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

Can Duyar
171044075

I) Searching a Product

```
public Void searchProducts (String product_name, String color, String model_name){
T1 [ boolean search_control = false; ⟶ Θ(1)
   for (int t=0; products[t] != null; t++){
      if(products[t].getProduct().equals(product_name) && products[t].getModel().
      equals(model_name) && products[t].getColor().equals(color)){
        if(products[t].getNumberInStock() > 0){
T2        System.out.println("Furniture that you searched was founded!!!(\n");
          System.out.println("Product: " + products[t].getProduct());
          System.out.println("Color: " + products[t].getColor());
          System.out.println("Model:" + products[t].getNumberInStock());
          System.out.println("\n");
          search_control = true;
          break;
        }
      }
   }
   if(search_control == false) ⟶ Θ(1)
T3    system.out.println("We couldn't find the furniture that you searched :(");→ Θ(1)
}
```

Θ(1)

Analyze the time complexity

$T_1(n) = \Theta(1)$ — constant time.

for T_2;

There is a "for" loop and two "if" statements in this loop. First, we should calculate the time complexity of these "if" statements,

for the first "if" statement, time complexity can have two cases. They are best and worst cases, because there is "equals" method in this "if" statement.

best case = Θ(1)
worst case = Θ(n) } ⟶ O(n), Ω(1)

for the second "if" statement, time complexity should be Θ(1)

If we analyze the time complexity of "for" loop, it can have two cases as best and worst cases,

$T_2(n)$(worst) = Θ(n) * Θ(n) = Θ(n²) — couldn't find or found in last one
$T_2(n)$(best) = Θ(1) * Θ(1) = Θ(1) — found in first one

} ⟶ $T_2(n) = O(n^2)$ (O(n²), Ω(1))

for T_3;

$T_3(n) = \Theta(1)$ → constant time.

Result ⟹ Time complexity = $T(n) = T_1(n) + T_2(n) + T_3(n)$
$= \Theta(1) + O(n^2) + \Theta(1) = O(n^2)$

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

II) Add Product

```
Public void addProduct(Furniture fur, int number_add){
  if(number_add>0 && furnitureNum>0){
T1 [ int control=-1;                                   → Θ(1)
   [ for(int t=0; t<furnitureNum; t++){                → Θ(1)
     [   if(Products[t].getProduct()==fur.getProduct()&& Products[t].getModel()=fur.getModel()){
T2   [     control=t;
     [     furnitureNum+=1;                            ] Θ(1)
     [     break;
     [   }
     [ }
     [ if(control!=-1){
T3   [   Products[control].setNumberInStock(Products[control].getNumberInStock()+number_add);
     [ }
     [ else if(control==-1){
     [   system.out.println("There is no furniture that you want to add");
     [ }
   }
  }
```

$T_{1(n)} = \Theta(1)$ → constant time.

for T_2

There is just one "if" statement in a for loop, "if" statement will have $\Theta(1)$ as time complexity but for calculating the time complexity of "for loop", there are two cases as worst and best cases.

$T_{2(n)}$ (worst) $= \Theta(n)$ → loop turns until the value of furnitureNum

$T_{2(n)}$ (best) $= \Theta(1)$ → loop turns just one time.

$\Theta(n) \rightsquigarrow O(n)$, $\Omega(1)$

$T_{2(n)} = O(n)$

for T_3

There are one "if" and one "else if" statements, they will have time complexity as $\Theta(1)$,

Result → `if(....){` → $\Theta(1)$ ... `}`

$T_1 + T_2 + T_3 = \Theta(1) + O(n) * \Theta(1) + \Theta(1) = O(n)$

Time Complexity $= T(n) = O(n)$ //

III) Remove Product

```
public void removeProduct (Furniture fur, int number_remove){
  if(number_remove > 0 && furnitureNum > 0){        → Θ(1) → constant time for "if" statement.
T1 [ int control = -1;
     for(int t=0; t<furnitureNum; t++){
       if(products[t].getProduct()== fur.getProduct() && products[t].getModel() == fur.getModel())
         {
T2         control = t;
           break;
       }
     }
T3   if(control != -1){
       products[control].SetNumberInStock(products[control].getNumberInStock()-number_remove);
     }
     else if (control == -1){
       system.out.println("There is no furniture that you want to remove");
     }
  }
}
```

Analyze the time complexity → Solution is same with "add product" method, solution steps are same, there are just small changes that are not affect the time complexity

$T_1(n) = \Theta(1)$ → constant time.

for T_2

There is just one "if" statement in a for loop, "if" statement will have $\Theta(1)$ as time complexity but for calculating the time complexity of "for loop", there are two cases as worst and best cases.

