Course Code: CSE 222
Course Name: Data Structures and Algorithms
Homework No: Homework 2

Student Name: Şeyda Nur DEMIR (GALAR)
Student No : 1210 44 042

Homework 2 Report

It is handwritten report, I will scan and then upload.
I did all part, only my own solutions.
I just use our lecture slide 2, lecture records, and text book.
I did not use any other source.
I explained my all assertions.

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

1. Searching a product (Attach the code of your solution for each part)

→ my solution:

```
int  search_product (int product_ids[], int size, int product_id) {
    for (int i=0; i< size, i++)
        if (product_ids[i] == product_id)    O(1)      O(n) * O(1) = O(n)
            return i                         Ω(1)
    return -1
}
```

- Time complexity: $\Theta(n)$

best case → may find first index → $\Omega(1)$

worst case → could not found, or find last index → $O(n)$

average case → may find first index 1

second " 2

third " 3

⋮

last index n

couldn't find n+1

$$\frac{n\cdot(n+1)}{2\cdot n} = \frac{1}{2}\cdot n + \frac{1}{2} = n$$

drop constants

ignore low order terms

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

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

most appropriate asymptotic notation is theta

$\Theta(n)$

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

II. Add/Remove product (Attach the code of your solution for each part)

→ my solution:

```
void addProduct (KWArrayList<Product> products, Product product) {
    KWArrayList<Integer> indexes = findByProperty(products, 2, product.getCategory());
    if (indexes.isEmpty()) { → O(1)
        products.add(product) → Ω(1) / O(n) (reallocate)
        product.setProductId(products.size()-1) → O(1)
        print("Product added") → O(1)
    } else
        print("Product category already exists") → O(1)
}
```

O(n)

Ω(1)

Ω(1) O(n) Θ(n)

O(n)

Ω(1)

- Time complexity: O(n)

best case → product is already exists, and it is at first index then Ω(1)
then runs else case with Ω(1)
totally Ω(1)

worst case → product is new, indexes is empty then O(n)
runs if case, adds product
but it is full, reallocates then O(n)
totally O(n)

average case → we can not calculate it (I didn't prefer to say Θ(n))
Ω(1) O(n)

most appropriate asymptotic notation is Big O

O(n)

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

II. Add/Remove product (Attach the code of your solution for each part)

→ my solution

```
void removeProduct (KWArrayList<Product> products, Product product) {
    KWArrayList<Integer> indexes = findByProperty(products, 2, product.getCategory());   ↓ Ω(1) O(n) Θ(n)
    if (indexes.isEmpty()) { → O(1)
        print ("Product could not find") → O(1)   } Ω(1)
    } else {
                                   → O(n) Ω(1)
        products.remove(indexes.get(0))
        print ("Product removed") → O(1)
    }
}
```

• Time complexity: O(n)

best case → product doesn't exist, then Ω(1)

worst case → product exist, at last index O(n) / at first index Ω(1)

removes from last Ω(1) / from first O(n)

totally O(n) Θ(n)

average case → we can not calculate it (I didn't prefer to say Θ(n))

Ω(1) O(n)

most appropriate asymptotic notation is Big O

O(n)

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

III. Querying the products that need to be supplied. (Attach your solution)

→ my solution :

```
void informManager (KWArrayList<Product> products,
                    KWArrayList<Integer> productIdsOutStock,
                    int requestedAmount) {
    productIdsOutStock.clear(); → O(1) Ω(1)
                                          → O(n) Ω(n)
    for(int i=0; i<products.size(); i++) {
        if (products.get(i).isOutStock(requestedAmount))   → O(1)
            productIdsOutStock.add(products.get(i).getProductId())
    }                                       → Ω(1)
                                              O(n)
    print ("manager informed")
                                 → O(1)
}
```

- Time Complexity : $\Theta(n)$

best case → adds id only, doesn't reallocate, although its $\Omega(n)$ because of loop

worst case → algorithm loops all cases, it is clearly $O(n)$

average case → we may calculate add probability averages, I didn't proper

we can use the rule

$\Omega(n) = O(n)$ then $= \Theta(n)$

$\Omega(n)$ $O(n)$ $\Theta(n)$

most appropriate asymptotic notation

$\Theta(n)$

Part 2:

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

→ Finding exact running time of algorithms is too hard, sometimes impossible.
So, we explain time complexity using asymptotic notations.
When using asymptotic notations, we drop constnts and ignore low order terms.
Also, we care about only upper bound, lower bound and average analysis.

