1)

a) $O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0 \}$

$\log_2 n^2 + 1 \leq c.n \Rightarrow \log_2 n^2 \leq c.n - 1$

$\Rightarrow n^2 \leq 2^{cn-1}$ $\Rightarrow$ it's true for $c=1$ and $\forall n \geq 7$

b) $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $c * g(n) \leq f(n)$ for all $n \geq n_0 . \}$

$f(n) \geq c * g(n)$

$\sqrt{n^2+n} \geq c * n$ $\rightarrow$ Since power of n^2 bigger than power of n, n can be ignored.

$\downarrow$

$n \geq c * n$ $\rightarrow$ for $c=1$ and all n, its true.

c) $\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0 \}$

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

$\Rightarrow (n-1) \log n = n \log n$

According to the asymptotic notation rules, lower order terms and constants are ignored

$\downarrow$

$n \log n = n \log n$

$f(n) = \Theta(g(n))$ $\longrightarrow$ its true for all n

2) $\lim\limits_{n\to\infty} \frac{\cancel{n^2}}{\cancel{n^3}} = \frac{1}{n} = \frac{1}{\infty} = 0$, so $n^3 > n^2$

$8^{\log_2 n} = n^{\log_2 8} = n^3 \longrightarrow n^3 = 8^{\log_2 n} > n^2$

$\lim\limits_{n\to\infty} \frac{\cancel{n^2}\log n}{\cancel{n^3}} = \frac{\log n}{n} = 0$, so $n^3 > n^2 \log n$

$\lim\limits_{n\to\infty} \frac{\cancel{n^2}\log n}{\cancel{n^2}} = \log n = \infty$, so $n^2 \log n > n^2$

$\lim\limits_{n\to\infty} \frac{\sqrt{n}}{n^2 \log n} = \frac{n^{\cancel{1/2}}}{n^{\cancel{2}} \log n} = \frac{1}{n^{3/2} \log n} = 0$, so $n^2 \log n > \sqrt{n}$

$\lim\limits_{n\to\infty} \frac{\sqrt{n}}{\log n} = \frac{\frac{1}{2}\log n}{\log(\log n)} = \infty$, so $\sqrt{n} > \log n$ (log (n) > log (log n), because n grows faster than log (n))

$\lim\limits_{n\to\infty} \frac{10^n}{2^n} = 5^n = \infty$, so $10^n > 2^n$

$\lim\limits_{n\to\infty} \frac{2^n}{n^3} = \frac{1}{\log 2} \lim\limits_{n\to\infty} \frac{3n}{2^n} = \frac{1}{\log 2^2} \lim\limits_{n\to\infty} \frac{3}{2^n} = \infty$, so $2^n > n^3$

Order of functions growth rate is;

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

a)

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

➔ This algorithm called Sylvester's sequence

➔ Program enters this line of code half of a time

➔ Program enters this line of code half of a time too.

➔Complexity of this algorithm is O(π*x/logloglog x)

b)

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

All assignments and random accesses are done in constant time. For loop executed for a sizeofArray time, and 'i' don't change inside for loop. Constants and lower order terms are ignored. So complexity of this algorithm is **θ(sizeofArray) = θ(n)**

c)

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

All processes are done in constant time. So complexity of this algorithm is **θ(1)**

d)

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

First line done in constant time. For statement executes $\frac{n}{5} + 1$ times. Inside of for loop executes for a $\frac{n}{5}$ time and 'i' don't change inside the loop. Lower order terms and constants can be ignored. So complexity of this algorithm is **θ(i) ➔ 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]);
}
```

First loop executes n times. Complexity of first loop is **θ(n).** Second loop grows exponentially. It executes log(i) times. Since 'i' grows up to n, we can say second loop's complexity is **O(logn)**. So, whole algorithm's complexity is **O(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))
}
```

If statement's complexity is **θ(n)**. inside of if's complexity is **θ(nlogn)** too. Else's complexity is **θ(1)* θ(n) = θ(n)**. So, best case of algorithm is **Ω(n)**, worst case of algorithm is **O(nlogn).**

g)

