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

Muhammet Fikret ATAR
1801042693

Part 1:

```
public boolean check(Products p2) {
        if(this.Product_Typep.equal(p2.Product_Type)) {          ------->

                return false;                                    ------->

        }
        else if(this.Model_name.equal(p2.Model_name)) {          ------->

                return false;                                    ------->

        }
        else if (this.Color_name.equal(p2.Color_name)){          ------->
                return false;                                    ------->


        }
        return true;                                             ------->

        }
```

General Running Time

$T(n) \rightarrow \theta(1)$

Steps/Exec	Freq	Total
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1

Total:7

I. Searching a product:

```
public void search () {

for(i=0;i<this.Products_count-1;i++) {                    --->

        if(products[i].check(new Products(Type,Model,Color) {    --->

                System.out.println("Product Faund");           --->

        }
    }

}
```

General Running Time

$T(n)=6n-8+2O(1) \rightarrow O(n)$

Steps/Exec	Freq	Total
2	n	2n
2	O(1) n-1	2(n-2)+2O(1)
1	n-1	2(n-2)

Products_count=n

II. Add/remove product:

Add;

```
public void add_product() {

String Model,Type,Color;


System.out.println("Choice Type:");
                choice = scan.nextInt();

System.out.println("Color Type:");

                        choice = scan.nextInt();

}
System.out.println("Model Type:");

                choice = scan.nextInt();

products[Products_count++]=new Products(Type,Model,Color);

}
```

Steps/Exec	Freq	Total
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1

Total:7

General Running Time

$T(n) \rightarrow \theta(1)$

Remove;

```
public void remove_product () {

String Model,Type,Color;


System.out.println("Choice Type:");
                choice = scan.nextInt();

System.out.println("Color Type:");

                        choice = scan.nextInt();

}
System.out.println("Model Type:");

                choice = scan.nextInt();

for(i=0;i<this.Products_count-1;i++) {

        if(products[i].check(new Products(Type,Model,Color) {

                for(int k=i;k<Products_count;k++) {
                products[k]=products[k+1];
                }
                Products_count--;
        }
}
```

Steps/Exec	Freq	Total
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
1	1	1
2	n	2n
2	(n-1)+O(1)	2n-2+2O(1)
2	n(n+1)/2+n	n(n+1)+2n
1	n(n+1)/2	n(n+1)/2
1	n	n

Product_count=n

$T(n): 2n^2+7n+8$

Best Case : $\Omega(n)$

Worst Case: $O(n^2)$

Querying the products that need to be supplied:

```
public void  Querying_Product() {
        int i;

        for(i=0;i<Products_count;i++) {
                if(products[i].getColor_name()!=null) {

                System.out.println(i+"->"+products[i].toString());

                }

                else if(products[i].getColor_name()==null) {


                System.out.println(i+"->"+products[i].to1String());
                }
         }
 }
```

Steps/Exec	Freq	Total
1	1	1
2	n+1	2n+2
1	n	n
1	n	n
1	n	n
1	n	n

Products_count=n Total:6n+3

General Running Time

T(n)=6n+3→ O(n)

Part 2 :

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

→ Let $T(n)$ be the running time for algorithm A and let a function $f(n) = O(n^2)$. The statement says that $T(n)$ is at least $O(n^2)$. That is, $T(n)$ is an upper bound of $f(n)$, since $f(n)$ could be any function "smaller" than n^2 (including constant function), we can rephase the statement as "The running time of algorithm A is at least constant." This is meaningless because the running time for every algorithm is at least constant.

---

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

→ First prove that $\max(f(n), g(n)) = O(f(n)+g(n))$

To see why this is true, let $\max(f(n), g(n)) = f(n)$ without loss of generality.

Then using $g(n) \geq 0$, it can be seen that

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

Therefore $\max(f(n), g(n)) \leq c(f(n)+g(n))$ for all $n \geq n_0$ with $c = 1$, $n_0 = 1$, proving that $\max(f(n), g(n)) = O(f(n)+g(n))$.

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

Again let $\max(f(n), g(n)) = f(n)$ without loss of generality..

Then $f(n) \geq g(n)$, therefore $2f(n) \geq g(n) + f(n)$.

Hence $2\max(f(n), g(n)) = 2f(n) \geq f(n) + g(n)$, or $\max(f(n), g(n)) \geq 1/2(f(n) + g(n))$.

Hence $\max(f(n), g(n)) \geq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = 1/2$, $n_0 = 1$. This proves $\max(f(n), g(n)) = \Omega(f(n) + g(n))$.

