)
        enlargeProduct();

    products[productNumber++] = newProduct;
    return true;
}
```

figure 2.2

If there is enough space to add best case is constant **Θ (1)**
If array is full we need to enlarge and we see enlargeProduct time compleity is Θ (n)

These means we have a condition for to be **Θ (n)**

complexity for **addProduct time complexity is O (n)**

Remove Product:

```java
/**<p>Removing product using array shifting method</p>*/
public boolean removeProduct(int index) {

    if(index >= productNumber)
        return false;

    while(index < productNumber)
        products[index] = products[++index];
    productNumber--;
    return true;
}
```

figure 2.3

Best case for this method is last element removing it last element it constant time for this its -> Θ(1)
Worst case is removing first element that happend you need to shift all array its time comp. -> Θ(2)

According to the removing status it **time complexity for this method is O(n)**

3.Query Product

Admin:

```java
public void querySupplyFurniture() {

    for(int i=0; i<getBranchNumber(); ++i) {
        for (Furniture f : allTypeFurnitureArray) {
            if(getBranch(i).queryProduct(f) == false) {
                System.out.println("Need to supplied " + f);
            }
        }
    }
}
```

figure 3.1

We mention that queryProduct() complexity in figure 1.1 method complexity is -> **O(n)**

Let accepts number of branch number is = m and all type furniture number is = p
This method has 3 nested loop and all complexıty has to complate for thıs way

functıon tıme complexıty T(n, m, p) = Θ(n * m * p) absulate value

Employee:

```java
public void querySupplyFurniture() {

    for (Furniture f : allTypeFurnitureArray) {
        if(workBranch.queryProduct(f) == false) {
            System.out.println("Need to supplied " + f);
        }
    }
    
