# GTU Department of Computer Engineering
# CSE 222/505 - Spring 2022

## HOMEWORK 2
Atacan BAŞARAN(200104004008)

**1)**

a) $O(f(n))$

By definition $c.f(n) \geq T(n)$ for $n \geq n_0$

$c=1$

$n_0=2$ $\quad 1.n \geq \log_2 n^2 + 1$ for all $n \geq 2$ ✓

TRUE

b) $\Omega(f(n))$

By definition $c.f(n) \leq T(n)$ for $n \geq n_0$

$c=1$

$n_0=1$ $\quad 1.n \leq \sqrt{n.(n+1)}$ for all $n \geq 1$ ✓

TRUE

c) $\Theta(n^n)$

By definition if $n^{n-1} = O(n^n)$ and $n^{n-1} = \Omega(n^n)$
Then $\Theta(n^n)$ is true

| $O(n^n)$ $(c.f(n) \geq T(n))$ | $\Omega(n^n)$ $(c.f(n) \leq T(n))$ |
|---|---|
| $\Rightarrow c=1$ $\quad 1.n^n \geq n^{n-1}$ | $\Rightarrow c = n^{-1}$ ✗ $\quad c.n^n \leq n^{n-1}$ |
| $n_0=1$ | For this equation |
| for all $n \geq 1$ | c has to be in terms of n. |
| TRUE ✓ | But c should be constant |
| | so FALSE ✗ |

Both $O(n^n)$ and $\Omega(n^n)$ are not true

So $\Theta(n^n)$ is FALSE ✗

2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$, $g(n)$ has bigger growth rate

$\neq 0 \Rightarrow f(n) = \Theta(g(n))$, they have same growth rate

$= \infty \Rightarrow g(n) = o(f(n))$, $f(n)$ has bigger growth rate

$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$ growth rate $\log n < \sqrt{n}$

$\lim_{n\to\infty} \frac{\sqrt{n}}{n^2} = 0$ growth rate $\sqrt{n} < n^2$

$\lim_{n\to\infty} \frac{n^2}{n^2 \cdot \log n} = 0$ growth rate $n^2 < n^2 \cdot \log n$

$\lim_{n\to\infty} \frac{n^2 \cdot \log n}{n^3} = 0$ growth rate $n^2 \cdot \log n < n^3$

$\lim_{n\to\infty} \frac{n^3}{8^{\log_2 n}} = 1$ growth rate $n^3 = 8^{\log_2 n}$

$\lim_{n\to\infty} \frac{n^3}{2^n} = 0$ growth rate $n^3 < 2^n$

$\lim_{n\to\infty} \frac{2^n}{10^n} = 0$ growth rate $2^n < 10^n$

Order of growth rates:

$\log n < \sqrt{n} < n^2 < n^2 \log n < n^3 = 8^{\log_2 n} < 2^n < 10^n$

## 3)

### a)

```
int p_1 ( int my_array[]){                                T(n) = O(logn)
        for(int i=2; i<=n; i++){  O(logn)
                if(i%2==0){   θ(1)
                        count++;  θ(1)
                } else{
                        i=(i-1)i;  θ(1)
                }
        }
}
```

### b)

```
int p_2 (int my_array[]){                                          T(n) = θ(n)
        first_element = my_array[0];  θ(1)
        second_element = my_array[0];  θ(1)
        for(int i=0; i<sizeofArray; i++){   θ(n)
                if(my_array[i]<first_element){  θ(1)
                        second_element=first_element;  θ(1)
                        first_element=my_array[i];    θ(1)
                }else if(my_array[i]<second_element){   θ(1
                        if(my_array[i]!= first_element){   θ(1)
                                second_element= my_array[i];   θ(1)
                        }
                }
}
```

**c)**

```
int p_3 (int array[]) {                                T(n) = θ(1)
        return array[0] * array[2];  θ(1)
}
```

**d)**

```
int p_4(int array[], int n) {                          T(n) = θ(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)
}
```

**e)**

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

**f)**

```
int p_6(int array[], int n) {
        If (p_4(array, n)) > 1000)  θ(n) + θ(1)(Comparison) = θ(n)
                p_5(array, n)  θ(n*logn)
        else printf("%d", p_3(array) * p_4(array, n))   θ(1) (p_3)
                                                       + θ(n) (p_4)
}                                                      + θ(1) (printf) = θ(n)
```

**Best case :** θ(n)
**Worst case:** θ(n*logn)
**Average case:** O(n*logn),Ω(n)

**g)**

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

**T(n) = θ(logn)*θ(n)  = θ(logn*n)**

**h)**

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

**T(n) = O(logn)*θ(logn) = O(log^2 n)**

**i)**

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

**Best case: θ(1)**
**Worst case: θ(n)**
**Average case: O(n)**

**Program runs for n times,**
**T(n) = θ(n)**

**j)**

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

**Best case: θ(1)**
**Worst case: θ(n^2)**
**T(n) = O(n^2)**

4)

a) Big O notation is used for upper bound so it can't be used for "at least" (lower bound) judgment. Also $c$ and $n_0$ values can change. Therefore, we can not reach a clear judgment.