The two statements together prove that

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

---

c) If we know the average case of a function, we know that there might be worse scenarios. However, if we know the worst case of another function, there connot be anyhing worse. Had it been 2 average case functions, the multiplication of those functions would also be that its average case is lower but is not higher. So, if we multiply one avarage case with a worst case, what we get is the worst case.

Thus $f(n) * g(n) = O(n^4)$

This one is not correct.

Part 3:

List the functions according to their order of growth

Order of growth	Name
1	constant
$\log_b n$	logarithmic (for any b)
n	Linear
$n \log_b n$	"en log en"
n^2	quadratic
n^3	cubic
c^n	exponential (for any c)

So, "$\log n$", "$n \log^2 n$", "$\sqrt{n}$", "$(\log n)^3$" < others.

logarithmic.

→ $\log n$ and $(\log n)^3$

$$\lim_{n \to \infty} \frac{\log n}{(\log n)^3} = \lim_{n \to \infty} \frac{1}{(\log n)^2} = \frac{\lim_{n \to \infty} (1)}{\lim_{n \to \infty} (\log_2 n)^3} = \frac{1}{\infty} = 0^+$$

$\log n$ here grows slower than $(\log n)^3$

Part 3:

→ $n^{1/2}$ and $(\log n)^3$

$$\lim_{n\to\infty} \frac{(\log n)^3}{n^{1/2}} = (\text{Apply L'Hospital's Rule}) = \frac{\frac{24}{x}}{\frac{1}{2\sqrt{x}}} : \frac{48}{\sqrt{x}}$$

$$= \lim_{n\to\infty} \left(\frac{48}{\sqrt{x}}\right) = 0^+ =$$

$$\text{Apply} \rightarrow \left(\lim_{x\to\infty} \left(\frac{c}{x^a}\right) = 0 \right)$$

so $(\log n)^3$ here grows slower than $n^{1/2}$

NOTE

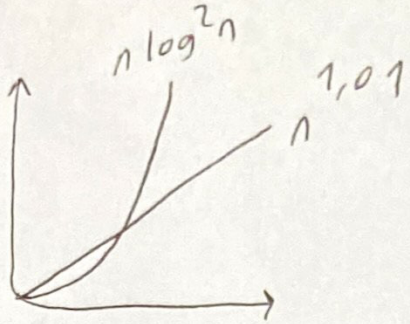
For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quicly, so exponential algorithms are only useful for small problems.

→ $n^{1,01}$ and $\sqrt{n}$

According to the table $n^{1,01} > n^{1/2}$ hence $n^{1,01}$ here grows faster than $\sqrt{n}$.

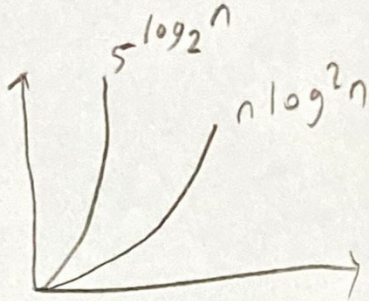
Part 3:

→ $n^{1,01}$ and $n\log^2 n$



According to the graphic $n\log^2 n$ grows faster than $n^{1,01}$.

→ $5^{\log_2 n}$ and $n\log^2 n$



According to the graphic $5^{\log_2 n}$ grows faster than $n\log^2 n$

→ $5^{\log_2 n}$ and 2^n

According to the growth table $5^{\log_2 n}$ grows slower than 2^n.

→ 2^n and 2^{n+1}

$n+1 > n$ so that 2^{n+1} grows faster than 2^{n+1}.

→ 2^{n+1} and $n2^n$

According to the table $n2^n$ grows faster than 2^{n+1}.

Part 3 :

→ $n2^n$ and 3^n

3^n grows faster because its base larger

As a result : Their order of growth.

$$3^n > n.2^n > 2^{n+1} > 2^n > 5^{\log_2 n} > n\log^2 n > n^{1,01} > \sqrt{n}$$

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

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.

(1) Find the minumum-valued item

Pseudo code.

```
minValue( arr[], int size)
Initialize array's first element to min element
While i smaller than ArraySize
If Array's i. element smaller then min element
Set Array's i element min
Increase i
```

Complexity Analysis

At every step of the loop, we are doing 1 comparisons in the worst case. Total no. of comparison (in worst case) = $(n-1)$

Time complexity = $O(n)$, space complexity = $O(1)$

In the best case, a total of $n-1$ comparisons have been made.

Part 4:

(2) Find the median item. Consider each element one by one and check whether it is the median.

