CSE232 – HW2

Oğulcan Kalafatoğlu 1801042613

Part1:

I.

```java
public int searchProduct(Product.type_t type, Product.color_t color, String model, String branchName){
    int branchIndex = getIndexOfBranch(branchName);
    for(int i = 0 ; i < branch.getData(branchIndex).getProducts().size() ; ++i){
        Product temp = branch.getData(branchIndex).getProducts().getData(i);
        if(temp.getEnumType() == type && temp.getEnumColor() == color && temp.getModel().equals(model)) return i;
    }
    return -1;
}
```

T(n) = O(n)

II.

```java
public class DynamicArray<T> {
    private Object[] data;
    private int capacity;
    private int used;
```

```java
public void insert(T obj){
    if(capacity <= used || capacity == 0){
        Object temp[] = new Object[capacity];
        for(int i = 0 ; i < used ; ++i){
            temp[i] = data[i];
        }

        data = new Object[capacity * 2];
        capacity *= 2;
        for(int i = 0 ; i < used ; ++i)
            data[i] = temp[i];
        data[used] = obj;
        ++used;
    }
    else{
        data[used] = obj;
        ++used;
    }
}
```

```java
public void addProduct(Product.type_t type, Product.color_t color, String model){
    Product temp = new Product(type,color,model);
    products.insert(temp);
}
```

addProduct's time complexity:

T(n) = O(n) = Ω(1)

```java
public int getIndexOfBranch(String name){
    int index = -1;

    for(int i = 0 ; i < branch.size(); ++i){
        Branch temp = branch.getData(i);

        if((temp.getName()).equals(name)){
            return i;
        }
    }
    return index;
}
```

T(n) = ϴ(n)

```java
public int searchProduct(Product.type_t type, Product.color_t color, String model, String branchName){
    int branchIndex = getIndexOfBranch(branchName);
    for(int i = 0 ; i < branch.getData(branchIndex).getProducts().size() ; ++i){
        Product temp = branch.getData(branchIndex).getProducts().getData(i);
        if(temp.getEnumType() == type && temp.getEnumColor() == color && temp.getModel().equals(model)) return i;
    }
    return -1;
}
```

T(n) = O(n)

```java
public void eraseProduct(Product.type_t type, Product.color_t color, String model, String branchName){
    int branchIndex = getIndexOfBranch(branchName);
    int productIndex = searchProduct(type, color, model, branchName);
    getBranch().getData(branchIndex).getProducts().eraseByIndex(productIndex);
}
```

T(n) = ϴ(n) + O(n) = O(n)

III.

```java
public int[] queryProduct(DynamicArray<Product> expected){
    int size = getBranch().getProducts().size();
    int indexArr[] = new int[size];
    int k = 0;

    for(int i = 0 ; i < size; ++i){
        if(searchProduct(expected.getData(i).getEnumType(),
            expected.getData(i).getEnumColor(), expected.getData(i).getModel(), branch.getName()) == -1){
            indexArr[k++] = i;
        }
    }

    return indexArr;
}
```

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

T(n) = O(n^2)

Part 2:

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

Big O notation describes worst case of a algorithm and this statement means $T(n) >= O(n^2)$, from this we cannot derive upper bound.

If we assume lower bound $f(n) = O(n^2)$, then from this statement we would derive $T(n) >= f(n)$ but $f(n)$ is not strictly n^2, $f(n)$ could be anything smaller than n^2, so we cannot derive lower bound too,

Hence this statement is meaningless.

b)

Since $f(n) <= f(n) + g(n)$ and $g(n) <= f(n) + g(n)$,

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

if $f(n) >= g(n)$,

$$f(n) + f(n) >= f(n) + g(n) \rightarrow f(n) + g(n) <= 2f(n)$$

if $g(n) >= f(n)$,

$$g(n) + g(n) >= f(n) + g(n) \rightarrow f(n) + g(n) <= 2g(n)$$

Hence,

$$f(n) + g(n) <= 2\max(f(n) + g(n))$$

So we get

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

Hence

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

c)

$$\lim_{N\to\infty}\frac{f(N)}{g(N)} = 0 \qquad \Rightarrow f(N) = o(g(N))$$

$$= c \neq 0 \qquad \Rightarrow f(N) = \theta(g(N))$$

$$= \infty \qquad \Rightarrow g(N) = o(f(N))$$

$$= \text{oscilate} \Rightarrow \text{there is no relation}$$

i)

$\lim\limits_{n\to inf} \frac{2^{n+1}}{2^n}$ = $\lim\limits_{n\to inf} \frac{2^n x\ 2}{2^n}$ = $\lim\limits_{n\to inf} \frac{2}{1}$ = 2

