GTU Department of Computer Engineering
CSE 222/505 - SPRING 2022
HOMEWORK 2 REPORT

BURAK ÇİÇEK
1901042260

1-) $\log_2 n^2 + 1 \stackrel{?}{=} O(n)$

$\log_2 n^2 + 1 \le c.n$

$n \to 2^n$

$\log_2 2^{2n} + 1 \le c.2^n$

$2n + 1 \le c.2^n$

for ex $\boxed{c=2}$ $\boxed{n \ge 0}$

$2n + 1 \le 2.2^n$

$\log_2 n^2 + 1 = O(n)$ ✓

$\log_2 n^2 + 1 = n$

$2 \log_2 n = n - 1$

$\boxed{\text{for } n_0 \quad n = 1}$

$2^n = n$

$2^n = 1$

$n = 0$

2-) $\sqrt{n(n+1)} \stackrel{?}{=} \Omega(n)$

$\sqrt{n(n+1)} \ge cn$

$n(n+1) \ge c^2 n^2$

$\boxed{n_0 = 0}$

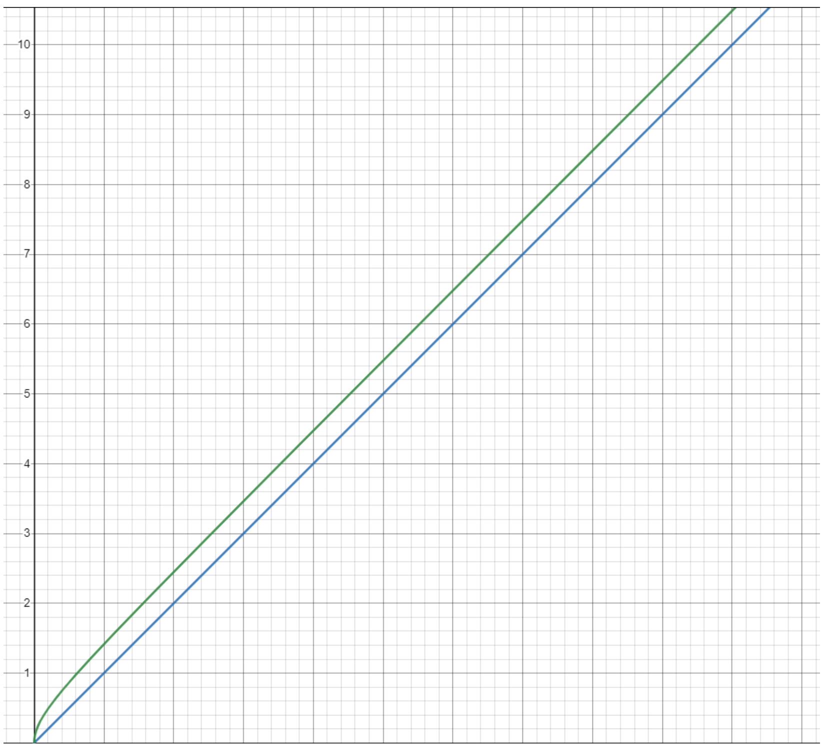
$n^2 + n \ge c^2 n^2$

$n \ge (c^2 - 1) n^2$

$\frac{1}{n} \ge c^2 - 1$

$n > 0$ $\boxed{c = 1}$ $\boxed{n > 0}$

$\sqrt{n(n+1)} \ge \Omega(n)$ ✓



3-) $n^{n-1} = \Theta(n^n)$

$k_1 * |n^n| \leq |n^{n-1}| \leq k_2 * |n^n|$

for $\boxed{k_1 = 0,1}$ and $\boxed{n_0 = 1}$

$0,1 * n^n \leq n^{n-1}$

always true for all $n > 1$

$\boxed{k_2 = 5}$ $\boxed{n_0 = 1}$

$n^{n-1} \leq 5 \cdot n^n$

always true for all $n > 1$

$\boxed{n^{n-1} = \Theta(n^n)}$ ✓

Part 2

$$\lim_{n\to\infty} \frac{n^2}{n^2\log n} = \frac{1}{\log n} = 0 = \boxed{n^2 < n^2\log n}$$

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \frac{1}{n} = -\frac{1}{n^2} = 0 \quad \boxed{n^3 > n^2}$$

$$\lim_{n\to\infty} \frac{n^2}{\sqrt{n}} = \frac{n^2}{n^{1/2}} = \frac{2n}{\frac{1}{2\sqrt{n}}} = 4n\sqrt{n} = \infty \quad \boxed{n^2 > \sqrt{n}}$$

$$\lim_{n\to\infty} \frac{n^2\log n}{\sqrt{n}} = \frac{\log n}{n^{-3/2}} = \frac{\frac{1}{\ln 10 \cdot n}}{n^{-5/2} \cdot -3/2} = n^{3/2} = \infty = \boxed{n^2\log n > \sqrt{n}}$$