Pseudo Code

```
Array A, Array Size n
    int i,j;
  for(i=0, i<n-1; i++)
      for(j=0; i<n-i-1; j++)
          if( A[j] > A[j+1])
              temp = A[i]
                A[i] = A[1+1)
                A[1+1] = temp
        end
    end
end
        middle = (N+1)/2
        Display A[middle] as median
```

Complexity Analysis:

Time Complexity : n BubbleSort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be $(n-1)+(n-2)+(n-3)+\ldots+3+2+1$

Sum = $n(n-1)/2$ $O(n^2)$ Hence the time complexity of Bubble Sort is $O(n^2)$. The space complexity for Bubble Sort is $O(1)$, because only a single additional memory space is required for temp variable. Also, the best case time complexity will be $O(n)$, it is when the list is already sorted.

Following are the Time and space complexity for the median algorithm.

Worst Case Time Complexity [Big-O] : $O(n^2)$

Best Case Time Complexity [Big-omega] : $O(n)$

Average Time Complexity [Big-theta] : $O(n^2)$

Space Complexity : $O(1)$

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

Pseudo Code

```
has Array Two Candidates (A[], ar_size, sum)
int I, r :
   sort ( A , A + arr_size );
    I = 0
    r = arr_size - 1
while ( I < r) {
      if ( A[I] + A[r] == sum)
         return 1;
      else if ( A[I] + A[r] < sum)
          I ++ ;
       else // A[i] + A[j] > sum.
          r-- ;
    }
    return 0 ;
}
```

Complexity Analysis

Time complexity

Depends on what sorting algorithm we use
If Merge sort or Heap sort is used then $\Theta(n \log n)$ in worst case, If Quick sort is used then $O(n^2)$ in the worst case.

Space Complexity

This too depends on sorting algorithm. The auxiliary space is $O(n)$ for merge sort and $O(1)$ for Heap Sort.

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

Pseudo Code

```
mergeAndSort (ArrayList<> arraylist1, ArrayList<>
                                        arraylist2)

arraylist1.addAll(listtwo)
    for(int i=0; i<arraylist1.size(); i++)
      for(int j=0; j<arraylist1.size(); j++)
       if(arraylist1[i].element < arraylist1[i+1].element)
                temp = arraylist1[i]
                arraylist1[i] = arraylist1[i+1]
                arraylist1[i+1] = temp

          end
       end
end
                return arraylist1;
```

Complexity analysis

Time Complexity

We used two loop and each loop size's n then time Complexity will be $O(n^2)$ in the worst case.

Space Complexity

we just used local variable for storage and then space complexity $O(1)$.

Part 5 :

a) 1 int p_1 (int array[]);

{

2 return array[0] * array[2]

}

	steps/exec	freq	total
			2
2	2	1	

NOTE : we don't have any condition so that we don't have best, worst and average case

Time complexity = $T(n) = \Theta(1)$

Space complexity = $O(1)$

---

b) 1 int p_2 (int array[], int n);

}

		steps/exec	freq	total
2	int sum = 0	→ 1	1	1
3	for (int i = 0 ; i < n ; i = i + 5)	→ 2	$\frac{n}{5}+1$	$\frac{2n}{5}+2$
4	sum += array[i] * array[i]	→ 3	$\frac{n}{5}$	$3n/5$
5	return sum	→ 1	1	1

$n+4$

}

NOTE : We don't have any condition so we don't need best, worst and average case.

$T(n) = n+4 \Rightarrow \Theta(n)$

Space Complexity $\Rightarrow O(1)$

Part 5 :

c)

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

step/exec	freq	total
2	n+1	2n+2
2	n(logn)	2n(logn)
	logn	logn
		2n+2 +2nlogn +logn

ignore low order terms!

$T(n) = 2n+2+2n\log n+\log n = \Theta(n\log n)$ //

Space Complexity $= O(1)$ //

Note: we don't have any condition so we don't need best, worst and average case.

d)

```
void p_4(int array[], int n):
{
    if(p_2(array,n)) > 1000)   } T3(n) = Θ(n)
        p_3(array,n)           } T1(n) = Θ(nlogn)
    else                                          } T(n)
        printf("%d", p_1(array) * p_2(array,n)) } T2(n) = Θ(n)
}
```

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

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

$T_{av}(n) = p(T)\,T_1(n) + p(F)\,T_2(n) + T_3(n) = n\log n$

$p(T) \rightarrow p(\text{condition} = \text{True})$

$p(F) \rightarrow p(\text{condition} = \text{False})$

if $p(T) = p(F) = 1/2$

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

$= \Omega(n)$

Θ

Space Complexity (1) //