}
```

figure 3.2

We mention that queryProduct() complexity in figure 1.1 method complexity is -> **O(n)**

Let accepts all type furniture count is = m
This method has 2 nested loop and all complexıty has to complate for thıs way

functıon tıme complexıty T(n, m) = Θ(n * m) absulate value

Part 2

A)

Big-O gives upper bound, or maximum runnnig time complexity of an algorithm.

Using “at least” here is not the correct way of using with Big-O notations.

At least gives upper - bound

O(n^2) gives upper – bound

B)

let $h = max(f(n), g(n))$

$f(n) + g(n) < 2h$

so $f(n) + g(n) = O(h)$.

If $f(n) + g(n)$ are positive then

$h \leq f(n) + g(n)$ and so $h = \Theta(f(n) + g(n))$

$max(f(n), g(n))$

figure 2.2

C)

1) if we take the limit to inf

$$\lim_{n \to \infty} \frac{2^{n+1}}{2^n} \Rightarrow \lim_{n \to \infty} \frac{\cancel{2^n} \cdot 2}{\cancel{2^n}} \Rightarrow \lim_{n \to \infty} 2 = 9$$

figure 2.3.1

That means they are equal time complexity

2)

c is constant

$$\ln \quad 2^{2n} \leq c\,2^{n}$$

$$\ln 2 . 2n \leq \ln c + \ln 2 . n$$

$$2n \leq \ln c + n$$

$$n \leq \ln c \quad !$$

figure 2.3.2

Which is clearly wrong because there is no such constant a number that inequelity R = {}

3)

$\underbrace{f(n) = O(n^2)}_{\text{this worstcase}}$ $\quad g(n) = \Theta(n^2)$

don't have the information absolutely like Θ

so if we multiplay with $O * \Theta =$

resut mustbe O and $T(n)$ n's mustbe

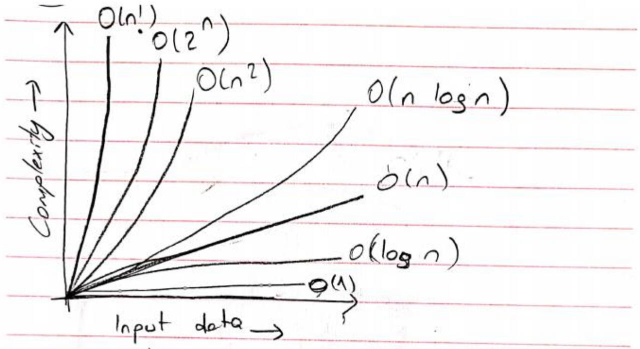
multiply by normal

$f(n) * g(n) \neq \Theta(n^4)$ X

$f(n) * g(n) = O(n^2 * n^2) = O(n^4)$ ✓

Figure 2.3.3

Part 3



As we can see the graph we can seperate de fuctions a couple of group firsty

I. group	II. group	3. group
$\sqrt{n}$	$n^{1.01}$	3^n
$(\log n)^3$	$n.\log^2 n$	$n2^n$
$\log(n)$		2^{n+1}
$5^{\log_2 n}$		2^n
$5^{\log_2 n} > \sqrt{n} > \log(n)^3 > \log(n)$	$n.\log^2 n > n^{1.01}$	$3^n > n2^n > 2^{n+1} > 2^n$

according to the ab sulute value sorted:

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

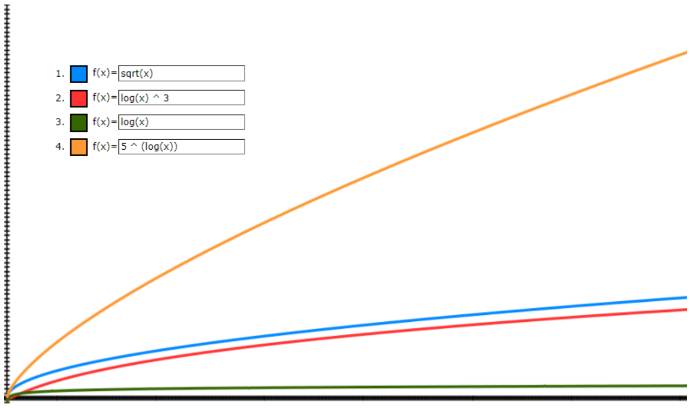
But if we want to sort using Big-O notation ve can ignore some constant and resut will be like this

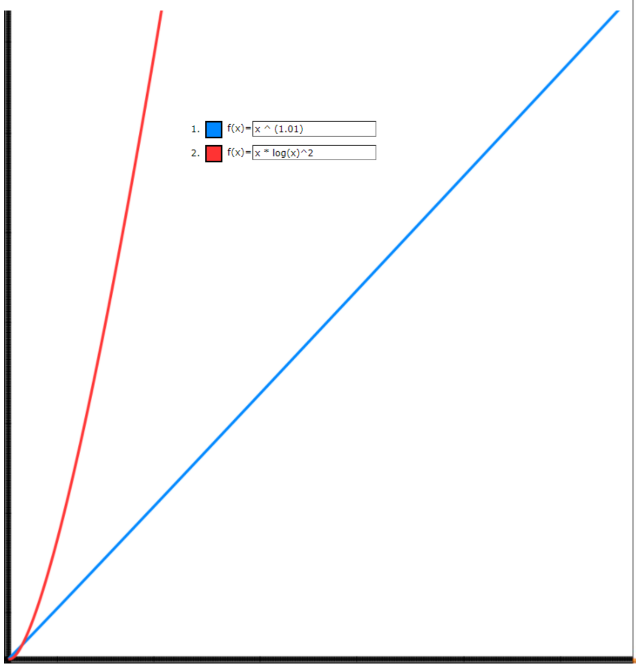
$$O(3^n) > O(n2^n) > O(2^n) = O(2^n) >$$
$$O(n\log^2 n) = O(n^{1.01}) > O(5^{\log_2 n})$$
$$> \sqrt{n} > \log(n)^3 > \log(n)$$

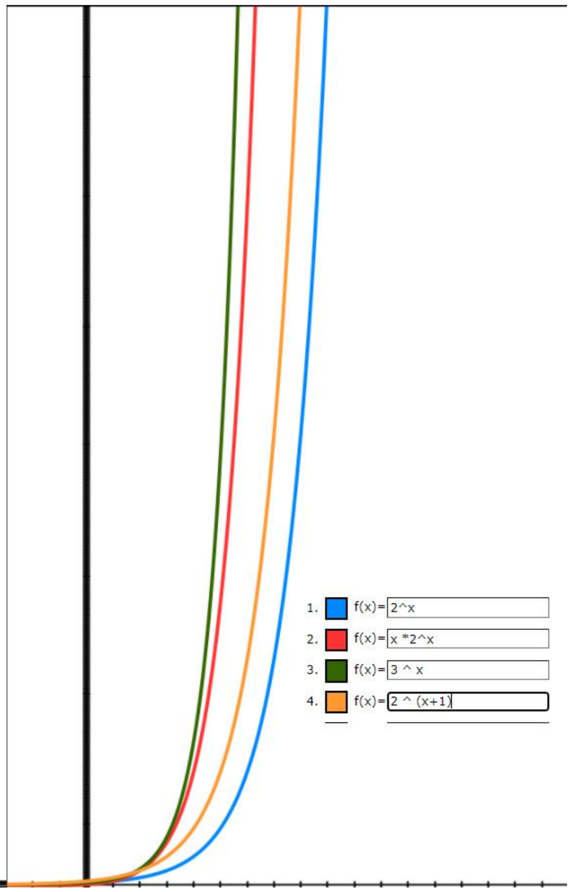
$$\lim_{n\to\infty} \frac{2^n}{2^{n+1}} = 0 \checkmark$$

$$\lim_{n\to\infty} \frac{n\log^2 n}{n^{1.01}} = 0 \checkmark$$

Because of these there are two equality in asymtotic ordering


1. f(x)= sqrt(x)
2. f(x)= log(x) ^ 3
3. f(x)= log(x)
4. f(x)= 5 ^ (log(x))


1. f(x)= x ^ (1.01)
2. f(x)= x * log(x)^2


1. f(x)= 2^x
2. f(x)= x *2^x
3. f(x)= 3 ^ x
4. f(x)= 2 ^ (x+1)

Part 4

A)