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

While loop's complexity is **θ(logn)**, for loop's complexity is **θ(n)**. So complexity of this algorithm is **θ(nlogn).**

h)

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

While loop's complexity is **θ(logn)**. For loop executes logn times too. So for loop's complexity is **θ(logn)** too. Overall complexity is **θ(logn) * θ(logn) = θ($\log^2 n$).**

.

i)

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

Let this algorithm's complexity be T(n). if statement's complexity is 1. Else line's complexity is T(n-1). So T(n)=T(n-1)+1. We can say "T(n) = T(n-1)+1 when n>0", "T(n) = 1 when n=0" according to the algorithm. This means, recursion will continue until n became 0. For T(n)=T(n-k)+k, assume n-k=0 ➔n=k. ➔T(n) = T(0)+n ➔So T(n) becames : T(n)=1+n ➔ Complexity is T(n) = **θ(n).**

j)

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

Let this algorithm's complexity be T(n). Loop's complexity is n. Recursive line's complexity is T(n-1). Return line's complexity is 1. We can say "T(n) = T(n-1)+n when n>1", "T(n) = 1 when n=1" according to the algorithm. This means, recursion will continue until n became 1. For T(n)=T(n-k)+(n-(k-1))+(n-(k-2))+….+(n-1)+n. Assume n-k=0➔n=k ➔T(n) = T(0)+1+2+3+…..+(n-1)+n. ➔ T(n)=1+$\frac{n*(n+1)}{2}$ ➔So, T(n) becames : T(n) = $1+\frac{(n^2+n)}{2}$. Complexity is T(n) = **θ($\boldsymbol{n^2}$).**

4)

a) Big O notation is used to find upper bound of running time of the algorithms. Algorithm's upper bound of running time may be $O(n)$ or $O(1)$ at least. we can't generalize that "Algorithms upper bound has to be $O(n^2)$ at least". So, statement is false.

b)

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

$\Rightarrow f(n) = 2^{n+1} = 2.2^n$

$\Rightarrow c_1 . 2^n \leqslant 2.2^n \leqslant c_2 . 2^n$ $\left.\begin{matrix} c_1 = 1 \\ c_2 = 2 \\ n = 1 \end{matrix}\right\}$ Proves this clause $\forall n \geqslant 1 \rightarrow$ True.

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

$f(n) = 2^{2n} \Rightarrow c_1 . 2^n \leqslant 2^{2n} \leqslant c_2 . 2^n$ $\left.\begin{matrix} c_1 = 1 \\ c_2 = 2 \\ n_0 = 1 \end{matrix}\right\}$ Proves this clause.

$\forall n \geqslant 1 \rightarrow$ True.

III) $\left.\begin{matrix} f(n) = O(n^2) \\ g(n) = \Theta(n^2) \end{matrix}\right\}$ $\begin{matrix} f(n) \leqslant c_1 . n^2 \\ c_2 n^2 \leqslant g(n) \leqslant c_3 n^2 \end{matrix}$

