GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CSE222/505 – Spring 2021

Homework 2 Report

Yakup Talha Yolcu

1801042609

Part 1

I wrote new code for these algorithms. I did not want to use my previous homework codes.

1. Searching a product

   Time complexity is just Θ(n) because for loop will be executed n times because of the branch products's length. In println and get methods, we have constant time.

   Time complexity= Θ (n)

```
public void search(String model_name,String color_name) {
    for(int i=0;i<branch_products.length;i++) {
        System.out.println(branch_products.get(i));
    }
}
```

2. Add/remove product.

   i) Add product

   Time complexity for the worst case is Θ (n) (amortized) for reallocation + Θ (1) for get_size method.

   Time complexity for the best case is Θ (1) because of get_size method.

   So, at the end our time complexity is O(n).

```
public void add_product(Product a1) {
    if(get_size()==branch_products.length) {
        reallocate();
    }
    branch_products[get_size()+1]=a1;
    size++;
}
```

   ii) Remove product

   To remove a product, we need to find this product by giving color and model names. After finding it, we need to decrement the stock.

   We have 2 for loops and 2 if's for each. For the worst case we will have Θ(n*m) complexity n for-> model name, m for -> color name.

   For the best case we have Θ(1) complexity. So our complexity is O(n*m)

   Getters are considered as constant time.

```
public void remove_product(String color,String model,int stock) {
    for(int i=0;i<branch_products.length;i++) {
        if(model.equals(branch_products.get(i).get_name())) {
            for(int j=0;j<branch_products.get(i).colors.length;j++) {
                if(color.equals(branch_products.get(i).colors.get(j).get_name())) {
                    (branch_products.get(i).colors.get(j).set_stock(get_stock()-stock)
                }
            }
        }
    }
}
```

3. Querying the products that need to be supplied.

   Admin can query by giving a branch to the system and learn whether product need to be supplied.
   Time comlexity is constant time because we have getter function and we have comparison.
   $T(n)=\Theta(1)$

```
public boolean query(Branch b1){
    return !(b1.get_stock()==false);
}
```

Part 2

- a) It's meaningless to say : "The running time of an algorithm A is at least $O(n^2)$ because when we are using Big Oh notation we are indicating that our algorithm's running time is less than or equal to n^2

- b) We know $\Theta(f(n)+g(n))=O(f(n)+g(n))=\Omega(f(n)+g(n))$

  max(f(n),g(n)) >= g(n) if max(f(n),g(n))=f(n)
  max(f(n),g(n)) >= f(n) if max(f(n),g(n))=g(n)
  Let's say that max(f(n),g(n))=f(n) and let $T(f(n))=\Theta(n)$
  So, there must be $\Theta(n)>=g(n)$
  g(n) could be $\Theta(n)$ or $\Theta(1)$
  Let's say that $g(n)=\Theta(n)$ -> then $O(f(n)+g(n))=O(n+n)$
  N+n 's time complexity is $\Theta(n)$. max(f(n),g(n)) was f(n) and it was $\Theta(n)$.
  It is appropriate.

  Let's say that $g(n)=\Theta(1)$. Then $O(f(n)+g(n))=O(n+1)$.
  $O(n+1)$?=max(f(n),g(n)) => $O(n+1)=\Theta(n)$. yes it is proved.

  If we had chose the g(n) as maximum, we would get the same result.

- c)
  1) $2^{n+1}=\Theta(2^n)$
     $2^{n+1}=2^n * 2$, so we have lower order term as 2. We can remove it.
     $2^n=\Theta(2^n)$ we can say this because Θ checks order of magnitude.
     It is true
  2) $2^{2n}=\Theta(2^n)$
     $2^{2n}=(2^2)n=4^n$ so we can't say that. It is wrong
  3) Let $f(n)=O(n^2)$ and $g(n)=\Theta(n^2)$. Prove or disprove that: $f(n) * g(n)=\Theta(n^4)$.

$O(n^2)$ means that f(n)'s complexity is greater than or equal to n^2.
$\Theta(n^2)$ means that g(n)'s complexity is exactly n^2.
Let's check $O(n^2)* \Theta(n^2)?= \Theta(n^4)$
We can say $O(n^2)=O(n^3)$. So $O(n^3)* \Theta(n^2)= O(n^5)$.
$O(n^5) = \Theta(n^4)$ for just some cases, not always. So we can't say this statement is absoulutely correct.