```
min_function( arrList )

    int min = arrList[0]

    for ( element : arrList )
        if min < element
            min = element        // If there is smaller than current number
                                 // take it as a new smallest number
    return min
```

figure 4.1

We are looking all element to find smallest element.It always time complexity is **-> Θ(n)**

B)

```
median_function( arrList )

    if arrList.length % 2 == 0 :
        boolean r1 = r2 = false
        int sum = 0
        for i = 0 ; i< arrList.length; ++i :

            int smallCounter = bigCounter = 0
            for j = 0; j<arrList.length; ++j

                if arrList[j] > arrList[i] :
                    bigCounter++

                else if arrList[j] < arrList[i] :
                    smallCounter++

            if bigCounter - smallCounter == 1
                r1 = true
                sum += arrList[i]
            if smallCounter - bigCounter == 1
                r2 = true
                sum += arrList[i]

            if r1 && r2
                int median = sum / 2
                return median
```

figure 4.2.1

This is work for array length is even because if array lenght is even there is two median value and return it avg of two values.

Algorithm is we take an number arrList[i] and counting smaller numbers and bigger numbers.
if sCounter – lCounter is equal is 1 or vise vierce to left side of equality (lCounter – sCounter == 1)

Best case is 2n for the method = **Θ** (2n)
Worst case is n^2 = **Θ** (n^2)
Time Complexity for the even arrayList length is - > O (n^2)

```
    if arrList.length % 2 != 0 :

        for i=0 ; i<arrList.length; ++i :

            int smallCounter = bigCounter = 0
            for j=0; j<arrList.length; ++j

                if arrList[j] > arrList[i]
                    bigCounter++

                else if arrList[j] < arrList[i]
                    smallCounter++

            if smallCounter == bigCounter
                int median = arrList[i]
                return median

```

figure 4.2.2

This is work for array length is odd there is absulute value

Algorithm is we take an number arrList[i] and counting smaller numbers and bigger numbers.
If sCounter == bCounter that means we find the median value of array

Best case is 2n for the method = **Θ** (n)
Worst case is n^2 = **Θ** (n^2)
Time Complexity for the odd arrayList length is - > O (n^2)
median_function() time complexity T(n) -> O(n^2)

C)