- Big-O notation gives us upper bound,
  stating a lower bound as Big O abuses the notation.
- Statement says $T(n)$ is at least (n^2),
  means $T(n)$ is upper bound of function $f(n)$,
  since $f(n)$ could be any function smaller than n^2.
  If $T(n) \geq O(n^2)$ than $n^2 > f(n) \geq 0$ (since running time is always non-negative)
- There is no any information about upper bound of $T(n)$,
  and lower bound of $f(n)$ too.
- This statement is true, but completely uninformative.
  Hence, the statement is redundant.

Part 2:

b) Let $f(n)$ and $g(n)$ be non-decreasing and non-negative function. Prove or disprove that:

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

→ Non-decreasing → Each element is equal or bigger than last one.

Non-negative → Each element is 0 or positive.

- Some definitions from a related book (Introduction to Algorithms) our second slide covers them also.
- $O(g(n)) = f(n)$ $0 \leq f(n) \leq c \cdot g(n)$

  $\Theta(g(n)) = f(n)$ $0 \leq c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

  $\Omega(g(n)) = f(n)$ $0 \leq c \cdot g(n) \leq f(n)$

  {∃ positive constants c, c_1, c_2 and for all $n \geq n_0$}
- A theorem and proof from our second slide.

  Theorem: $f(N) = O(g(n)) \iff g(n) = \Omega(f(N))$

  Proof: $f(N) \leq c_1 \cdot g(n) \iff g(n) \geq c_2 \cdot f(N)$

  $1/c_1 \cdot f(N) \leq g(n) \iff g(n) \geq c_2 \cdot f(N)$

  We choose c_2 as $1/c_1$ then theorem is right.
- Also, $T(n) = O(T_{worst}(n)) = \Omega(T_{best}(n))$
- We prove before $\max(f(n), g(n)) \in O(f(n) + g(n))$

  $\max(f(n), g(n)) \in \Omega(f(n) + g(n))$

  then we say; $\max(f(n), g(n)) = \Theta(f(n) + g(n))$
- Note that, $f(n) \leq f(n) + g(n)$ and $g(n) \leq f(n) + g(n)$

  {for all $n \geq n_0$ when $c = 1$ $n_0 = 1$}
- Note that, $f(n) + g(n) \leq 2 \times \max(f(n), g(n))$

  {for all $n \geq n_0$ when $c = 1/2$ $n_0 = 1$}
- Hence, these two statements (and definitions above); prove that, $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

  It is always true.

Part 2:

c) Are the following true? Prove your answer.

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

→ Let look two definition about that

$\Theta(g(n)) = f(n) \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$T(n) = O(T_w(n)) = \Omega(T_b(n))$

Then we can prove that.

- $0 \leq c_1 \cdot 2^n \leq 2^{n+1} \leq c_2 \cdot 2^n$ for all $n \geq n_0$

  when $c_1 = c_2 = 2$ this statement is right.

- Also we can explain it with limit.

  $\lim_{N \to \infty} \frac{f(N)}{g(N)} = c \neq 0 \Rightarrow f(N) = \Theta(g(N))$

  Then we can prove that

- $\lim_{n \to \infty} \frac{2^{n+1}}{2^n} = \lim_{n \to \infty} \frac{2^n \cdot 2}{2^n} = \lim_{n \to \infty} 2 = 2 = c \neq 0$

  when $c = 2$ we can say $2^{n+1} = \Theta(2^n)$ is true.

Part 2:

c) Are the following true? Prove your answer

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

→ We can try taking limit

$\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} \frac{2^n \cdot 2^n}{2^n} - \lim_{n\to\infty} 2^n = \infty$

$\infty \to g(n) = O(f(n))$ we can say.

- $O \to 0 \leqslant 2^{2n} \leqslant c \cdot 2^n$

  $\ln 2 \cdot 2n \leqslant \ln c + \ln 2 \cdot n \to 2n \leqslant \ln c + n \to n \leqslant \ln c$

  (There is no such constant that satisfy such inequality. It is wrong for O)

  $\Omega \to 0 \leqslant c \cdot 2^n \leqslant 2^{2n}$

  $\ln c + \ln 2 \cdot n \leqslant \ln 2 \cdot 2n \to \ln c + n \leqslant 2n \to \ln c \leqslant n$

  (It is possible for all $n \geqslant n_0$ but can not help us, because of wrong for Big O)

  $\Theta \to 0 \leqslant \underbrace{c_1 \cdot 2^n \leqslant}_{\checkmark} \underbrace{2^{2n} \leqslant c_2 \cdot 2^n}_{\times}$

  We can not find any c_1, c_2 constants such that.