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \frac{\frac{1}{\ln 10 \cdot n}}{\frac{1}{2\sqrt{n}}} = \frac{\sqrt{n}}{n} = \frac{1}{2\sqrt{n}} = 0 = \boxed{\sqrt{n} > \log n}$$

$$\lim_{n\to\infty} \frac{n^3}{2^n} = \frac{3n^2}{2^n\ln 2} = \frac{6n}{2^n \cdot \ln^2(2)} \cdot \frac{6}{2^n \cdot \ln^3(2)} = 0 = \boxed{2^n > n^3}$$

for $8^{\log_2 n} = (2^3)^{\log_2 n} = (2^{\log_2 n})^3 = n^3 \quad \boxed{8^{\log_2 n} = n^3}$

$$\lim_{n\to\infty} \frac{10^n}{2^n} = \frac{5^n \cdot 2^n}{2^n} = 5^n = \infty \quad \boxed{10^n > 2^n}$$

$$\lim_{n\to\infty} \frac{n^3}{n^2\log n} = \frac{n}{\log n} = \frac{1}{\frac{1}{\ln 10 \cdot n}} = \infty = \boxed{n^3 > n^2\log n}$$

Result: $10^n > 2^n > n^3 = 8^{\log_2 n} > n^2\log n > n^2 > \sqrt{n} > \log n$

Part 3

a) int p_1 (int my_array[]){

```
for(int i=2; i<=n; i++){          -> O(n)
        if(i%2==0){               -> O(1)
                count++;          -> O(1)
        } else{
                i=(i-1);          -> O(1)
        }
    }
}
```

Time Complexity: O(n)

b) int p_2 (int my_array[]){

```
first_element = my_array[0];                                    -> O(1)
second_element = my_array[0];                                   -> O(1)
        for(int i=0; i<sizeofArray; i++){                       -> O(n)
                if(my_array[i]<first_element){                  -> O(1)
                        second_element=first_element;           -> O(1)
                        first_element=my_array[i];              -> O(1)
                }else if(my_array[i]<second_element){           -> O(1)
                        if(my_array[i]!= first_element){        -> O(1)
                                second_element= my_array[i];    -> O(1)
                        }
                }
}
```

Time Complexity: O(n)

c) int p_3 (int array[]) {

```
return array[0] * array[2];       -> O(1)
}
```

Time Complexity: O(n)

d) int p_4(int array[], int n) {

```
        Int sum = 0                             -> O(1)
        for (int i = 0; i < n; i=i+5)           -> O(n)
                sum += array[i] * array[i];     -> O(1)
        return sum;                             -> O(1)
}
```

Time Complexity: O(n)

e)

```
void p_5 (int array[], int n){
    for (int i = 0; i < n; i++)                          -> O(n)
        for (int j = 1; j < i; j=j*2)                    -> O(logn)
            printf("%d", array[i] * array[j]);           -> O(1)
}
```

Time Complexity: O(nlogn)

f)

```
int p_6(int array[], int n) {
    If (p_4(array, n)) > 1000)                           -> O(1)
        p_5(array, n)                                    -> O(nlogn)
    else printf("%d", p_3(array) * p_4(array, n))        -> O(n)
}
```

Time Complexity: O(nlogn)

g)

```
int p_7( int n ){
    int i = n;                                           -> O(1)
    while (i > 0) {                                      -> O(logn)
        for (int j = 0; j < n; j++)                      -> O(n)
            System.out.println("*");                     -> O(1)
        i = i / 2;                                       -> O(1)
    }
}
```

Time Complexity: O(nlogn)

h)

```
int p_8( int n ){
    while (n > 0) {                                      -> O(logn)
        for (int j = 0; j < n; j++)                      -> O(n)
            System.out.println("*");                     -> O(1)
        n = n / 2;                                       -> O(1)
    }
}
```

Time Complexity: O(n)

The time complexity is not O(nlogn) because the inner loop doesn't return n times at each step. It returns a series: n(1 + 1/2 + 1/4 + 1/8 + 1/16 + ...) and the sum of all the steps in this series is **2*n** which gives us the answer **O(n).**

i)

```
int p_9(n){
        if (n = 0)                              -> O(1)
                return 1                        -> O(1)
        else
                return n * p_9(n-1)             -> O(n)
}
```

Time Complexity: O(n)

j)

```
int p_10 (int A[ ], int n) {
        if (n == 1)                             -> O(1)
                return;                         -> O(1)
        p_10 (A, n − 1);                   -> O(n)
        j = n − 1;                              -> O(1)
        while (j > 0 and A[j] < A[j − 1]) {     -> O(n)
                SWAP(A[j], A[j − 1]);           -> O(1)
                j = j − 1;                      -> O(1)
        }
}
```

T(0) = 0 T(1) = 1 T(2) = 3 T(3) = 7 T(4) = 15 so Easily see that (2^n) - 1 that makes O(2^n)

Time Complexity: O(2^n)

Part 4

a) Explain what is wrong with the following statement. “The running time of algorithm A is at least O(n2)”.

