Emre YILMAZ

1901042606

Computer Engineering

NOTE: Question-3 is at the end of the PDF.

1-)

a-) $\log_2 n^2 + 1 = O(n)$

* $T(n) = O(f(n))$
* $c \cdot f(n) \geq T(n)$

$\log_2 n^2 \leq c \cdot n$

$\frac{\log_2 n_0^2}{n_0} = c \quad \Rightarrow \quad n_0 = 2, \quad c = 1,5$

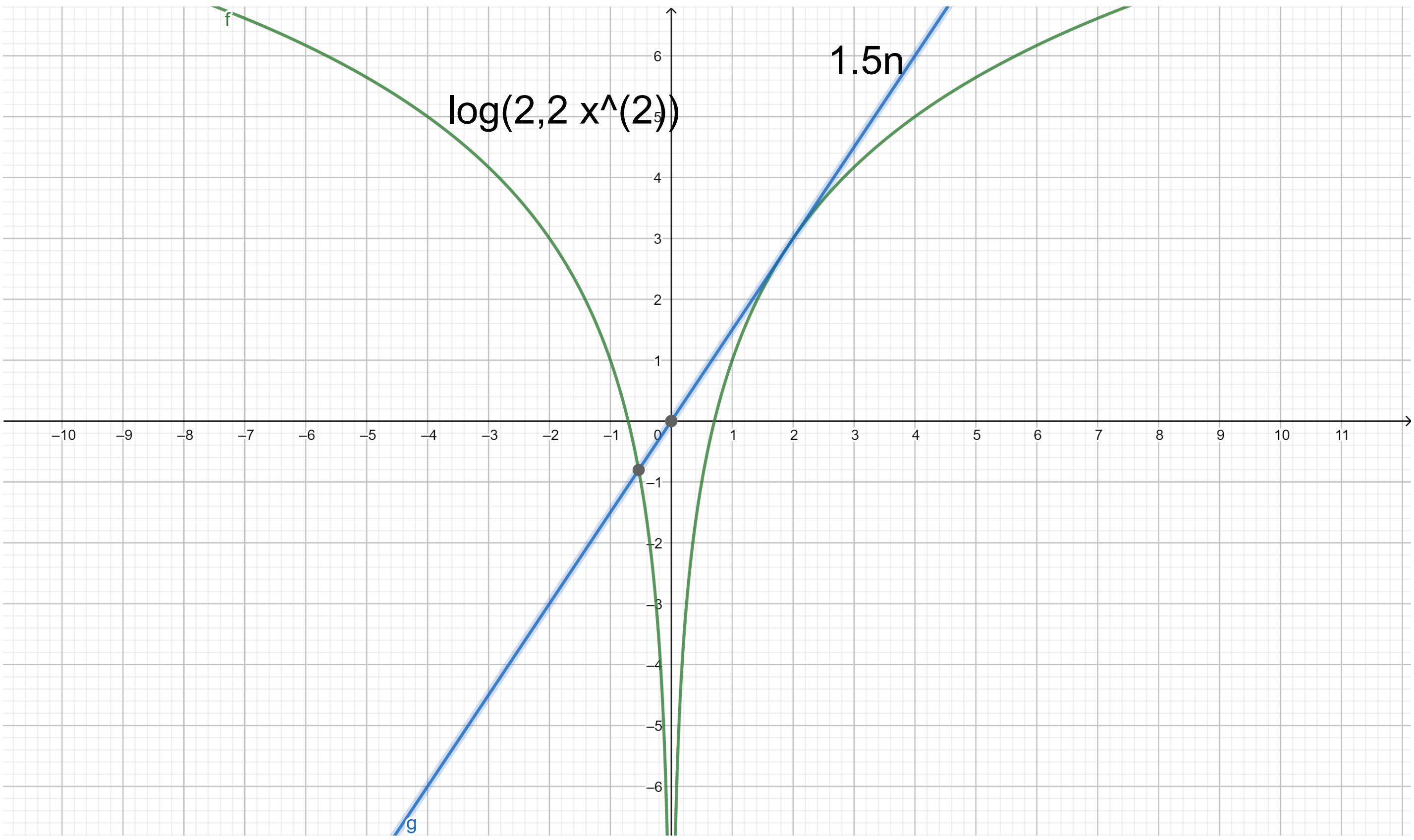
$\rightarrow \quad 1,5n \geq \log_2 n^2 + 1 \quad , \quad n > 2$ ✓