$T_2(n)(worst) = \Theta(n)$ → loop turns until the value of furnitureNum.

$T_2(n)(best) = \Theta(1)$ → loop turns just one time

$\Theta(n)$ ~ $\Theta(1)$ → $O(n)$

$T_2(n) = O(n)$

for T_3

→ There are one "if" and one "else if" statements, they will have time complexity as $\Theta(1)$,

Result → `if(....){` ⋮ `}` → $\Theta(1)$

$T_1 + T_2 + T_3 = \Theta(1) + O(n) * \Theta(1) + \Theta(1) = O(n)$

Time Complexity	$= T(n) = O(n)$

IV) Querying the Products that need to be supplied.

```
Public void productQuery(){
T1 [ boolean queryCheck=false;                                  → Θ(1)
   [ for(int t=0; t<furnitureNum; t++){
     if(products[t].getNumberInStock()==0){                     → Θ(1)
        System.out.println("Product: "+products[t].getProduct());
        System.out.println("Color: "+products[t].getColor());
T2      System.out.println("Model: "+products[t].getModel());
        System.out.println("Number of stock: "+products[t].getNumberInStock());
        System.out.println("\n");
        queryCheck=true;
     }
   }
T3 [ if(queryCheck==false){
     System.out.println("There is no furniture that need to be supplied...");
   }
}
```

Analyze the time complexity

$T_1(n) = \Theta(1)$ → constant time.

for T_2

There is just one "if" statement in a for loop, "if" statement will have $\Theta(1)$ as time complexity but for calculating the time complexity of "for loop" we should think the value of "n".

So, $T_2(n) = \Theta(n)$ → because loop turns until the value of "n"

for T_3

→ There is just one "if" statement and it has $\Theta(1)$ as time complexity so,

$T_3(n) = \Theta(1)$

Result → Time Complexity = $T(n) = T_1(n) + T_2(n) + T_3(n) = \Theta(1) + \Theta(n) + \Theta(1)$

$\boxed{T(n) = \Theta(n)}$ //

Part 2

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

Big-O → asymptotic upper bound.
* we can not say "at least" for the Big-O because "at least" means asymptotic lower bound Ω, for this reason, it's meaningless to say.

b) Let $f(n)$ and $g(n)$ be non-decreasing and non-negative functions. Prove or disprove that: $\max(f(n), g(n)) = \Theta(f(n) + g(n))$

$$0 \le c_1 n \le \max(f(n)+g(n)) \le c_2 n$$

$$0 \le c_1(f(n)+g(n)) \le \max(f(n)+g(n)) \le c_2(f(n)+g(n))$$

$$f(n)+g(n) \ge 0$$

$$f(n)+g(n) \ge \max(f(n)+g(n))$$

$$\frac{1}{2}(f(n)+g(n)) \le \max(f(n)+g(n))$$

$$\frac{1}{2}(f(n)+g(n)) \le \max(f(n)+g(n)) \le (f(n)+g(n))$$

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

$\Theta(f(n)+g(n))$

$\Rightarrow$ Therefore, $\max(f(n)+g(n)) = \Theta(f(n)+g(n))$ is provided. ✓

c) Are the following true? Prove your answer.

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

$$c_1 \cdot n \le 2^{n+1} \le c_2 \cdot 2^n$$

$$2^{n+1} \le c_2 \cdot 2^n$$

$$2^{n+1-n} \le c_2 \cdot 2^{n-n}$$

$$2^1 \le c_2$$

$$c_1 n \le 2^{n+1}$$

Therefore, $2^{n+1} = \Theta(2^n)$ → so it's true //

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

$c_1 n \leq 2^{2n} \leq c_2 \cdot 2^n$

$2^{2n} \leq c_2 2^n$

$2^{2n-n} \leq c_2 \cdot 2^{n-n}$ → because c_2 must be constant.

$2^n \neq c_2$

Therefore, $2^{2n} \neq \Theta(2^n)$ → $2^{2n} = \Theta(2^n)$ is false.

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

→ then we can say that, $g(n) = O(n^2)$ and $g(n) = \Omega(n^2)$

$f(n) \leq cn^2$

$g(n) \leq cn^2$

$f(n) * g(n) \leq cn^4$ ← c must be constant

Big-O → asymptotic upper bound.

This is also same and we can say that if $f(n) * g(n) = \Theta(n^4)$ then $f(n) * g(n) = O(n^4)$ and $f(n) * g(n) = \Omega(n^4)$

Therefore, we can say that,

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

$f(n) * g(n) = O(n^4)$ and $f(n) * g(n) = \Omega(n^4)$

are provided ✓

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}, $5^{\log_2 n}$, $\log n$

Solution

2^n
$n \cdot 2^n$
3^n
2^{n+1}
$5^{\log_2 n}$
→ exponential

Sorting of exponential function

$3^n > n \cdot 2^n > 2^{n+1} = 2^n > 5^{\log_2 n}$ → $5^{\log_2 n}$ is smaller than others because of logarithmic exponent.