-O(n^2) means that the algorithm is O(n^2) in the worst case scenario. Describe using "at least" is highly meaningless because if we use this sentence like this , we said that O(n^2) as the best-case scenario. This is absolutely wrong, because O(n^2) time complexity would mean the longest time that takes in the worst case. In the best-case scenario, this time can be less than O(n^2).

b) Prove that clause true or false? Use the definition of asymptotic notations.

I-II

$$k_1 * |2^{n+1}| \le |2n| \le k_2 * |2^{n+1}|$$

$$\underbrace{k_1 * |2^n| \le |n|}_{\text{There is no way to proof it!}} \le k_2 * |2^n|$$

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

FALSE!

Same scenario

$$k_1 * |2^{2n}| \le |2n| \le k_2 * |2^{2n}|$$

$$\underbrace{k_1 * |4^n| \le |2n|}_{\text{There is no way to proof it!}} \le k_2 * |4^n|$$

so, $2^{2n} = \Theta(2n)$

FALSE!

III. To Remind that for our informations, we are using Big-O notation for worst case scenario. We are using Theta Notation (Θ-notation) for the certain informations. Long story short for Theta notation enclose the function from above and below. In this question, when we think logically f(n) = O(n^2) gives us info for worst case. g(n) = Θ(n^2) gives us certain info so when we multiply that, we can not talk about Θ or exact values. so The true answer is f(n) * g(n) = O(n^4)

Part 5 -

5-) a) $T(1)=1$ and $T(n)=2T(n/2)+n$, Lets assume that $T(n)=O(n\log n)$ so we have to prove $T(n)\le cn\log n$

$$T(n)\le 2(c(n/2)\log(n/2))+n$$
$$\le cn\log(n/2)+n$$
$$\le cn\log n - cn\log 2+n$$
$$= cn\log n - cn+n$$
$$\le cn\log n \ (\text{for } c\ge 1)$$

so, $T(2)=2T(1)+2=4$

$T(3)=2T(1)+3=5$

OK, fine. now just select c for satisfy the condition on $T(2)$ and $T(3)$ just choose $c=2$ cause:

$$4\le 2.2.\log 2 \quad \& \quad 5\le 2.3\log 3$$

Finally, $T(n)\le 2n\log n$ for all $n\ge 2$, so $T(n) = \boxed{O(n\log n)}$

b-)

n	$T(n)$
0	0
1	1
2	3
3	7
4	15
5	31
6	63
7	127
⋮	⋮

We can easily see that $T(n)=2^n-1$ so the answer is $O(2^n)$ //

Part 6 -

```java
    public static void pairsCountIterative(int[] arr, int sum)
    {
        int count = 0;                                   //O(1)
        for (int i = 0; i < arr.length; i++)             //O(n)
            for (int j = i + 1; j < arr.length; j++)     //O(n)
                if ((arr[i] + arr[j]) == sum)            //O(1)
                    count++;                             //O(1)

        System.out.printf("Count of pairs is %d", count); //O(1)
    }
}
```

Theoretical Time Complexity: O(n^2)

```
For 100 instance time calculation: 1 ms
For 1000 instance time calculation: 5 ms
For 10000 instance time calculation: 19 ms
For 100000 instance time calculation: 4691 ms
```

Results are really close to Time Complexity because of growth rate is so close to n^2

Part 7 -

```java
public static void pairsCounterRecursive(int startIndex, int[] arr, int sum){
    for(int j = startIndex+1; j<arr.length;j++){                      //O(n)
        if(arr[startIndex] + arr[j] == sum)                          //O(1)
            countRecursive++;                                        //O(1)
    }

    if(startIndex < arr.length){                                     //O(1)
        pairsCounterRecursive( startIndex: startIndex+1,arr,sum);    //O(n)
    }
}
```

```
For 100 instance time calculation: 2 ms
For 1000 instance time calculation: 6 ms
For 10000 instance time calculation: 25 ms
```

Theoretical Time Complexity: O(n^2) : For loop calls n * n times because of recursion. So the Results are close with Part 6.