$1,5n \geq \log_2 2n^2$ ✓

$1,5n = \log_2 2n^2 \quad , \quad$ if $n = 2$ ✓

* $1,5n$ is upper limit for $\log_2 2n^2$

The statement is <u>true.</u>



Scanned with CamScanner

b.) $\sqrt{n(n+1)} = \Omega(n)$

* $T(n) = \Omega(f(n))$
* $c \cdot f(n) \le T(n)$

$n \cdot c \le \sqrt{n \cdot (n+1)}$

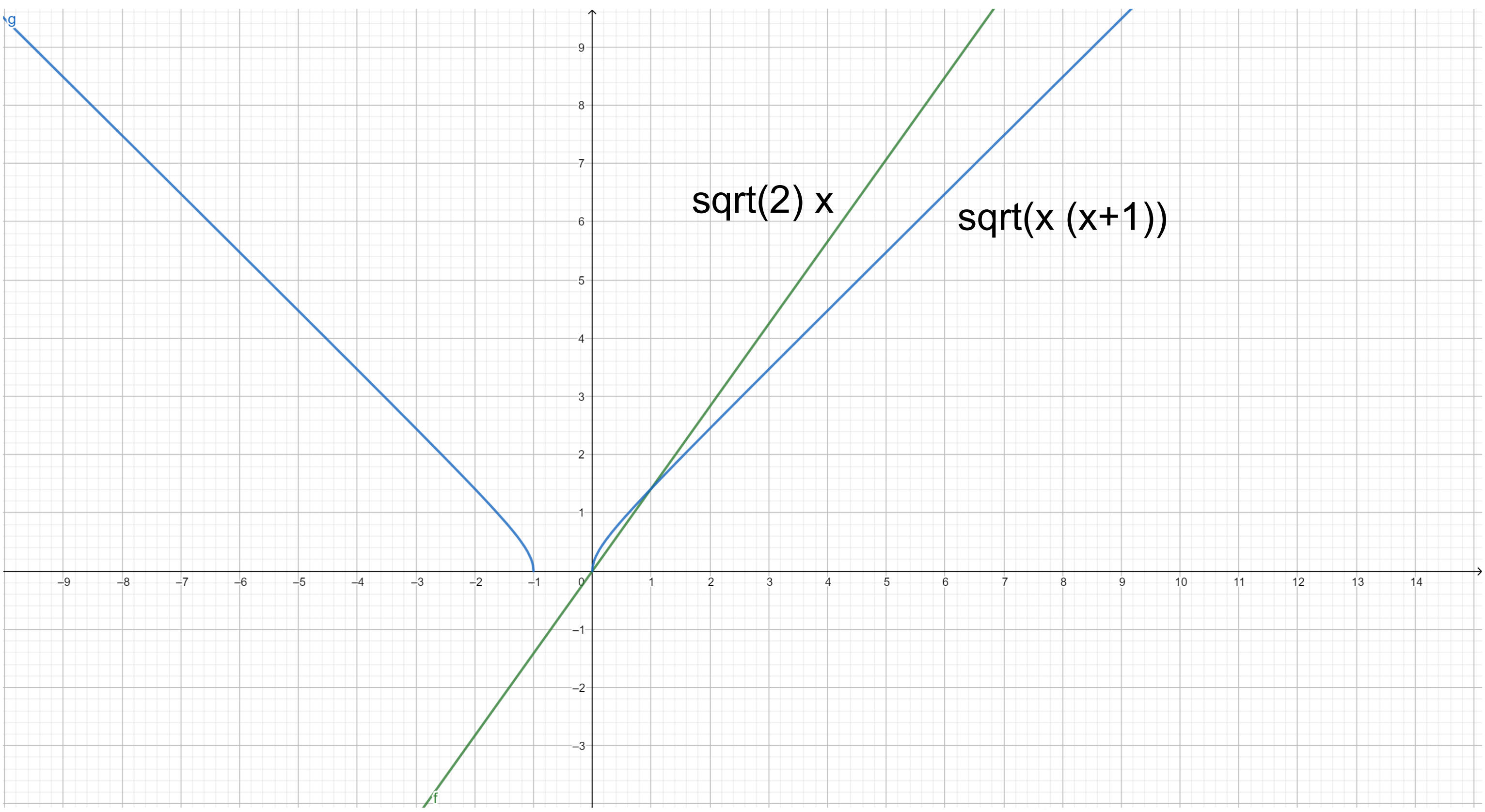
$c = \frac{\sqrt{n_0(n_0+1)}}{n_0} \Rightarrow \begin{array}{l} n_0 = 1 \\ c = \sqrt{2} \end{array}$