Part 3

$n^{1.01}$, $n\log^2 n$, 2^n $n^{1/2}$, $\log^3 n$, $n2^n$, 3^n , 2^{n+1} , $5^{\log n}$, $\log n$

We know exponential fuctions are the largest ones.

We need to start from 2^n , $n2^n$, 3^n , 2^{n+1} , $5^{\log n}$.

$5^{\log n}$ will be the least because logn is less than n.

2^n and 2^{n+1} will be equal because of constant 2.

Let's say lim $(3^n)/ (n2^n)$, n goes infinity, we will get 0. It means that $(n2^n)$ is larger.

Up to now we have $3^n > n.2^n > 2^{n+1} = 2^n > 5^{\log_2 n}$

Remained: $n^{1.01}$, $n\log^2 n$, $n^{1/2}$, $\log^3 n$ and logn

We know $n^{1.01} > n\log^2 n > \log^3 n > \log n$

If we have lim(n goes infinity) $(n^{1/2}) / \log^3 n$ = (By L'opital) $(1/(2* n^{1/2}))/ (1/n(\ln 2))*3$ close the infinity. We will get ln2 at the end. So $(n^{1/2})$ is larger.

The growth order is ->

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

Part 4

1) Minimum valued item

```
ArrayList<int> x;
set min=x.get(0);
for (i in iterable x) {
    if i<min do
        i=min
    end
}
return min
```

Time complexity is -> in for loop we have if statement. In if statement we have comparison and it is contant time. We have Θ(1). Loop is executed n times. So we have Θ(n)*Θ(1) = Θ(n). It is complexity is Θ(n).

2) Median item

- For set j=0,j<n,j++
    - -Determine minimum and maximum of the ArrayList // Θ(n)
- -While
    - -For set i=0 , i<n , i++ // Θ(n^2)
        - İf min<ArrayList[i] and there is no such element between these two value // Θ(n)
          Min=arraylist[i]
        - İf max>ArrayList[i] and there is no such element between these two value // Θ(n)

          Max=ArrayList[i]
    - İf max equals min, break
    - If absolute value of max-min less than or equal to 1, break

//For best case, while will be executed one times, and complexity will be Θ(n^2)

//For the worst case, while will be executed n/2 times, So complexity will be Θ(n^2 * n/2) which is Θ(n^3)

//Total complexity will be O(n^3)

- Return (min+max)/2

3) Find two elements whose sum is equal to given value.

- Set boolean flag=false // Θ(1)
- For i=0,i<n,i++ // Θ(1) for best case, Θ(n^2) for worst case (in total)
    - For j=0,j<n,j++ // Θ(1) for best case, Θ(n) for worst case
        - İf (the value that in the ith index) + (the value that in the jth index) equals the given value // Θ(1)
            - Flag=true
            - Break
    - İf flag equals true //Θ(1)
        - Break
- We have i and j now
- **//At the end, we have O(n^2)**

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

- For i=0,i<n+m,i++ // Θ(n+m)
  - İf i<n // Θ(1)
    - Set list’s ith index value as the first list’s ith index value
  - Else Θ(1)
    - Set list’s ith index value as the second list’s i-n’th index value
- Sort the merged list // O(nlogn) considered as merge sort
- **So our time complexity is O(nlogn) at the end**

Part 5

1)

```
int p_1 (int array[]): {
        return array[0] * array[2]) // Complexity is Θ(1)
//Total time complexity is Θ(1)
//Total space complexity is S(1)

}
```

2)

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

//Total space complexity is S(1)
//Total time complexity is Θ(1)+ Θ(n)+ Θ(1) = Θ(n)
    }
```

3)

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

        //Total space complexity is S(1)
        //Total time complexity is Θ(1)*O(n*log n)= O(n*log n)
```

4)

```
void p_4 (int array[], int n): {
        If (p_2(array, n)) > 1000) //Space complexity = S(1) T(n)= Θ(n)
                p_3(array, n) // Space complexity = S(1) T(n)=O(n*logn)
        else printf("%d", p_1(array) * p_2(array, n)) //  Space complexity= S(1)
        //T(n)= Θ(n) + Θ(1) = Θ(n)

        //Best case S(n)= Worst case S(n)=S(1)+S(1)

        //Best case T(n)= Θ(n)+  Θ(n)= Θ(n)
        //Worst case T(n) Θ(n)+ O(n*logn)= O(n*logn)
        Total time complexity = O(n*logn)
        Total space complexity = S(1)

  }
```

//I thought that space complexity does not depend on the input size, so all space complexities are 1 as constant