$\lim_{n\to\infty} \frac{3^n}{2^n} = \infty$ so $3^n > 2^n$ ↝ 2^{n+1} and 2^n's growth rates are same

$(\log n)^3$
$\log n$
→ Logarithmic

Sorting of Logarithmic functions

$(\log n)^3 > \log n$

↳ it's greater, because $\lim_{n\to\infty} \frac{(\log n)^3}{\log n} = \infty$

$n \cdot \log^2 n$
$n^{1.01}$
n
$\sqrt{n}$
→ Polynomial

Sorting of Polynomial functions

$n^{1.01} > n \cdot \log^2 n > \sqrt{n}$

It's smaller than $n^{1.01}$ because of logarithmic coefficient.

$n^{1.01} > n^{0.5}$
$n > n$
so $n^{1.01}$ is greater than $n^{0.5}$

As a Result

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

exponential sorting

Logarithmic sorting.

$\lim_{n\to\infty} \frac{n^{1.01}}{\sqrt{n}} = \infty$ so $n^{1.01} > \sqrt{n}$

$\lim_{n\to\infty} \frac{n \cdot \log^2 n}{\sqrt{n}} = \infty$ so $n \cdot \log^2 n > \sqrt{n}$

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.

```
minimum ← arrList.get(0)                    } T1
T2 { FOR i ← 1 TO n
     IF arrList.get(i) < minimum THEN       } T3
        minimum ← arrList.get(i)
     ENDIF
   ENDFOR
print minimum
```

Time Complexity

T1

It has $\Theta(1)$ constant time.

T2

It has $\Theta(n)$ as time complexity because loop turns until the value of "n" in every case.

$\because T_2(n) = \Theta(n)$

T3

If statement is $\Theta(1)$ and also "minimum ← arrList.get(i)" has $\Theta(1)$

So, $T_3(n) = \Theta(1) + \Theta(1) = \Theta(1)$

$T_3(n) = \Theta(1)$

Time Complexity $= T_1(n) + T_2(n) * T_3(n) = \Theta(1) + \Theta(n) * \Theta(1)$

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

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

```
FOR i ← 0 TO n                                   ] T0
    arrList_temp.add(arrList.get(i))
ENDFOR
FOR t ← 0 TO n                                   ] T1
  FOR k ← t+1 TO n                  → Θ(1)
    IF arrList.get(t) > arrList.get(k)
      temp ← arrList.get(t)
      arrList.get(t) ← arrList.get(k)   → Θ(1)
      arrList.get(k) ← temp
    ENDIF
  ENDFOR                                         ] T2
ENDFOR
IF arrList.size() is odd
    median ← arrList.get((arrList.size()+1)/2 - 1)   ] T3
ELSE
    median ← arrList.get(arrList.size()/2)
ENDIF
FOR g ← 0 TO n                                   ] T4
IF arrList_temp.get(g) == median
    print median
ENDIF
ENDFOR
```

T0
"for" Loop turns until the value of n, so it has Θ(n).
For "add" method in for loop there are two cases → best case → Θ(1), worst case → Θ(n) → O(n), Ω(1)
$T_0(n) = \Theta(n) * O(n) = O(n^2)$

T1
It's time complexity for outer "for" loop.
$T_1(n) = \Theta(n)$ ⇝ loop turns until the value of n

T2
It's time complexity for inner "for" loop.
$T_2(n) = \Theta(n)$ ⇝ loop turns until the value of n

T3
"size" method is Θ(1) in arrayList.
"get" method is also Θ(1) So time complexity has Θ(1) for "if" "else" statements.

T4
"For" loop has Θ(n) as time complexity because loop turns until the value of n
"if" statement has Θ(1) as time complexity.