$\rightarrow$ • $\sqrt{2} \cdot n \le \sqrt{n \cdot (n+1)}$, $n > 1$

• $2n^2 \le n^2 + n$, for $n = 3$, the statement is false.

• $\sqrt{2}n$ is NOT the lower bound for $\sqrt{n \cdot (n+1)}$.

* The statement is false.



c-)

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

* $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$

* $c_1 \cdot h(n) \leq f(n) \leq c_2 \cdot h(n)$, $\forall n \geq n_0$

* $\underbrace{c_1 \cdot n^n \leq n^{n-1}}_{*_2} \;\; \underbrace{n^{n-1} \leq c_2\, n^n}_{*_1}$, $\forall n \geq n_0$

$*_1$ $c \cdot f(n) \geq T(n)$

$c \cdot n^n \geq n^{n-1}$

$c \geq \frac{1}{n}$ $\Rightarrow$ $n_0 = 1$, $c = 1$

$n^n > n^{n-1}$, we have an upper bound $T(n) = O(h(n))$

$n^{n-1} = O(n^n)$

$*_2$ $c \cdot f(n) \leq T(n)$

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

$c \leq \frac{1}{n}$ $n_0 = (5, 1)$, $c = (1/5, 1)$

$1/5 \leq 1/n$ $n > 5$ ✗

$1 \leq 1/n$ $n > 1$ ✗

} we do not have a lower bound $T(n) = \Omega(h(n))$

$n^{n-1} = \Omega(n^n)$

☆ So, The statement is <u>false</u>.

2-)

* $10^n > 2^n > n^3 = 8^{\log_2 n} > n^2 \log n > n^2 > \sqrt{n} > \log n$ ✓

- $8^{\log_2 n} = n^{\log_2 8} = n^3 = n^3 \rightarrow$ Comparing n^3 and $8^{\log_2 n}$

- $\lim_{n\to\infty} \frac{n^2 \log n}{n^3} = \lim_{n\to\infty} \frac{\log n}{n}$
$=$ Apply L'Hopital $\rightarrow \lim_{n\to\infty} \frac{1}{\ln 10 \cdot n}$
$= \ln 10 . \lim_{n\to\infty} \frac{1}{n} = 0$, So, $n^3 > n^2 \log n$
Comparing n^3 and $n^2 \log n$

- $\lim_{n\to\infty} \frac{n^2 \log n}{n^2} = \lim_{n\to\infty} \log n = \infty$
So, $n^2 \log n > n^2$
Comparing $n^2 \log n$ and n^2

- $\lim_{n\to\infty} \frac{n^2}{\sqrt{n}} = \lim_{n\to\infty} n^{3/2} = \infty$
So, $n^2 > \sqrt{n}$
Comparing n^2 and $\sqrt{n}$

- $\lim_{n\to\infty} \frac{\sqrt{n}}{\log n} \rightarrow$ Apply L'Hopital $= \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n}}}{\frac{1}{n \ln 10}}$
$= \lim_{n\to\infty} \frac{1}{2} \ln 10 \sqrt{x} = \infty$, So, $\sqrt{n} > \log n$
Comparing $\sqrt{n}$ and $\log n$

- $\lim_{n\to\infty} \frac{n^3}{2^n} \rightarrow$ Apply L'hopital $\times 3 = \lim_{n\to\infty} \frac{6}{\ln^3 2 . 2^x}$
$= \frac{6}{\ln^3 2} \lim_{n\to\infty} \frac{1}{2^x} = 0$, So, $2^n > n^3$
Comparing 2^n and n^3

Scanned with CamScanner

4-)

a-) The big-O notation denotes the upper bound.