```

sum_function(  arrList , value):

    map<Integer,Integer> myMap

    int result[] = new int [2]
    for i = 0 ; i< arrList.length ; ++i:

        if(myMap.containsKey(value - arrList[i])
            // Look for the need number to reach target
            // is in the list or not if it in list take it using get
            return {myMap.get(value - arrList[i]) , i}

        myMap.put(arrList[i], i) //storeing data to reach if need

    return result

```

figure 4.3

We could use onether way : iterate over list for each elemet as : arr[i] + arr[j] == value if we use this algorithm our time complexity wolud be O(n^2)

But in this way we decrease the complexity using map data structur to reach consider the contains method and **our algorithm time complexity O(n) ignoring constant time operations**

D)

```
LinkedList merge_two_sortedArr_getLinkedList( arrL_1, arrL_2 )

    LinkedList<type> ll = new LinkedList<>()

    for i = 0, j = 0 ; i < arrL_1.size() && j < arrL_2.size(); :
        // Getting number respectively dont mess up sorted
        // until one of reach of the array size
        if arrL_1.get(i) < arrL_2.get(j) :
            ll.add(arrL_1.get(i))
            i++
        else :
            ll.add(arrL_2.get(j))
            j++


    // One of the while is gonna work because
    // one of the counter reach the array size in for loop
    while(i < arrL_1.size())
        ll.add(arrL_1.get(i++))

    while(j < arrL_2.size())
        ll.add(arrL_2.get(j++))

    return ll
```

figure 4.4

if we accepts sizes as arr1.size is -> n and arr2.size -> m

Fisrt for loop wil work time complexity is **if n < m -> Θ(2n)**
else Θ(2m)

And then it wil work in while loop **if n > m - > Θ(n - m)**
else Θ(m - n)

Ll.add method is working time T(p) - > Θ(1) and **this is gonna work n+m times** to add all element

Our time complexity will gonna be -> Θ((n+m) * 1) = Θ(n+m)

Note : we ignore the constant operations.

Part 5

A)

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

figure 5.1

All operations of functions is completing in constant time T(n) = Θ(1)

B)

```
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
}
```

Figure 5.2

In sum = 0 & sum += arr[i] + arr[i] & return sum is completing in constant time Θ(1)

But for loop work n time that means time complexity is Θ(n / 5) = Θ(n)

T(n) - > Θ(n) + Θ(1) = Θ(n)

C)

```
c)
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])
}
```

Figure 5.3

printf ( ) $\Rightarrow \Theta(1)$

for (j=1 ; j< i ; j = j*2) $\Rightarrow$

$\Rightarrow$ 1, 2, 4, 8, 16

loop will gonna stop

log / $2^k > n$

$k > \log n - 2$

There fore, the number of iterations is only $O(\log n)$

so the total complexity is $O(\log n)$ for 2th loop

for (i=0 ; i<n ; ++i) $\Rightarrow \Theta(n)$ for 1th loop

so the total complexity is

$\Theta(1) * \Theta(n) * O(\log n) \Rightarrow O(n \log n)$

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

figure 5.3.2

O (n * log n)

D)

```
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))
}
```

figure 5.4

Firstly let do the easy one if the condition is not true in if statment is

Best case : T(n) = Θ(1)

If the if condition is true it would take alot time than else statment

Worst case when the condition is true :
as we calculate the figure 5.2 p_2() time complexity is **T(n) - > Θ(n)**

p_3() time complexity is in figure 5.3.2 complexity is **T(n) -> O (n * log n)**

time complexity is

T(n) - > Θ(n) + O (n * log n) = O (n * log n) (we just take dominant term)

P_4() complexity **is T(n) - > O (n * log n)**

Muhammed Bedir ULUCAY

1901042697

CSE 222 HW2