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

ii)

$\lim\limits_{n\to inf} \frac{2^{2n}}{2^n}$ = $\lim\limits_{n\to inf} \frac{2^n x 2^n}{2^n}$ = $\lim\limits_{n\to inf} \frac{2^n}{1}$ = inf

Hence $2^n = o(2^{2n})$ and 2^{2n} != $\Theta(2^n)$

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

By definition $f(n) <= c \times n^2$ and $c_0 \times n^2 <= g(n) <= c_1 \times n^2$,
since f(n) could be anything smaller than n^2, $f(n) * g(n) = \Theta(n^4)$ is not always true.

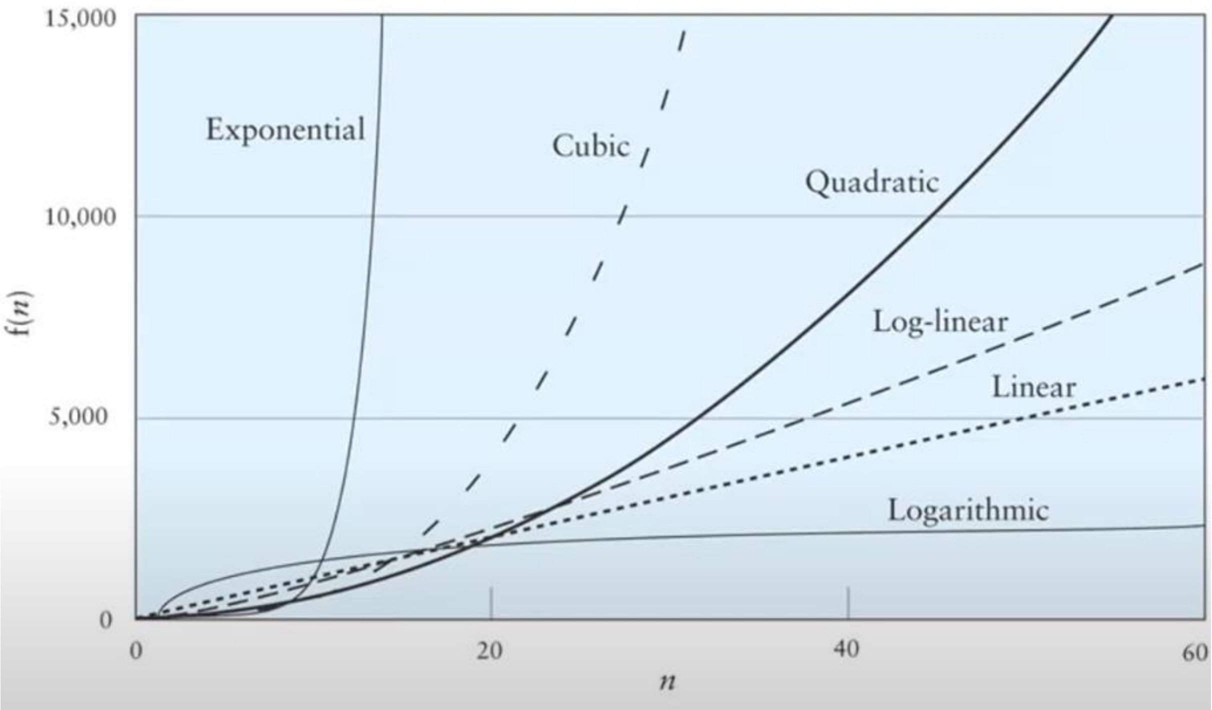
For example : $f(n) = n$, $g(n) = n^2$ so $f(n) * g(n)$ could be $\Theta(n) * \Theta(n^2) = \Theta(n^3)$.

Hence statement "$f(n) * g(n) = \Theta(n^4)$" is not true for every f(n).

Part 3:

Common growth rates are as follows :

Constant < Logarithmic < Linear < Log-linear < Quadratic < Cubic < Exponential < Factorial



According to this

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

$5^{\log_2 n} = n^{\log_2 5} = n^{2.3}$

$\lim_{n\to inf} \frac{2^{n+1}}{2^n} = \lim_{n\to inf} \frac{2^n x\, 2}{2^n} = \lim_{n\to inf} \frac{2}{1}$ = 2 so both functions grows at same rate.

$\lim_{n\to inf} \frac{3^n}{n2^n} = \lim_{n\to inf} \frac{(\frac{3}{2})^n}{n}$ = inf so 3^n grows faster.