* A function with $O(n^2)$ complexity can run $\Theta(1)$ complexity.
* So, in this case, $O(n^2)$ run-time cannot be at least $O(n^2)$. But, we can say that $O(n^2)$ run-time is at most $O(n^2)$.

b-)

I.) $2^{n+1} = \Theta(2^n)$ ✓

* $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$

* $\underbrace{c_1 \cdot 2^n \leq 2^{n+1}}_{*_2} \; \underbrace{\leq c_2 \cdot 2^n}_{*_1} \quad , \forall n \geq n_0$

$*_1 = c_2 \cdot 2^n \geq 2^{n+1}$

$c_2 \geq \frac{2^{n+1}}{2^n}$ $\Big\}$ $n_0 = 1$, $c = 2$

$2 \cdot 2^n \geq 2^{n+1}$

$2^{n+1} \geq 2^{n+1}$ ✓ We have an upper bound $T(n) = O(h(n))$

$*_2 = c_1 \cdot 2^n \leq 2^{n+1}$

$c_1 \leq \frac{2^{n+1}}{2^n}$ $\Big\}$ $n_0 = 1$, $c = 2$

$2^{n+1} \leq 2^{n+1}$ ✓ We have an lower bound $T(n) = \Omega(h(n))$

* We have both $O(h(n))$ and $\Omega(h(n))$. So, $2^{n+1} = \Theta(2^n)$ statement is true.

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

* $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$
* $\underbrace{c_1 . 2^n \leq 2^{2n}}_{*_2} \underbrace{\leq c_2 . 2^n}_{*_1}$

$*_1 \Rightarrow c_2 . 2^n \geq 2^{2n}$

$\left. c_2 \geq 2^n \right\} \begin{matrix} n_0 = 1 \\ c = 2 \end{matrix}$

$2.2^n \geq 2^{2n}$

$2 \geq 2^n, \forall n \geq 1$ X

* We do not have an upper bound. So, the notation $2^{2n} = \Theta(2^n)$ is <u>false</u>

III.)

* $O(n^2) \times \Theta(n^2) = O(n^4)$

$\downarrow$ may be $\Theta(1)$. So, the multiplication is $O(n^4)$

* $\Theta(n^4)$ denotes both upper and lower bounds. But, $O(n^4)$ is only denotes upper bound. Hence, the statement is <u>false</u>

5-)

a-)

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n \neq 1 \end{cases}$$

→ First 4 steps :

$$2 \times \left(2 \times \left(2 \times \left(2 \times T(n/16) + n/8\right) + n/4\right) + n/2\right) + n$$

$$= 16 \times T(n/16) + 4n$$

$$= 2^4 \times T(n/2^4) + 4\,n$$

→ K'th step

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

→ Ending step

$$n/2^k = 1$$

$$n = 2^k$$

$$\log_2 n = \log_2 2^k$$

$$\log_2 n = k$$

$$= 2^k . T(1) + k.n$$ → General Rule

$$= n. \underbrace{T(1)}_{1} + n.\log n$$

→ Time complexity $= \Theta(n + n\log n) = \boxed{\Theta(n\log n)}$

Scanned with CamScanner

b-)

$$T(n) = \begin{cases} 0 & n=0 \\ 2T(n-1)+1 & n \neq 0 \end{cases}$$

→ First 4 steps

$$2 \times \left(2 \times \left(2 \times \left(2 \times T(n-4) + 1\right) + 1\right) + 1\right) + 1$$

$$= 2^4 \times T(n-4) + 15$$

→ First 5 steps

$$= 2^5 \times T(n-5) + 31$$

→ k'th step

$$= 2^k \times T(n-k) + (2^k - 1)$$

→ Ending step

$$n-k = 1$$
$$n = k$$

$$= 2^k \times T(1) + (2^k - 1)$$ → General Rule

$$= 2^n \times T(1) + 2^n - 1$$

$$= 2^n - 1$$

→ Time complexity $= \Theta(2^n - 1) = \Theta(2^n)$

Q-6)

```c
void search(int *arr, int n /* array size */, int sum /* target sum */)
{
    for(int i=0;i<n;i++)
    {
        for(int j=i;j<n;++j)
        {
            if((arr[i]+arr[j])==sum)
                printf("The pair: %d and %d\n",arr[i],arr[j]);
        }
    }
}
```

► The algorithm is above.