- These statements disproves that $2^{2n} \neq \Theta(2^n)$.

Part 2:

c) Are the following true? Prove your answer

III. Let $f(n) = O(n^2)$ and $g(n) = \Theta(n^2)$

Prove or disprove that:

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

→ Let see this rule

if $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then

$T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$

$T_1(N) * T_2(N) = O(f(N) * g(N))$

From definition, we can say at first look

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

We explain it with Big-O because of Big-O notation is larger than Θ.

• We need to define f(n) with $\Theta(n^2)$ to prove that.

But we have not any information more than we have.

So, these statements disprove that.

We say,

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

But we can say,

$f(n) * g(n) = O(n^4)$, proves by definition

Part 3: List the following functions according to their order of growth by explaining your assertions

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

→ First of all, let explain these functions growth rates in Big-O notations:

1. $n^{1.01} = O(n.\log n)$

$\lim_{n\to\infty} (n \log n)/(n^{1.01}) = \lim_{n\to\infty} (\log n)/(n^{0.01}) = \lim_{n\to\infty} (1/n)/(0.01.n^{-0.99})$

$= \lim_{n\to\infty} (n^{0.99})/(0.01\,n) = \lim_{n\to\infty} (1/0.01.n^{0.01}) = 0$

That means $n^{1.01}$ asymptotically dominates $n.\log n$.

2. $n.\log^2 n = O(n.\log^2 n) = O(n.\log n^2)$
3. $2^n = O(2^n)$ (exponential)
4. $\sqrt{n} = O(\sqrt{n})$
5. $\log n^3 = O(\log n^3)$ (logaritmic)
6. $n.2^n = O(n.2^n)$ (exponential)
7. $3^n = O(3^n)$ (exponential)
8. $2^{n+1} = 2.2^n = O(2^n)$ (exponential)
9. $5^{\log_2 n} = O(5^{\log_2 n})$ (linear)
10. $\log n = O(\log n)$ (logaritmic)

- Also we have a common growth rate order:

constant < logaritmic < linear < polynomial < exponential < factorial

- With some examples:

$1 < 2^{100} < \log n < \log n^2 < \sqrt{n} < n < 2^{\log n} < n \log n < n \log n^2 < n^2 < 0.0001 n^2$

$n^2 \log n < n^3 < n^3 \log n < n^{100} < 2^n < n2^n < 4^n < n! < n^n$

- We can order our list as follows according to their growth rate:

$\log n < \log n^3 < \sqrt{n} < 5^{\log_2 n} < n^{1.01} < n.\log^2 n < 2^n \leq 2^{n+1} < n.2^n < 3^n$

10 5 4 9 1 2 3 8 6 7

10: logaritmic ($\log n$)

5: logaritmic ($\log n^k$)

4: ($\sqrt{n}$)

9: linear ($a^{\log n}$) — linear

1: $= (n.\log n)$ — ($n.\log n$) — asymptotically $n.\log n$

2: ($n.\log n^k$)

3: exponential (a^n)

8: $= (2^n)$ — exponential (a^n)

$\underbrace{3 \quad 8}_{\text{asymptotically equal}}$

6: Exponential ($n.a^n$)

7: exponential (a^n)

$\underbrace{6 \quad 7}_{\text{they may change}}$

$\approx 3^n < n.2^n$

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.

⇒ int findMinimum (ArrayList<E> arr)

```
{
    int index = -1                              O(1)
    E minVal = arr.get(0)         -> O(1)  -> O(1)
    for (int i=1; i<arr.size(); i++){
        if (arr.get(i) < minVal) {  O(1)
            minVal = arr.get(i)  -> O(1)   O(1)      } O(1)   } O(n)*O(1) = O(n)
            index = i    O(1)
        }
    }
    return index     O(1)
}
```