$f(n) * g(n) \leqslant c_1 * c_3 * n^4$) We obtain upper bound

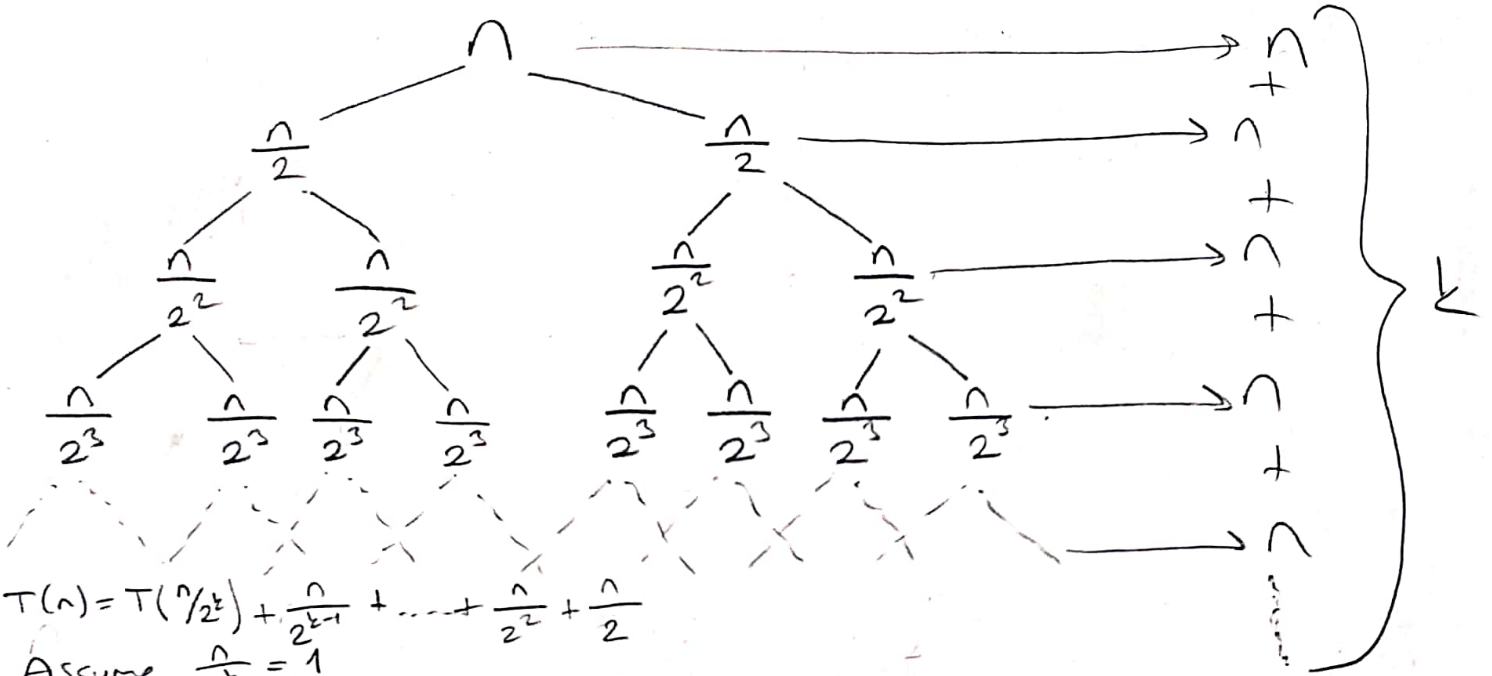
We can't obtain lower bound of running time.

$f(n) * g(n) = O(n^4) \longrightarrow$ its disproved.

not $\Theta(n^4)$

5)