► Theoritically, time complexity of the function is $\Theta(n^2)$

► With 10 input, run-time of the function is: 0.0000146000 s

```
The pair: 0 and 5
The pair: 1 and 4
The pair: 2 and 3
Time: 0.0000146000
```

► With 100 input, run-time of the function is: 0.0000302000

```
The pair: 0 and 5
The pair: 1 and 4
The pair: 2 and 3
Time: 0.0000302000
```

► With 1000 input, run-time of the function is: 0.0012281000

```
The pair: 0 and 5
The pair: 1 and 4
The pair: 2 and 3
Time: 0.0012281000
```

► With 10.000 input, run-time of the function is: 0.0919501000

```
The pair: 0 and 5
The pair: 1 and 4
The pair: 2 and 3
Time: 0.0919501000
```

► With 100.000 input, run-time of the function

```
The pair: 0 and 5
The pair: 1 and 4
The pair: 2 and 3
Time: 9.4981378000
```

- Our expectation is that for every 10x the number of inputs, the run-time will increase 100x.
- When we increase the number of inputs from 10 to 100, we see that the run time increases 2 times.
- When we increase the number of inputs from 100 to 1000, we see that the run time increases 40 times.
- When we increase the number of inputs from 1000 to 10,000, we see that the run time increases 81 times.
- When we increase the number of inputs from 10,000 to 100,000, we see that the run time increases 103 times.
-
- ✓ As the number of inputs increases, we see that the theoretical value of time complexity is correct. ✓

Q-7)

```c
void search_recursion(int *arr, int n /* array size */,
                      int sum /* target sum*/, int number /* previous number */)
{
    if(n<0) return;

    if(number+arr[n]==sum)
        printf("The pair: %d and %d\n",number,arr[n]);

    search_recursion(arr,n-1,sum,number);

    if(arr[n] == number)
        search_recursion(arr,n-1,sum,arr[n-1]);
}
```

- The algorithm is above.
- Theoritically, time complexity of the function is $\Theta(n^2)$
- With 10 input, run-time of the function is: 0.0000820000 s

```
The pair: 5 and 0
The pair: 4 and 1
The pair: 3 and 2

Time: 0.0000820000
```

- With 100 input, run-time of the function is: 0.0001430000

```
The pair: 5 and 0
The pair: 4 and 1
The pair: 3 and 2

Time: 0.0001430000
```

- With 1000 input, run-time of the function is: 0.0045320000

```
The pair: 5 and 0
The pair: 4 and 1
The pair: 3 and 2

Time: 0.0045320000
```

► With 10.000 input, run-time of the function is: 0.4468600000

```
The pair: 5 and 0
The pair: 4 and 1
The pair: 3 and 2

Time: 0.4468600000
```

► With 100.000 input, run-time of the function is: 54.8830620000

```
The pair: 5 and 0
The pair: 4 and 1
The pair: 3 and 2

Time: 54.8830620000
```

- Our expectation is that for every 10x the number of inputs, the run-time will increase 100x.
- When we increase the number of inputs from 10 to 100, we see that the run time increases 1.7 times.
- When we increase the number of inputs from 100 to 1000, we see that the run time increases 31 times.
- When we increase the number of inputs from 1000 to 10,000, we see that the run time increases 98 times.
- When we increase the number of inputs from 10,000 to 100,000, we see that the run time increases 122 times.
-
- ✓ As the number of inputs increases, we see that the theoretical value of time complexity is correct. ✓
- In addition, we see that the recursion algorithm is more costly than the iterative algorithm in this implementation. The recursion algorithm also reaches the result **$\Theta(n^2)$** time complexity value that we expect faster.

Q-3.)

a-)

```
int p_1 ( int my_array[]){
    for(int i=2; i<=n; i++){
        if(i%2==0){  condition: Θ(1)
            count++; Θ(1)
        } else{
            i=(i-1)i; Θ(1)
        }
    }
}
```

Θ(logn)   Θ(1)

► If condition in the loop, and expressions in the if and else blocks are always constant time as indicated in the code above.
►Inside the loop, ***always***, the condition will be true only in the first step, and in every other step the else condition will run, in which ***i*** is multiplied by the number that is one less. The time complexity of this function is **θ(logn)** because the loop variable ***i*** is multiplied at each step.
► i variable in the loop respectively: *2, 3, 7, 43, 1807, 3263443* ...