- Time complexity: $O(n) = \Omega(n) \longrightarrow \boxed{\Theta(n)}$

Analyze:

get and size methods, only accessed values, then $O(1)$

loop turns $n-1$, then $O(n)$

all other operations are $O(1)$

best or worst, all possibilities are the same

so, we can explain running time with theta notation

$\Theta(n)$

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

```
int findMedian (ArrayList<E> arr)
{
    int index = -1                                              O(1)
    ArrayList<E> newArr = (ArrayList<E>) arr.clone()            O(n) ***
    Collections.sort(newArr)                                    O(n logn) ***
    int middle = newArr.size() / 2                              O(1)
    middle = ((middle > 0) && (middle % 2 == 0)) ? (middle-1) : (middle)   O(1)
    for (int i=0; i < arr.size(); i++) {                        O(n * O(1)) = O(n)
        if (arr.get(i) == newArr.get(middle)) {                 O(1)   O(1)   O(1)
            index = i   O(1)
            break
        }
    }
    return index        O(1)
}
```

max (O(n), O(nlogn), O(n)) = O(nlogn)

- Time Complexity : $O(n\log n) = \Omega(n\log n) \rightarrow \boxed{\Theta(n\log n)}$

Analyze:
clone method takes O(n) time
sort method takes O(nlogn) time
get and size methods take O(1) time
checking all items whether it is median or not, takes O(n) time
other all operations take O(1) time (return, initialize, assign, compare...)
total running time is max(O(n logn), O(n), O(1)) = O(nlogn)
we can explain running time with BigO notation simply
O(n.logn)
best case: array may be already sorted, but it can not change time
Collections.sort() method uses merge sort, mergesort have (nlogn) all time
also, arrays' first element may be median, so loop takes Ω(1) best case
Then max(Ω(nlogn), Ω(1)) = Ω(nlogn)
Ω(n logn)

Part 4: Give the pseudo-code for each of the following operations for an arraylist that has n elements and analyze the time complexity

- Find two elements whose sum is equal to a given value.

→
```
void printSumPairs (ArrayList<E> arr, E sum)
{
    for (int i=0; i<arr.size(); i++)                                          O(n) * O(n) = O(n²)
        for (int j=i+1; j<arr.size(); j++)                                  O(n)
            if ((arr.get(i) + arr.get(j)) == sum)               O(1)    O(1)
                Print (arr.get(i).toString() +" "+arr.get(j).toString())   O(1)
}
```

- Time complexity: $O(n^2) = \Omega(n^2) \rightarrow \boxed{\Theta(n^2)}$

Analyze:
Best case $\Omega(n^2)$ and worst case $O(n^2)$ are same.
we have two nested loops, it will turn all cases, so we can use theta
get and size methods takes $O(1)$ time
loops takes $\frac{n \cdot (n-1)}{2}$ times, we say n^2 running time
all other operations take $O(1)$ time
then time complexity
$\Theta(n^2)$

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

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

```
→ SingleLinkedList <E> mergeTwoOrderedArrayList (ArrayList <E> arr1, ArrayList <E> arr2)
{
    singleLinkedList <E> merged = new SingleLinkedList <E> ()
    int i=0 , j=0 , k=0
    if ( arr1.isEmpty()) {
        for (k=0, k<arr2.size(), k++)
            merged.add (arr2.get(k))
        return merged                    Ω(1)
    }
    if (arr2.isEmpty()) {
        for (k=0; k<arr1.size(), k++)
            merged.add (arr1.get (k))
        return merged
    }
    while ((i <arr1.size()) && (j < arr2.size())) {
        if (arr1.get(i) < arr2.get(j))
            merged.add(arr1.get(i++))
        else if (arr1.get(i) == arr2.get(j)) {
            merged.add(arr1.get(i++))
            merged.add(arr2.get(j++))
        }
        else
            merged.add (arr2.get (j++))
    }
    if (i < arr1 size())
        for(k=i; k<arr1.size(), k++)
            merged.add (arr1.get (k))
    if (j< arr2.size())
        for (k=j, k<arr2.size(); k++)
            merged.add(arr2.get(k))
    return merged
}
```

- Time Complexity : $O(n+m)$

Analyze :

Best case → two array lists one empty
algorithm returns empty single linked list
$\Omega(1)$ time

Worst case → two array lists are not empty
two arrays' each elements are different
then $O(n+m)$ time.

Average → we can not analyze average time.
It can be change for all different sizes.