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

By the definition if $2^{n+1} = O(2^n)$ and $2^{n+1} = \Omega(2^n)$

Then $\Theta(2^n)$ is true

$O(2^n) \Rightarrow (c.f(n) \geq T(n))$

$\Rightarrow c=3$ $\quad 3.2^n \geq 2^{n+1}$

$n_0=1$ $\quad$ for all $n \geq 1$

TRUE ✓

$\Omega(2^n) \Rightarrow (c.f(n) \leq T(n))$

$c=1$ $\quad 1.2^n \leq 2^{n+1}$

$n_0=1$ $\quad$ for all $n \geq 1$

TRUE ✓

Both $O(2^n)$ and $\Omega(2^n)$ are true so $\Theta(2^n)$ is TRUE ✓

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

By same asymptotic definition as above.

$O(2^n) \Rightarrow (c.f(n) \geq T(n))$

$\Rightarrow c=10^n$ X $\quad c.2^n \geq 2^n.2^n$

$n_0=1$ $\quad$ c should be equal or bigger than $2^n$ (ex. $10^n$) but c should be constant

so $O(2^n)$ is FALSE X

We dont need to look $\Omega(2^n)$ because $O(2^n)$ already false

$\boxed{\text{So } \Theta(2^n) \text{ is FALSE X}}$

III) $O(n^2)$ notation can represent linear function $(f(n) = n)$

$\Theta(n^2)$ notation represent quadratic function $(g(n) = n^2)$

$f(n) * g(n)$ can be $n * n^2 = n^3$ and $n^3$ can not be represented as $\Theta(n^4)$ because $\Theta(n^4)$ represent fourth degree function.

5)

a) $T(n) = 2T(n/2) + n$

$T(n) = 2^2 T(n/2^2) + n + n$

$T(n) = 2^3 T(n/2^3) + 3n$

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k.n$

$T(n) = n.1 + n.\log n$

$\hookrightarrow O(n \log n)$

$T\left(\frac{n}{2^k}\right) = T(1)$

$\frac{n}{2^k} = 1$

$k = \log n$

b) $T(n) = 2T(n-1) + 1$

$T(n) = 2[2T(n-2) + 1] + 1$

$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$

$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^1 + 1$

$T(n) = 2^n . T(0) + 2^n - 1$

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

$\hookrightarrow O(2^n)$

$T(n-k) = T(0) = 0$

$k = n$

# 6)

```java
public static int findPair(int arr[], int sum) {
    int counter =0;  θ(1)

    for (int i = 0; i < arr.length; i++) {  θ(n)
        for (int j = i+1; j < arr.length; j++) { θ(n)
            if (arr[i] + arr[j] == sum)  θ(1)
                counter++;  θ(1)
        }
    }
    return counter;  θ(1)
}
```

$T(n) = \theta(n)*\theta(n) = \theta(n\hat{}2)$

**Running time in 1000 size array(millisecond) : 3**

**Running time in 10 000 size array(millisecond) : 27**

**Running time in 100 000 size array(millisecond) : 2151**

**Running time in 1 000 000 size array(millisecond) : 220 140**

**Size 1k to 10k time increase 9x**
**Size 10k to 100k time increase 80x**
**Size 100k to 1million time increase 102x**

**Theoretically, time should increase 100x when i increase size 10x because of time complexity θ(n^2)**

**Theoretical and practical values almost same in big sizes**

# 7)

```
public static int recursiveFindPair(int arr[], int sum, int index, int index2) {
    if (index >= arr.length) θ(1)
        return 0; θ(1)

    if (index2 >= arr.length) { θ(1)
        return recursiveFindPair(arr, sum, index + 1, index + 2); θ(n)
    }
    else if (arr[index] + arr[index2] == sum) θ(1)
        return 1 + recursiveFindPair(arr, sum, index, index2 + 1); θ(n)

    return 0 + recursiveFindPair(arr, sum, index, index2 + 1); θ(n)
}
```

T(n) = θ(n)*θ(n) = θ(n^2)

**Running time in 5 size array(microsecond) : 3**

**Running time in 10 size array(microsecond) : 6**

**Running time in 50 size array(microsecond) : 213**

**Running time in 100 size array(microsecond) : 592**

**Size 5 to 10 time increase 2x**
**Size 10 to 50 time increase 35x**
**Size 50 to 100 time increase 2.77x**

**Theoretically, time should increase 4x when i increase size 2x because of time complexity θ(n^2)**
**Theoretical and practical values almost same**