The answer is ***Θ(logn)***

b)

```
int p_2 (int my_array[]){
        first_element = my_array[0];  Θ(1)
        second_element = my_array[0]; Θ(1)
        for(int i=0; i<sizeofArray; i++){
            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)
                }
            }
}
```

Time complexity of the loop is Θ(n)

Θ(1)

Θ(1)

Time complexity of this conditional statement is: Θ(1)

► First two assignment statements are constant time **Θ(1)**
► All the if conditions and all the assignment statements in the if and else-if blocks are constant time. **Θ(1)**
So, the time complexity of the conditional statment is **Θ(1)**
►Time complexity of the loop is **Θ(sizeOfArray)** which simplified **Θ(n)**
► Time complexity of the p_2 function is **Θ(n)**

The answer is ***Θ(n)***

c)

```
Θ(1)  int p_3 (int array[]) {
              return array[0] * array[2];  Θ(1)
      }
```

The *multiplication* and *return* statements are *always* constant time Θ(1)
So, p_3 function's time complexity is Θ(1)

The answer is: **<u>Θ(1)</u>**

d)

```
        int p_4(int array[], int n) {
                 Int sum = 0  Θ(1)
Θ(n)   Θ(n)      for (int i = 0; i < n; i=i+5)
                          sum += array[i] * array[i];  Θ(1)
                 return sum;  Θ(1)
        }
```

Loop runs always n/5 times. Complexity of the expressions in the loop are always Θ(1).
So, time complexity of p_t is Θ(n/5). We can simplify it as <u>Θ(n)</u>

<u>The answer</u> is: ***Θ(n)***

e)

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

$\Theta(\log n)$ $\Theta(1)$

- Inner loop always runs *logn* times. So, its complexity is **Θ(logn).**
- Outer loop always runs *n* times, we have to multiply n and logn to get time complexity of the all loop.
- Time complexity of the loop is **Θ(logn)** * **Θ(n)** = **Θ(nlogn)**
- Time complexity of the p_5 function is **Θ(nlogn)**

The answer is: **Θ(nlogn)**

f)

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

O(nlogn) **Θ(n)** **Θ(nlogn)** **Θ(n)**

- Time complexity of the condition is: **Θ(n)**
- Time complexity of the expression in the if block is **Θ(nlogn)**
- Time complexity of the expression in the else block is **Θ(1)** * **Θ(n)** = **Θ(n)**
- T(n) = T(*condition*) + max(T(*if-block*), T(*else-block*)

  T(n) = Θ(n) + O(nlogn) = O(nlogn)

The answer is: **O(nlogn)**

g)

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

▶ The inner loop runs *n* time. So, time complexity of the inner loop is: **Θ(n)**
▶ The control variable *i* decreases continuously by dividing by two. So, time complexity of the outer loop is Θ(logn). To get time complexity of the all loop, we have to multiply **Θ(n)** by **Θ(logn)**.
▶ Time complexity of the all loop is: **Θ(n)** * **Θ(logn)** = **Θ(nlogn)**

The answer is: **Θ(nlogn)**

h)

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

▶ The inner loop runs *n* time. So, time complexity of the inner loop is: **Θ(n)**
▶ The control variable *i* decreases continuously by dividing by two. So, time complexity of the outer loop is Θ(logn). To get time complexity of the all loop, we have to multiply **Θ(n)** by **Θ(logn)**.
▶ Time complexity of the all loop is: **Θ(n)** * **Θ(logn)** = **Θ(nlogn)**

The answer is: **Θ(nlogn)**

i)

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

This is just looks like the iteration for(int i=0;i<n;i++)

So, its complexity is **Θ(n)**

j)

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

Θ(n)

► *NOTE:* It is assumed that the SWAP function is runs constant time **Θ(1)** . If the time complexity of the SWAP function is **Θ(n)**, The complexity of the function is **$\Theta(n^3)$.**

► The recursion process is Θ(n)

► The loop runs Θ(n)

► So, the all function's time complexity is **$\Theta(n^2)$**

The answer is: **$\Theta(n^2)$**