Then, we can say running time is,
$O(n+m)$

+ **Note:** Sample worst case scenario for my algorithm:

arr1 → 1 3 6 (size → 3) (n)

arr2 → 2 4 5 7 8 9 (size → 6) (m)

merged → empty

---

		i	j
arr1(0) < arr2(0) → merged adds arr1(0)	i++	1	0
arr1(1) > arr2(0) → merged adds arr2(0)	j++	1	1
arr1(1) < arr2(1) → merged adds arr1(1)	i++	2	1
arr1(2) > arr2(1) → merged adds arr2(1)	j++	2	2
arr1(2) > arr2(2) → merged adds arr2(2)	j++	2	3
arr1(2) < arr2(3) → merged adds arr1(2)	i++	3	3

works ... times ... < (n+m) ... here : ... n + x

n times → reached arr1 size.

x times

ends loop, works last statements :

if (i(3) < arr1.size()(3))
x does not run

if (j(3) < arr2.size()(6))
✓ runs
↳ for(k = j(3); k < arr2.size()(6); k++) → works (m-x) times
merged adds arr2(k) → arr2(3)
arr2(4)
arr2(5)

works remaining n or m here : (m-x)

totally

$\boxed{n+x+m-x = n+m}$ times !!

Part 5: Analyze the time complexity and space complexity of the following code segment

a) int p_1 (int array[]):
{
return (array[0] * array[2]) $O(1)$ $\Omega(1)$
}

→ getting arrays' first element
getting arrays' third element index 2
multiplying two elements
returning result
✓ did not use any extra space, only accessed elements
- Time Complexity: $O(1) = \Omega(1) \rightarrow \Theta(1)$
- Space Complexity: $O(1)$

Part 5: Analyze the time complexity and space complexity of the following code segments

b)
```
int p_2 (int array[], int n):
{
    int sum = 0                              O(1)
    for (int i=0; i<n; i=i+5)
        sum += array[i] * array[i]           O(1)
    return sum        O(1)
}
```

$O(n) * O(1) = O(n)$

$O(1) + O(n) + O(1) = O(n)$

→ initialize sum
initialize i
compare i is lower than n or not
adding i and 5
assigning i to addition of i and 5
* loop works n times (n/5 times)
getting array element index i
getting array element index i
multiply two elements
add sum and multiplication result
assign sum to addition of sum and multiplication result

$T(n) * T(1)$

return sum

✓ created integer variable sum
created integer variable i
did not use any other space, remaining parts accessed only

- Time Complexity: $O(n) = \Omega(n) = \Theta(n)$
- Space Complexity: $O(1)$

Part 5: Analyse the time complexity and space complexity of the following code segments.

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

$\left.\left.\begin{matrix} \text{base } 2 \\ O(\log n) * O(1) \\ = O(\log n) \end{matrix}\right\}\right\} O(n) * O(\log n) = O(n.\log n)$

→ initialize i

compare i and n

increase i by 1

initialize j

compare j and i

multiply j by 2

assign j result of multiplication j and 2

print screen

get value, assign result

get array element index i

get array element index j

multiply two elements

* first loop works n times

* second loop works logn times

✓ created integer variable i

created integer variable j

and get space to put result as integer

did not use any other space, remaining parts accessed only

- Time Complexity : $O(n.\log n) = \Omega(n.\log n) = \Theta(n.\log n)$
- Space Complexity : $O(1)$

Part 5: Analyze the time complexity and space complexity of the following code segments.

d)

```
void p_4 (int array[], int n)
{
    if (p_2(array, n) > 1000)          O(n)        } O(n) + O(n logn)
        p_3(array, n)                  O(n logn)   } = O(n logn)
    else
        printf("%d", p_1(array) * p_2(array, n))   O(1) * (O(1) + O(n))
}                                                           = O(n)
```

$= O(n)$

$+ O(n)$

$= O(n)$

→ calls p_2 with $O(n)$ time

compare p_2 result with 1000

if case may work

calls p_3, with $O(n.logn)$ time

else case may work

prints screen

calls p_1 with $O(1)$ time

calls p_2 $O(n)$ time

multiply two results

✓ calls p_2 with $O(1)$ space

calls p_3 with $O(1)$ space

calls p_1 with $O(1)$ space

calls p_2 with $O(1)$ space

%d gets space to put result as integer

- Time Complexity : best → else case → $\Omega(n)$

  worst → if case → $O(n.logn)$

- Space Complexity : All situations, it is $O(1)$.