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



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

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

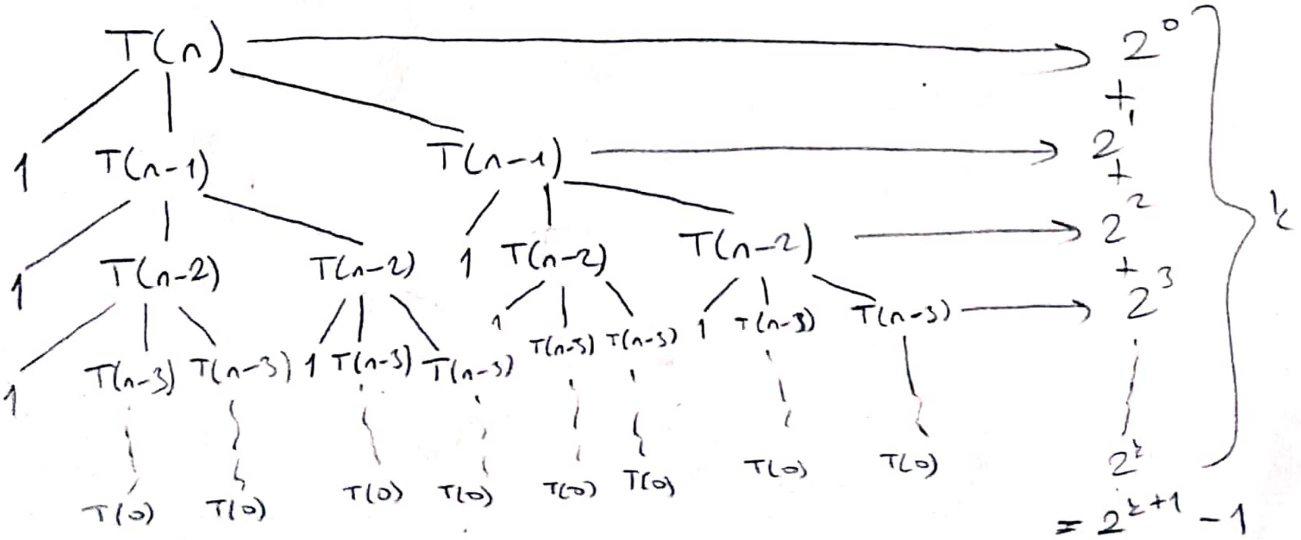
$\Rightarrow n = 2^k \Rightarrow \underline{k = \log n}$ $\qquad k.n$

$T(n) = k.n = n.\log n$

$\Rightarrow O(n \log n)$

---

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



$= 2^{k+1} - 1$

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

Assume $n-k=0$

$\Rightarrow n=k$

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

$\Rightarrow T(n) = \Theta(2^n)$

6) Time complexity of iterative algorithm is $T(n) = \Theta(n^2)$.

I used 3 different size of arrays (10, 100, 200) to compare my theoretical result and test result.

My test results are;

10 element array → 0.041 ms
100 element array → 3.042 ms
1000 element array → 13.672 ms

I take my 10 element array's runtime as reference to compare other arrays runtime.

$$\text{Computer constant } k = \frac{\text{Time}}{T(n)} = \frac{0.041}{n^2} = \frac{0.041}{100} = \underline{0.00041}$$

Theoretical result of 100 element array;

$\text{Time} = T(n) * k \Rightarrow \text{Time} = 100^2 * 0.00041$

$\Rightarrow \text{Time} = 4,1\text{ms}$ → Theoretical time

Test = 3,042 ⇒ 4,1 ~ 3,042
approximately same

Theoretical result of 1000 element array;

$\text{Time} = 200^2 * 0,00041 = 16,4\text{ms}$ → Theoretical time

Test = 13,672ms ⇒ 16,4 ~ 13,672
approximately same

7) My test results are;

10 element array → 0,9ms
15 element array → 3,814ms
20 element array → 36,546ms

Time Complexity of recursive method;

$\underline{T(n) = 2^n}$

```java
public static void pairsOfNums(int[] myArr, int sum){
    int i,j,n=myArr.length;
    for(i=0;i<n;i++){                           //Program enters n+1 times here
        for(j=0;j<n;j++){                       //Program enters n(n+1) times here
            if(myArr[i] + myArr[j] == sum)      //Program enters n*n times here
                continue;
        }                               //Could be printing nums or assigning to another
    }                                   //array in this line. But, to spend less time in
}                                       //method, I left if body empty.
```

```
Without Recursion

10 element array = 0.041ms
100 element array = 3.042ms
200 element array = 13.672ms
```

```java
public static boolean pairsOfNumsRecursive(int[] myArr, int myindex, int sum){
    if (sum == 0)
        return true;
    if (myArr.length - myindex == 1)
            return false;

    boolean firstResult = pairsOfNumsRecursive(myArr,  myindex: myindex + 1,  sum: sum - myArr[myindex]);

    boolean secondResult = pairsOfNumsRecursive(myArr,  myindex: myindex + 1, sum);

    return firstResult | secondResult;
}
```

```
With Recursion
10 element array = 0.9ms
15 element array = 3.814ms
20 element array = 36.546ms
```