Result = Time Complexity $= T(n) = T_0(n) + T_1(n) * T_2(n) + T_3(n) + T_4(n) = O(n^2) + \Theta(n) * \Theta(n) + \Theta(1) + \Theta(n) * \Theta(1)$

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

- Find two elements whose sum is equal to "given value"

```
FOR i←0 TO n                                                    T1
{ FOR t←i+1 TO n                                                T2
    IF arrList.get(i) + arrList.get(t) is equal to "given_value" THEN   T3
       Print arrList.get(i)
       Print arrList.get(t)
    ENDIF
  ENDFOR
ENDFOR
```

Time Complexity

T1

outer "for" loop turns until the value of "n" in every case, so time complexity of this "for" loop

$T_1(n) = \Theta(n)$

T_2

inner "for" loop turns until the value of "n" in every case, so time complexity of this "for" loop,

$T_2(n) = \Theta(n)$

T_3

If statement has $\Theta(1)$ and also "print parts" have $\Theta(1)$

so, $T_3(n) = \Theta(1) + \Theta(1) = \Theta(1)$

$T_3(n) = \Theta(1)$

Time Complexity $= T_1(n) * T_2(n) * T_3(n) = \Theta(n) * \Theta(n) * \Theta(1)$

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

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

```
t ← 0
k ← 0

WHILE t<n and k<n                                   } T1
  IF arrList1.get(t) < arrList2.get(k) THEN         } T2
    arrList3.add(arrList1.get(t++)                  } O(n)
  ELSE
    arrList3.add(arrList2.get(k++))
  ENDIF
ENDWHILE

WHILE t<n
  arrList3.add(arrList1.get(t++))                   } T3
ENDWHILE

WHILE k<n
  arrList3.add(arrList2.get(k++))                   } T4
ENDWHILE
```

because of "add" method of the array list.

Time Complexity

T_1

"while" loop turns until the value of "n" in every case so time complexity of "while" loop is:

$T_1(n) = \Theta(n)$

T_2

There are two cases for T_2 as best and worst cases. because of "add" method.

$T_2(n)(worst) = \Theta(1) + \Theta(n) = \Theta(n)$

$T_2(n)(best) = \Theta(1) + \Theta(n) = \Theta(n)$

$\rightarrow O(n)$ - $\Omega(n)$

T_3

There are two cases for T_3 as best and worst cases, because of "add" method.

$T_3(n)(worst) = \Theta(n) * \Theta(n) = \Theta(n^2)$

$T_3(n)(best) = \Theta(n) * \Theta(1) = \Theta(n)$

T_4

There are two cases for T_4 as best and worst cases.

$T_4(n)(worst) = \Theta(n) * \Theta(n) = \Theta(n^2)$

$T_4(n)(best) = \Theta(n) * \Theta(1) = \Theta(n)$

Result:

Time Complexity $= T(n) = T_1(n) * T_2(n) + T_3(n) + T_4(n)$

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

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

Part 5:

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

a)

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

→ $\Theta(1)$

Time Complexity

$T(n) = \Theta(1)$ → because there is just one "return", nothing else

Space Complexity

$S(n) = \Theta(n)$ → because of array's size

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
}
```

$\Theta(1)$ → constant time

T_1 [Int sum = 0

loop turns until the value of n so, $T_2(n) = \Theta(n)$

T_2 [for (int i = 0; i < n; i=i+5) sum += array[i] * array[i]

return sum] $T_3 \rightsquigarrow \Theta(1)$

Time Complexity

$T(n) = T1 + T_2 + T_3$

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

$= \Theta(n)$

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

Space complexity

for "sum" and "n" they have space complexity as $\Theta(1)$ because they have constant space complexity, but for array[n], it has $\Theta(n)$ as space complexity.

∴ $S(n) = \Theta(n)$ //

c)

```
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])
}
```

for (int i = 0; i < n; i++) → $\Theta(n)$

for (int j = 0; j < i; j=j*2) → $\Theta(\log n)$ → because of $j = j*2$

printf("%d", array[i] * array[j]) → $\Theta(1)$

Time complexity

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

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

Space Complexity

→ for "i" and "n" variables space complexity is $\Theta(1)$ → constant space c.

but array is different because of loop and it has $\Theta(n)$ space - c

∴ $S(n) = \Theta(n)$

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))
}
```

T_1 [If (p_2(array, n)) > 1000) → $\Theta(n)$ because of P_2

T_2 [p_3(array, n) → $\Theta(n \cdot \log n)$ → because of P_3

else

T_3 [printf("%d", p_1(array) * p_2(array, n)) → $\Theta(1) * \Theta(n) = \Theta(n)$ because of P_1 and because of P_2

Time Complexity

Worst Case

$T(n) = T_1(n) + \max(T_2(n), T_3(n)) = \Theta(n) + \Theta(n \cdot \log n) = \Theta(n \cdot \log n)$

best Case

$T(n) = T1(n) + \min(T_2(n), T_3(n)) = \Theta(n) + \Theta(n) = \Theta(n)$

Average Case → $\Theta(n \cdot \log n)$

Space Complexity

for "n" variable, space complexity is constant, it means $\Theta(1)$ but for the "array" it has $\Theta(n)$ because of size,

$S(n) = \Theta(n)$