$\lim_{n\to inf} \frac{n^{1.01}}{nlog^2 n} = \lim_{n\to inf} \frac{n^{0.01}}{(\log_2 n)^2} = \lim_{n\to inf} \frac{n^{0.01}\, x\, log2}{logn}$ = log2 x $\lim_{n\to inf} \frac{n^{0.01}}{logn}$ = inf

$n^{0.01}$ grows faster than logn. So $n^{1.01}$ grows faster than $n\log^2 n$.

Part 4:

a-)Find the minimum valued item

```
initialize min to list[0]
for i = 1 to n
        if min is greater than list[i]
                SET min to list[i]
end for
return min
```

Time complexity is $\Theta(n)$

b-)Find the median item

```
mySort(list)                              -> O(n²)
for i = 0 to n                            -> O(n)
        if size % 2 equals to 1
                if i equals to (size+1)/2 – 1
                        return list[i]

        else
                if i equals to size/2 – 1
                        return (list[i] + list[i+1]) / 2
end for
```

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

```
function mySort(arr: arrayList, n)
        for i = 0 to n-1                  ->O(n)
                for j = 0 to n-i-1        ->O(n)
                        if arr[j] is greater than arr[j+1]
                                swap arr[j] and arr[j+1]
                end for
        end for
```

(mySort function has $O(n^2)$ of time complexity) c-)Find two elements whose sum is equal to a given value

inputs: value
outputs: list2->2 size of arrayList

```
    initialize num1, num2

    for i = 0 to n-1                         -> O(n-1)
        for j = i+1 to n-1                   -> O(n-2)
            if list[i] + list[j] is equal to value
                list2[0] = list[i]  -> O(1)
                list2[1] = list[j]  -> O(1)
                break

        end for
    end for
    return list2
```

$T(n) = O((n-1) * (n-2)) = O(n^2)$

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

inputs: list, list2
output: 2n sized list3

```
initialize i to 0
initialize j to 0
initialize k to 0
declare an array list of 2n elements -> list3
while i is less than n and j is less than n        -> O(n)
        if list[i] is less than list2[j]
                set list3[k] to list[i]
                increment k
                increment i

        else
                set list3[k] to list2[j]
                increment k
                increment j
end while

while i is less than n                              -> O(n)
        set list3[k] to list[i]
        increment k
        increment i
end while

while j is less than n                              -> O(n)
        set list3[k] to list2[j]
        increment k
        increment j
end while

return list3
```

T(n) = O(n)

Part 5:

time, space

a)

```
int p_1 (int array[]):
{
        return array[0] * array[2]) -> Θ (1) + Θ(1) = Θ(1)
}
```

T(n) = Θ(1) since return statement and operations are constant time

S(n) = O(1) since space required by algorithm is constant

b-)

```
int p_2 (int array[], int n):
{                                       time, space
        Int sum = 0                     -> Θ(1), O(1)
        for (int i = 0; i < n; i=i+5)   -> Θ(n/5), O(1)
                sum += array[i] * array[i]) -> Θ(1)
        return sum                      -> Θ(1)
}
```

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

$S(n) = O(1)$

c-)

```
void p_3 (int array[], int n):
{
        for (int i = 0; i < n; i++)                    -> Θ(n), O(1)
                for (int j = 1; j < i; j=j*2)          -> Θ(log₂ n), O(1)
                        printf("%d", array[i] * array[j]) -> Θ(1)
}
```

$T(n) = \Theta(n) * \Theta(\log_2 n) = \Theta(n * \log_2 n)$

$S(n) = O(1)$

d-)

```
void p_4 (int array[], int n):

{

        If (p_2(array, n)) > 1000)        -> Θ(n) + 1= T3(n)
                                             O(1) = S3(n)
                p_3(array, n)             -> Θ(n log₂ n)+ 1 = T1(n)
                                             O(1) =  S1(n)
        else
                printf("%d", p_1(array) * p_2(array, n))        -> Θ(1) +  Θ(n) + 1 = T2(n)
                                                                    O(1) = S2(n)

}
```

$T_1(n) = \Theta(n\log n)$, $T_2(n) = \Theta(n)$, $T_3(n) = \Theta(n)$

$T_w = T_3(n) + \max(T_1(n), T_2(n)) = \Theta(n) + \Theta(n * \log n) = \Theta(n\log n)$

$T_b(n) = T_3(n) + \min(T_1(n), T_2(n)) = \Theta(n) + \Theta(n) = \Theta(n)$

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

$= \Omega(n)$

$S(n) = O(1)$